

Interfaces graphiques en Java - Introduction

Université de Nice - Sophia Antipolis

Version TA 2.1 – 13/4/02

Richard Grin

Contributions

- Beaucoup d'exemples de cette partie du cours sont fortement inspirés du livre

Au cœur de Java 2

Volume I - Notions fondamentales
de Horstmann et Cornell

The Sun Microsystems Press

Java Series

- De nombreuses images proviennent du tutoriel en ligne de *Sun* (gratuit) :

<http://java.sun.com/docs/books/tutorial/>

Plan de cette partie

- Généralités sur les interfaces graphiques
- Affichage d'une fenêtre
- Classes de base ; AWT et Swing
- Placer des composants dans une fenêtre
- Gestion des événements
- Modèle MVC ; exemple des listes
- Dessiner ; afficher une image

Généralités sur les interfaces graphiques

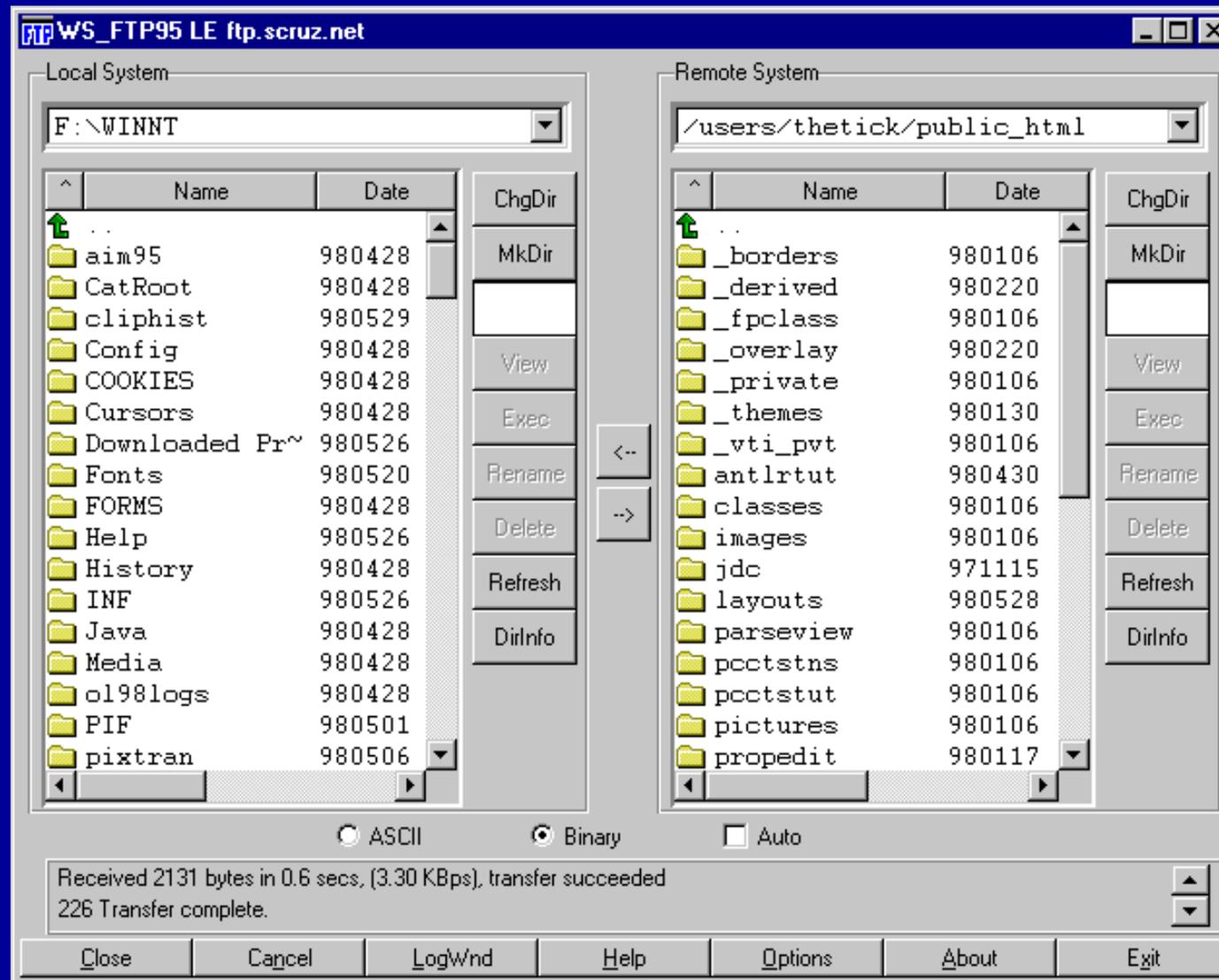
Interface avec l'utilisateur

- La quasi-totalité des programmes informatiques nécessitent
 - l'affichage de questions posées à l'utilisateur
 - l'entrée de données par l'utilisateur au moment de l'exécution
 - l'affichage d'une partie des résultats obtenus par le traitement informatique
- Cet échange d'informations peut s'effectuer avec une interface utilisateur (UI en anglais) en mode texte (ou console) ou en mode graphique

Interface graphique

- Une interface graphique est formée d'une ou plusieurs fenêtres qui contiennent divers composants graphiques (*widgets*) tels que
 - boutons
 - listes déroulantes
 - menus
 - champ texte
 - etc.
- Les interfaces graphiques sont souvent appelés **GUI** d'après l'anglais *Graphical User Interface*

Un exemple



Programmation avec interface graphique

- L'utilisateur peut interagir **à tout moment** avec plusieurs objets graphiques : cliquer sur un bouton, faire un choix dans une liste déroulante ou dans un menu, remplir un champ texte, etc...
- Ces actions peuvent modifier totalement le cheminement du programme, sans que l'ordre d'exécution des instructions ne puisse être prévu à l'écriture du code

Programmation conduite par les événements

- L'utilisation d'interfaces graphiques impose une façon particulière de programmer
- La programmation « conduite par les événements » est du type suivant :
 - les actions de l'utilisateur engendrent des **événements** qui sont mis dans une file d'attente
 - le programme récupère un à un ces événements et les traite

Boîtes à outils graphiques

- Les boîtes à outils graphiques offrent des facilités pour utiliser et gérer la file d'attente des événements
- En particulier pour associer les événements avec les traitements qu'ils doivent engendrer

La solution Java : les écouteurs

- Le JDK utilise une architecture de type « observateur - observé » :
 - les composants graphiques (comme les boutons) sont les **observés**
 - chacun des composants graphiques a ses **observateurs** (ou écouteurs, *listeners*), objets qui s'enregistrent (ou se désenregistrent) auprès de lui comme écouteur d'un certain type d'événement (par exemple, clic de souris)

Rôle des écouteurs

- Ces écouteurs sont prévenus par le composant graphique dès qu'un événement qui les concerne survient sur ce composant
- Le code de ces écouteurs exécute les actions à effectuer en réaction à l'événement
- Par exemple, l'écouteur du bouton « **Exit** » demandera une confirmation à l'utilisateur et terminera l'application

Les API utilisées pour les interfaces graphiques en Java

Les API

- 2 bibliothèques :
 - **AWT** (*Abstract Window Toolkit*, JDK 1.1)
 - **Swing** (JDK/SDK 1.2)
- Swing et AWT font partie de **JFC** (*Java Foundation Classes*) qui offre des facilités pour construire des interfaces graphiques
- Swing est construit au-dessus de AWT
 - même gestion des événements
 - les classes de *Swing* héritent des classes de AWT

Swing ou AWT ?

- Tous les composants de AWT ont leur équivalent dans Swing
 - en plus joli
 - avec plus de fonctionnalités
- Swing offre de nombreux composants qui n'existent pas dans AWT

Mais Swing est plus lourd et plus lent que AWT

⇒ Il est fortement conseillé d'utiliser les composants Swing et **ce cours sera donc centré sur Swing**

Paquetages principaux

- AWT : `java.awt` et `java.awt.event`
- Swing : `javax.swing`, `javax.swing.event`, et tout un ensemble de sous-paquetages de `javax.swing` dont les principaux sont
 - liés à des composants ; `table`, `tree`, `text` (et ses sous-paquetages), `filechooser`, `colorchooser`
 - liés au *look and feel* général de l'interface (plaf = *pluggable look and feel*) ; `plaf`, `plaf.basic`, `plaf.metal`, `plaf.windows`, `plaf.motif`

Afficher une fenêtre

Afficher une fenêtre

```
import javax.swing.JFrame;
```

```
public class Fenetre extends JFrame {
```

```
    public Fenetre() {
```

```
        super("Une fenêtre");
```

```
        setSize(300, 200);
```

```
        pack();
```

```
        setVisible(true);
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        JFrame fenetre = new Fenetre();
```

```
    }
```

```
}
```

ou
setTitle("...")

ou setBounds(...)

compacte le contenu de la fenêtre
(annule setSize)

ou show(),
affiche la fenêtre

Taille d'une fenêtre

- **pack ()** donne à la fenêtre la taille nécessaire pour respecter les tailles préférées des composants de la fenêtre (tout l'écran si cette taille est supérieure à la taille de l'écran)
- Taille ou un emplacement précis sur l'écran (en pixels) :
 - setLocation**(int xhg, int yhg) (ou `Point` en paramètre)
 - setSize**(int largeur, int hauteur) (ou `Dimension` en paramètre)
 - setBounds**(int x, int y, int largeur, int hauteur) (ou `Rectangle` en paramètre)

Positionnement d'une fenêtre et icône

(On doit importer `java.awt.*`)

```
public Fenetre() {
    // Centrage de la fenêtre
    Toolkit tk = Toolkit.getDefaultToolkit();
    Dimension d = tk.getScreenSize();
    int hauteurEcran = d.height;
    int largeurEcran = d.width;
    setSize(largeurEcran/2, hauteurEcran/2);
    setLocation(largeurEcran/4, hauteurEcran/4);
    // tant qu'on y est, ajoutons l'icône...
    Image img = tk.getImage("icone.gif");
    setIconImage(img);
    . . .
}
```

Classe `java.awt.Toolkit`

- Les sous-classes de la classe **abstraite** `Toolkit` implantent la partie de AWT qui est en contact avec le système d'exploitation hôte
- Quelques méthodes publiques :
`getScreenSize`, `getScreenResolution`,
`getDefaultToolkit`, `beep`, `getImage`,
`createImage`, `getSystemEventQueue`
- **`getDefaultToolkit`** fournit une instance de la classe qui implante `Toolkit` (classe donnée par la propriété `awt.toolkit`)

Composants lourds et légers

Classes `Container` et `JComponent`

Composants lourds

- Pour afficher des fenêtres (instances de **JFrame**), Java s'appuie sur les fenêtres fournies par le système d'exploitation hôte dans lequel tourne la JVM
- On dit que les **JFrame** sont des composants « lourds »
- L'utilisation de composants lourds améliore la rapidité d'exécution mais nuit à la portabilité et impose les fonctionnalités des composants

Composants légers

- Au contraire de AWT qui utilise les *widgets* du système d'exploitation pour tous ses composants graphiques (fenêtres, boutons, listes, menus, etc.), Swing ne les utilise que pour les fenêtres de base « *top-level* »
- Les autres composants, dits légers, sont **dessinés** dans ces containers lourds, par du code « pur Java »

Containers lourds

- Il y a 3 sortes de containers lourds (un autre, **JWindow**, est plus rarement utilisé) :
 - **JFrame** fenêtre pour les applications
 - **JApplet** pour les *applets*
 - **JDialog** pour les fenêtres de dialogue
- Pour construire une interface graphique avec Swing, il faut créer un (ou plusieurs) container lourd et placer à l'intérieur les composants légers qui forment l'interface graphique

Libérer les ressources associées à une **JFrame**

- En tant que composant lourd, une **JFrame** utilise des ressources du système sous-jacent
- Si on ne veut plus utiliser une **JFrame** (ou **JDialog** ou **JWindow**), mais continuer l'application, il faut lancer la méthode **dispose** () de la fenêtre ; les ressources seront rendues au système
- Voir aussi la constante **DISPOSE_ON_CLOSE** de l'interface `javax.swing.WindowConstants`

Composants légers (*lightweight*)

- La plupart des *widgets* de Swing sont des instances de sous-classes de la classe **JComponent**
- Les instances des sous-classes de **JComponent** sont de composants « légers »
- Attention, les composants lourds s'affichent toujours au-dessus des composants légers

