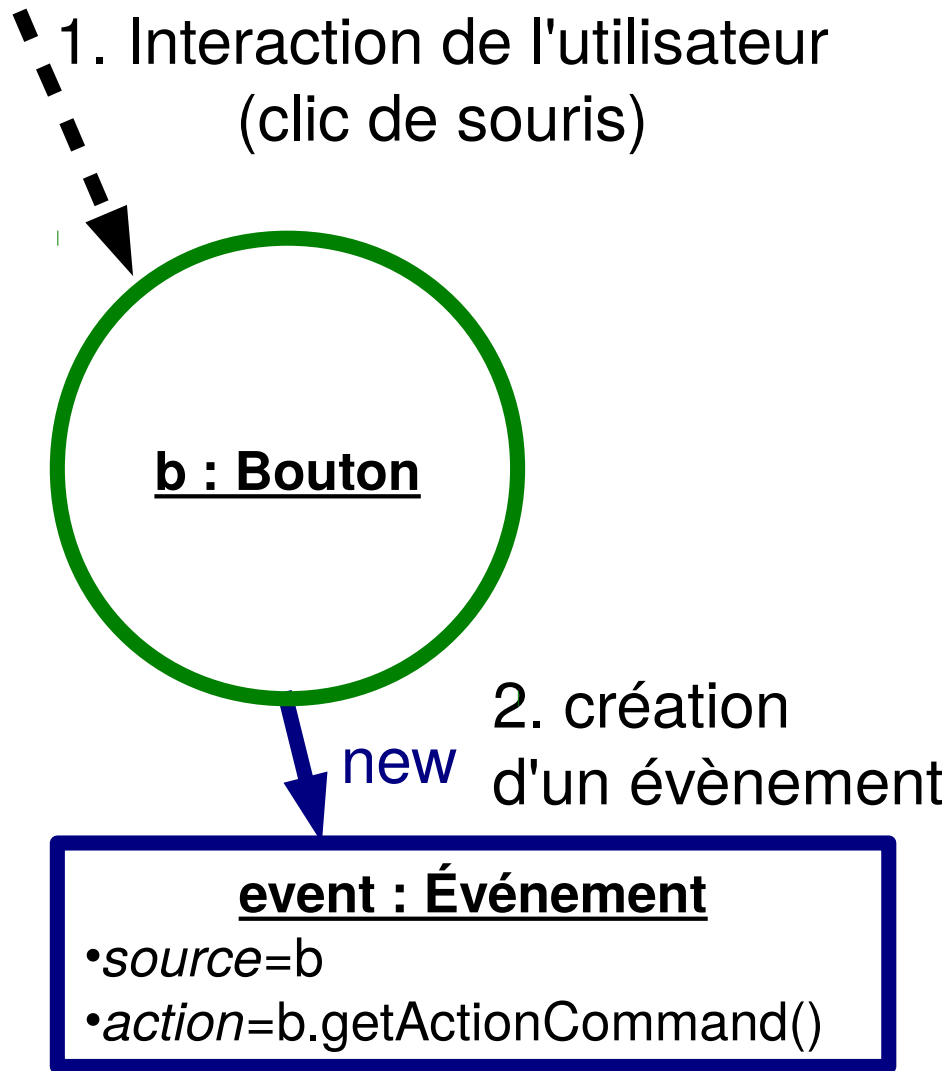


Programmation événementielle appliquée aux interfaces graphiques

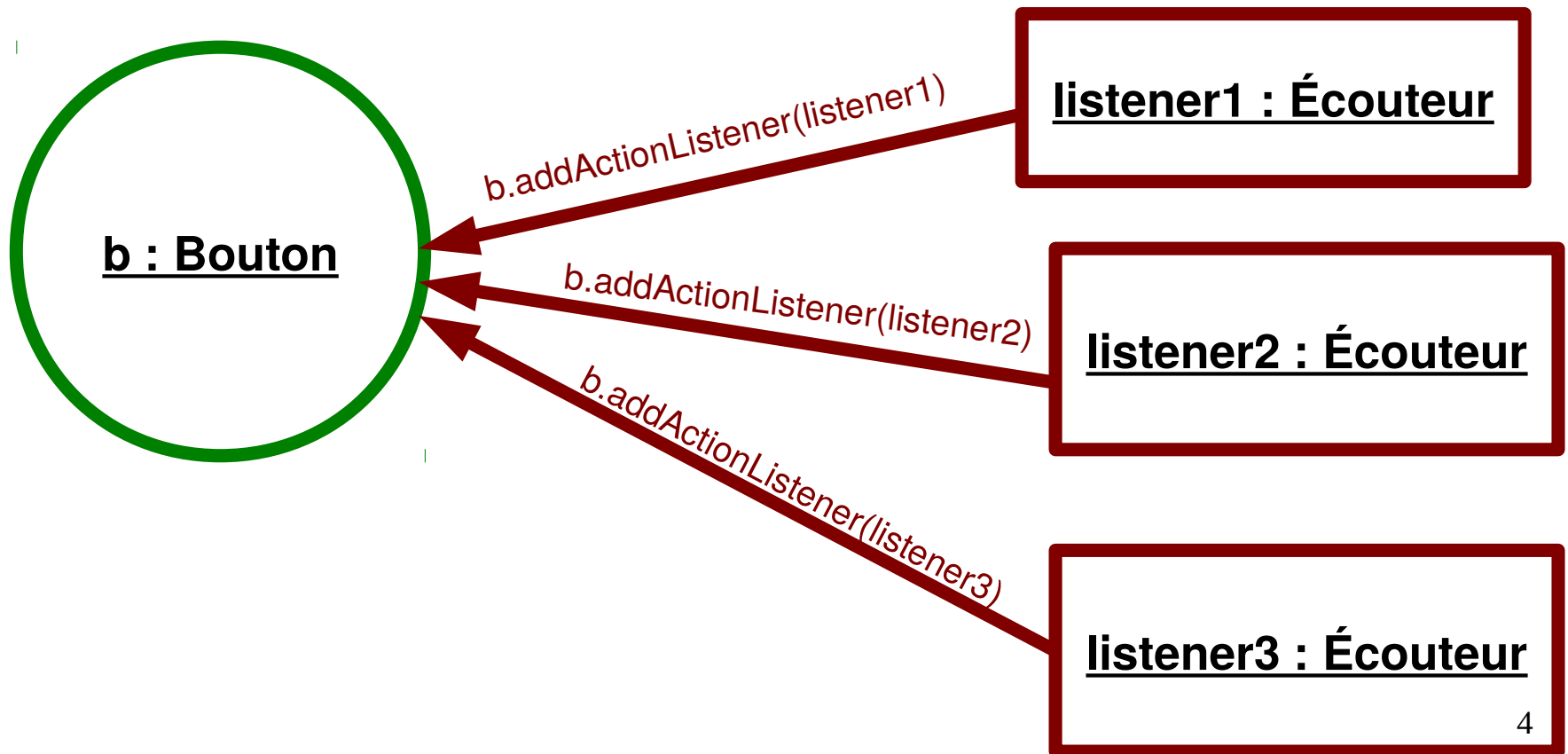
Résumé du cours précédent

