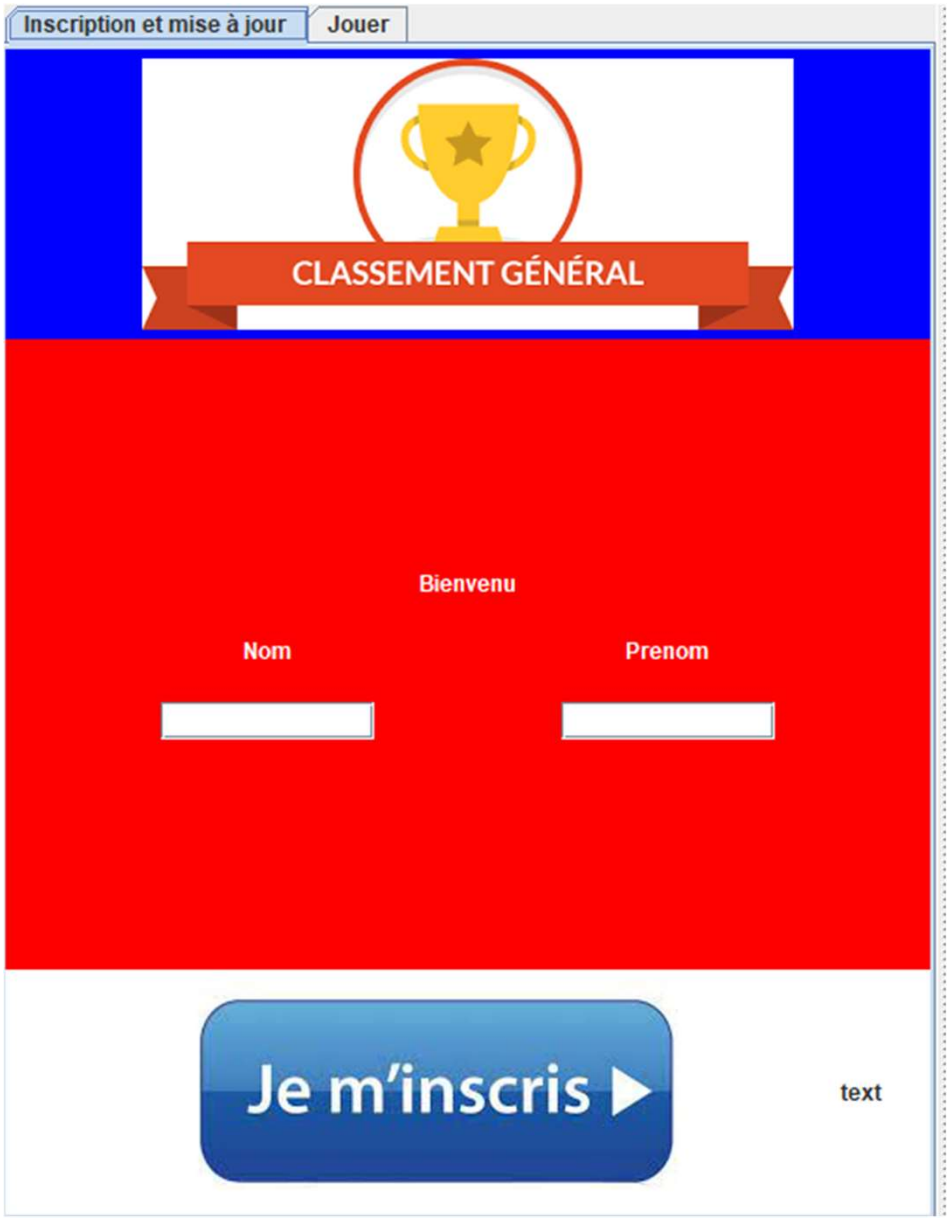
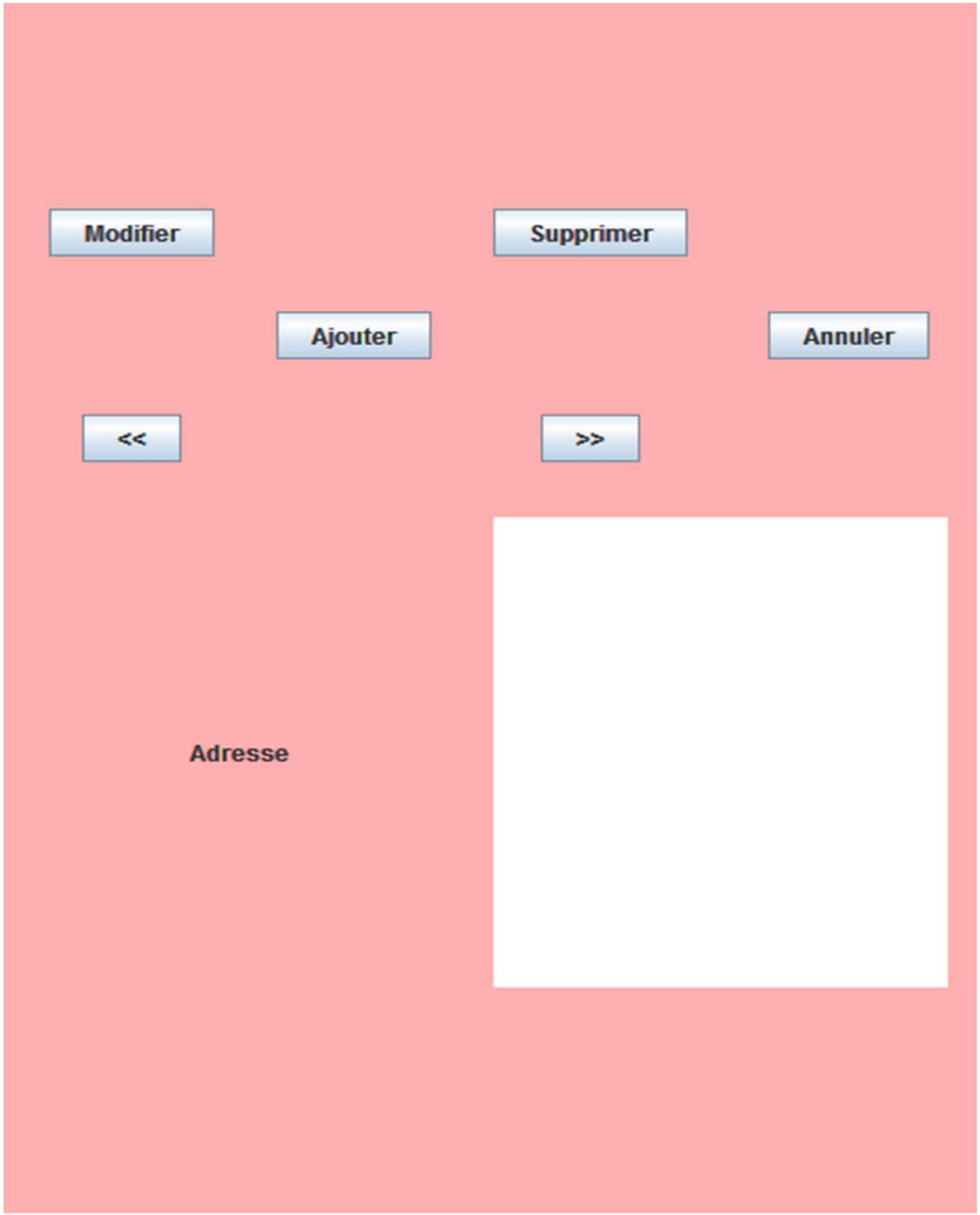
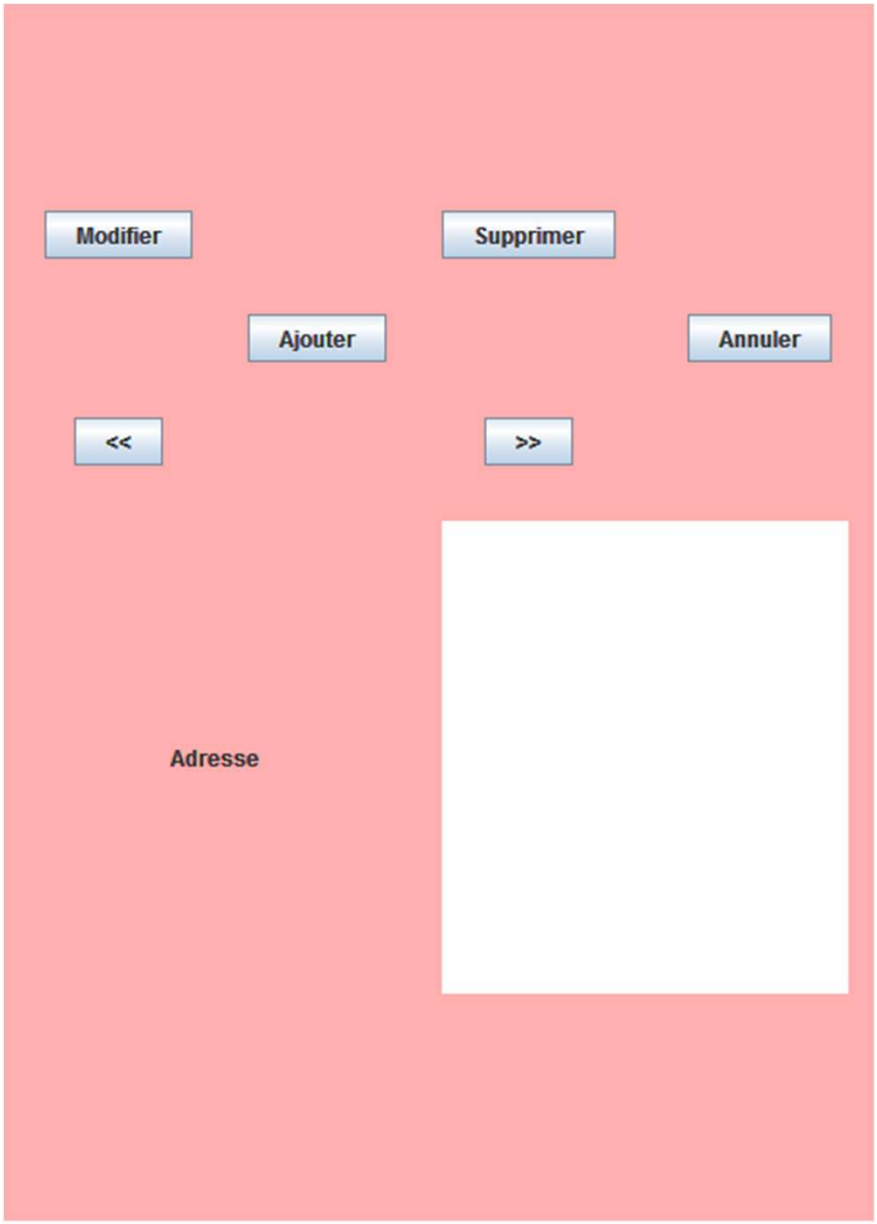
```
pCentre.setLayout(new GridBagLayout());
GridBagConstraints regle= new GridBagConstraints();

regle.insets= new Insets(10,50,10,50);
regle.gridwidth = 2;
pCentre.add(bonjour, regle);
regle.gridwidth= 1;
regle.gridy=1;
pCentre.add(nom, regle);
regle.gridx=1;
pCentre.add(prenom, regle);
regle.gridx=0; regle.gridy=2;
pCentre.add(txtNom, regle);
regle.gridx=1;
pCentre.add(txtPrenom, regle);
```



```
pane.setLayout(new GridBagLayout());

regle.insets= new Insets(10,20,20,10);
modifier= new JButton("Modifier");
supprimer= new JButton("Supprimer");
ajouter= new JButton("Ajouter");
annuler= new JButton("Annuler");
suivant= new JButton(">>");
precedant= new JButton("<<");
txtAdresse= new JTextArea(16,20);
regle.gridx=0;regle.gridy=0;
pane.add(modifier, regle);
regle.gridx=2;
pane.add(supprimer, regle);

regle.gridx=1;regle.gridy=1;
pane.add(ajouter, regle);
regle.gridx=3;
pane.add(annuler, regle);

regle.gridx=0;regle.gridy=2;
pane.add(precedant, regle);
regle.gridx=2;
pane.add(suivant, regle);
```



```
regle.gridx=0;regle.gridy=3;
regle.gridwidth=2;
pane.add(adresse, regle);
regle.gridx=2;
regle.gridwidth=GridBagConstraints.REMAINDER;
pane.add(txtAdresse, regle);



}
public static void main( String[] args) {
JFrame f= new Fenetre();
}


}
```



