

Interface Homme-Machine

- Deux parties
 - Programmation (AWT/Swing)
 - Spécification (Javastates)
- La partie *Spécification* sera assurée par **Laurent Henocque** (en janvier) tandis que je m'occuperai de la partie *Programmation*

Bibliographie (bouquins)

- 2002 : “ Modélisation objet avec UML ”. Muller et Gaertner. Eyrolles.
- 2005 : “ UML 2 et les design patterns ”. Larman. Pearson.
- 2005 : “ Au cœur de Java 2 ”. Volumes 1 et 2. Hortsman & Cornell. CampusPress.
- 2005 : “ Swing : la synthèse ”. Berthié et Briaud. Dunod.
- 2006 : “ Le langage Java ”. Charon. Lavoisier.
- 2007 : “ Design Patterns ”. Gamma et all. Vuibert.
- 2007 : “ Génériques et collections Java ”. Naftalin & Wadler. O'Reilly
- 2008 : “ Effective Java ”. Bloch. Sun.
- 2009 : “ UML 2 par la pratique ”. Roques. Eyrolles.
- 2009 : “ UML 2 de l'apprentissage à la pratique ”. Audibert. ellipses.
- 2014 : “ Programmer en Java ”. Delannoy. Eyrolles.
- 2014 : “ Conception d'applications en Java ”. Lonchamp. Dunod.

Interface Homme Machine - Cours n°1

Introduction à la programmation graphique en Java

- Introduction
- Fenêtre
- Dessin, Panneau
- Formes 2D
- Couleurs, Fontes, Images
- Bouton
- Gestionnaires de mise en forme

Alain SAMUEL
Polytech Marseille / Informatique

OBJECTIFS

Fondements de la première partie de ce cours

(Programmation d'interfaces graphiques)

- **Interfaces graphiques :**
 - composants graphiques
 - capacités de dessins
 - fonctionnalités évolués

Ce cours s'appuiera sur *Java* et particulièrement son API *Swing*

- Trois cours :
 - **Graphisme de base**
 - **Programmation événementielle**
 - **Composants graphiques évolués**

Objectifs

- **Améliorer l'interaction avec les utilisateurs** en proposant des systèmes multi-fenêtrés avec des menus déroulants, des barres d'outils, des boîtes de dialogue, des tableaux, etc.
- Utiliser des **aspects graphiques** (dessin, couleur, image) pour enrichir les interactions
- L'utilisateur aura alors l'impression de **piloter le programme** qui réagira à ses demandes ...

Objectifs (suite)

- **Programmation événementielle** : le programme réagira à des événements provoqués par l'utilisateur (clics de souris, de bouton, d'élément de menu, etc.)
- Le cours est illustré par *Java/Swing* mais la plupart des concepts introduits sont communs à tous les environnements graphiques
- Le cours est pratique : c'est un **cours de programmation d'interfaces**

Java et la programmation graphique

- *Java* propose des **classes standard** de gestion des interfaces graphiques
- Il existe des outils graphiques de développement d'interface pour *Java* mais on ne les utilisera pas, afin de mieux comprendre comment cela marche ...

AWT et Swing

- *AWT* est la plus ancienne bibliothèque de classes graphiques *Java*, elle utilise des **composants** « **lourds** » (qui s'appuient sur les ressources du système d'exploitation hôte)
- *Swing* est apparue plus tard utilisant des **composants** « **légers** » beaucoup plus portables
- *Swing* ne remplace pas *AWT*, les **deux bibliothèques cohabitent**

FENETRE

Création d'une fenêtre

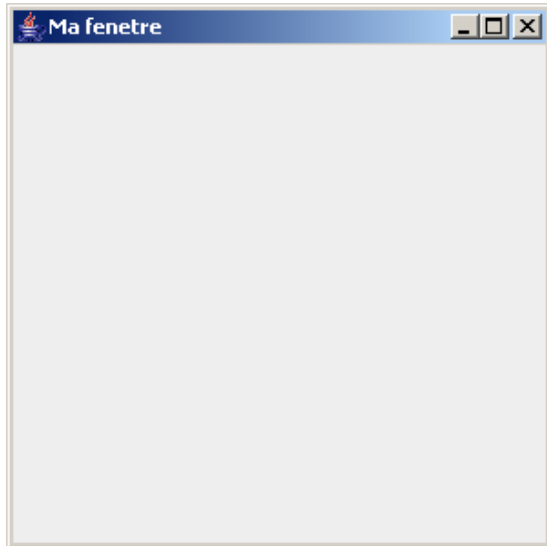
- De manière **explicite** par la classe *JFrame* (*javax.swing*) :

```
JFrame fenetre = new JFrame();  
fenetre.setTitle("Ma fenetre");           // texte dans la barre de titre  
fenetre.setBounds(150, 150, 300, 300);    // coordonnées + taille  
fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
           // arrêt du programme quand l'utilisateur ferme la fenêtre  
fenetre.setVisible(true);  
           // rend visible la fenêtre (invisible par défaut)
```

Commentaires

- La fenêtre obtenue est **vide** (avec une couleur d'arrière-plan par défaut)
- Néanmoins, l'utilisateur pourra **la redimensionner, la déplacer, la réduire à une icône et la fermer** (fonctionnalités prises en charge par la classe *JFrame* elle-même)

Exemple



Taille de l'écran

- Définir la position et la dimension d'une fenêtre **par rapport à la taille de l'écran** en pixels
- Utiliser la classe *Toolkit* qui **interface avec le système hôte** :

```
Toolkit tk = Toolkit.getDefaultToolkit();  
Dimension dim = tk.getScreenSize();  
int largeur = dim.width;  
int hauteur = dim.height;
```
- **Maximiser la taille** d'une fenêtre par :

```
fenetre.setExtendedState(Frame.MAXIMIZED_BOTH);
```

Création d'une classe fenêtre personnalisée

- **Sous-classe de *JFrame***

- Exemple :

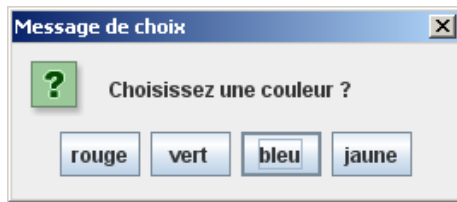
```
class MaFenetre extends JFrame {  
    private JButton j;          // attribut spécifique  
    public MaFenetre(String titre, int x, int y, int l, int h) {  
        setTitle(titre);  
        setBounds(x, y, l, h); }  
    ... }
```

```
MaFenetre fenetre = new MaFenetre("Ma fenetre", 150, 150, 300, 300);
```

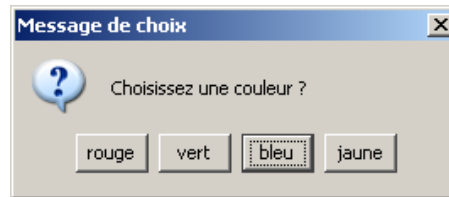
Look and feel (apparence)

- Choisir l'**apparence des composants graphiques** (le rendu visuel de chaque composant)
- Le *look and feel* par défaut de Java est *metal* (thème *ocean*),
- NB : pour chaque composant Swing, il existe un objet *look and feel* associé ... ce qui rend ce dispositif assez coûteux ...

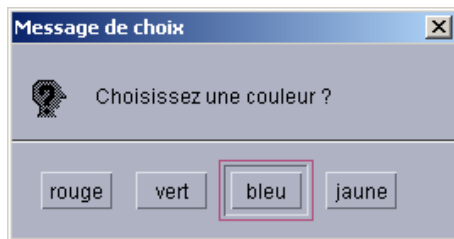
Divers styles de *look and feel*



Metal



Windows



Motif

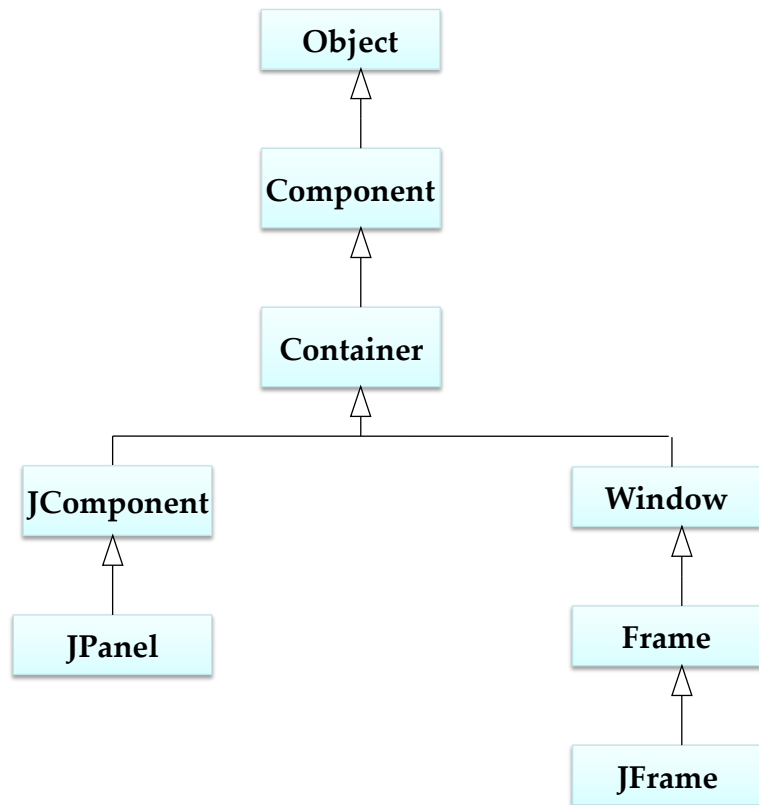
Gestion des *look and feel*

- La classe *UIManager* gère les différents *look and feel*
- La méthode statique *getInstalledLookAndFeels()* renvoie les **différents styles disponibles** dans l'environnement utilisé
- La méthode statique *setLookAndFeel(String ou LookAndFeel)* **change le look and feel courant**
- La méthode statique *updateComponentTreeUI(Component)* de la classe *SwingUtilities* **change le look and feel d'un composant déjà affiché** (et celui de ses sous-composants)

Hiérarchie de quelques classes

Mélange de *AWT* et *Swing* :
complexe et peu cohérent

La plupart des classes *Swing*
commence par la lettre *J*



Quelques méthodes utiles de ces classes

Component

| | |
|---|--------------------------------------|
| <code>void setSize(int l, int h)</code> | // modifie la taille du composant |
| <code>Dimension getSize()</code> | // taille actuelle du composant |
| <code>void setLocation(int x, int y)</code> | // déplace le composant |
| <code>Point getLocation()</code> | // position du coin supérieur gauche |

Window

| | |
|-----------------------------|---|
| <code>void toFront()</code> | // affiche cette fenêtre par-dessus toutes les autres |
|-----------------------------|---|

Quelques méthodes utiles de ces classes (suite)

Frame

```
void setResizable(boolean b)  
// pour modifier la taille de la fenêtre par l'utilisateur (vrai par défaut)  
  
void setIconImage(Image i)  
// i servira d'icône à cette fenêtre lorsqu'elle sera réduite
```

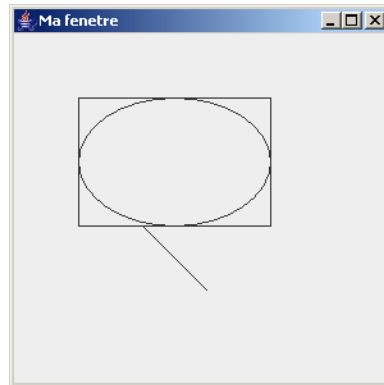
Toolkit

```
Image getImage(String f)  
// charge une image à partir du fichier spécifié par f (nom complet)
```

DESSIN, PANNEAU

Notion de « panneau »

- *Swing* permet de **dessiner** (des formes, des images, du texte, etc.) sur un composant, en général **un « panneau »** (objet *JPanel* sous-classe de *JComponent*)



- Contrairement aux apparences le dessin ci-dessus n'est pas effectué dans une *JFrame* mais dans un *JPanel* ...

Précisions

- *JPanel* est une **sorte de rectangle sans titre ni bordure** qui ne peut pas exister de manière autonome (il se confond souvent avec le composant dans lequel il se trouve)
- *JPanel* est **le plus pratique des « conteneurs »** (il peut contenir d'autres composants et possède une surface sur laquelle on peut dessiner) et doit être inclus lui-même dans un conteneur (généralement une fenêtre)

Notion de « permanence » du dessin

- Attention à la nécessité de **permanence** du dessin qui ne doit pas être altéré par une modification du composant comme :
 - modification de sa taille
 - transformation en icône
 - masquage par un autre composant
 - etc.
- Et qui bien sûr **doit être initialisé, et mis à jour si nécessaire ...**

La méthode *paintComponent* de *JComponent*

- Afin d'assurer la permanence du dessin, placer les instructions de dessin dans une redéfinition de la méthode *paintComponent* : en effet celle-ci sera automatiquement appelée par *Swing* chaque fois que le composant aura besoin d'être (re)dessiné
- Attention, les fenêtres de premier niveau (par exemple *JFrame*) ne sont pas des sous-classes de *JComponent*, donc ne pas dessiner directement dans ces fenêtres ...

Remarque (à ne pas faire !)

- Dessiner ailleurs que dans des *JComponent* en redéfinissant la méthode *paint* de *Component*
- **Cette façon de procéder n'est pas conseillée** car avec AWT (ou avec quelques rares composants *Swing* comme *JFrame*) l'utilisation des fenêtres natives des diverses plates-formes peut compliquer les opérations de dessin (**en clair problèmes possibles !**)
- De plus, en redéfinissant *paint* au lieu de *paintComponent*, il faudra également s'occuper de redessiner les sous-composants

Utilisation de *paintComponent*

- On redéfinira donc *paintComponent* dans une classe dérivée de *JPanel* afin d'y écrire le code du dessin
- L'en tête de *paintComponent* est le suivant :
void paintComponent(Graphics g) où **g est le contexte graphique qui interface avec le système hôte** et dispose de paramètres et de méthodes permettant de dessiner sur le composant associé

La classe *Graphics*

Les paramètres de ce contexte graphique portent essentiellement sur :

- Le composant sur lequel dessiner
- La couleur pour effectuer les tracés (celle du premier plan du composant concerné)
- La fonte (celle du composant concerné)
- Le rectangle de « clipping » qui correspond à la partie à repeindre du composant concerné (par exemple si une partie du composant a été momentanément recouverte par une fenêtre, ce rectangle décrit la partie détériorée)

Exemple

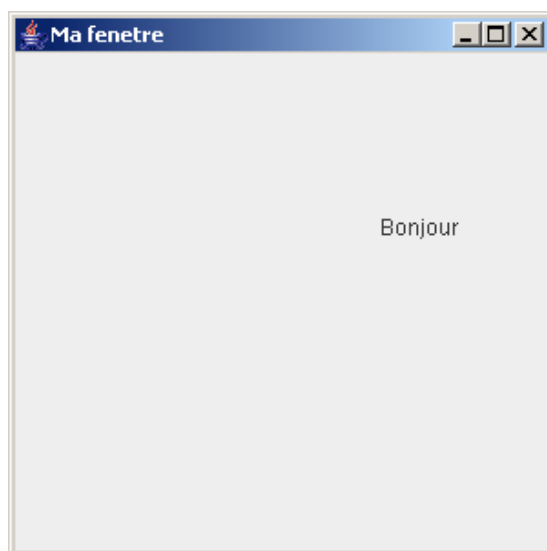
```
class BonjourPanneau extends JPanel {  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        g.drawString("Bonjour", 200, 100);  
        // dessine la chaîne de caractères donnée à la position relative donnée  
        // avec la fonte et la couleur courante    } }
```

NB : Ne pas oublier d'appeler *super.paintComponent(g)* pour qu'elle fasse sa part de travail (ici le dessin de la couleur de fond)

