Interfaces graphiques en Java - Introduction

Université de Nice - Sophia Antipolis

Version 3.5.1 – 21/4/05

Richard Grin

Contributions

- Des 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 tutorial 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

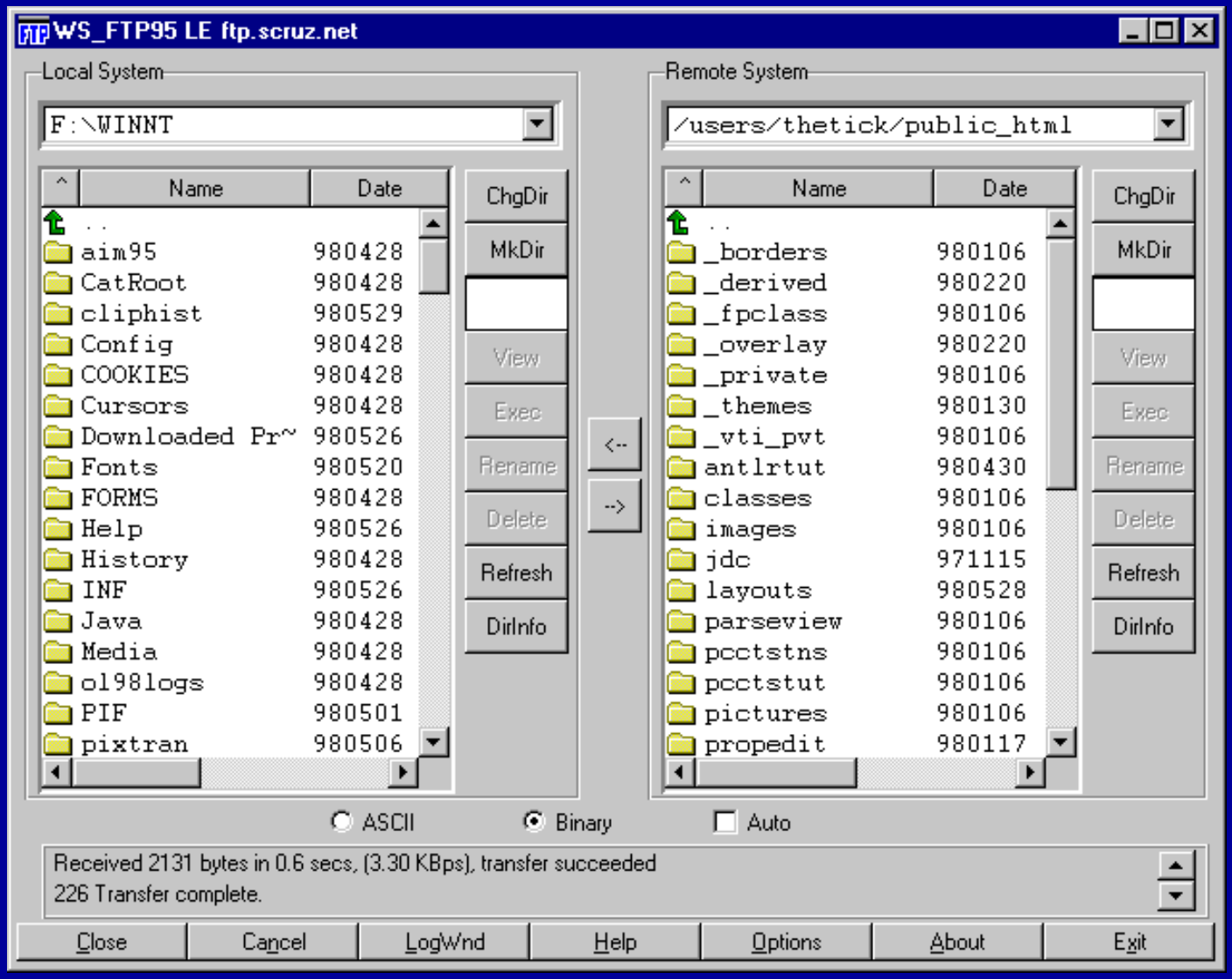
Interface avec l'utilisateur

- 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


WS_FTP95 LE ftp.scruz.net
Local System
F:\WINNT
Name
Date
..
aim95 980428
CatRoot 980428
cliphist 980529
Config 980428
COOKIES 980428
Cursors 980428
Downloaded Pr~ 980526
Fonts 980520
FORMS 980428
Help 980526
History 980428
INF 980526
Java 980428
Media 980428
ol98logs 980428
PIF 980501
pixtran 980506
ChgDir
MkDir
View
Exec
Rename
Delete
Refresh
DirInfo
<--
-->
Remote System
/users/thetick/public_html
Name
Date
..
_borders 980106
_derived 980220
_fpclass 980106
_overlay 980220
_private 980106
_themes 980130
_vti_pvt 980106
antlrtut 980430
classes 980106
images 980106
jdc 971115
layouts 980528
parseview 980106
pcctstns 980106
pcctstut 980106
pictures 980106
propedit 980117
ChgDir
MkDir
View
Exec
Rename
Delete
Refresh
DirInfo
ASCII
Binary
Auto
Received 2131 bytes in 0.6 secs, (3.30 KBps), transfer succeeded
226 Transfer complete.
Close
Cancel
LogWnd
Help
Options
About
Exit

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 déclencher

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 d'un écouteur

- Il est prévenu par le composant graphique dès qu'un événement qui le concerne survient sur ce composant
- Il exécute alors l'action 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`)

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);
  . . .
}
```

Depuis SDK 1.4,
`setLocationRelativeTo(null)`
centre une fenêtre sur l'écran

Problèmes d'affichage ?

- Dans certaines situations, assez rares, il peut se produire des problèmes d'accès concurrents (voir plus loin « Swing n'est pas *thread-safe* »)
- L'interface graphique se fige alors et ne fonctionne plus
- En ce cas, il faut lancer l'affichage de la fenêtre selon le schéma indiqué dans le transparent suivant

Afficher une fenêtre (2ème façon)

