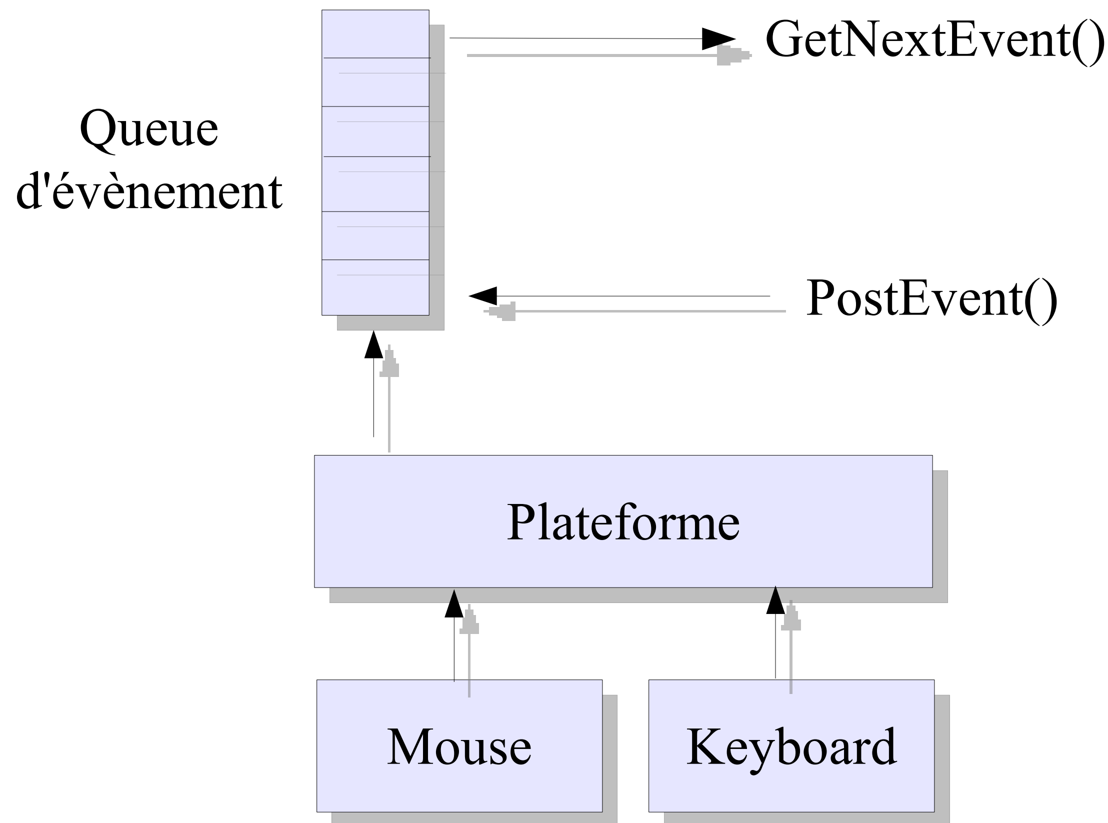
Gestion d'évènements

- Principes de la gestion d'évènements
- Gestion d'évènements par listener
- Composants comme Java Beans

La queue d'évènements

- Les évènements sont stockés dans une queue alimentée soit par la plateforme soit de façon programmé.



Evènements dans Windows :main loop

- Dans Windows, un événement au sens usuel est matérialisé par un message.
- Une application Windows est construite autour d'une boucle de messages, qui distribue les messages aux fenêtres.

```
Int WINAPI WinMain(...) {
    while (GetMessage(&msg, NULL, 0, 0))
        DispatchMessage(&msg);
    return msg.wParam;
}
```

- Le comportement d'une classe de fenêtres est régi par sa *procédure de fenêtre*.
- Cette procédure décrit la gestion de tous les messages par un grand aiguillage (switch).

Exemple de procédure de fenêtre

- Une procédure de fenêtre gère les évènements pour une fenêtre entière

```
LRESULT CALLBACK ProcFen(HWND hwnd, UINT uMsg,
                    WPARAM wParam, LPARAM lParam) {
    switch(uMsg) {
      case WM_LBUTTONDOWN:
        Dessiner(hwnd);
        break;
      case WM_DESTROY:
        PostQuitMessage(0);
        break;
      default:
        return DefWindowProc(hwnd, uMsg, wParam,
          lParam);
    }
    return 0;
}
```