Programme complet

```
class MaFenetre extends JFrame {  
    private BonjourPanneau panneau;  
    public MaFenetre(String titre, int x, int y, int l, int h) {  
        setTitle(titre);  
        setBounds(x, y, l, h);  
        panneau = new BonjourPanneau();  
        add(panneau); } }      // ajoute le panneau dans la fenêtre  
  
public class Premier {  
    public static void main(String args[]) {  
        MaFenetre f = new MaFenetre("Ma fenetre", 150, 150, 300, 300);  
        fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        fenetre.setVisible(true); } }
```

Exemple



Forcer l'opération de dessin

- Il peut être nécessaire de forcer un panneau à se redessiner (sans attendre l'appel automatique de *paintComponent*, par exemple dans le cas où on a modifié volontairement le dessin), **pour cela il faudra appeler la méthode *repaint* de *Component*** (et non la méthode *paintComponent* elle-même !)
- NB : Il peut être plus **efficace** d'utiliser la version de *repaint* avec arguments (qui définit le rectangle qui doit être mis à jour) que celle sans arguments (qui remet à jour le composant entier)

Dessiner à la volée ! (sans passer par *paintComponent*)

- Obtenir **directement** un contexte graphique pour un *JComponent* :

```
Graphics g = panneau.getGraphics();  
g.drawString("Bonjour", 200, 100);  
g.dispose(); // pour libérer le contexte graphique (donc la mémoire)
```
- Cela peut servir à dessiner rapidement un prototype (sans gérer la permanence du dessin par *paintComponent*, **mais cette méthode n'est pas conseillée ...**)

Autres méthodes de dessin de la classe *Graphics* (mais généralement on utilisera plutôt *Graphics2D*)

```
drawLine(int x1, int y1, int x2, int y2)    // entre (x1,y1) et (x2,y2)
drawRect(int x, int y, int l, int h) // à partir de (x,y), largeur l et hauteur h
drawOval(int x, int y, int l, int h)    // ellipse inscrite dans le rectangle
drawRoundRect(int x, int y, int l, int h, int al, int ah)
    // rectangle à coins arrondis, al et ah diamètres de l'arc des 4 coins
drawPolygon(int[] x, int[] y, int n)
    // polygone défini par n points donnés par (x[i], y[i])
drawPolyLine(int[] x, int[] y, int n)    // même chose mais ne joint pas
    // le dernier point au premier (lignes brisées)
drawArc(int x, int y, int l, int h, int a1, int a2)
    // arc d'ellipse, a1 et a2 angle définissant l'arc
```

Formes 2D

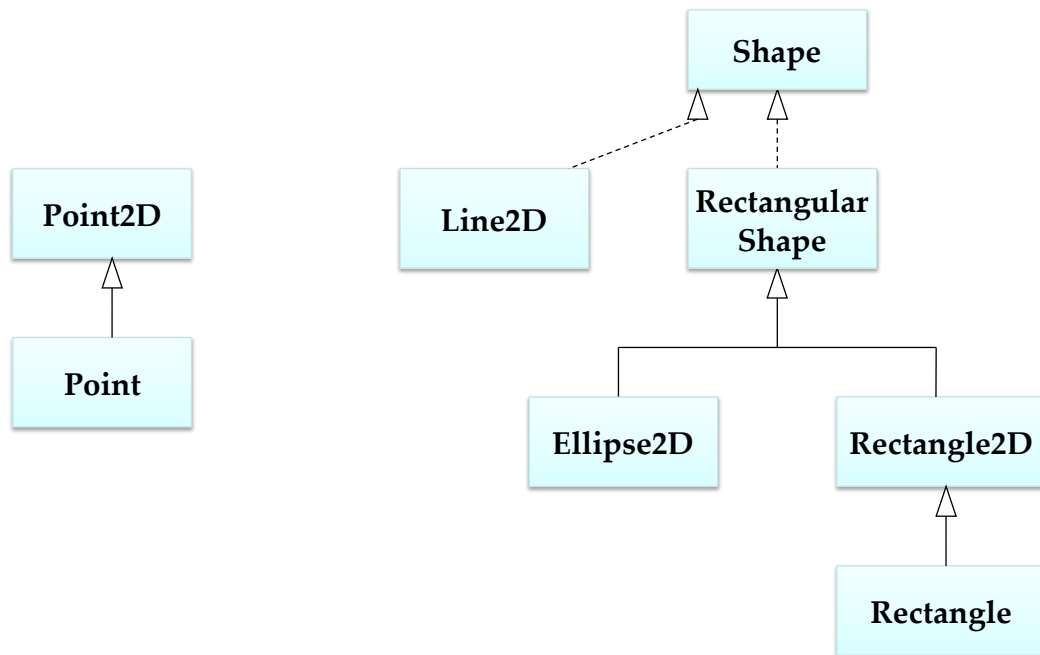
Formes 2D

- Les opérations de dessin précédentes sont limitées (traits d'épaisseurs fixes, formes trop élémentaires, formes ne pivotant pas, pas d'outils de manipulation, etc.)
- **La bibliothèque *Java 2D* implémente un jeu d'opérations graphiques puissant**
- Pour l'utiliser, il faut passer par un objet *Graphics2D*, sous-classe de *Graphics*

Utilisation

- La méthode *paintComponent* reçoit automatiquement un objet *Graphics2D*, il suffira de transtyper
- Pour dessiner une forme, il faudra créer un objet d'une classe qui implémente l'interface *Shape* et **appeler la méthode *draw(Shape s)* sur un objet *Graphics2D***

Hierarchie des classes Formes 2D (non exhaustif)



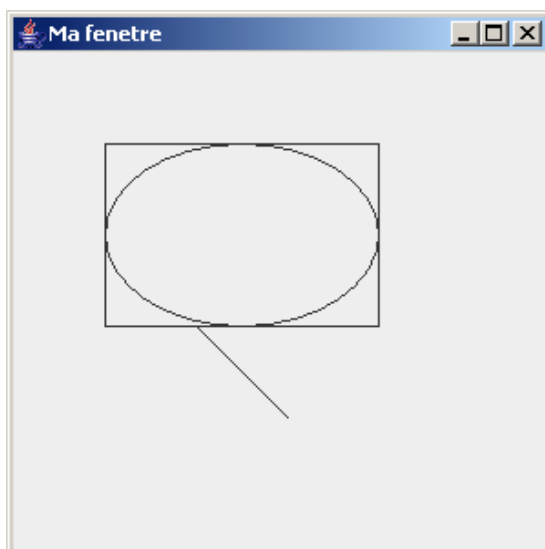
Remarques

- **Java 2D est précis**, il emploie des valeurs de coordonnées *float* (économe en mémoire et rapide en calcul) ou *double* (standard il évite le transtypage), et non *int* comme les méthodes de *Graphics*
- **Java 2D offre deux versions de chaque classe de forme**, par exemple *Rectangle2D.Float* et *Rectangle2D.Double*, sous-classes et classes internes statiques de la classe abstraite *Rectangle2D* (cela pour éviter de renommer complètement *Rectangle2D*)

Exemple

```
public void paintComponent(Graphics g) {  
    super.paintComponent(g);  
    Graphics2D g2 = (Graphics2D) g;  
    Rectangle2D rect = new Rectangle2D.Double(50, 50, 150, 100);  
    g2.draw(rect);  
    Ellipse2D ellipse = new Ellipse2D.Double();  
    ellipse setFrame(rect); // setFrame construit l'ellipse inscrite dans rect  
    g2.draw(ellipse);  
    Point2D org = new Point2D.Double(100, 150);  
    Point2D fin = new Point2D.Double(150, 200);  
    Line2D ligne = new Line2D.Double(org, fin);  
    g2.draw(ligne); }
```

Exemple



Comment efface t-on un dessin ?

- **Tout redessiner plutôt que d’effacer !**
- Néanmoins, la méthode *clearRect* de la classe *Graphics* permet de spécifier un rectangle à l’intérieur duquel sera dessinée une couleur de fond
- Il est donc conseillé d’enchaîner les instructions (*g* étant un *Graphics2D* et *c* le composant associé) :

```
g.clearRect(i, j, l, h); g.setPaint(c.getBackground()); g.fillRect(i, j, l, h);
```

Transformations géométriques

- **La classe *AffineTransform* permet de faire des transformations géométriques sur les formes** (translation, rotation, homothétie, etc.)
- Quelques méthodes de la classe *AffineTransform* :

```
Shape createTransformedShape(Shape s) // renvoie la transformée de s
```

```
void rotate(double t, double x, double y) // compose cette transforma-  
-tion avec une rotation d’un angle t (en radians) autour de (x,y)
```

```
void setToTranslation(double x, double y)
```

```
// change cette transformation en translation de (x, y)
```

Transformations de *Graphics2D*

- Il est également possible de composer la transformation *Graphics2D* courante avec une translation (méthode *translate* de *Graphics2D*), une rotation (méthode *rotate* de *Graphics2D*), etc.
- Dès lors, **toutes les formes à dessiner se verront appliquer au préalable cette nouvelle transformation**
- Enfin, il existe également une méthode *transform(AffineTransform)* de *Graphics2D* qui permet de composer la transformation *Graphics2D* courante avec la transformation affine donnée en paramètre

Questions ?

COULEURS, FONTES, IMAGES

Style de trait

- **Il est possible de définir son propre style de trait en implémentant l'interface *Stroke*, par exemple la classe *BasicStroke* dispose de plusieurs constructeurs précisant :**
 - l'épaisseur
 - le style de la terminaison (brut, carré, arrondi)
 - le style de jonction (prolongé, arrondi, droit)
 - le style des éventuels pointillés
- **Le style de trait est transmis au contexte graphique par la méthode *setStroke* de *Graphics2D***

Couleurs

- En *Java*, les couleurs sont définies à l'aide de la **classe** *Color* qui propose 13 objets constants (*Color.RED*, *Color.BLUE*, etc.)
- Une **couleur personnalisée** est définie par un objet *Color* avec des proportions de rouge, de vert et de bleu sur une échelle de 0 à 255 :
Color bleuVert = new *Color*(0, 128, 128);
Color rougeVif = new *Color*(255, 0, 0);
- Les méthodes *brighter()* et *darker()* de la classe *Color* permettent d'obtenir des couleurs **plus brillantes ou plus sombres**

Modification de couleur

- La méthode *setPaint(Paint)* de *Graphics2D* définit les paramètres de peinture de ce contexte graphique
- *Color* implémentant l'interface *Paint*, on peut donc **modifier la couleur courante par *setPaint*** (il faut utiliser *setColor(Color)* sur un objet *Graphics*)

Couleur d'arrière-plan et d'avant-plan

- Pour spécifier la **couleur d'arrière-plan** (ou de fond), on utilise *setBackground(Color)* de la classe *Component*
- Pour spécifier la **couleur d'avant-plan** (pour les dessins) d'un composant, on utilise *setForeground(Color)*; cette couleur d'avant-plan peut être modifiée pour un contexte graphique dans *paintComponent* par *setPaint* (ou *setColor*) mais sera restituée à la sortie de *paintComponent*

Remplissage de formes fermées

- On peut **remplir des formes fermées** avec la couleur courante (ou plus généralement avec le motif courant); pour cela il suffit d'**appeler *fill* au lieu de *draw*** :

```
Rectangle2D rect = new Rectangle2D.Double(50, 100, 300, 200);  
g2.setPaint(Color.BLUE);  
g2.fill(rect);
```

NB : si on utilise la classe *Graphics*, on utilisera *fillXXX* au lieu de *drawXXX* pour remplir les formes fermées

- NB : La classe *SystemColor* donne accès aux **couleurs employés pour les divers éléments de l'environnement**, par exemple :
`panneau.setBackground(SystemColor.window);`

Effets de couleur

- **Effet de transparence** (le quatrième argument définit le niveau de transparence)

```
g2.setPaint(new Color(255, 0,0,155)); // définit un rouge transparent
```

- **Effet de dégradé** pour remplir une forme, il faut passer par la classe *GradientPaint* et un des ses constructeurs qui précisera le point et la couleur de départ ainsi que le point et la couleur d'arrivée

```
GradientPaint gp = new GradientPaint(P1, C1, P2, C2);  
g2.setPaint(gp);
```

Fontes

- **Il est possible de changer de fonte de caractères en créant un objet de la classe *Font* qu'on transmet à la méthode *setFont* (de *Graphics* ou de *Component*) :**

```
Font f = new Font("Serif", Font.BOLD, 20); // famille, style, taille  
g.setFont(f);
```

- *Serif* est une fonte logique proposée par *Java* qui assure automatiquement la correspondance avec une fonte installée dans l'environnement

Fontes physiques

- Les fontes physiques sont également accessibles :

```
Font f = new Font("Helvetica", Font.ITALIC, 20);  
g.setFont(f);
```

- Pour connaître les fontes disponibles dans un environnement donné

```
GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();  
// fournit l'environnement graphique du système utilisateur
```

```
String[] fontes = ge.getAvailableFontFamilyNames();  
// fournit un tableau de String des fontes
```

Exemple

```
class MaFenetre extends JFrame {  
    private BonjourPanneau panneau;  
    public MaFenetre() {  
        setTitle("Ma fenetre"); setBounds(150, 150, 300, 300);  
        panneau = new BonjourPanneau();  
        panneau.setBackground(Color.GREEN);  
        panneau.setForeground(Color.RED);  
        add(panneau); } }  
  
class BonjourPanneau extends JPanel {  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g); g.drawString("Bonjour", 200, 100);  
        Font f = new Font("Serif", Font.BOLD, 36); g.setFont(f);  
        g.setColor(Color.BLUE);  
        g.drawString("Bonjour", 100, 200); } }
```

Exemple



Position du texte

- Quelle est précisément la position du texte dans le cas suivant :
`g.drawString("Bonjour", 200, 100);`
quelle est la longueur de "Bonjour" ?
quelle est la signification précise de 100 ?
- La méthode `getFontMetrics` de *Graphics* renvoie un objet *FontMetrics* qui contient les caractéristiques de la fonte courante
`FontMetrics fm = g.getFontMetrics();`
`int longueur = fm.stringWidth("Bonjour");`
// longueur en pixels de "Bonjour" dans la fonte courante
d'où la réponse à la première question, **cela permettra d'afficher un second texte juste à la suite du premier (200 + longueur)**

Position du texte (suite)

- De plus :

```
int hauteur = fm.getHeight();
```

```
// hauteur en pixels d'une ligne de texte dans la fonte courante
```

d'où la réponse à la seconde question, **cela permettra d'afficher à la position idéale un second texte en dessous de "Bonjour"** (à l'ordonnée **100 + hauteur**)
- Plus précisément, **la hauteur est égale à la distance verticale entre deux lignes de base successives** et équivaut à la somme entre le jambage descendant, l'interligne et le jambage ascendant (valeurs accessibles par *getDescent*, *getLeading* et *getAscent* de *FontMetrics*)

Images

Il est très simple d'**afficher des images** prises dans des fichiers en utilisant la méthode statique *read(File)* de la classe *ImageIO* :

```
String fichier = " ... ";           // nom complet
```

```
BufferedImage im = ImageIO.read(new File(fichier));
```

puis en utilisant la méthode *drawImage* de la classe *Graphics* :

```
g.drawImage(im, x, y, null); // position (x,y), pas de mise à l'échelle
```

```
ou g.drawImage(im, x, y, l, h, null);
```

```
// position (x,y), mise à l'échelle dans un rectangle l x h
```

NB : le dernier paramètre (de type *ImageObserver*) peut servir à récupérer des informations sur la construction de l'image

Texture

- La classe *TexturePaint* permet de **remplir une forme avec une texture créée en répétant une même image**
- Il faut passer par son constructeur :
`TexturePaint(BufferedImage i, Rectangle2D r)`
où *r* spécifie un rectangle de référence contenant l'image
- Il suffit alors de transmettre l'objet *TexturePaint* créé, en argument de la méthode *setPaint* de *Graphics2D* (car *TexturePaint*, comme *Color* et *GradientPaint*, implémente l'interface *Paint*) avant de dessiner

