IHM

Interfaces Graphiques en Java

Frédéric MOAL
Université d'Orléans

2011/2012

IHM : Plan général

Objectifs du module IHM :

- Compréhension des architectures Modèle Vue Contrôleur
- Maîtrise du développement et de la maintenance d'IHM pour les architectures clients légers et clients lourds

Programme :

- Programmation des interfaces graphiques en client « lourd » (exemple de Java/SWING), MVC
- Architectures des interfaces Web (JSP/servlets, exemples de Struts, JSF…), MVC 2
- Utilisation des frameworks Javascript (exemples de jQuery, Wicket, GWT)
- Les interfaces des terminaux portables (exemple d'Android)

Pré-requis :

Programmation Java, maîtrise de la programmation orientée objet

2

IHM partie 1

Interfaces Graphiques en Java/SWING

Frédéric MOAL
Université d'Orléans

2011/2012

Plan

1. Généralités sur les interfaces graphiques
2. Affichage d'une fenêtre
3. Classes de base ; AWT et Swing
4. Placer des composants dans une fenêtre
5. Gestion des événements

Ressources

- **LE Cours :**
  Richard GRIN, Université de Nice
  http://deptinfo.unice.fr/~grin/messupports/index.html
- **Docs/Tutos ~~SUN~~ Oracle de Java :**
  http://download.oracle.com/javase/ tutorial/index.html
  dans la version du JDK installée sur votre machine (java –version)
- **Le site officiel de ~~Sun~~ Oracle**
  http://java.oracle.com
- **Livres**
  Java - la synthèse, 4ème édition, Clavel…
  Penser en Java vX
  Java, tête la première, O'reilly
  Au cœur de java 2, vol. 1, Sun Microsystems Press
- **Indispensable :**
  http://www.developpez.com, l'onglet java
  http://java.dzone.com, news, tutos, articles

5

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 (Graphical User Interface)

6

Programmation d'interface graphique

- L'utilisateur peut interagir à tout moment avec plusieurs objets graphiques

  cliquer sur un bouton,

  faire un choix dans une liste déroulante,

  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

7

Prog. 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

8

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

9

La solution Java : les écouteurs

Le JDK utilise une architecture de type « observateur - observé » :

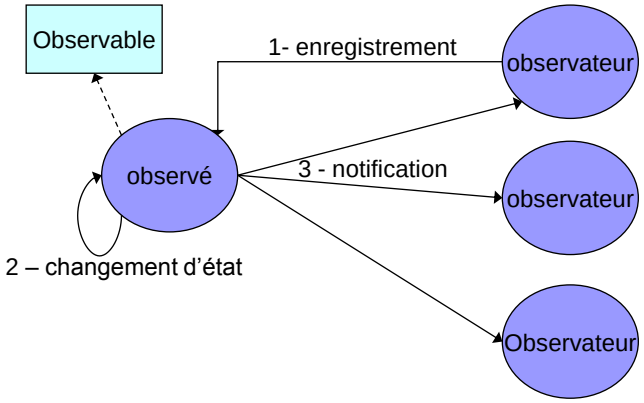
– les composants graphiques (comme les boutons) sont les observés

– 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 (eg: clic de souris)

10

La solution Java : les écouteurs

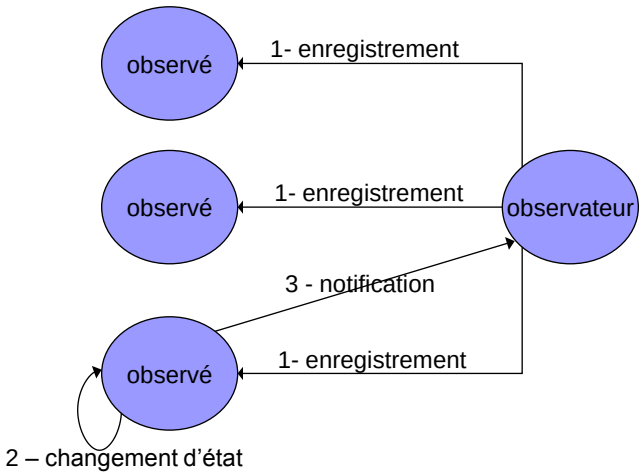
« observateur - observé »



11

La solution Java : les écouteurs

« observateur - observé »



12

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

13

Les API utilisées pour les interfaces graphiques en Java

14

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

15

Swing ou AWT ?

- Tous les composants de AWT ont leurs équivalents dans Swing
  - en plus joli
  - avec plus de fonctionnalités

Mais Swing est + lourd et + lent que AWT….