```
import javaw.swing.SwingUtilities;
. . .
private static void afficherGUI() {
  JFrame frame = new JFrame("Titre");
  . . .
  frame.pack();
  frame.setVisible(true);
}
public static void main(String[] args) {
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      afficherGUI();
    }
  });
}
```

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`**)

Émettre un bip

- La méthode **`beep()`** de la classe **`Toolkit`** permet d'émettre un bip :
  `tk.beep();`
- L'instance de **`Toolkit`** s'obtient par la méthode **`getDefaultToolkit()`**
- Le plus souvent ce bip prévient l'utilisateur de l'arrivée d'un problème ou d'un événement

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
- Les composants Java qui, comme les **`JFrame`**, s'appuient sur des composants du système hôte sont dit « lourds »
- L'utilisation de composants lourds améliore la rapidité d'exécution mais nuit à la portabilité et impose les fonctionnalités des composants

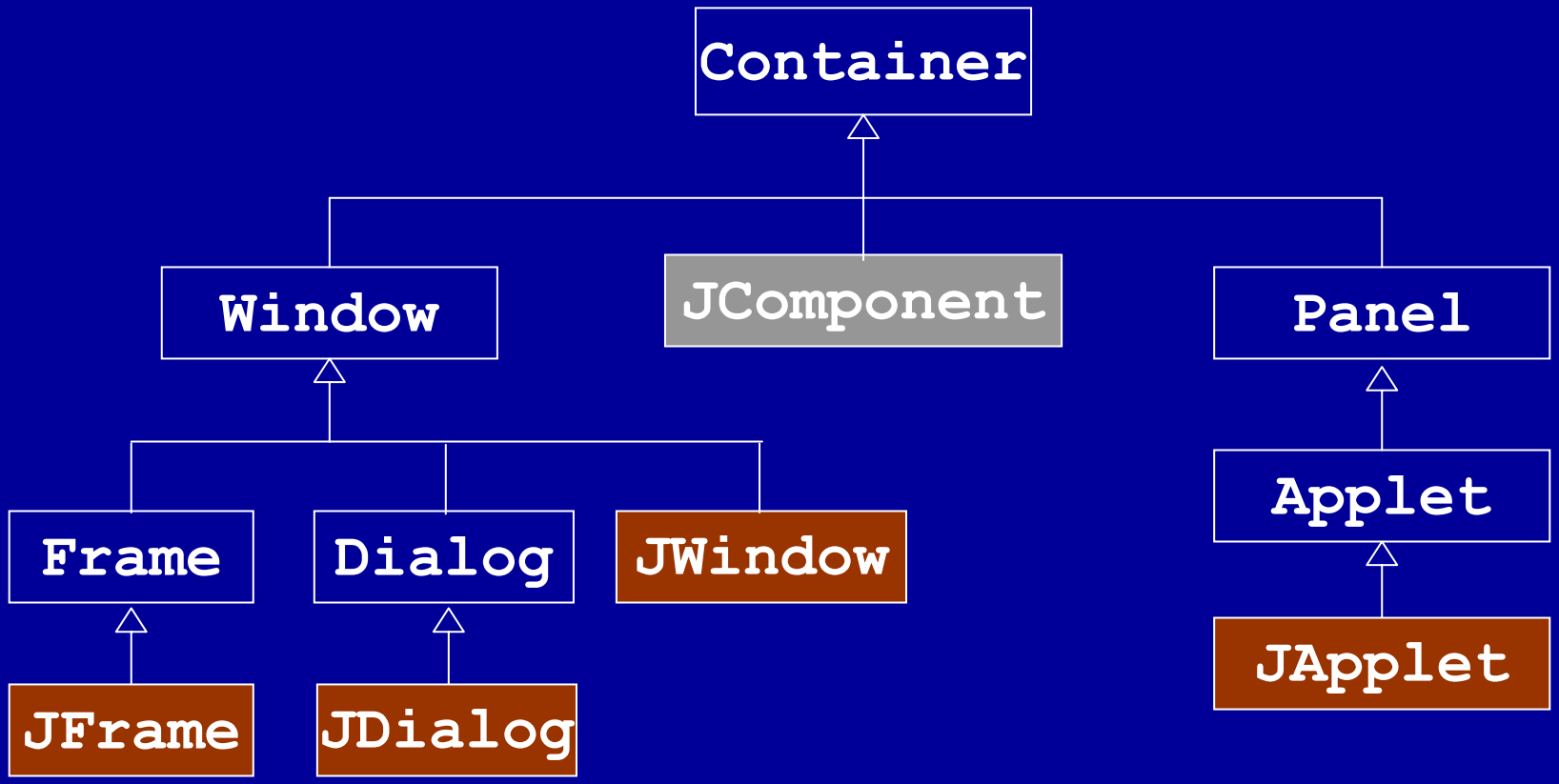
Composants légers

- AWT utilise les *widgets* du système d'exploitation pour tous les composants graphiques (fenêtres, boutons, listes, menus,…)
- Swing ne les utilise que pour les fenêtres de base « *top-level* »
- Les autres composants, dits légers, sont dessinés par Swing dans ces containers lourds
- Attention, les composants lourds s'affichent toujours au-dessus des composants légers

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

Hiérarchie d'héritage des containers lourds



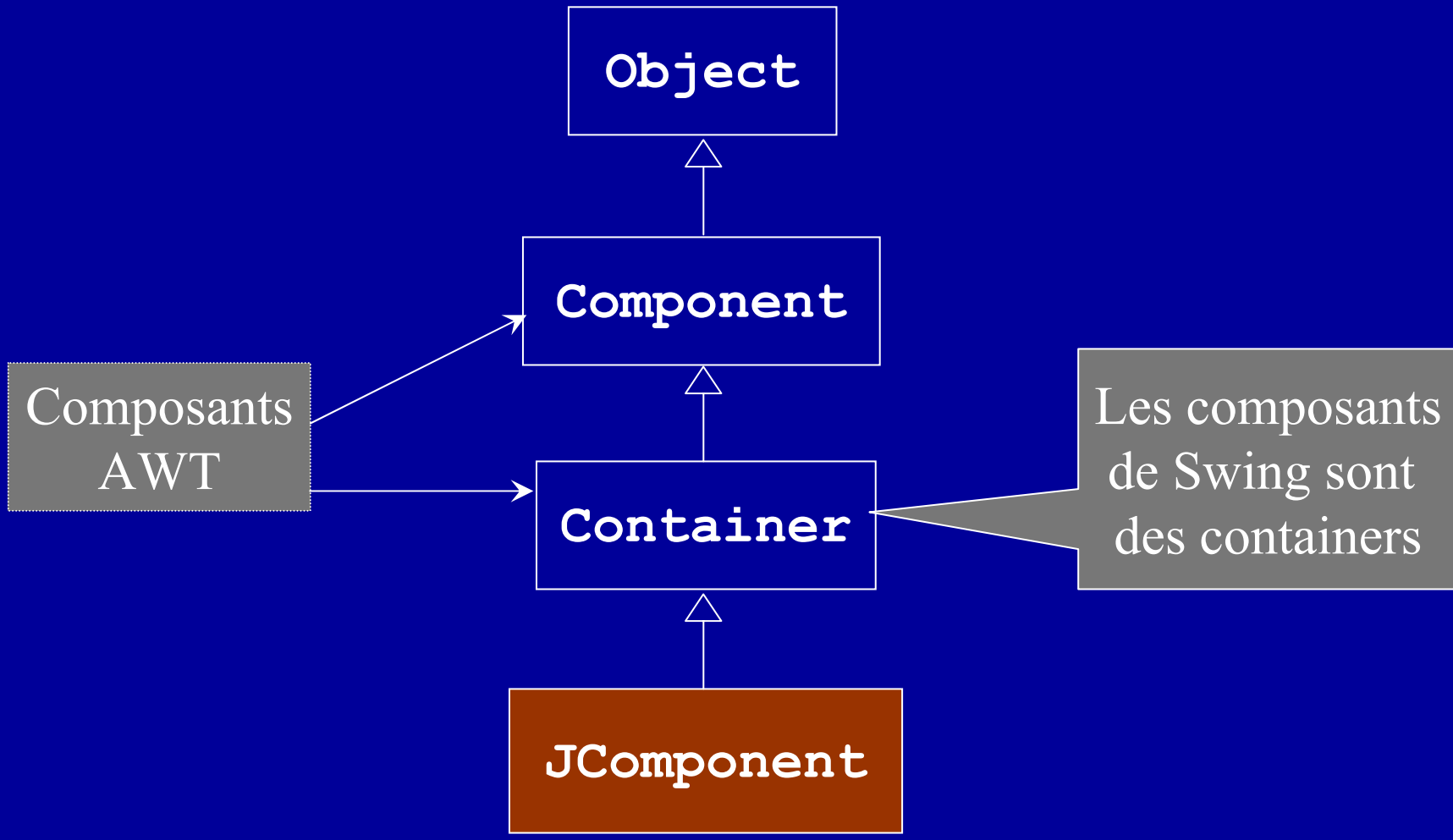
Libérer les ressources associées à une `JFrame`

- 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`

Classe `JComponent`

- 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 »
- **`JComponent`** héritent de la classe **`Container`**
- Tous les composants légers des sous-classes de **`JComponent`** peuvent donc contenir d'autres composants

Classe abstraite `JComponent`



Les Containers

- 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
- Un **`JPanel`** sert à regrouper des composants dans une zone d'écran
- Il 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`**)

Ajouter des composants dans une fenêtre

Le « *ContentPane* »

- Avant le JDK 5, les containers « *top-level* » ne pouvaient contenir *directement* d'autres composants
- Ils sont associés à un autre container, le « *content pane* » dans lequel on ajoutait les composants
- On obtient ce *content pane* par (**`topLevel`** est un container lourd ; **`JFrame`** par exemple)

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

Ajouter les composants

- Le plus souvent on ajoute les composants dans le constructeur du composant « top-level » :

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

Depuis JDK 5

- On peut désormais ajouter directement les composants dans un composant « top-level » :