BOUTON

Création d'un bouton

- Un bouton est un composant élémentaire dont l'usage est très intuitif : c'est un objet de la classe *JButton* qu'on crée en donnant le texte qu'on souhaite voir figurer à l'intérieur

```
JButton bouton = new JButton("Essai");
```

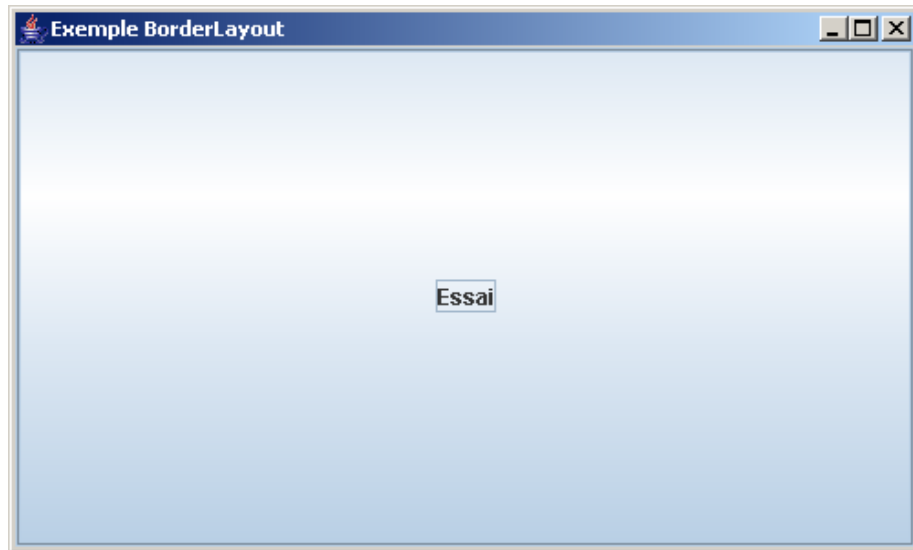
- Pour introduire ce composant dans une fenêtre, il faut utiliser la méthode *add* de la classe *Container* :

```
fenetre.add(bouton); // ajoute le composant donné à un JFrame
```

Affichage d'un bouton

- Contrairement à une fenêtre, un bouton est visible par défaut mais lors de l'affichage de *fenetre*, on se rend compte que **le bouton occupe tout l'espace disponible de la fenêtre**
- La disposition des composants dans un conteneur est gérée par un **gestionnaire de mise en forme**, il en existe plusieurs en *Java* qui ont chacun un comportement spécifique

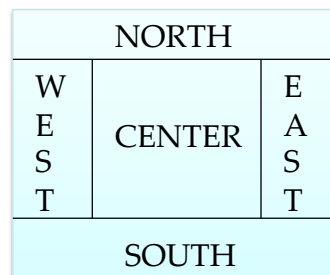
Exemple



***GESTIONNAIRES DE MISE
EN FORME***

Le gestionnaire *BorderLayout*

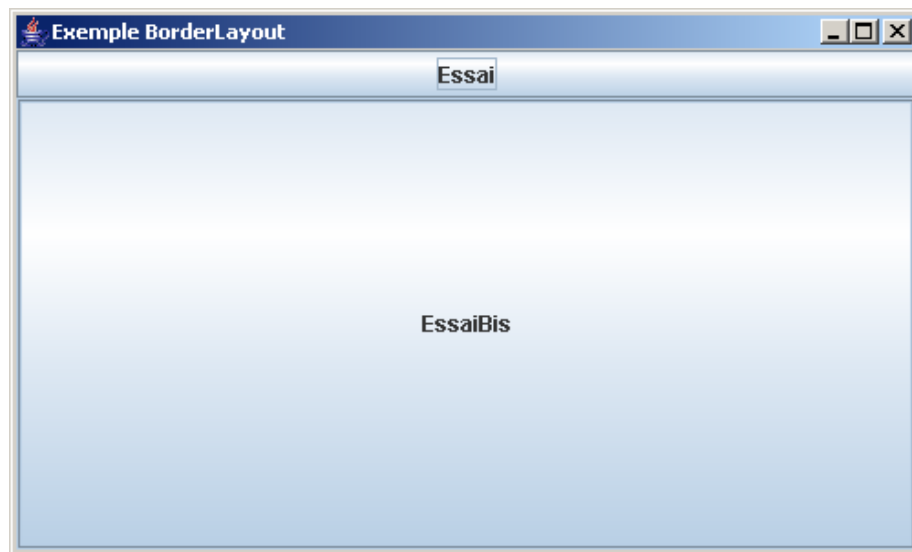
- Il s'agit du gestionnaire par défaut d'un *JFrame*, mais en l'absence d'informations spécifiques, un composant occupera tout l'espace disponible (quitte à masquer un composant introduit auparavant !)
- En réalité, *BorderLayout* est capable de disposer les composants **sur les quatre bords du conteneur ou au centre** :



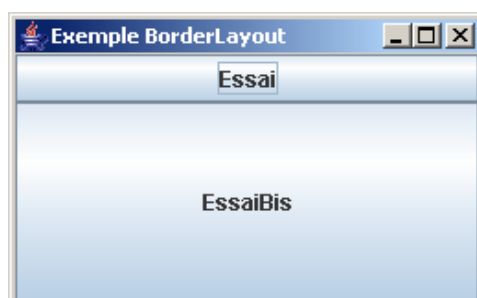
Positionnement

- Les composants placés en bordure sont positionnés en premier et le composant placé au centre occupera l'espace disponible restant
- L'emplacement d'un composant est donné en argument de la méthode *add* par une constante CENTER, NORTH, SOUTH, EAST ou WEST de la classe *BorderLayout* (CENTER par défaut) :
`fenetre.add(bouton, BorderLayout.NORTH);`
- Lorsque le conteneur est redimensionné, **l'épaisseur des composants en bordure ne change pas** mais la dimension du composant central est modifiée

Exemple



Redimensionnements



Précisions

- Par défaut les composants sont espacés de 5 pixels mais **il est possible de définir les espaces horizontaux et verticaux avec des intervalles différents**, soit au moment de la construction du gestionnaire, soit en utilisant les méthodes *setHgap* et *setVgap* de la classe *BorderLayout*

```
fenetre.setLayout(new BorderLayout(15, 30));
```

OU

```
BorderLayout bl = new BorderLayout(); ...
```

```
bl.setHgap(15); bl.setVgap(30);
```

```
fenetre.setLayout(bl);
```

NB : *setLayout(LayoutManager)* étant une méthode de *Container* qui modifie le gestionnaire de mise en forme

Le gestionnaire *FlowLayout*

- Il s'agit du gestionnaire par défaut d'un *JPanel* qui dispose les différents composants "en flot" (les uns à la suite des autres, sur une même ligne, puis ligne après ligne)
- Contrairement à *BorderLayout*, la taille des composants est respectée; celle-ci est définie a priori suivant la nature et le contenu du composant (par exemple le texte avec sa fonte influe sur la taille d'un bouton).
- Lorsque le conteneur est redimensionné, la disposition des composants évolue, toujours selon la logique "en flot"

Positionnement

- **Par défaut, sur chaque ligne, les composants sont centrés horizontalement;** il est possible de choisir un alignement à gauche ou à droite en spécifiant les constantes LEFT ou RIGHT de la classe *FlowLayout* dans le constructeur du gestionnaire :

```
fenetre.setLayout(new FlowLayout(FlowLayout.RIGHT));
```

- Comme avec *BorderLayout* on peut modifier les espacements horizontaux et verticaux des composants (5 pixels par défaut) :

```
fenetre.setLayout(new FlowLayout(FlowLayout.RIGHT, 10, 15));
```

OU

```
FlowLayout fl = new FlowLayout(); ...
```

```
fl.setHgap(10); fl.setVgap(15);
```

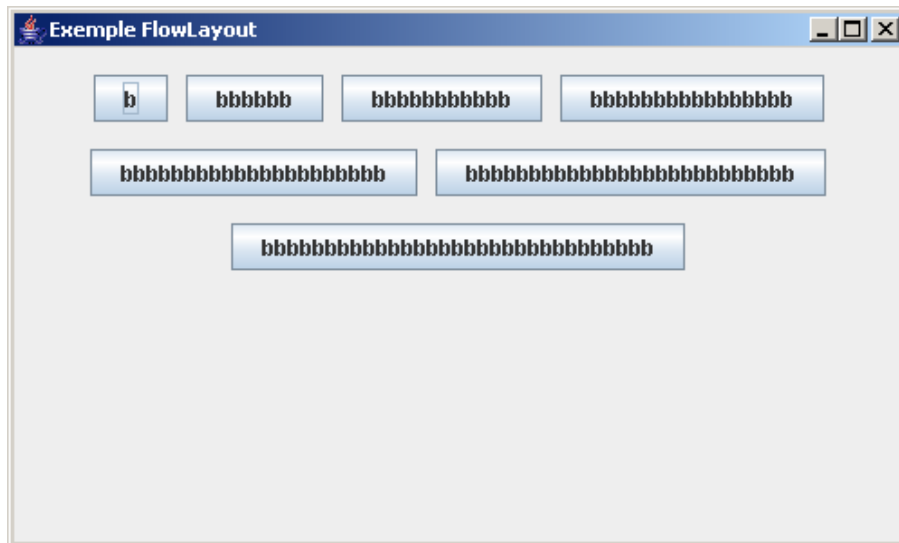
```
fenetre.setLayout(fl);
```

Exemple (disposition de 7 boutons de tailles différentes avec *FlowLayout*)

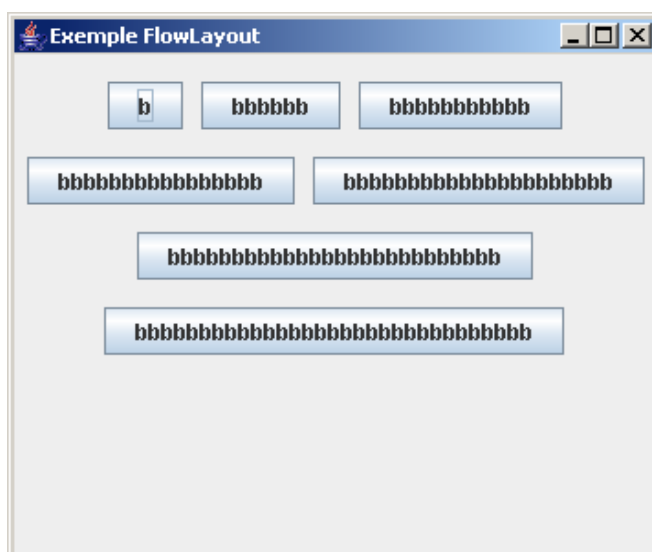
```
class MaFenetre extends JFrame {
    private JButton boutons[];
    public MaFenetre() {
        setTitle("Exemple FlowLayout"); setBounds(10, 150, 500, 300);
        setLayout(new FlowLayout(FlowLayout.CENTER, 10, 15));
        boutons = new JButton[7]; String ch = "b"; String pl = "bbbbbb";
        for (int i=0; i<7; i++)
            {boutons[i] = new JButton(ch); ch += pl; add(boutons[i]); } }

    public class Exemple {
        public static void main(String args[]) {
            MaFenetre fenetre = new MaFenetre();
            fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            fenetre.setVisible(true); }}
}
```

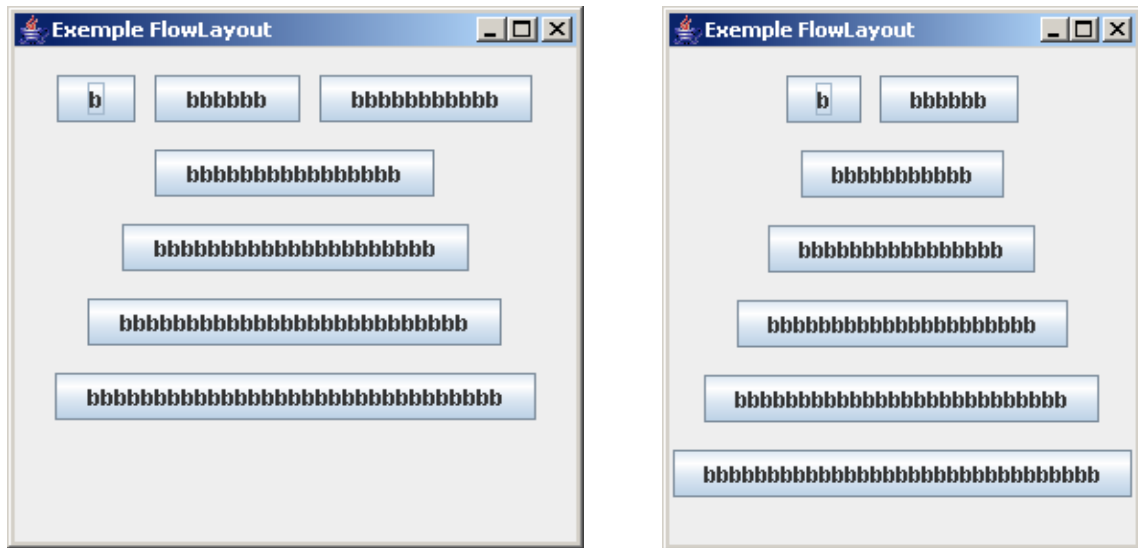
Exemple



Redimensionnement



Autres redimensionnements



Combinaison de *BorderLayout* et *FlowLayout*

- Moyen d'obtention d'un **positionnement relativement précis** des différents composants
- **Utilisation de panneaux supplémentaires** qui disposeront de leur propre gestionnaire de mise en forme (par défaut *FlowLayout*)
- Par exemple, pour disposer des boutons en bas de la fenêtre
`panneau.add(monB1); panneau.add(monB2); panneau.add(monB3);`
`fenetre.add(panneau, BorderLayout.SOUTH);`
- On verra plus tard qu'il existe d'autres gestionnaires plus précis ...

Gestion des dimensions d'un composant

- Utilisation de la méthode *getSize* de *Component* pour **récupérer la taille d'un composant**, ainsi pour connaître la taille des boutons précédents :

```
for (int i = 0; i < 7; i++)  
    {Dimension dim = fenetre.boutons[i].getSize();  
    System.out.println(dim.width + " " + dim.height);}
```

- De plus, on peut toujours **imposer une taille donnée à un composant** grâce à la méthode *setPreferredSize* :

```
bouton.setPreferredSize(new Dimension(50, 50));
```

mais on verra que cette information n'est pas toujours bien traitée par les gestionnaires ...

Modification de la taille d'un composant

- Le gestionnaire *BorderLayout* ne tient pas compte de l'information donnée sur la taille du composant tandis que *FlowLayout* en tient compte dans la mesure du possible ...
- De plus, si on modifie la taille du composant par *setPreferredSize* **après l'affichage du conteneur** auquel il est rattaché, il faudra obliger le gestionnaire à refaire ses calculs en invoquant la méthode *revalidate* pour le composant :

```
bouton.setPreferredSize(new Dimension(largeur, hauteur));  
bouton.revalidate();
```


Questions ?

Interface Homme Machine – Cours n°2

Gestion des événements – Composants de texte

- Méthodologie générale
- Catégorie d'événements
- Événements de bas niveau
- Actions
- Composants de texte

Alain SAMUEL
Polytech Marseille / Informatique

GESTION DES EVENEMENTS

Programmation événementielle

- Applications standards : **l'application garde le contrôle des enchaînements d'instructions**, en particulier elle sollicite (quand elle le souhaite) l'utilisateur pour obtenir des informations nécessaires au traitement
- Interfaces graphiques : **l'utilisateur prend (en partie) le contrôle des enchaînements d'instructions**, et peut décider (quand il le souhaite) du déclenchement de tel ou tel traitement, en fonction de tel ou tel événement d'interfaces

Comment cela fonctionne ?