LA GESTION DES ÉVÉNEMENTS

Gérer les événements

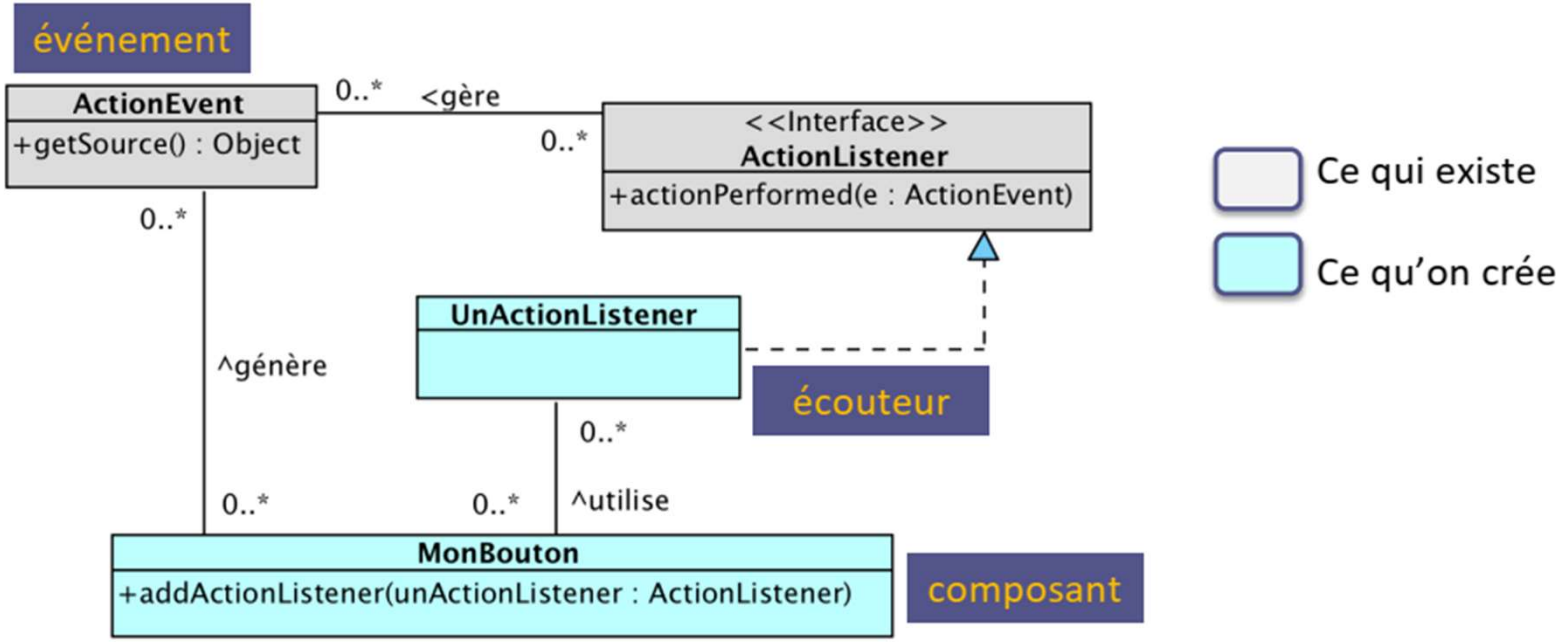
- Après avoir créé la partie visible de l'interface (Vue), il faut lier les actions réalisées avec le modèle.
- Il faut gérer les évènements (souris, clavier, etc.) :
  - les « intercepter » : lier une action à un composant (bouton, list, tableau, etc.)
  - les « interpréter » : implémenter l'interface correspondante
- Avec Swing, vous devez implémenter une interface pour chaque signal.
- Par exemple pour actionner un bouton :
  - l'interface est ActionListener dont la seule méthode est actionPerformed.
- Rappel : Utilisation du mot clef « implements », et vous devez implémenter toutes les méthodes de l'interface.

Gérer les événements

- L'utilisateur effectue
  - une action au niveau de l'interface utilisateur (clic souris, sélection d'un item, etc)
  - alors un **événement graphique** est émis.
- Lorsqu'un événement se produit
  - il est reçu par le composant avec lequel l'utilisateur interagit (par exemple un bouton, un curseur, un champ de texte, etc.).
  - Ce composant transmet cet événement à un autre objet, un **écouteur** qui possède une méthode pour traiter l'événement
    - cette méthode reçoit l'objet événement généré de façon à traiter l'interaction de l'utilisateur.

Principe

- Un composant peut générer certains **évènements**
- Un objet **évènement** reflète une **action** de l'utilisateur (clic, passage souris)
- La gestion de l'évènement est déléguée à un **écouteur** d'évènements qui active le **traitement** associé selon le type d'évt (Action, Key, Mouse listener)

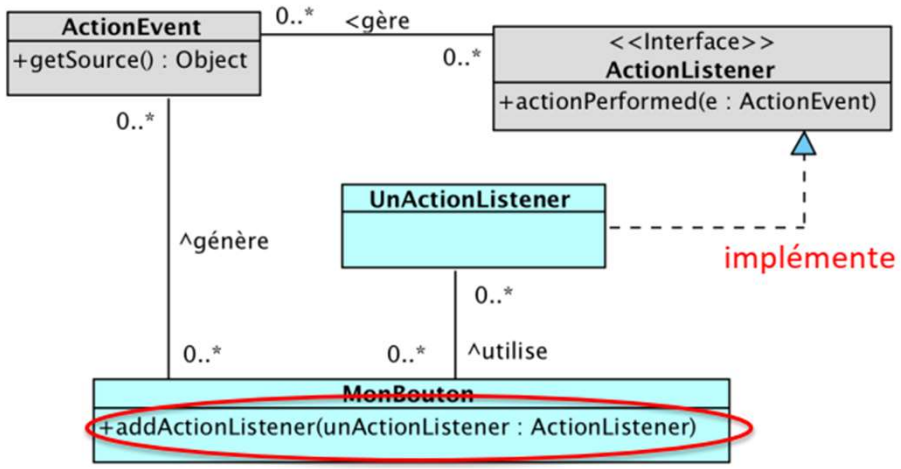


Principe

- Les écouteurs sont des objets qui implémentent des interfaces prédéfinies (`MouseListener, ActionListener`)

- Ces écouteurs doivent être explicitement affectés aux composants concernés

```
monBouton.addActionListener(new UnActionListener() );
```



Les événements

- Tous les composants génèrent des événements
  - Car ils dérivent de la classe Component qui génère des événements
- Tous les composants ne génèrent pas tous les même événements
  - Un bouton ne génère pas d'événements de type text
- Il existe pour les composants élémentaires un événement de sémantique générale appelé ActionEvent, qui représente l'interaction standard avec l'utilisateur
  - Click sur bouton ==> ActionEvent
  - DoubleClick sur une liste ==> ActionEvent
  - Click sur un élément de liste ==> ActionEvent
  - <Return> à la fin d'une saisie dans un TextField ==> ActionEvent

Démarche

En résumé, il faut :

– Identifier les **objets source** et le **type d'événement**, donc l'**écouteur** nécessaire,

– Personnaliser le traitement des événements en **implémentant les méthodes** des écouteurs,