le mécanisme des événements

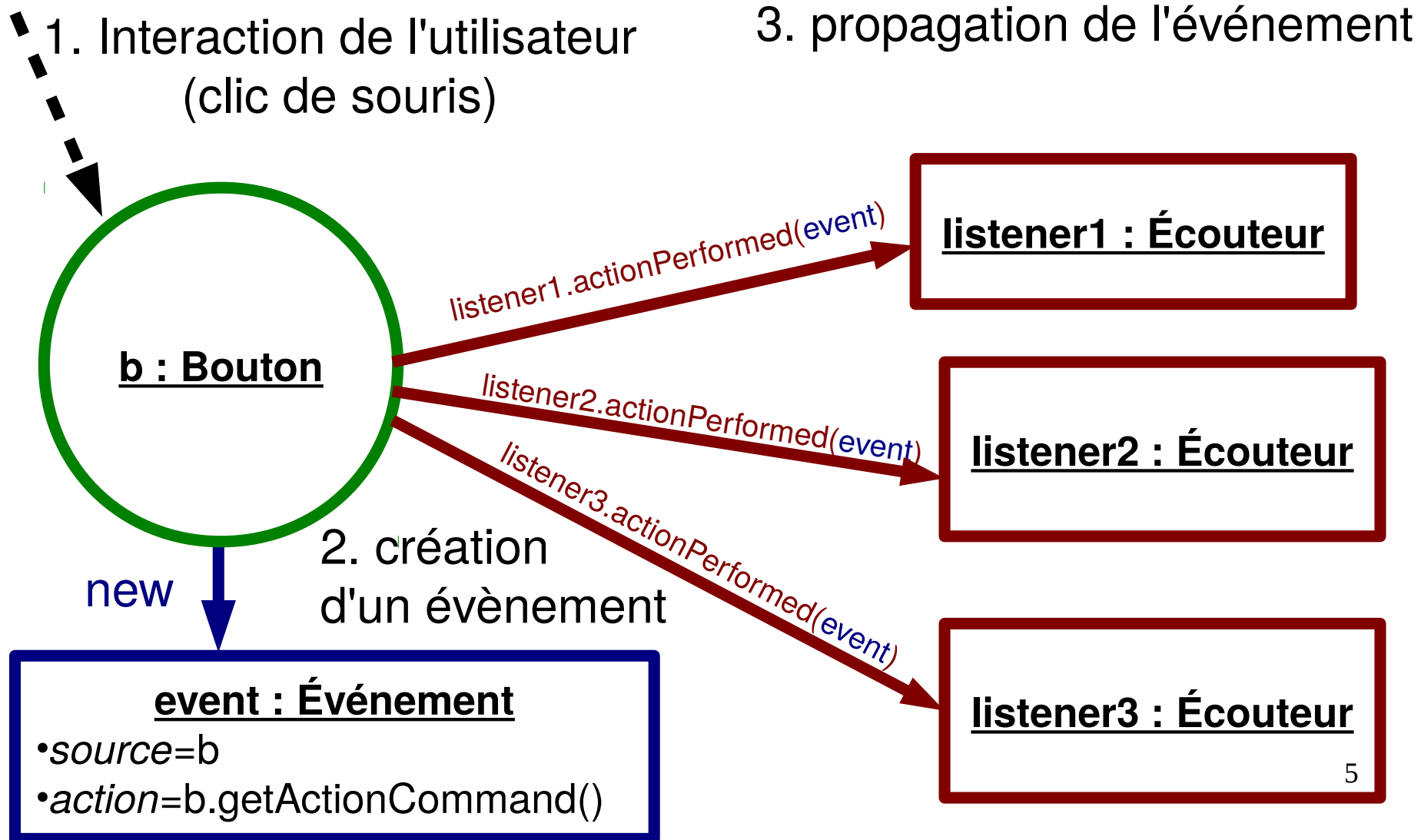
Un **événement** est un **objet** créé lorsqu'une action particulière est effectuée sur un **objet source**



