

POO-IG

Programmation Orientée Objet et Interfaces Graphiques

Cristina Sirangelo
IRIF, Université Paris Cité
cristina@irif.fr

Éléments d'interfaces graphiques

Exemples et matériel empruntés :

- * Transparents de cours de H.Fauconnier
- * Core Java - C.Horstmann - Prentice Hall Ed.
- * POO in Java - L.Nigro & C.Nigro - Pitagora Ed.

Principes de base

- Dans une interface graphique (IG ou GUI), le programme réagit aux **interactions** avec l'utilisateur
- Les interactions génèrent des **événements** (click de la souris, bouton sélectionné etc.)
- Le programme réagit à ces événements en exécutant des méthodes de réponse (**actions**)
- On dit que l'exécution est dirigée par les événements (**event-driven**)

Principes de base

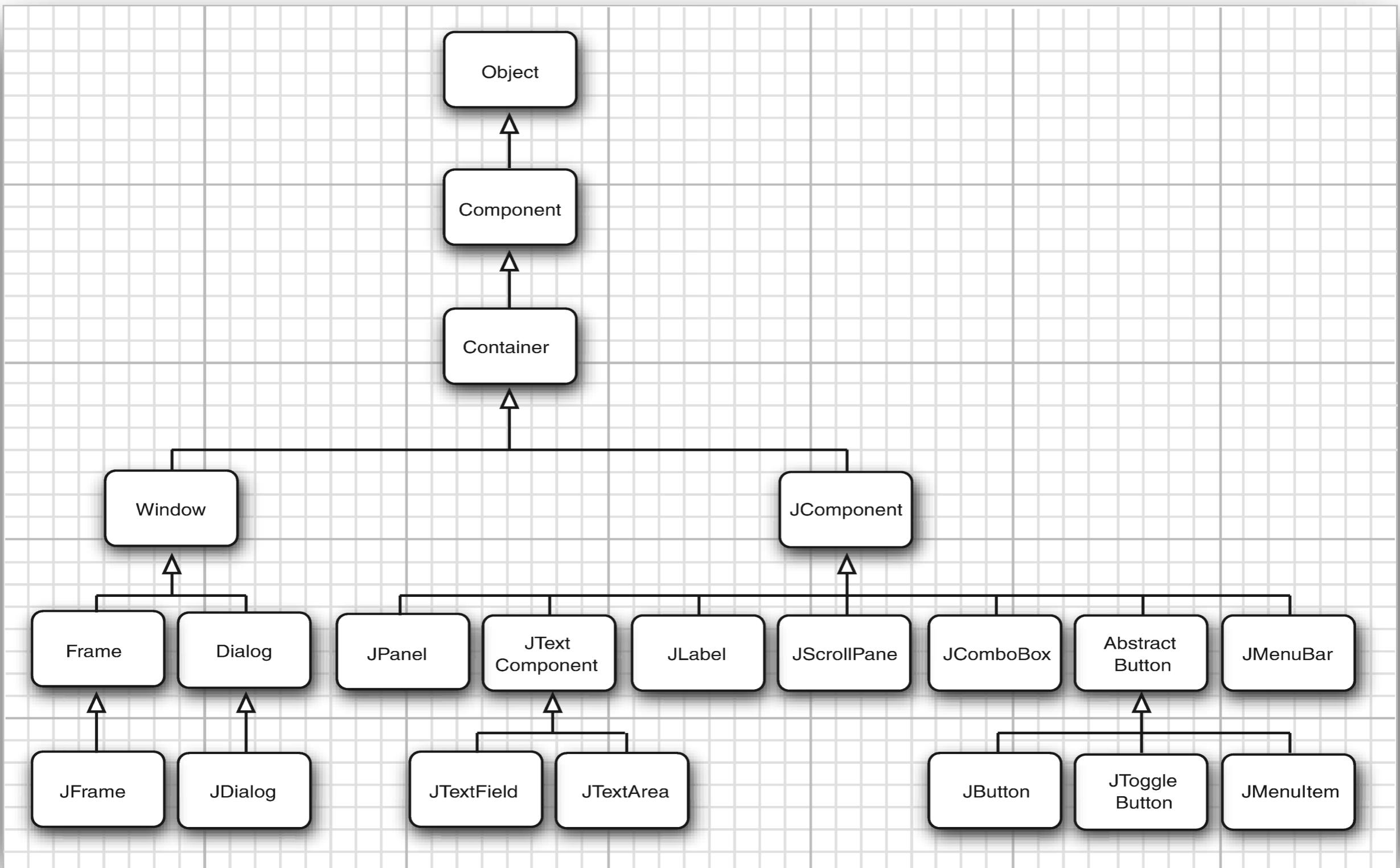
- Pour développer une interface graphique d'un programme:
 - Définir les **composants graphiques** (fenêtres, boutons, champs de texte etc) - instances de classes
 - Les placer
 - soit "à la main" (i.e. avec les méthodes d'une classe **Layout Manager**)
 - soit en "wysiwyg" en utilisant des **IDE** comme Eclipse ou Netbeans qui génèrent le code Java correspondant
 - Définir les actions associées aux événements (**Listeners**) et les associer aux composants graphiques
- La logique du programme sera dans un ensemble de classes le plus possible séparées des classes de l'interface graphique
- Les listeners font le lien entre les deux

Les composantes graphiques

- Toute une hiérarchie de classes
 - packages `java.awt` et `javax.swing`
 - `javafx` censé remplacer `swing` à terme, mais supprimé du JDK depuis Java 11 (2018)
 - devenu un projet indépendant