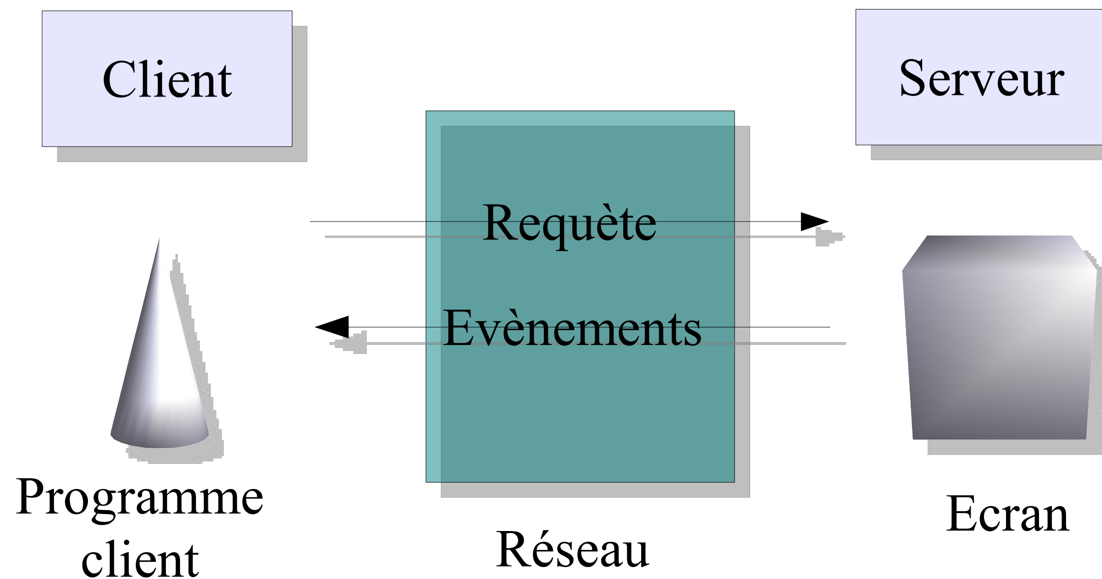
La boucle d'événements en Xlib

- Xlib est la base du système X-Window.
- La boucle est écrite explicitement
- Environ 30 évènements différents.

```
for (;;) {
     XNextEvent(&e);
     switch(e.type) {
         case Expose :
           draw(e.window);
           break;
         case ButtonPress:
           ...
           break;
     }
}
```

Serveur X et Client X

- Le Serveur X est responsable de l'envoi d'évènements (et de l'affichage des requêtes).
- Le Client X lit des évènements, et envoie des requêtes.



Evènements en Xt Intrinsics

- Boucle principale encapsulée
- Fonction réflexe

```
Void salut(Widget w, XtPointer clientData,
  XtPointer callData)
{
   // do what you need
}
```

```
main(int argc, char **argv)
{
   Widget racine, boite, bouton;
   XtAppContext app;
       ...
   XtAddCallback(bouton , XmNactivateCallback,
     salut, NULL);
       ...
   XtAppMainLoop(app);
}
```

Boucle principale

- La gestion des évènements est prise en charge par la fonction **XtAppMainLoop** qui
  - lit le prochain événement,
  - et l'envoie au composant approprié.

```
main(int argc, char **argv)
{
   Widget racine, boite, bouton;
   XtAppContext app;
       ...
   XtAddCallback(bouton , XmNactivateCallback,
     salut, NULL);
       ...
   XtAppMainLoop(app);
}
```

Liste de fonctions réflexes

- Chaque classe de widgets possède des *listes de fonction réflexes*.
- La liste est identifiées par un nom, p. ex: **`XmNactivateCallback`** (activation pour Motif).
- la liste est initialement vide mais modifiable à volonté, de fonctions (*fonction réflexes* ou "callbacks").
- Une fonction réflexe est exécutée lorsqu'un événement du type représenté par la liste a eu lieu.

- Les besoins de la gestion d'évènement en Java :
- Le dispatch d'un évènement doit se faire suivant deux critères :
  - Le composant qui reçoit l'évènement
  - Le type de l'évènement reçu
- Besoin **double-dispatch** ou **Visitor** (*design pattern*), dispatch suivant le composant puis suivant le type d'évènement.
- Malheureusement pas implanté dans AWT !!!

Boucle des évènements en Java

- La boucle d'évènements en Java est implicite :
  - Une classe dérivant de Component hérite la gestion des évènements
  - Un thread spécifique est en charge de délivrer les évènements.

- Il est donc strictement **INTERDIT** d'utiliser une autre thread que celle qui gère les évènements pour effectuer des changements sur l'interface graphique.

