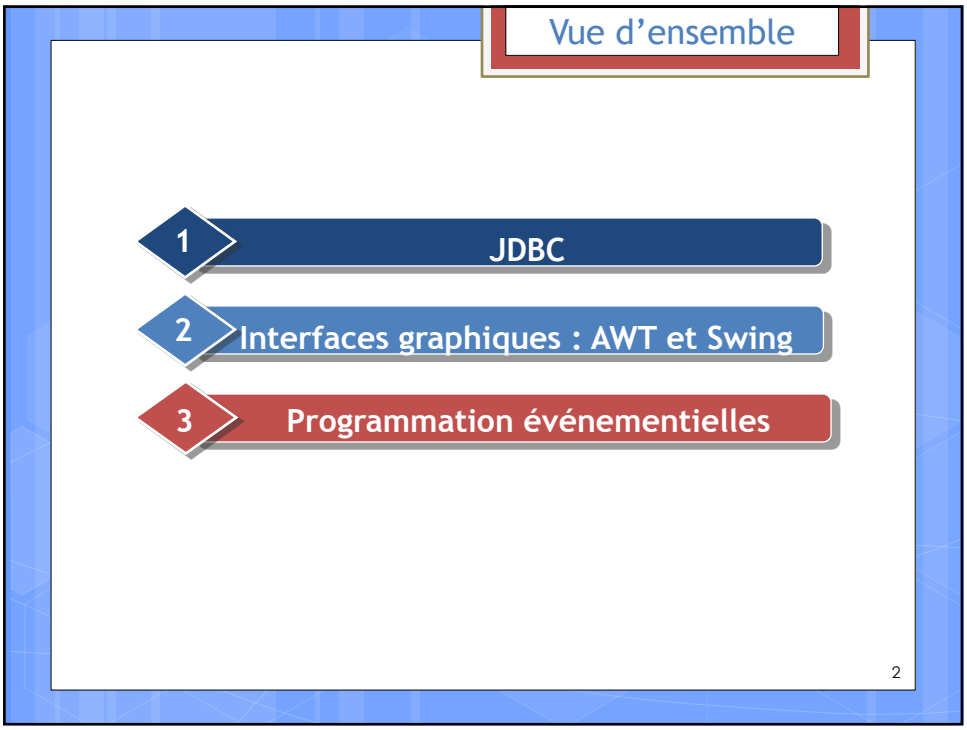

Java
Introduction à Java
Partie 3
(GUI & JDBC)
GCR1
I. AZAIEZ
Avril 2024

1


Vue d'ensemble
1 JDBC
2 Interfaces graphiques : AWT et Swing
3 Programmation événementielles
2

2

JDBC : ACCÈS BASE DE DONNÉES

3

- La technologie jdbc de Oracle (Java DataBase Connectivity) regroupe des classes permettant à une application de s'interfacer avec des bases de données relationnelles
- Il est nécessaire d'importer en début de fichier "java.sql.*" pour pouvoir utiliser les méthodes indispensables.
- La connexion application/base peut être directe ou peut se faire par l'intermédiaire d'un serveur de base de données dédié (ce qui permet des contrôles d'erreurs).
- Pour que l'application puisse se connecter à la base, un driver (pilote) est indispensable afin que les deux éléments puissent utiliser un langage commun et se comprendre (le sql n'est qu'un langage d'interrogation des bases: il ne peut servir à la connexion).

4

JDBC

- Nous citons deux types de catégories de drivers parmi ce qui existe:
  - Type 1: les passerelles JDBC-ODBC convertissent les appels jdbc en appels odbc. Le pilote odbc de la plateforme se charge ensuite de la connexion à la base.
  - Type 4: pilotes JDBC pur java (qui peuvent être implémentés directement par les SGBD: systèmes de gestion de bases de données: le système qui permet de gérer les bases, de les interroger, etc. C'est avec lui que l'on communique). Le driver se connecte directement au SGBD et les appels JDBC sont reconnus par le SGBD.

5

5

JDBC

- S'interfacer avec une base peut être délicat. il convient d'être méthodique.
- Prenons l'exemple d'une base MySql. La connexion classique se fait via un connecteur jdbc.
- il faut :
  - Créer la base, évidemment
  - Ajouter le connecteur jdbc spécifique au SGCBD aux librairies externe (cas de eclipse).

6

6

JDBC

- L'exploitation d'une base de données par une application java repose sur certaines étapes bien définies.
- Première étape: charger le driver (en fait, on charge une classe permettant de dialoguer avec le pilote) :

```
Class.forName("com.mysql.jdbc.Driver");
//Entre parenthèse, le driver est spécifié.
```

- Seconde étape: se connecter à la base. Pour cala, on créé un objet Connection de la classe du même nom. Cet objet est créé grâce à la méthode getConnection() de la classe DriverManager.

```
String url ="jdbc:mysql://localhost:port/nom_base";
Connection con=DriverManager.getConnection(url, String User, String Password);
```

7

7

JDBC

- Troisième étape: créer un objet de type *Statement*. Ce dernier possède les méthodes permettant d'interagir avec la base (c'est sur lui que l'on "applique" les requêtes). Il est créé à partir de l'objet *Connection*.

```
Statement stmt= con.createStatement();
```

  - Les éléments nécessaires à l'utilisation d'une base de données sont créés.
  - Lorsque l'usage de la base n'est plus nécessaire, il faut fermer la connexion:

```
con.close();
```

- En général, toutes les actions précitées ainsi que celles qui vont suivre sont susceptibles de lever une exception. Il est donc conseiller de disposer des blocs try...catch autour des étapes cruciales (connexion, interrogation, création de tables, etc.)

8

8

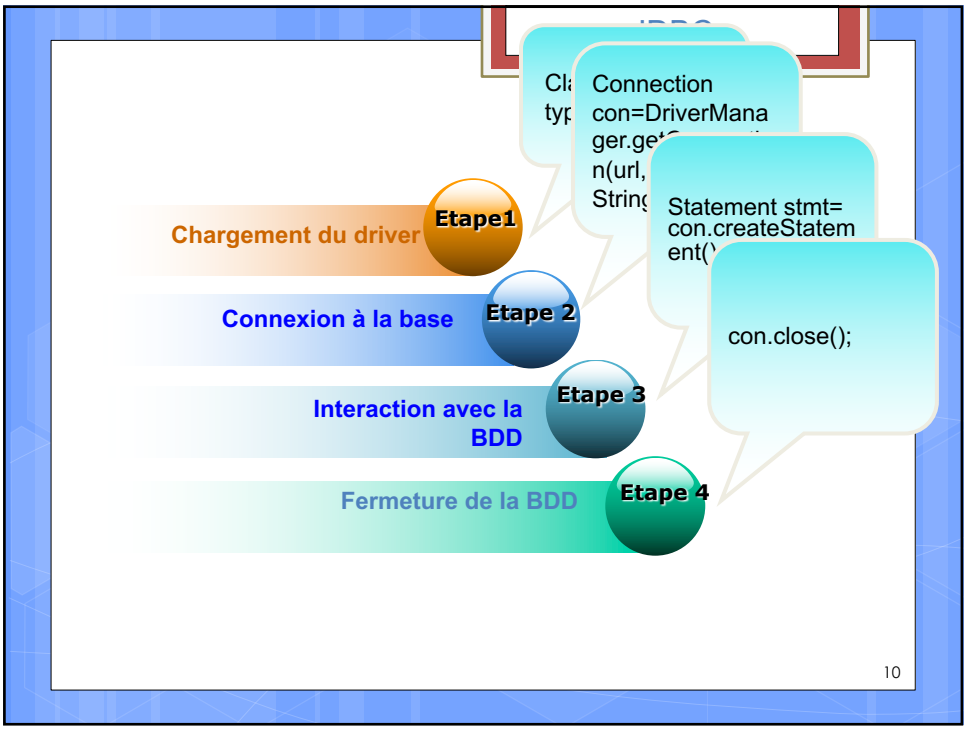
JDBC

- En fonction des requêtes, trois types de méthodes sont utilisables:
  - executeUpdate(): pour les requêtes CREATE TABLE, INSERT, UPDATE, DELETE et DROP TABLE
  - executeQuery(): permet de récupérer les enregistrements sélectionnés dans un objet de type ResultSet()
  - execute: plus rare (pour les procédures stockées).
- Création et remplissage d'une table:

```
stmt.executeUpdate("create  table pers (nom varchar2(30), age integer)");
stmt.executeUpdate("Insert into pers values('Toto',24)");
stmt.executeUpdate ("Insert into pers values('Titi',31)");
```

9

9



10

JDBC

- Interrogation d'une table (avec exemple de try…catch)

```
try {
        String maRequete="SELECT nom, age FROM pers";
        ResultSet  resultats = stmt.executeQuery(maRequete);
}
catch (Exception e){
 System.out.println("Problème lors de la récupération des données!');
}
```

11

11

JDBC

- Les résultats sont récupérés sous la forme d'un objet *ResultSet*. Il est en quelque sorte une représentation de la table qui ne contiendrait que les données désirées. Il est donc formé de colonnes, ou champs, référencés par un nom (ici deux: *"nom"* et "age"), ainsi que d'un certain nombre de lignes" contenant les données (ici, deux lignes qui concernent "Toto" et "Titi"). Les colonnes sont également désignées par un numéro (1,2,3,etc.).
- Il est alors possible de parcourir ce *ResultSet* ligne par ligne.