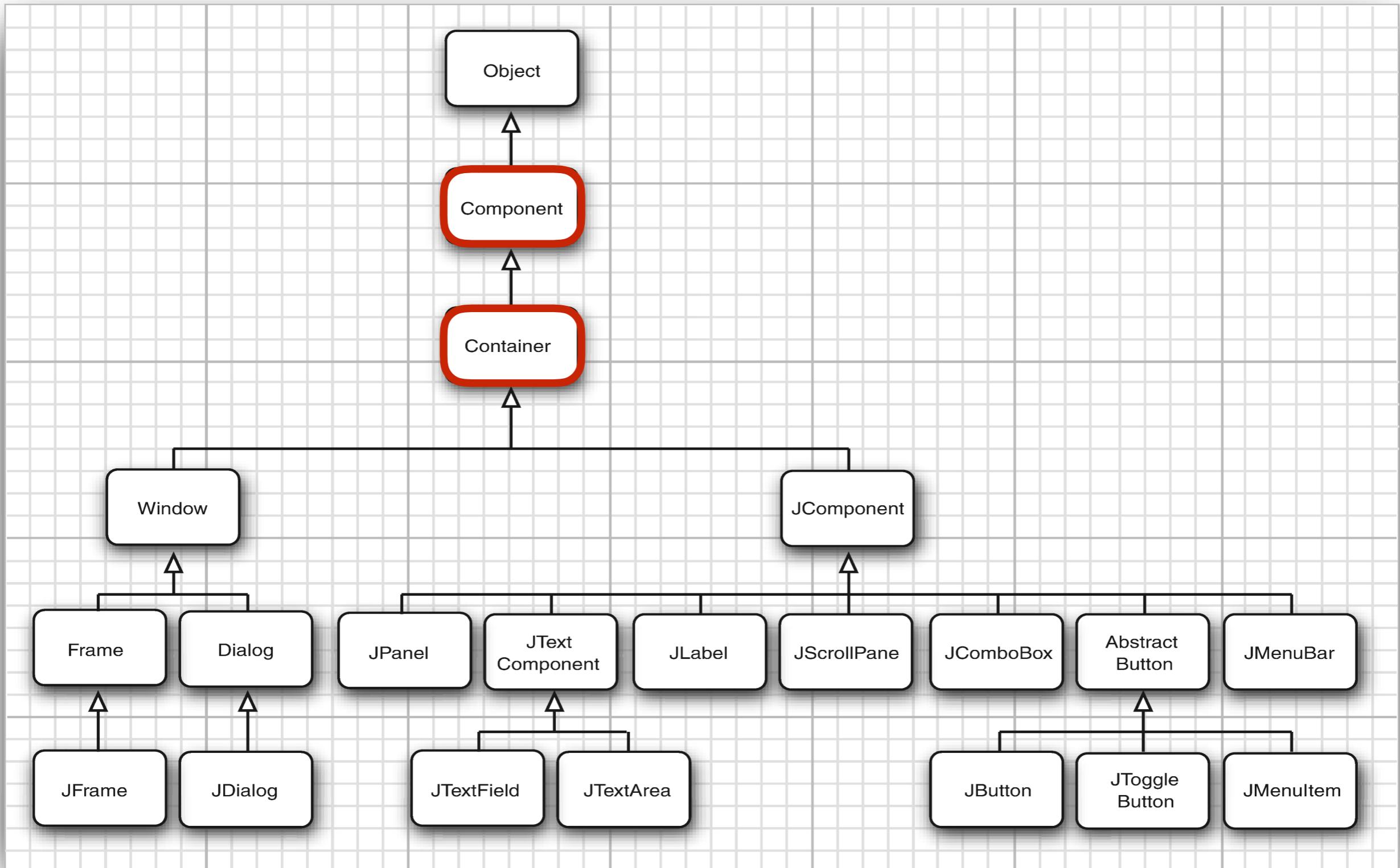
Les composantes graphiques

- Toute une hiérarchie de classes



Les composantes graphiques

Classes de base : Component et Container

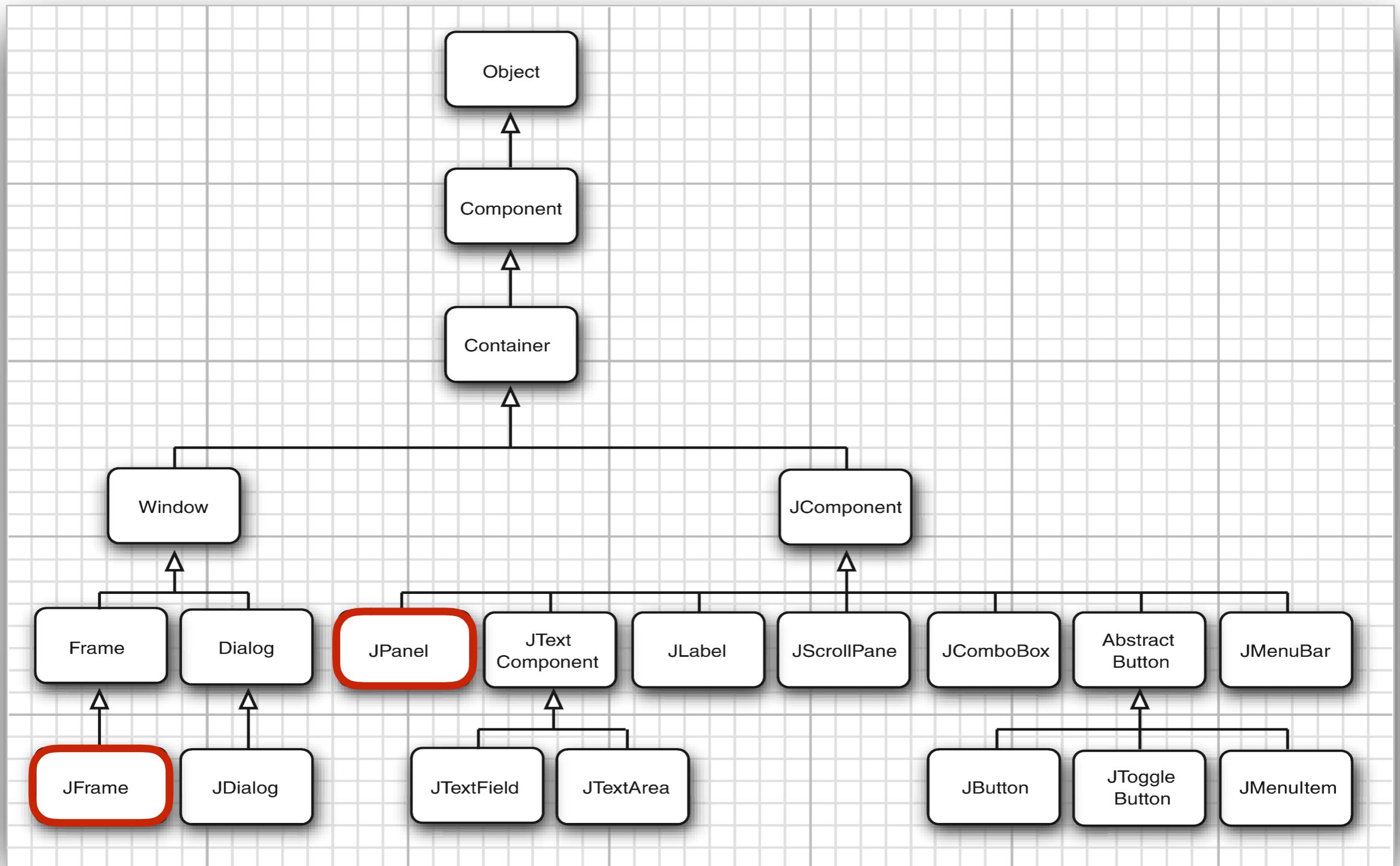


Component et Container

- Classes de base : Component et Container
 - Component : les composants graphiques
 - Container : les composants graphiques qui peuvent en contenir d'autres (e.g. fenêtre, menu, panneau, ...)
- (Quelques incohérence dans la hiérarchie : e.g bouton est un container...)

Conteneurs de plus haut niveau

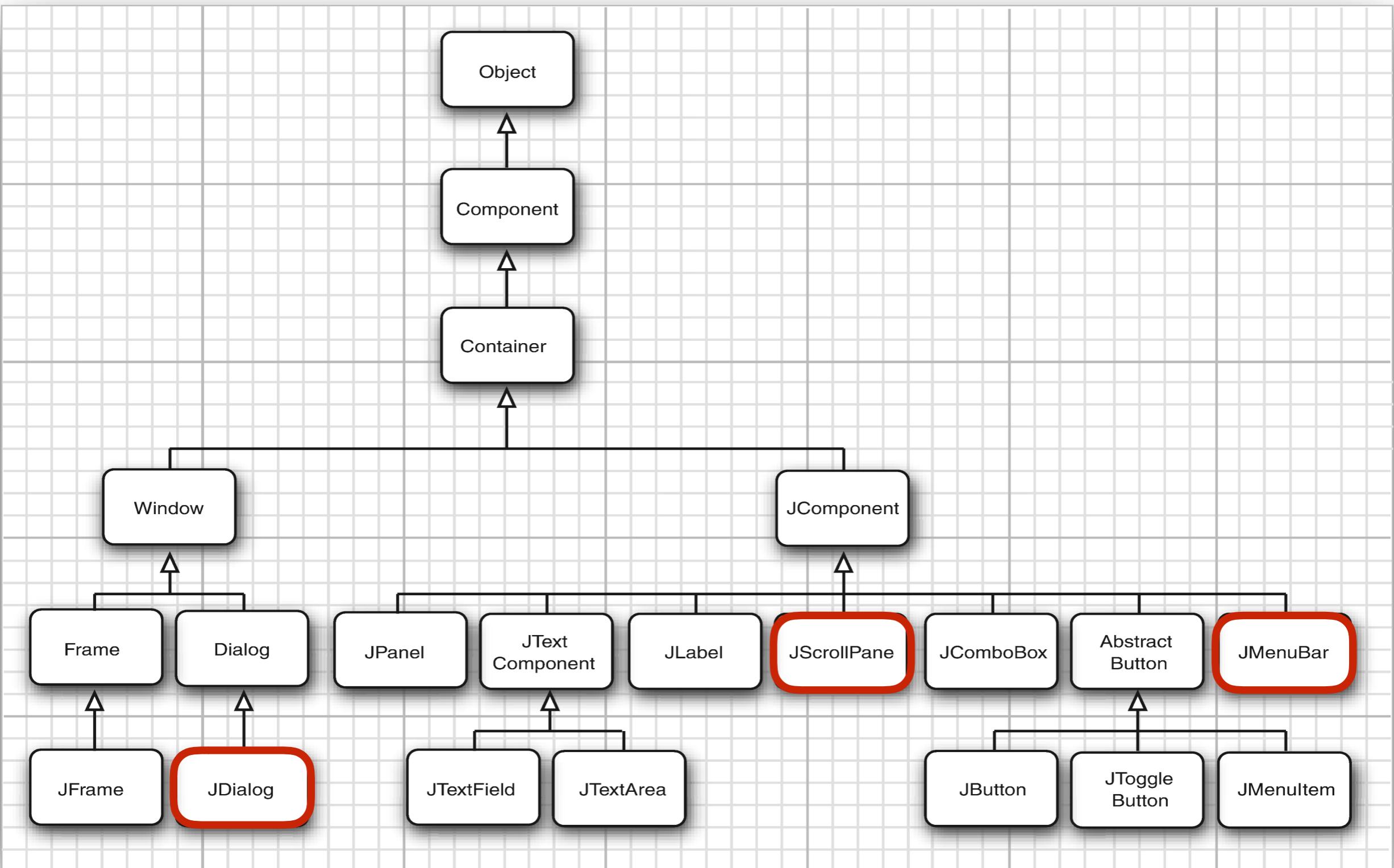
- Les Conteneurs les plus utilisés pour contenir les éléments de l' IG