```
frame.add(label, BorderLayout.NORTH);
frame.add(b1, BorderLayout.SOUTH);
```

- Ce container va en fait déléguer à son *content pane* qui existe toujours

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
- Accesseurs et modificateurs associés : **`{get|set}{Maximum|Preferred|Minimum}Size`**

Taille préférée

- La taille préférée est la plus utilisée par les *layout managers* ; un composant peut l'indiquer en redéfinissant la méthode « `Dimension getPreferredSize()` »
- On peut aussi l'imposer « de l'extérieur » avec la méthode « `void setPreferredSize(Dimension)` »
- Mais le plus souvent, les gestionnaires de mise en place ne tiendront pas compte des tailles imposées de l'extérieur

Taille minimum

- Quand un composant n'a plus la place pour être affiché à sa taille préférée, il est affiché à sa taille minimum sans passer par des tailles intermédiaires
- Si la taille minimum est très petite, ça n'est pas du plus bel effet ; il est alors conseillé de fixer une taille minimum, par exemple par `c.setMinimumSize(c.getPreferredSize());`

Changer le *Layout manager*

- Tous les containers ont un gestionnaire de placement par défaut
- On peut changer ce gestionnaire de placement d'un **`Container`** par la méthode **`setLayout(LayoutManager)`** de la classe **`Container`**

Layout manager par défaut d'une fenêtre

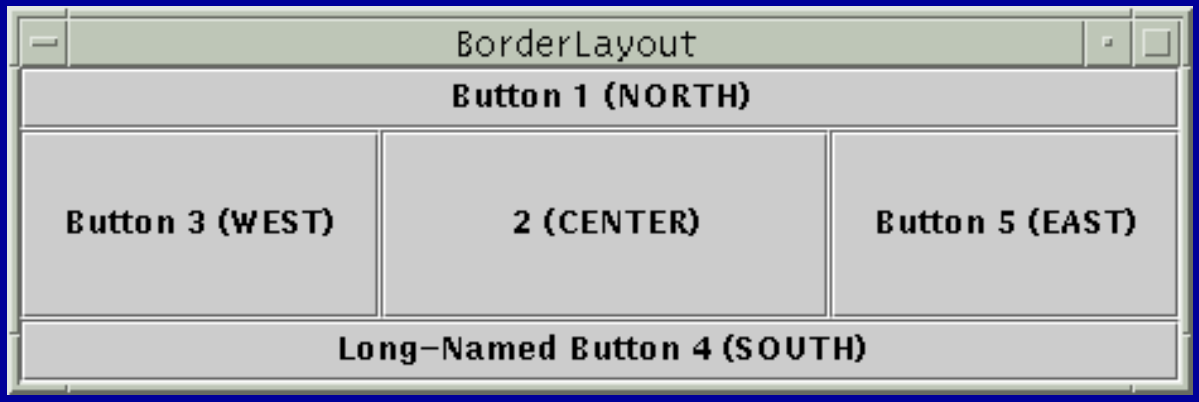
- Le gestionnaire de mise en place par défaut des fenêtres **`JFrame`** est du type **`BorderLayout`**
- On peut changer ce gestionnaire de mise en place en envoyant la méthode **`setLayout(`*`LayoutManager`*`)`** de la classe **`Container`** au *content pane*
- Depuis le JDK 5 on peut envoyer la méthode setLayout directement à la fenêtre (elle délègue au *content pane*)

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

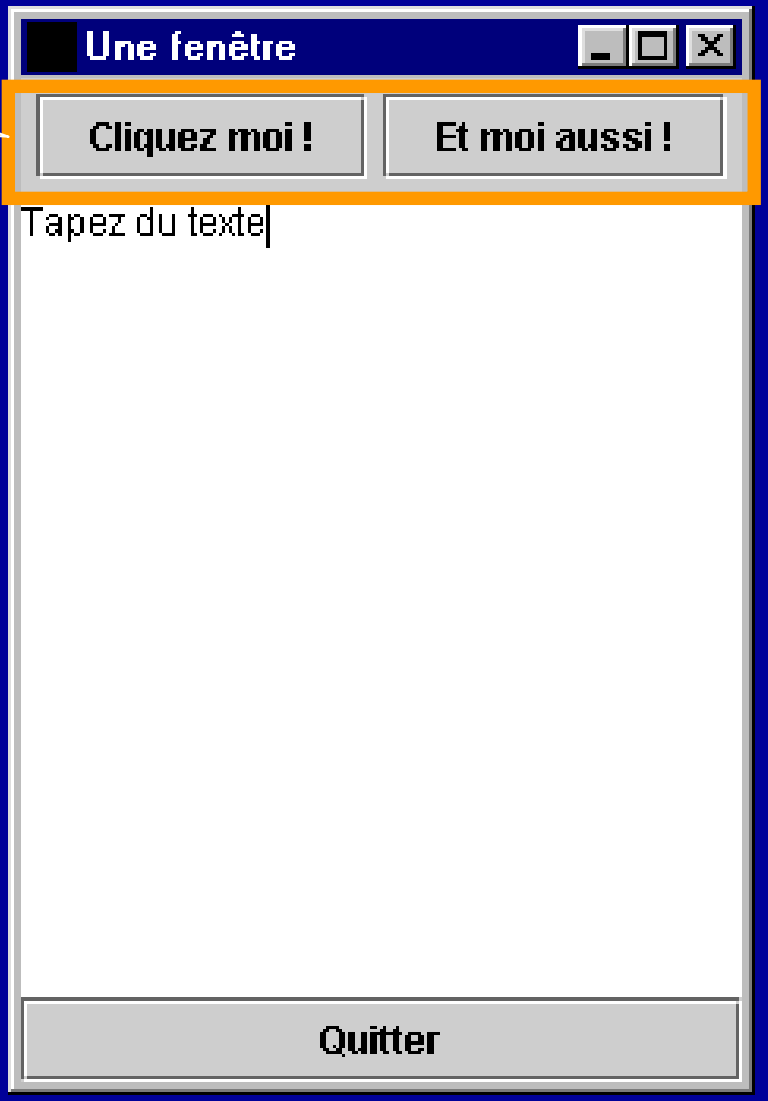
- Les composants sont centrés dans leur zone
- On peut spécifier des espacement horizontaux et verticaux minimaux entre les composants
- Si on oublie de spécifier le placement lors de l'ajout d'un composant, celui-ci est placé au centre (source de bug !)
- Règle pratique : l'est et l'ouest peuvent être 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` au nord



Utiliser un `JPanel`

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

Source de la classe Exécution

FlowLayout

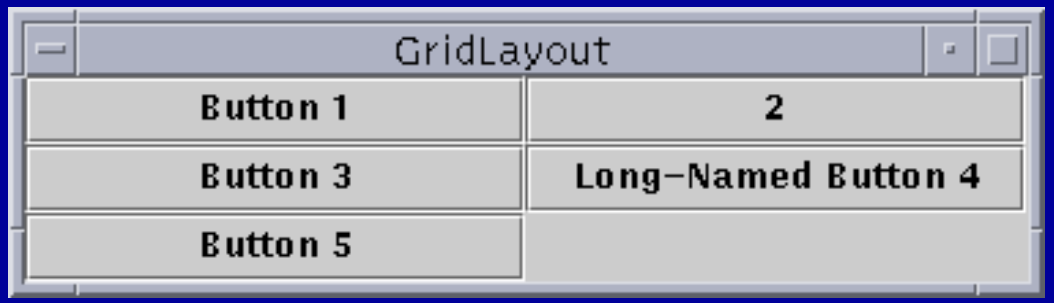
- Rangement de haut en bas et de gauche à droite
- Les composants sont affichés à leur taille préférée
- *layout manager* par défaut de **`JPanel`** et **`JApplet`**
- Attention, la taille préférée d'un container géré par un **`FlowLayout`** est calculée en considérant que tous les composants sont sur 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 ou colonne par colonne (suivant le nombre indiqué au constructeur)



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

Contraintes de base

```
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 (occupe 2 lignes ici)
contraintes.gridheight = 2;
contraintes.gridwidth = 1;
// Chaque élément peut avoir ses propres contraintes
panel.add(bouton, contraintes);
```

Autres contraintes : placement

- **`fill`** détermine si un composant occupe toute la place dans son espace réservé (constantes de la classe **`GridBagConstraint`** : **`BOTH`**, **`NONE`**, **`HORIZONTAL`**, **`VERTICAL`**) ; par défaut **`NONE`**
- **`anchor`** dit où placer le composant quand il est plus petit que son espace réservé (**`CENTER`**, **`SOUTH`**, …) ; par défaut **`CENTER`**

Répartition de l'espace libre

- **`weightx`**, **`weighty`** (de type **`double`**) indique comment sera distribué l'espace libre. Plus le nombre est grand, plus le composant récupérera d'espace ; par défaut 0
  Si tous les poids sont 0, l'espace est réparti dans les espaces placés entre les composants
  Algorithme : poids d'une colonne = **`weightx`** maximum des composants de la colonne ; total = somme des poids de toutes les colonnes ; l'espace libre d'une colonne est donné par **`weightx`**/total

La place minimale occupée

- **`insets`** ajoute des espaces autour des composants : **`contraintes.insets = new Insets(5,0,0,0)`** (ou **`contraintes.insets.left = 5`**) ; par défaut 0 partout
- **`ipadx`**, **`iapdy`** ajoutent des pixels à la taille minimum des composants ; par défaut 0
- Le layout manager tient compte des tailles préférées et minimales pour afficher les composants

Remarque sur le positionnement

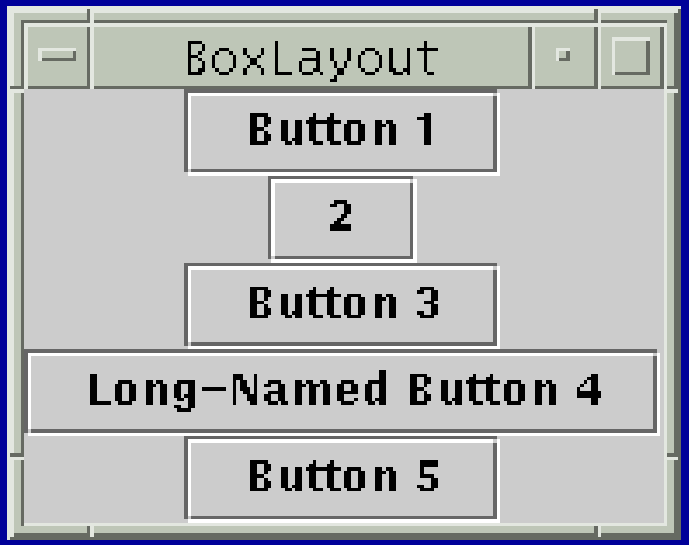
- La valeur par défaut pour **`gridx`** et **`gridy`** est **`GridBagConstraints.RELATIVE`** (se place en dessous ou à droite du précédent)
- Il suffit donc de fixer **`gridx`** à la valeur d'une colonne pour les composant de la colonne et de garder la valeur par défaut de **`gridy`** pour placer tous les composants de la colonne les uns à la suite des autres (idem pour les lignes en fixant **`gridy`**)

Quand faut-il choisir GridBagLayout ?

- Dès que l'interface devient trop compliquée, il est souvent plus simple d'utiliser **`GridBagLayout`** plutôt que de trouver des constructions très rusées avec des containers intermédiaires
- On peut rendre le code plus lisible avec des méthodes qui facilitent l'affectation des contraintes

BoxLayout

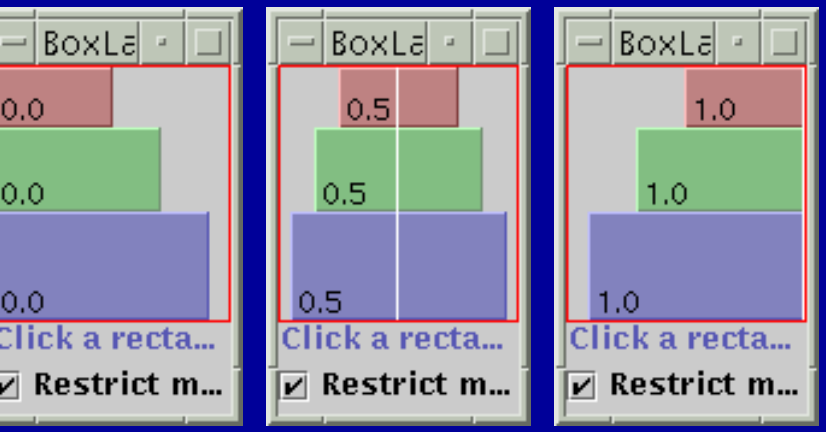
- Aligne les composants sur une colonne ou une ligne (on choisit à la création)
- Respecte la largeur (resp. hauteur) préférée et maximum, et l'alignement horizontal (resp. vertical)
- *Layout manager* par défaut de **`Box`** et de **`JToolBar`**

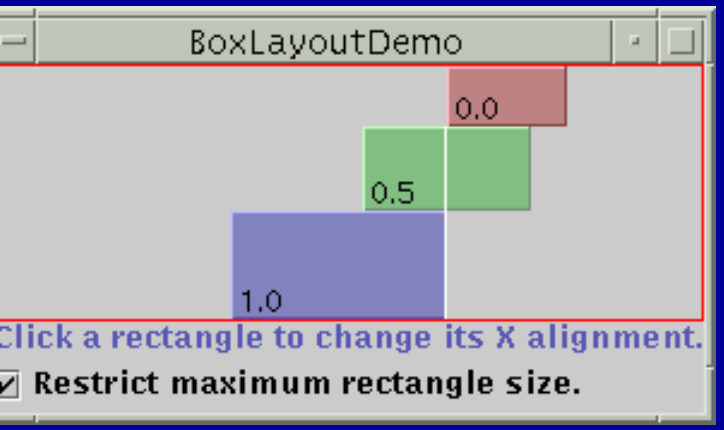


BoxLayout

- 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**

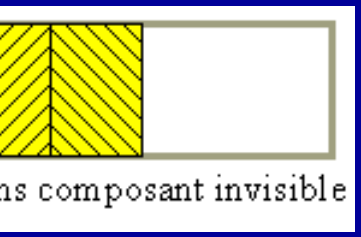
Problèmes d'alignement

- Si tous les composants géré 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

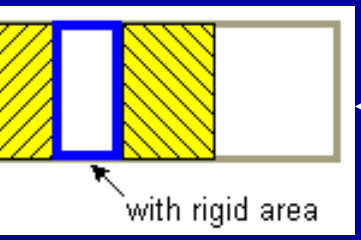
Classe **`Box`**

- 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*, étais et zones rigides

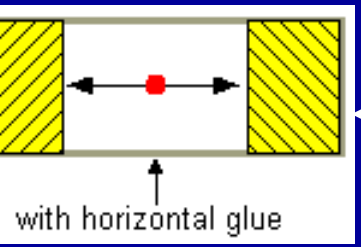
Classe **Box** ; composants invisibles



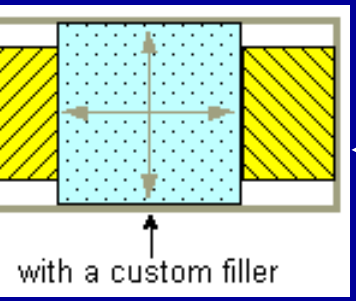