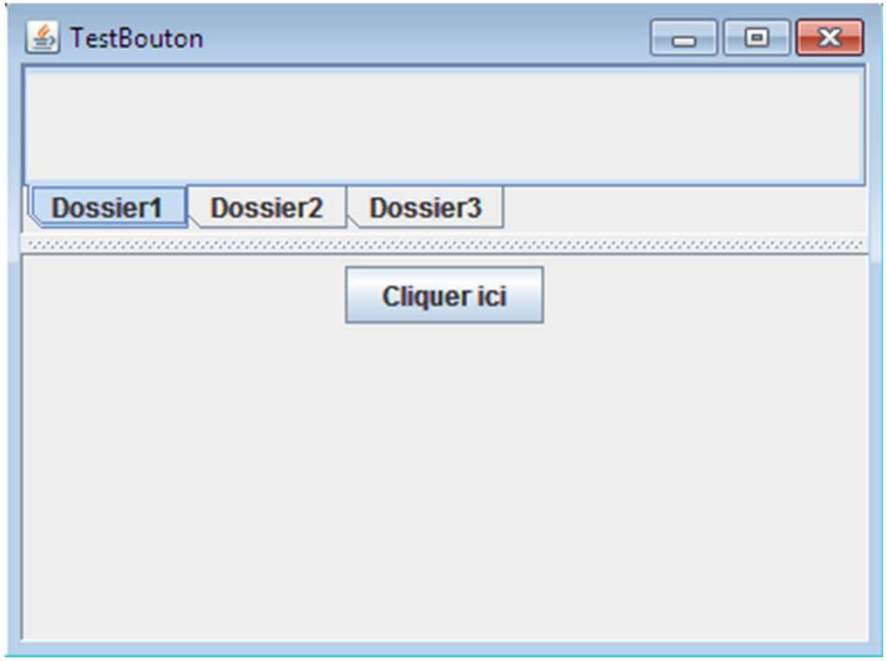
– **Relier** les objets écouteurs aux objets sources pour permettre le traitement.

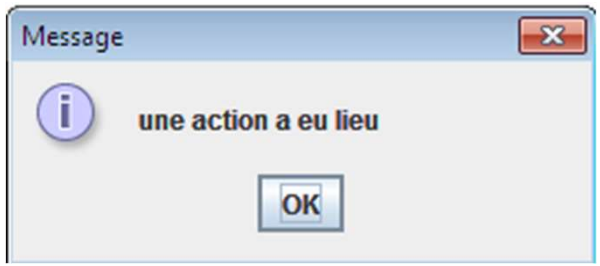
Les Listeners

- Les événements qui intéressent le programmeur doivent être capturés dans des écouteurs
- Ces écouteurs sont des objets qui implémentent des interfaces prédéfinies (MouseListener, ActionListener)
- Par exemple, l'interface ActionListener contient la méthode:
  - public void actionPerformed(ActionEvent e)
- Ces écouteurs doivent être explicitement affectés aux composants concernés (boutons,..)
  - monBouton.addActionListener(new EcouteurSimple());

La gestion des événements

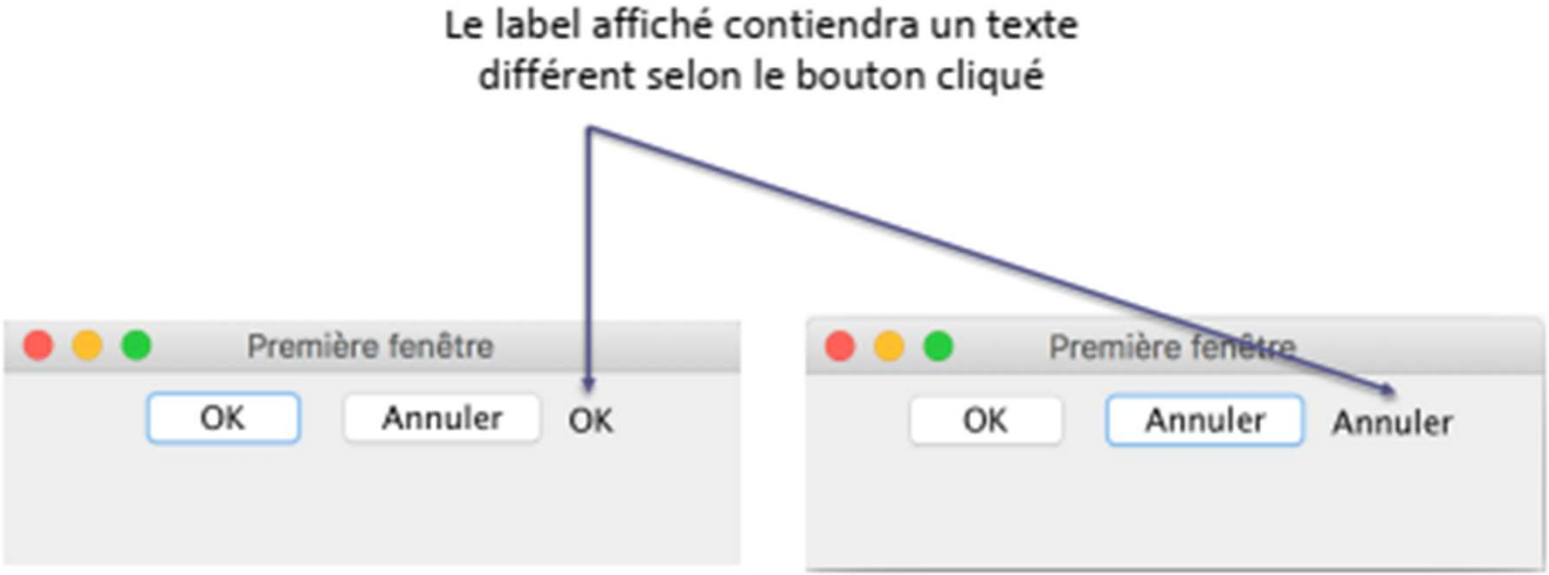
```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JSplitPane;
import javax.swing.JTabbedPane;

public class TestBouton  extends JFrame implements ActionListener{
JSplitPane pane;
JTabbedPane onglets;
JPanel panneau, panel1, panel2, panel3;
JButton b;
public TestBouton(){
super ("TestBouton");setSize(400,300);
panel1 =new JPanel();panel2 =new JPanel();panel3 =new JPanel();
onglets= new JTabbedPane(JTabbedPane.BOTTOM);
onglets.addTab("Dossier1", panel1);onglets.addTab("Dossier2", panel2);onglets.addTab("Dossier3", panel3);
panneau= new JPanel();
pane= new JSplitPane(JSplitPane.VERTICAL_SPLIT,onglets, panneau);
b = new JButton ("Cliquer ici");
add(pane);
panneau.add (b) ;
b.addActionListener (this);}

public void actionPerformed (ActionEvent e)  {
//System.out.println ("Une action a eu lieu") ;
 JOptionPane.showMessageDialog(this, "une action a eu lieu");
  public static void main(String args[]) {
    TestBouton test = new TestBouton();
    test.setVisible (true) ;
    test.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);}
}
```


TestBouton
Dossier1
Dossier2
Dossier3
Cliquer ici


Message
une action a eu lieu
OK

3 façons pour implémenter les écouteurs

1. Définir les classes écouteurs en classes internes
   - ✓Définir une classe xxxListener au sein de la classe principale
2. Faire que la classe principale soit son propre écouteur
   - ✓Très fréquent
   - ✓Utilisé pour les IHM simplesCoder
3. un écouteur en classe anonyme
   - ✓ (classe écouteur sans nom)
   - ✓ Là où on ajoute un écouteur
   - ✓ Lorsque le code associé au traitement est court