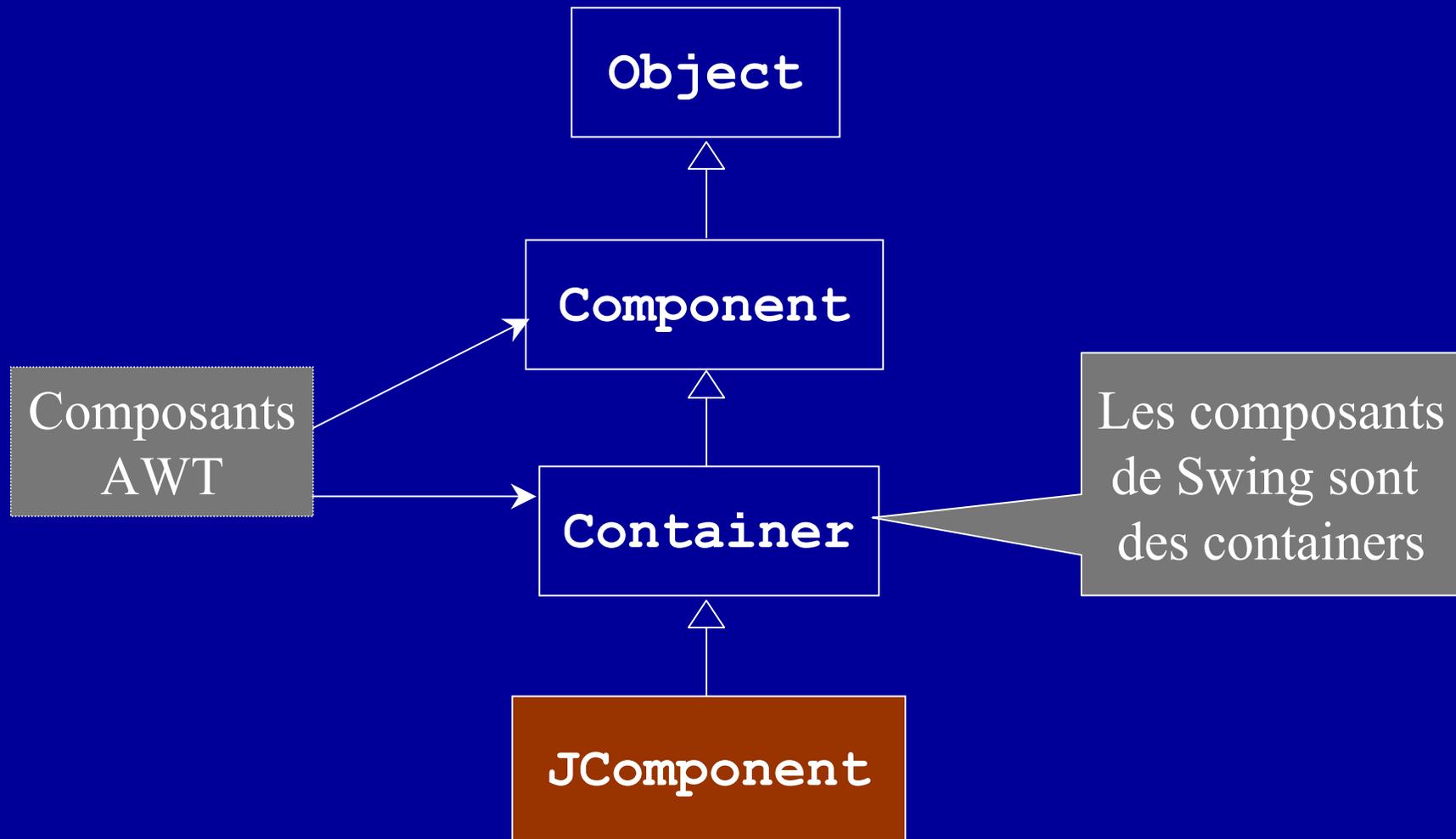
Les Containers

- Tous les composants légers des sous-classes de **JComponent** héritent de la classe **Container** et peuvent donc contenir d'autres composants
- Des composants sont destinés **spécifiquement** à recevoir d'autres éléments graphiques :
 - les containers « *top-level* » lourds **JFrame**, **JApplet**, **JDialog**, **JWindow**
 - les containers « **intermédiaires** » légers **JPanel**, **JScrollPane**, **JSplitPane**, **JTabbedPane**, **Box** (ce dernier est léger mais n'hérite pas de **JComponent**)

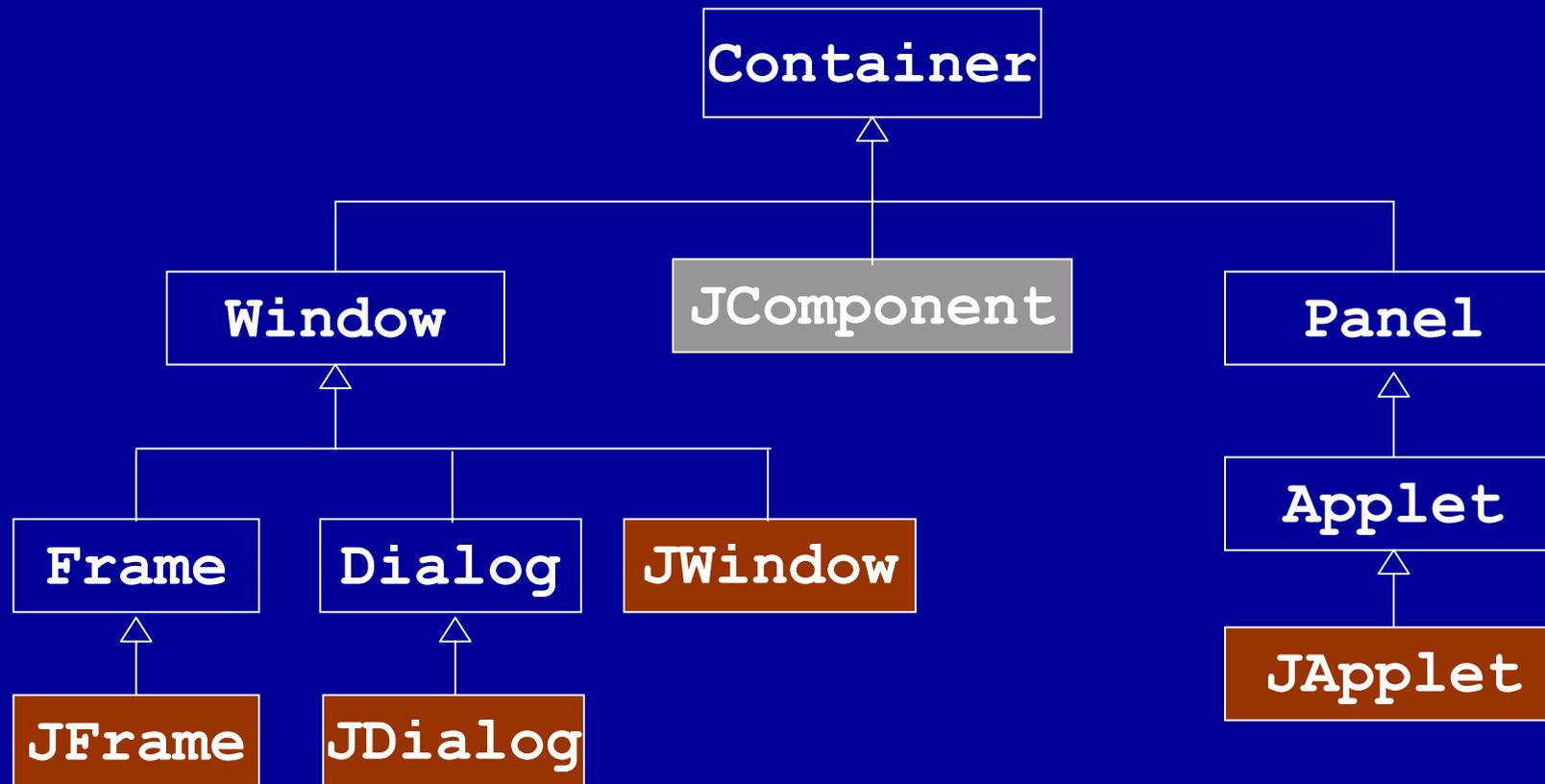
JPanel

- **JPanel** est la classe mère des **containers intermédiaires** les plus simples ; il sert à regrouper des composants dans une zone d'écran
- Un **JPanel** n'a pas d'aspect visuel déterminé ; son aspect visuel est donné par les composants qu'il contient
- Il peut aussi servir de composant dans lequel on peut dessiner ce que l'on veut, ou faire afficher une image (par la méthode **paintComponent**)

Classe abstraite JComponent



Hiérarchie d'héritage des containers lourds



Ajouter des composants dans une fenêtre

Le « *ContentPane* »

- Les containers « *top-level* » ne peuvent contenir *directement* d'autres composants
- Ils sont associés à un autre container, le « *content pane* » dans lequel on peut ajouter les composants
- On obtient ce *content pane* par (`topLevel` est un container lourd quelconque ; `JFrame` par exemple)

```
Container contentPane =  
    topLevel.getContentPane ();
```

Placer les composants

```
public Fenetre() {  
    . . .  
    Container contentPane = getContentPane();  
    JLabel label = new JLabel("Bonjour");  
    JButton b1 = new JButton("Cliquez moi !");  
    contentPane.add(label, BorderLayout.NORTH);  
    contentPane.add(b1, BorderLayout.SOUTH);  
    . . .  
}
```

Gestionnaires de mise en place

Layout managers

- L'utilisateur peut changer la taille d'une fenêtre ; les composants de la fenêtre doivent alors être repositionnés
- Les fenêtres (plus généralement les containers) utilisent des **gestionnaires de mise en place** (*layout manager*) pour repositionner leurs composants
- Il existe plusieurs types de *layout managers* avec des algorithmes de placement différents

Indications de positionnement

- Quand on ajoute un composant dans un container on ne donne pas la position exacte du composant
- On donne plutôt des indications de positionnement au gestionnaire de mise en place
 - explicites (`BorderLayout.NORTH`)
 - ou implicites (ordre d'ajout dans le container)

Algorithme de placement

- Un *layout manager* place les composants « au mieux » suivant
 - l’algorithme de placement qui lui est propre
 - les indications de positionnement des composants
 - la taille du container
 - les tailles préférées des composants

Classe `java.awt.Dimension`

- Cette classe est utilisée pour donner des dimensions de composants en pixels
- Elle possède 2 variables d'instance publiques de type `int`
 - `height`
 - `width`
- Constructeur : `Dimension(int, int)`

Tailles des composants

- Tous les composants graphiques (classe **Component**) peuvent indiquer leurs tailles pour l'affichage
 - taille maximum
 - taille préférée
 - taille minimum
- La taille préférée est la plus utilisée par les *layout managers* ; un composant peut l'indiquer avec la méthode **Dimension** **getPreferredSize()** ; on peut aussi l'imposer « de l'extérieur » avec la méthode **void** **setPreferredSize(Dimension)**
- Les méthodes **{get|set}{Maximum|Minimum}Size** sont liées aux autres tailles

Layout manager par défaut d'une fenêtre

- Par défaut, les fenêtres **JFrame** ont un gestionnaire de mise en place qui est une instance de la classe **BorderLayout**
- On peut changer le gestionnaire de mise en place d'un **Container** par la méthode **setLayout (*LayoutManager*)** de la classe **Container**

Types de *Layout manager*

- Les types les plus courants de gestionnaire de mise en place :
 - **BorderLayout** : placer aux 4 points cardinaux
 - **FlowLayout** : placer à la suite
 - **GridLayout** : placer dans une grille
 - **BoxLayout** : placer verticalement ou horizontalement
 - **GridBagLayout** : placements complexes

BorderLayout

- Affiche au maximum 5 composants (aux 4 points cardinaux et au centre)
- Essaie de respecter la hauteur préférée du nord et du sud et la largeur préférée de l'est et de l'ouest ; le centre occupe toute la place restante
- *layout manager* par défaut de **JFrame** et **JDialog**



BorderLayout

- Les composants sont centrés dans leur zone
- On peut spécifier des espacement horizontaux et verticaux minimaux entre les composants
- Si on oublie de spécifier le placement lors de l'ajout d'un composant, celui-ci est placé au centre (source de bug !)
- **Règle pratique** : l'est et l'ouest peuvent être étirés en hauteur mais pas en largeur ; le contraire pour le nord et le sud ; le centre peut être étiré en hauteur et en largeur

Placement dans une fenêtre complexe

- Pour disposer les composants d'une fenêtre de structure graphique complexe on peut :
 - utiliser des containers intermédiaires, ayant leur propre type de gestionnaire de placement, et pouvant éventuellement contenir d'autres containers
 - utiliser un gestionnaire de placement de type **GridBagLayout** (plus souple mais parfois plus lourd à mettre en œuvre)
 - mixer ces 2 possibilités

Utiliser un JPanel

panelBoutons



Utiliser un JPanel

```
public Fenetre() {  
    . . .  
    Container contentPane = getContentPane();  
    JPanel panelBoutons = new JPanel();  
    JButton b1 = new JButton("Cliquez moi !");  
    JButton b2 = new JButton("Et moi aussi !");  
    panelBoutons.add(b1); // FlowLayout  
    panelBoutons.add(b2);  
    contentPane.add(panelBoutons,  
                    BorderLayout.NORTH);  
    JTextArea textArea = new JTextArea(15, 5);  
    contentPane.add(textArea, BorderLayout.CENTER);  
    JButton quitter = new JButton("Quitter");  
    contentPane.add(quitter, BorderLayout.SOUTH);  
    . . .  
}
```

Source de la classe

Exécution

FlowLayout

- Rangement de haut en bas et de gauche à droite
- Les composants sont affichés à leur taille préférée
- *layout manager* par défaut de **JPanel** et **JApplet**
- Attention, la taille préférée d'un container géré par un **FlowLayout** est calculée en considérant que tous les composants sont sur une seule ligne



Code avec **FlowLayout**

```
JPanel panel = new JPanel();  
// FlowLayout par défaut dans un JPanel  
// mais si on ne veut pas centrer :  
panel.setLayout(  
    new FlowLayout(FlowLayout.LEFT));  
// On pourrait aussi ajouter des espaces  
// entre les composants avec  
// new FlowLayout(FlowLayout.LEFT, 5, 8)  
JButton bouton = new JButton("Quitter");  
JTextField zoneSaisie = new JTextField(20);  
panel.add(bouton);  
panel.add(zoneSaisie);
```

GridLayout

- Les composants sont disposés en lignes et en colonnes
- Les composants ont tous la même dimension
- Ils occupent **toute la place** qui leur est allouée
- On remplit la grille ligne par ligne



Code avec GridLayout

```
// 5 lignes, n colonnes  
// (on pourrait ajouter des espaces entre composants)  
panel.setLayout(new GridLayout(5,0));  
panel.add(bouton); // ligne 1, colonne 1  
panel.add(zoneSaisie); // ligne 2, colonne 1
```

- On doit indiquer le nombre de lignes *ou* le nombre de colonnes et mettre 0 pour l'autre nombre (si on donne les 2 nombres, le nombre de colonnes est ignoré !)
- L'autre nombre est calculé d'après le nombre d'éléments ajoutés