Conteneurs de plus haut niveau

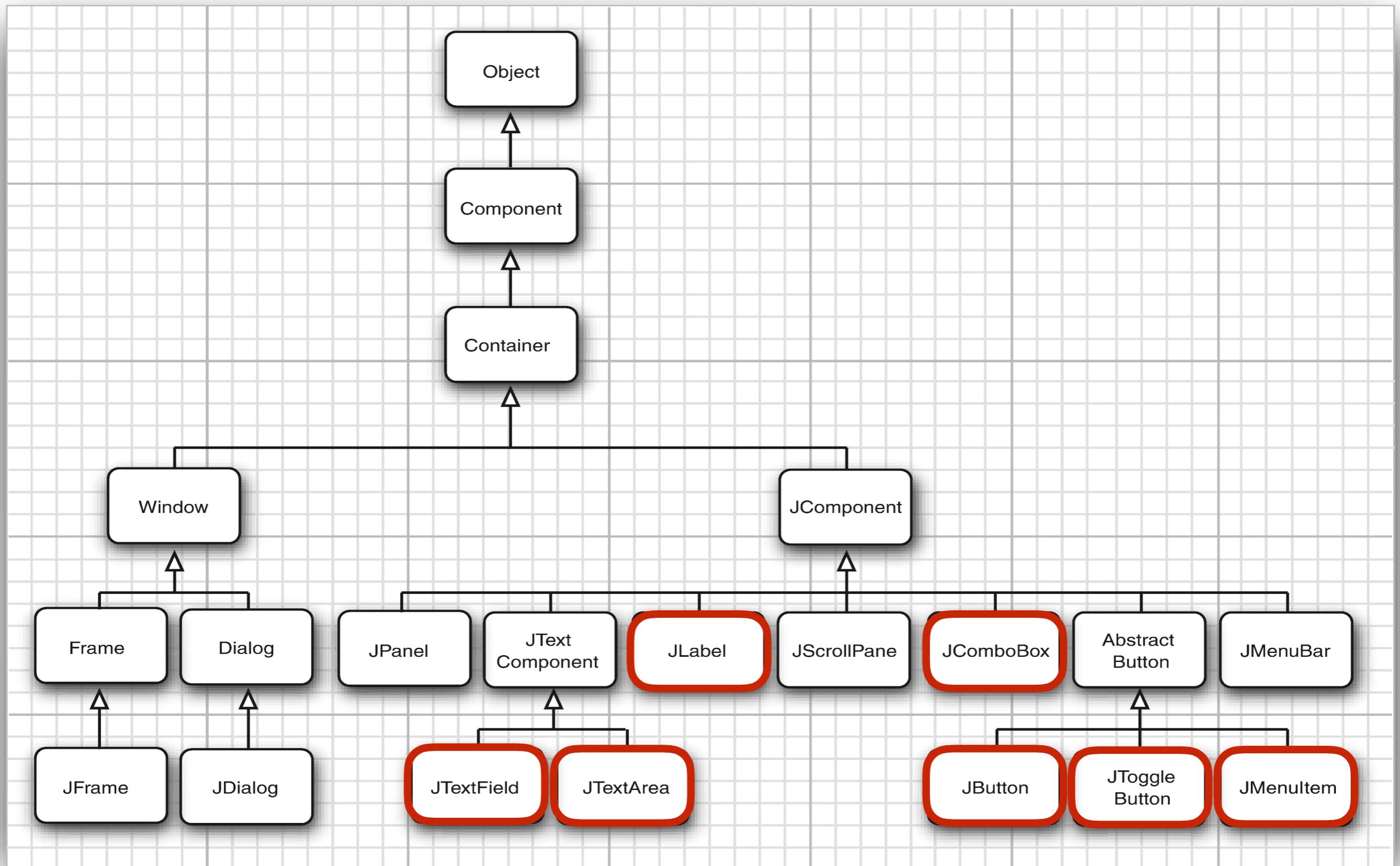
- Les Containers les plus utilisés pour contenir les éléments de l' IG
 - **JFrame** :
 - pour définir une fenêtre : définir une classe qui étend JFrame
 - **JPanel** :
 - pour regrouper plusieurs composants graphiques à positionner dans une fenêtre :
 - créer un JPanel (affiché comme un panneau vide)
 - lui ajouter les composants graphiques
- Typiquement : À des JFrame on ajoutera plusieurs JPanel qui contiendront les composants graphiques (JButton, JTextField, ...)
- Remarque : une JFrame contient automatiquement un "panel" (le "content pane")

Exemples d'autres conteneurs



Les composants graphiques de plus bas niveau

- Le plus souvent ajoutés au containers pour peupler l' IG :



Exemple

```
/*Classe qui définit la fenêtre de plus haut niveau de l' IG*/
class MaFenetre extends JFrame {
    private static final int largeur = 300;
    private static final int hauteur = 200;
    MaFenetre() {
        // on cree 4 boutons
        JButton boutonJ = new JButton("Jaune");
        JButton boutonB = new JButton("Bleu");
        JButton boutonR = new JButton("Rouge");
        JButton boutonV = new JButton("Violet");
        //on cree un panneau pour regrouper les boutons
        JPanel panel = new JPanel();
        // on ajoute les boutons au panneau
        panel.add(boutonJ);
        panel.add(boutonB);
        panel.add(boutonR);
        panel.add(boutonV);
        // on ajoute le panneau à la fenêtre (à son content pane)
        add(panel);
        // continue...
```

Exemple

```
/* suite de MaFenetre() */
    //on fixe la taille de la fenêtre
    setSize(largeur, hauteur);

    //on fixe son titre affiché
    setTitle("Exemple de fenêtre avec boutons");

    //on établit que le programme terminera quand la fenêtre
    //sera fermée
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}//MaFenetre()
}//class MaFenetre
```

Remarque : dans cette fenêtre les boutons ne font rien.

Pour leur donner un comportement il faudrait explicitement leur associer une action (cf. plus loin)

Exemple - remarques

- Fermer une fenêtre la rend non-visible mais ne termine pas le programme
- le programme ne termine même pas quand toutes ses fenêtres sont fermées
- On peut explicitement demander que le programme termine (EXIT) quand la fenêtre est fermée

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Exemple

```
//un main qui cree et affiche une instance de MaFenetre
public class MaFenetreTest {
    public static void main (String args[ ]) {
        //on cree et affiche la fenêtre directement dans le
        //main pour l'instant, une meilleur solution plus loin...
        MaFenetre frame = new MaFenetre();

        // on rend la fenêtre visible
        frame.setVisible(true);
    }
}
```

```
> javac poo/gui/MaFenetreTest.java
> java poo.gui.MaFenetreTest
```

Composants lightweight et heavyweight

- Les composants IG de Java peuvent être
 - **Heavyweight**
 - déléguent leur création, affichage et comportement à l'interface graphique du système d'exploitation (e.g. Windows, Mac OS, ...)
 - **Lightweight**
 - dessinés par Java dans un composant heavyweight - indépendants de la plateforme