12

12

JDBC

- Il convient maintenant de récupérer des valeurs au sein de cet objet ResultSet.
- Pour accéder aux valeurs contenues dans l'objet, de nombreuses méthodes de ResultSet sont disponibles. En général, elles sont doublées afin de permettre d'accéder aux colonnes soit par leur nom (String) soit par leur numéro (int).

boolean getBoolean(int);	boolean getBoolean(String);
byte getByte(int);	byte getByte(String);
Date getDate(int);	Date getDate(String);
double getDouble(int);	double getDouble(String);
float getFloat(int);	float getFloat(String);
int getInt(int);	int getInt(String);
long getLong(int);	long getlong(String);
short getShort(int);	short getShort(String);
String getString(int);	String getString(String);

13

13

JDBC

- Afin d'accéder à toutes les valeurs, le Resultset est parcouru ligne par ligne.
  - Pour cela, on utilise une boucle *while* qui récupère les valeurs d'une ligne puis place le marqueur de ligne courante sur la suivante grâce à la méthode next() (au début, ce marqueur est situé avant la première ligne).
- Dans le cas présent, Si l'on veut faire un affichage des valeurs recueillies par notre requête:

```
while(resultats.next()) {
        System.out.println ("nom:"+
resultats.getString("nom")+ "et age:"
+resultats.getInt("age"));
}
```

- A noter qu'il est possible de faire "resultats.getString (1)" et "resultats.getInt(2)".

14

14

JDBC

- Résumé (gérer les exceptions si nécessaires):

```
public class EssaiJDBC {
  public static void main(String[]  args ) throws Exception {
    Class.forName("com.mysql.jdbc.Driver");
Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/MaBase","user","pswd");
    Statement stmt = con.createStatement ();
 ResultSet resultats = stmt.executeQuery ( "SELECT * FROM pers");
            while (resultats.next()) {
                        String name = resulats.getString(1);
                        int Age = resultats.getInt ("age")
                        System.out.println(name +","+Age+" ans");
                        }
              } // fin de main ()
            } //  fin de la classe
```

15

15

JDBC

PreparedStatement :

- Avec des paramètres symbolises par ? et setXXX
- Exemple

```
PreparedStatement psm = myconnexion.prepareStatement (
"UPDATE employe SET prime = prime + ? WHERE num employe =
?AND service = ?");

    psm.setInt (1, 300);
    psm.setInt (2, 123);
    psm.setString (3, "informatique");
    int myresultat = psm.executeUpdate ( );
```

16

16

JDBC

DÉMONSTRATION



17

JDBC

- Un problème se pose dans le cas où l'on ne connaît pas les structures de la base et des tables qui la composent.
- L'objet DatabaseMetaData permet de récupérer les méta données de la base: nom des tables, indexs, méthodes sql supportées par la base,...
- *Ex:*

```
DatabaseMetaData dbmd =
con.getMetaData();
```

- Quelques méthodes de la classe DatabaseMetaData:

getDriverName(): retourne le nom du driver JDBC
getDriverVersion() : retourne la version du driver
getDatabaseProductName(): nom de la base de données
getDatabaseProductVersion(): retourne la version du logiciel …

18

18

JDBC

- L'objet ResultSetMetaData récupère les informations concernant un résultat de requête. Il provient de l'application de la méthode getMetaDate() sur l'objet ResultSet.

  Il permet ainsi de connaître :

  - le nombre de colonnes: Int getColumnCount()
  - le nom d'une colonne: String getColumnName(int col)
  - le type d'une colonne: String getColumnType(int col)
  - la taille d'une colonne: Int getColumnDisplaySize(int col)

- D'autres fonctions existent:
  - boolean isReadOnly(int col)
  - boolean isNullable(int col) //accepte la valeur null
  - ...

19

19

JDBC

- Exemple

```
ResultSet resultats =
   stmt.executeQuery("SELECT....");

   ResultSetMetaData rsmd;
   rsmd = resultats.getMetaData();

   int colonnes = rsmd.getColumnCount():
   boolean Nul = rsmd.isReadOnly(1);
```

20

20

Exemple : JDBC

```
try {
String strClassName = "com.microsoft.jdbc.sqlserver.SQLServerDriver";
  String strUrl = "jdbc:microsoft:sqlserver://hostname:1433;"
                          +"user=sa;password=pass;DatabaseName=dbName";
  String strInsert = "INSERT INTO T_Users "
                          + "(Login, Password, ConnectionNumber) "
                          + "VALUES ('Toto', 'Titi', 0);";
  Class.forName(strClassName);
  Connection con = DriverManager.getConnection(strUrl);
  Statement stAddUser = con.createStatement();
  stAddUser.executeUpdate(strInsert);
  con.close();
} catch(ClassNotFoundException e) {
  // . . . } catch(SQLException e) { // . . . }
```

21

21

Interfaces graphiques

AWT ET SWING

22

Interfaces graphiques

L'une des préoccupations principales des concepteurs du langage Java a été de fournir aux développeurs des moyens de mettre en place facilement une interface graphique au sein des applications :

- **GUI: Graphic User interface**

Les interfaces homme machine ont pour finalité une interaction rapide, facile et ergonomique entre l'utilisateur et lignes de commandes afin de transcrire visuellement des informations vers et depuis la machine (commandes, entrées-sorties, messages, avertissements,…).

Une GUI est composée d'élément graphiques (composants), généralement interactifs (le but restant la communication homme-machine).

La programmation est dite "évènementielle" car elle repose sur l'analyse des évènements (click sur un bouton, choix dans une liste, etc.) et sur l'implémentation des actions à effectuer lorsque l'un de ces évènements survient.

23

23

Interfaces graphiques

- Les possibilités offertes par Java en termes d'interfaces graphiques sont extrêmement importantes.
- Le langage fournit des éléments graphiques préexistants, ainsi que toutes les méthodes pour les gérer, les modifier ou faire de la programmation événementielle.
- Il autorise également à créer ses propres éléments graphiques.
- Enfin, il met à disposition des fonctionnalités plus générales comme la possibilité de dessiner des formes.
- Il est conseiller de se référer le plus souvent possible au site de Oracle tant les bibliothèques graphiques sont conséquentes et complètes.

24

24

Interfaces graphiques

AWT est le premier essai des concepteurs de java pour offrir un ensemble de classes visant à la conception d'une GUI.

- AWT: Abstract Windowing Toolkit
- Les fenêtres, boutons et autres composants graphiques sont construits autour de directives propres au système d'exploitation. Les composants en jeu utilisent donc les ressources de l'OS: composant dits "lourds".
- Des défauts persistants et surtout un relatif "manque" d'efficacité par rapport aux résultats désirés ont amené l'apparition d'un nouvel ensemble de classes appelé SWING (depuis le jdk 1.2), beaucoup plus complet et optimisé.

A la différence d'AWT, les composant Swing sont des composants "légers". Ils sont purement java (pas d'utilisation des directives du système d'exploitation).

- Les composants swing sont donc plus portables que les composants AWT. Ils sont également plus faciles à utiliser et plus robustes.

25

25

Interfaces graphiques

Swing introduit également la possibilité de modifier le *look and feel* des composants :

- c'est-à-dire leur aspect général (couleurs, polices graphismes, etc.). Si l'aspect par défaut des composant est le *look and feel metal*, il est tout à fait possible de faire en sorte que ces derniers affichent un autre aspect ou le look général de l'OS sur lequel tourne l'application, comme si les composants avaient été programmés spécifiquement pour ce système (Linux,Windows, ...).

Les composants AWT restent tout de même utilisés, notamment dans les applets (on utilise de plus en plus des JApplets construites à partir de Swing).

De plus, le package AWT est indispensable pour l'utilisation des gestionnaire de placement (layout) et pour les gestions d'événement. Swing ne remplace donc pas totalement AWT (en revanche, il fournit une nouvelle version plus légère et efficace de presque tous les composants, tels les boutons, panels, etc.)

26

26

Interfaces graphiques

D'une manière générale, on distingue d'un point de vue syntaxique les composants swing des composant AWT par un "J" placé en début de nom.

Ainsi, le Button de AWT possède son équivalent swing JButton, Panel devient JPanel, etc.