```
container.add(composant1);
// On ajoute ici un composant invisible
container.add(. . .);
container.add(composant2);
```




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




```
Box.createHorizontalGlue())
```




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

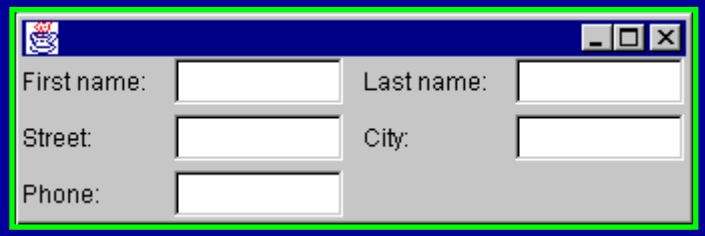
Code avec **Box**

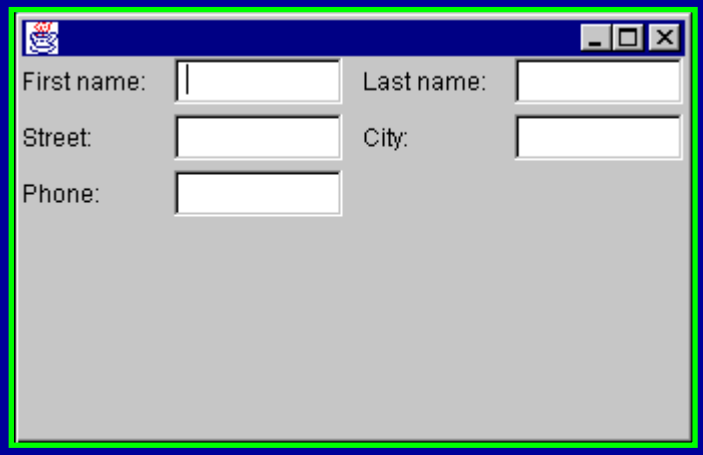
```
// Le plus simple est d'utiliser une Box
// mais on peut aussi mettre un BoxLayout
// pour gérer un autre container
Box b = Box.createVerticalBox();
b.add(bouton);
// On peut ajouter des zones invisibles
// entre les composants :
b.add(Box.createVerticalStrut(5));
// ou b.add(Box.createRigidArea(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

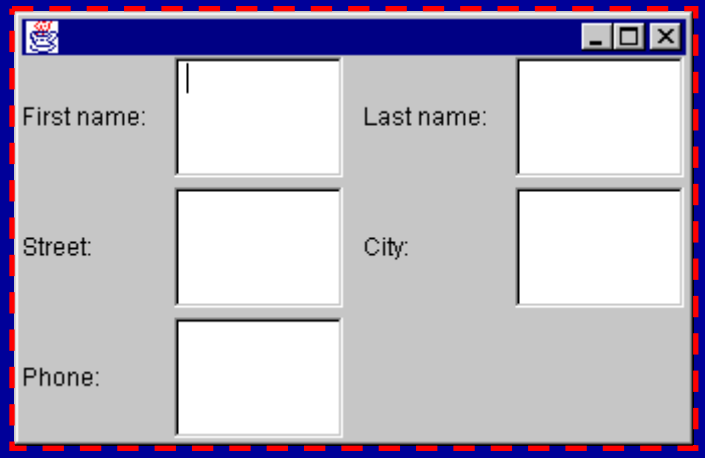
Exemples et contre-exemples

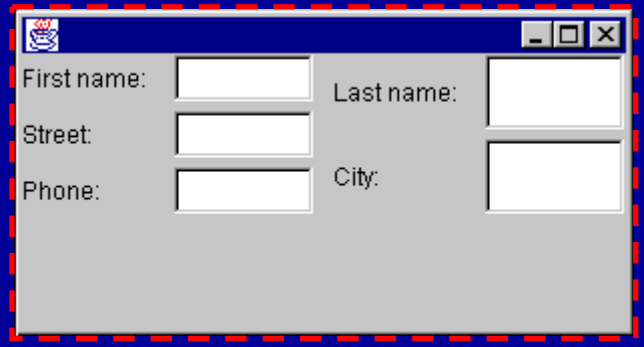




Bon

Mauvais





Exercice : donner 2 solutions

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

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 complète 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, retaillé, 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

Une question

- Quel message sera envoyé par le composant à ses écouteurs pour les prévenir que l'événement qui les intéresse est arrivé ?
- Réponse : à chaque type d'écouteur correspond une interface que doit implémenter la classe de l'écouteur ; par exemple **`ActionListener`**, **`MouseListener`** ou **`KeyListener`**
- Le message doit correspondre à une des méthodes de cette interface

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 qui vont 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)

contient des informations sur l'événement

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 (ce sont des constantes de l'interface **`WindowConstants`**, implémenté par 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`

- Classe ancêtre des classes d'événements
- Une instance d'une classe d'événement est passée en paramètre aux méthodes des écouteurs
- Cette instance décrit l'événement qui a provoqué l'appel de la méthode

Méthode `getSource`

- L'écouteur peut interroger l'événement pour lui demander le composant qui l'a généré
- La méthode « `public Object getSource()` » de `EventObject` renvoie le composant d'où est parti l'événement, par exemple un bouton ou un champ de saisie de texte
- Souvent indispensable 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

- La classe **`InputEvent`** fournit la méthode **`getModifiers()`** qui renvoie un entier indiquant les touches modificatrices (Shift, Ctrl, Alt, Alt Gr) qui étaient appuyées au moment de l'événement souris ou clavier
- Depuis la version 1.4, il est conseillé d'utiliser plutôt la méthode **`getModifiersEx()`** qui corrige des petits défauts de la méthode précédente

Tester les touches modificatrices (2)

- On utilise les constantes **`static`** de type **`int`** suivantes de la classe **`InputEvent`** :
  `SHIFT_MASK` , **`CTRL_MASK`**, **`ALT_MASK`**, **`META_MASK`**, **`ALT_GRAPH_MASK`**
- Autres constantes de la classe **`InputEvent`** pour tester les boutons de la souris :
  `BUTTON1_MASK`, **`BUTTON2_MASK`**, **`BUTTON3_MASK`**
- Si on utilise **`getModifiersEx`**, il faut utiliser les constantes qui comportent **`DOWN`** dans leur nom ; par exemple, **`SHIFT_DOWN_MASK`**

Exemples d'utilisation

- Tester si la touche Ctrl était appuyée :

```
(e.getModifiers() & InputEvent.CTRL_MASK)
!= 0)
```

ou

```
e.isControlDown()
```

- Tester si l'une des touches Ctrl ou Shift était appuyée :

```
e.getModifiers() & (SHIFT_MASK | CTRL_MASK)
!= 0)
```

Exemples d'utilisation (2)

- Tester si les 2 touches Ctrl ou Shift étaient appuyées, mais pas la touche Meta (il faudrait beaucoup de doigts !) :

```
int on = InputEvent.SHIFT_MASK
         | InputEvent.CTRL_MASK;
int off = InputEvent.META_MASK;
boolean result =
  (e.getModifiers() & (on | off)) == on;
```

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 b qui contient un 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

ActionCommand

- On peut associer à chaque bouton (et choix de menus) une chaîne de caractères par

```
bouton.setActionCommand("chaine");
```