- Une boucle infinie est mise en place pour récupérer les événements, les mettre dans une file d'attente (par l'intermédiaire d'un *thread*), puis les traiter et réafficher l'interface graphique (par un autre *thread*)

NB : une application Swing génère automatiquement trois *threads* : le *main*, la boucle infinie et l'EDT (*event dispatching thread*) qui traite les événements et les affichages

Problème

- Tant que le traitement des événements de la file d'attente n'est pas terminé, l'interface n'est pas mise à jour ...
- Dans le cas d'un traitement long (accès à une base de données, calculs ou dessins complexes, ...) cela peut évidemment perturber les interactions avec l'utilisateur ...
- Dans de telles situations (à éviter autant que possible), il sera nécessaire d'effectuer les traitements longs dans des *threads* à part (voir la classe *SwingUtilities*)

Objectif

- La **gestion des événements** est le processus de base permettant de programmer des **IHM réagissant aux interactions de l'utilisateur** (appui d'une touche au clavier, clic sur un bouton, sur un élément de menu, etc.) qui se traduisent par des événements en *Java*
- L'**objectif** consiste à associer du code à l'émission d'un événement

Méthodologie : source, écouteur, abonnement, notification

- Tout événement possède une **source** : c'est l'objet qui lui a donné naissance (fenêtre, bouton, panneau, etc.)
- Pour intercepter et traiter un événement, on utilise un **écouteur** : c'est l'objet qui se tient prêt à intervenir en cas de besoin
- N'importe quel objet peut être écouteur d'événement mais il doit auparavant **s'abonner** auprès de la source
- Lorsqu'un événement se produit, la source envoie une **notification** à tous ses écouteurs

Catégories d'événements

- Il existe des **classes d'événements spécifiques** pour chaque type d'interaction avec l'utilisateur
- Par exemple, *ActionEvent* correspond (entre autres) à un clic sur un bouton, *KeyEvent* à la frappe d'une touche au clavier, *MouseEvent* à un clic de la souris, *WindowEvent* à un changement d'état de la fenêtre, etc.

Jonction avec l'écouteur

- **Pour abonner un écouteur auprès de la source :**
`objetSource.addXXXListener(objetEcouteur);`
où XXX dépend de la catégorie d'événements (*XXXEvent*)
- **Un écouteur est une instance d'une classe qui implémente une interface spéciale nommée interface écouteur** (*listener interface*)
- Pour la catégorie d'événements *XXXEvent*, cette interface écouteur se nomme *XXXListener* (il peut y en avoir d'autres) qu'il faut donc implémenter :
`class MaClasseXL implements XXXListener{ ...}`

Jonction avec le code

- L'interface écouteur fournit un **ensemble de méthodes** qu'il faut (en partie) implémenter : chacune de ces méthodes correspond à un événement de la catégorie concernée et **le corps de la méthode correspond au traitement de l'événement**
- Finalement, **l'émission d'un événement se traduit par un simple appel de méthode qui le traite** : plus précisément quand un événement se produit, la notification envoyée par la source à son écouteur déclenchera l'appel de la méthode associée ...

Exemple d'un clic sur un bouton qui met en rouge la couleur de fond du panneau

```
class MaFenetre extends JFrame {  
    private MonPanneau panneau;  
    public MaFenetre(String titre, int x, int y, int l, int h) {  
        setTitle(titre); setBounds(x, y, l, h);  
        panneau = new MonPanneau(); add(panneau);}}  
  
class MonPanneau extends JPanel {  
    private JButton bouton;  
    public MonPanneau() {  
        bouton = new JButton("Rouge"); add(bouton);  
        RougeAction ra = new RougeAction(this);           // écouteur  
        bouton.addActionListener(ra);                     // abonnement }}
```

Implémentation de *ActionListener*

- class RougeAction implements ActionListener {
 private MonPanneau p;
 public RougeAction(MonPanneau x) { p = x; }
 public void actionPerformed(ActionEvent e) {
 p.setBackground(Color.RED); }}

• *actionPerformed* est l'unique méthode de *ActionListener* (i.e unique événement de la catégorie *ActionEvent*)

• NB : pour modifier la couleur de fond du panneau, on transmet celui-ci en paramètre du constructeur de l'écouteur

Simplification du code par utilisation d'une classe interne

- Pour éviter de passer des paramètres au constructeur de l'écouteur, on peut utiliser une classe interne :

```
class MonPanneau extends JPanel {  
    private JButton bouton;  
    public MonPanneau() {  
        bouton = new JButton("Rouge"); add(bouton);  
        RougeAction ra = new RougeAction(); // pas de paramètres  
        bouton.addActionListener(ra); }  
    private class RougeAction implements ActionListener {  
        public void actionPerformed(ActionEvent e) {  
            setBackground(Color.RED); }}}}
```

Simplification du code par utilisation d'un objet anonyme

- Pour éviter de créer une classe écouteur et la nommer, on peut utiliser un objet anonyme :

```
class MonPanneau extends JPanel {  
    private JButton bouton;  
    public MonPanneau() {  
        bouton = new JButton("Rouge"); add(bouton);  
        bouton.addActionListener(  
            new ActionListener() {          // objet anonyme  
                public void actionPerformed(ActionEvent e) {  
                    setBackground(Color.RED); } } );    }  
}
```

Informations sur les événements

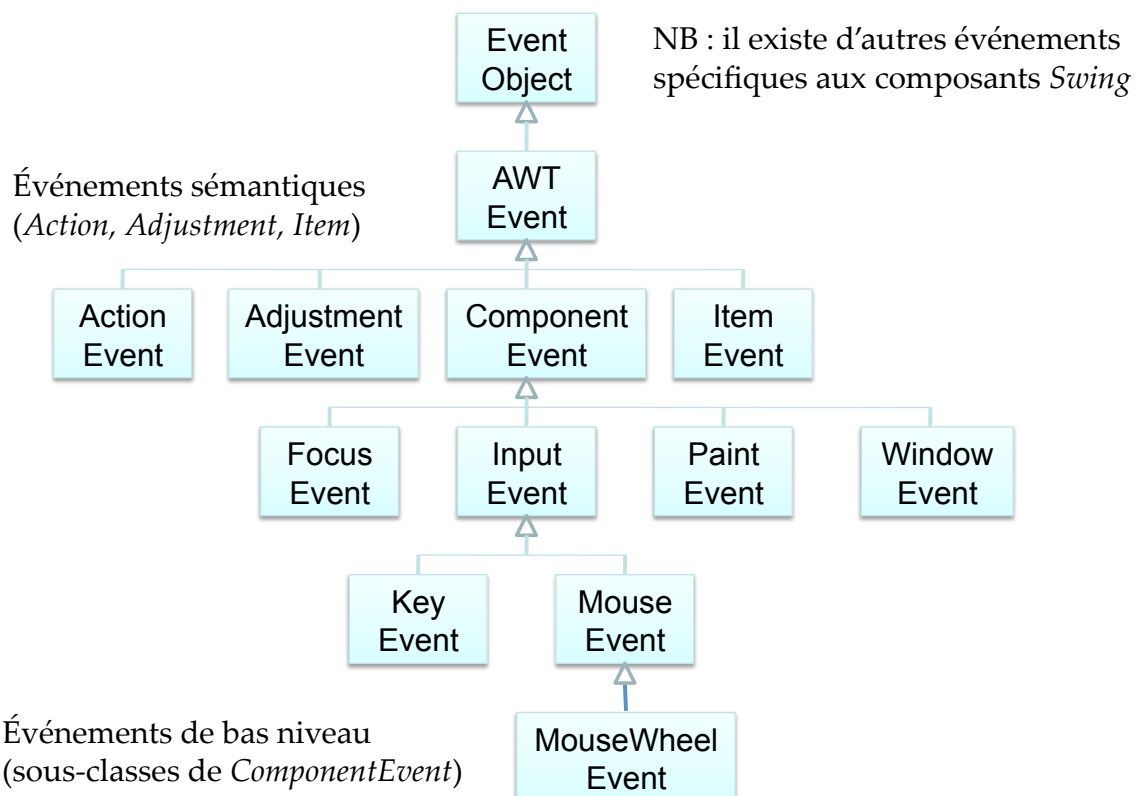
- Quand la source envoie une notification d'événement à ses écouteurs, elle transmet un **objet événement contenant des informations sur son contexte d'apparition**
- Cet objet événement est transmis à chaque méthode de traitement de l'événement sous forme de **paramètre de type *XXXEvent***
- Exemple :

```
public void actionPerformed(ActionEvent e) {  
    setBackground(Color.RED); }
```


Exemples d'utilisation

- Cela permet par exemple de :
 - connaître l'objet source de n'importe quel événement (*EventObject*) en utilisant la méthode *getSource*
 - connaître les coordonnées du clic d'un *MouseEvent* en utilisant *getX* et *getY*
 - connaître le caractère tapé d'un *KeyEvent* en utilisant *getKeyChar*
 - etc.

Hierarchie des événements AWT



Classes adaptateurs

- Certaines interfaces écouteurs comportent de nombreuses méthodes (7 pour *WindowListener*)
- Afin d'éviter d'implémenter certaines de ces méthodes avec un code vide, ces interfaces sont accompagnées de **classes adaptateurs** *XXXAdapter* dont les méthodes ne font rien ...
- Ainsi, **il suffira d'étendre la classe adaptateur afin de spécifier le traitement à effectuer** pour un (ou plusieurs) événement(s) sans s'occuper des autres

Exemple

- ```
class MaWindow extends WindowAdapter {
 public void windowClosing(WindowEvent e)
 { <traitement de cet événement> } // unique méthode définie
```
- NB : on peut définir un objet anonyme de la classe adaptateur (à la place d'un objet anonyme de l'interface écouteur)  

```
f.addWindowListener(
 new WindowAdapter() {
 public void windowClosing(WindowEvent e)
 { <traitement de cet événement.> } })
```

## Comment s'y retrouver avec toutes ces classes ?

- Il existe une interface écouteur (*XXXListener*) pour chaque classe d'événements (*XXXEvent*)
- Pour trouver l'interface à implémenter, **on peut se baser sur le composant qui va générer les événements**, puis repérer les méthodes nommées *addXXXListener* de sa classe (ou ascendante) et cerner celle qui pourrait convenir ...
- Enfin, il faut chercher dans les méthodes de *XXXListener* celle(s) qui correspond(ent) à notre (nos) événement(s)

## ***EVENEMENTS DE BAS NIVEAU***

## Événements de bas niveau : fenêtre (*WindowEvent*)

L'interface écouteur associée (*WindowListener*) comporte sept méthodes :

|                          |                                                         |
|--------------------------|---------------------------------------------------------|
| <i>windowOpened</i>      | appelée lorsque la fenêtre a été ouverte                |
| <i>windowClosing</i>     | lorsqu'une commande de fermeture de fenêtre a été émise |
| <i>windowClosed</i>      | lorsque la fenêtre a été fermée                         |
| <i>windowIconified</i>   | lorsque la fenêtre a été transformée en icône           |
| <i>windowDeiconified</i> | lorsque la fenêtre n'est plus à l'état d'icône          |
| <i>windowActivated</i>   | lorsque la fenêtre est devenue active                   |
| <i>windowDeactivated</i> | lorsque la fenêtre a été désactivée                     |

### Seconde interface écouteur associée

- Une seconde interface associée (*WindowStateListener*) comporte une unique méthode *windowStateChanged* **appelée lorsque la fenêtre a été changée d'état** (maximisée ou transformée en icône)
- L'objet événement (*WindowEvent*) transmis à toutes ces méthodes contient, en cas de changement d'état, **l'ancien et le nouvel état de la fenêtre, accessibles par :**

*getNewState* et *getOldState* qui renvoient un entier correspondant à une des valeurs suivantes : `Frame.NORMAL`, `Frame.ICONIFIED`, `Frame.MAXIMIZED_HORIZ`, `Frame.MAXIMIZED_VERT` et `Frame.MAXIMIZED_BOTH`

## Événements de bas niveau : clavier (*KeyEvent*)

- L'interface écouteur associée (*KeyListener*) comporte trois méthodes :  
*keyPressed* appelée lorsqu'une touche a été enfoncée  
*keyReleased* appelée lorsqu'une touche a été relâchée  
*keyTyped* appelée lors de la frappe d'un caractère Unicode
- L'objet événement (*KeyEvent*) transmis à ces trois méthodes contient **les informations nécessaires à l'identification de la touche ou du caractère concerné** :  
*getKeyCode* renvoie un entier qui est le code de la touche concernée (équivalent à une des constantes définies dans *KeyEvent* et correspondant à une des touches du clavier)  
*getKeyChar* renvoie le caractère concerné

## Autres informations sur le clavier

- On dispose également des méthodes *isAltDown*, *isAltGraphDown*, *isControlDown*, *isShiftDown* et *isMetaDown* qui renvoient *true* si la touche correspondante *Alt*, *Alt graphic*, *Ctrl*, *Shift* et « l'une des quatre » est pressée lors de l'événement
- NB : A un instant donné, un seul composant est sélectionné (on dit qu'il a le « focus »), il peut alors recevoir les événements « clavier » correspondants. Or, **les composants comme les panneaux ne peuvent pas recevoir la focalisation par défaut. Pour supprimer cette limitation, on doit utiliser la méthode *setFocusable*** :  
`panneau.setFocusable(true);` // pour réagir à des événements clavier

## Événements de bas niveau : souris (*MouseEvent*)

- L'interface écouteur associée (*MouseListener*) comporte 5 méthodes :

|                      |                                                                                       |
|----------------------|---------------------------------------------------------------------------------------|
| <i>mousePressed</i>  | lors d'un appui sur un bouton                                                         |
| <i>mouseReleased</i> | lors du relâchement d'un bouton                                                       |
| <i>mouseClicked</i>  | lors d'un clic complet (à condition que la souris n'ait pas été déplacée entre temps) |
| <i>mouseEntered</i>  | lors d'une entrée sur un composant                                                    |
| <i>mouseExited</i>   | lors d'une sortie de composant                                                        |


- Sur l'objet événement *MouseEvent*, on peut utiliser :

|                                              |                                        |
|----------------------------------------------|----------------------------------------|
| <i>getX</i> , <i>getY</i> et <i>getPoint</i> | renvoient les coordonnées du clic      |
| <i>getClickCount</i>                         | renvoie le nombre de clics consécutifs |

## Autres informations sur la souris

- La méthode *getModifiers* de *MouseEvent* permet de repérer quel bouton de souris a été enfoncé : elle renvoie un entier qu'il suffira de comparer avec les constantes définies dans *InputEvent* (*BUTTON1\_MASK*, *BUTTON2\_MASK*, *BUTTON3\_MASK*) pour savoir de quel bouton il s'agit (gauche, centre, droit)
- Il est possible de donner au pointeur de souris une forme différente (par exemple une croix) en utilisant la méthode *setCursor(Cursor)* de *Component* associée à la méthode *getPredefinedCursor* de *Cursor* :

```
setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
```



## Seconde interface écouteur associée

- Les événements de déplacement de la souris, très fréquents, sont définis dans une interface écouteur séparée (*MouseMotionListener*) qui comporte deux méthodes :  
*mouseMoved*            lors d'un déplacement sans bouton enfoncé  
*mouseDragged*        lors d'un déplacement avec un bouton enfoncé
- La catégorie d'événements concernée reste *MouseEvent*, par contre il existe une classe adaptateur *MouseMotionAdapter*
- NB : la méthode *mouseMoved* n'est plus appelée quand la souris sort du composant au contraire de *mouseDragged* qui continue à l'être tant que le bouton n'est pas relâché

## Événements de bas niveau : focus (*FocusEvent*)

- Une seule fenêtre à la fois peut être active et un seul composant à la fois dans une fenêtre peut avoir le focus. Celui-ci peut être modifié par un clic de souris sur un autre composant ou par la touche de tabulation qui permet de passer au composant « suivant » (de gauche à droite et de haut en bas).
- L'interface écouteur associée *FocusListener* comporte 2 méthodes :  
*focusGained* appelée lorsqu'un composant prend le focus  
*focusLost*    appelée lorsqu'un composant perd le focus
- La méthode *getOppositeComponent* de la classe *FocusEvent* renvoie le composant qui a perdu le focus en cas de gain et inversement

# *ACTIONS*

## Objectif

- *Java* offre un dispositif permettant d'associer **le même écouteur à plusieurs sources d'événement : les actions**. Ainsi, on pourra par exemple sélectionner une couleur de fond par un clic sur un bouton, un élément de menu, une frappe au clavier, etc. sans dupliquer le code.
- Ce dispositif s'appuie sur l'interface *Action* (qui dérive de l'interface *ActionListener*) et en particulier sur **la méthode *actionPerformed* qui contiendra le code de l'action**



## Interface *Action*

- Un objet d'une classe implémentant l'interface *Action* contiendra des informations descriptives sur l'action (comme le nom ou l'icône associés) ainsi que des paramètres nécessaires à l'exécution du traitement (par exemple la couleur de fond souhaitée)
- L'interface *Action* comporte les méthodes suivantes :
  - actionPerformed* // déclarée dans *ActionListener*, comporte le code
  - setEnabled* // active ou désactive l'action
  - isEnabled* // vérifie si l'action est activée
  - putValue* // permettent de stocker et récupérer des propriétés
  - getValue* // comme le nom ou l'icône associée
  - addPropertyChangeListener* // permettent de notifier le changement de
  - removePropertyChangeListener* // propriétés aux objets concernés

## La classe *AbstractAction*

- **Classe *Java* implémentant l'interface *Action*** (sauf la méthode *actionPerformed*) et qui est donc capable de gérer les propriétés, de notifier leurs changements d'états, etc.
- **Le programmeur n'a plus qu'à étendre la classe *AbstractAction* et à implémenter la méthode *actionPerformed* pour arriver à ses fins**
- NB : les noms des propriétés prédéfinies d'une action sont données dans l'interface *Action* sous forme de constantes : *NAME*, *SMALL\_ICON*, *SHORT\_DESCRIPTION*, etc.