Les évènements en Java 1.0

- Chaque composant a sa procédure de traitement d'évènements handleEvent().
- Cette fonction consomme l'évènement, ou le transmet à son conteneur hiérarchique pour traitement.

```
public boolean handleEvent() {
  switch (e.id) {
      case Event.MOUSE_ENTER :
          return mouseEnter(e, e.x, e.y) ;
      case Event.MOUSE_EXIT :
          return mouseExit(e, e.x, e.y) ;
      case Event.MOUSE_MOVE :
          return mouseMove(e, e.x, e.y) ;
      case Event.MOUSE_DOWN :
          return mouseDown(e, e.x, e.y) ;
      case Event.MOUSE_DRAG :
          return mouseDrag(e, e.x, e.y) ;
      case Event.MOUSE_UP :
          return mouseUp(e, e.x, e.y) ;
      case Event.KEY_PRESS :
      case Event.KEY_ACTION :
          return keyDown(e, e.key) ;
      case Event.KEY_RELEASE :
      case Event.KEY_ACTION_RELEASE :
          return keyUp(e, e.key) ;
      case Event.ACTION_EVENT :
          return action(e, e.arg) ;
      case Event.GOT_FOCUS :
            return gotFocus(e, e.arg) ;
      case Event.LOST_FOCUS :
            return lostFocus(e, e.arg) ;
  }
  return false ;
}
```

Les évènements en Java 1.0 (suite)

- Inconvénients :
  - Nécessité du sous-classement pour changer le comportement (redéfinir la méthode **`handleEvent()`**).
  - Un évènement qui n'est pas attendu descend toute la hiérarchie pour s'en rendre compte.
    => lenteur

- Modèle d'évènement pas adapté, mais nécessité de changer de modèle mais sans perdre la compatibilité.

Les évènements en Java 1.0 (fin)

- Changement de modèle entre Java 1.0 et Java 1.1 :
  - Java 1.0 proche de Windows
  - Java 1.1 proche de Xlib Motif

- Pour résoudre les problèmes, il faut que le code qui réagisse à un évènement soit séparé du code du composant.

Les événements en Java 1.1

- Les évènements sont groupés en familles.
- L'enregistrement des fonctions réflexes n'est pas possible (pas de pointer de fonction en Java :)
- On enregistre des objets réflexe (listeners) encapsulant les fonctions.
  design pattern
  Observer/Observable.

```
AWTEvent
    ActionEvent
    AdjustmentEvent
    ComponentEvent
        ContainerEvent
        FocusEvent
        InputEvent
            KeyEvent
            MouseEvent
        PaintEvent
        WindowEvent
    ItemEvent
    TextEvent
    ...
EventListener
    ActionListener
    AdjustmentListener
    ComponentListener
    FocusListener
    MouseListener
    MouseMotionListener
    WindowListener
    ...
```

Interface EventListener

All Known Subinterfaces:

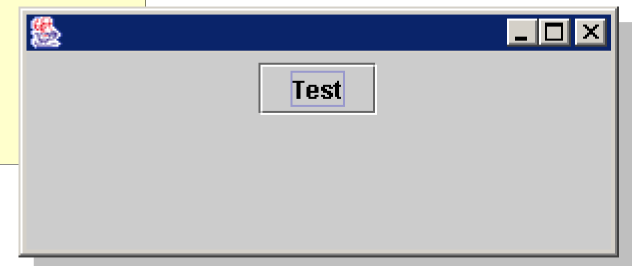
Action, ActionListener, AdjustmentListener, AncestorListener, AWTEventListener, BeanContextMembershipListener, BeanContextServiceRevokedListener, BeanContextServices, BeanContextServicesListener, CaretListener, CellEditorListener, ChangeListener, ComponentListener, ContainerListener, ControllerEventListener, DocumentListener, DragGestureListener, DragSourceListener,DropTargetListener, FocusListener, HierarchyBoundsListener, HierarchyListener, HyperlinkListener, InputMethodListener, InternalFrameListener, ItemListener, KeyListener, LineListener, ListDataListener, ListSelectionListener, MenuDragMouseListener, MenuKeyListener, MenuListener, MetaEventListener, MouseInputListener, MouseListener, MouseMotionListener, NamespaceChangeListener, NamingListener, ObjectChangeListener, PopupMenuListener, PropertyChangeListener, TableColumnModelListener, TableModelListener, TextListener, TreeExpansionListener, TreeModelListener, TreeSelectionListener, TreeWillExpandListener, UndoableEditListener, UnsolicitedNotificationListener, VetoableChangeListener, WindowListener

- Les listeners sont utilisés pour non seulement pour les évènements générés par l'utilisateur de l'interface graphique mais aussi par les composants plus évolués, par exemple pour indiquer une sélection.