Bibliothèques AWT et Swing

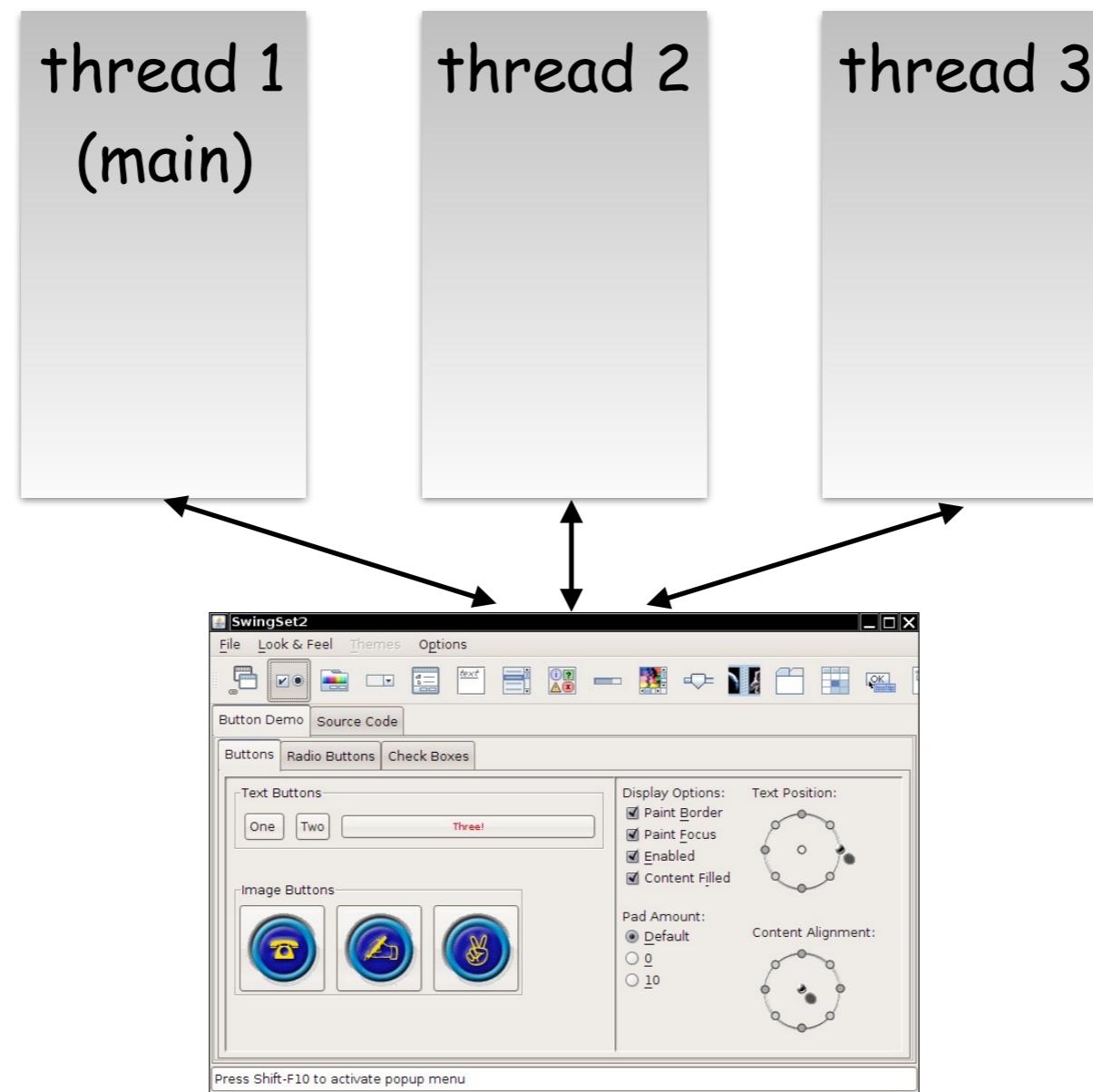
- AWT (Abstract Window Toolkit) prend en charge les composants **heavyweight**
 - Il s'agit des composants en haut de la hiérarchie (Window, Frame, Component, ...)
- Swing prend en charge la gestion des composants **lightweight**
 - Il s'agit des classes dont le nom est typiquement préfixé par "J"
 - Swing dessine le composants dans un canevas AWT
- On utilise en général uniquement des composants Swing (JFrame, JPanel, JButton,...) mais ceux-là sont construits par dessus les composantes AWT (héritage)
- AWT prend également complètement en charge le traitement des événements de l' IG

Gestion de l' IG et threads

- Un programme Java est en général composé par plusieurs processus qui s'exécutent en parallèle (appelé threads)
- Les programmes qu'on a vu jusqu'à maintenant ont un seul thread : celui qui execute le main
 - toutefois le main pourrait créer et lancer d'autres threads
- En revanche l'interface graphique d'un programme est une seule
- => elle est partagée par tous les threads du programme

Gestion de l' IG et threads

- Le partage direct de l' IG pourrait générer des problèmes de coordination
 - plusieurs threads qui essaient de modifier l' IG en même temps...

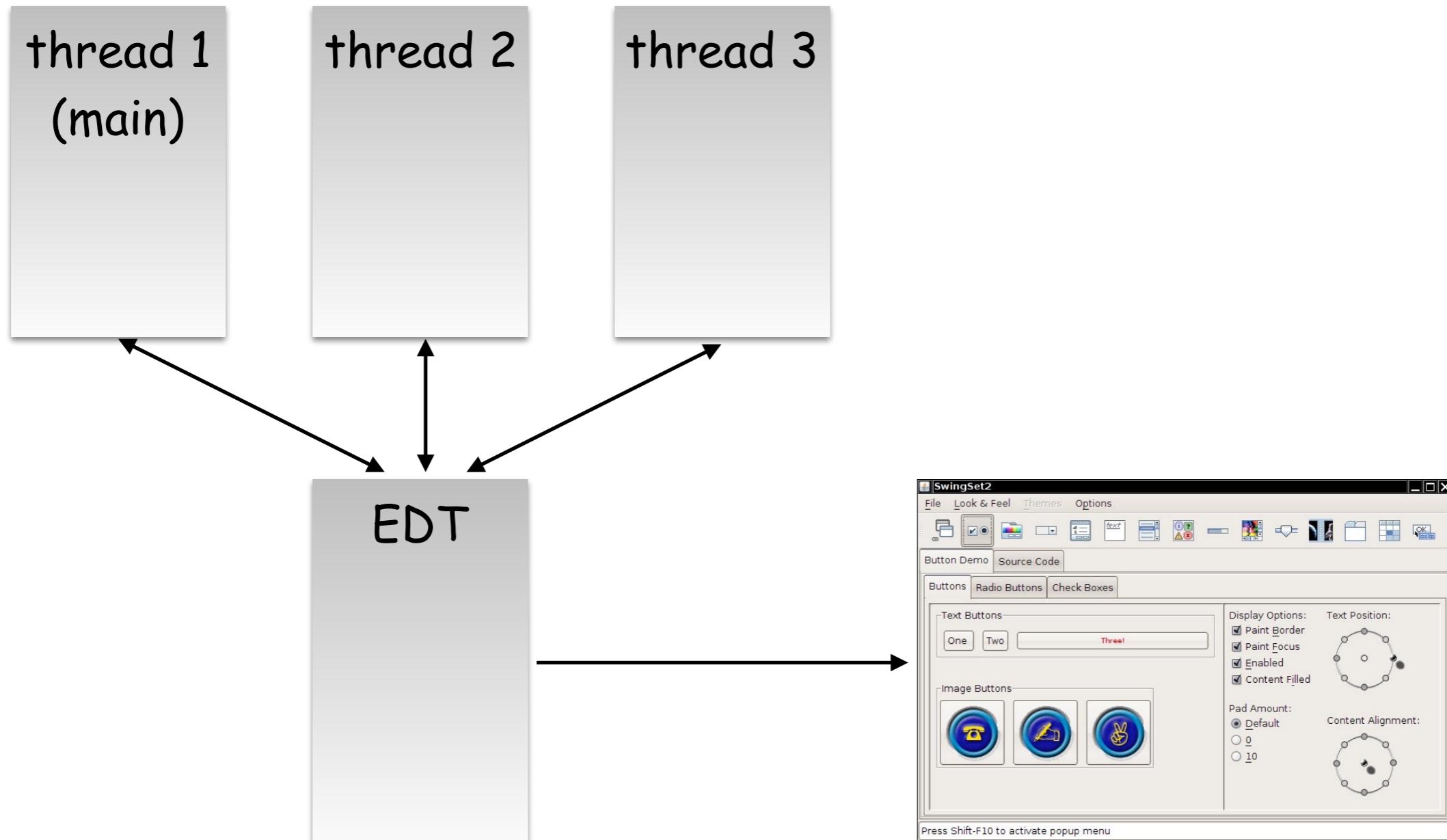


L'Event Dispatcher Thread

- Pour cette raison Swing met en place un thread additionnel dans le programme, l'EDT, uniquement dédié à interagir avec l' IG
 - affichage des composants graphiques, receptions des événements, invocations des actions correspondantes (listeners)