## Exemple (clic sur un bouton générant une action)

```
class MonPanneauAction extends JPanel {
 private JButton bouton;
 public MonPanneauAction() {
 RougeAction ra = new RougeAction(
 "Rouge", new ImageIcon ("C:\\im\\red-ball.gif"));
 bouton = new JButton(ra); add(bouton); }
 // ce constructeur de JButton permet de définir l'action en tant
 // qu'écouteur, il n'y a pas besoin d'abonnement explicite
 private class RougeAction extends AbstractAction {
 public RougeAction(String n, Icon i) {
 putValue(Action.NAME, n); putValue(Action.SMALL_ICON, i);
 putValue("Couleur", Color.RED);
 putValue(Action.SHORT_DESCRIPTION, "changer en rouge");}
 public void actionPerformed(ActionEvent e) {
 Color c = (Color) getValue("Couleur"); setBackground(c); }}
}
```

## Autres exemples générant la même action

- Pour les événements sémantiques, portant par exemple sur un élément de menu, l'association de la même action à ce composant se fait de manière analogue au slide précédent
- Pour les événements de bas niveau, portant par exemple sur une frappe au clavier, c'est un peu plus compliqué mais on arrive au même résultat
- En bout du compte, la même méthode *actionPerformed* sera déclenchée quelle que soit la source de l'événement

## Opération inverse : 1 source d'événement, plusieurs écouteurs

- *Java* permet d'ajouter plusieurs écouteurs à une source d'événement de façon à ce qu'un même événement puissent activer plusieurs écouteurs qui réagiront en même temps
- Par exemple, un clic sur un seul bouton pourra fermer toutes les fenêtres créées jusqu'alors ...
- Attention, *Java* ne garantit pas l'ordre dans lequel les événements sont diffusés à plusieurs écouteurs abonnés à la même source, donc il ne faut pas proposer de traitement dépendant de l'ordre ...

## ***COMPOSANTS DE TEXTE***

## Composants de texte

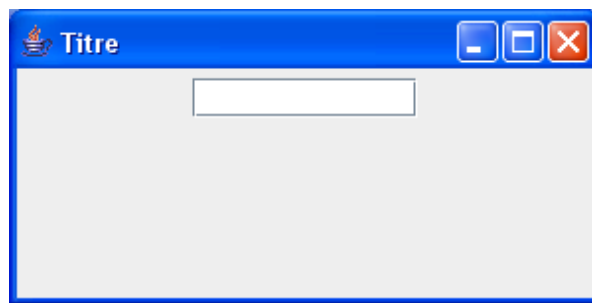
- Les composants de texte permettent la saisie de texte (pour s'identifier, donner un mot clé, sauvegarder une donnée, donner des informations, etc.) sur une ou plusieurs lignes
- Ils font partie de **composants plus élaborés** permettant de créer des interfaces dotées d'un plus grand nombre de fonctionnalités que celles étudiées jusqu'à présent

## Champ de texte

- Ce composant, qui est défini par la classe *TextField*, permet de **saisir un texte d'une seule ligne**
- Il est nécessaire de donner une **taille au constructeur** qui exprime le nombre de colonnes du champ
- Il est possible de donner un **texte par défaut** en premier paramètre du constructeur

## Exemple

```
public class MaFenetre extends JFrame {
 private JTextField jtf;
 public MaFenetre(String titre, int x, int y, int l, int h) {
 setBounds(x, y, l, h); setTitle(titre);
 setLayout(new FlowLayout());
 jtf = new JTextField(10);
 add(jtf); }}
```

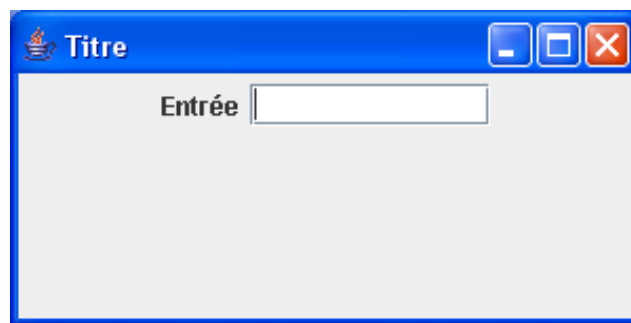


## Etiquette

- **Pour associer une étiquette identifiant le champ de texte, il faut passer par un composant *JLabel* avec le texte (ou l'icône) voulu**
- **Prendre soin de positionner l'étiquette convenablement** par rapport au composant qu'elle est censée identifier (gestionnaire de mise en forme)
- Le texte de l'étiquette n'est pas modifiable par l'utilisateur mais il l'est par la méthode *setText* (ou *setIcon*) de *JLabel* à tout moment

## Exemple

```
public class Mafenetre extends JFrame {
 private JTextField jtf; private JLabel jl;
 public Mafenetre(String titre, int x, int y, int l, int h) {
 setBounds(x, y, l, h); setTitle(titre);
 setLayout(new FlowLayout());
 jl = new JLabel("Entrée"); add(jl);
 jtf = new JTextField(10); add(jtf);
 }
}
```



## Remarques

- Quelques méthodes :  
    setText(String)      modifie le texte du champ  
    getText()            renvoie le texte contenu dans le champ  
    setColumns(int)      modifie le nombre de colonnes du champ
- Après avoir utilisé *setColumns*, il faudra appeler la méthode *revalidate* (de *JComponent*) sur le champ de texte qui sera alors redimensionné, ou la méthode *validate* (de *Component*) sur la fenêtre dont tous les composants seront alors recalculés
- Si le nombre de colonnes est insuffisant, un mécanisme de défilement est géré par *Swing* pour saisir le bon nombre de caractères

## Gestion de l'événement associé

- Pour exploiter la saisie ou la modification du texte, une première solution consiste à **gérer l'événement *ActionEvent* provoqué par l'appui sur la touche de validation** quand le champ de texte est sélectionné :

...

```
ActionListener al = new MonActionListener();
jtf.addActionListener(al); }
```

```
private class MonActionListener implements ActionListener {
 public void actionPerformed(ActionEvent evt){
 System.out.println(jtf.getText()); } ...
```

## Autre solution

- **Gérer l'événement *FocusEvent* provoqué par la perte de focus du champ de texte** (au cas où l'utilisateur quitterait le champ de texte sans valider son contenu) :

...

```
FocusListener fl = new MonFocusListener();
jtf.addFocusListener(fl); }
```

```
private class MonFocusListener implements FocusListener {
 public void focusGained (FocusEvent evt){ }
 public void focusLost(FocusEvent evt){
 System.out.println(jtf.getText()); } ...
```

## Meilleure solution !

- Garder la trace de tout changement intervenant dans un champ de texte
- Pour cela il n'est pas suffisant de traiter les événements « clavier » car certaines touches ne modifient pas le texte et le texte peut être modifié autrement que par le clavier (par exemple un copier-coller)
- Nécessité de passer par la notion de « modèle » associée aux composants de texte

## Séparation Modèle-Vue

- Les composants *Swing* sont implémentés en utilisant **un modèle séparé de la vue** : le modèle contient les données et la vue donne une représentation visuelle de ces données
- **Un modèle peut avoir plusieurs vues différentes, et la cohérence entre le modèle et les différentes vues est assurée** en cas de modification de données



## Utilisation

- Dans de nombreux cas, il est possible d'utiliser des composants graphiques *Swing* sans se soucier de la notion de modèle
- Mais parfois le modèle est bien utile :
  - pour séparer les opérations liées à la vue et celles liées au modèle
  - pour avoir plusieurs vues d'un même ensemble de données (sans dupliquer le stockage des données en mémoire)
  - pour récupérer ou mettre à jour des données provenant de sources extérieures

## Exemple de l'interface *Document*

- Elle décrit le modèle pour tous les composants de texte
- Pour garder la trace de tout changement intervenant dans un champ de texte, il faudra **gérer les événements *DocumentEvent* provoqués par toute modification de l'objet *Document* associé au champ de texte** (NB : cet objet s'obtient par la méthode *getDocument* de *JTextComponent*)

## L'interface *DocumentListener*

- L'interface écouteur associée (*DocumentListener*) comporte trois méthodes (pas de classe adaptateur) :

*insertUpdate* appelée lorsque des caractères ont été insérés

*removeUpdate* appelée lorsque des caractères ont été supprimés

*changedUpdate* appelée pour certains types de modifications comme des changements de mise en forme (jamais pour un champ de texte)

- La gestion des deux premiers événements permettra donc de garder la trace précise des modifications du champ de texte

## Exemple

...

```
DocumentListener dl = new MonDocumentListener();
```

```
jtf.getDocument().addDocumentListener(dl); }
```

```
private class MonDocumentListener implements DocumentListener {
```

```
 public void insertUpdate (DocumentEvent evt){
```

```
 System.out.println(jtf.getText()); }
```

```
 public void removeUpdate(DocumentEvent evt){
```

```
 System.out.println(jtf.getText()); }
```

```
 public void changedUpdate(DocumentEvent evt){} } ...
```

**NB : pour un (ou plusieurs en même temps) caractère(s) inséré(s) ou supprimé(s), un événement est généré**

## Champ de mot de passe

- Type spécial de champ de texte (implémenté par *JPasswordField*) dans lequel les caractères tapés ne sont pas affichés (à la place apparaissent des caractères d'écho, par défaut des \*, mais qui peuvent être redéfinis par la méthode *setEchoChar* de *JPasswordField*)
- NB : on retrouve la séparation modèle-vue qui utilise le modèle d'un champ de texte pour stocker les données, et une vue qui n'affiche que des caractères d'écho
- La méthode *getPassword* renverra le texte contenu dans le champ de mot de passe sous forme de tableau de caractères

## Champ de texte mis en forme

- Pour faciliter les saisies « numériques » (nombres, dates, etc.) *Swing* fournit une classe *JFormattedTextField* qui permet de valider une entrée, en utilisant *NumberFormat*, *DateFormat*, etc.
- *DefaultFormatter* fournit des formateurs d'objets de toute classe qui disposent d'un constructeur avec un paramètre *String*
- *MaskFormatter* fournit des formateurs à taille fixe avec des caractères constants et des caractères variables
- Il est également possible de définir des formateurs personnalisés en créant une classe qui dérive de *DefaultFormatter*.

## Zone de texte

- Ce composant, qui est défini par la classe *JTextArea*, permet de **saisir un texte d'une longueur supérieure à une ligne**, en utilisant la touche « Entrée » pour séparer les lignes
- Il est nécessaire de donner le nombre de lignes et de colonnes au constructeur pour définir la dimension de la zone
- Il est possible d'activer le retour automatique à la ligne avec la méthode *setLineWrap* de *JTextArea* (mais aucun caractère '*\n*' n'est inséré dans le modèle)

## Gestion des événements associés

- La méthode *getText* (de *JTextComponent*) fonctionne donc pour *JTextArea* par contre l'appui sur la touche de validation (« Entrée ») ne provoque pas d'événement *ActionEvent*
- Pour exploiter la saisie ou la modification du texte, il faut passer par un clic sur un bouton, un changement de focus, une modification de l'objet *Document* associé à la zone de texte , etc.
- La méthode *append* de *JTextArea* ajoute le texte spécifié en argument à la fin du texte présent dans la zone de texte, tandis que *setColumns* et *setRows* modifient le nombre de colonnes et de lignes de la zone

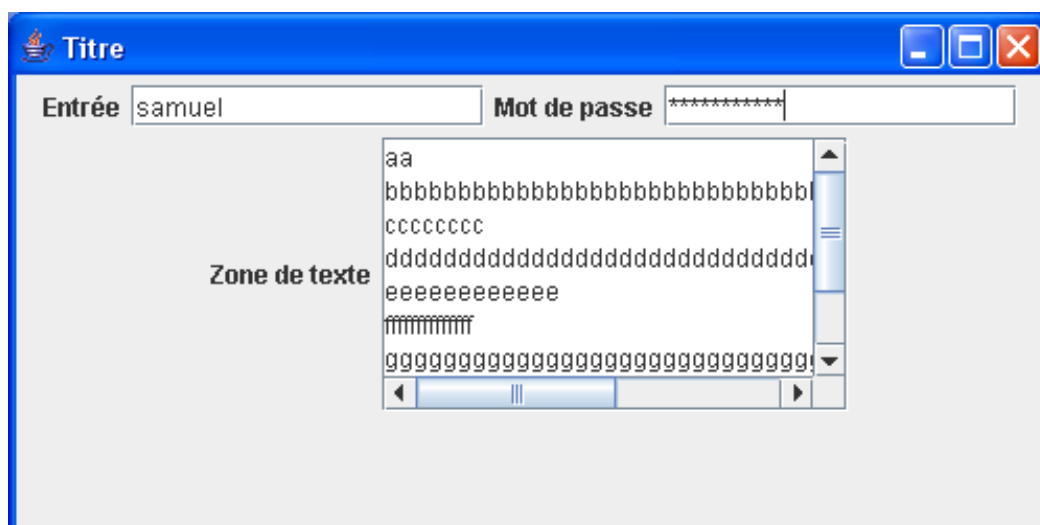
## Panneau de défilement

- Pour ajouter des barres de défilement à une zone de texte, il faut mettre celle-ci dans un **panneau de défilement** *JScrollPane* :

```
zt = new JTextArea(4, 10);
JScrollPane sp = new JScrollPane(zt);
add(sp);
```

- Ainsi, des barres de défilement apparaîtront automatiquement si le texte saisi dépasse la zone d'affichage (en hauteur ou en largeur) et disparaîtront si le texte supprimé permet de remettre le texte restant dans la zone d'affichage