Exemple d'utilisation

- On veut être averti lorsqu'un utilisateur clique sur un bouton
- On utilise l'interface **`ActionListener`** qui définit la méthode **`actionPerformed()`**.
- Cette méthode sera appelée lorsque d'un clic sur le bouton ou en appuyant sur entrée

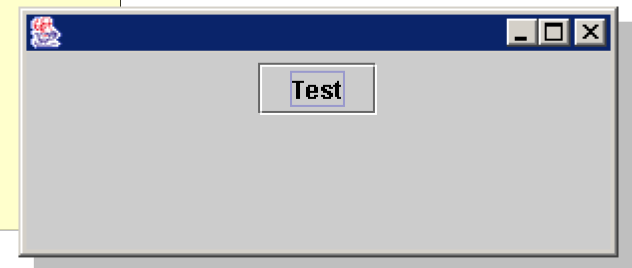
```
import java.awt.event.*;

public class MyActionListener implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    System.out.println(event);
  }
}
```



Exemple d'utilisation (suite)

- Reste à lié un objet de type MyActionListener à un bouton
- Les méthodes **add/removeActionListener()** sur un bouton en/désenregistre les objets réflexes au niveau du composant graphique

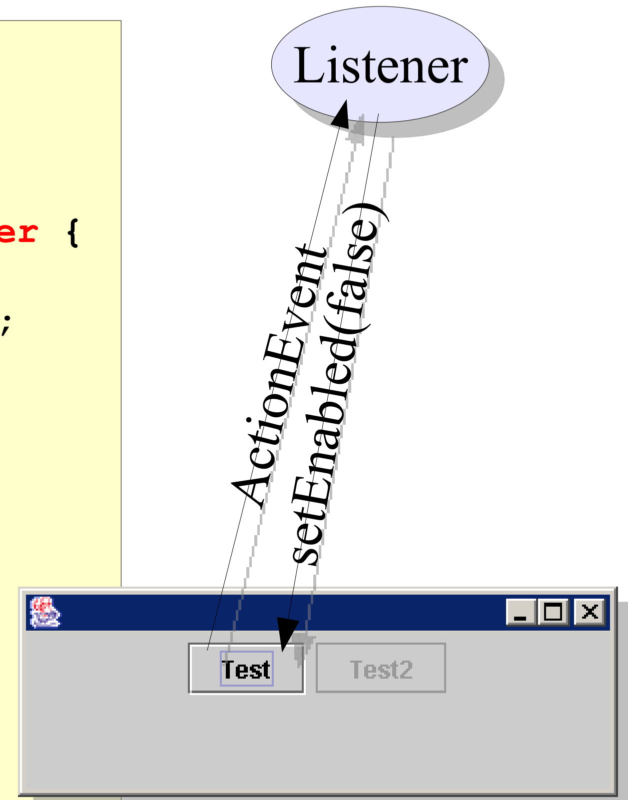
```
import javax.swing.*;

public class FirstExample {
  public static void main(String[] args) {
    JButton button=new JButton("Test");
    ActionListener l=new MyActionListener();
    button.addActionListener(l);
    ...
```



Exemple 2

- Il est possible d'enregistrer un même listener pour plusieurs composants
- `getSource()` permet d'obtenir le composant ayant créé l'évènement

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

public class FirstExample2 {
  static class MyActionListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      ((JComponent)event.getSource()).setEnabled(false);
    }
  }
  public static void main(String[] args) {
    JButton button=new JButton("Test");
    JButton button2=new JButton("Test2");