- 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 une 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();
}
```

Événements souris

2 types d'écouteurs

- Un type d'événement **`MouseEvent`** pour 2 types d'écouteurs
- **`MouseListener`** : bouton enfoncé ou relâché, clic (bouton enfoncé et tout de suite relâché), entrée ou sortie du curseur dans la zone d'un composant
- **`MouseMotionListener`** : déplacement ou glissement (*drag*) de souris

Écouteur `MouseListener`

- L'interface `MouseListener` contient les méthodes
  - `mousePressed(MouseEvent)`
  - `mouseReleased(MouseEvent)`
  - `mouseClicked(MouseEvent)`
  - `mouseEntered(MouseEvent)`
  - `mouseExited(MouseEvent)`
- Adaptateur `MouseAdapter`

Écouteur `MouseMotionListener`

- L'interface **`MouseMotionListener`** contient les méthodes
  - **`mouseMoved(MouseEvent)`**
  - **`mouseDragged(MouseEvent)`**
- Adaptateur **`MouseMotionAdapter`**

Roulette de la souris

- La version 1.4 du SDK a ajouté le type d'événement **`MouseWheelEvent`**, associé aux souris à roulettes
- Ces événement peuvent être pris en compte par les écouteurs **`MouseWheelListener`**
- Le composant **`ScollPane`** a un tel écouteur, ce qui permet de déplacer la vue en utilisant la roulette de la souris

Traitement des `MouseWheelEvent`

- Ce type d'événement a un traitement particulier : l'événement est propagé au premier composant intéressé, englobant le composant où est situé le curseur
- Le plus souvent aucun codage ne sera nécessaire pour traiter ce type d'événements car le composant qui recevra l'événement sera un `ScrollPane` qui est d'origine codé pour le traiter

Focus

- La gestion du *focus* a 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*

- 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`**

Compléments : modifier les touches de déplacement

- Les touches qui permettent de se déplacer entre les composants peuvent être modifiées en utilisant le **`KeyboardFocusManager`**

Compléments : modifier la politique de déplacement

- L'ordre de déplacement peut être modifié en choisissant une politique de déplacement
- Par défaut, sous Swing, l'ordre de déplacement dans un container dépend de la position sur l'écran
- On peut choisir un autre ordre avec la méthode de la classe **`Container`**

```
setFocusTraversalPolicy(
        FocusTraversalPolicy policy)
```

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 le *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* »

- Un seul *thread* appelé le *thread* de distribution des événements (*event dispatch thread*) effectue
  - la récupération des événements dans la file d'attente,
  - le traitement de ces événements (méthodes des écouteurs)
  - l'affichage de l'interface graphique

Traitements longs des événements

- Tout traitement long effectué par ce *thread* fige l'interface graphique qui répond mal, ou même plus du tout, aux actions de l'utilisateur
- Il faut donc effectuer les traitements longs (accès à une base de données, calculs complexes,…) des écouteurs et des méthodes `paintComponent` dans des tâches à part

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 par le *thread* de distribution des événements

Modifier un composant depuis d'autres threads

- 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 thread de distribution des événements :
  - **`void invokeLater(Runnable`** *runnable*`)`
  - **`void invokeAndWait(Runnable`** *runnable*`)`

Utiliser d'autres *threads* …

- **`invokeLater`** dépose une tâche à accomplir dans la file d'attente des événements ; la méthode retourne ensuite sans attendre l'exécution de la tâche par le TDE
- Le TDE exécutera cette tâche comme tous les traitements d'événements
- **`invokeAndWait`**, dépose une tâche mais ne retourne que lorsque la tâche est exécutée par le TDE

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 (dans un paquetage)
  - classes pour l'interface graphique (GUI) (dans un autre paquetage)
  - les écouteurs du GUI 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(. . .);
   }
}
```

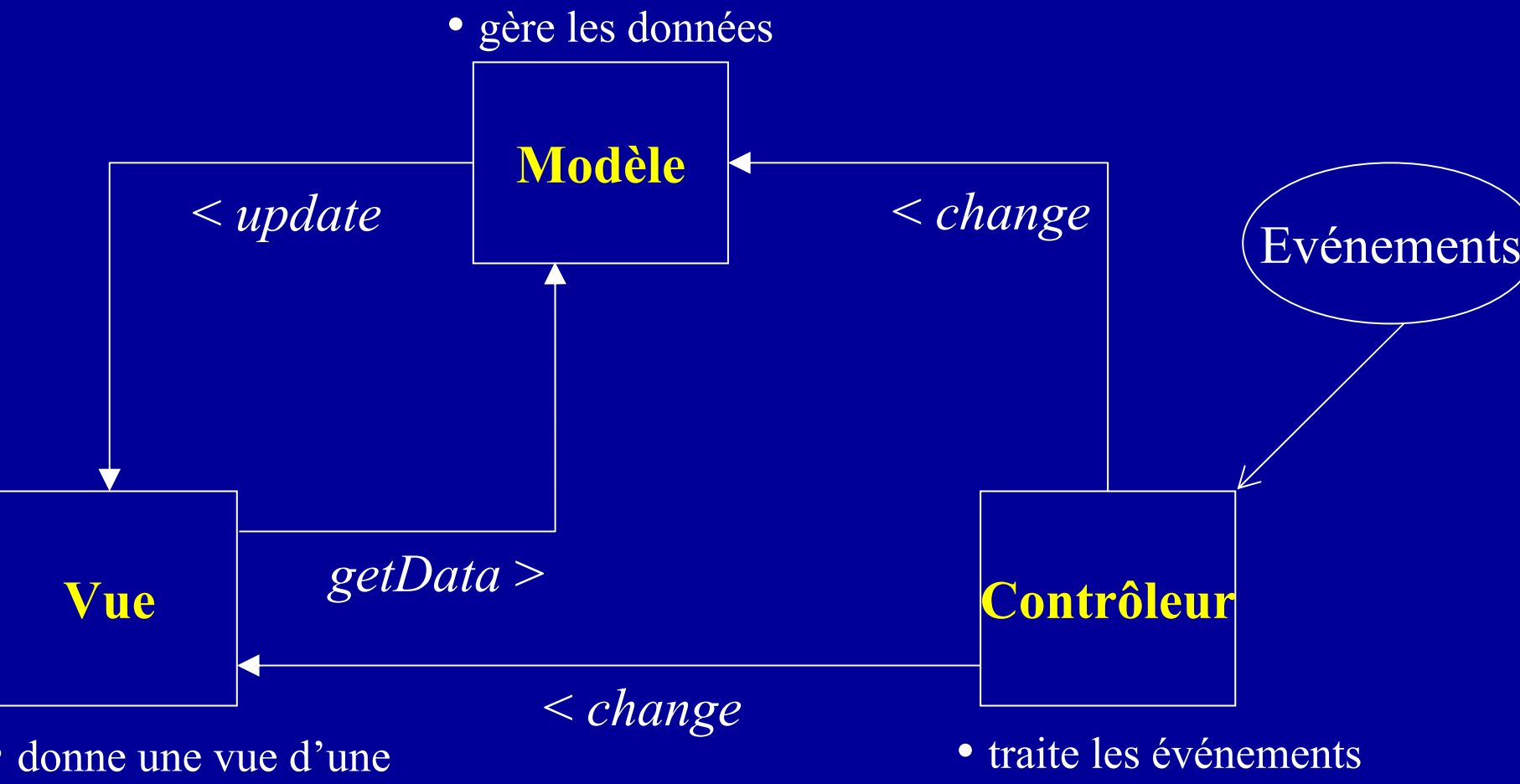
Les traitements de fin éventuels sur les classes métiers peuvent être placés dans les écouteurs au moment de quitter le GUI