Le label affiché contiendra un texte différent selon le bouton cliqué
Première fenêtre
OK
Annuler
OK
Première fenêtre
OK
Annuler
Annuler

1ère implémentation d’un écouteur : classe écouteur en **classe interne**

Ecouteur défini en classe interne

- Créer la ou les classes internes (au sein de la classe principale)
  - Il faut indiquer que la classe interne est **un écouteur** : elle implémente **l'interface** `XXXListener`
- **Dans la classe interne**, redéfinir la ou les méthodes de l'interface d'écoute pour les événements à gérer :
  - Ex.: la méthode `actionPerformed()` si l'événement est un `ActionEvent`
  - L'événement est passé en paramètre
- Le composant doit ajouter un écouteur pour **chaque classe** d'événements à traiter, par exemple:
  - `boutonA.addActionListener(new MonEcouteurAction() );`
  - `boutonA.addMouseListener(new MonEcouteurSouris() );`

```
public class TestEvents1 extends JFrame{
    JButton ButOK, ButAnnuler;
    JPanel pane;
    JLabel label;
    TestEvents1(){
        ButOK=new JButton("OK");
        ButAnnuler=new JButton("Annuler");
        label=new JLabel();
        pane=(JPanel) getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(ButOK);
        pane.add(ButAnnuler);
        pane.add(label);

        ButOK.addActionListener(new okButtonListner());
        ButAnnuler.addActionListener(new AnnulerButtonListner());
    }
    public static void main(String args[]){
        JFrame frame=new TestEvents1();
        frame.setTitle("Première fenêtre");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300,100);
        frame.setVisible(true);
    }

    class okButtonListner implements ActionListener {
        public void actionPerformed(ActionEvent e){
            label.setText("OK");
        }
    };

    class AnnulerButtonListner implements ActionListener {
        public void actionPerformed(ActionEvent e){
            label.setText("Annuler");
        }
    };
}
```

Ce label contiendra un texte différent selon le bouton cliqué

Ajouter à chaque bouton l'écouteur associé
Ici on choisit de définir un ***écouteur interne*** *pour chaque bouton*

On sait que les événements liés à l'action des boutons sont des ActionEvent : on implémente donc ActionListener, qui n'a qu'une seule méthode: actionPerformed()

```java
public class TestEvents2 extends JFrame{
    JButton ButOK, ButAnnuler;
    JPanel pane;
    JLabel label;
    TestEvents2(){
        ButOK=new JButton("OK");
        ButAnnuler=new JButton("Annuler");
        label=new JLabel();
        pane=(JPanel) getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(ButOK);
        pane.add(ButAnnuler);
        pane.add(label);

        ButOK.addActionListener(new MyButtonListner());
        ButAnnuler.addActionListener(new MyButtonListner());
    }
    public static void main(String args[]){
        JFrame frame=new TestEvents2();
        frame.setTitle("Première fenêtre");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300,100);
        frame.setVisible(true);
    }

    class MyButtonListner implements ActionListener {
        public void actionPerformed(ActionEvent e){
            if(e.getSource()==ButOK){
                label.setText("OK");
            }
            else
               label.setText("Annuler");
        }
    };
}
```

Ici on définit une seule **classe écouteur interne** pour tous les composants

On doit donc tester dans la méthode, de quel bouton provient l'évt capté : la méthode getSource() d'un ActionEvent retourne le composant concerné

Discussion : implémentation en classe interne

- Gestion des événements bien isolée dans une classe
- Rajoute du code

*On choisira la classe interne si cette classe n'a **aucun intérêt à être exposée** aux autres classes ; elle peut être utilisée par différents composants de la classe principale (**factorisation**).*

*C'est donc un choix intéressant si on a **beaucoup d'actions différentes** à coder en fonction des composants sollicités.*

2ème implémentation d’un écouteur : la classe principale est son **propre** écouteur

La classe principale est son propre écouteur

- Indiquer que la classe principale implémente la ou les interfaces XXXListener

- Comme dans la 1ère implémentation :
  Redéfinir la ou les méthodes de l'écouteur pour les événements à gérer (mais cette fois dans la classe même) :

  - Ex.: la méthode `actionPerformed()` si l'écouteur est un ActionListener pour un `ActionEvent`

- Cette fois, chaque composant qui va réagir doit ajouter l'**écouteur** (qui est la classe) pour toutes les classes d'événements à traiter :
  - `boutonA.addActionListener(this );`
  - `boutonA.addMouseListener(this );`

```java
public class TestEvents3 extends JFrame implements ActionListener{
    JButton ButOK, ButAnnuler;
    JPanel pane;
    JLabel label;
    TestEvents3(){
        ButOK=new JButton("OK");
        ButAnnuler=new JButton("Annuler");
        label=new JLabel();
        pane=(JPanel) getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(ButOK);
        pane.add(ButAnnuler);
        pane.add(label);

        ButOK.addActionListener(this);
        ButAnnuler.addActionListener(this);
    }
    public static void main(String args[]){
        JFrame frame=new TestEvents3();
        frame.setTitle("Première fenêtre");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300,100);
        frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent e){
        if(e.getSource()==ButOK){
            label.setText("OK");
        }
        else
           label.setText("Annuler");
    }
}
```

Indiquer que la classe principale implémente l'interface XXXListener

Ajouter à chaque composant l'objet écouteur associé, qui est la classe elle-même (this)

Redéfinir les méthodes de l'interface XXXListener (ici actionPerformed() pour définir le traitement

Discussion : la classe est son propre écouteur

- Pas de code en plus
- Gestion des événements noyée dans le code de la classe

Rmq : ce choix d'implémentation n'est pas limitant pour la définition de la classe, *puisque l'héritage multiple* ***d'interfaces*** *est possible en Java.*

Pour la **lisibilité** du code, on utilisera ce mode quand on a *peu* de traitements à coder (*peu* ? subjectif !).

3ème implémentation d'un écouteur : Coder un écouteur **en classe anonyme**

Coder un écouteur en classe anonyme

- Lorsque l'action relative à un évènement est **simple** on passe par une **classe d'écouteur anonyme** pour définir les observateurs associés au composant
  - Rappel *classe anonyme* : classe instanciée sans nom, juste avec `new NomClasse () {… code de la classe }`
  - Dans le cas d'interface, le code *new InterfaceEcouteur()* crée en fait une classe anonyme qui implémente l'interface…

- Plus besoin d'indiquer que la classe implémente l'interface écouteur

- Cela permet de définir **un écouteur par composant source**

```java
public class TestEvents4 extends JFrame {
    JButton ButOK, ButAnnuler;
    JPanel pane;
    JLabel label;
    TestEvents4(){
        ButOK=new JButton("OK");
        ButAnnuler=new JButton("Annuler");
        label=new JLabel();
        pane=(JPanel) getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(ButOK);
        pane.add(ButAnnuler);
        pane.add(label);

        ButOK.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
            label.setText("OK");
        }});
        ButAnnuler.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
            label.setText("Annuler");
        }});
    }
    public static void main(String args[]){
        JFrame frame=new TestEvents4();
        frame.setTitle("Première fenêtre");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300,100);
        frame.setVisible(true);
    }
}
```

Pas d'implémentation d'interface XXXListener

Ecouteur placé au niveau de la classe

Ecouteur instancié en classe anonyme

On ne code que les méthodes qui seront utilisées

Discussion : écouteur codé en classe anonyme

- Pas de code en plus
- Pas de déclaration en entête de classe des écouteurs utilisés par la classe

- Ecouteurs moins visibles, noyé dans le code
- Pas de déclaration en entête de classe : lisibilité des dépendances moins facile
- Code parfois difficile à lire

Mode utilisé par les IDE générant automatiquement le code des IHMs. Assez utilisé en réalité car peu contraignant au niveau de la déclaration de la classe, mais code difficile à maintenir.