## Exemple



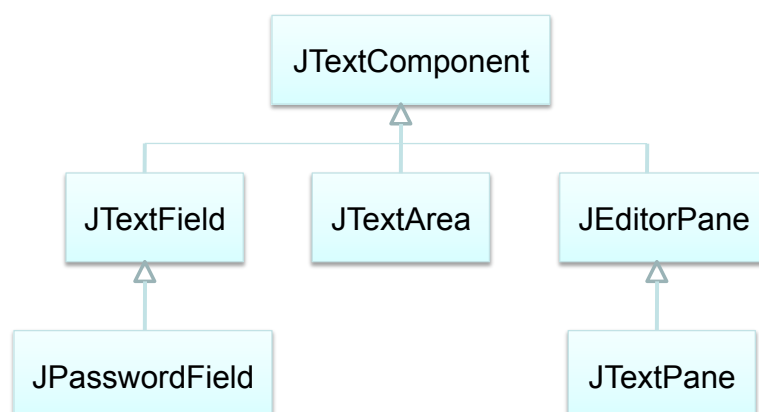
## *JEditorPane*

- *JEditorPane* affiche et modifie du texte au format HTML ou RTF, et permet d'implémenter, grâce à la classe *EditorKit*, d'autres formats
- Ce composant permet d'obtenir un navigateur très simple avec des liens hypertextes actifs si on gère l'événement «clic sur un lien hypertexte» (*HyperlinkEvent*) dont l'interface écouteur *HyperlinkListener* contient une seule méthode *hyperlinkUpdate*
- En mode «édition», il est possible de saisir du texte et de le modifier mais les fonctionnalités offertes par *JEditorPane* sont assez limitées

## Hiérarchie des composants de texte

- *JTextPane* peut, en plus, gérer du texte stylisé et ajouter des composants dans l'objet *JTextPane*

- Hiérarchie :



# Interface Homme Machine – Cours n°3

## Autres Composants Swing

- Composants de choix
- Menus
- Boîtes de dialogue
- Composants évolués (*JList, JTree, JTable*)
- Gestionnaires de mise en forme avancés
- Compléments

Alain SAMUEL

Polytech Marseille / Informatique

## ***COMPOSANTS DE CHOIX***

## Cases à cocher (*JCheckBox*)

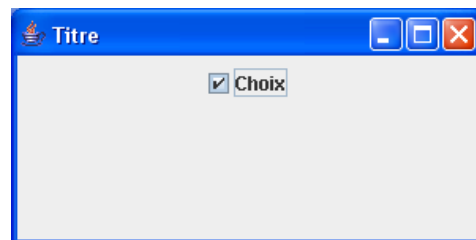
- **Proposer des choix à l'utilisateur** (plutôt que de saisir des données dans un composant de texte)

- **Case à cocher : réponse en oui/non**

Elle est accompagnée d'un libellé, donné au constructeur, qui permet d'identifier le choix associé

```
cc = new JCheckBox("Choix");
```

après un clic de l'utilisateur



## Événements associés

- Lorsque l'utilisateur clique sur une case à cocher (ou sur un «espace» si la case a le focus d'entrée), **un événement *ActionEvent* et un événement *ItemEvent* sont générés** (l'interface écouteur *ItemListener* contient une seule méthode *itemStateChanged*), on traitera celui qu'on veut ...
- Indépendamment de l'action de l'utilisateur, on peut, à tout instant, imposer à une case un état donné grâce à la méthode *setSelected* :  
`cc.setSelected(true); // coche cc et déclenche l'événement associé`
- On peut connaître l'état d'une case grâce à la méthode *isSelected*



## Remarques

- Le constructeur de *JCheckBox* permet d'imposer un état initial coché :  
`cc = new JCheckBox("Choix" , true); // par défaut non coché`
- On utilise souvent plusieurs cases à cocher côte à côte : leurs états (cochés ou non) sont totalement indépendants



## Boutons radio

- A l'inverse, les boutons radio ne permettent qu'un seul choix parmi un ensemble (lorsqu'un second choix est sélectionné, le premier est désactivé)
- La classe *JRadioButton* définit chaque bouton radio qui doivent être regroupés au sein d'un groupe grâce à la classe *ButtonGroup* :  

```
JRadioButton r1 = new JRadioButton("Radio1");
JRadioButton r2 = new JRadioButton("Radio2", true); // sélectionné
JRadioButton r3 = new JRadioButton("Radio3");
ButtonGroup g = new ButtonGroup();
g.add(r1); g.add(r2); g.add(r3);
add(r1); add(r2); add(r3); // chaque bouton et non le groupe
```

## Evénements associés

- Lorsque l'utilisateur clique sur un bouton radio *r1* (ou sur un «espace» si *r1* a le focus d'entrée), cela génère :
  - un *ActionEvent* et un *ItemEvent* pour *r1*
  - un *ItemEvent* pour le bouton radio préalablement sélectionné
- Comme pour les cases à cocher, on dispose des méthodes *setSelected* et *isSelected*
- Apparence des boutons radio :



## Bordures

- *Swing* fournit un ensemble de bordures qui s'applique à n'importe quel composant : on pourra donc placer une bordure autour d'un panneau contenant des boutons radio (pour les regrouper graphiquement)
- La méthode *setBorder(Border)* de *JComponent* définit la bordure où l'interface *Border* est implémentée par un ensemble de classes (*TitledBorder*, *LineBorder*, *SoftBevelBorder*, etc.) créant des bordures, de plus la classe *BorderFactory* regroupe toutes ces fonctionnalités

## Exemples

- `panneau.setBorder(new TitledBorder("Choisissez une option"));`



- `panneau.setBorder(new LineBorder(Color.RED, 5, true));`



- `Border etc = BorderFactory.createEtchedBorder  
    (EtchedBorder.RAISED, Color.RED, Color.BLACK);`  
`Border tit = BorderFactory.createTitledBorder(etc, "Choix");`  
`panneau.setBorder(tit);`

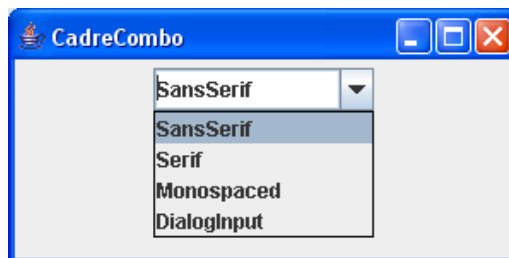


## Listes déroulantes (ou combinées)

- Quand ils sont nombreux les **boutons radio occupent trop d'espace**, dans ce cas **on peut leur substituer une liste déroulante**
- Lorsque l'utilisateur clique sur la liste (flèche de droite), **un ensemble de choix se déroule et il peut ainsi faire sa sélection**
- Un autre avantage de la liste déroulante est que, si elle est éditable, **l'utilisateur peut directement saisir sa propre sélection** : elle permet donc d'allier la souplesse d'un champ de texte et d'une liste de choix prédéfinie

## Exemple

Liste déroulante pliée et déroulée



## JComboBox

- ***JComboBox* implémente les listes déroulantes**, avec les méthodes :  
setEditable(boolean) rend la liste éditable ou pas (*false* par défaut)  
addItem(Object), removeItem(Object) ajoute, supprime un élément  
getSelectedIndex() renvoie le rang de la sélection (-1 si saisie directe)  
setSelectedIndex(int) force la sélection d'un élément de rang donné  
Object getSelectedItem() renvoie la sélection courante
- Un constructeur de *JComboBox* initialise la liste avec un tableau :  
String[] couleurs = {"rouge", "bleu", "vert", "jaune", "noir"}  
JComboBox cb = new JComboBox(couleurs);

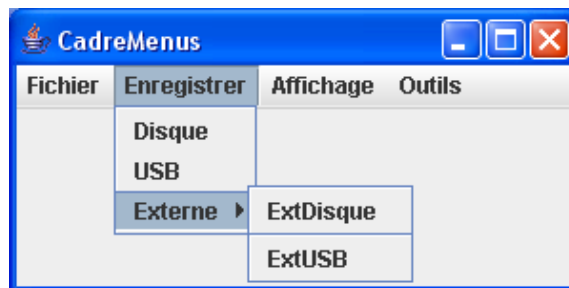
## Événements associés

- Lorsque l'utilisateur sélectionne une option de la liste (ou valide une saisie directe), **un *ActionEvent* est généré** (le choix est alors accessible par *getSelectedItem*)
- En cas de saisie directe non validée, et comme pour les champs de texte, aucun événement *ActionEvent* n'est généré en cas de perte de focus, il faudra donc décider si on gère un *FocusEvent* ...
- Par ailleurs, à chaque modification de la sélection, **deux *ItemEvent* sont générés** (suppression d'une sélection, nouvelle sélection)

## *MENUS*

## Menus déroulants

- Les menus déroulants sont constitués par une **barre de menus** (*JMenuBar*) située en haut de la fenêtre, et dans laquelle **chaque menu** (*JMenu*) pourra faire apparaître une **liste d'options** (*JMenuItem*)



## Création de menus déroulants

- La création d'une barre de menus se fait en utilisant la classe *JMenuBar* et la méthode *setJMenuBar* de *JFrame* :

```
JMenuBar mb = new JMenuBar(); mafenetre.setJMenuBar(mb);
```

- La création d'un menu se fait en utilisant la classe *JMenu* et la méthode *add* (à un argument *JMenu*) de *JMenuBar* :

```
JMenu m = new JMenu("Fichier"); mb.add(m);
```

- La création d'une option se fait en utilisant la classe *JMenuItem* et la méthode *add* (à un argument *JMenuItem*) de *JMenu* :

```
JMenuItem mi = new JMenuItem("Disque"); m.add(mi);
```

## Remarques

- Il est possible de créer un sous-menu à la place d'une option (*JMenu* est une sous-classe de *JMenuItem*)
- Un trait séparateur entre deux options peut être obtenu en utilisant la méthode *addSeparator* de *JMenu*
- Il existe également une méthode *add* à un argument *String* de *JMenu* qui permet d'ajouter une option sans passer par un *JMenuItem*
- Il est possible d'associer une icône à une option de menu :  

```
JMenuItem mi = new JMenuItem("Disque",
 new ImageIcon("C:\\i386\\redshd.gif"));
```

## Événements associés

- Lorsque l'utilisateur sélectionne une option, un événement *ActionEvent* est déclenché
- Des événements *MenuEvent* sont également générés par des opérations sur les menus, l'interface écouteur *MenuListener* contient trois méthodes (sans adaptateurs) *menuSelected*, *menuDeselected* et *menuCanceled* (en pratique, la gestion de ces événements est peu utilisée)

## Option « case à cocher » ou « boutons radio »

- Il est possible de proposer une case à cocher ou des boutons radio comme option de menu en utilisant les classes *JCheckBoxMenuItem* et *JRadioButtonMenuItem* de la même manière que *JMenuItem*
- Au contraire des options usuelles, il n'est pas obligatoire de traiter ces options au moment de leur sélection, on peut se contenter de s'intéresser à leur état à un moment donné (par la méthode *isSelected*) pour accomplir un traitement particulier
- Par contre, on peut traiter les événements habituels associés à ces composants (*ActionEvent* et *ItemEvent*) si besoin ...

## Menus surgissants

- Un menu surgissant n'est pas attaché à une barre de menu mais apparaît quelque part dans la fenêtre suite à une action de l'utilisateur
- La création et l'utilisation d'un menu surgissant se fait en utilisant la classe *JPopupMenu*, la classe *JMenuItem* déjà vue pour ajouter des options, et les événements habituels associés à la sélection des options (*ActionEvent* et éventuellement *ItemEvent*)

```
JPopupMenu m = new JPopupMenu(); // pas de nom de menu
JMenuItem i1 = new JMenuItem("Disque");
m.add(i1);
JMenuItem i2 = new JMenuItem("USB");
m.add(i2);
```



## Affichage

- L'affichage d'un menu surgissant se fait en utilisant la méthode **show(Component, int, int)** de *JPopupMenu*

// composant parent et position relative du menu surgissant

// attention, le composant parent doit avoir été rendu visible ...

- Le menu restera affiché jusqu'à ce que l'utilisateur sélectionne une option ou bien qu'il clique à côté du menu pour le faire disparaître

## Facteur déclencheur

- Il est possible de faire apparaître un menu surgissant en réaction à n'importe quel événement généré par l'utilisateur mais **l'usage est de le réserver à un clic sur le bouton (en général le droit) de la souris**

- Pour cela, il faudra implémenter *mouseReleased* de *MouseListener* :

```
public void mouseReleased (MouseEvent e)
```

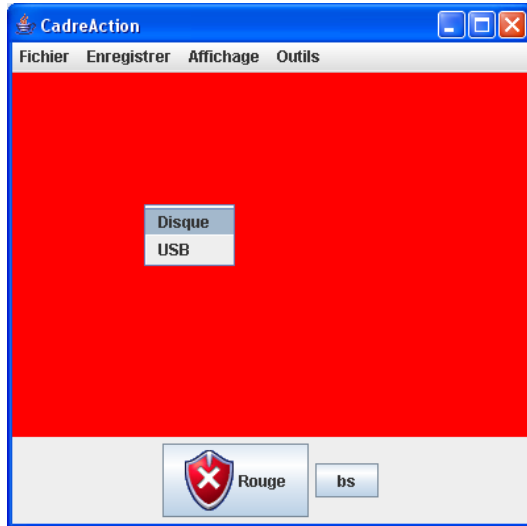
```
{ if (e.isPopupTrigger()) ms.show(composant, e.getX(), e.getY()); }
```

```
// si c'est le bouton usuellement déclencheur du menu surgissant
```

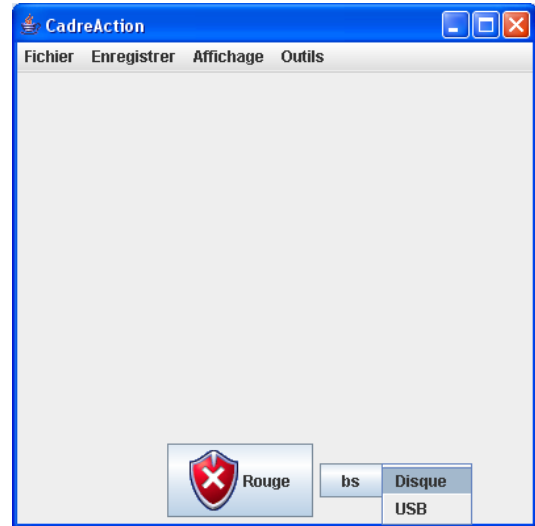
- Pour un *JComponent* on peut faire apparaître un menu surgissant si l'utilisateur clique sur ce composant, avec la méthode :

**setComponentPopupMenu(JPopupMenu)** de *JComponent*

## Exemple



Par un événement quelconque  
(clic sur le bouton « Rouge »)  
Position donnée dans le panneau



Par clic droit sur le composant  
*JButton* « bs »  
Position du clic

## *BOITES DE DIALOGUE*

## Boîtes de dialogue

- Etablir un dialogue temporaire avec l'utilisateur : la boîte de dialogue peut apparaître ou disparaître globalement
- Il existe des boîtes de dialogue **modales** (l'utilisateur ne peut pas agir sur d'autres composants que ceux de la boîte tant qu'il n'a pas mis fin au dialogue) et d'autres **non modales** (celles offertes en standard par *Java* sont modales mais il est possible d'en créer des non modales)

### Boîtes de dialogue « standard »

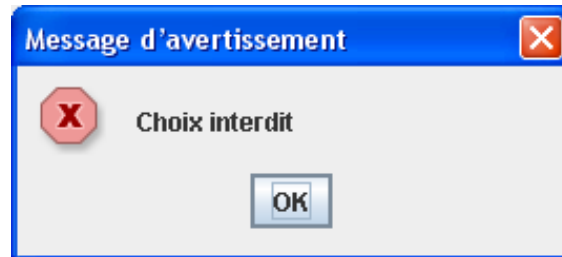
- Elles permettent de demander à l'utilisateur de fournir une information, par l'intermédiaire de la classe *JOptionPane* et des ses quatre méthodes *static*, qui affiche un message et :

|                                |                                                                  |
|--------------------------------|------------------------------------------------------------------|
| <code>showMessageDialog</code> | attend que l'utilisateur clique sur OK                           |
| <code>showConfirmDialog</code> | attend un choix de type oui/non                                  |
| <code>showOptionDialog</code>  | attend un choix parmi plusieurs                                  |
| <code>showInputDialog</code>   | attend une saisie de texte ou un choix dans une liste déroulante |