Un import de package est nécessaire pour utiliser les éléments de AWT et/ou de swing:

- import java.awt.*; // pour les composants AWT, ainsi //que les gestionnaires de placement.
- import javax.swing.*; // pour les composants swing (ne // pas oublier le x de javax)

27

27

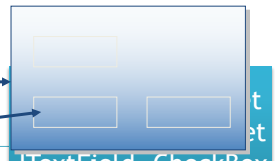
Composants graphiques

On distingue deux types de composants:

Des composants de hauts niveaux, ou Containers, principalement destinés à recevoir eux-mêmes d'autre composants et des gestionnaires de placement.

Exemples: Frame et JFrame , Panel et JPanel

Des composants de niveaux inférieurs typiquement les composants usuels disposés au sein des containers.



JTextField, CheckBox et JCheckBox

28

28

Les containers

- Les containers les plus usités sont les Frame et les Panel (et leurs homologues en swing). Ce sont aux containers que sont associés les gestionnaires de placement (layout).
- Hiérarchies de classes: la classe *Container* possède plusieurs sous-classes, dont *Window*, *Panel* et *ScrollPane*. De *Window* sont dérivées les classes *Frame*(et sa sous-classe *JFrame*) et *Dialog* (et ses sous classes *FileDialog* et *JDialog* entre autres). Enfin, il est noté que de *Panel* est dérivée la classe *Applet* (et sa sous-classe *JApplet*).
- un *ScrollPane* est un Container avec barres de défilement ("ascenseurs").

29

29

Les containers

- Comme elles héritent de Window, Frame et Dialog instancient des "fenêtre" (en fait, des "cadres" et des "boites de dialogue"). Ainsi, une Frame et une fenêtre.
- Ces classes "fenêtre" gèrent notamment les barres de titre et de menu
- A noter qu'à la différance des autre composants Swing, les Containers JWindow, JFrame, JDialog et JApplet ne sont pas pur Java: ils dérivent de Window (java.awt) et utilisent par conséquent le système natif de fenêtre de l'OS.

30

30

Composants graphiques

JFrame - Constructeurs et méthodes (non exhaustif):

- JFrame()
- JFrme (String titre)
- String getTitle ()
- void setTitle(String title) // Fixe le titre de la fenêtre
- void setResizable (boolean b) // Rend la fenêtre redimensionnable
- boolean isResizable()
- void pack() // réduit la fenêtre à la taille de ses composants
- void show() // Indispensable: rend la fenêtre visible (sinon, hide)
- void toFront() // Place la fenêtre au premier plan
- void toBlack() // Place la fenêtre en dessous des autres
- void setJMenuBar(JMenuBar J) // place la barre de menu
- JMenuBar getMenuBar() // Retourne la barre de menu (ou null)
- void repaint() // rafraîchissement de la fenêtre
- void setBackground(Color c) // Définit la couleur de frond
- Color getBackgound()
- void setSize(int width, int height) // Dimensionne la fenêtre
- Dimension getSize()
- void setVisible(boolean b) // Idem show()
- void requestFocus() // La fenêtre demande le focus

31

31

Composants graphiques

Les composants: Ils sont très variés. Entre autres:

- Button et JBoutton: bouton à presser. Il est possible de fixer leur texte, mais également de leur attribuer une image de fond.
- Label et JLabel: étiquette. Texte non modifiable par l'utilisateur de l'interface graphique.
- TextField et JTextField: texte d'une ligne sur lequel peut agir l'utilisateur. Affichage ou saisies d'informations.
- TextArea et JTextArea: idem, mais sur plusieurs lignes.
- Checkbox et JCheckbox: case à cocher (booléen)
- RadioButton et JRadioButton: ensembles de cases à cocher dans lequel un seul choix est possible à la fois.
- JProgressBar: barre de progression
- Choice ou JcomboBox: choix dans une liste déroulante.
- List et JList: choix parmi des éléments listés (choix multiples possibles).
- JSlider: barre avec curseur permettant un choix de valeur numérique.

http://download.oracle.com/javase/tutorial/uiswing/TOC.html#%20components

32

32

Composants graphiques

Quelques méthodes

- JButton:
  - JButton()
  - JButton (String texte)
  - JButton (Icon icone)
  - // avec ImageIcon icone =createImageIcon ("repImages/logo.gif");
  - JButton (String texte, Icon icon)
  - boolean isDefaultButton()
  - void setText (String texte)
- JLabel:
  - JLabel (String texte)
  - JLabel (setTextIcon image)
  - void setText(String texte)
  - String getText ()

33

33

Composants graphiques

- **JTextField**
  - JTextField ()
  - JTextField (String texte)
  - JTextField (int colonne)
  - String getText() et setText(String texte)
  - …
- **JCheckBox**
  - JCheckBox ()
  - JCheckBox (String text)
  - JCheckBox (String texte, Icon icone, boolean select)
  - void setSelected(boolean b)
  - boolean isSelected()
  - …

34

34

Composants graphiques

- **JComboBox**

```
JComboBox()
JComboBox ( Object[] tab)  //ComboBox avec les éléments
                           // contenus dans le tableau tab
JComboBox (Vector v)       // idem avec vecteur
void addItem (Object obj)
Object getItemAt (int indice)
int getItemCount ()
int getSelectedIndex ()          // retourne l'indice de
                                 // l'élément sélectionné
Object getSelectedItem ()    // retourne l'élément sélectionné
void setSelectedIndex (int indice) // sélectionne l'élément
                                   // à l'indice indice
void setSelectedItem (Object obj) // fixe l'élément à la
                                  // position courante
```

35

35

Composants graphiques

- **Jlist**

```
JList()
JList ( Object[] tab)        //Liste avec les éléments
                             // contenus dans le tableau tab
JList ( Vector v)            // idem avec un vecteur
Object getSelectedValue () // retourne l'élément
                           // sélectionné sinon null
Object[] getSelectedValues () // retourne les élément
                              // sélectionnés
```

- **JSlider**

```
JSlider ()          // slider de 0 à 100, valeur intiale à 50
JSlider ( int min, int max, int defaut) // slider entre m
                    // et n positionné à défaut
int getValue () // retourne la valeur du curseur
void setValue ( int n) // positionne le curseur
```

36

36

Composants graphiques

La manière de coder des interfaces graphiques est toujours la même:

- création d'un ou plusieurs container(s) (au moins une fenêtre)
- implémentation d'un composant
- ajout d'un composant à un container
- implémentation d'un composant
- et ainsi de suite…

Ne pas oublier de rendre la fenêtre visible!

37

37

Interface graphique

De manière plus précise, avec des composants AWT, il suffit de créer un frame, puis un composant et d'ajouter le second à la première via une méthode add():

```
import java.awt.*;
public class Essai {
    Frame f = new Frame ();              // container
    Button bouton1 = new Button();       //  composant
    public Essai() {                     //  constructeur
            f. add(bouton1);             // ajout
            f. pack();  // donne à la fenêtre les dimensions de
                        // ses composant (ici, juste un bouton)
            f.show();  // ou f.setVisible(true); rend la fenêtre visible.
             }
}
```

38

38

Interface graphique

Il est également possible de travailler sur un Panel au lieu d'ajouter les composants directement sur la fenêtre:

```
import java.awt.*;
public class Essai {
        Frame f = new Frame ();          // container
        Panel p = new Panel ();// second Container
        Button bouton1 = new Button();
        public Essai () {
                p.add(bouton1);     // ajout du bouton au panel
                f.add(p);           // ajout du panel à la fenêtre
                f.pack();
                f.show();
                }
}
```

39

39

Interface graphique

Autre manière de coder: faire hériter la classe de Frame. Les objets instancier sont donc "sortes de fenêtres", alors que dans les cas précédents ils implémentaient une fenêtre.

```
import java.awt.*;
public class Essai extends Frame {
        Panel p = new Panel ();              // second Container
        Button bouton1 = new Button();   // composant
        public Essai() {
                p.add(bouton1);     //ajout du bouton au panel
                this.add(p); // ajout du panel à la fenêtre (this)
                this.pack();
                this.show(); }
}
```

40

40

Interface graphique

Dans le cas de l'emploi des composants swing, la manière de procéder est similaire. L'héritage utilisé dans l'exemple précédent est également possible (en faisant hériter la classe de JFrame et non de Frame).

Cependant, quelques points sont à souligner:

- les composants ne sont évidemment pas les mêmes. Ne pas oublier que leurs noms commencent par un "J".
- le package à importer n'est bien entendu pas le même non plus.