GridBagLayout

- Comme `GridLayout`, mais un composant peut occuper plusieurs « cases » du quadrillage ; la disposition de chaque composant est précisée par une instance de la classe `GridBagConstraints`
- C'est le *layout manager* le plus souple mais aussi le plus complexe



Code avec GridBagLayout

```
panel.setLayout(new GridBagLayout());
GridBagConstraints contraintes =
    new GridBagConstraints();
// ligne et colonne du haut gauche
contraintes.gridx = 0;
contraintes.gridy = 0;
// taille en lignes et colonnes (tient sur 2 lignes ici)
contraintes.gridheight = 2;
contraintes.gridwidth = 1;
// degré d'agrandissement relatif ; 0 : ne peut grandir
contraintes.weightx = 0.7;
contraintes.weighty = 0.5;
// Chaque élément ajouté peut avoir ses propres
    contraintes
panel.add(bouton, contraintes);
```

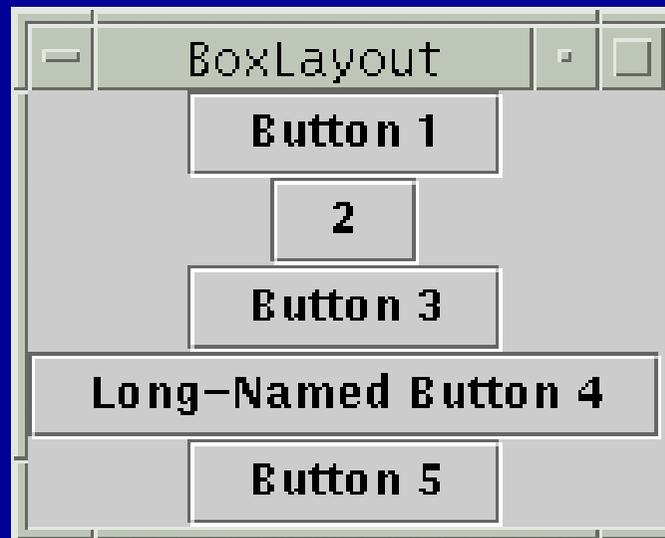
weightx : comment sera distribué l'espace libre entre les colonnes

Autres contraintes

- **fill** détermine si un composant occupe toute la place dans son espace réservé (constantes de la classe `GridBagConstraints` : `BOTH`, `NONE`, `HORIZONTAL`, `VERTICAL`)
- **anchor** dit où placer le composant quand il est plus petit que son espace réservé (`CENTER`, `SOUTH`, ...)
- **insets** ajoute des espaces autour des composants :
`contraintes.insets = new Insets(5,0,0,0)`
(OU `contraintes.insets.left = 5`)
- **ipadx**, **iapdy** ajoutent des pixels à la taille minimum des composants

BoxLayout

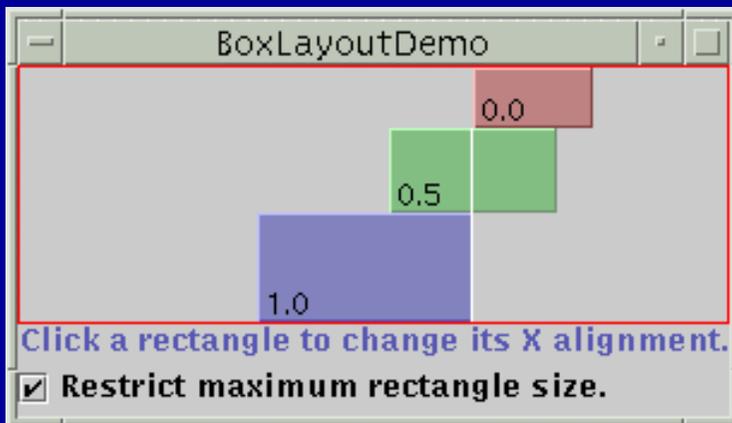
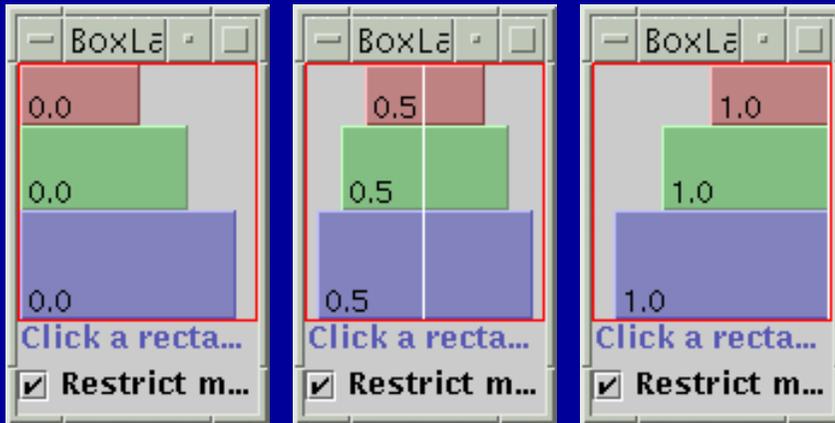
- Aligne les composants sur une colonne ou une ligne (on choisit à la création)
- Respecte la largeur (resp. hauteur) préférée et maximum, et l'alignement horizontal (resp. vertical)
- *Layout manager* par défaut de **Box** et de **JToolBar**



BoxLayout

- Pour un alignement vertical, les composants sont affichés centrés et si possible
 - à leur largeur préférée
 - respecte leur hauteur maximum et minimum
(`get{Maxi|Mini}mumSize()`)
- Pour un alignement horizontal, idem en intervertissant largeur et hauteur
- Pour changer les alignements, on peut utiliser les méthodes de la classe Component
`setAlignment{X|Y}`

Alignment {X|Y}



- Constantes de Component :
 - {LEFT|CENTER|RIGHT}_ALIGNMENT
 - {TOP|BOTTOM}_ALIGNMENT
- Méthodes :
 - `setAlignmentX` et `setAlignmentY`

Problèmes d'alignement

- Si tous les composants gérés par un **BoxLayout** n'ont pas le même alignement, on peut avoir des résultats imprévisibles
- Par exemple, le seul composant aligné à gauche peut être le seul qui n'est pas aligné à gauche !
- Il vaut donc mieux avoir le même alignement pour tous les composants

Classe **Box**

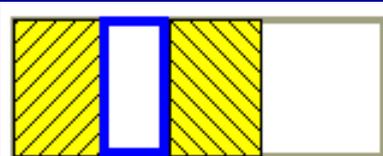
- Cette classe est un container qui utilise un **BoxLayout** pour ranger ses composants horizontalement ou verticalement
- Elle fournit des méthodes **static** pour obtenir des composants invisibles pour affiner la disposition de composants **dans un container quelconque** : *glue*, états et zones rigides

Classe **Box** ; composants invisibles



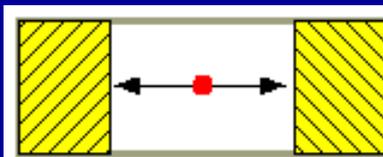
Sans composant invisible

```
container.add(firstComponent);  
// On ajoute ici un composant invisible  
container.add(. . .);  
container.add(secondComponent);
```



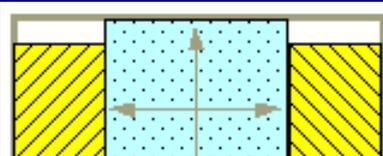
with rigid area

```
Box.createRigidArea(new Dimension(5,0))
```



with horizontal glue

```
Box.createHorizontalGlue()
```



with a custom filler

```
new Box.Filler(  
    new Dimension(5,100),  
    new Dimension(5,100),  
    new Dimension(Short.MAX_VALUE,100))
```

Code avec Box

```
// Le plus simple est d'utiliser une Box
// mais on peut aussi mettre un BoxLayout
// pour gérer un autre container
Box b = Box.createVerticalBox();
b.add(bouton);
// On peut ajouter des zones invisibles
// entre les composants :
b.add(Box.createVerticalStrut(5));
// ou b.add(Box.createRigidArea(
//           new Dimension(5, 15))
b.add(zoneSaisie);
```

CardLayout

- **CardLayout** : affiche un seul composant à la fois ; les composants sont affichés à tour de rôle
- Ce *layout manager* est plus rarement utilisé que les autres
- **JTabbedPane** est un composant qui offre le même type de disposition, en plus simple mais plus puissant, avec des onglets

Exemples et contre-exemples

First name: Last name:
Street: City:
Phone:

First name: Last name:
Street: City:
Phone:

Bon

Mauvais

First name: Last name:
Street: City:
Phone:

First name: Last name:
Street: City:
Phone:

First name: Last name:
Street: City:
Phone:

Exercice : donner 2 solutions

- avec des composants emboîtés
- avec un **GridBagLayout**

La solution Java pour traiter les événements : les écouteurs

Exposition du problème

- L'utilisateur utilise le clavier et la souris pour intervenir sur le déroulement du programme
- Le système d'exploitation engendre des **événements** à partir des actions de l'utilisateur
- Le programme doit lier des traitements à ces événements

Types d'événements

- Événements de **bas niveau**, générés directement par des actions élémentaires de l'utilisateur
- Événements « logiques » de plus **haut niveau**, engendrés par plusieurs actions élémentaires, qui correspondent à une action "logique" de l'utilisateur

Exemples d'événements

- De bas niveau :
 - appui sur un bouton de souris ou une touche du clavier
 - relâchement du bouton de souris ou de la touche
 - déplacer le pointeur de souris
- Logiques :
 - frappe d'un A majuscule
 - clic de souris
 - lancer une action (clic sur un bouton par exemple)
 - choisir un élément dans une liste
 - modifier le texte d'une zone de saisie

Événements engendrés

- La frappe d'un A majuscule engendre 5 événements :
- 4 événements de bas niveau :
 - appui sur la touche *Majuscule*
 - appui sur la touche A
 - relâchement de la touche A
 - relâchement de la touche *Majuscule*
- 1 événement logique :
 - frappe du caractère « A » majuscule

Classes d'événements

- Les événements sont représentés par des instances de sous-classes de `java.util.EventObject`
- Événements liés directement aux actions de l'utilisateur :
KeyEvent, MouseEvent
- Événements de haut niveau :
FocusEvent, WindowEvent, ActionEvent, ItemEvent, ComponentEvent

fenêtre ouverte,
fermée, icônifiée
ou désicônifiée

choix dans une liste,
dans une boîte à cocher

composant
déplacé, retailé,
caché ou montré

déclenchent
une action

Écouteurs

- Chacun des composants graphiques a ses **observateurs** (ou écouteurs, *listeners*)
- Un écouteur doit s'enregistrer auprès des composants qu'il souhaite écouter, en lui indiquant le type d'événement qui l'intéresse (par exemple, clic de souris)
- Il peut ensuite se désenregistrer

Relation écouteurs - écoutés

- Un composant peut avoir plusieurs écouteurs (par exemple, 2 écouteurs pour les clics de souris et un autre pour les frappes de touches du clavier)
- Un écouteur peut écouter plusieurs composants

Événements étudiés dans ce cours

- En exemple, ce cours étudie principalement
 - les événements **ActionEvent** qui conduisent à des traitements simples (écouteur **ActionListener**)
 - les événements **KeyEvent**, au traitement plus complexe (écouteur **KeyListener** et adaptateur **KeyAdapter**)

ActionEvent

- Cette classe décrit des événements de haut niveau très utilisés qui correspondent à un type d'action de l'utilisateur qui va le plus souvent déclencher un traitement (une *action*) :
 - clic sur un bouton
 - *return* dans une zone de saisie de texte
 - choix dans un menu
- Ces événements sont très fréquemment utilisés et ils sont très simples à traiter

Interface **ActionListener**

- Un objet *ecouteur* intéressé par les événements de type « action » (classe **ActionEvent**) doit appartenir à une classe qui implémente l'interface `java.awt.event.ActionListener`

- Définition de **ActionListener** :

```
public interface ActionListener  
    extends EventListener {  
    void actionPerformed(ActionEvent e);  
}
```

interface vide
qui sert de
marqueur pour
tous les
écouteurs

message qui sera envoyé
à l'écouteur (méthode **callback**)

Inscription comme **ActionListener**

- On inscrit un tel écouteur auprès d'un composant nommé *composant* par la méthode `composant.addActionListener(ecouteur) ;`
- On précise ainsi que *ecouteur* est intéressé par les événements **ActionEvent** engendrés par *composant*

Message déclenché par un événement

- Un événement *unActionEvent* engendré par une action de l'utilisateur sur *bouton*, provoquera l'envoi d'un message **actionPerformed** à tous les écouteurs :

```
ecouteur.actionPerformed(unActionEvent) ;
```

- Ces messages sont envoyés **automatiquement** à tous les écouteurs qui se sont enregistrés auprès du bouton

Interface **Action**

- Cette interface hérite de **ActionListener**
- Elle permet de partager des informations et des comportements communs à plusieurs composants
- Par exemple, un bouton, un choix de menu et une icône d'une barre de menu peuvent faire quitter une application
- Elle est étudiée à la fin de cette partie du cours

Conventions de nommage

- Si un composant graphique peut engendrer des événements de type **TrucEvent** sa classe (ou une de ses classes ancêtres) déclare les méthodes `{add|remove}TrucListener()`
- L'interface écouteur s'appellera **TrucListener**

Écouteur `MouseListener`

- Des interfaces d'écouteurs peuvent avoir de nombreuses méthodes
- Par exemple, les méthodes déclarées par l'interface `MouseListener` sont :

```
void mouseClicked(MouseEvent e)
void mouseEntered(MouseEvent e)
void mouseExited(MouseEvent e)
void mousePressed(MouseEvent e)
void mouseReleased(MouseEvent e)
```

Adaptateurs

- Pour éviter au programmeur d'avoir à implanter **toutes** les méthodes d'une interface « écouteur », AWT fournit des classes (on les appelle des adaptateurs), qui implantent toutes ces méthodes
- Le code des méthodes ne fait rien
- Ça permet au programmeur de ne redéfinir dans une sous-classe que les méthodes qui l'intéressent