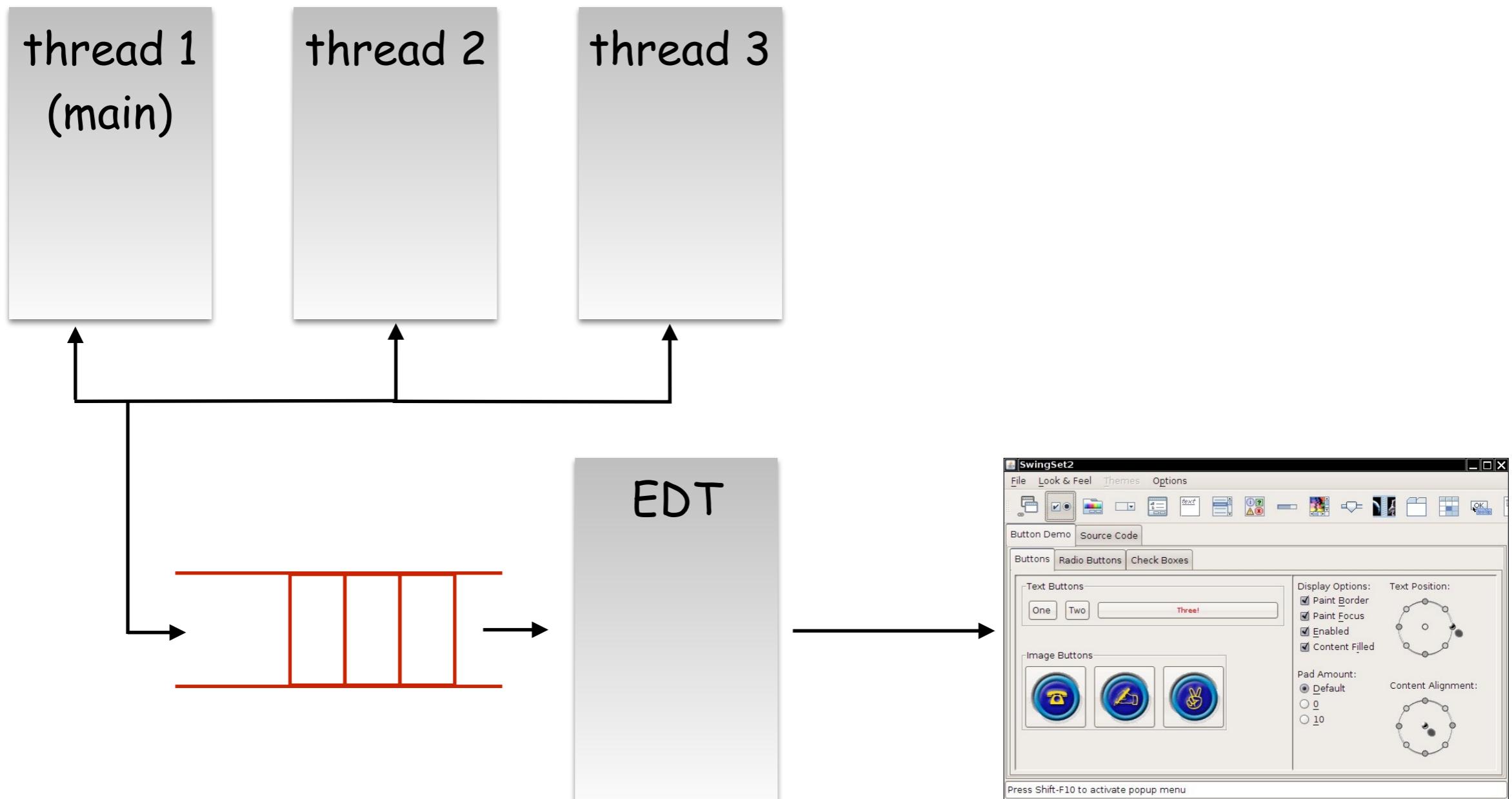
L'Event Dispatcher Thread

- Chaque thread doit s'astreindre à déléguer à l'EDT toute interaction avec l'IG (e.g, création et affichage d'une fenêtre)



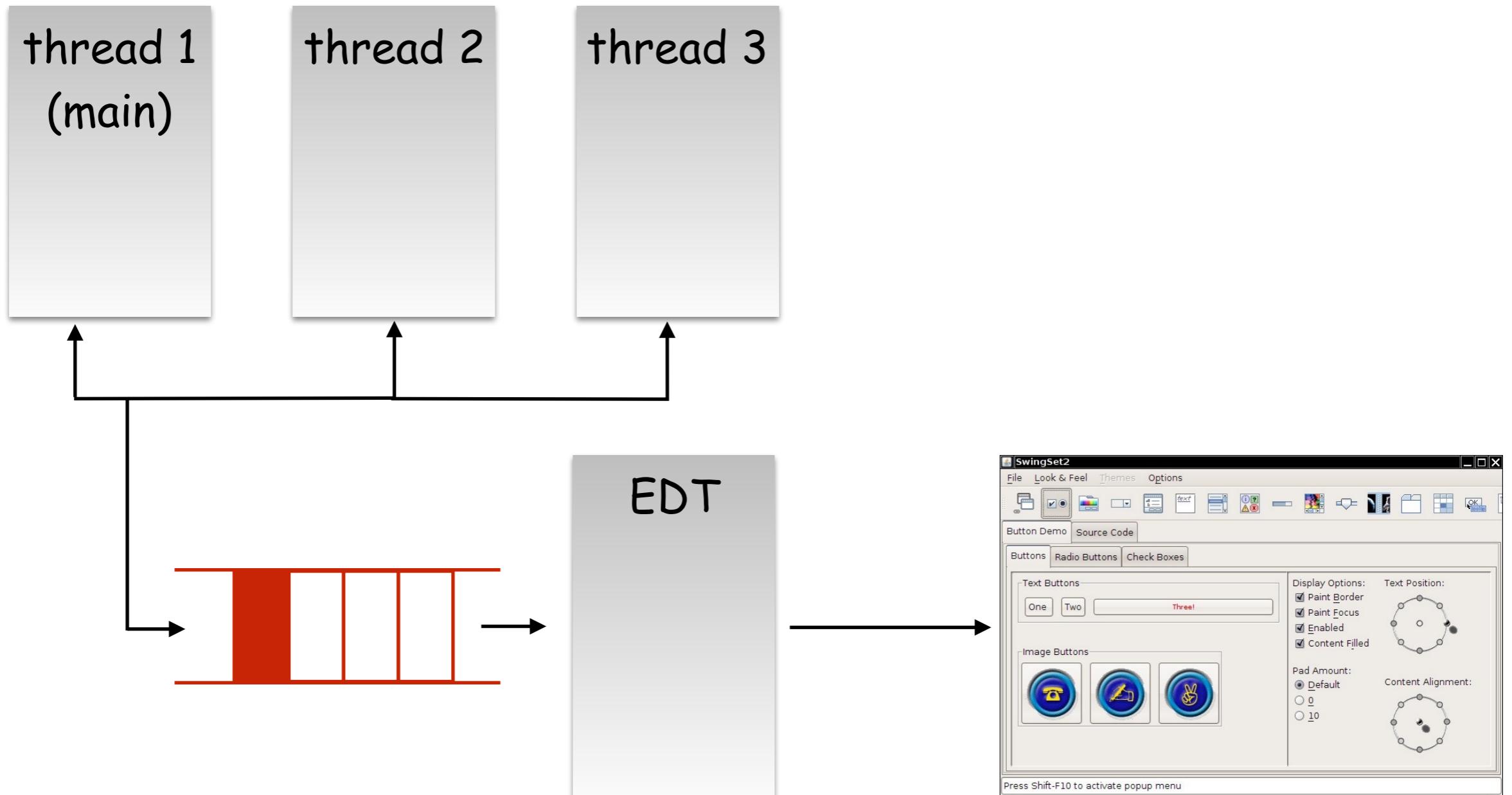
L'EDT et la file d'événements

- La communication entre les threads et l'EDT se fait à travers la file d'événements AWT (Event Queue)