41

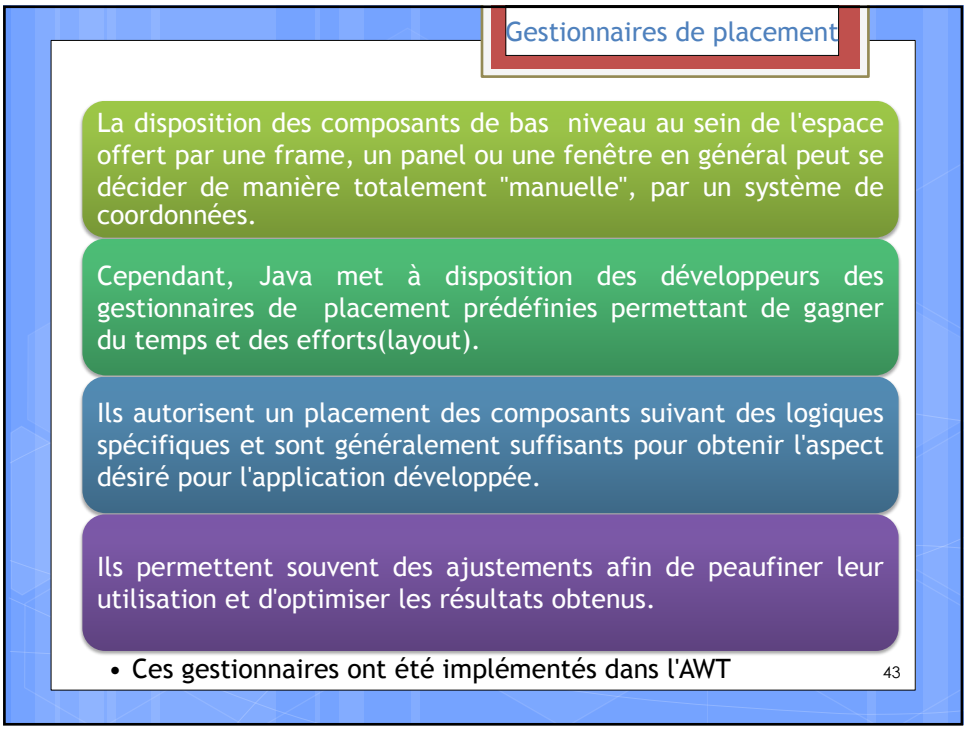
41

Interface graphique

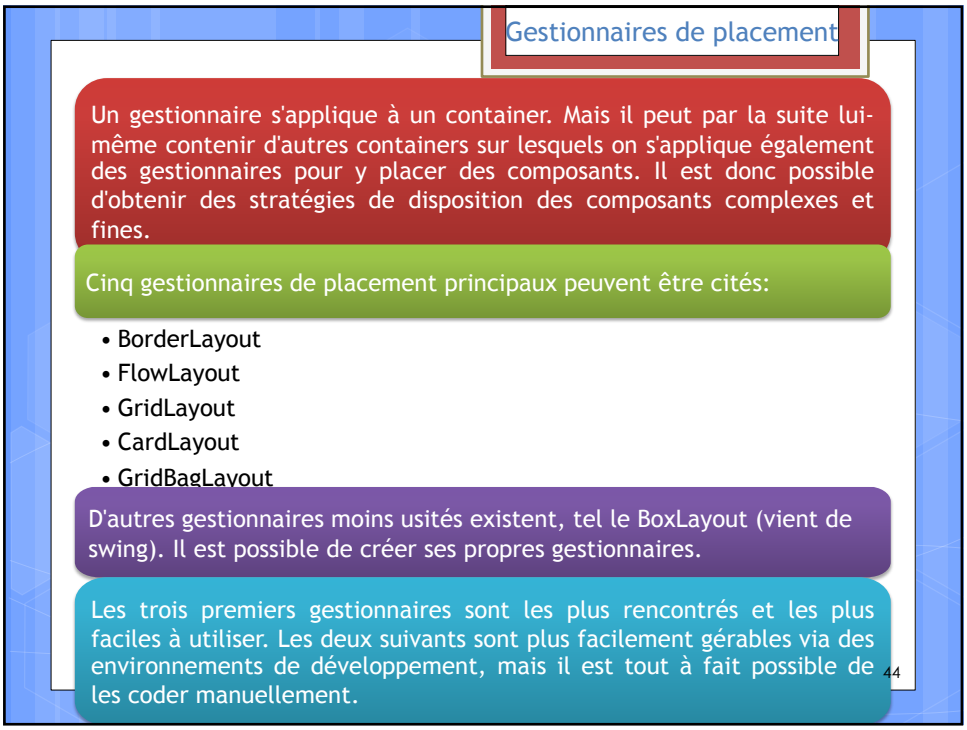
```
import java.awt.*;
import javax.swing.*;
public class Essai {
        JFrame f = new JFrame();
        JButton bouton1 = new JButton();
        public Essai () {
                f.add(bouton1);      //ajout du bouton au JFrame
                f.pack();
                f. show();
                }
}
```

42

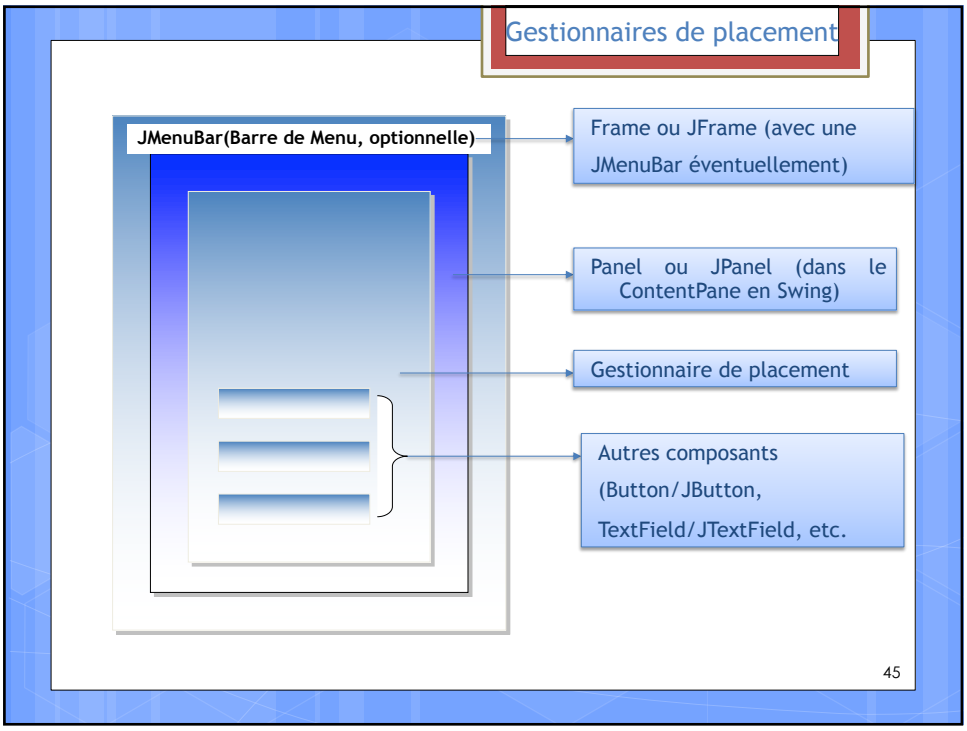
42


Gestionnaires de placement
La disposition des composants de bas niveau au sein de l'espace offert par une frame, un panel ou une fenêtre en général peut se décider de manière totalement "manuelle", par un système de coordonnées.
Cependant, Java met à disposition des développeurs des gestionnaires de placement prédéfinies permettant de gagner du temps et des efforts(layout).
Ils autorisent un placement des composants suivant des logiques spécifiques et sont généralement suffisants pour obtenir l'aspect désiré pour l'application développée.
Ils permettent souvent des ajustements afin de peaufiner leur utilisation et d'optimiser les résultats obtenus.
• Ces gestionnaires ont été implémentés dans l'AWT
43