    ActionListener l=new MyActionListener();
    button.addActionListener(l);
    button2.addActionListener(l);
    ...
```



Comment bien utiliser les listeners ?

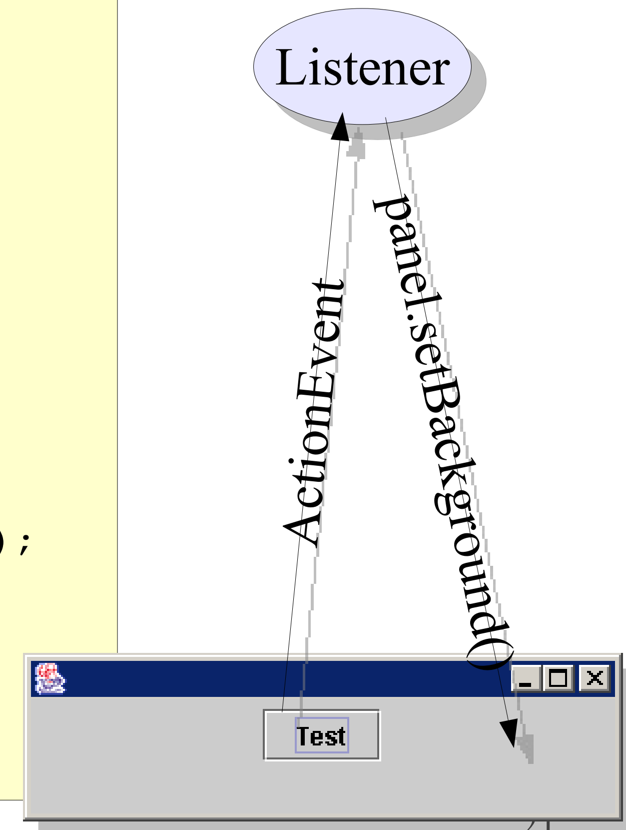
- Problème :
  pour enregistrer un listener, il faut une instance d'une classe implantant une certaine interface.

- Il existe plusieurs façons d'obtenir un tel objet :
  quel est la meilleure façon de créer un listener ?

- Voici quatre conseils pour écrire un code propre et maintenable facilement.

- La classe qui crée l'interface ne doit pas implanter l'interface du listener (dépendance inutile).

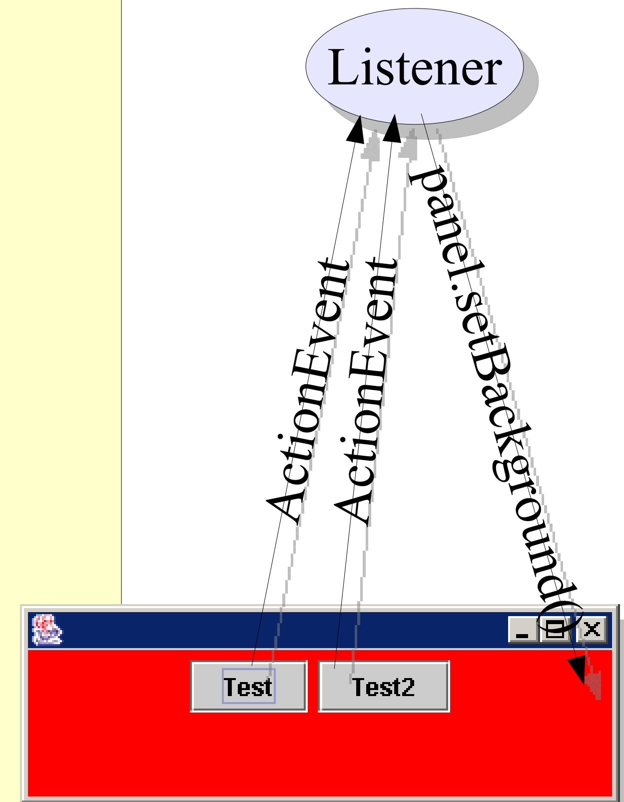
```
public class DontDoThis implements ActionListener {
  private final JPanel panel;

  public DontDoThis() {
    panel=new JPanel();
    JButton button=new JButton("Test");
    button.addActionListener(this);
    panel.add(button);
  }
  public void actionPerformed(ActionEvent event) {
    panel.setBackground(Color.RED);
  }
  public static void main(String[] args) {
    DontDoThis ddt=new DontDoThis();
    JFrame frame=new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setContentPane(ddt.panel);
    frame.setSize(300,200);
    frame.show();
  }
}
```



Conseil 2

- On stocke les objets auxquels on veux accéder dans le listener pas dans l'objet.

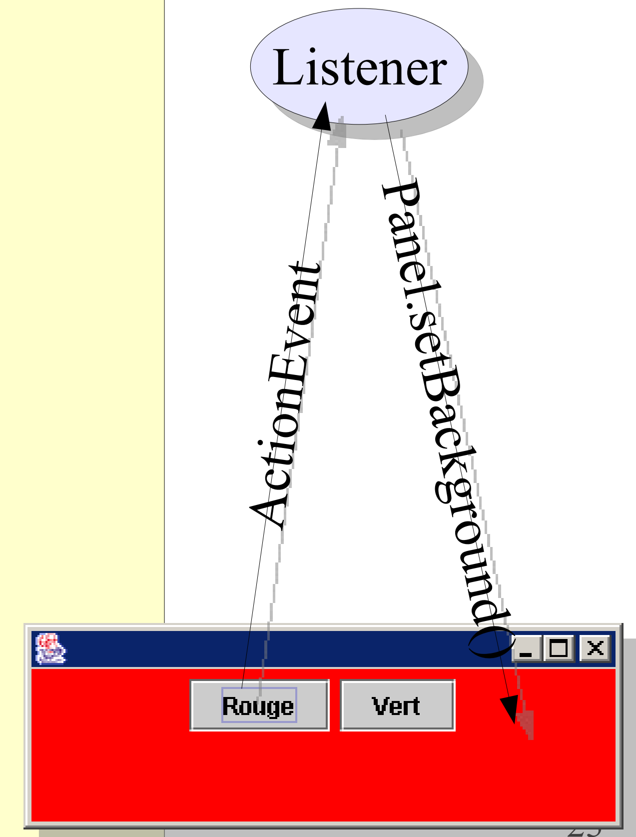
```
public class FieldInListener {
  static class MyActionListener implements ActionListener {
    public MyActionListener(JPanel panel) {
      this.panel=panel;
    }
    public void actionPerformed(ActionEvent event) {
      panel.setBackground(Color.RED);
    }
    private final JPanel panel;
  }
  public static void main(String[] args) {
    JPanel panel=new JPanel();