Pattern MVC en Swing

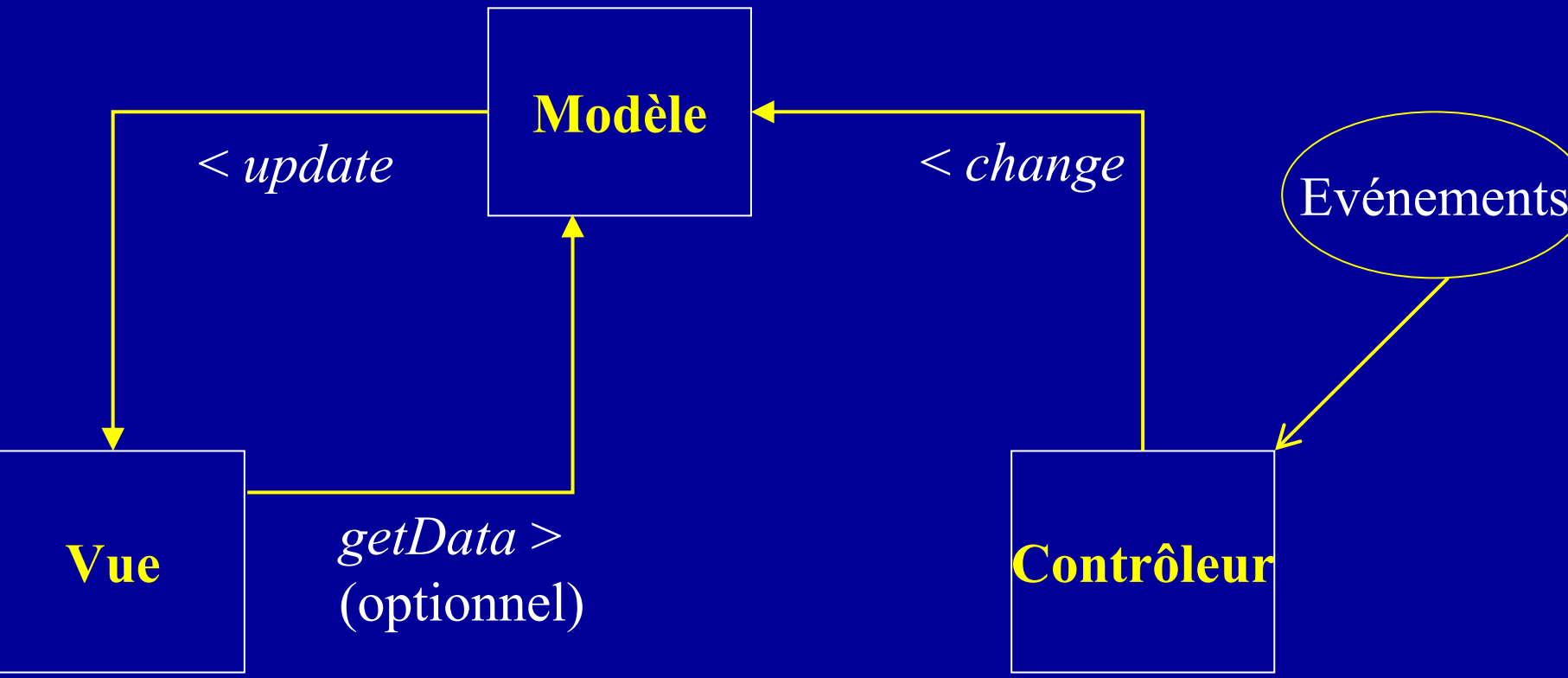
Architecture MVC

- 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

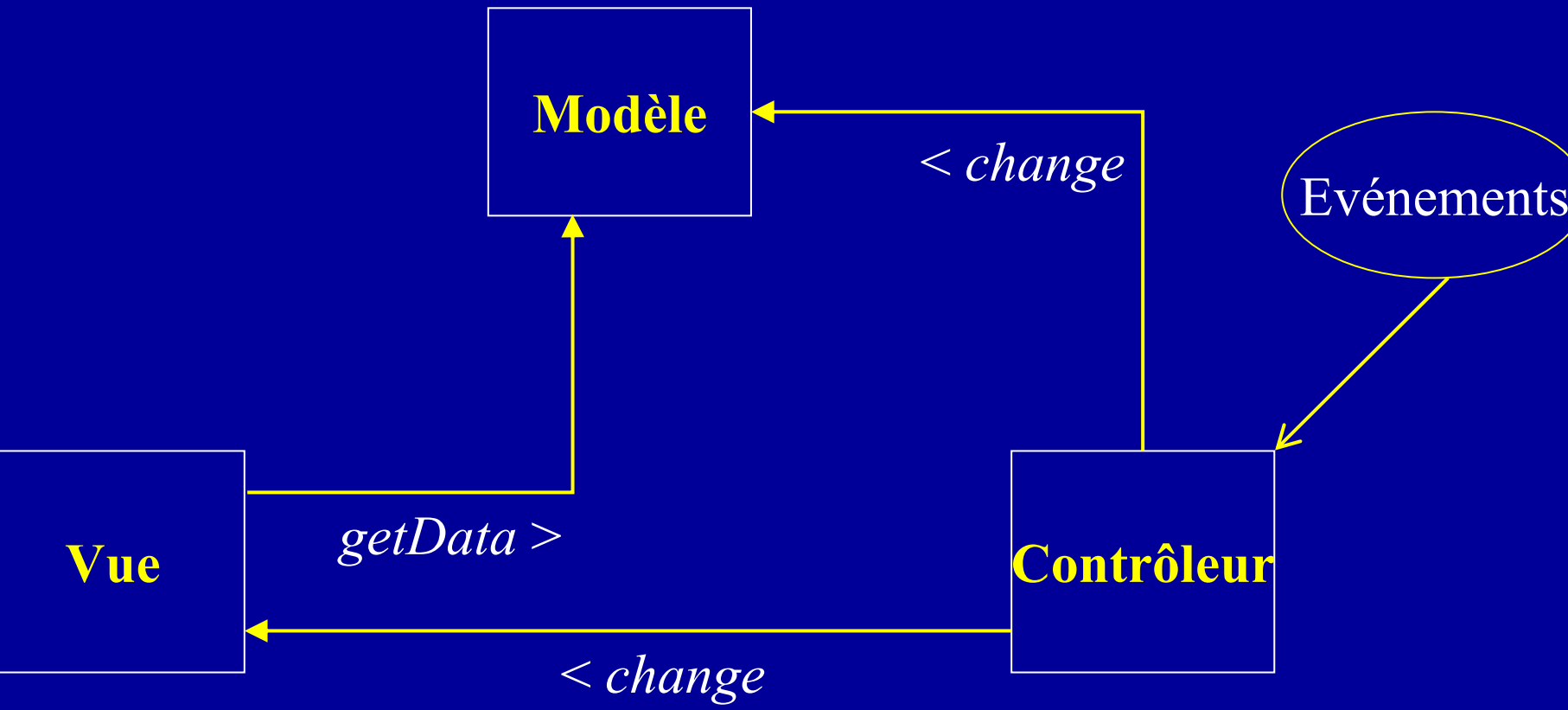


Architecture MVC



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

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
- Un *look and feel* est constitué de tous les objets *UI-delegate* associés

Pattern « fabrique abstraite »

- Le changement de *look and feel* utilise le modèle de conception (*design pattern*) « fabrique abstraite » qui permet de choisir une famille de classes indépendamment du reste de l'application
- Chaque *look and feel* correspond à une fabrique d'un certain type de *UI-delegate* (gérée par le *UI-manager*)
- Pour changer de *look and feel*, il suffit de dire que l'on veut changer de fabrique

« *Pluggable Look and feel* »

- Pour 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);
```

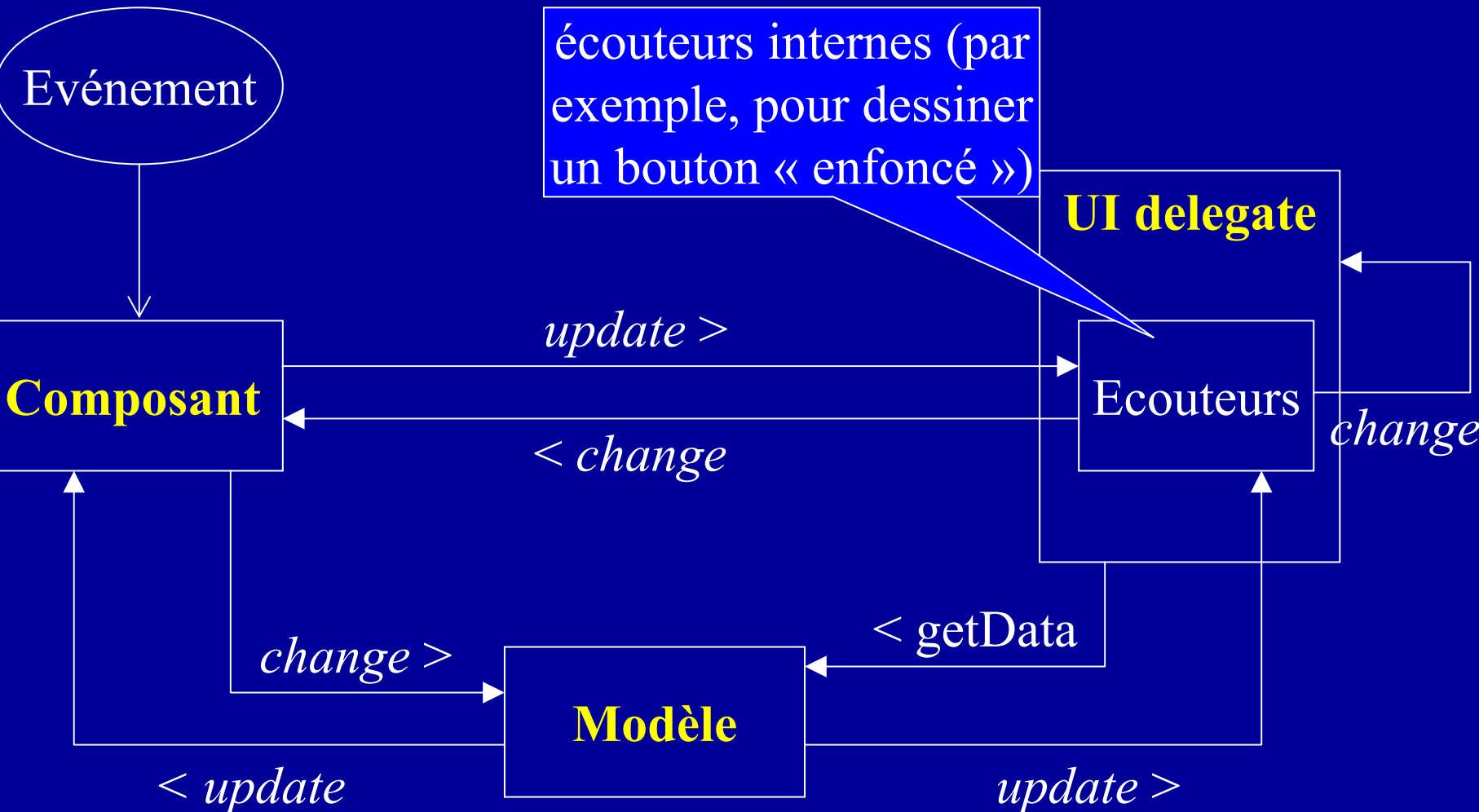
Architecture avec UI-delegate

- Le composant
  - est l'interlocuteur pour les objets extérieurs
  - contient le code pour les comportements de base du composant
  - reçoit les événements (générés par le système)
- Le modèle contient les données

Architecture avec UI-delegate (2)

- Le UI-delegate
  - représente une façon de voir ces données et de traiter les événements
  - écoute le composant (pour traiter les événements en tant que contrôleur) ; il contient des classes internes auxquelles il délègue ce traitement
  - écoute le modèle (pour afficher une vue de ce modèle)

Architecture avec *UI delegate*



Un exemple d'utilisation des modèles : les listes

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

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(
    ListSelectionModel.SINGLE_SELECTION);
```