La gestion des événements

- Les écouteurs sont des interfaces
- Donc une même classe peut implémenter plusieurs interfaces écouteur.
  - Par exemple une classe héritant de JFrame implémentera les interfaces MouseMotionListener (pour les déplacements souris) et MouseListener (pour les clics souris).
- Chaque composant de l'AWT est conçu pour être la source d'un ou plusieurs types d'événements particuliers.
  - Cela se voit notamment grâce à la présence dans la classe de composant d'une méthode nommée add**XXX**Listener().

Catégories d'événements

- Plusieurs types d'événements sont définis dans le package java.awt.event.
- Pour chaque catégorie d'événements, il existe une interface qui doit être définie par toute classe souhaitant recevoir cette catégorie d'événements.
  - Cette interface exige aussi qu'une ou plusieurs méthodes soient définies.
  - Ces méthodes sont appelées lorsque des événements particuliers surviennent.

Interfaces écouteurs - Description - Composants

Interface écouteur	Description	Composant qui génère l'événement
ActionListener	Ecoute les actions de l'utilisateur : clics, barre d'espacement, entrée … • `JButton, JRadioButton et JCheckBox` : clic gauche et barre d'espacement • `JComboBox` : déplacement dans la zone déroulante avec les flèches de déplacement • `JTextField` : touche Entrée	`JButton, JRadioButton, JCheckBox` `JComboBox, JTextField`
AdjustmentListener	Ajustement d'une barre de défilement	`JScrollBar`
ComponentListener	Déplacement, affichage, masquage ou modification de la taille de composants	`Component`
ContainerListener	Ajout ou suppression d'un composant dans un conteneur	`Container`
FocusListener	Obtention ou perte du focus par un composant	`Component`
ItemListener	Sélection d'un élément dans un combobox ou dans une liste (mais pas avec `JList`) ou dans un groupe de cases à cocher	`Checkbox,JComboBox,List,` `JRadioButton…`
KeyListener	Ecoute les `KeyEvents` : actions sur les touches du clavier (pressée ou relâchée)	`Component`
DocumentListener	Changement ou insertion dans un document	`Document`

Interfaces écouteurs - Description - Composants

Interface écouteur	Description	Composant qui génère l'événement
LsitSelectionListener	Ecouteur qui notifie quand la sélection d'éléments dans une liste change (L'utilisateur sélectionne (désélectionne) un item ou plusieurs items dans une `JList`, une `JTable`, …)	`JList`, `JTable`, …
MouseListener	Action sur la souris (Clic sur le bouton de la souris: appuyer, relâcher, déplacer le pointeur)	`Component`
MouseMotionListener	Action de la souris sur un composant (Evénements de glisser-déplacer)	`Component`
WindowListener	Action sur la fenêtre (Fenêtre activée, désactivée, réduite, fermée, …)	`Window`
TextListener	Changement d'une zone de texte	`JTextField`

Interfaces écouteurs - Traitement et enregistrement

Interface écouteur	Méthodes de traitement	Méthode d'enregistrement
ActionListener	actionPerformed(ActionEvent e)	addActionListener()
AdjustmentListener	adjustmentValueChanged(adjustmentEvent e)	addAdjustmentListener()
ComponentListener	componentHidden(ComponentEvent e) componentMoved(ComponentEvent e) componentResized(ComponentEvent e) componentShown(ComponentEvent e)	addComponentListener()
ContainerListener	componentAdded(ContainerEvent e) componentRemoved(ContainerEvent e)	addContainerListener()
FocusListener	focusGained(FocusEvent e) focusLost(FocusEvent e)	addFocusListener()
ItemListener	itemStateChanged(ItemEvent e)	addItemListener()
KeyListener	keyPressed(KeyEvent e) keyReleased(KeyEvent e) keyTyped(KeyEvent e)	addKeyListener()
DocumentListener	changedUpdate(DocumentEvent e) insertUpdate(DocumentEvent e) removeUpdate(DocumentEvent e)	addDocumentListener()

Enregistrement de l'écouteur sur un composant :
monBouton.addActionListener(xxx);

Interfaces écouteurs - Traitement et enregistrement

Interface écouteur	Méthodes de traitement	Méthode d'enregistrement
LsitSelectionListener	`valueChanged (ListSelectionEvent e)`	`addListSelectionListener()`

MouseListener	`mouseClicked(MouseEvent e)`
	`mouseEntered(MouseEvent e)`
	`mouseExited(MouseEvent e)`
	`mousePressed(MouseEvent e)`
	`mouseReleased(MouseEvent e)`
		`addMouseListener()`

MouseMotionListener	`mouseDragged(MouseEvent e)` `mouseMoved(MouseEvent e)`	`addMouseMotionListener()`

WindowListener	`windowActivated(WindowEvent e)`
	`windowClosed(WindowEvent e)`
	`windowClosing(WindowEvent e)`
	`windowDeactivated(WindowEvent e)`
	`windowDeiconified(WindowEvent e)`
	`windowIconified(WindowEvent e)`
	`windowOpened(WindowEvent e)`
		`addWindowListener()`

TextListener	`textValueChanged(TextEvent e)`	`addTextListener()`

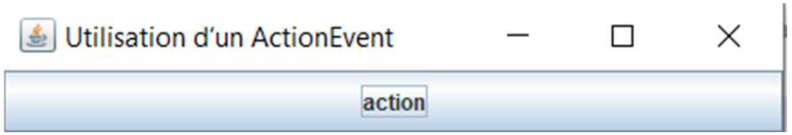
Catégories d'événements

- ActionListener
  - Action (clic) sur un bouton, retour chariot dans une zone de texte, « tic d'horloge » (Objet Timer)
- WindowListener
  - Fermeture, iconisation, etc. des fenêtres
- TextListener
  - Changement de valeur dans une zone de texte
- ItemListener
  - Sélection d'un item dans une liste
- FocusListener
  - Pour savoir si un élément a le "focus"
- KeyListener
  - Pour la gestion des événements clavier