- Il faudra donc choisir un type de boîte de dialogue, une icône, un message et les options correspondant au type de boîte choisi

## Exemple de boîte de message

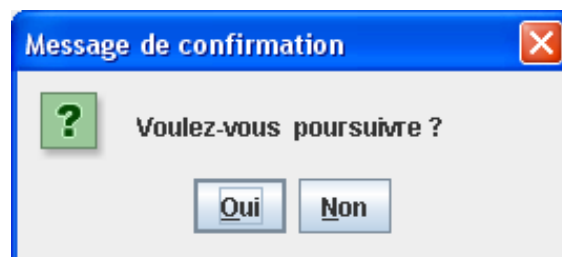
- `JOptionPane.showMessageDialog(fenetre, "Choix interdit", "Message d'avertissement", JOptionPane.ERROR_MESSAGE);`



- NB : le premier argument est le composant dans lequel la boîte de dialogue va s'afficher (si *null*, la boîte de dialogue sera indépendante) tandis que le dernier argument désigne un type d'icône

## Exemple de boîte de confirmation

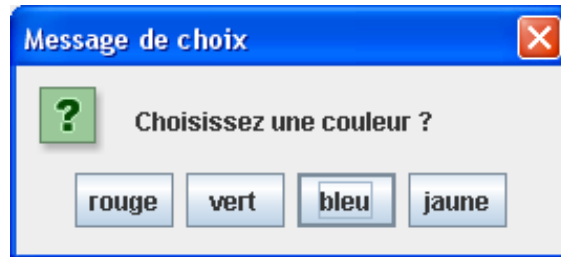
- `JOptionPane.showConfirmDialog(fenetre, "Voulez-vous poursuivre ?", "Message de confirmation", JOptionPane.YES_NO_OPTION);`



- NB : le dernier argument désigne la nature des boutons présents (il existe également les boutons *OK* et *CANCEL*) tandis que la méthode *showConfirmDialog* renvoie une constante (`JOptionPane.YES_OPTION`, `JOptionPane.NO_OPTION`, etc.) qui identifie la réponse de l'utilisateur

## Exemple de boîte de choix

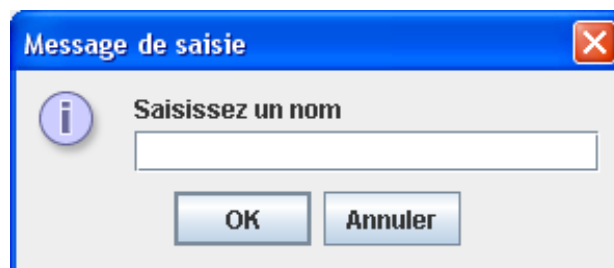
- String [] couleurs = {"rouge", "vert", "bleu", "jaune"};  
JOptionPane.showOptionDialog(fenetre, "Choisissez une couleur ?",  
"Message de choix", JOptionPane.DEFAULT\_OPTION,  
JOptionPane.QUESTION\_MESSAGE, null, couleurs, couleurs[2]);



- NB : la méthode *showOptionDialog* renvoie l'indice de l'option choisie par l'utilisateur (-1 s'il ferme la boîte de dialogue)

## Exemple de boîte de saisie

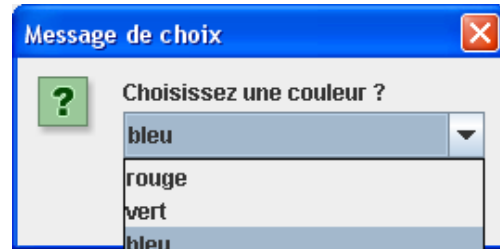
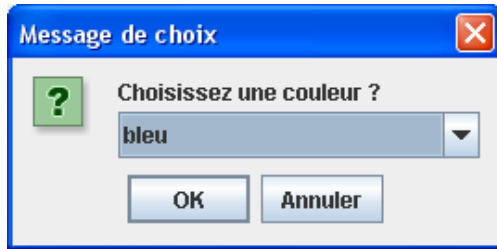
- JOptionPane.showInputDialog(fenetre, "Saisissez un nom",  
"Message de saisie", JOptionPane.INFORMATION\_MESSAGE);



- NB : la méthode *showInputDialog* renvoie une *String* correspondant au texte saisi par l'utilisateur (et validé par «Entrée» ou le bouton «OK»), ou la valeur *null* si l'utilisateur a agi sur «Annuler» ou a fermé la boîte

## Autre exemple de boîte de saisie (liste déroulante)

- String [] couleurs = {"rouge", "vert", "bleu", "jaune"};  
JOptionPane.showInputDialog(fenetre, "Choisissez une couleur ?",  
"Message de choix", JOptionPane.QUESTION\_MESSAGE,  
null, couleurs, couleurs[2]);



- La méthode *showInputDialog* renvoie un *Object* correspondant au choix de l'utilisateur (et validé par «Entrée» ou le bouton «OK»), ou la valeur *null* si l'utilisateur a agi sur «Annuler» ou a fermé la boîte

## Boîtes de dialogue « personnalisées »

- Pour disposer de boîtes de dialogue plus élaborées, il faut les créer à partir de la **classe *JDialog*** (conteneur de 1<sup>er</sup> niveau), par exemple :  
`JDialog d = new JDialog(cadre, "Ma boîte", false);`  
où *cadre* est le conteneur de 1<sup>er</sup> niveau dont la boîte de dialogue dépend (lorsqu'il sera fermé ou iconifié les boîtes de dialogue qui en dépendent seront fermés ou iconifiées), puis le titre et la modalité
- Le processus est alors identique à celui qui permet de créer la fenêtre principale à partir de *JFrame* (ajouter des composants et des gestionnaires d'événements, définir la taille, la position, etc.) y compris l'utilisation de la méthode *setVisible(true)* pour l'afficher

## Exemple

- La classe ci-dessous est une classe «dialogue» personnalisée (sous-classe de *JDialog*) avec ses paramètres de base, deux composants et un gestionnaire de mise en forme *FlowLayout*
- ```
class MonDialog extends JDialog {  
    private JButton mBouton;  
    private JTextField cTexte;  
    public MonDialog(JFrame p) {  
        super(p, "Ma boîte", false); setBounds(200, 100, 200, 200);  
        mBouton = new JButton("OK");  cTexte = new JTextField(5);  
        setLayout(new FlowLayout()); // BorderLayout par défaut  
        add(mBouton); add(cTexte);} ...}
```

Fin du dialogue

- Si l'utilisateur clique sur le bouton "OK" (ou sur un éventuel bouton "Annuler"), le **gestionnaire d'événement doit appeler *setVisible(false)*** qui rend la boîte invisible et met fin au dialogue (si l'utilisateur ferme directement la boîte de dialogue, le processus par défaut fera de même)
- Dans le cas d'une validation du dialogue ("OK"), il faudra récupérer les informations données par l'utilisateur dans la boîte de dialogue
- Quand on n'a plus besoin d'une boîte de dialogue, la méthode *dispose* permet de libérer les ressources mémoires associées

COMPOSANTS EVOLUES

(*JList*, *JTree*, *JTable*)

Composant évolué : *JList*

- Ce composant, variante évoluée de *JComboBox* (mais non éditable), **permet de choisir plusieurs valeurs dans une liste prédéfinie** (on peut le restreindre à sélectionner une seule valeur ou une seule plage de valeurs contiguës)
- *JList* ne génère pas d'événement *ActionEvent* mais des événements *ListSelectionEvent* dont l'interface écouteur *ListSelectionListener* comporte une seule méthode *valueChanged*

Exemple simple

```
JList jl = new JList(new String[] {"bleu", "blanc", "rouge", "jaune", "vert",  
    "noir", "orange", "violet", "marron", "gris"});  
  
jl.setVisibleRowCount(4);  
  
// jl.setLayoutOrientation(JList.VERTICAL_WRAP);  
  
// jl.setLayoutOrientation(JList.HORIZONTAL_WRAP);  
  
JScrollPane sp = new JScrollPane(jl);
```

Variantes selon l'orientation de la disposition



- Pour sélectionner plusieurs éléments, il faut appuyer sur la touche *Ctrl* tout en cliquant sur chaque élément
- Pour sélectionner plusieurs éléments contigus, il faut cliquer sur le premier d'entre eux puis cliquer sur le dernier d'entre eux en appuyant sur la touche *Maj*

Quelques méthodes

- La méthode *getSelectedValues* de *JList* renvoie un tableau d'objets contenant tous les éléments sélectionnés tandis que *setSelectedIndex(int)* force une sélection

- La méthode *setSelectionMode* de *JList* permet de restreindre la sélection à un choix unique :

ListSelectionModel.SINGLE_SELECTION ou bien à un choix unique ou une plage de choix :

ListSelectionModel.SINGLE_INTERVAL_SELECTION ou bien à un choix multiple sur plusieurs intervalles (mode par défaut) :

ListSelectionModel.MULTIPLE_INTERVAL_SELECTION

Evénements de transition

- Lors d'une sélection d'éléments de liste, des événements de transition sont générés (par exemple lorsque l'utilisateur appuie sur le bouton de la souris, puis lorsqu'il le relâche), ce qui multiplie le nombre d'événements ...
- La méthode *getValueIsAdjusting* de *ListSelectionEvent* renvoie *true* si c'est un événement de transition, ce qui permet de le détecter, pour éventuellement ne pas le traiter ...

Exemple



```
jl.addListSelectionListener(new ListSelectionListener() {  
    public void valueChanged(ListSelectionEvent e)  
        {if (!e.getValueIsAdjusting())  
        {Object[] valeurs = jl.getSelectedValues();  
        for (Object v : valeurs) System.out.println((String) v);}}  
});
```

Modèles de sélection de listes

- Il est possible de **personnaliser son mode de sélection de liste**, par exemple de façon à ne pas utiliser la touche *Ctrl* pour ajouter un élément mais de simples clicks de sélection ou de désélection
- Pour cela, il faut **définir son propre modèle de sélection** en utilisant l'interface *ListSelectionModel*, plus précisément en créant une sous-classe de *DefaultListSelectionModel* (classe qui implémente *ListSelectionModel*) redéfinissant la méthode *setSelectionInterval* qui gère l'ajout ([i1,i2] dernier intervalle à ajouter à la sélection) :

```
public void setSelectionInterval(int i1, int i2) {  
    if (i1 == i2) if (isSelectedIndex(i1)) removeSelectionInterval(i1, i1);  
    else addSelectionInterval(i1, i1); else super.setSelectionInterval(i1, i2); }
```

Caractéristiques de *JList*

- ***JList* est un composant beaucoup plus évolué que *JComboBox*** car il permet de manipuler des listes d'objets arbitraires qui peuvent être modifiés, affichés de manière personnalisée, etc.
- En premier lieu, et comme les composants de texte avec l'interface *Document*, ***JList* utilise un modèle pour récupérer les données** par un objet qui implémente l'interface *ListModel*

Modèles de listes

- ```
public interface ListModel {
 int getSize();
 Object getElementAt(int i);
 void addListDataListener(ListDataListener l);
 void removeListDataListener(ListDataListener l); }
```
- Les deux premières méthodes renvoient le nombre d'éléments et l'élément à la position donnée tandis que les deux suivantes ajoutent et suppriment des écouteurs au modèle

## Insertion et suppression d'éléments de liste

- Pour cela, **il est nécessaire de passer par un modèle**, en particulier par *DefaultListModel* (sous-classe de *AbstractListModel* qui gère l'ajout et la suppression d'écouteurs), qui implémente l'interface *ListModel* en utilisant un *Vector*
- Plus précisément, **il faut remplir un objet *DefaultListModel* avec les données initiales et l'associer à la liste :**  

```
DefaultListModel m = new DefaultListModel();
m.addElement("bleu"); m.addElement("blanc"); ...
JList jl = new JList(m);
```
- Il est désormais possible d'insérer ou de supprimer des éléments du **modèle qui notifiera alors à la liste les changements effectués pour qu'elle se redessine**

## Exemple



Les instructions

```
m.removeElement("rouge"); m.addElement("rose");
```

déclenchent la mise à jour immédiate de la liste de sélection



## Affichage personnalisé d'éléments de liste

- **Personnaliser l'affichage des éléments de liste** en implémentant l'interface *ListCellRenderer*, ainsi chaque élément de liste appellera un « renderer » qui prendra en charge son dessin (une classe *DefaultListCellRenderer* est utilisée pour les cas standards)
- Par exemple, on pourra faire en sorte que les éléments de la liste apparaissent en blanc sur fond rouge s'ils sont sélectionnés et en bleu sur fond blanc sinon
- **Dans ce cas simple nous pouvons nous appuyer sur la classe *JLabel*** qui fournit déjà toutes les fonctionnalités nécessaires à l'affichage d'un élément de liste (en particulier *JLabel* implémente déjà *paintComponent*)

### L'interface *ListCellRenderer*

- Elle possède une seule méthode :  
`Component getListCellRendererComponent(JList l, Object element, int indice, boolean selection, boolean focus);`  
qui renvoie un composant configuré pour dessiner le contenu de la cellule
- Les arguments de cette méthode fournissent les informations nécessaires au dessin (la liste à laquelle il s'applique, l'élément à dessiner, son indice, s'il est sélectionné, s'il possède le focus)

## Cas simple (on s'appuie sur *JLabel*)

```
class CouleurCellRenderer extends JLabel implements ListCellRenderer {
 public Component getListCellRendererComponent(JList l, Object e, int i,
 boolean sel, boolean foc) {
 setOpaque(true);
 setText((String) e);
 if (sel) {setBackground(Color.red); setForeground(Color.white);}
 else {setBackground(Color.white); setForeground(Color.blue);}
 return this;}}

```

Utilisation sur l'objet *jl* de *JList* :

```
jl.setCellRenderer(new CouleurCellRenderer());
```



## Cas extrêmes

- Dans le cas où l'affichage est plus compliqué que du texte et un changement de couleur, le « **renderer** » **devra étendre la classe *JPanel*, implémenter l'interface *ListCellRenderer*, et définir lui-même le code du dessin dans la méthode *paintComponent***
- A l'inverse, s'il s'agit d'un affichage à peine différent de celui par défaut, par exemple afficher tous les éléments en majuscule, il suffit d'étendre la classe *DefaultListCellRenderer* (« **renderer** » par défaut, sous-classe de *JLabel*) en redéfinissant la méthode *setText* :

```
public void setText(String value)
 {super.setText(value.toUpperCase());}
```

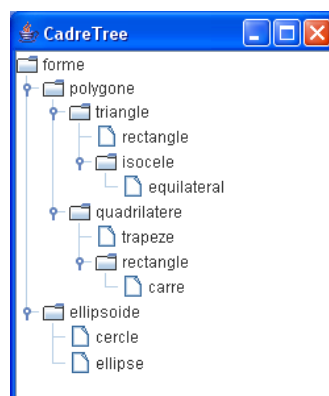
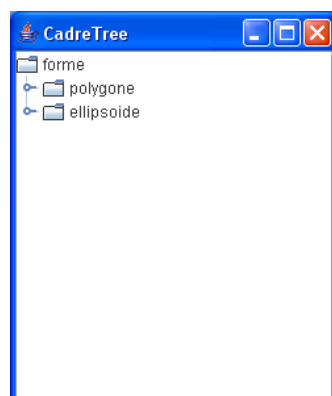
## Composant évolué : *JTree*