Un **écouteur** est un **objet** réagissant aux événements d'un **objet source**. Pour qu'un objet puisse écouter un objet source, il doit s'enregistrer auprès de celui-ci



Lorsqu'un **objet source** crée un **événement**, il l'envoie à tous ses **écouters**



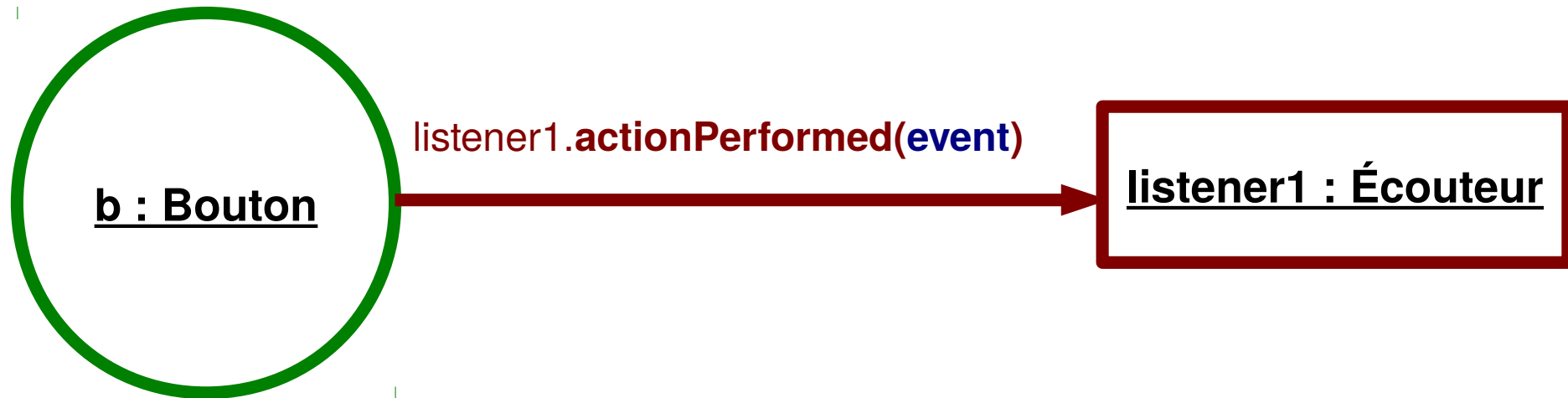
Rappel du mécanisme général

- Un objet qui peut être **source** d'événements possède une **liste d'objets écouteurs** (et même plusieurs de plusieurs types)
- Lorsque le système l'avertit d'une certaine action :
 - il génère un **événement** d'un certain type,
 - il déclenche la **méthode appropriée**(prévue dans l'interface correspondante)
sur **tous les objets de cette liste**
en passant l'événement en paramètre