- Swing offre de nombreux composants qui n'existent pas dans AWT

➡ Il est fortement conseillé d'utiliser les composants Swing

[cours centré sur Swing]

16

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

17

Afficher une fenêtre

18

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 (annule setSize)

ou show()

19

Taille/position 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)

20

Taille/position d'une fenêtre

```
import 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);
   . . .
}
```

setLocationRelativeTo(null)
centre une fenêtre sur l'écran

21

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 suivant :

➡

22

Afficher une fenêtre (2ème façon)

```
import javaw.swing.SwingUtilities;
. . .
private static void afficherGUI() {
  JFrame frame = new JFrame("Titre");
  frame.pack();
  frame.setVisible(true);
  frame.setAlwaysOnTop(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**)

Ecran de démarrage

- Depuis JDK 6, affichage d'une image en attendant l'affichage de la fenêtre

- `java –splash:image.jpg Main`
- Dans un jar, on indique le splash dans le MANIFEST du jar :

```
Manifest-Version: 1.0
Main-Class: Test
SplashScreen-Image: image.gif
```

25

Composants lourds et légers
Classes Container et JComponent

26

Composants lourds et légers

- 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

27

Composants lourds et légers

- AWT utilise les widgets du système d'exploitation pour ses 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 dans ces containers lourds, par du code « pur Java »
- Attention, les composants lourds s'affichent toujours au-dessus des composants légers

28

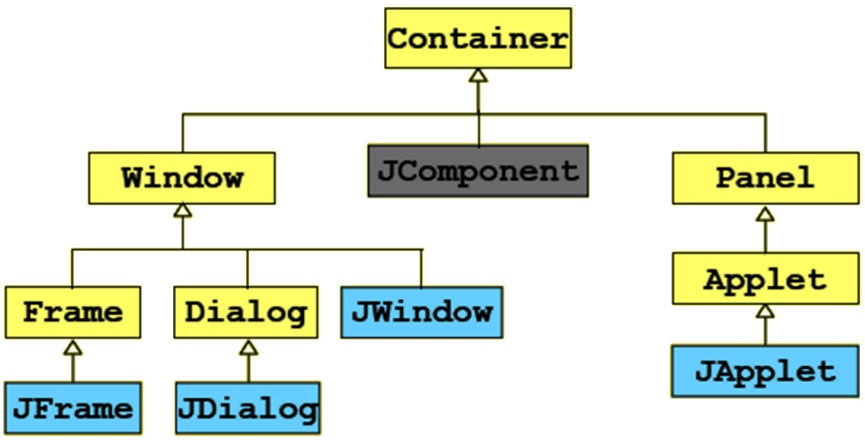
Containers lourds

- 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 : créer un (ou plusieurs) container lourd et placer à l'intérieur les composants légers de l'interface graphique

29

Héritage des containers lourds



30

Libérer les ressources

- 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

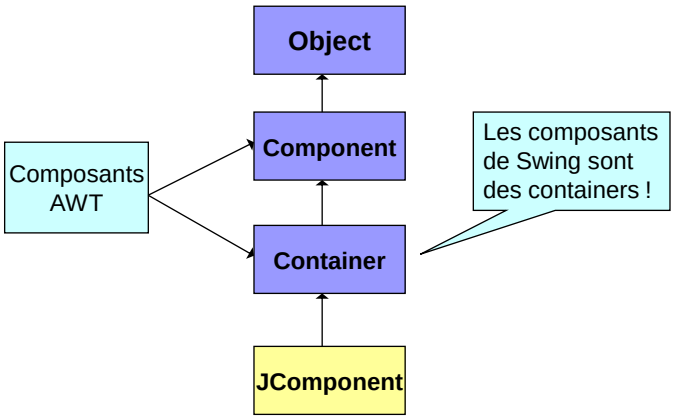
31

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

32

Classe abstraite **JComponent**



33

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**)

34

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**)

35

Ajouter des composants dans une fenêtre

36

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 :

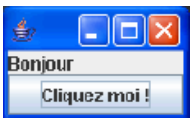
  (**topLevel** est un container lourd ; **JFrame** par exemple)

  Container contentPane =
  topLevel.getContentPane();

37

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);
   . . .
}
```

38

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)

39

Gestionnaires de mise en place

40

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

41

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)

42

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

43

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)**

44

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**

45

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

46

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());

47

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**

48

Layout manager par défaut

- 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*)

49

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

50

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**

51

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

52

Placement complexe

Pour disposer les composants d'une fenêtre de structure graphique complexe on peut :