Exemples d'adaptateurs

- Les classes suivantes du paquetage `java.awt.event` sont des adaptateurs :
`KeyAdapter`, `MouseAdapter`,
`MouseMotionAdapter`, `FocusAdapter`,
`ComponentAdapter`, `WindowAdapter`

Fermer une fenêtre

- Pour terminer l'application à la fermeture de la fenêtre, on ajoute dans le constructeur de la fenêtre un écouteur :

```
public Fenetre() { // constructeur
```

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

Classe anonyme interne
pour décrire l'écouteur

ne pas confondre avec
windowClosed()
appelée quand les ressources
système de la fenêtre sont
libérées (méthode **dispose()**)

Fermer une fenêtre

- Depuis JDK 1.3, on peut se passer d'écouteur pour arrêter l'application à la fermeture d'une fenêtre :

```
public Fenetre() { // constructeur
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    . . .
}
```

- Autres actions possibles à la fermeture d'une fenêtre (constantes de l'interface `WindowConstants`, pas de la classe `JFrame`) :

`DO_NOTHING_ON_CLOSE`, `HIDE_ON_CLOSE`, `DISPOSE_ON_CLOSE`

Paquetage `java.awt.event`

- Ce paquetage comprend les interfaces « écouteurs » et les classes d'événements

Écriture d'un écouteur

Classe `EventObject`

- Le paramètre passé à une méthode d'un écouteur décrit l'événement qui a provoqué l'appel de la méthode
- La méthode « `public Object getSource ()` » de la classe `EventObject` permet d'obtenir l'objet d'où est parti l'événement, par exemple un bouton ou un champ de saisie de texte
- Cette méthode est utile si l'écouteur écoute plusieurs composants

Exemple de code d'un écouteur

- Le code suivant modifie le texte du bouton sur lequel l'utilisateur a cliqué :

```
class EcouteurBouton extends MouseAdapter {  
    public void mousePressed(MouseEvent e) {  
        ((JButton)e.getSource()).  
            setText("Appuyé");  
    }  
    public void mouseClicked(MouseEvent e) {  
        ((JButton)e.getSource()).  
            setText(e.getClickCount() + " clics");  
    }  
}
```

Tester les touches modificatrices

- Pour tester si l'utilisateur a appuyé sur les touches Maj, Ctrl, Alt pendant qu'il cliquait sur la souris ou qu'il appuyait sur une autre touche, on utilise les constantes `static` de type `int` suivantes de la classe `InputEvent` :
`SHIFT_MASK` , `CTRL_MASK`, `ALT_MASK`, `META_MASK`,
`ALT_GRAPH_MASK`
- D'autres constantes pour tester le bouton de la souris :
`BUTTON1_MASK`, `BUTTON2_MASK`, `BUTTON3_MASK`
- Exemple d'utilisation :

```
if ((e.getModifiers() & SHIFT_MASK) == 0)
```
- Pour les boutons de la souris, on peut aussi utiliser des méthodes `static` de la classe `SwingUtilities` telles que `isLeftMouseButton`

Classe de l'écouteur

- Soit **C** la classe du container graphique qui contient le composant graphique
- Plusieurs solutions pour choisir la classe de l'écouteur de ce composant graphique :
 - classe **C**
 - autre classe spécifiquement créée pour écouter le composant :
 - classe externe à la classe **C** (rare)
 - classe interne de la classe **C**
 - classe interne anonyme de la classe **C** (fréquent)

Solution 1 : classe C

- Solution simple
- Mais peu extensible :
 - si les composants sont nombreux, la classe devient vite très encombrée
 - de plus, les méthodes « *callback* » comporteront alors de nombreux embranchements pour distinguer les cas des nombreux composants écoutés

Exemple solution 1

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

public class Fenetre extends JFrame
    implements ActionListener {
    private JButton b1 = new JButton("..."),
                b2 = new JButton("...");
    private JTextField f = new JTextField(10);

    public Fenetre() {
        . . . // ajout des composants dans la fenêtre
        // Fenetre écoute les composants
        b1.addActionListener(this);
        b2.addActionListener(this);
        f.addActionListener(this);
        . . .
    }
}
```

Exemple (suite)

```
public void actionPerformed(ActionEvent e) {  
    Object source = e.getSource();  
    if (source == b1) {  
        . . . // action liée au bouton b1  
    }  
    else if (source == b2) {  
        . . . // action liée au bouton b2  
    }  
    else if (source == f) {  
        . . . // action liée au JTextField f  
    }  
    . . .  
}
```

Suppose que
l'on est dans
la portée de **b1**
qui contient une
référence au
bouton

Risque de nombreux
if ... else

Variante de la solution 1

- Quand les composants à écouter sont très nombreux, on peut regrouper dans le code les traitements par types de composants

Exemple variante solution 1

```
public void actionPerformed(ActionEvent e) {
    Object source = e.getSource();
    String classeSource = source.getClass().getName();
    if (classeSource.equals("JButton")) {
        if (source == b1) {
            . . . // action liée au bouton b1
        }
        else if (source == b2) {
            . . . // action liée au bouton b2
        }
    }
    else if (classeSource.equals("JTextField")) {
        . . . // action liée au JTextField f
    }
}
```

Regroupement
optionnel

Remarque sur les boutons

- On peut associer à chaque bouton (et choix de menus) une chaîne de caractères par `bouton.setActionCommand("chaîne");`
- Cette chaîne
 - par défaut, est le texte affiché sur le bouton
 - permet d'identifier un bouton ou un choix de menu, indépendamment du texte affiché
 - peut être modifiée suivant l'état du bouton
 - peut être récupérée par la méthode `getActionCommand()` de la classe `ActionEvent`

Solution 2 : classe interne

- Solution plus extensible : chaque composant (ou chaque type ou groupe de composants) a sa propre classe écouteur
- Le plus souvent, la classe écouteur est une classe interne de la classe **C**

Exemple

```
public class Fenetre extends JFrame {
    private JButton b1, b2;
    . . .
    public Fenetre() {
        . . .
        ActionListener eb = new EcouteurBouton();
        b1.addActionListener(eb);
        b1.setActionCommand("b1");
        b2.addActionListener(eb);
        b2.setActionCommand("b2");
        . . .
    }
}
```

Exemple

```
// Classe interne de Fenetre
class EcouteurBouton implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String commande = e.getActionCommand();
        if (commande.equals("b1")) {
            . . . // action liée au bouton b1
        }
        else if (commande.equals("b2")) {
            . . . // action liée au bouton b2
        }
    }
}
. . .
```

Exemple d'utilisation de
getActionCommand()

Solution 3 : classe interne anonyme

- Si la classe écoute un seul composant et ne comporte pas de méthodes trop longues, la classe est le plus souvent une classe interne anonyme
- L'intérêt est que le code de l'écouteur est proche du code du composant
- Rappel : une classe interne locale peut utiliser les variables locales et les paramètres de la méthode, **seulement s'ils sont `final`**

Exemple

```
 JButton ok = new JButton("OK");  
 ok.addActionListener(new ActionListener() {  
     public void actionPerformed(ActionEvent e) {  
         // action à exécuter  
         . . .  
     }  
 });
```

Événements clavier

Événements clavier

- Un événement clavier de **bas niveau** est engendré par une action élémentaire de l'utilisateur, par exemple, appuyer sur une touche du clavier (génère un appel à la méthode **keyPressed()** de **KeyListener**)
- Plusieurs actions élémentaires de l'utilisateur peuvent engendrer un seul événement de **haut niveau** ; ainsi, appuyer et relâcher sur les touches Shift et A pour obtenir un A majuscule, génère un appel à la méthode **keyTyped()** de **KeyListener**

KeyEvent

- Cette classe correspond aux événements (*evt*) engendrés par l'utilisation du clavier
- 3 types d'événements repérés par `evt.getID()` :
 - `KeyEvent.KEY_PRESSED` et `KeyEvent.KEY_RELEASED` sont des événements de bas niveau et correspondent à une action sur une seule touche du clavier
 - `KeyEvent.KEY_TYPED` est un événement de haut niveau qui correspond à l'entrée d'un caractère Unicode (peut correspondre à une combinaison de touches comme Shift-A pour A)

KeyEvent

- Si on veut repérer la saisie d'un caractère Unicode, il est plus simple d'utiliser les événements de type **KEY_TYPED**
- Pour les autres touches qui ne renvoient pas de caractères Unicode, telle la touche F1 ou les flèches, il faut utiliser les événements **KEY_PRESSED** ou **KEY_RELEASED**

KeyListener

```
public interface KeyListener
    extends EventListener {
    void keyPressed(KeyEvent e);
    void keyReleased(KeyEvent e);
    void keyTyped(KeyEvent e);
}
```

- Si on n'est intéressé que par une des méthodes, on peut hériter de la classe **KeyAdapter**
- Dans ces méthodes, on peut utiliser les méthodes **getKeyChar()** (dans **keyTyped**) et **getKeyCode()** (dans les 2 autres méthodes)

Exemple de `KeyListener`

```
class EcouteCaracteres extends KeyAdapter {
    public void keyTyped(KeyEvent e) {
        if (e.getKeyChar() == 'q')
            quitter();
    }
    public void keyReleased(KeyEvent e) {
        if (e.getKeyCode() == KeyEvent.VK_ESCAPE)
            actionQuandEsc();
    }
}
```

Focus

- La gestion du *focus* a radicalement changé à partir du SDK 1.4
- Ce cours s'appuie sur la version 1.4

Avoir le *focus*

- Un seul composant peut « avoir le *focus* » à un moment donné
- C'est le composant qui va recevoir tous les caractères tapés au clavier par l'utilisateur
- La fenêtre qui « a le *focus* » est celle qui contient ce composant
- Un composant ne peut engendrer un **KeyEvent** que s'il a le *focus*

Cycles pour obtenir le *focus*

- Tous les composants ont le *focus* à tour de rôle
- Par exemple, quand l'utilisateur tape [Enter] dans un champ de saisie, le composant « suivant » obtient le *focus*
- Des touches spéciales permettent aussi de passer d'un composant à l'autre ([Tab] et [Maj]-[Tab] sous Windows et sous Unix)
- L'ordre est le plus souvent déterminé par la position sur l'écran des composants
- Il peut aussi être fixé par programmation

Obtenir le *focus*

- 2 autres façons pour obtenir le *focus*
- Le plus souvent un composant obtient le *focus* quand l'utilisateur clique sur lui
- Il peut aussi l'obtenir par programmation avec la méthode **boolean requestFocusInWindow()** de la classe **Component** (meilleur que **requestFocus** du SDK 1.3)

Obtenir le *focus*

- Tous les composants ne peuvent avoir le *focus*
- En général les zones de saisie de texte peuvent l'avoir mais pas les boutons ou les labels
- Dans la classe **Component**
 - la méthode **boolean isFocusable()** permet de savoir si un composant peut l'avoir (**isFocusTraversal** en SDK 1.3)
 - **void setFocusable(boolean)** permet de modifier cette propriété

Méthodes diverses

- Dans la classe **Component**,
 - 2 méthodes pour passer au composant suivant ou précédent : **transfertFocus** et **transfertFocusBackward**
 - des méthodes pour gérer les cycles de passage de *focus*
- Dans la classe **Window**, beaucoup de méthodes diverses à utiliser si on veut gérer le *focus* au niveau des fenêtres

Focus et **KeyListener**

- Si votre **KeyListener** ne réagit pas, demandez vous si le composant qui est écouté a bien le *focus*
- Vous pouvez utiliser pour cela la méthode **boolean isFocusOwner()** de la classe **Component** (**hasFocus()** du SDK 1.3 est obsolète)

FocusListener

- Écouteur pour savoir quand le *focus* change de main
- Méthodes `focusGained` et `focusLost`

Mécanismes internes pour traiter les événements

Etapes du traitement

- Les événements sont
 1. mis dans la file d'attente des événements
 2. récupérés un à un par un *thread* de distribution des événements
 3. redistribués par le *thread* aux composants concernés
 4. traités par les écouteurs concernés du composant (enregistrés auprès du composant)
 5. traités par le composant pour son propre compte

File d'attente des événements

- Les événements sont engendrés le plus souvent par une action de l'utilisateur perçue par le système de fenêtrage natif sous-jacent
- Les événements sont mis automatiquement par le système AWT dans la file d'attente des événements (instance unique de la classe **EventQueue**)

Thread de distribution des événements

- Un *thread* **unique** (*Event Dispatch Thread*)
 1. récupère un à un les événements dans la file d'attente des événements
 2. passe la main au composant concerné (appel de **dispatchEvent (AWTEvent)** de la classe **Component**)
- Ce *thread* est créé automatiquement dès que l'on utilise une classe de AWT

Traitement des événements

- Le composant fait appel à `processEvent(e)` (classe `Component`) qui fait appel à la méthode appropriée :
 - `processMouseEvent(MouseEvent)` pour les événements liés à la souris
 - `processFocusEvent(FocusEvent)` pour les événements liés au focus
 - `processKeyEvent(KeyEvent)` pour ceux liés à la frappe de touches du clavier
 - etc.

Appel des écouteurs

- Le composant lance la méthode « *callback* » des écouteurs correspondant au type de l'événement (**MouseClicked**, **KeyReleased**, ...)

Consommation d'un **InputEvent**

- Après avoir été traité par tous les écouteurs, l'événement est traité par le composant lui-même
- Par exemple, un caractère tapé par l'utilisateur sera affiché dans un **JTextField**
- Un écouteur d'un **InputEvent** (**Mouse** et **KeyEvent**) peut empêcher cette dernière étape en consommant l'événement (méthode **consume ()** de la classe **InputEvent**)

Génération d'un événement

- On peut créer un événement par du code, non provoqué par une action de l'utilisateur, pour l'envoyer directement à un certain composant
- On le fait en 2 étapes :
 - créer l'événement
 - envoyer un message « **dispatchEvent** » au composant, avec l'événement créé en paramètre
- Remarque : les événements de haut niveau tels que **ActionEvent** ne peuvent être simulés de cette façon

Exemple de génération d'un événement

- Dans un écouteur de menu (un `ActionListener` qui est ici la fenêtre qui contient le menu), le code suivant simule la fermeture de la fenêtre par l'utilisateur (et sans doute la sortie de l'application) :