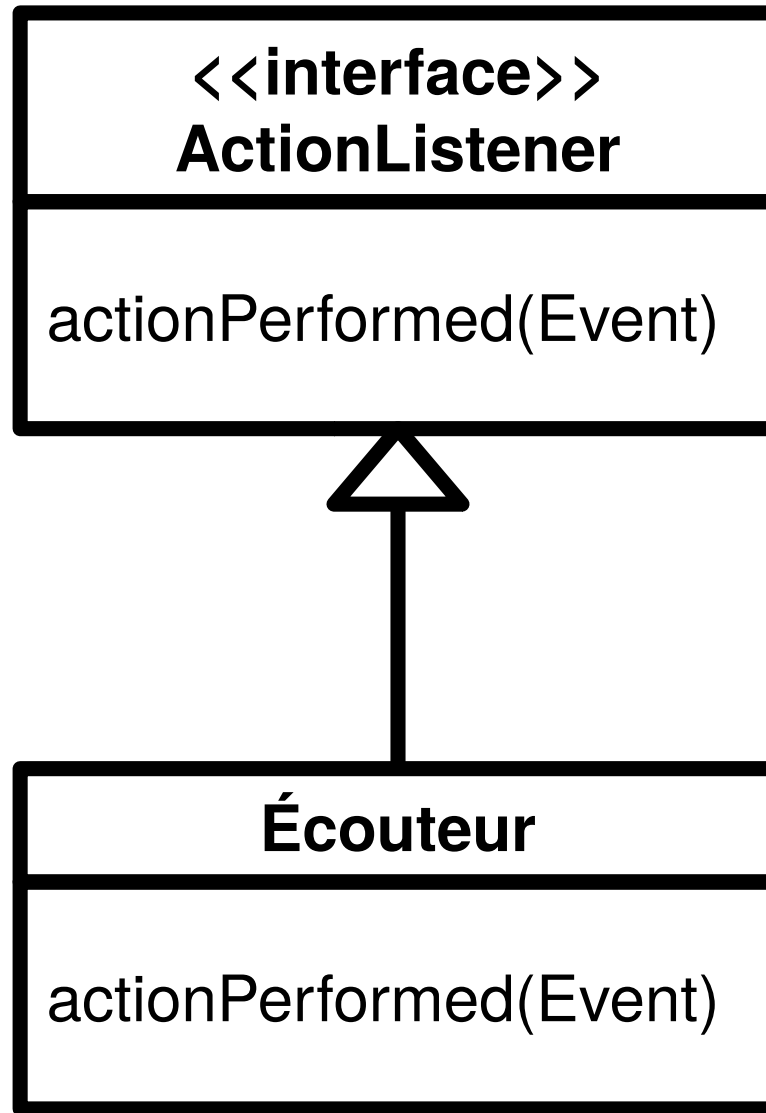
Résumé du cours précédent

les écouteurs

Lorsqu'un **objet source** crée un **événement**, il l'envoie à tous ses **écouteurs**. Il est donc nécessaire que tous les écouteurs implémentent la méthode traitant l'envoi.



La méthode d'envoi est déclarée dans une **interface** à **implémenter** par tous les écouteurs.



Dans certains cas, le nombre de méthodes dans l'interface à implémenter peut être important. Des classes Adapter sont alors prévues dans l'API Java qui les implémentent mais sans code.

```
public class MouseAdapter implements MouseListener {  
    public void mouseClicked(MouseEvent e) {}  
    public void mouseEntered(MouseEvent e) {}  
    public void mouseExited(MouseEvent e) {}  
    public void mousePressed(MouseEvent e) {}  
    public void mouseReleased(MouseEvent e) {}  
}
```

Résumé du cours précédent

les interfaces graphiques

Il existe 2 librairies pour créer des interfaces graphiques dans l'API java :

- **AWT** (java.awt): librairie complète (composants graphiques, événements, layout managers ...).
- **SWING** (javax.swing) : réécriture des composants graphiques uniquement.

Il est fortement déconseillé de mélanger des **composants graphiques SWING et AWT**.

Ajout d'éléments à une fenêtre

- Spécialisation de la classe JFrame
- Déclaration du LayoutManager
- Ajout d'éléments au container de la fenêtre



```
public class Essai2 extends JFrame {  
  
    public Essai2() {  
        this.setTitle("essai2");  
        this.setLayout(new BorderLayout());  
        JButton boutonHaut=new JButton();  
        JPanel jpCentre=new JPanel();  
        JPanel jpSud=new JPanel();  
        this.add( boutonHaut, BorderLayout.NORTH);  
        this.add( jpCentre, BorderLayout.CENTER);  
        this.add( jpSud, BorderLayout.SOUTH);  
        ...  
    }  
}
```

Ajout d'éléments à une fenêtre

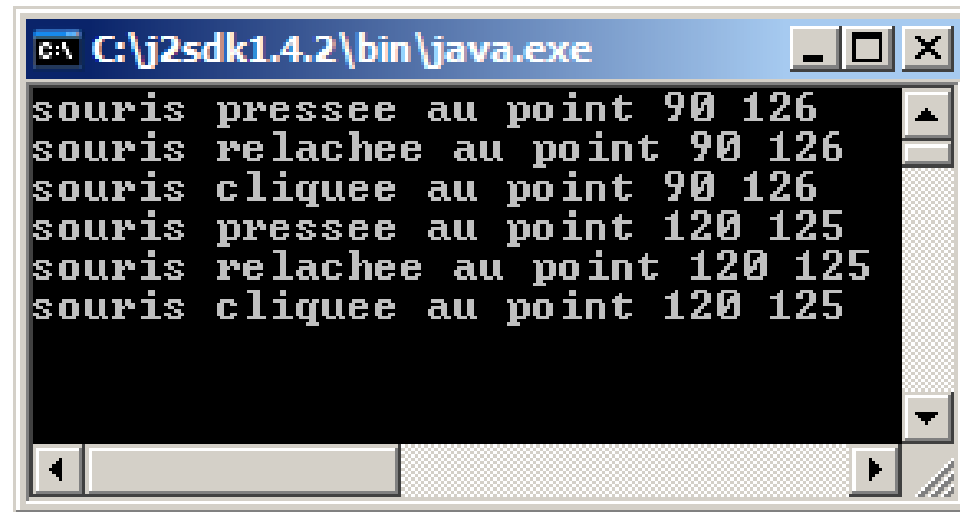
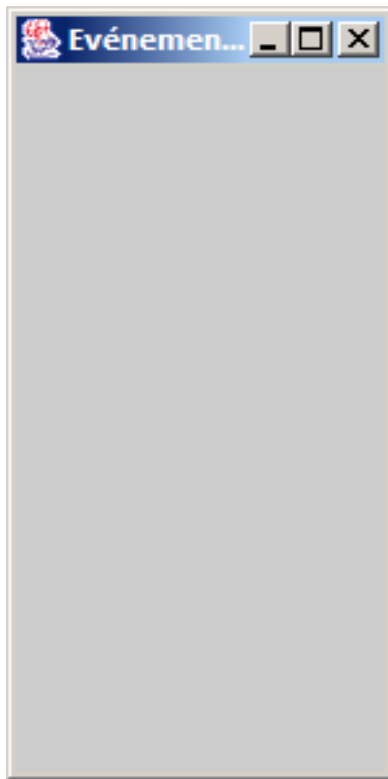
- Spécialisation de la classe JFrame
- Déclaration du LayoutManager
- Ajout d'éléments au container de la fenêtre



```
public class Essai2 extends JFrame {  
    ...  
    jpCentre.setLayout(new BorderLayout());  
    jpCentre.add(new JButton("numero 0"),  
                  BorderLayout.NORTH);  
    jpCentre.add(new JButton("numero 1"),  
                  BorderLayout.WEST);  
    jpCentre.add(new JButton("numero 2"),  
                  BorderLayout.CENTER);  
    jpCentre.add(new JButton("numero 3"),  
                  BorderLayout.EAST);  
    jpCentre.add(new JButton("numero 4"),  
                  BorderLayout.SOUTH);  
  
    jpSud.add(new JButton("bas"));  
    ...  
}
```