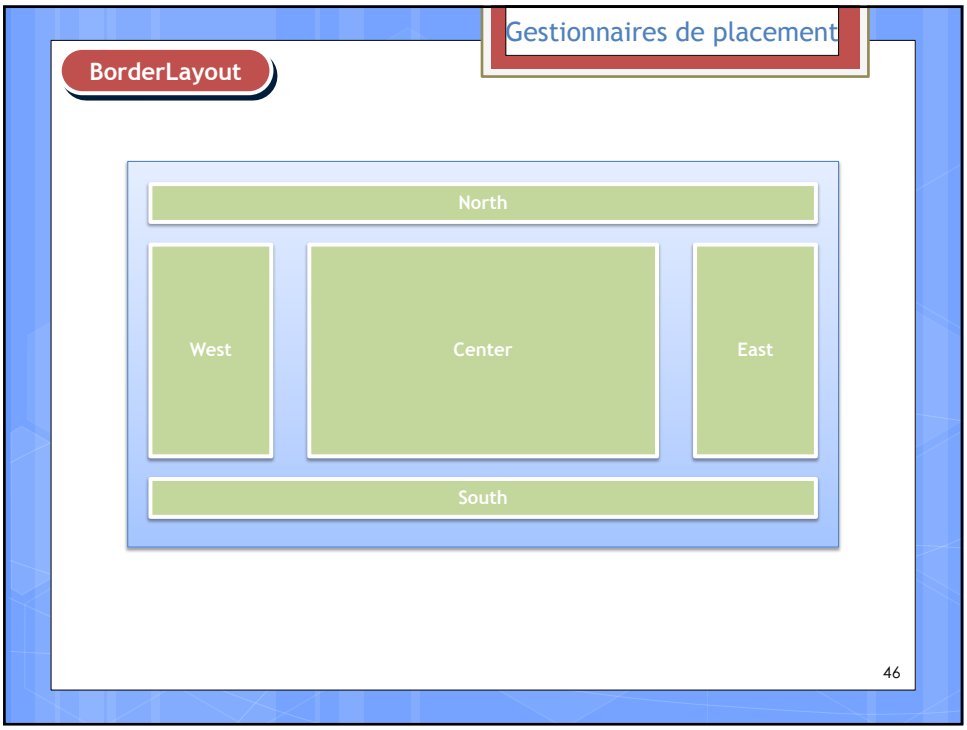
43


Gestionnaires de placement
Un gestionnaire s'applique à un container. Mais il peut par la suite lui-même contenir d'autres containers sur lesquels on s'applique également des gestionnaires pour y placer des composants. Il est donc possible d'obtenir des stratégies de disposition des composants complexes et fines.
Cinq gestionnaires de placement principaux peuvent être cités:
• BorderLayout
• FlowLayout
• GridLayout
• CardLayout
• GridBagLayout
D'autres gestionnaires moins usités existent, tel le BoxLayout (vient de swing). Il est possible de créer ses propres gestionnaires.
Les trois premiers gestionnaires sont les plus rencontrés et les plus faciles à utiliser. Les deux suivants sont plus facilement gérables via des environnements de développement, mais il est tout à fait possible de les coder manuellement.
44

44


Gestionnaires de placement
JMenuBar(Barre de Menu, optionnelle)
Frame ou JFrame (avec une JMenuBar éventuellement)
Panel ou JPanel (dans le ContentPane en Swing)
Gestionnaire de placement
Autres composants (Button/JButton, TextField/JTextField, etc.
45

45


Gestionnaires de placement
BorderLayout
North
West
Center
East
South
46

46

Gestionnaires de placement

Le BorderLayout

- C'est le gestionnaire par défaut des Frame.
- Il divise le Container sur lequel il est appliqué en 5 zones (North, South, East, West et Center). Les composants qui sont placés dans ces zones (un par zone) y occupent l'ensemble de la place disponible. Chaque zone peut elle-même contenir un container sur lequel on peut appliquer un BorderLayout (découpage très fin).

Constructeurs et méthodes (non exhaustif)

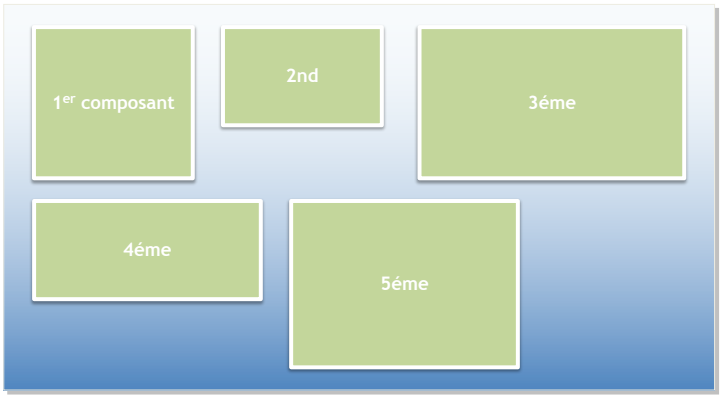
- **BorderLayout()** // constructeur par défaut
- **BorderLayout(int h, int v)** //Crée un *Borderlayout* au sein du quel les composant sont //séparés par un espace horizontal *h* et un espace vertical v
- **void LayoutContainer(Container cont)** // Applique le gestionnaire au container .
- **void addLayoutComponent(String p, Component c)** // Ajoute le composant c à la //position p (qui peut être " North", "south", //"East" , "West" ou "Center").
- **Void removeLayoutComponent(Component c)** // retire le composant *c*
- **String toString ()** // toString existe aussi pour les gestionnaires de placement .
- **Dimension minimumLayoutSize (Container cont)** // Retourne les dimensions minimum //du container cible dans lequel des composants ont été ajoutés.
- **Dimension preferredLayoutSize(Container cont)** //indique les dimensions optimales // du container (contenant des composants).

47

47

Gestionnaires de placement

FlowLayout



48

48

FlowLayout

- C'est le gestionnaire par défaut des Panel et applets.
- Il dispose les composants les uns à la suite des autres, en ligne, et change de ligne lorsqu'il atteint le bord du container.
- Constantes: CENTER, LEFT et RIGHT sont trois entiers indiquant que les composants disposés sur une ligne sont soit à centrer, soit à aligner à gauche, soit à aligner à droite.

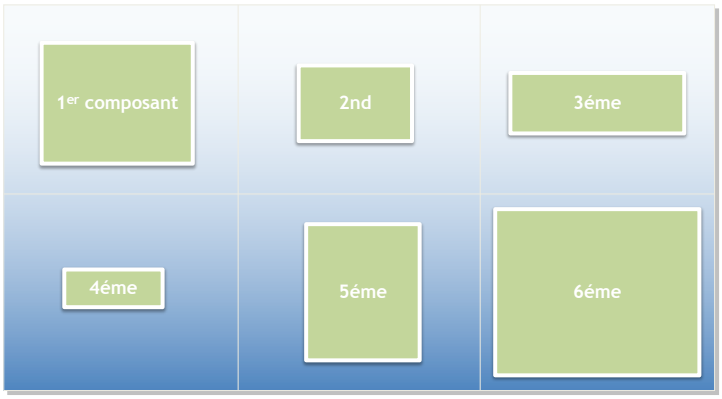
Constructeur et méthodes (non exhaustif)

- **FlowLayout ()** // Constructeur par défaut(lignes centrées et //espacements verticaux et horizontaux de 5 pixels entre les //composants)
- **FlowLayout (int alignement)** //Crée un gestionnaire dont //l'alignement est choisi par l'utilisateur (CENTER, LEFT ou RIGHT)
- **FlowLayout (int alignement, int h, int v)** //Alignement de type //alignement et espacements horizontal h et vertical v entres les //composants
- **Autres méthodes: voir le BorderLayout**

49

49

GridLayout



50

50

Le GridLayout

- Il divise le Container en un nombre donné de cases identiques. Un composant par case.

Constructeurs et méthodes (non exhaustif)

public GridLayout (int rows, int cols)
// Gestionnaire de placement de *rows* lignes et
//*cols* colonnes. Les espacements entre les
//composants est de 0 (vertical et horizontal)
public GridLayout (int rows, int cols, int h, int V)
//Espacement horizontal h et vertical v entre les
//composants

- Autres méthodes: voir le BorderLayout.

51

51

Le CardLayout

- N'affiche qu'un composant à un moment donné. Les différents composants
- sont considérés empilés, comme dans un jeu de cartes.

Constructeurs et méthodes (non exhaustif)

- **CardLayout ()** // constructeur par défaut
- **CardLayout (int h, int v)** // Espacements horizontal h et vertical v //entre les composants
- **void first (Container cont)** // Affiche le premier des composants. cont //est le container sur lequel est appliqué le gestionnaire de placement
- **void last (Container cont)** /* Affiche le dernier des composants cont est le container sur lequel est appliqué le gestionnaire de placement*/
- **void next (Container cont)** // Affiche le composant suivant cont est le //container sur lequel est appliqué le gestionnaire de placement
- **public void previous (Container cont)** // Affiche le composant //précédent. cont est le container sur lequel est appliqué le //gestionnaire de placement
- **public void show (Container cont, String n)** // Affiche le composant //associé à la chaîne n cont est le container sur lequel est appliqué le //gestionnaire
- Autres méthodes: voir le BorderLayout.

52

52

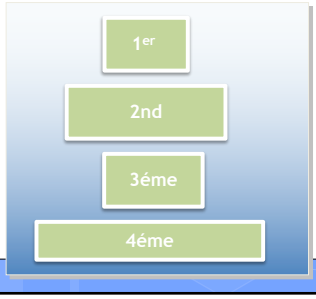
Gestionnaires de placement

Le GridBagLayout

- C'est le gestionnaire de placement le plus sophistiqué et flexible. Il permet une disposition fine des composants en proposant la mise an place d'une grille. A la différence du GridLayout, les cases de la grille peuvent avoir des dimensions variables. De plus, les composants peuvent occuper plus d'une case.