```
if (e.getActionCommand().equals("sortie"))  
    dispatchEvent(new WindowEvent(  
        this, WindowEvent.WINDOW_CLOSING));
```

Swing et *threads*

« *Event dispatch thread* »

- La récupération des événements dans la file d'attente, le traitement de ces événements et l'affichage de l'interface graphique est effectué par Swing dans un seul et même *thread* appelé le *thread de distribution des événements* (*event dispatcher*)
- Tout traitement long effectué par ce *thread* va donc figer l'interface graphique qui ne répondra plus aux actions de l'utilisateur

Traitements longs effectués par les écouteurs

- Il faut donc effectuer les traitements longs des écouteurs et des méthodes `paintComponent` dans des tâches à part
- Sinon, l'interface graphique répondra mal, ou même pas du tout, aux événements engendrées par les actions de l'utilisateur

Exemple de traitement à lancer dans un *thread* à part

- Accéder à une base de données pour récupérer les éléments affichés par une liste déroulante
- Calcul complexe de la position d'un objet graphique

Swing n'est pas « *thread-safe* »

- Pour des raisons de performance et de facilité de mise en œuvre par les programmeurs, les composants de Swing ne sont pas prévus pour être utilisés par plusieurs tâches en même temps
- Si un composant a déjà été affiché, tout code qui modifie l'état (le modèle) du composant doit être exécuté dans le *thread* de distribution des événements

Utiliser d'autres *threads* qui modifient l'interface graphique

- 2 méthodes utilitaires `public static` de la classe `javax.swing.SwingUtilities` permettent de lancer des opérations qui modifient des composants de l'interface graphique depuis un autre *thread* que le *event dispatcher* :
 - `void invokeLater(Runnable runnable)`
 - `void invokeAndWait(Runnable runnable)`

Utiliser d'autres *threads* ...

- **invokeLater** permet de déposer une tâche à accomplir dans la file d'attente des événements
- Le *thread dispatcher* exécutera cette tâche comme tous les traitements d'événements, sans occasionner de problèmes de multitâche
- Avec **invokeAndWait**, on attend que la tâche soit exécutée par le *event dispatcher* avant de continuer

Utilisation de `invokeLater()`

- Le schéma est le suivant si on a un traitement long à effectuer, qui a une interaction avec l'interface graphique :
 - on lance un *thread* qui effectue la plus grande partie de la tâche, par exemple, accès à une base de donnée et récupération des données
 - au moment de modifier l'interface graphique, ce *thread* appelle `invokeLater()` en lui passant un **Runnable** qui exécutera les instructions qui accèdent ou modifient l'état du composant

Utilisation de `invokeLater()`

```
class AfficheurListe extends Thread {
    private Runnable modifieurListe;
    private Collection donneesListe;
    AfficheurListe(List l) {
        modifieurListe = new Runnable() {
            public void run() {
                . . .
            }
        }
    }
    public void run() {
        // Remplit donneesListe avec les données
        // lues dans la base de données
        . . .
        SwingUtilities.invokeLater(modifieurListe);
    }
}
```

Modifie la liste en utilisant `donneesListe`

invokeAndWait

- La différence avec `invokeLater` est que l'instruction qui suit l'appel de `invokeAndWait` n'est lancée que lorsque la tâche est terminée
- On peut ainsi lancer un traitement et tenir compte des résultats pour la suite du traitement long
- **Attention**, `invokeAndWait` provoquera un blocage s'il est appelé depuis le *event dispatch thread* ; ça pourrait arriver si on veut partager du code entre un écouteur et une méthode ; pour éviter ce problème, utiliser la méthode `static EventQueue.isDispatchThread()`

Classe `SwingWorker`

- Le tutoriel de *Sun* offre la classe `SwingWorker` pour faciliter l'utilisation des méthodes `invokeLater()` et `invokeAndWait()`

Architecture des applications avec interface graphique

Séparation du GUI et des classes métier

- Les classes métier doivent être indépendantes des interfaces graphiques qui les utilisent
- L'architecture est la suivante :
 - classes métier
 - classes pour l'interface graphique dont les écouteurs font appel aux méthodes des classes métier

Classe « principale » schématique

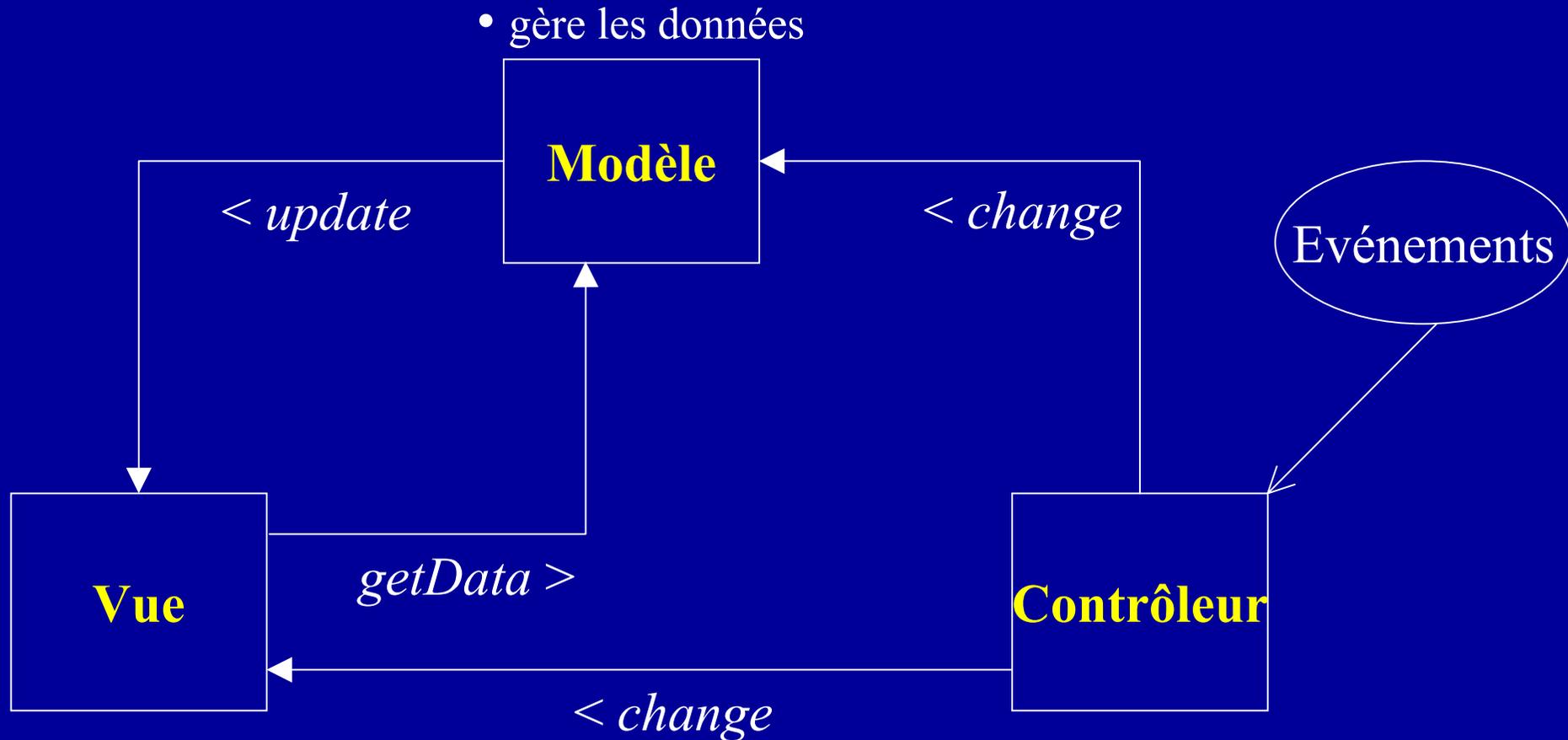
```
public class Application {  
    . . .  
    public static void main(String[] args) {  
        // Initialisation (utilise classes métier)  
        . . .  
        // Appelle la classe GUI principale  
        new GUI(. . .);  
        // Fermeture (utilise classes métier)  
        . . .  
    }  
}
```

Pattern MVC en Swing

Architecture MVC

- L'architecture MVC (Modèle-Vue-Contrôleur) est utilisée par Swing pour modéliser les composants graphiques qui contiennent des données (listes, tables, arbres,...) :
 - le **modèle** contient les données
 - les **vues** donnent une vue des données (ou d'une partie des données) du modèle à l'utilisateur
 - le **contrôleur** traite les événements reçus par le composant graphique

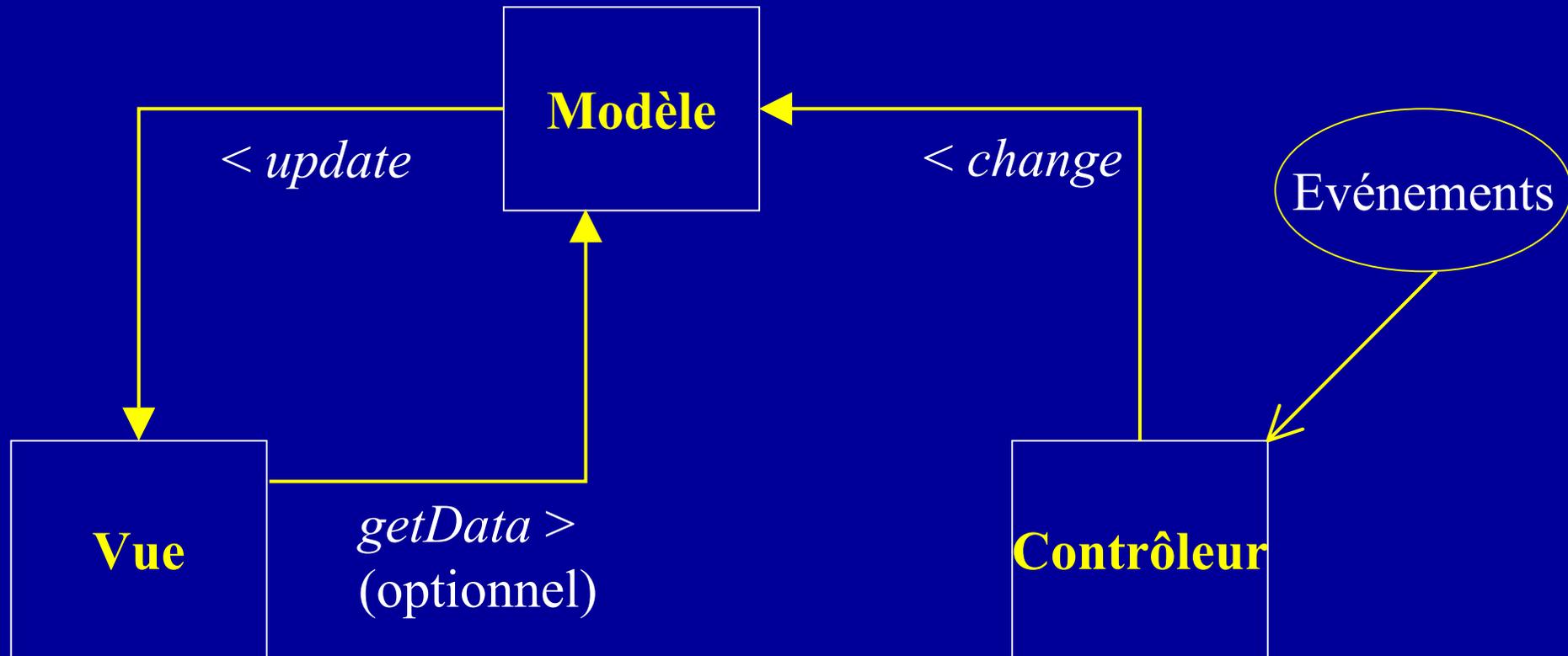
Architecture MVC



- donne une vue d'une partie des données
- observe le modèle

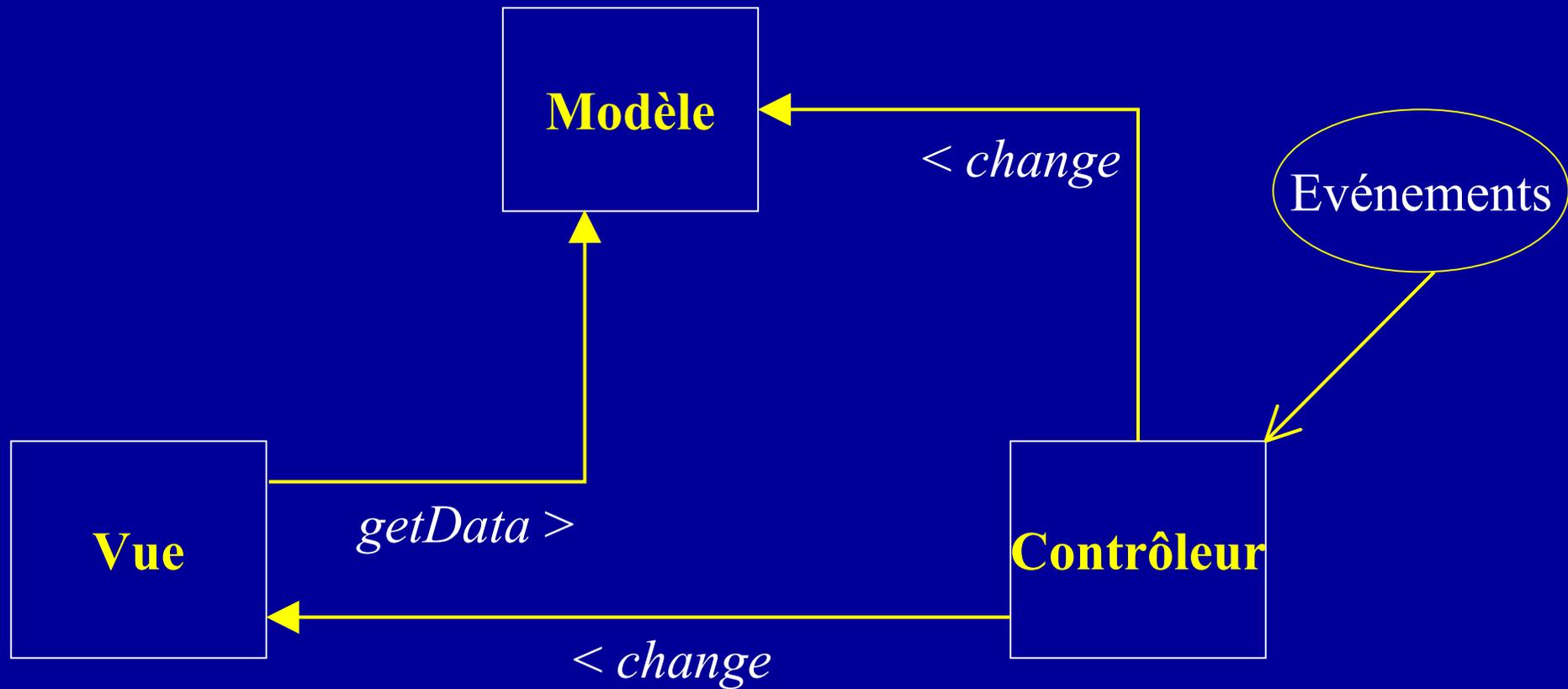
- traite les événements

Architecture MVC



Les vues écoutent le modèle

Variante architecture MVC



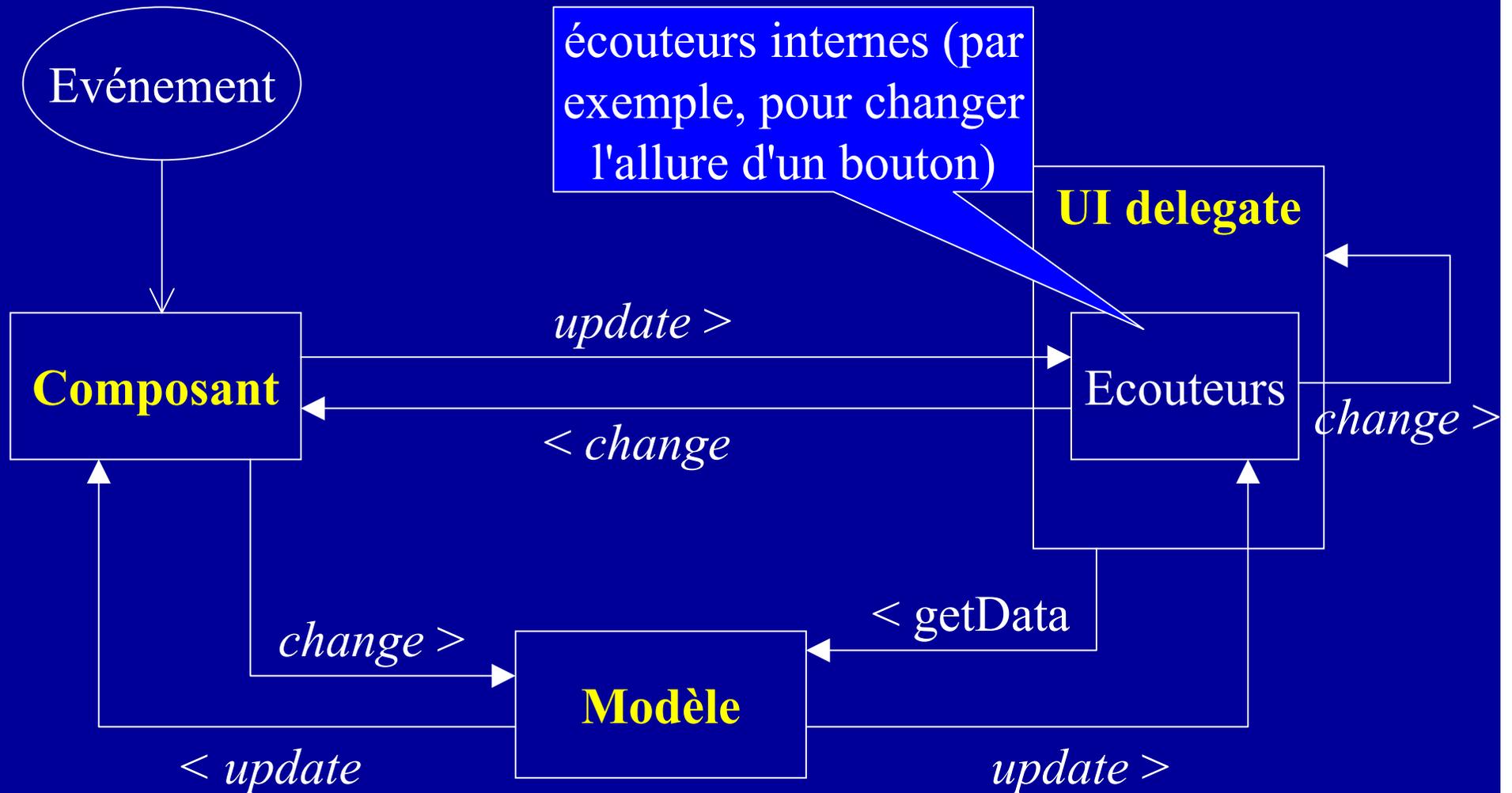
Exemple de processus engendré par une action de l'utilisateur

1. Le contrôleur reçoit un événement
2. Il informe le modèle (*change*)
3. Celui-ci modifie ses données en conséquence
4. Le contrôleur informe la vue d'un changement (*change*)
5. La vue demande au modèle les nouvelles données (*getData*)
6. La vue modifie son aspect visuel en conséquence

Architecture avec UI-delegate

- Pour représenter ses composants graphiques, Swing utilise une variante de MVC où contrôleur et vue sont réunis dans un objet *UI-delegate* qui donne le *look and feel* du composant :
 - le **modèle** contient les données
 - les **UI-delegates** représentent une façon de voir ces données et de traiter les événements
 - le **composant** est l'interlocuteur pour les objets extérieurs et il contient le code pour le comportement de base du composant

Architecture avec *UI delegate*



« *Pluggable Look and feel* »

- L'architecture avec UI-delegate permet de dissocier complètement le *look and feel* du composant
- Il est ainsi possible en Swing de changer de *look and feel* pour l'ensemble des composants d'une interface graphique :