implémentation des écouteurs

Exemple : quitter l'application quand on ferme la fenêtre graphique



Créer un **écouteur des événements** de type **WindowEvent** (événements de haut niveau) sur cette fenêtre

Interface *java.awt.event.WindowListener*

Method Summary

void	<u>windowActivated</u> (WindowEvent e) Invoked when the window is set to be the user's active window, which means the window (or one of its subcomponents) will receive keyboard events.
void	<u>windowClosed</u> (WindowEvent e) Invoked when a window has been closed as the result of calling dispose on the window.
void	<u>windowClosing</u> (WindowEvent e) Invoked when the user attempts to close the window from the window's system menu.
void	<u>windowDeactivated</u> (WindowEvent e) Invoked when a window is no longer the user's active window, which means that keyboard events will no longer be delivered to the window or its subcomponents.
void	<u>windowDeiconified</u> (WindowEvent e) Invoked when a window is changed from a minimized to a normal state.
void	<u>windowIconified</u> (WindowEvent e) Invoked when a window is changed from a normal to a minimized state.
void	<u>windowOpened</u> (WindowEvent e) Invoked the first time a window is made visible.

implémentation des écouteurs

la fenêtre écoute

Si c'est la fenêtre qui écoute ses propres événements :

```
public class Fenetre extends JFrame
                        implements WindowListener
{
    .....
    public Fenetre() // constructeur
    {
        addWindowListener(this);
    }

    public void windowClosing (WindowEvent e)
    { System.exit(0); }

    // et les 6 autres méthodes de l'interface
    // WindowListener avec un corps vide

    .....
}
```

implémentation des écouteurs

un objet dédié écoute

Si c'est un objet dédié qui écoute les événements de la fenêtre :

```
public class Fenetre extends JFrame
{
    .....

    public Fenetre() // constructeur
    {
        .....
        Terminator t = new Terminator();
        addWindowListener(t);
    }
    .....
}
```

*Ecrivons la classe **Terminator***

```
public class Terminator
    implements WindowListener
{
    public void windowClosing (WindowEvent e)
    { System.exit(0); }

    // et les 6 autres méthodes avec un corps
    vide }
```

Ouf! On peut dériver de la classe **WindowAdapter**

```
public class Terminator extends WindowAdapter
{
    public void windowClosing (WindowEvent e)
    { System.exit(0); }
}
```

Revenons à la classe Fenetre :

```
public Fenetre() // constructeur
{
    .....
    Terminator t = new Terminator();
    addWindowListener(t);
}
```

La référence **t** n'est pas utile

```
addWindowListener(new Terminator());
```

Le nom **Terminator** non plus....

on fait de Terminator une classe **interne anonyme**

implémentation des écouteurs

les classes internes


```
public class Fenetre extends JFrame{  
  
class Terminator extends WindowAdapter  
{  
    public void windowClosing (WindowEvent e)  
    { System.exit(0); }  
}  
  
public Fenetre() // constructeur  
{  
    .....  
    Terminator t = new Terminator();  
    addWindowListener(t);  
}
```

Problème : On a gagné un fichier .java, mais le nom Terminator n'est toujours pas utile pour une seule instance...

```
public class Fenetre extends JFrame{

    public Fenetre() // constructeur
    {
        .....

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

Classe interne ANONYME

Problème

Lorsque l'utilisateur actionne le bouton, le panneau doit prendre une couleur différente

(par exemple, en boucle : noir, bleu, jaune, ...)

action



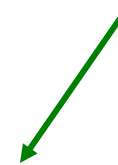
action



action



action