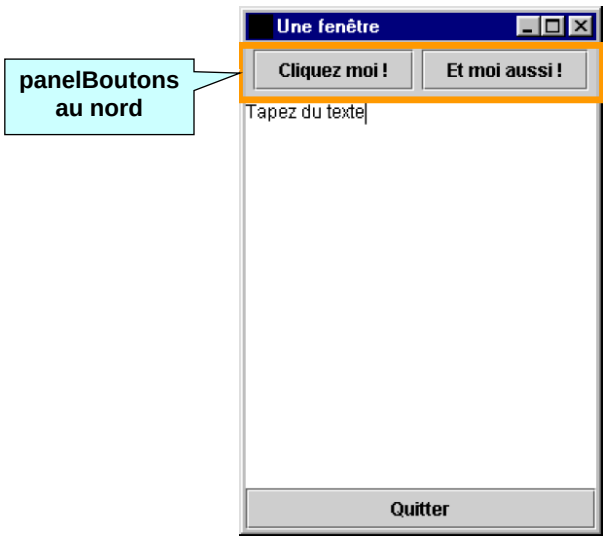
– utiliser des containers intermédiaires, ayant leur propre type de gestionnaire de placement, et pouvant éventuellement contenir d'autres containers

– utiliser un gestionnaire de placement de type **GridBagLayout** (plus souple mais parfois plus lourd à mettre en oeuvre)

– mixer ces 2 possibilités

53

Utiliser un **JPanel**



54

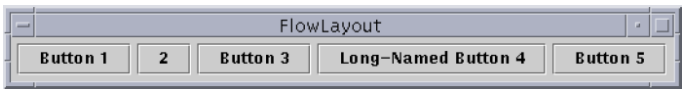
Utiliser un **JPanel**

```
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);
   . . .
}
```

55

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



56

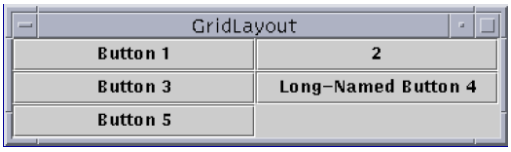
FlowLayout

```
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);
```

57

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)



58

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 !)

59

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

60

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);
```

61

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**

62

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

63

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

64

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**)

65

Quand faut-il choisir GridBagLayout ?

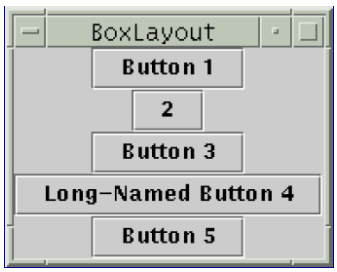
- Dès que l'interface devient trop compliquée, il est souvent plus simple d'utiliser **GridBagLayout** plutôt que de trouver des constructions très rusées avec des containers intermédiaires

- On peut rendre le code plus lisible avec des méthodes qui facilitent l'affectation des contraintes

66

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**



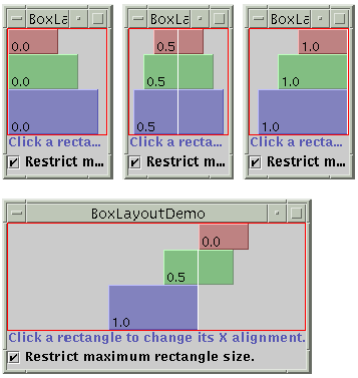
67

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}**

68

Alignment{X|Y}



- Constantes de **Component** :

  {LEFT|CENTER|RIGHT}_ALIGNMENT

  {TOP|BOTTOM}_ALIGNMENT
- Méthodes :

  setAlignmentX et **setAlignmentY**

69

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

70

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

71

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))
```

72

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);
```

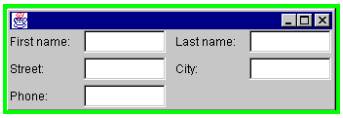
73

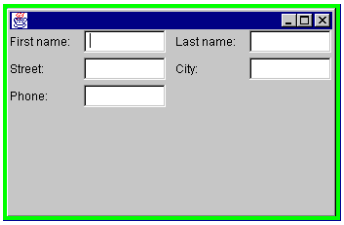
CardLayout

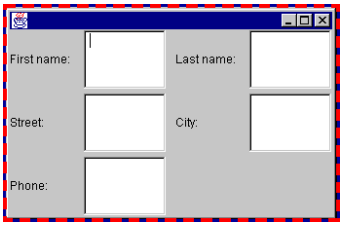
- **CardLayout** : affiche un seul composant à la fois ; les composants sont affichés à tour de rôle
- Ce *layout manager* est plus rarement utilisé que les autres
- **JTabbedPane** est un composant qui offre le même type de disposition, en plus simple mais plus puissant, avec des onglets

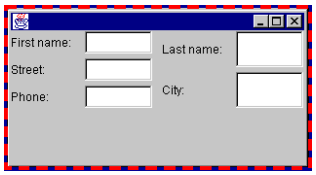
74

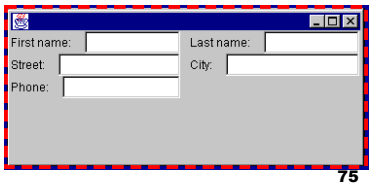
Exemples et contre-exemples











75

GroupLayout

- Depuis JDK 6
- Dissociation le placement vertical et horizontal
- Proposé par des outils de création de GUI (Matisse, Netbeans…)
- Complexe et difficilement lisible

76

Traiter les événements : les écouteurs

77

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

78

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

79

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

80

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

81

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

déclenchent une action

choix dans une liste, une action dans une boîte à cocher

composant déplacé, retaillé, caché ou montré

82

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

83

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

84

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**

85

Événements étudiés ici

On étudiera principalement :

– les événements **ActionEvent** qui conduisent à des traitements simples (écouteur **ActionListener**)

– les événements **KeyEvent**, au traitement plus complexe (écouteur **KeyListener** et adaptateur **KeyAdapter**)

86

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

87

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

88

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*

89

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

90

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**

91

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)

92

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

93

Exemples d'adaptateurs

- Les classes suivantes du paquetage **java.awt.event** sont des adaptateurs :

  KeyAdapter, **MouseAdapter**, **MouseMotionAdapter**, **FocusAdapter**, **ComponentAdapter**, **WindowAdapter**

94

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);
        }
   });
   . . .
}
```