```
UIManager.setLookAndFeel (  
    "javax.swing.plaf.metal.MetalLookAndFeel") ;  
// Change pour les composants déjà affichés  
SwingUtilities.updateComponentTreeUI (  
    fenetre) ;
```

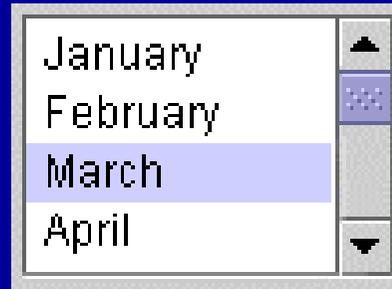
Utilisation explicite du modèle

- Le modèle des composants n'est pas toujours utilisé explicitement dans le code
- Il est souvent caché par les implémentations utilisées dans le cas les plus simples (boutons par exemple)
- Dans le cas des listes, l'utilisation explicite du modèle n'est indispensable que lorsque la liste est modifiable

Un exemple d'utilisation des modèles : les listes

Listes

- Une liste permet de présenter une liste de choix à l'utilisateur :



- Celui-ci peut cliquer sur un ou plusieurs choix

Barre de défilement pour les listes

- Le plus souvent une liste a des barres de défilement ; pour cela, il faut insérer la liste dans un `ScrollPane` :

```
JScrollPane sList =  
    new JScrollPane (uneListe) ;
```

- Par défaut, 8 éléments de la liste sont visibles ; on peut modifier ce nombre :

```
uneListe.setVisibleRowCount (5) ;
```

Mode de sélection

- L'utilisateur peut sélectionner un ou plusieurs éléments de la liste suivant le mode de sélection de la liste :
 - `SINGLE_SELECTION`
 - `SINGLE_INTERVAL_SELECTION`
 - `MULTIPLE_INTERVAL_SELECTION` (mode par défaut)

```
uneListe.setSelectionMode(  
    ListSelectionMode.SINGLE_SELECTION);
```

Constructeurs des listes

- 2 constructeurs pour des listes **non modifiables** :

```
public JList(Object[] listData)  
public JList(Vector listData)
```

- Une **liste modifiable** est associée à un modèle qui fournit les données affichées par la liste :

```
public JList(ListModel dataModel)
```

Afficher les éléments des listes

- Une liste affiche ses éléments en utilisant `toString()` (elle sait aussi afficher des instances de `ImageIcon`)
- On peut aussi programmer des affichages particuliers avec un « *renderer* » (`setCellRenderer(ListCellRenderer)`)

Exemple de liste non modifiable

```
JList liste = new JList(new String[]
    {"Un", "Deux", "Trois", "Quatre", ...});
JScrollPane sp = new JScrollPane(liste);
liste.setSelectionMode(
    ListSelectionMode.SINGLE_SELECTION);
// Pour cet exemple, la classe est l'écouteur
liste.addListSelectionListener(this);
```

Utilisation standard d'une liste

- Il est rare d'écrire un écouteur de liste
- L'utilisation standard d'une liste est de demander à l'utilisateur de cliquer sur un bouton lorsqu'il a fini de faire ses choix dans la liste
- On récupère alors la sélection de l'utilisateur par une des méthodes **getSelectedValue()** ou **getSelectedValues()** dans la méthode **actionPerformed()** de l'écouteur du bouton

Écouteur d'une liste

- La classe d'un écouteur de liste doit implémenter l'interface **ListSelectionListener** qui contient la méthode

```
void valueChanged(ListSelectionEvent e)
```

- Attention, une nouvelle sélection engendre 2 événements :
 - un pour désélectionner la précédente sélection
 - l'autre pour informer de la nouvelle sélection
 - **getValueIsAdjusting()** renvoie vrai pour le premier événement

Exemple d'écouteur d'une liste

```
public class EcouteurListe
    implements ListSelectionListener
public void valueChanged(ListSelectionEvent e) {
    if (e.getValueIsAdjusting()) return;
    JList source = (JList)e.getSource();
    // getSelectedValue() si un seul choix possible
    Object[] choix = source.getSelectedValues();
    for (int i = 0; i < choix.length; i++) {
        // Ajoute les choix dans une zone de texte
        textArea.append(choix[i] + "\n");
    }
}
}
```

Listes modifiables

- Elles s'appuient sur un modèle qui « contient » les données : une classe qui implémente l'interface `ListModel`

Modèle de données pour une liste

```
public interface ListModel {  
    int getSize();  
    Object getElementAt(int i);  
    void addListDataListener(ListDataListener l);  
    void removeListDataListener(ListDataListener l);  
}
```

- Pour écrire une classe qui implémente **ListModel**, le plus simple est d'hériter de la classe abstraite **AbstractListModel** qui implémente les 2 méthodes de **ListModel** qui ajoutent et enlèvent les écouteurs

Listes modifiables simples

- Les plus simples ont un modèle de la classe `DefaultListModel` qui hérite de la classe `AbstractListModel`
- On peut gérer les données dans le modèle avec des méthodes semblables aux méthodes `add` et `remove` des collections

Exemple de liste modifiable simple

```
pays = new DefaultListModel();  
pays.addElement("France");  
pays.addElement("Italie");  
pays.addElement("Espagne");  
pays.addElement("Maroc");  
  
liste = new JList(pays);
```

Listes plus complexes

- Pour le modèle on doit créer une classe qui héritent de la classe **AbstractListModel**

Liste modifiable sans enregistrement physique des données

```
/** Les 1000 premiers entiers composent le
 * modèle de données de la liste */
class Entiers1000 extends AbstractListModel {
    public int getSize() {
        return 1000;
    }
    public Object getElementAt(int n) {
        return new Integer(n + 1);
    }
}
```