```
public class PanneauBouton extends JPanel
{
    private JButton b;

    public PanneauBouton()
    {
        b = new JButton("Couleurs");
        add(b);
        setBackground(Color.white);
        b.addActionListener(new EcouteBouton(this));
    }
}
```

Notez que l'instance de **EcouteBouton**
est créée par une instance de **PanneauBouton**

```

class EcouteBouton implements ActionListener
{
    private static Color[] tCol = {Color.black,
    Color.blue, Color.yellow};
    private int numCol = -1;
    private JPanel p;

    public EcouteBouton(JPanel p)
    { this.p = p; }

    public void actionPerformed(ActionEvent e)
    {
        numCol = (numCol + 1) % tCol.length;
        p.setBackground(tCol[numCol]);
    }
}

```

Si **EcouteBouton** devient une **classe interne** à **PanneauBouton**, une instance de **EcouteBouton** aura accès aux **attributs et méthodes** de l'instance de **PanneauBouton** qui l'a créée

```

public class PanneauBouton extends JPanel
{
    private JButton b;
    private static Color[] tCol = {Color.black,
    Color.blue, Color.yellow};
    private int numCol = -1;
    public PanneauBouton()
    {
        b = new JButton("Couleurs");
        this.add(b);
        setBackground(Color.white);
        b.addActionListener(new EcouteBouton());
    }
    class EcouteBouton implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            numCol = (numCol + 1) % tCol.length;
            setBackground(tCol[numCol]);
        }
    }
}

```

Puisque la classe `EcouteBouton` ne sert qu'à créer **une** instance, on peut en faire une classe **interne anonyme** :

c'est-à-dire **définie « à la volée »** lors de la création de l'instance de cette classe

```
b.addActionListener(new EcouteBouton() );
```

```
class EcouteBouton implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        numCol = (numCol + 1) % tCol.length;
        setBackground(tCol[numCol]);
    }
}
```

*On remplace le **nom** de la classe par le **nom de la classe dont elle dérive** ou de l'interface qu'elle implémente, et on met le « corps » de la classe après les paramètres du constructeur*

Classes internes

- Classe interne : classe définie à l'intérieur d'une classe

Intérêts

- Une instance d'une classe interne a des **privileges** : elle peut accéder aux **attributs et méthodes de l'instance de la classe englobante** qui l'a créée, même s'ils sont privés
- Une classe interne peut être **cachée** aux autres classes, même à celles du même package (on peut en faire une classe **privée**)
- Les classes internes (et anonymes!) sont très pratiques pour définir des **écouteurs d'événements**

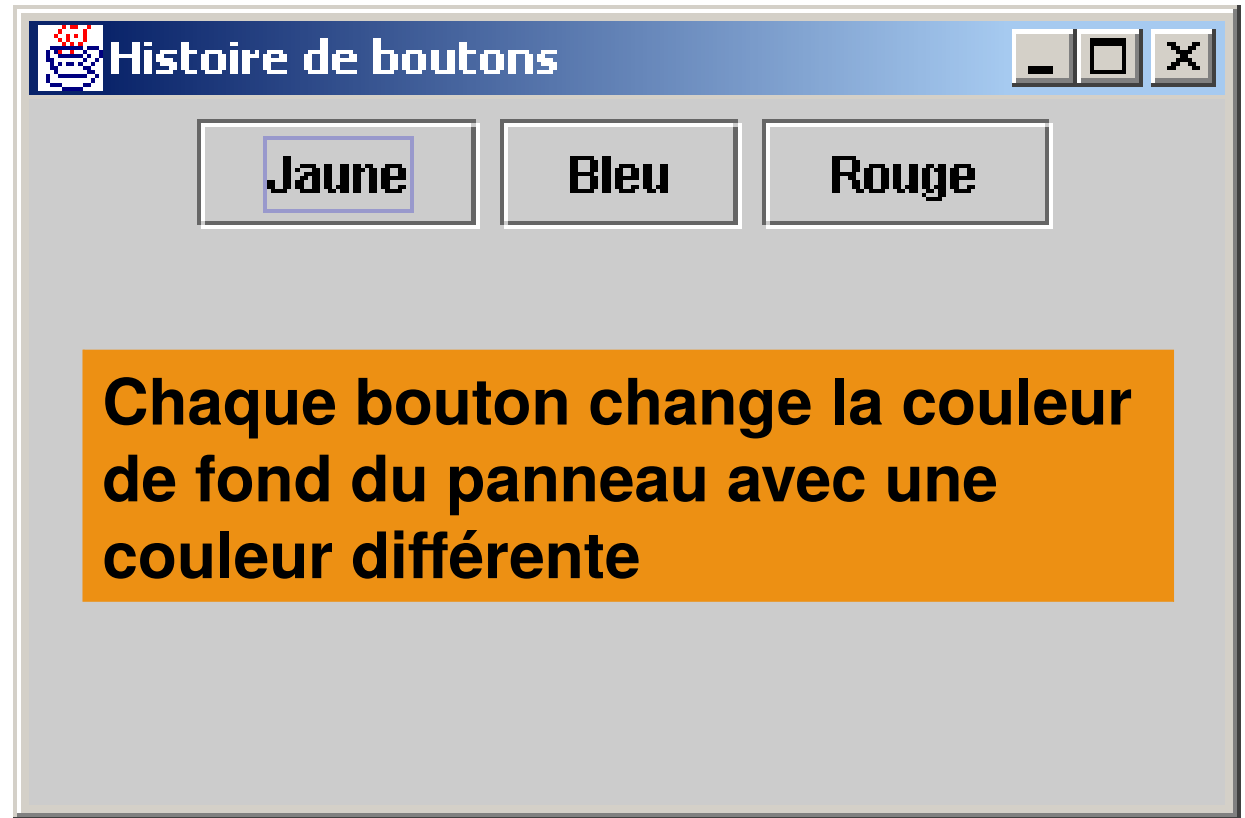
Attention : complexifie rapidement le code

On restreindra l'utilisation des classes internes au point 3