Catégories d'événements

- MouseListener
  - Clic, enfoncement/relâchement des boutons de la souris, etc.
- MouseMotionListener
  - Déplacement de la souris, drag&drop avec la souris, etc.
- AdjustmentListener
  - Déplacement d'une échelle
- ComponentListener
  - Savoir si un composant a été caché, affiché …
- ContainerListener
  - Ajout d'un composant dans un Container

Catégories d'événements




```
public class EssaiActionEvent1 extends JFrame
                        implements ActionListener
{
  public static void main(String args[])
  {EssaiActionEvent1 f= new EssaiActionEvent1();}
  public EssaiActionEvent1()
  {
   super("Utilisation d'un ActionEvent");
   JButton b = new JButton("action");
   b.addActionListener(this);
   add(BorderLayout.CENTER,b);pack();setVisible(true);
  }
  public void actionPerformed( ActionEvent e )
 {
   setTitle("bouton cliqué !");
 }}
```

Implémentation de l'interface ActionListener

On enregistre l'écouteur d'evt action auprès de l'objet source "b"

Lorsque l'on clique sur le bouton dans l 'interface, le titre de la fenêtre change

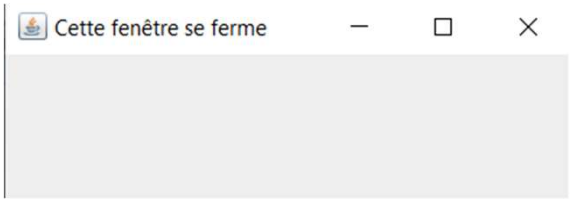
Catégories d'événements

```
public class EssaiActionEvent2 extends JFrame
                          implements ActionListener
{ private JButton b1,b2;
  public static void main(String args[])
  {EssaiActionEvent2 f= new EssaiActionEvent2();}
  public EssaiActionEvent2(){
   super("Utilisation d'un ActionEvent");
   b1 = new JButton("action1");
   b2 = new JButton("action2");
   b1.addActionListener(this);
   b2.addActionListener(this);
   add(BorderLayout.CENTER,b1);
   add(BorderLayout.SOUTH,b2);
   pack();setVisible(true); }
  public void actionPerformed( ActionEvent e ) {
   if (e.getSource() == b1) setTitle("action1 cliqué");
   if (e.getSource() == b2) setTitle("action2 cliqué");
  }}
```

Les 2 boutons ont le même écouteur (la fenêtre)

e.getSource()" renvoie l'objet source de l'événement. On effectue un test sur les boutons (on compare les références)

Catégories d'événements




```
import javax.swing.*; import java.awt.event.*;
public class WinEvt extends JFrame
       implements WindowListener
{
 public static void main(String[] args) {
   WinEvt f= new WinEvt();}
   public WinEvt() {
     super("Cette fenêtre se ferme");
setSize(500,400);
     addWindowListener(this);
   setVisible(true);}
    public void windowOpened(WindowEvent e){}
    public void windowClosing(WindowEvent e)
    {System.exit(0);}
    public void windowClosed(WindowEvent e){}
    public void windowIconified(WindowEvent e){
System.out.println("Fenetre en icone");}
    public void windowDeiconified(WindowEvent e){}
    public void windowActivated(WindowEvent e){}
    public void windowDeactivated(WindowEvent e){}
}
```

Implémenter cette interface impose l'implémentation de bcp de méthodes

La fenêtre est son propre écouteur

WindowClosing() est appelé lorsque l'on clique sur la croix de la fenêtre

"System.exit(0)" permet de quitter une application java

JOptionPane

La classe *JOptionPane* fournit des façons simples de créer des dialogues élémentaires modaux en spécifiant un message, un titre, une icône, et un type de message ou un type d'option.
Si on ne spécifie pas d'icône, des icônes sont fournies par le système en fonction du type de message.

L'utilisation typique de cette classe consiste à des appels à une des méthodes statiques de la forme `showXxxDialog` qui suivent :

Méthodes	Description	Retourne
showConfirmDialog	Demande une question qui se répond par oui/non/annulé.	Option
showInputDialog	Demande de taper une réponse.	String (Objet)
showMessageDialog	Transcrit un message à l'usager.	rien
showOptionDialog	Une méthode générale réunissant les 3 précédentes.	Option

Dialogue d'information

L'appel de la méthode *showMessageDialog* de la classe *JOptionPane* permet de faire apparaître des messages
`JOptionPane.showMessageDialog(Component parentComponent, String message, String title, int messageType); où :`

- Component parentComponent : correspond au composant parent ; ici, il n'y en a aucun, nous mettons donc null.
- String message : permet de renseigner le message à afficher dans la boîte de dialogue.
- String title : permet de donner un titre à l'objet.
- int messageType : permet de savoir s'il s'agit d'un message d'information, de prévention ou d'erreur.

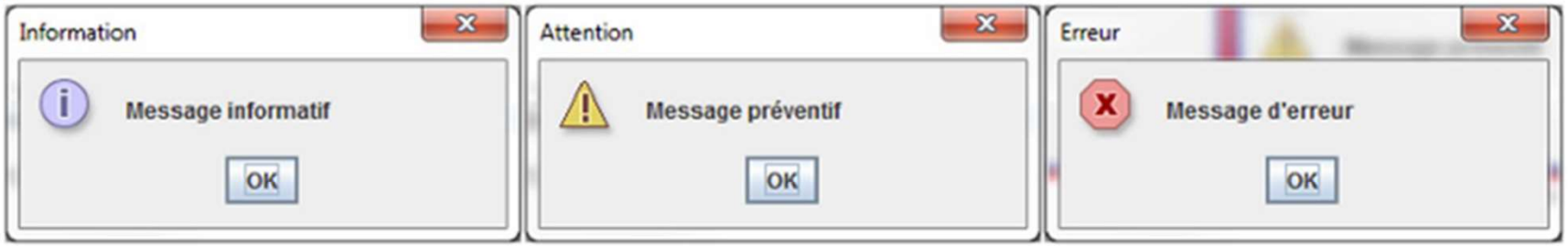
Le type de message est l'une des valeurs suivantes :

- JOptionPane.PLAIN_MESSAGE
- JOptionPane.ERROR_MESSAGE
- JOptionPane.INFORMATION_MESSAGE
- JOptionPane.WARNING_MESSAGE
- JOptionPane.QUESTION_MESSAGE

Type	Icône par défaut (Look & Feel Java)
ERROR_MESSAGE	
INFORMATION_MESSAGE	
WARNING_MESSAGE	
QUESTION_MESSAGE	
PLAIN_MESSAGE	aucun

Exemple :

```
//Boîte du message d'information
JOptionPane.showMessageDialog(null, "Message informatif", "Information", JOptionPane.INFORMATION_MESSAGE);
//Boîte du message préventif
JOptionPane.showMessageDialog(null, "Message préventif", "Attention", JOptionPane.WARNING_MESSAGE);
//Boîte du message d'erreur
JOptionPane.showMessageDialog(null, "Message d'erreur", "Erreur", JOptionPane.ERROR_MESSAGE);
);
```