```
. . .
JList liste = new JList(new Entiers1000());
```

Dessiner

Classe `Graphics`

- L'affichage d'un `JComponent` est effectué par la méthode `paintComponent(Graphics g)`
- `g` contient le contexte graphique dans lequel se fait l'affichage
- Cette instance est passée en paramètre par le système graphique Java à la méthode `paintComponent()`

Ce que contient un **Graphics**

- « endroit » où afficher
- zone de « *clip* » (l'utilisateur peut restreindre la zone de travail)
- couleur de tracé
- fonte
- mode de tracé : mode « normal » ou mode « XOR » dans lequel les nouveaux tracés se tracent en changeant les pixels de la couleur de tracé actuelle en une autre couleur (couleur de XOR), et vice-versa
- couleur de XOR

Dessiner dans quel composant ?

- On peut dessiner dans tout **JComponent** mais il est recommandé de dessiner dans un **JPanel** (avec un **JComponent**, le fond d'écran risque donc de ne pas être repeint lorsqu'il le faut)
- Pour dessiner, on crée une classe qui hérite de **JPanel**, dans laquelle on redéfinit la méthode **paintComponent()**, avec le code du dessin

Écriture de la méthode `paintComponent()`

- La méthode `paintComponent(Graphics g)` doit avoir « `super.paintComponent(g)` » comme première instruction
- Cela permet à la méthode `paintComponent()` de la classe `JPanel` d'effectuer sa part du travail (en particulier peindre le fond d'écran)

Précaution à prendre quand on redéfinit `paintComponent()`

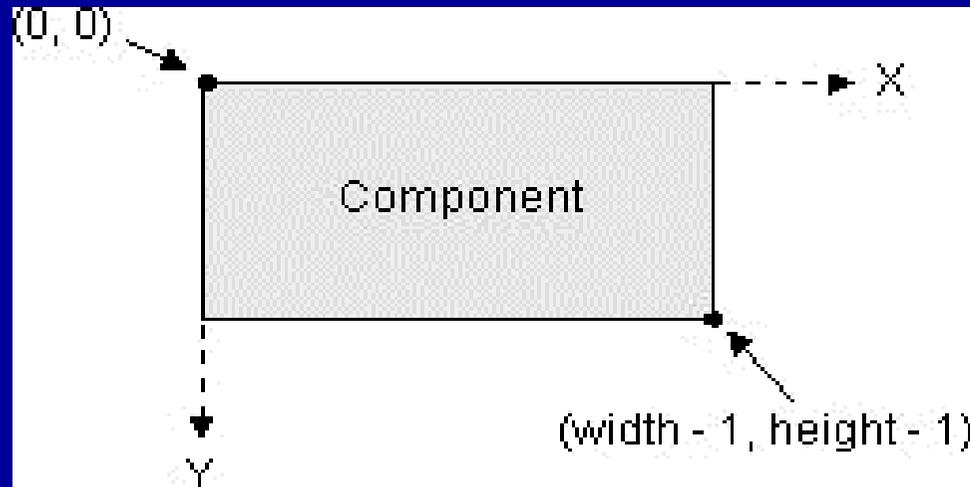
- Attention ! il est indispensable de préciser les tailles préférée et minimale du composant qui redéfinit la méthode `paintComponent()`, pour que le *layout manager* qui placera le composant puisse réserver un emplacement à la bonne taille pour le composant (`composant.setSize()` ne marche pas dans ce cas)
- On le fait en redéfinissant les méthodes `getPreferredSize()` et `getMinimalSize()`

Dessiner avec la classe **Graphics**

- **Graphics** offre plusieurs méthodes pour afficher divers éléments :
 - **drawString()** affiche du texte
 - **drawImage()** affiche une image
 - **drawLine()** affiche une ligne
 - **fillOval()** affiche une ellipse
 - **drawPolygon()** affiche un polygone
 - ...

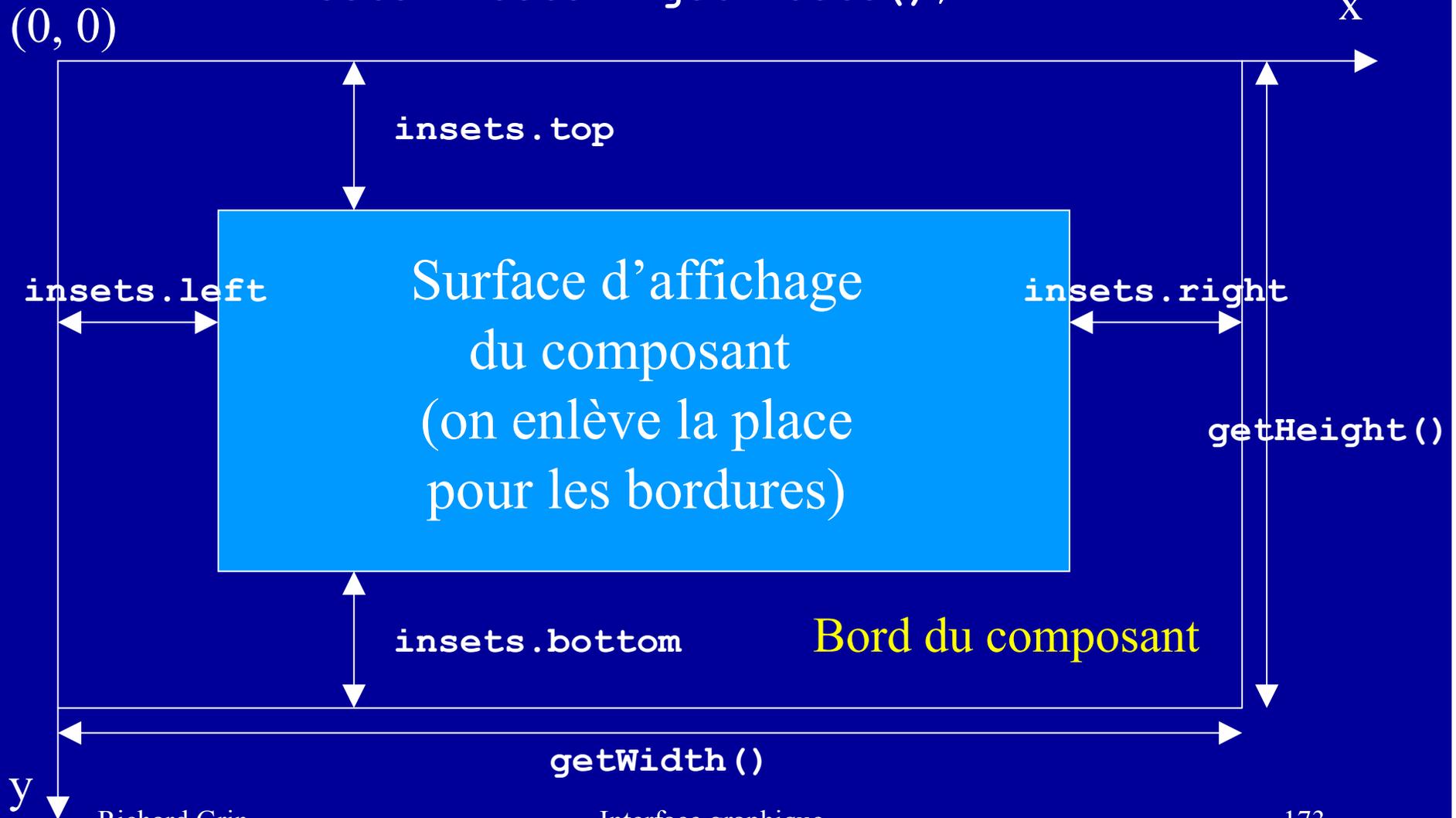
Systeme de coordonnees

- L'emplacement d'affichage est passé à ces méthodes par des coordonnées (x, y) ; l'unité est le pixel et l'origine est placée en haut à gauche du composant dans lequel on affiche



Surface disponible pour l'affichage

```
Insets insets = getInsets();
```



Surface disponible pour l'affichage

```
public void paintComponent(Graphics g) {  
    ...  
    Insets insets = getInsets();  
    int largeurDisponible =  
        getWidth() - insets.left - insets.right;  
    int hauteurDisponible =  
        getHeight() - insets.top - insets.bottom;  
    ...  
    /* Peindre dans le rectangle défini par  
       les points (insets.left, insets.height) et  
       (getWidth()-insets.right, getHeight()-insets.bottom)  
    */  
}
```

« Dessiner » du texte

```
public Fenetre() {  
    . . .  
    Container contentPane = getContentPane();  
    contentPane.add(new HelloWorldComponent());  
    . . .  
}
```

```
class HelloWorldComponent extends JPanel {  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        g.drawString("Hello world!", 75, 100);  
    }  
}
```

Ajouter `getPreferredSize` si nécessaire

Dessiner

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawRect(10, 50, 100, 80);
    Color couleur = g.getColor();
    g.setColor(Color.green);
    g.fillRect(20, 60, 50, 40);
    g.drawOval(5, 40, 110, 110);
    g.setColor(couleur);
    g.drawPolygon(
        new int[] {200, 210, 250, 280, 200},
        new int[] {110, 120, 60, 200, 300},
        5);
}
```

Restreindre la zone de traitement

- On peut restreindre la zone sur laquelle s'appliqueront les ordres que l'on envoie à un **Graphics**,
- Les méthodes **setClip** permettent de donner une zone rectangulaire, ou même une zone quelconque délimitée par une forme (**Shape**)
- Cette zone peut être obtenue par la méthode **getClip** (**getClipBounds** pour obtenir le rectangle circonscrit à la forme)

Classe `Graphics2D`

- En fait, la méthode `paintComponent` reçoit une instance de la classe `Graphics2D`, sous-classe de `Graphics`
- `Graphics2D` offre beaucoup plus de possibilités que `Graphics`
- Parmi les possibilités les plus simples, elle permet les rotations, les mises à l'échelle, le choix de la largeur de trait, le tracé de rectangle 3D

Classe `Graphics2D`

- Pour utiliser ces possibilités, il suffit de caster en `Graphics2D` le paramètre de

`paintComponent` :

```
Graphics2D g2 = (Graphics2D)g;
```

« Rendre » un Graphics

- Un Graphics est une ressource système qu'il faut rendre avec la méthode **dispose()** (classe **Graphics**) quand on n'en a plus besoin
- On ne doit rendre que les **Graphics** que l'on a obtenu par une méthode telle que **getGraphics** ou **createGraphics**
- Il faut laisser cette responsabilité aux méthodes appelantes si on travaille avec un **Graphics** que l'on a reçu en paramètre d'une méthode telle que **paint** ou **paintComponent**

Java2D

- Les possibilités de base pour dessiner fournies par le SDK sont étendues énormément par Java 2D apparue avec le SDK 1.2
- Java 2D apporte à la fois plus de souplesse et de nombreuses nouvelles fonctionnalités
- Java 2D utilise des classes de `java.awt`, `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.image` et `java.awt.print`
- Nous n'étudierons pas Java2D dans ce cours

Faire se redessiner un composant

- On envoie le message **repaint()** au composant (hérité de la classe **Component**)
- Pour les dessins complexes, on peut gagner en performance si la zone à redessiner est incluse dans un rectangle, en indiquant à la méthode **repaint()** le rectangle à redessiner :
repaint(x, y, largeur, hauteur)

revalidate, invalidate, validate

- **invalidate()** rend « non correct » le composant qui reçoit le message, et tous les composants qui le contiennent (le *layout manager* doit recalculer son emplacement)
- **validate()** envoyé à un **container**, lui indique qu'il doit réorganiser ses composants (il ne tient compte que des modifications des composants qui se déclarent « non corrects »)
- **revalidate()** envoyé à un composant, conjugue un **invalidate()** du composant et un **validate()** de son container

Couleurs et polices de caractères

- On peut utiliser les classes **Color** et **Font** pour améliorer la qualité de l'interface

Couleurs et polices de caractères

Exemple

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Font f = new Font("SansSerif", Font.BOLD, 14);
    Font fi = new Font("SansSerif",
                      Font.BOLD + Font.ITALIC, 14);
    g.setFont(f);
    g.setColor(Color.red);
    setBackground(Color.blue); // couleur de fond
    g.drawString("Hello world!", 75, 100);
    g.drawImage(image, 100, 75, null);
    g.setFont(fi);
    g.drawString("Hello world!", 75, 200);
}
```

Calcul de la taille d'un texte

- On peut avoir besoin de calculer la taille d'un texte en pixels, par exemple pour le centrer

- Centrer un message :

de la classe **Component**

```
FontMetrics fm = getFontMetrics(getFont());  
int hauteurTexte = fm.getHeight();  
int largeurTexte = fm.stringWidth(msg);  
g.drawString( msg,  
              (int) ((largeur - largeurTexte) / 2),  
              (int) ((hauteur - hauteurTexte) / 2));
```

- On a besoin d'un **Graphics** pour obtenir la taille d'un texte (ici celui du **Component**) car la taille d'un caractère dépend de l'épaisseur du trait pour le tracer

Taille utilisable d'un Graphics

- Il arrive qu'une méthode reçoive en paramètre une instance de **Graphics** qui provient d'un composant graphique extérieur à la classe de la méthode
- On peut récupérer le rectangle affiché par cette instance par la méthode **getClipBounds ()** :