```
b.addActionListener(new EcouteBouton () );
```

```
b.addActionListener(new ActionListener()  
{  
    public void actionPerformed(ActionEvent e)  
    {  
        numCol = (numCol + 1) % tCol.length;  
        setBackground(tCol[numCol]);  
    }  
});
```

Exemple : écouter 3 boutons



Qui écoute ?

1. le panneau
2. un objet d'une classe dédiée écoute les 3 boutons
3. Chaque bouton est écouté par un objet différent

Dans les cas 1 et 2 se pose le problème
de la reconnaissance du bouton source de l'événement

Deux méthodes :

- `java.util.EventObject`

```
public Object getSource()
```

```
// retourne (une référence sur) l'objet source
```

- `java.awt.event.ActionEvent`

```
public String getActionCommand()
```

```
// retourne une chaîne de caractères
```

```
// c'est par défaut le label du bouton
```

```
// (au moment de l'action)
```

```
// on peut modifier cette valeur par défaut par la
```

```
// méthode setActionCommand du bouton
```

```
class PanneauBoutons extends JPanel  
                        implements ActionListener
```

1

```
{  
    private JButton bJaune, bBleu, bRouge;  
    public PanneauBoutons()  
    {  
        bJaune = new JButton("Jaune"); add(bJaune);  
        bBleu = new JButton("Bleu"); add(bBleu);  
        bRouge = new JButton("Rouge"); add(bRouge);  
        bJaune.addActionListener(this);  
        bBleu.addActionListener(this);  
        bRouge.addActionListener(this);  
    }  
    public void actionPerformed(ActionEvent e)  
    {  
        Object source = e.getSource();  
        Color c = null;  
        if (source == bJaune) c = Color.yellow;  
        else if (source == bBleu) c = Color.blue;  
        else c = Color.red;  
        setBackground(c);  
    }  
}
```

class EcouteBoutons implements ActionListener

{ private JPanel p; *//référence sur le panneau*

public EcouteBoutons(JPanel p){ this.p = p; }

public void actionPerformed(ActionEvent e)

{

String command = e.getActionCommand();

Color c = null;

if (command.equals("Jaune")) c=Color.yellow;

else if (command.equals("Bleu")) c=Color.blue;

else c=Color.red;

p.setBackground(c);

}

}

Dans le constructeur du panneau :

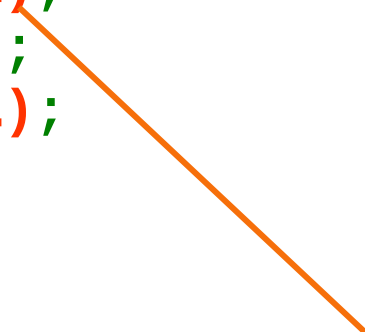
EcouteBoutons ecouteur = new EcouteBoutons(this);

bJaune.addActionListener(ecouteur);

bBleu.addActionListener(ecouteur);

bRouge.addActionListener(ecouteur);

```
class PanneauBoutons extends JPanel
{
    private JButton bJaune, bBleu, bRouge;
    public PanneauBoutons()
    {
        bJaune = new JButton("Jaune"); add(bJaune);
        bBleu = new JButton("Bleu"); add(bBleu);
        bRouge = new JButton("Rouge"); add(bRouge);
        bJaune.addActionListener(...);
        bBleu.addActionListener(...);
        bRouge.addActionListener(...);
    }
}
```