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

95

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**

96

Paquetage **java.awt.event**

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

97

Écriture d'un écouteur

98

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.getClickCou
    nt() + "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

- 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

- 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 solution 1 - suite

5- Gestion des évènements

```
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
    }
    . . .
}
```

109

Variante de la solution 1

5- Gestion des évènements

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

110

Exemple variante solution 1

5- Gestion des évènements

```
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
}
```

111

ActionCommand

5- Gestion des évènements

- 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**

112

Solution 2 : classe interne

5- Gestion des évènements

- 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 – solution 2

5- Gestion des évènements

```
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 – solution 2

5- Gestion des évènements

```
// 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
        }
    }
}
. . .
```


Solution 3 : classe interne anonyme

5- Gestion des évènements

- 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 – solution 3

5- Gestion des évènements

```
JButton ok = new JButton("OK");

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

117

Evènements clavier

5- Gestion des évènements

118

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

5- Gestion des évènements

- 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**

121

KeyListener

5- Gestion des évènements

```
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)

122

Exemple de **KeyListener**

5- Gestion des évènements

```
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();
  }
```

123

5- Gestion des évènements

Evènements souris

124

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**

5- Gestion des évènements

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

127

Roulette de la souris

5- Gestion des évènements

- 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 **ScrollPane** a un tel écouteur, ce qui permet de déplacer la vue en utilisant la roulette de la souris

128

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
(version SDK 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

5- Gestion des évènements

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

Les touches de déplacement :

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

137

politique de déplacement

5- Gestion des évènements

- 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)

138

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

5- Gestion des évènements

- 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

141

Swing n'est pas « *thread-safe* »

5- Gestion des évènements

- Pour des raisons de performance et de facilité de mise en oeuvre 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

142

Modifier un composant

5- Gestion des évènements

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***)**

143

Utiliser d'autres *threads*…

5- Gestion des évènements

- **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

144

Utilisation de **invokeLater**

5- Gestion des évènements

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

145

Utilisation de **invokeLater**

5- Gestion des évènements

```
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

146

invokeAndWait

5- Gestion des évènements

- 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()

147

Classe **SwingWorker**

5- Gestion des évènements

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

148

SwingWorker<T,V>

- JDK 6
- **SwingWorker** comprend la méthode abstraite **doInBackground**
- Cette méthode devra être définie dans une classe fille
- Elle sera exécutée en parallèle au TDE lorsque la méthode **execute** sera lancée
- **T** : type du résultat final renvoyé par la méthode **doInBackground()** et récupéré avec la méthode **get(), V** : type des résultats intermédiaires renvoyés par la méthode **doInBackground()**

149

SwingWorker : utilisation

```
class Tache extends SwingWorker<Integer,
  Integer>() {
@Override
public Integer doInBackground() {
  int val = 0; boolean fini = false;
  while (!isCancelled && !fini) {
  val += calculLong();
  publish(val);
  fini = ...;
  }
  return val;
}
```

150

SwingWorker : utilisation

```
@Override
public void done() {
  int val = get();
  // Met à jour le GUI avec cette valeur
  ...
}
@Override
public void process(List<Integer> vals) {
  // Met à jour le GUI avec les valeurs
  ...
  }
} // end of class Tache
```

151

SwingWorker : utilisation

```
// Utilisation du SwingWorker dans le GUI
public void actionPerformed(ActionEvent e) {
  Tache tache = new Tache();
  tache.execute();
}
```

152