Constructeurs des listes non modifiables

- 2 constructeurs pour des listes non modifiables :

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

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(
    ListSelectionModel.SINGLE_SELECTION);
// Pour cet exemple, la classe est l'écouteur
liste.addListSelectionListener(this);
// Ajoute le scrollPane dans le container,
// ce qui ajoutera la liste
c.add(sp);
```

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 qu'on ignore le plus souvent

Exemple d'écouteur d'une liste

```
public class EcouteurListe
        implements ListSelectionListener
  public void valueChanged(ListSelectionEvent e) {
    // On ignore les désélections
    if (e.getValueIsAdjusting()) return;
    // On traite les sélections
    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

- Une liste modifiable est associée à un modèle qui fournit les données affichées par la liste
- Ce modèle est une classe qui implémente l'interface **`ListModel`**
- On construit la liste avec le constructeur

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

Modèle de données pour une liste

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

- Pour faciliter l'écriture d'une classe qui implémente **`ListModel`**, le JDK fournit 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`**
- Avec **`DefaultListModel`** on gère les données avec des méthodes semblables aux méthodes **`add`** et **`remove`** des collections : **`add(int, Object)`**, **`addElement(Object)`**, **`remove(int)`**, **`removeElement(Object)`**, **`clear()`**

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

- On peut ne pas vouloir enregistrer tous les éléments de la liste en mémoire centrale
- Le modèle héritera alors directement 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 et couleur « XOR »
- 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 la couleur de XOR, et vice-versa

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 souvent indispensable de redéfinir les méthodes `getPreferredSize()` et `getMinimumSize()` du composant qui redéfinit la méthode `paintComponent()` (*`composant`*`.setSize()` ne marche pas dans ce cas)
- On peut aussi utiliser la méthode `setPreferredSize(Dimension)` pour cela

Dimensionner un composant

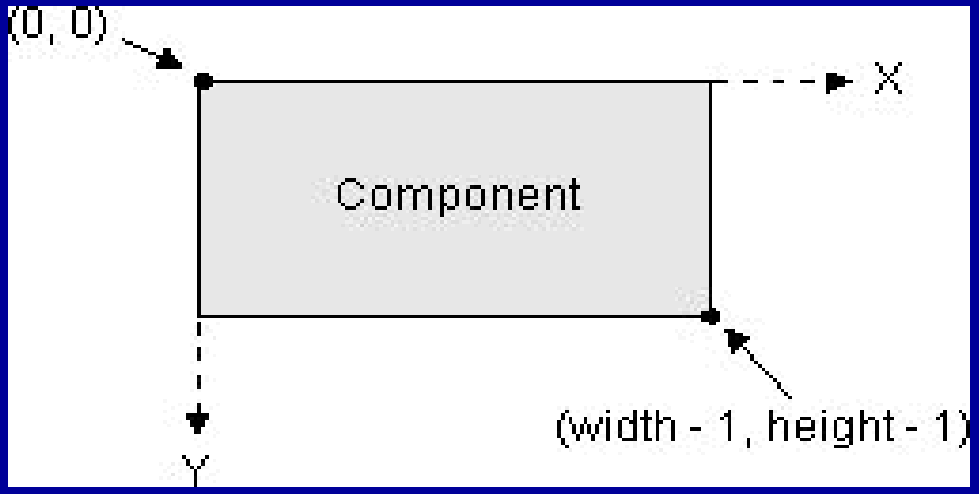
- **`setSize`** détermine les valeurs des variables internes associées à la dimension d'un composant
- Cette méthode ne convient pas toujours pour dimensionner un composant interne à une fenêtre
- En effet, les gestionnaires de placement utilisent les méthodes **`get{Preferred|Minimum}Size`** qui peuvent ne pas tenir compte des valeurs internes pour l'affichage d'un composant (par exemple un bouton ne tient compte que de la taille de son texte)

Dessiner avec la classe `Graphics`

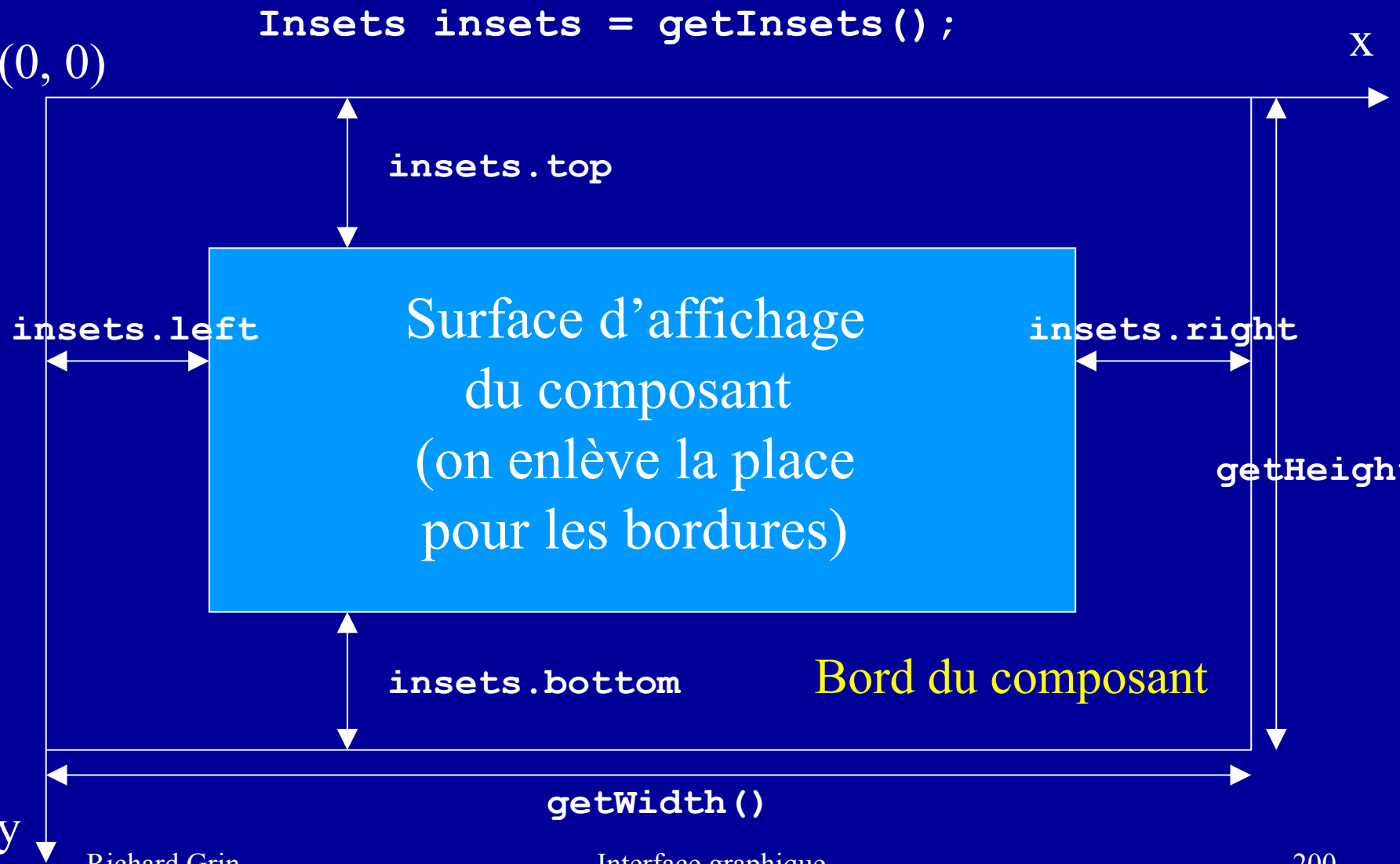
- `Graphics` offre plusieurs méthodes pour afficher divers éléments :
  - `drawString()` affiche du texte
  - `drawImage()` affiche une image
  - `drawLine()` affiche un segment
  - `fillOval()` affiche une ellipse
  - `drawPolygon()` affiche un polygone
  - …

Système de coordonnées

- 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



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() {
   . . .
   this.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)`**
- Si **`repaint`** ne suffit pas pour faire afficher correctement les composants mais que tout s'affiche correctement après avoir redimensionné la fenêtre, il manque sans doute un **`revalidate`** sur le composant à réafficher

`revalidate`, `invalidate`, `validate`

- **`invalidate()`** rend « non correct » le composant qui reçoit le message, et tous les composants qui le contiennent
- **`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