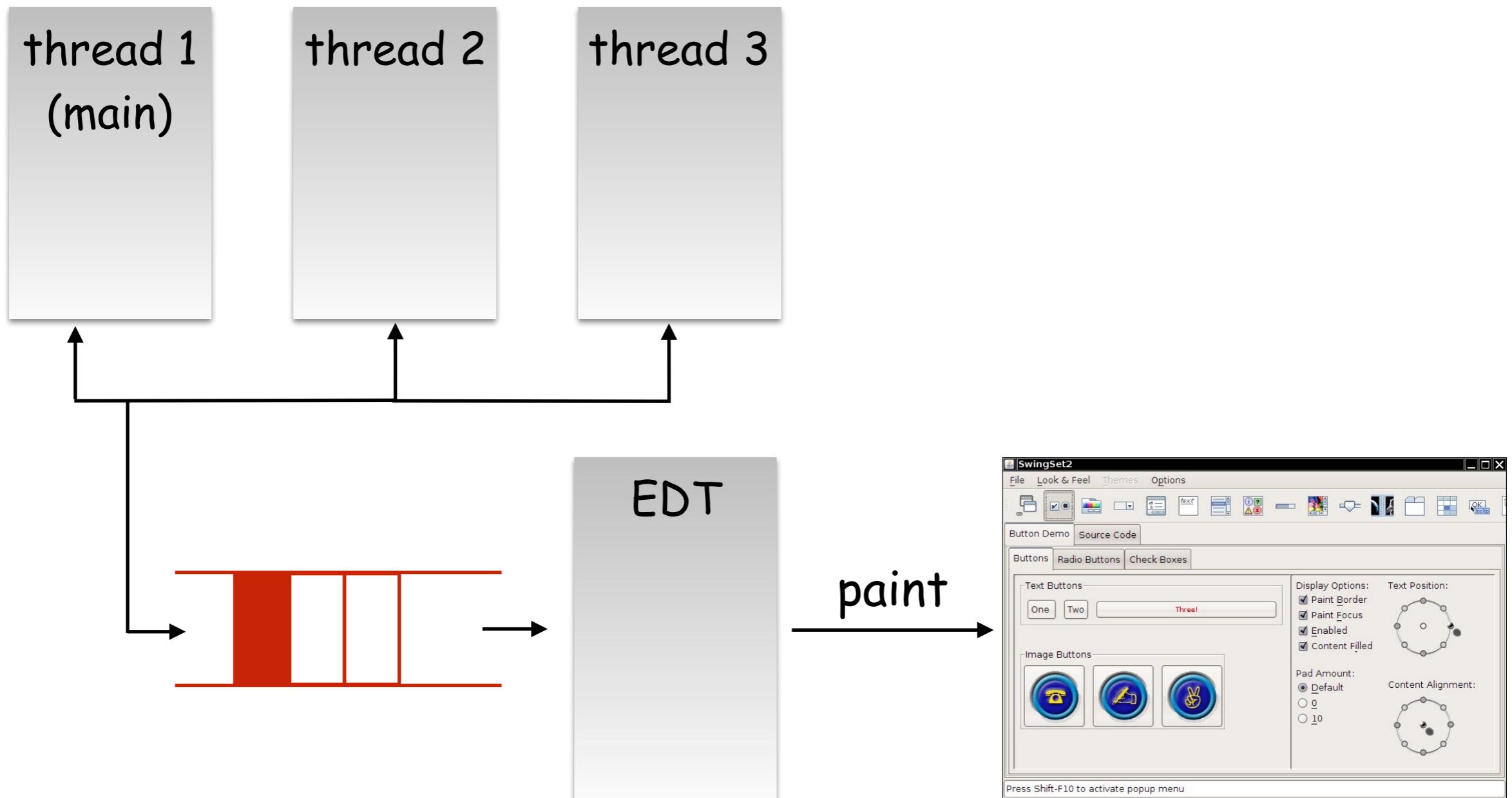
L'EDT et la file d'événements

- Chaque fois qu'un thread doit exécuter une action de manipulation de l'IG, il ajoute un "message" dans la file des événements



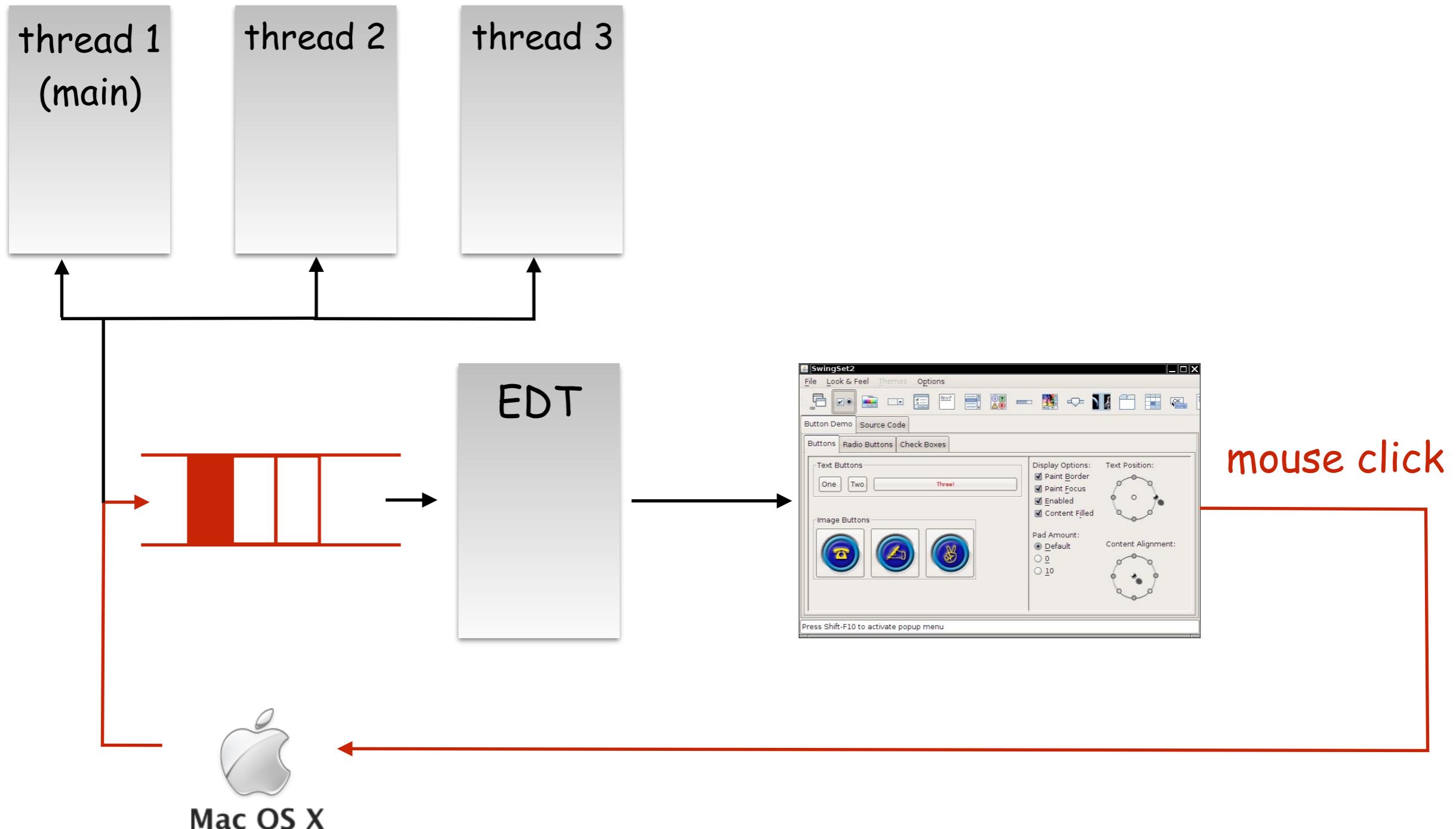
L'EDT et la file d'événements

- L'EDT défile et traite un message à la fois (boucle d'événements)
- execution asynchrone



L'EDT et la file d'événements

- Dans la file des événements également les événements générés par l' IG (voir plus loin)



Communication avec l'EDT

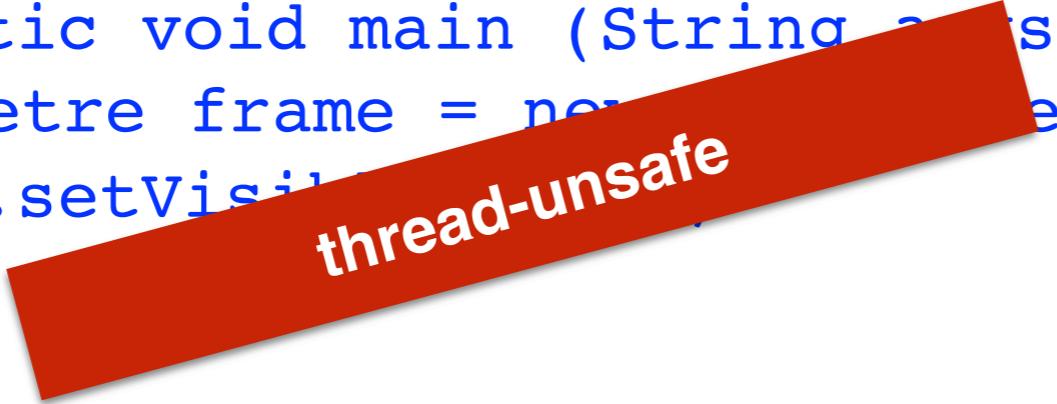
- L'ajout des événements IG par l'OS est automatique
- l'ajout des messages de la part de threads ne l'est malheureusement pas !
- Dans le programme suivant c'est le thread du main qui manipule directement l' IG (création et affichage d'une fenêtre), et non pas l'EDT :

```
public static void main (String args[ ]) {  
    MaFenetre frame = new MaFenetre();  
    frame.setVisible(true);  
}
```