Le BoxLayout

- Il dispose les composants les uns au dessus des autres, sur une seule colonne. Il permet les alignements.



53

53

Gestionnaires de placement

Principe général d'utilisation (exemple avec deux BorderLayout):

Définir le container

Construire le gestionnaire et l'appliquer au container via la méthode setLayout()

Disposer les composants dans le container grâce au gestionnaire.

54

54

Gestionnaires de placement

```
import  java.awt.*;                    // pour les gestionnaires
import  javax.swing.*;                 // pour les composants
class  essaiGestionnaire {
        public static void main (String arg[] ) {
        JFrame f = new JFrame(" fenêtre avec BorderLayout").
        JPanel pan = new JPanel();
        f.add(pan);
        pan.setLayout(new BorderLayout ());
        pan.add("East", new JButton( "Ceci est un bouton"));
        pan.add("Center", new JTexField("Ceci est une zone de texte"));
        JPanel p = new JPanel();
        p.setLayout(new BorderLayout());
        JButton bouton1 = new JButton("Un autre bouton");
        p.add("North", bouton1);
        pan.add("South", p);
        f.pack());
        f.show();  } }
```

55

55

Interface graphique

PROGRAMMATION ÉVÉNEMENTIELLE

56

56

Gestion des événements

Une fois l'interface graphique créée, il convient d'implémenter le comportement de l'application face aux événements (actions appliquées sur les composants).

Par exemple, que doit-il se passer lorsque l'on clique sur l'un des boutons ? Lorsque l'on effectue un choix dans une liste?

Pour cela, il est nécessaire de préciser l'intérêt que porte un composant a un type d'événement (ou a plusieurs): notion d'abonnement.

Java permet ainsi, de ne s'intéresser qu'aux types d'évènements désires pour un composant donné (sans avoir a traiter tous les types d'événements existants pouvant potentiellement être gérés par ce composant). Ceci représente un moyen aisé, efficace et fin pour réaliser de la programmation événementielle.

Il est nécessaire d'importer java.awt.event.*;

57

57

Gestion des événements

La logique générale de gestion d'événements est assez simple:

- Des objets (en général des composants) sont *sources d'évènements*. Par là, on entend qu'ils sont à l'écoute d'événements (clic, choix, etc.) et qu'ils provoquent en réponse l'instanciation d'un *objet évènement (Event)*.
  - Par exemple, un bouton est en attente d'un clic. Si ce dernier se produit, un objet *événement* est créé.
- d'autres objets sont *écouteurs d'événements*, c'est-à-dire qu'ils sont en attente d'un *objet événement*'. Ces objets écouteurs sont des instances de classes implémentant des *interfaces Listeners* (écouteur).
- A noter que les objets sources *référencient des objets écouteurs*, c'est-à-dire qu'ils indiquent quels objets vont écouter les évènements créés: lorsque qu'un événement se produit, un objet événement est adresse à *tous les objets écouteurs référencés*.

Il est assez fréquent que l'écouteur soit le container. Cependant, ceci n'est pas obligatoire.

58

58

Gestion des événements

Exemple: un bouton n'a d'autre utilité que d'être pressé. Lorsque l'on clique sur un JButton *b1* (évènement), quelque chose doit se passer (affichage à l écran, par exemple).

La gestion des événements se déroule selon un mécanisme très précis.

- Il faut en premier préciser qu'un composant source porte de l'intérêt à un type d'évènement (il s'abonne à ce type d'événement). Pour cela, on utilise sur ce composant une méthode *addxxxListener(...)* où xxx représente un type d'événement. En argument de cette méthode doit être précisé un élément écouteur qui est susceptible de connaître la manière de réagir face à l'événement.
- Ex: un bouton s'abonne au clic de la souris.

59

59

Gestion des événements

- Il faut donc également coder le fait qu'un élément du programme (une classe) puisse être intéressé par le fait que l'événement attendu par le composant soit produit (ex: clic sur bouton). Pour cela, cette classe (écouteuse) doit implémenter l'interface *xxxListener* et va contenir une (ou plusieurs) méthode *xxxaction(xxxEvent)*. Cette dernière attend un *xxxEvent* (objet Event créé par le composant source lorsque l'évènement survient) et elle indique ce qu'il se passe lorsque l'évènement de départ survient.
  - NB: action dans "xxxaction" représente une action réalisable au niveau du composant (ex: actionPerformed pour un bouton pressé, mouseMoved lors d'un déplacement de souris,...).
- Si l'événement survient (Ex: clic) et qu'il concerne ce composant source (Ex: bouton), un objet de type *xxxEvent* est instancié. Celui-ci est capturé par un objet de la classe écouteuse qui implémente l'interface *xxxListener*. La méthode xxxaction de l'objet écouteur est alors appelée.

60

60

Gestion des événements

Exemple

1) prenons une interface graphique contenant uniquement un JButton.

```
import javax.swing.*;
public class Appli {
      JFrame f  = new JFrame ();
      JPanel p  = new JPanel();
     JButton b = new JButton();
            public Appli() {
                        f.add(p) ;
                        p.add(b);
                        f.show();
            }
}
```

61

61

Gestion des événements

2) Nous voulons provoquer un affichage à l'écran lorsque le bouton b est pressé. il faut donc préciser que b est à l'écoute (abonné) des événements de type "clic" qui le concernent (dans le cas d'un bouton, *xxx* équivaut à *Action*).

```
import javax.swing.*;
import java.awt. event.*;
public class Appli {
      JFrame f  = new JFrame ();
      JPanel p  = new JPanel();
     JButton b = new JButton();
 public Appli(){
            f.add(p) ;
            b.addActionListener(…); //abonnement de b aux événement de type Action
                                    //INCOMPLET!!
            p.add(b);
            f.show();
} }
```

62

62

Gestion des événements

3) b est abonné aux événements de type *Action*. Lorsque un événement de ce type va concerner b (clic), un objet de type *ActionEvent* va être instancié. b est donc source de l'objet *événement (Event)*. Ce dernier contient des informations, comme l'objet source dé sa création (le bouton b), etc.

A noter que la source d'un objet *xxxEvent* peut être obtenu via la méthode getSource() qui retourne un objet de type *Object* (il faut donc généralement utiliser un *cast)*.

63

63

Gestion des événements

4) Cet objet doit être traité par une méthode *actionPerformed()* qui indique la réaction face à l'événement. Donc une classe doit coder cette méthode. Pour cela, la classe en question doit implémenter l'interface *ActionListener*. Le fait d'implémenter cette interface implique deux choses: les instances de cette classe sont en attente d'objets *ActionEvent* (donc d'événements de type Action) et la classe DOIT coder la méthode *actionPerformed()*. Ce dernier point est obligatoire car les méthodes d'une interface sont abstraites:

- Si une classe implémente cette dernière, elle doit en redéfinir les méthodes (toutes les méthodes).

64

64

Gestion des événements

Il faut donc:

- qu'une classe implémente l'interface. Dans le cas présent, notre seule classe est Appli. Nous allons donc faire en sorte qu'elle soit à l'écoute des événements gérés par son propre bouton.
- au niveau de l'abonnement du bouton b pour les événement de type *Action* (méthode *addActionListener),* indiquer un élément susceptible de gérer l'objet *ActionEvent* créé. Cet élément est une instance de la classe implémentant l'interface *ActionListener* et donc, ce sera l'objet courant (this) car ici c'est le seul objet créé.

Une autre manière de voir les choses est de dire que la classe écouteuse s'enregistre auprès de l'objet source b via la méthode *addActionListener()* afin d'écouter les événements.

- que la classe écouteuse possède une méthode *actionPerformed*.

65

65

Gestion des événements

```
import javax.swing.*;
import java.awt. event.*;
public class Appli implements ActionListener { //classe à l'écoute des événements ActionEvent
      JFrame f  = new JFrame ();
      JPanel p  = new JPanel();
     JButton b = new JButton();
                public Appli() {
                                f.add(p);
                                b.addActionListener(this); // Si  b pressé, l'objet ActionEvent créé est supposé
                                                           // Être traité par l'objet courant
                                p.add(b);
                                f.show();  }
     public void ActionPerformed (ActionEvent evt){
                                System.out.println ( "le bouton a été pressé");
                                // pour être plus complet, réaliser  avec un  test de la source:
                                Object source = evt.getSource();
                                // if (source==b) System.out.println("…");  }
 }
}
```

66

66

Gestion des événements

Java propose de nombreux composants, donc de très nombreux types d'événements pouvant être traités.

Un composant peut être abonné à plusieurs types d'événements, une classe peut implémenter plusieurs interfaces (et donc posséder plus leurs méthodes permettant de gérer plusieurs types d'événements).

Tous les composants ne peuvent pas être abonnés aux mêmes événements.

La logique est toujours identique. Pour un type d'événement xxx, nous avons.