    JButton button=new JButton("Test");
    JButton button2=new JButton("Test2");

    ActionListener l=new MyActionListener(panel);
    button.addActionListener(l);
    button2.addActionListener(l);

    panel.add(button);
    panel.add(button2);
    ...
```



Conseil 3

- Pas de **if/switch** suivant la valeur d'un composant ou d'un texte dans un listener (pas objet)

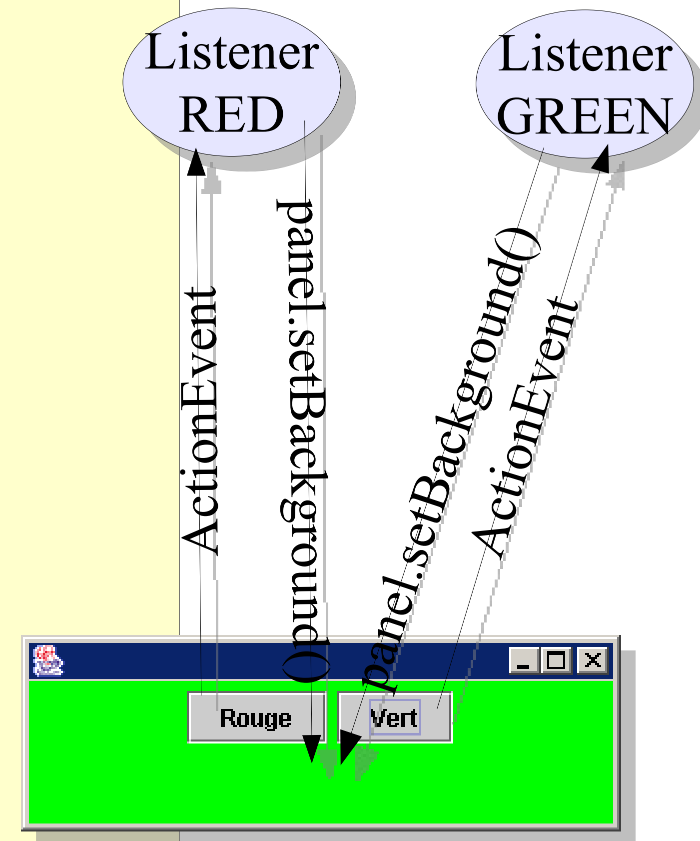
```
public class Beurk {
  static class MyActionListener implements ActionListener {
    public MyActionListener(JPanel panel) {
      this.panel=panel;
    }
    public void actionPerformed(ActionEvent event) {
      if (button==event.getSource())
        panel.setBackground(Color.RED);
      else
        panel.setBackground(Color.GREEN);
    }
    private final JPanel panel;
  }
  static JButton button, button2;
  public static void main(String[] args) {
    JPanel panel=new JPanel();
    button=new JButton("Rouge");
    button2=new JButton("Vert");
    ActionListener l=new MyActionListener(panel);
    button.addActionListener(l);
    button2.addActionListener(l);
```



Conseil 3 (suite)

- On passe ce qu'il faut au listener.

```
public class PasBeurk {
  static class MyActionListener implements ActionListener {
    public MyActionListener(JPanel panel,Color color) {
      this.panel=panel;
      this.color=color;
    }
    public void actionPerformed(ActionEvent event) {
      panel.setBackground(color);
    }
    private final JPanel panel;
    private final Color color;
  }

  public static void main(String[] args) {
    JPanel panel=new JPanel();