Communication avec l'EDT

- L'ajout des événements IG par l'OS est automatique
- l'ajout des messages de la part de threads ne l'est malheureusement pas !
- Dans le programme suivant c'est le thread du main qui manipule directement l' IG (création et affichage d'une fenêtre), et non pas l'EDT :

```
public static void main (String args[]) {  
    MaFenetre frame = new MaFenetre();  
    frame.setVisible(true);  
}
```



Communication avec l'EDT

- Pour poster une action à executer dans la file des événements, le main doit invoquer la méthode
`EventQueue.invokeLater(Runnable)`
- Runnable est une interface fonctionnelle : sa méthode `run()` implémente l'action à executer par l'EDT quand le message sera traité

```
public static void main (String args[]) {  
    EventQueue.invokeLater( () ->  
    {  
        MaFenetre frame = new MaFenetre();  
        frame.setVisible(true);  
    } );  
}
```

thread-safe

```
javac poo/gui/MaFenetreTest.java  
> java poo.gui.MaFenetreTest
```

Communication avec l'EDT

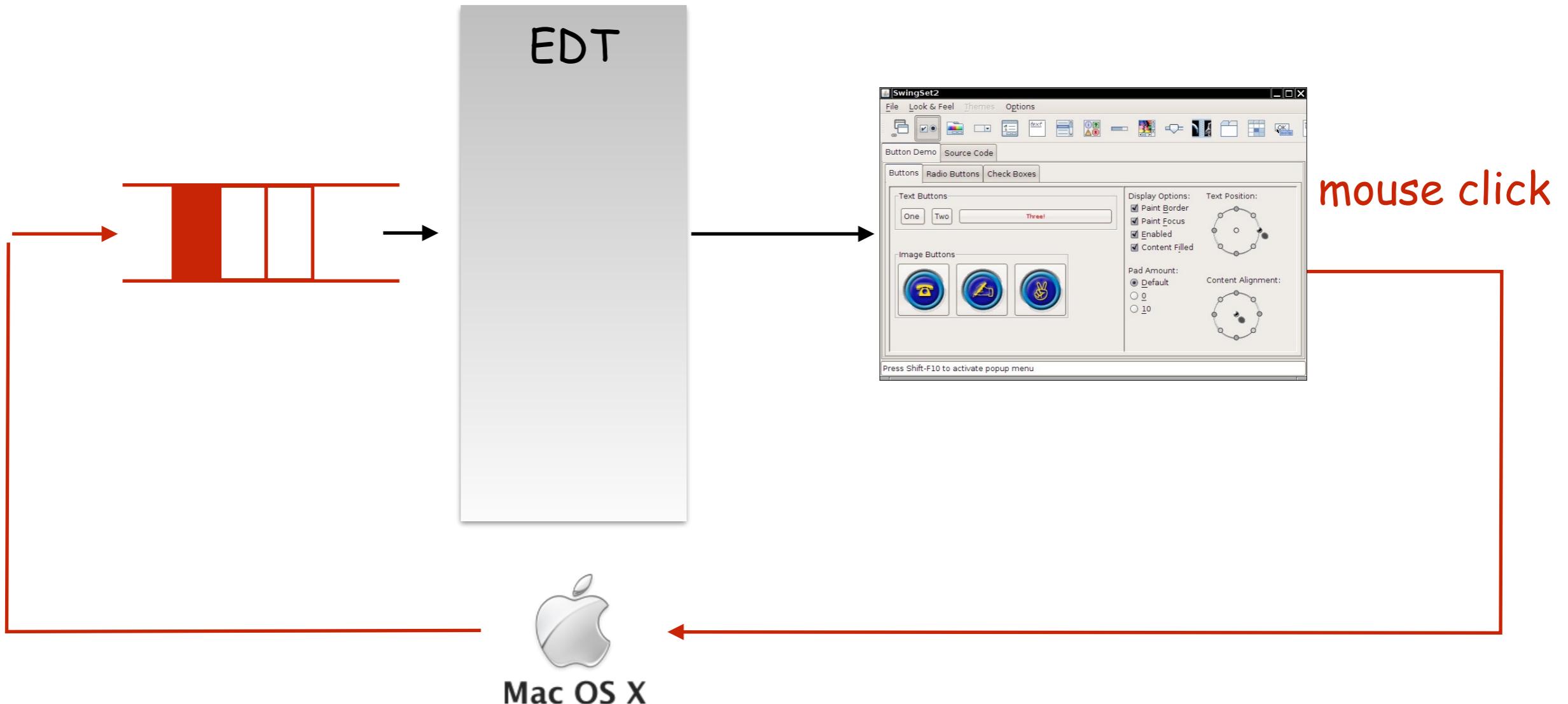
- Avec `EventQueue.invokeLater(...)`
- Le thread du main programme une tâche dans l'EDT et continue son execution (termine peut-être avant que la tâche soit exécutée)
- Pour que en revanche le main attende l'execution de la tâche programmée :

```
try {  
    EventQueue.invokeLater(...);  
} catch (InterruptedException e) {}  
catch (InvocationTargetException e) {}
```

> javac poo/gui/MaFenetreTest1.java
> java poo.gui.MaFenetreTest1

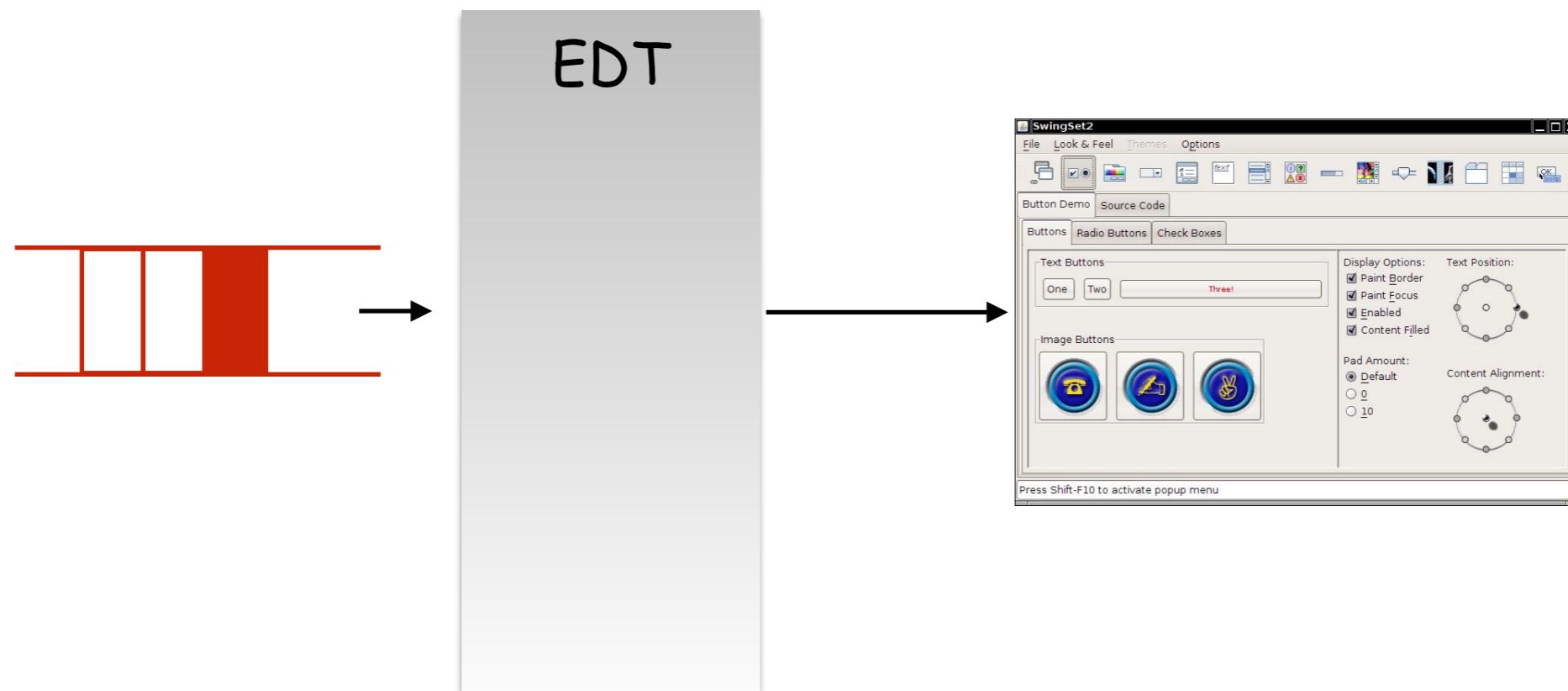
Événements IG: principes

- Chaque composant graphique (e.g un bouton) génère des événements
- Quand l'utilisateur presse le bouton, un événement est posté et va dans la boucle d'événements