- une interface xxxListener
- un type d'objet événement xxxEvent
- une (ou plusieurs) méthode xxxaction attendant comme argument un objet xxxEvent. (rappel: action ne représente pas un terme générique des méthodes mais simplement rappelle le fait qu'une action bien précise est gérée par la méthode. Ainsi, action peut valoir "pressed", "released", "clicked", "opened", "closed", etc).

67

67

Gestion des événements

Catégories	- Événements - Interfaces	Méthodes	Composants intéressés (non exhaustif)
Action (clic, touche entrée, sélection d'un élément)	- ActionEvent - ActionListener	actionPerformed (ActionEvent)	AbstractButton, Button/JButton, JComboBox, JFileChooser,JTextField , List, JList, MenuItem et JMenuItem et leurs dérivées, TextField,
Adjustement *(déplacement du curseur d'une barre de défilement)*	- AdjustementEvent - AdjustementListener	adjustmentValueChanged (AdjustmentEvent)	Scrollbar, JScrollbar et tout *ce* qui implémente l'interface Adjustable
Focus (obtention ou perte du focus pur un composant)	- FocusEvent - FocusListener	FocusGained(FocusEvent) FocusLost(FocusEvent)	component et derivées (Button, JPanel, JFrame, JTextField,JLabel, etc.)

68

68

Gestion des événements

catégorie	- Événements - Interfaces	Méthodes	Composants Intéressés (non exsudatif)
Component *(déplacement affichage, masquage ou modification de taille de composants)*	ComponentEvent ComponentListener	- componentMoved (ComponentEvent) - componentRisezed (ComponentEvent). - componentShown (ComponentEvent) - componentHidden (componentEvent)	Component et ses dérivées.
Container *(ajout ou suppression d'un composant dans un conteneur)*	-ContainerEvent -ContainerListener	- componentsAdded (ContainerEvent) - componentRemoved (ContainerEvent)	Container et ses dérivées (JPanel , JApplet, JFrame, JDialog, etc.)
Key (action sur une touche du clavier)	-KeyEvent -KeyListener	-keyPressed (KeyEvent) -keyReleased (KeyEvent) -keyTyped (KeyEvent)	Component et ses dérivées Exp. TextField

69

69

Gestion des événements

catégorie	Événements Interfaces	Méthodes	Composants Intéressés (non exsudatif)
Item *(sélection dans une liste ou dans un groupe de cases à cocher)*	- ItemEvent - ItemListener	itemStateChanged (itemEvent)	AbstractButton, CheckBox, JCheckBox, CheckBoxMenuItem, Choice, ComboBox, JComboBox,List/JList et tout ce qui implémente l'interface ItemSelectable.
Text (modification du texte d'un composant texte)	-TextEvent -TextListener	textValueChanged (TextEvent)	TextComponent, JTextComponent et leurs dérivées (JtextArea, JTextField, etc,)

70

70

Gestion des événements

catégorie	Événements Interfaces	Méthodes	Composants Intéressés (non exsudatif)
Mouse *(clic sur bouton, déplacement du pointeur)*	- MouseEvent - MouseListener	mousePressed (MouseEvent) mouseReleased (MouseEvent) mouseClicked (MouseEvent) mouseEntered (MouseEvent) mouseExited (MouseEvent)	Component et ses dérivées
MouseMotion (glisser – déplacer)	- MouseEvent (idem que catégorie précédente) MouseMotionListener	- mouseDragged (MouseEvent) - mouseMoved (MouseEvent)	Component et ses dérivées

71

71

Gestion des événements

catégorie	Événements Interfaces	Méthodes	Composants Intéressés (non exsudatif)
Window (fenêtre activée désactivée, réduite, fermée, ...)	WindowEvent WindowListener	windowOpened (WindowEvent) windowClosing (WindowEvent) windowClosed (WedowEvent) windowIconified (WindowEvent) windowDeiconified (WindowEvent) windowActivated (WindowEvent) wiedowDeactivated (WiedowEvent)	window *et ses dérivées* (JDiolog, JFileDialog, JFrame,etc.)

L'ensemble de ces éléments proviennent du package AWT mais sont utilisés également avec les composants swing. Le package swing à augmenté cette liste et le nombre d'événements gérables s'est fortement accru.

72

72

Gestion des événements

Adaptateurs (adapters)

- Parmi les interfaces développées dans AWT et swing certaines sont accompagnées d'*Adapters*. Ce sont les mécanismes permettant de faciliter l'emploi de ces interfaces.
- Une interface est constituée de méthodes abstraites. Une classe implémentant une interface et dans l'obligation de redéfinir l'ensemble des méthodes de cette dernière
- Par exemple, si l'on veut gérer la fermeture d'une fenêtre, il faut que la fenêtre (Frame) soit abonnée aux événements de type *WindowEvent* et que la classe implémente l'interface *WindowListener*.
- Or cette dernière contient 7 méthodes abstraites dont une seule utile pour fermer la fenêtre.
- Les *Adapters* permettent de ne pas gérer l'ensemble de ces méthodes mais uniquement celle(s) qui nous est (sont) utile(s).

73

73

Gestion des événements

- Au lieu de coder

```
JFrame f = JFrame();
f.addWindowListener(this);
public void windowClosing (WindowEvent e){
                        System.exit (0); }
public void windowOpend(WindowEvent e) {}
public void windowClosed(WindowEvent e) {}
public void windowIonified(WindowEvent e) {}
public void windowDeiconified(WindowEvent e) {}
public void windowActivated(WindowEvent e) {}
public void windowDeaActivated(WindowEvent e) {}
```

- Il est possible de coder

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

74

74

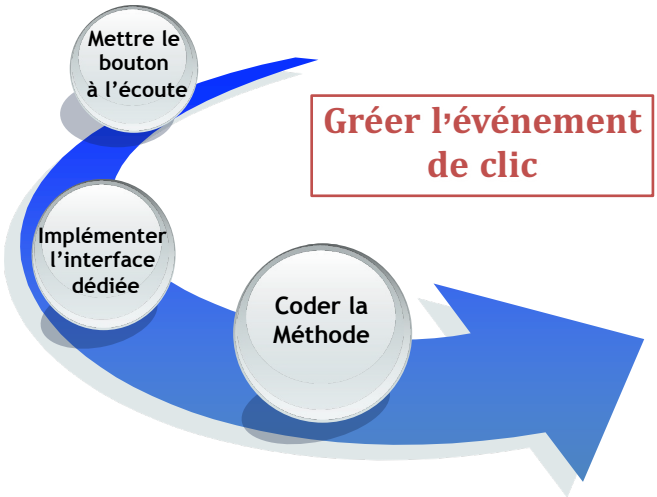
Adapters

Exemples d'adapters:

- WindowAdapter
- MouseAdapter
- MouseMotionAdapter
- ContainerAdapter
- ComponentAdapter
- FocusAdapter
- KeyAdapter

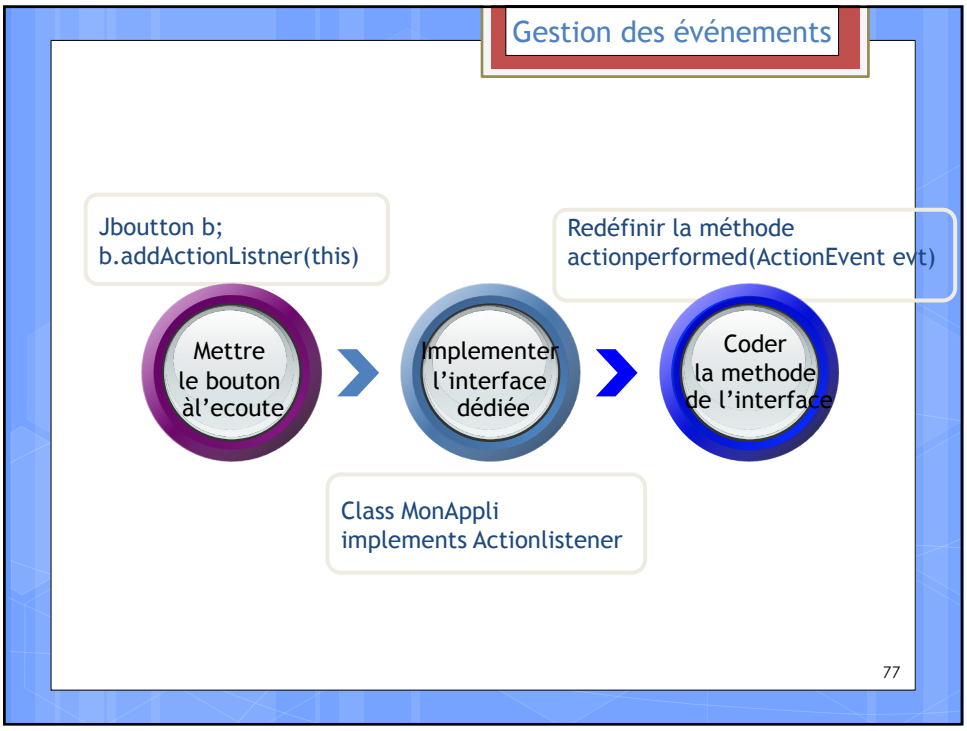
75

Démarche à suivre



76

Gestion des événements



77

Gestion des événements

```
import javax.swing.*;
import java.awt.event.*;
public class Appli implements ActionListener { //classe à l'écoute des
                                                //événements ActionEvent
    JFrame f  = new JFrame ();
    JPanel p  = new JPanel();
   JButton b = new JButton();
          public Appli() {
                    f.add(p) ;
                    b.addActionListener(this); // si  b pressé, l'objet
                              // ActionEvent créé est supposé
                             // Être traité par l'objet courant
                    p.add(b);
                    f.show(); }
    public void ActionPerformed (ActionEvent evt){
       if ((( JButton)evt.getSource()) = =b) System.out.printin("...");) }}
```