    JButton button=new JButton("Rouge");
    JButton button2=new JButton("Vert");

    button.addActionListener(
      new MyActionListener(panel,Color.RED));
    button2.addActionListener(
      new MyActionListener(panel,Color.GREEN));
    ...
```



Listener et classe anonyme

- Les classes anonymes permettent d'écrire des listeners de façon plus compacte
- Historiquement les classes anonymes ont été créées pour écrire les listeners

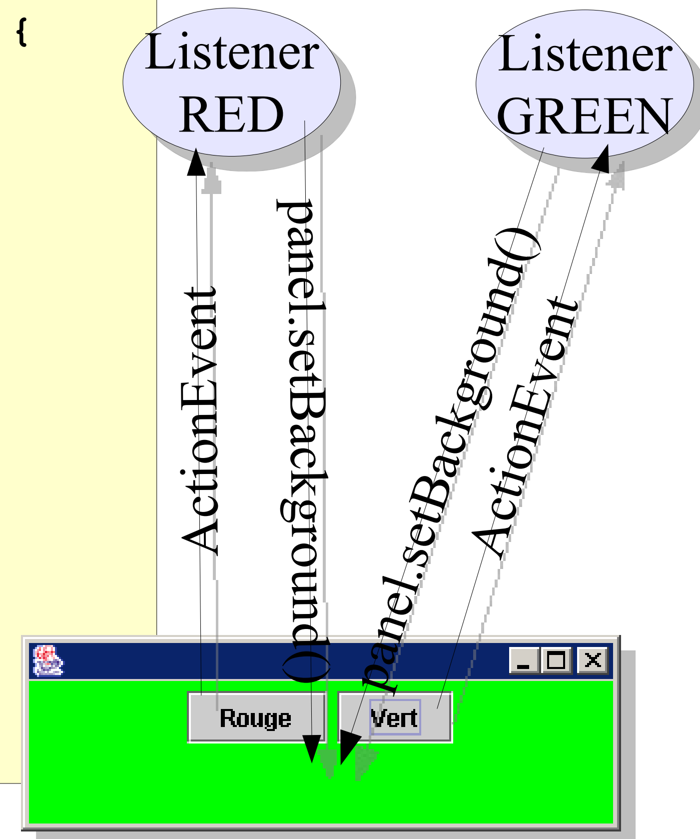
```
public class FirstExample {
  static class MyActionListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      System.out.println(event);
    }
  }
  public static void main(String[] args) {
    JButton button=new JButton("Test");

    ActionListener l=new MyActionListener();
    button.addActionListener(l);
    ...
```

```
public class AnonymousExample {
  public static void main(String[] args) {
    JButton button=new JButton("Test");

    ActionListener l=new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        System.out.println(event);
      }
    };
    button.addActionListener(l);
```

Listener et classe anonyme (2)

- Dans une classe anonyme il est possible d'accéder aux champs de la classe et aux **variables locales**.

```
public class AnonymousPasBeurk {
  private static void addButton(
    final JPanel panel,String text,final Color color) {

    JButton button=new JButton(text);
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        panel.setBackground(color);
      }
    });
    panel.add(button);
  }

  public static void main(String[] args) {
    JPanel panel=new JPanel();

    addButton(panel,"Rouge",Color.RED);
    addButton(panel,"Vert",Color.GREEN);
    ...
```



Conseil 4

- Quand doit-on utiliser une classe anonyme pour implanter un listener ?

- Réponse : **partout** où c'est possible.

- Limitation des classes anonymes :
  - ne peut pas implanté deux interfaces
  - ne peut pas implanté une classe et une interface.

Dans ces cas utiliser des classes internes de méthode

- Si l'action a effectuée dépend d'un état, mieux vaut utiliser une **Action** .

Evénement souris

- Les événements souris sont attrapés par trois listeners
  - MouseListener
    événement souris classique
  - MouseMotionListener
    événement souris lorsque la souris se déplace
  - MouseWheelListener
    événement lié au déplacement de la molette

```
public class FirstExample3 {
  static class MouseManager
    implements MouseListener, MouseMotionListener {
    ...
  }
  public static void main(String[] args) {
    ...
    MouseManager manager=new MouseManager();
    button.addMouseListener(manager);
    button.addMouseMotionListener(manager);
    ...
```

- Il est posible de s'enregistrer sur un des listener, sur deux, voir sur tous.

MouseListener

- L'interface MouseListener récupère les évènements liés à la souris :
  - **`void mousePressed(MouseEvent e)`**
    bouton appuyé
  - **`void mouseReleased(MouseEvent e)`**
    bouton relaché
  - **`void mouseClicked(MouseEvent e)`**
    bouton cliqué, peut être déclenché avant ou après le mouseReleased.
  - **`void mouseEntered(MouseEvent e)`**
    souris entre dans la zone d'un composant
  - **`void mouseExited(MouseEvent e)`**
    souris sort de la zone d'un composant

MouseAdapter

- Classe qui implante l'interface MouseListener et dont chaque méthode ne fait rien.