```
public void dessine(Graphics g) {  
    Rectangle r = g.getClipBounds();  
    g.drawLine(0, 0, r.width, r.height);  
}
```

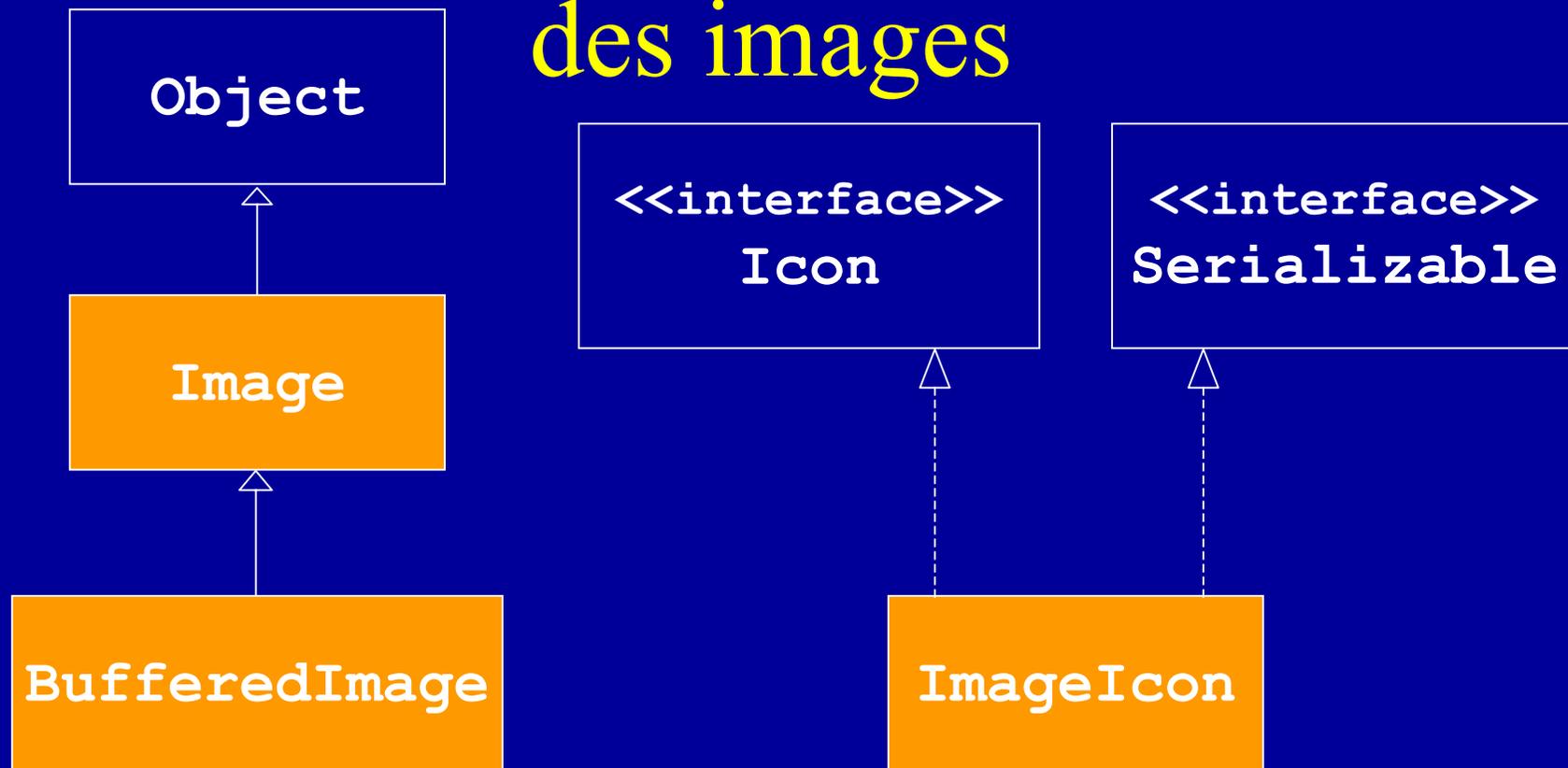
Trace une diagonale

Images

Images

- Les images sont des objets complexes (pixels, modèle de représentation des pixels, en particulier pour les couleurs)
- Elles sont représentées en Java par 3 classes
 - `java.awt.Image` : une image de base
 - `java.awt.image.BufferedImage` : permet de manipuler les pixels de l'image
 - `javax.swing.ImageIcon` : correspond à une image de taille fixe utilisée pour décorer un composant
- De base, Java sait travailler avec les images GIF, JPEG ou PNG, et les GIF animés

Arborescence d'héritage des images



La méthode `getImage ()` de la classe **ImageIcon** fournit une instance de **Image**

Un des (nombreux) constructeurs de **ImageIcon** prend une **Image** en paramètre

Quelle classe choisir ?

- Si on veut seulement afficher une image, sans la retoucher, le plus simple est d'utiliser **ImageIcon**
- Si l'on veut effectuer le chargement d'une image en tâche de fond, ou pouvoir modifier la taille de l'image, il faudra utiliser **Image**
- Si on veut faire des traitements sur une image, il est souvent nécessaire d'utiliser **BufferedImage**
- JAI (*Java Advanced Imaging*), autre API, facilite les manipulations complexes des images

Afficher une image

- Les classes d'images ne dérivent pas de la classe **JComponent** ; si on veut afficher une image, il faut l'inclure dans un composant
- En Swing, on peut l'inclure
 - dans un **JLabel** (le plus simple)
 - mais aussi dans un **JPanel** ou un **JComponent**
- Avec AWT, on inclut les images dans un **Canvas**

ImageIcon

Obtenir une `ImageIcon`

- Il existe des constructeurs de `ImageIcon` pour créer une instance à partir de :
 - un nom de fichier absolu ou relatif ; le séparateur est « / » quel que soit le système
 - un URL (adresse Internet ; `URL`)
 - une image (`Image`)
 - un tableau de byte (`byte []`)

Afficher une ImageIcon dans un JLabel

```
Icon icone =  
    new ImageIcon("images/image.gif");  
JLabel label = new JLabel(icone);  
JLabel label2 =  
    new JLabel("Texte label", icone);
```

Obtenir une `ImageIcon` depuis une *applet*

```
String nomFichier = "images/image.gif";  
try {  
    URL url = new URL(getCodeBase(),  
                      nomFichier);  
    ImageIcon imageIcon = new ImageIcon(url);  
}  
catch (MalformedURLException e) { . . . }
```

Obtenir une `ImageIcon` comme une ressource

- Au lieu de donner le lieu exact où trouver l'image, on peut déléguer la recherche du fichier qui contient l'image au chargeur de classes :

```
URL url = getClass().getResource(nomFichier);  
ImageIcon icone = new ImageIcon(url);
```

- Le chargeur de classe recherche l'image selon son algorithme de recherche, typiquement, dans le *classpath* si le chemin est absolue et à partir de la classe si le chemin est relatif ; il recherche aussi dans les fichiers jar

Image

Obtenir une **Image** dans une application

- ```
Image image = Toolkit.getDefaultToolkit()
 .getImage("uneImage.gif");
```
- Cette méthode lie seulement l'image à un fichier sur disque ou à un URL sur le réseau (le paramètre est de la classe `String` ou `URL`) ; l'image ne sera chargée que lorsqu'elle sera affichée
- **Attention**, aucun message d'erreur s'il n'y a pas d'image à l'emplacement indiqué !

# Obtenir une **Image** depuis une *applet*

- Une applet peut créer une image par les méthodes `Image getImage(URL url)` ou `Image getImage(URL base, String nom) :`

```
image = getImage(getCodeBase(),
ou getDocumentBase() "image1.gif")
```

- Si aucune politique de sécurité spécial n'a été installée, seules les images qui proviennent du même serveur Web que l'*applet* peuvent être manipulées

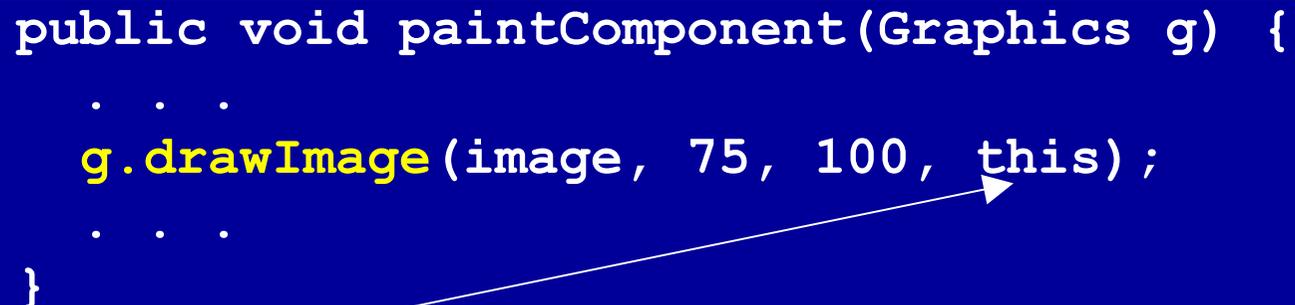
# Obtenir une image comme une ressource

- Même principe que pour une **ImageIcon** (utiliser Toolkit pour récupérer l'image avec l'URL)
- C'est la façon la plus souple de récupérer une image (ou une **ImageIcon**)

# Afficher une **Image** dans un **JComponent** ou un **JPanel**

- Il faut redéfinir la méthode `paintComponent` dans une sous-classe de `JComponent` ou de `JPanel`

```
public void paintComponent(Graphics g) {
 . . .
 g.drawImage(image, 75, 100, this);
 . . .
}
```



- Ce paramètre sert à indiquer un `ImageObserver` qui est informé du chargement de l'image ; la classe `Component` implémente `ImageObserver` en appelant la méthode `paint()` à chaque fois qu'une nouvelle partie de l'image est chargée

# Créer et dessiner une BufferedImage

```
// Créer une BufferedImage (avec un composant)
bufferedImage =
 (BufferedImage) composant.createImage(w, h);
// Créer une BufferedImage (sans composant)
bufferedImage =
 new BufferedImage(w, h,
 BufferedImage.TYPE_INT_RGB);
// Dessiner sur l'image
Graphics2D g2d = bufferedImage.createGraphics();
dessiner(g2d); // fait des dessins sur g2d
g2d.dispose();
```

# Changer les dimensions d'une image

- La méthode **drawImage** permet de modifier les dimensions de l'image affichée :

```
boolean
drawImage(Image img,
 int x, int y,
 int largeur, int hauteur,
 ImageObserver observateur)
```

- On peut aussi changer la couleur de fond

# Chargement d'une image

- La méthode `drawImage()` retourne immédiatement, même si l'image n'est pas entièrement disponible (une image peut être longue à charger en mémoire)
- Le chargement de l'image et son affichage est effectué par un *thread* en parallèle du traitement principal du programme

# Savoir si une image est chargée

- Pour savoir si une image est déjà chargée, le plus simple est d'utiliser la méthode `checkID` de la classe **MediaTracker** ; on passe le composant qui va utiliser l'image au `mediaTracker` (ou un autre composant) :

```
MediaTracker md = new MediaTracker(composant);
md.addImage(image, 0);
if (md.checkID(0)) {
 System.out.println("Chargement terminé");
 if (md.isErrorID(0))
 System.err.println("Erreur pendant chargement");
}
else
 System.out.println("Pas encore chargée");
```

# java.awt.MediaTracker

- On peut aussi utiliser un `MediaTracker` pour attendre le chargement complet d'une image
- Un *mediaTracker* peut surveiller le chargement de plusieurs images (chacune a un numéro) :

```
MediaTracker md = new MediaTracker(comp);
Image image = Toolkit.getDefaultToolkit()
 .getImage(image);
md.addImage(image, 0);
try {
 md.waitForID(0); // ou waitForAll()
}
catch (InterruptedException e) {}
```

# Taille d'une image

- Les méthodes **getHeight** (`ImageObserver observer`)  
**getWidth** (`ImageObserver observer`)  
de la classe `Image` renvoient les dimensions de l'image (-1 si l'information n'est pas encore disponible)
- Les méthodes **getIconWidth** () et **getIconHeight** () peuvent être utilisées pour les icônes

# Chargement d'une **ImageIcon**

- Une **ImageIcon** est automatiquement chargée par un *mediatracker* quand l'image est créée à partir d'un nom de fichier ou d'une URL
- La méthode **getImageLoadStatus ()** permet de savoir si l'image a pu être chargée
- **setImageObserver (ImageObserver)** permet d'observer le chargement de l'image (utile pour les images gif animées)

# Interface **Action**

# Représenter une action

- Il arrive souvent qu'une même action (par exemple, quitter l'application, imprimer, ouvrir un fichier, obtenir de l'aide) puisse être déclenchée par différents moyens :
  - choix d'un menu
  - clic sur un bouton de la souris
  - frappe d'une combinaison de touches au clavier (Ctrl-A)
  - etc.

# Informations centralisées

- Une action permet de centraliser
  - un texte (qui s’affiche sur les boutons ou les menus)
  - une icône
  - un traitement à exécuter
  - le fait que ce traitement est permis ou pas
  - un texte « `actionCommand` »
  - un mnémonique
  - un raccourci clavier
  - un texte qui décrit l’action (version longue ou courte utilisée par les bulles d’aide)

# Utilisation des actions

- Cette action peut être utilisée par plusieurs composants de types éventuellement différents
- Certains attributs de ces composants sont alors fixés par l'action
  - le texte des boutons ou des menus
  - l'action leur est ajoutée comme **ActionListener**
  - ...

# Classe et interface pour les actions

- L'interface **Action** (hérite de **ActionListener**) permet de représenter une telle action
- On héritera le plus souvent de la classe abstraite **AbstractAction**
- Des constructeurs de cette classe prennent en paramètres (optionnels) un texte et une icône

# Interface **Action**

- L'interface **Action** a les méthodes suivantes
  - **actionPerformed** (héritée de **ActionListener**)
  - **setEnabled** et **isEnabled** indiquent si l'action peut être lancée ou non
  - **putValue** et **getValue** permettent d'ajouter des attributs (paire "nom-valeur") à l'action ; 2 attributs prédéfinis : **Action.NAME** et **Action.SMALL\_ICON** (utilisés si l'action est associée à un menu ou à une barre d'outils)
  - **{add|remove}PropertyChangeListener** pour, par exemple, notifier un menu associé à une action que l'action est invalidée

# Classes qui peuvent utiliser une action

- Ce sont des sous-classes de **AbstractButton** ; elles ont en particulier un constructeur qui prend en paramètre une action :
  - Boutons (**JButton**)
  - Boutons radio (**JRadioButton**), y compris dans un menu (**JRadioButtonMenuItem**)
  - Boîtes à cocher (**JCheckBox**), y compris dans un menu (**JCheckBoxMenuItem**)
  - Menu (**JMenu**)
  - Choix de menu (**JMenuItem**)

# Utilisation des actions

```
ImageIcon image = new ImageIcon("gauche.gif");
```

```
actionPrecedent =
 new AbstractAction("Question précédente", image) {
 public void actionPerformed(ActionEvent e) {
 . . .
 }
 };
```

Définition de l'action

```
. . .
```

```
JButton bPrecedent = new JButton(actionPrecedent);
panel.add(bPrecedent);
```

Utilisation de l'action

```
bPrecedent.setText(""); // bouton "image" avec tooltip
```

# Utilisation des actions

- On peut associer une action à un bouton (en fait à un `AbstractButton`, c'est-à-dire `JButton`, `JMenuItem`, `JToggleButton`) par la méthode `setAction (Action)`
- Cette méthode fait tout ce qu'on peut en attendre ; elle ajoute en particulier l'action comme écouteur du bouton, met le nom (propriété `NAME`), l'icône (`SMALL_ICON`), le texte de la bulle d'aide du bouton (`SHORT_DESCRIPTION`)
- La méthode `getAction ()` récupère l'action

# *Java Look and Feel Repository*

- Sun fournit un ensemble d'icônes associées à des actions à l'adresse suivante :  
`http://developer.java.sun.com/developer/techDocs/hi/repository/`