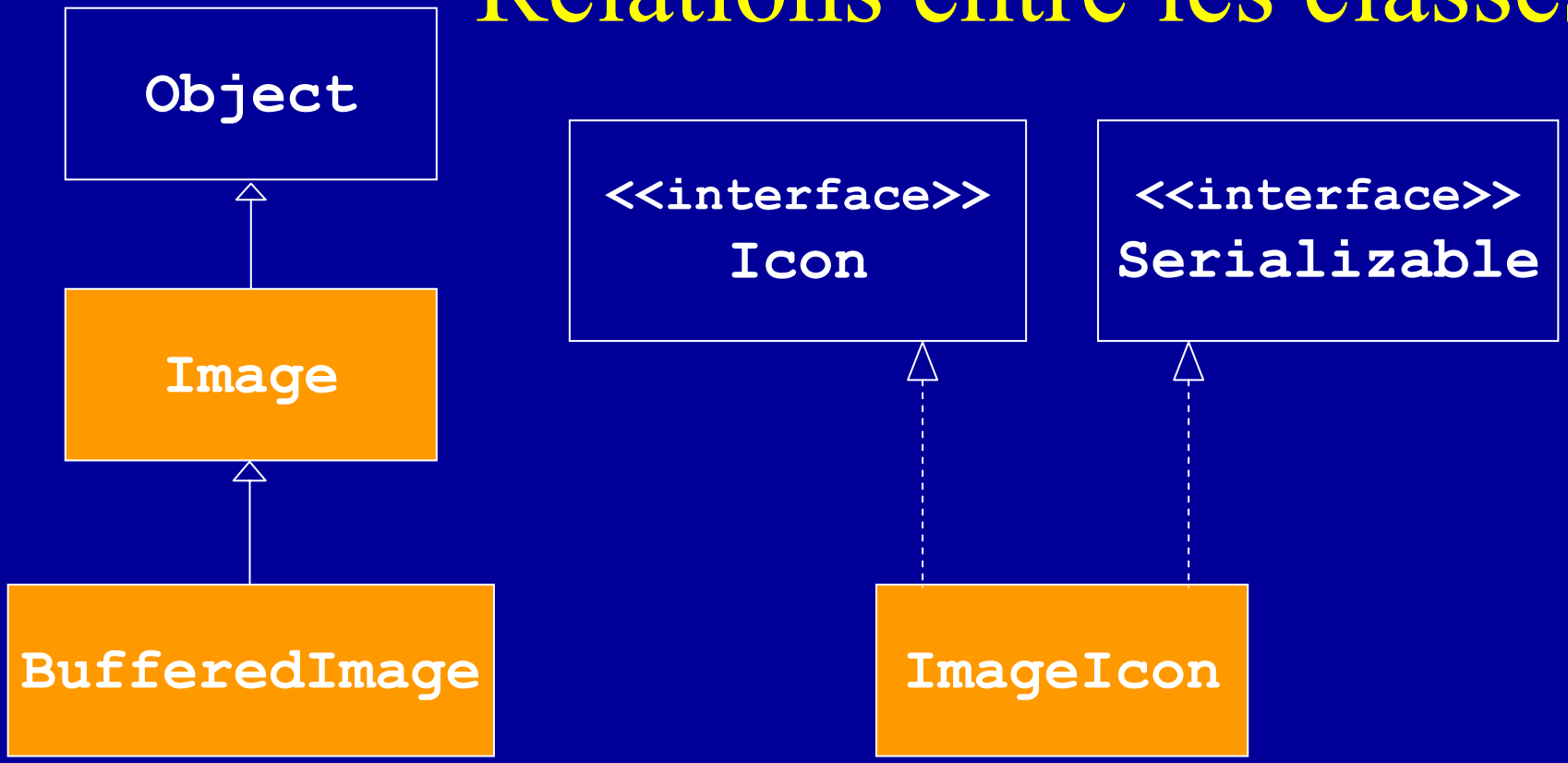
Images

- Les images sont des objets complexes (pixels, modèle de représentation des pixels, en particulier pour les couleurs)
- De base, Java sait travailler avec les images GIF, JPEG ou PNG, et les GIF animés

Classes pour les images

- 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

Relations entre les classes



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

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

- On va commencer par des généralités sur les images et on étudiera ensuite les 3 classes d'images plus en détails

Traitement standard avec une image

1. On récupère l'image depuis un fichier ou URL local ou depuis une URL distante
2. On affiche l'image dans un composant (on ne peut afficher directement une image dans un container)

- Les **`BufferedImage`** permettent de faire des traitements sur l'image avant de l'afficher

Récupérer une image

- Le chargement d'une image peut être long
- Le plus souvent ce chargement doit être effectuée par un *thread* en parallèle pour libérer le *thread* de distribution des événements et ne pas figer l'interface graphique

Affichage d'une image

- Lorsque le chargement s'effectue en parallèle, l'image s'affiche petit à petit, comme sur un navigateur Web
- Si ce comportement ne convient pas, on peut utiliser un **`MediaTraker`** pour attendre le chargement complet de l'image par le *thread* ; on n'affiche l'image qu'ensuite

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`**, un **`JComponent`** ou un **`JButton`**
- Avec AWT, on inclut les images dans un **`Canvas`**

Désigner une image

- Désigner une image locale par l'emplacement de son fichier dans l'arborescence ne marche pas si l'image est dans un fichier jar
- C'est une des erreurs les plus fréquemment rencontrées dans les projets d'étudiants
- On pourra récupérer les images qu'elles soient dans un fichier jar ou non, si on désigne l'image comme une ressource en utilisant la méthode **`getResource`**

Exemple : obtenir une `ImageIcon` comme une ressource

- On délègue au chargeur de classes la recherche du fichier qui contient l'image :

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

- Le chargeur de classe recherche l'image selon son algorithme de recherche, typiquement, à partir du *classpath* si `nomFichier` est un nom absolu, ou à partir de la classe si le nom est relatif

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 nécessaire d'utiliser **`BufferedImage`**

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`

```
URL url =
  getClass().getResource("/img/image.gif");
Icon icone = new ImageIcon(url);
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) { . . . }
```

Image

Chargement en tâche de fond

- Les méthodes **`getImage`** des classes **`Applet`** ou **`Toolkit`** permettent de lier une instance de **`Image`** à une image ; elles ne chargent pas l'image en mémoire
- Ces méthodes retournent immédiatement ; elles ne renvoient pas d'erreur si l'image n'existe pas
- L'image sera ensuite chargée petit à petit par un *thread* de chargement lorsqu'on demandera son affichage

Obtenir une `Image` dans une application

- ```
  Image image =
    Toolkit.getDefaultToolkit()
           .getImage("uneImage.gif");
  ```
- `getImage` prend en paramètre un nom de fichier ou un URL
- **Rappel : ne marche pas si l'image est dans un jar**
  ```

Obtenir une image comme une ressource

- Même principe que pour une **`ImageIcon`** : on utilise le chargeur de classe (**`getClass().getResource(…)`**)
- On utilise ensuite **`java.awt.Toolkit.getImage(URL url)`** pour récupérer l'image avec l'URL fourni par le chargeur de classe

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(),
                 "image1.gif")
```

ou `getDocumentBase()`

- 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

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

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

Affichage par `drawImage`

- La méthode **`drawImage()`** retourne immédiatement, même si l'image n'est pas entièrement disponible
- 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) :

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

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 (peut être utile pour les images gif animées)

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

Mettre à jour une image

- Pour des raisons de performances les images sont gardées dans des caches par le système d'affichage
- Si l'image est modifiée entre 2 affichages, il peut être nécessaire d'appeler la méthode **`flush`** :

```
image.flush();
```

BufferedImage

- Classe fille de **`Image`**, elle permet d'effectuer des traitements complexes sur les images
- Elle ne sera pas traitée en détails dans ce cours

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();
```

Passer de **Image** à **BufferedImage**

```
Image img = …;
BufferedImage bi = new BufferedImage(
    img.getWidth(null),
    img.getHeight(null),
    BufferedImage.TYPE_INT_RGB);
Graphics g = bi.getGraphics();
g.drawImage(img, 0, 0, null);
```

javax.imageio

Paquetage `javax.imageio`

- Ce paquetage permet de charger et sauvegarder facilement n'importe quel type d'image
- Il utilise la notion de *plugin* pour supporter les différents types d'image
- Dans l'API Java de base depuis le JDK 1.4

Classe `ImageIO`

- Classe du paquetage **`javax.imageio`**
- Méthodes **`static read`** pour lire depuis un fichier local (File), un flot ou un URL (lecture directe sans utilisation d'un *thread* en parallèle)
- Méthodes **`write`** pour écrire un **`BufferedImage`** dans un fichier ou un flot, en choisissant le codage d'écriture

Codages supportés par imageio

- En lecture les formats gif, png et jpeg sont supportés par l'API
- En écriture, seuls les formats png et jpeg sont supportés
- On peut trouver des paquetages sur le Web pour ajouter d'autres formats
- Obtenir les formats supportés en lecture :

```
String[] formatNames =
   ImageIO.getReaderFormatNames();
```

Exemples pour `ImageIO`

```
URL url =
  getClass().getResource("/img/image.gif");
BufferedImage bi = ImageIO.read(url);
. . .
String nomFichier = "http://.../truc.gif";
bi = ImageIO.read(new URL(nomFichier));
```

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) {
       . . .
    }
  };
. . .
JButton bPrecedent = new JButton(actionPrecedent);
panel.add(bPrecedent);
bPrecedent.setText(""); // bouton "image" avec tooltip
```

Définition de l'action

Utilisation de l'action

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

Interfaces graphiques dynamiques

- Il s'agit d'interfaces graphiques qui sont modifiées après un premier affichage
- Ces modifications sont le plus souvent engendrées par les actions de l'utilisateur
- Elles peuvent aussi être provoquées par le chargement d'un certain type de fichier ou par le passage à une nouvelle étape du traitement

Moyens à utiliser

- Modifier l'interface graphique à l'aide des méthodes **`add`**, **`remove`** ou **`removeAll`** de la classe **`Container`**
- Ensuite, faire afficher la nouvelle interface par **`repaint`**
- Le plus souvent on devra envoyer des **`revalidate`** aux composants qui ont changé d'aspect et des **`validate`** sur les containers à qui on a ajouté ou retirer des composants

Démo

- On récupère une démo (avec le code source) des tous les composants quand on récupère le JDK ; elle est à l'adresse **`jdk1.5.0\demo\jfc\SwingSet2`**

Java Look and Feel Repository

- Sun fournit un ensemble d'icônes que l'on peut utiliser dans ses propres interfaces graphiques à l'adresse suivante :
  `http://developer.java.sun.com/developer/techDocs/hi/repository/`