Dialogue de confirmation

L'appel de la méthode *showConfirmDialog* de la classe*JOptionPane* permet de faire apparaître des messages, avec demande de confirmation :

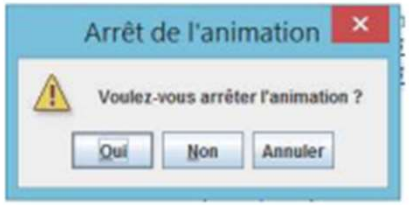
```
int retour = JOptionPane.showConfirmDialog(Component parentComponent, String message, String title, int option message, int messageType);
```

L'option de message est l'une des valeurs suivantes :

- JOptionPane.DEFAULT_OPTION (ok)
- JOptionPane.YES_NO_OPTION
- JOptionPane.YES_NO_CANCEL_OPTION (*)
- JOptionPane.OK CANCEL OPTION

Exemple :

```
int retour = JOptionPane.showConfirmDialog(null, "Voulez-vous arrêter l'animation ?", "Arrêt de l'animation", JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.WARNING_MESSAGE);
```



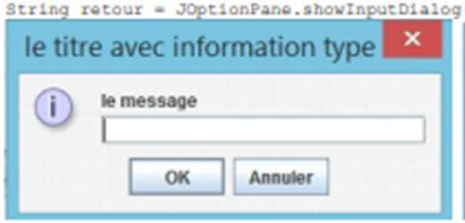
La valeur retournée par l'appel de méthode est l'une des trois suivantes :

- *YES_OPTION*
- *NO_OPTION*
- *CANCEL_OPTION*

Dialogue de saisie

L'appel de la méthode *showInputDialog* de la classe *JOptionPane* permet de faire une saisie de chaîne de caractères :

```
String retour = JOptionPane.showInputDialog(null, "le message", "le titre", messageType);
String retour = JOptionPane.showInputDialog(null, "le message", "VALEUR INITIALE");
```



Si retour vaut *null* la saisie n'est pas validée, sinon retour vaut la chaîne tapée par l'utilisateur. Si la méthode doit retourner un objet (String ou Object) et que l'usager décide d'appuyer sur annuler (Cancel), la valeur retournée est *null*.

```
String[] sexe = {"masculin", "féminin", "indéterminé");

    String nom = (String) JOptionPane.showInputDialog(null,
      "Veuillez indiquer votre sexe !",
      "Gendarmerie nationale !",
      JOptionPane.QUESTION_MESSAGE,
      null,
      sexe,
      sexe[2]);
    JOptionPane.showMessageDialog(null, "Votre sexe est " + nom, "Etat civil", JOptionPane.INFORMATION_MESSAGE);
```

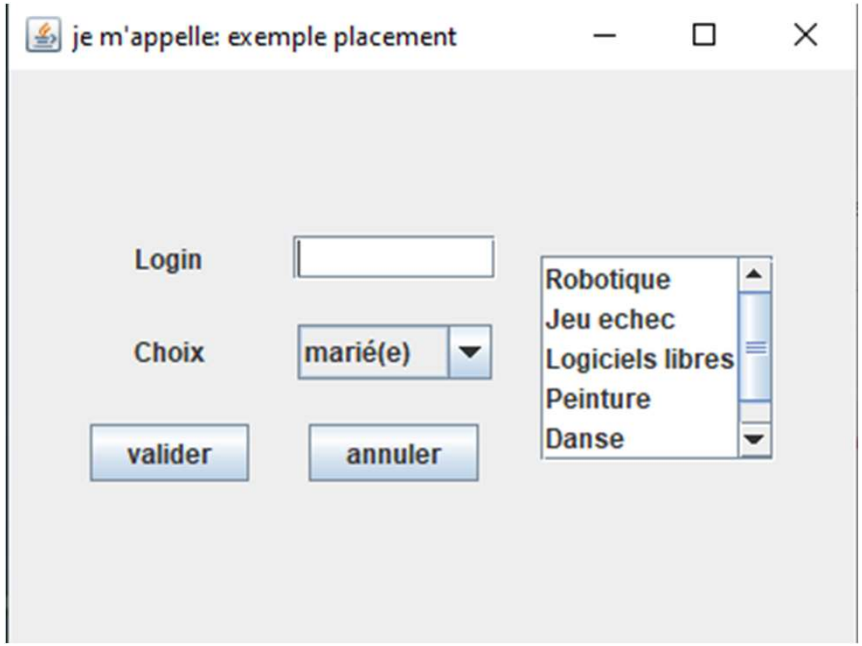
détail des paramètres utilisés dans cette méthode :

- les quatre premiers, vous connaissez ;
- le deuxième `null` correspond à l'icône que vous souhaitez passer ;
- ensuite, vous devez passer un tableau de `String` afin de remplir la combo (l'objet `JComboBox`) de la boîte ;
- le dernier paramètre correspond à la valeur par défaut de la liste déroulante.
- Cette méthode retourne un objet de type Object, comme si vous récupériez la valeur directement dans la combo ! Du coup, n'oubliez pas de faire un cast.

Voici maintenant une variante de ce que vous venez de voir : nous allons utiliser ici la méthode`showOptionDialog()`. Celle-ci fonctionne à peu près comme la méthode précédente, sauf qu'elle prend un paramètre supplémentaire et que le type de retour n'est pas un objet mais un entier. Ce type de boîte propose un choix de boutons correspondant aux éléments passés en paramètres (tableau de `String`) au lieu d'une combo ; elle prend aussi une valeur par défaut, mais retourne l'indice de l'élément dans la liste au lieu de l'élément lui-même.

```
import javax.swing.JOptionPane;
public class Test {
  public static void main(String[] args) {
    String[] sexe = {"masculin", "féminin", "indéterminé"};
       int rang = JOptionPane.showOptionDialog(null,
      "Veuillez indiquer votre sexe !",
      "Gendarmerie nationale !",
      JOptionPane.YES_NO_CANCEL_OPTION,
      JOptionPane.QUESTION_MESSAGE,
      null,
      sexe,
      sexe[2]);
    JOptionPane.showMessageDialog(null, "Votre sexe est " + sexe[rang], "Etat civil", JOptionPane.INFORMATION_MESSAGE);
  } }
```

Ce qui nous donne la figure suivante.




```
valider.addActionListener(new ActionListener(){
public void actionPerformed(ActionEvent e) {
JOptionPane.showMessageDialog(null, txtlogin.getText()+"
      "+listchoix.getSelectedItem());
}
});

clubs.addListSelectionListener(new ListSelectionListener(){
public void valueChanged(ListSelectionEvent e) {
JOptionPane.showMessageDialog(null,
     clubs.getSelectedValue());//getSelectedValuesList()
}
});
```

getSelectedValues()	**Deprecated.** As of JDK 1.7, replaced by getSelectedValuesList()