```
import java.awt.event.*;
import javax.swing.*;
public class FirstExample4 {
  static class MouseManager extends MouseAdapter {
    public void mouseClicked(MouseEvent event) {
      System.out.println("click");
    }
  }
  public static void main(String[] args) {
    JFrame frame=new JFrame();
    MouseManager manager=new MouseManager();
    Container contentPane=frame.getContentPane();
    contentPane.addMouseListener(manager);
  }
}
```

- Pratique car n'oblige pas à redéfinir toutes les méthodes.

MouseEvent

- Evénement sur l'état de la souris :
  - **`getX()/getY()/getPoint()`**
    coordonnées de la souris
  - **`getModifiersEx()`**
    masque (bits) des touches controle/shift/méta
  - **`getButton()`**
    soit NOBUTTON, BUTTON1, BUTTON2 ou BUTTON3
  - **`isPopupTrigger()`**
    indique un clique d'affichage de menu contextuel
    Note: suivant la plateforme doit être testé dans mousePressed **ou** dans mouseReleased

```
class MyMouseListener implements MouseListener {
  public void mouseClicked(MouseEvent event) {
    System.out.println(event.getPoint());
  }
  ...
}
```

MouseMotionListener & MouseWheelListener

- L'interface MouseMotionListener récupère les évènements liés aux déplacements de la souris :
  - **`void mouseMoved(MouseEvent e)`**
    la souris est déplacée
  - **`void mouseDragged(MouseEvent e)`**
    la souris est déplacée un bouton appuyé

- L'interface MouseWheelListener récupère les évènements liés à la molette.
  - **`mouseWheelMoved(MouseWheelEvent event)`**
  - Utilise un MouseWeelEvent !

Evènements de fenêtre

- Il existe deux *listeners* pour récuperer les évènements de fenêtre (icônification, fermeture, etc)
- **`WindowStateListener`**
  - **`windowStateChanged()`**, changement d'état
- **`WindowListener`**
  - **`windowOpened()`**, ouverture
  - **`windowClosing()`**, clique sur la croix
  - **`windowClosed()`**, fermeture
  - **`windowActivated()`**, activé (avant-plan)
  - **`windowDeactivated()`**, désactivé
  - **`windowIconified()`**, icônification
  - **`windowDeiconified()`**, dé-icônification

Evènements de fenêtre (2)

- La classe **WindowAdapter** implante l'interface **WindowListener**, chaque méthode ne fait rien.

```
public class WindowExample {
  static class Closer extends WindowAdapter {
    public WindowCloser(JFrame frame) {
      this.frame=frame;
    }
    public void windowClosed(WindowEvent e) {
      System.out.println("arg, je meurs");
    }
    public void windowClosing(WindowEvent e) {
      frame.dispose();
    }
    private final JFrame frame;
  }
  public static void main(String[] args) {
    final JFrame frame=new JFrame("WindowExample");
    frame.addWindowListener(new Closer(frame));
    frame.setSize(400,300);
    frame.setVisible(true);
  }
}
```

Libère la ressource de la plateforme

Norme Java Bean

Tout composant ***Swing*** est un JavaBean.

Un ***bean*** est un objet

- est capable d'**introspection**
- possède et gère des **propriétés**
- **expose** les propriétés
- communique par **évènements** les changements de propriétés
- réalise la **persistance** car sait se **sérialiser** (**`XMLEncoder`**/**`XMLDEcoder`**).

- Une Propriété
  - est **exposée** par `setXX()`, `getXX()`, `isXX()`
  - est **simple**, **liée** (bound) ou **contrainte**.
- Propriété **liée** envoit un `PropertyChangeEvent`, chaque fois qu'elle change, aux `PropertyChangeListener`.
- Propriété **contrainte** envoit un `PropertyChangeEvent`
  - juste avant qu'elle ne change,
  - et d'autres composants peuvent s'opposer (véto) au changement (`VetoableChangeListener`).

Evènements (Bean)

- `PropertyChangeEvent` a les méthodes
  - `getSource()`
  - `getOldValue()`
  - `getNewValue()`
  - `getPropertyName()`
- De plus, les composants ***Swing*** ont un événement "léger": `ChangeEvent`
- `ChangeEvent` n'a que la méthode `getSource()` (comme tout évènement).

Exemple (Bean)

- En installant un **PropertyChangeEvent** sur une frame, on obtient les évènements suivant :

```
Public static void main(String[] args) {
  JFrame frame=new JFrame();
  frame.addPropertyChangeListener(
    new PropertyChangeListener() {
      public void propertyChange(PropertyChangeEvent event) {
        System.out.println(event.getPropertyName()+" "+
          event.getOldValue()+" "+event.getNewValue());
    }
  });

  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setSize(300,200);
  frame.setVisible(true);
}
```