78

78

Codage

Plusieurs remarques:

- La classe peut très bien dériver de Frame ou JFrame. Cela ne l'empêche pas d'être écouteuse.
- Ce n'est pas forcément la classe qui contient les composants qui écoute. Les seuls impératifs à respecter sont l'existence d'une classe écouteuse (qui peut être située ailleurs dans l'application) et le fait que les composants sources aient comme écouteur un objet de cette classe
- Les possibilités de codage sont donc importantes.
- Les documents suivants présentent deux cas qui peuvent être rencontrés :

79

79

Codage

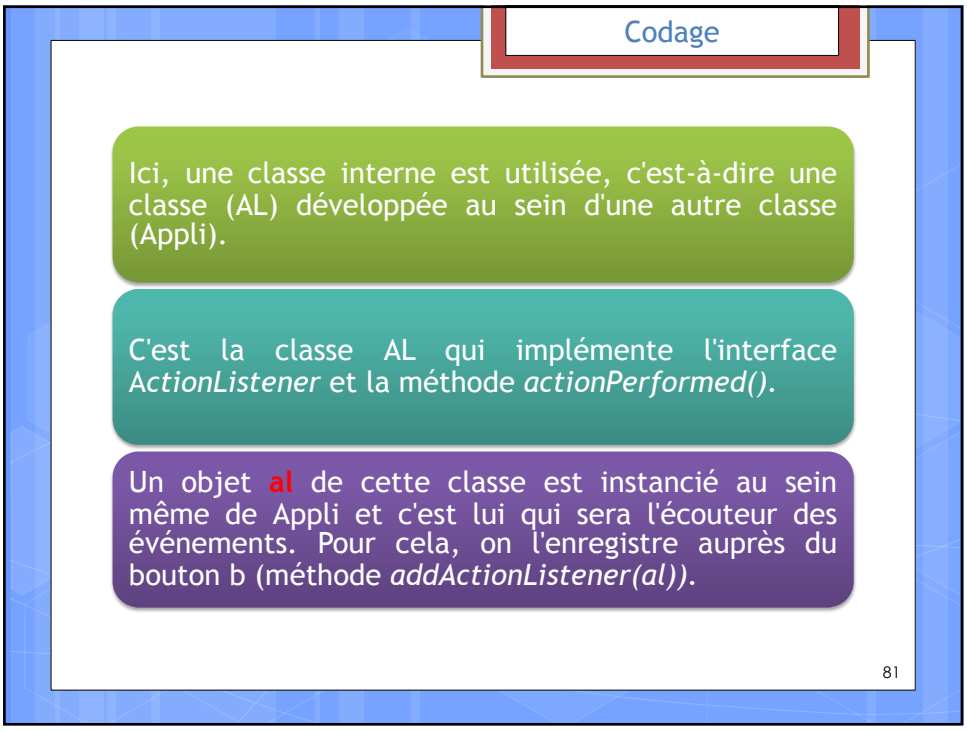
```
import javax.swing.*;
import java.awt.event.*;
public class Appli {
      JFrame f  = new JFrame ();
      JPanel p  = new JPanel();
      JButton b = new JButton("OK");

      class AL implements ActionListener {
      public void actionPerformed (ActionEvent evt) {
      if ((( JButton)evt.getSource()) ==b)
      System.out.println("Bonjour");
      }
      }

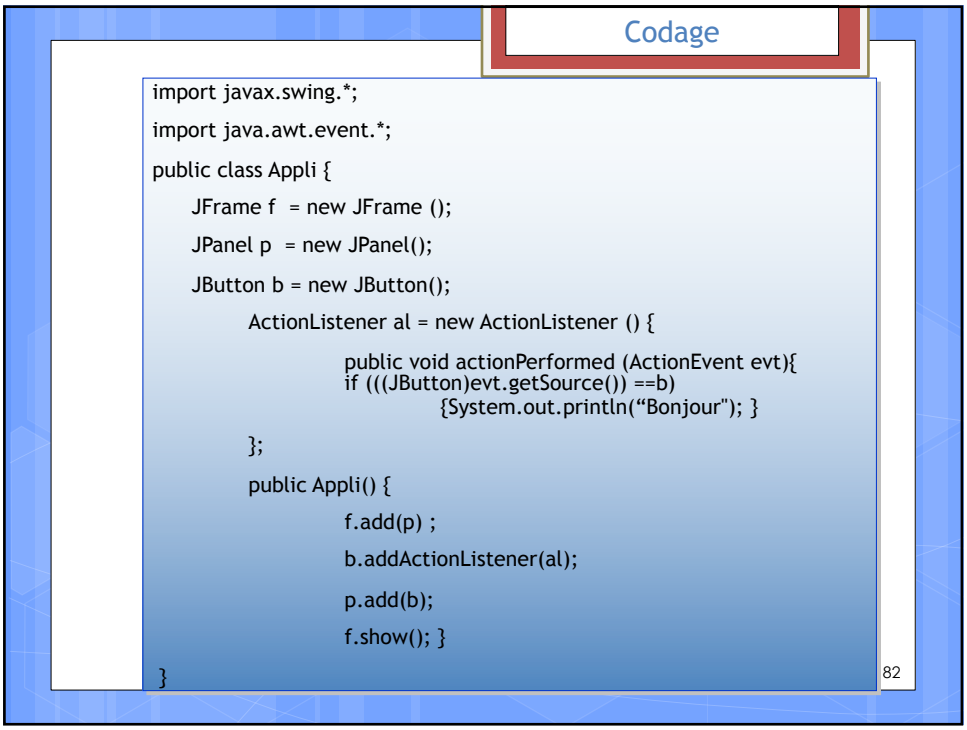
       AL al = new AL();
            public Appli() {
            f.add(p) ;
            b.addActionListener(al);
            p.add(b);
            f.show();
            }
}
```

80

80


Codage
Ici, une classe interne est utilisée, c'est-à-dire une classe (AL) développée au sein d'une autre classe (Appli).
C'est la classe AL qui implémente l'interface ActionListener et la méthode actionPerformed().
Un objet al de cette classe est instancié au sein même de Appli et c'est lui qui sera l'écouteur des événements. Pour cela, on l'enregistre auprès du bouton b (méthode addActionListener(al)).
81

81


Codage
import javax.swing.*;
import java.awt.event.*;
public class Appli {
JFrame f = new JFrame ();
JPanel p = new JPanel();
JButton b = new JButton();
ActionListener al = new ActionListener () {
public void actionPerformed (ActionEvent evt){
if (((JButton)evt.getSource()) ==b)
{System.out.println("Bonjour"); }
};
public Appli() {
f.add(p) ;
b.addActionListener(al);
p.add(b);
f.show(); }
}
82

82

Codage

Dans le dernier exemple est codée une classe interne anonyme. C'est un raccourci pour la classe interne vue précédemment. On crée un objet al d'une classe (anonyme) "dérivée de" (en fait implémentant) l'interface ActionListener.

83

FIN



QUESTIONS !!!

84

Question 1

- Comment vous indiquer où un composant sera positionné en utilisant Flowlayout?

A. North, South,East,West;
B. Attribution row/column de la grille de référence;
C. Passer un X / Y pourcentage paramètre à la méthode add;
D. Ne rien faire, le FlowLayout positionnera la composante

85

85

Question 2

- Lorsque deux ou plusieurs composants ont été ajoutés à l'écoute d'un même événement, qui est le premier composant invoqué pour traiter l'action ?

A. Le premier composant qui a été mit à l'écoute.
B. Le dernier objet qui a été mit à l'écoute
C. Il n'existe aucun moyen de déterminer qui écoute sera invoquée en premier.
D. Il est impossible d'avoir plus d'un abonné pour un événement donné.

86

86