- **Mêmes principes que ceux d'une liste (JList)** pour représenter un arbre, à peine plus complexes ...
- On retrouve l'architecture modèle/vue avec la notion de **modèle d'arbre** représenté par une classe qui implémente l'interface *TreeModel*, par exemple *DefaultTreeModel*
- **Les nœuds de l'arbre sont représentés par des objets *TreeNode***, interface implémentée par exemple par *DefaultMutableTreeNode*
- Un constructeur de *DefaultTreeModel* ou de *JTree* construira alors le modèle d'arbre par l'intermédiaire du nœud racine *TreeNode*

## Exemple de création d'arbre

```
DefaultMutableTreeNode racine = new DefaultMutableTreeNode("forme");
DefaultMutableTreeNode n1 = new DefaultMutableTreeNode("polygone");
DefaultMutableTreeNode n2 = new DefaultMutableTreeNode("ellipsoïde");
racine.add(n1); racine.add(n2); ...
jt = new JTree(racine); add(new JScrollPane(jt));
```

A l'affichage  
seuls le nœud  
racine et ses  
enfants sont  
visibles



Après avoir  
cliqué sur les  
poignées, la  
totalité de  
l'arbre est  
visible



## Modifier un arbre

- Pour identifier les nœuds d'un arbre, la classe *JTree* gère des chemins d'arbre (listes d'ancêtres) jusqu'au nœud considéré
- Par exemple, la méthode *getLastSelectedPathComponent* de *JTree* renvoie l'objet du premier nœud sélectionné
- Connaissant ce nœud, vous pourrez insérer un nouveau nœud enfant, supprimer ou modifier ce nœud, en utilisant les méthodes *insertNodeInto*, *removeNodeFromParent* ou *nodeChanged* de la classe *DefaultTreeModel*

## Afficher les nœuds d'un arbre

- De manière comparable à la classe *JList*, il est possible de **personnaliser l'affichage d'un nœud grâce aux «renderers»**
- Pour un arbre, on manipulera l'interface *TreeCellRenderer* ainsi que la classe *DefaultTreeCellRenderer* qui l'implémente et étend *JLabel*
- NB : La méthode *getTreeCellRendererComponent* de *TreeCellRenderer* contient plus d'arguments que son homologue utilisé pour *JList*

## Autres opérations

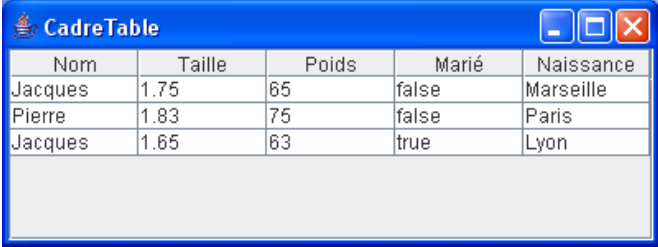
- De manière comparable à la classe *JList*, **pour réagir à la sélection de nœuds de l'arbre**, il faut installer un écouteur de sélection qui implémente l'interface *TreeSelectionListener* possédant une seule méthode *valueChanged*
- **Pour récupérer les sélections courantes** (sous forme de listes d'ancêtres), vous pouvez utiliser la méthode *getSelectionPaths* de *JTree*
- **Pour définir le mode de sélection des nœuds**, vous devez utiliser l'interface *TreeSelectionModel*

## Composant évolué : *JTable*

- **Mêmes principes que pour *JList* et *JTree*** pour représenter et manipuler des tableaux, structures bi-dimensionnelles d'objets
- Exemple simple :

```
Object [] [] donnees = {
 {"Jacques", 1.75, 65, false, "Marseille"},
 {"Pierre", 1.83, 75, false, "Paris"},
 {"Jacques", 1.65, 63, true, "Lyon"} };
String[] colonnes = {"Nom", "Taille", "Poids", "Marié", "Naissance"};
JTable t = new JTable(donnees, colonnes);
JScrollPane p = new JScrollPane(t);
```

## Exemple simple



| Nom     | Taille | Poids | Marié | Naissance |
|---------|--------|-------|-------|-----------|
| Jacques | 1.75   | 65    | false | Marseille |
| Pierre  | 1.83   | 75    | false | Paris     |
| Jacques | 1.65   | 63    | true  | Lyon      |

- Ce tableau possède par défaut des fonctionnalités très riches :
  - Les noms des colonnes restent visibles même avec un ascenseur vertical
  - Les colonnes peuvent être déplacées
  - La largeur des colonnes peut être modifiée
  - Les lignes peuvent être sélectionnées
  - Les données peuvent être modifiées (néanmoins, sans code supplémentaire, les modifications ne seront pas reportées dans le modèle de données)

## Modèles de tableaux

- **Il vaut mieux implémenter son propre modèle de tableau** (que de placer les données directement dans un tableau à deux dimensions)
- De manière analogue à *JList* et *JTree*, vous bénéficiez de l'interface *TableModel* implémentée par *AbstractTableModel* et *DefaultTableModel*

- **Il suffira de fournir trois méthodes :**

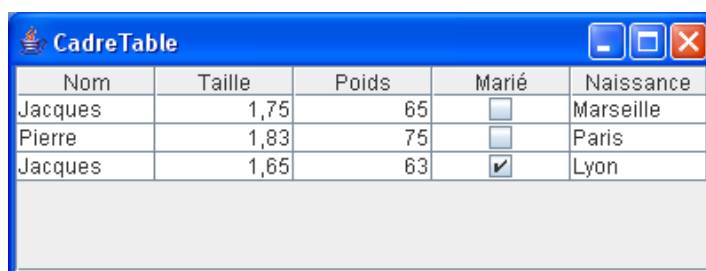
```
int getRowCount() // nombre de lignes
int getColumnCount() // nombre de colonnes
Object getValueAt(int ligne, int colonne)
```

**// cette dernière méthode pourra calculer la réponse, ou chercher la valeur dans une BD ou une autre source de données, car le modèle de tableau n'a pas besoin de stocker les données**

## Affichage des cellules de tableaux

- Les modèles de tableaux permettent d'utiliser les « renderer » par défaut de certains types (*Icon*, *Boolean*, *Number*, etc.) : il suffira de définir la méthode *Class getColumnClass(int indiceColonne)* de façon à ce qu'elle renvoie le type des objets de la colonne donnée
- Pour personnaliser les autres types, on pourra manipuler l'interface *TableCellRenderer* ainsi que la classe *DefaultTableCellRenderer* qui l'implémente et étend *JLabel*
- Les paramètres de *getTableCellRendererComponent* désignent en particulier le tableau, la cellule, sa ligne et sa colonne

### Exemple d'utilisation des « renderer » par défaut



| Nom     | Taille | Poids | Marié                               | Naissance |
|---------|--------|-------|-------------------------------------|-----------|
| Jacques | 1,75   | 65    | <input type="checkbox"/>            | Marseille |
| Pierre  | 1,83   | 75    | <input type="checkbox"/>            | Paris     |
| Jacques | 1,65   | 63    | <input checked="" type="checkbox"/> | Lyon      |

- Les nombres sont alignés à droite et les booléens sont représentés par des cases à cocher

- Le code est le suivant :

```
JTable tab = new JTable(modele); // tableau défini à partir d'un modèle
public Class getColumnClass(int c) {return getValueAt(0, c).getClass();}
// définition de getColumnClass de AbstractTableModel
```

## Autres opérations possibles

- Modifier une cellule
- Créer un éditeur de cellules personnalisé
- Sélectionner des lignes, des colonnes ou des cellules
- Cacher et afficher des colonnes
- Ajouter et supprimer des lignes
- etc.

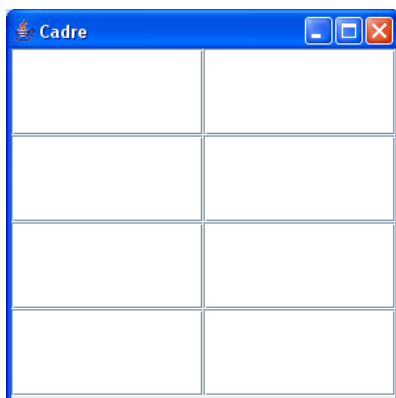
NB : Le composant *JTable* est celui qui offre le plus de fonctionnalités bien qu'il soit simple à analyser au niveau conceptuel. Son implémentation complète est donc la plus complexe à analyser et dépasse le cadre de ce cours ...

# ***GESTIONNAIRES DE MISE EN FORME AVANCES***

## Gestionnaires de mise en forme avancés

- Les gestionnaires *BorderLayout* et *FlowLayout*, même en les combinant, ne suffisent pas toujours à placer et organiser précisément les composants. *Java* propose de nombreux gestionnaires de mise en forme qui permettent, éventuellement en les combinant, un contrôle total de l'apparence de l'application.
- Le gestionnaire *GridLayout* permet d'organiser les composants dans une grille, un peu comme un tableur, mais toutes les lignes et les colonnes ont une taille identique. Lors d'un re-dimensionnement de la fenêtre, les composants s'agrandissent ou diminuent tout en conservant des tailles identiques entre eux.

### Exemples de *GridLayout*



avec 8 *TextField*



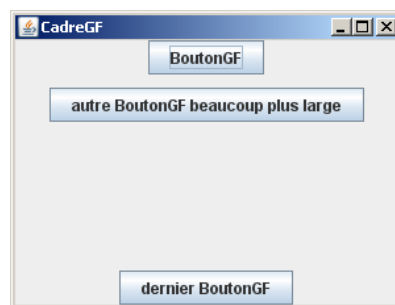
avec 16 *Button*

NB : *GridLayout* prend comme paramètres les nombres de lignes et de colonnes mais s'il a plus (resp. moins) de composants à gérer que lignes x colonnes, il ajoutera (resp. enlèvera) des colonnes. De plus, si un des paramètres a la valeur 0, cela signifie que le nombre de lignes (ou de colonnes) est indifférent

## *BoxLayout*

- Le gestionnaire *BoxLayout* permet d'organiser les composants sur **une seule ligne ou une seule colonne** (contrairement à *GridLayout*, les composants n'ont pas forcément la même largeur et la même hauteur)
- La classe conteneur *Box*, dont le gestionnaire par défaut est *BoxLayout*, fournit un ensemble de méthodes statiques facilitant la disposition des composants, par exemple :
  - *createGlue* crée un composant invisible qui s'étire avec la fenêtre
  - *createVerticalStrut(int)* (ou *createHorizontalStrut*) crée un composant invisible de hauteur (ou largeur) fixe exprimée en pixels

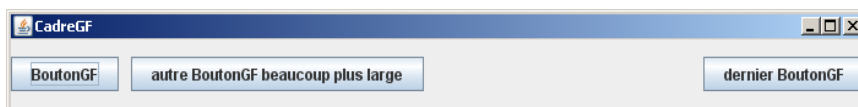
## Exemples de *BoxLayout*



NB : Avec une taille fixe à 10 pixels entre les deux premiers boutons et une « glu » entre les deux derniers qui permet de les écarter quand on agrandit la fenêtre



NB : idem à l'horizontal



## Autres gestionnaires

- Le **gestionnaire** *GridBagLayout* permet d'organiser les composants dans une grille, comme *GridLayout*, mais les composants peuvent occuper plusieurs cases. Les contraintes de placement, pour chaque composant, sont données par un objet *GridBagConstraints*.
- Le **gestionnaire** *SpringLayout* associe une notion de « ressort » à chaque composant, qui permet de spécifier leur position précise

## Autres mises en forme

- **N'utiliser aucun gestionnaire pour placer un composant à un endroit précis**, en définissant le gestionnaire à *null* (par *setLayout*) et en utilisant *setBounds*
- **Concevoir son propre gestionnaire de mise en forme**, en implémentant l'interface *LayoutManager* et ses cinq méthodes :

|                                    |                                            |
|------------------------------------|--------------------------------------------|
| <code>addLayoutComponent</code>    | // ajoute un composant                     |
| <code>removeLayoutComponent</code> | // supprime un composant                   |
| <code>preferredLayoutSize</code>   | // calcule la taille préférée et la taille |
| <code>minimumLayoutSize</code>     | // minimale du conteneur                   |
| <code>layoutContainer</code>       | // effectue toute la mise en forme         |



# COMPLEMENTS

## Raccourcis clavier

- Il est possible d'associer un **caractère mnémonique** à un menu ou une option de menu (il apparaîtra souligné) :

```
JMenu jm = new JMenu("Fichier");
jm.setMnemonic('F'); // pareil pour JMenuItem
```

ou bien (seulement pour *JMenuItem*) :

```
JMenuItem jmi = new JMenuItem("Disque", 'D');
```

- Pour sélectionner une option, il suffira de taper son caractère mnémonique quand le menu est déroulé
- Pour sélectionner un menu dans la barre de menus, il suffira de taper la touche *Alt* et son caractère mnémonique

## Raccourcis clavier (suite)

- Il est également possible d'associer une **combinaison de touches** à une option sans avoir à ouvrir le menu (elle apparaîtra à côté du nom de l'option) :

```
JMenuItem jmi = new JMenuItem("Disque");
jmi.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_D,
 InputEvent.CTRL_MASK)); // pour Ctrl-D
```

- Les combinaisons de touches ne sont pas utilisables pour les menus (qu'elles n'ouvrent pas), elles déclenchent directement l'événement associé à l'option (comme lors d'une sélection manuelle)

## Activation et désactivation d'options

- Dans certains contextes, une option peut n'avoir aucun sens, il faut alors la désactiver (elle apparaîtra grisée et ne pourra pas être sélectionnée) en utilisant la méthode *setEnabled* :

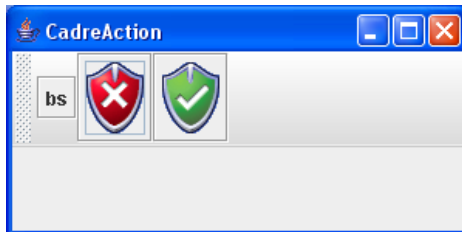
```
jmi.setEnabled(false);
```

- Pour faire cela de façon organisée, on peut implémenter la méthode *menuSelected* de l'interface écouteur *MenuListener* de façon à désactiver (ou réactiver) les options concernées juste avant qu'elles ne soient affichées par le clic sur le menu (attention cela ne marchera pas pour les options déclenchées par des combinaisons de touches car dans ce cas il n'y a pas de sélection de menu)

## Barres d'outils

- Les barres d'outils contiennent des boutons (en général sous forme d'icônes) qui permettent d'accéder rapidement aux commandes les plus utilisées, elles sont créées par la classe *JToolBar* :

```
JToolBar tb = new JToolBar("Barre d'Outils"); // titre facultatif
tb.add(monbouton); // un bouton existant (il est alors déplacé)
tb.add(ra); // une action (seul l'icône apparaît)
tb.add(new JButton(new ImageIcon("vert.gif"))); // nouveau bouton
mafenetre.add(tb, BorderLayout.NORTH); // en haut de la fenêtre
```



On remarque la « poignée » de la barre d'outils

## Barres d'outils (suite)

- Il est possible de faire glisser une barre d'outils vers l'un des quatre bords de la fenêtre (pour la mettre en position verticale par exemple), ou bien vers l'intérieur de la fenêtre (la barre se transforme alors en une petite fenêtre, avec son titre qui apparaît)
- De plus, même si leur vocation est de contenir des boutons, elles peuvent contenir d'autres composants, par exemple une case à cocher ou un champ de texte ...



## Bulles d'aide

- Il est possible d'activer une bulle d'aide (dont le texte apparaît dans un rectangle coloré) lorsque le curseur stationne au dessus d'un objet *JComponent* et de la désactiver lorsque la souris est déplacée, grâce à la méthode *setToolTipText(String)* :

```
monbouton.setToolTipText("menu surgissant");
```

- Quand on utilise des objets *Action*, il faut utiliser la propriété prédéfinie *Action.SHORT\_DESCRIPTION* pour associer le texte de la bulle d'aide à l'action :

```
putValue(Action.SHORT_DESCRIPTION, "changer en rouge");
```