```
bJaune.addActionListener( new ActionListener()
{ public void actionPerformed(ActionEvent e)
    { setBackground(Color.yellow); }
}
);
```

Idem pour les autres boutons en changeant la couleur

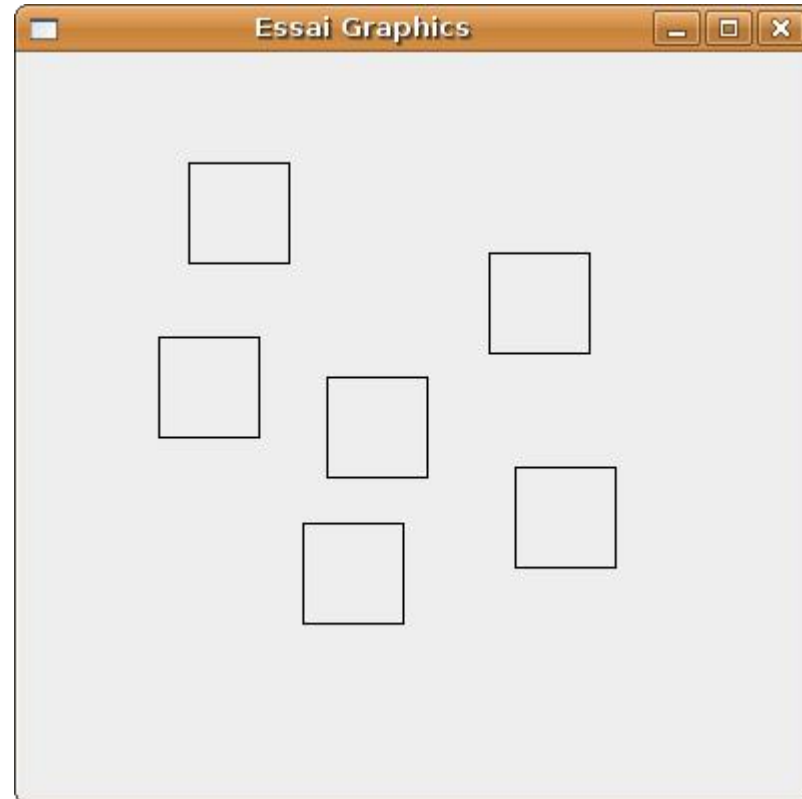
Dessiner sur des composants

Dessiner sur des composants

Le dessin se fait par un « contexte graphique »

- instance d'une classe dérivée de **Graphics** ou **Graphics2D**
- **associé au composant** sur lequel on veut dessiner (donc connaissant ses coordonnées à l'écran)
- **gérant les outils de dessin** : sélection d'une couleur de dessin, d'une police, ...
- **sachant tracer des formes** sur le composant : une ligne, un rectangle, un rectangle plein, une ellipse, une chaîne de caractères, ...

Exemple : Dans une fenêtre, un clic sur un panneau doit faire apparaître un carré à l'endroit du clic.



```
public class EssaiGraphics extends JFrame {  
  
    EssaiGraphics()  
    {  
        setTitle("Essai Graphics");  
        add(new Panneau());  
        setSize(400,400);  
        setVisible(true);  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
    }  
  
    public static void main(String[] args) {  
        EssaiGraphics essai=new EssaiGraphics();  
    }  
}
```

```
public class Panneau extends JPanel{

public Panneau(){
    this.addMouseListener(
        new MouseAdapter(){

            public void mouseClicked(MouseEvent e) {
                Graphics g=Panneau.this.getGraphics();
                g.drawRect(e.getX()-25, e.getY()-25, 50, 50);
            }
        });
}
```

Je récupère l'environnement graphique et je dessine dessus un rectangle à l'endroit du clic.

Problème : quand le panneau doit être redessiné (expl : redimensionnement du panneau), tout ce que j'ai dessiné s'efface.

A chaque fois qu'un composant graphique a besoin d'être repeint, **appel automatique** de la méthode paint(), qui elle même fait appel à :

```
public void paintComponent(Graphics g)
{
    ici, ce qu'il faut faire
    pour repeindre le composant
}
```

ATTENTION ! Ne jamais redéfinir la méthode paint(),
toujours passer par paintComponent

L'objet Graphics est passé en paramètre lors de l'appel
– automatique – de la méthode paintComponent

Code hors contrôle du programmeur :

```
Graphics g =  
panneau.getGraphics();  
panneau.paintComponent(g);  
g.dispose();
```

ATTENTION !

Ceci n'est pas fait pas le programmeur.

Ne **JAMAIS** appeler explicitement paintComponent

```
public class Panneau extends JPanel{
    ArrayList<Rectangle> figures=new ArrayList<Rectangle>();
    public Panneau(){
        addMouseListener(
            new MouseAdapter(){
                public void mouseClicked(MouseEvent e) {
                    figures.add(
                        new Rectangle(e.getX()-25, e.getY()-25, 50, 50));
                }
            });
    }
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (Rectangle r: figures) {
            g.drawRect((int)r.getX(), (int)r.getY(),
                (int)r.getWidth(), (int)r.getHeight());
        }
    }
}
```

Problème : ça fonctionne « bien », **mais** je ne vois les carrés apparaître **que** lorsque je redimensionne la fenêtre ...

Solution : il faut demander au panneau de se redessiner dès que possible après l'évènement grâce à la méthode **repaint()** qui se chargera d'appeler **paintComponent()**.

```
public class Panneau extends JPanel{

    public Panneau(){
        addMouseListener(
            new MouseAdapter(){
                public void mouseClicked(MouseEvent e) {
                    figures.add(
                        new Rectangle(e.getX()-25, e.getY()-25, 50, 50));
                    Panneau.this.repaint();
                }
            });
    }
    ...
}
```

ATTENTION !

Ne **JAMAIS** appeler explicitement paint()