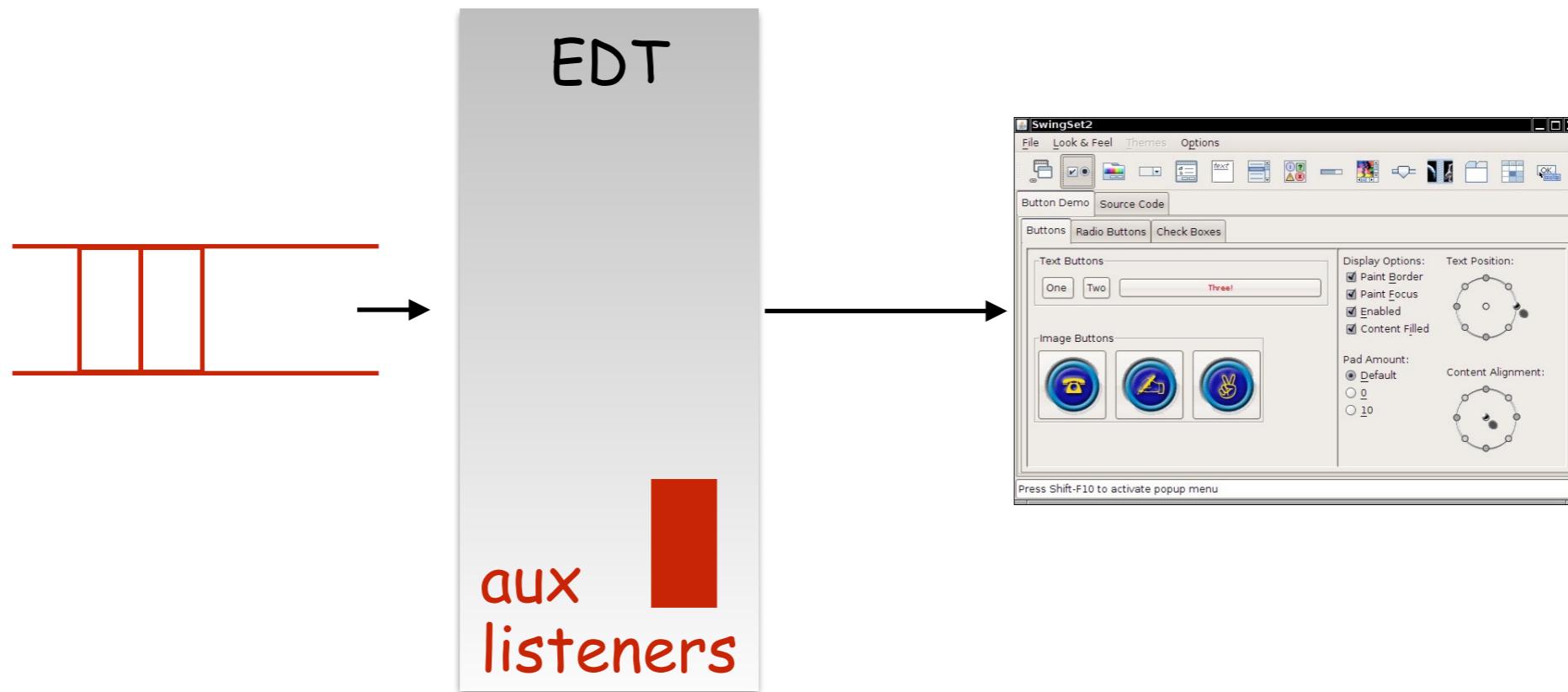
Listeners

- Les événements dans la boucle sont traités un par un (par l'EDT en Java) et transmis aux objets qui se sont enregistrées pour les écouter (**listeners**)



Listeners

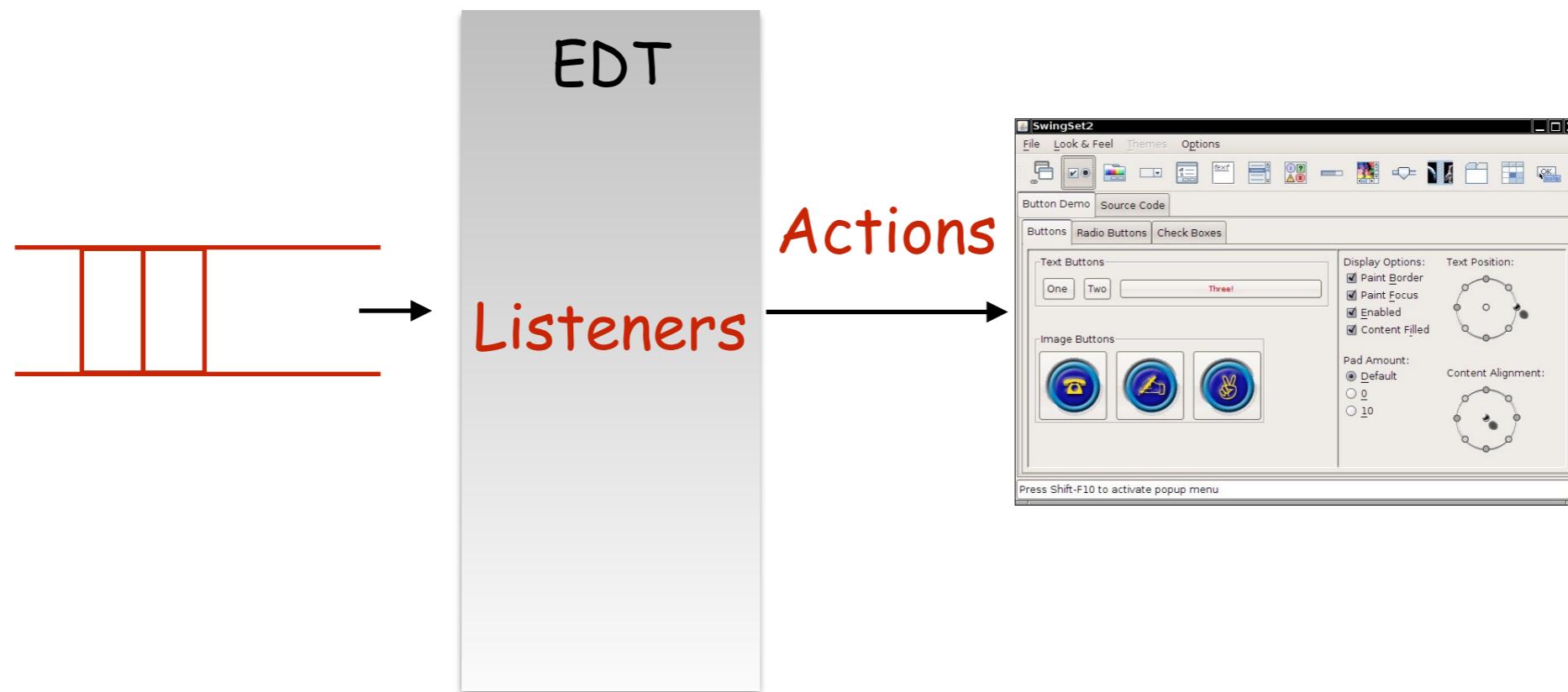
- Les événements dans la boucle sont traités un par un (par l'EDT en Java) et transmis aux objets qui se sont enregistrées pour les écouter (**listeners**)



- L'événement fait suivre au listener des informations sur la situation qui l'a généré (e.g., quel bouton, position de la souris, modificateurs...)

Listeners et Actions

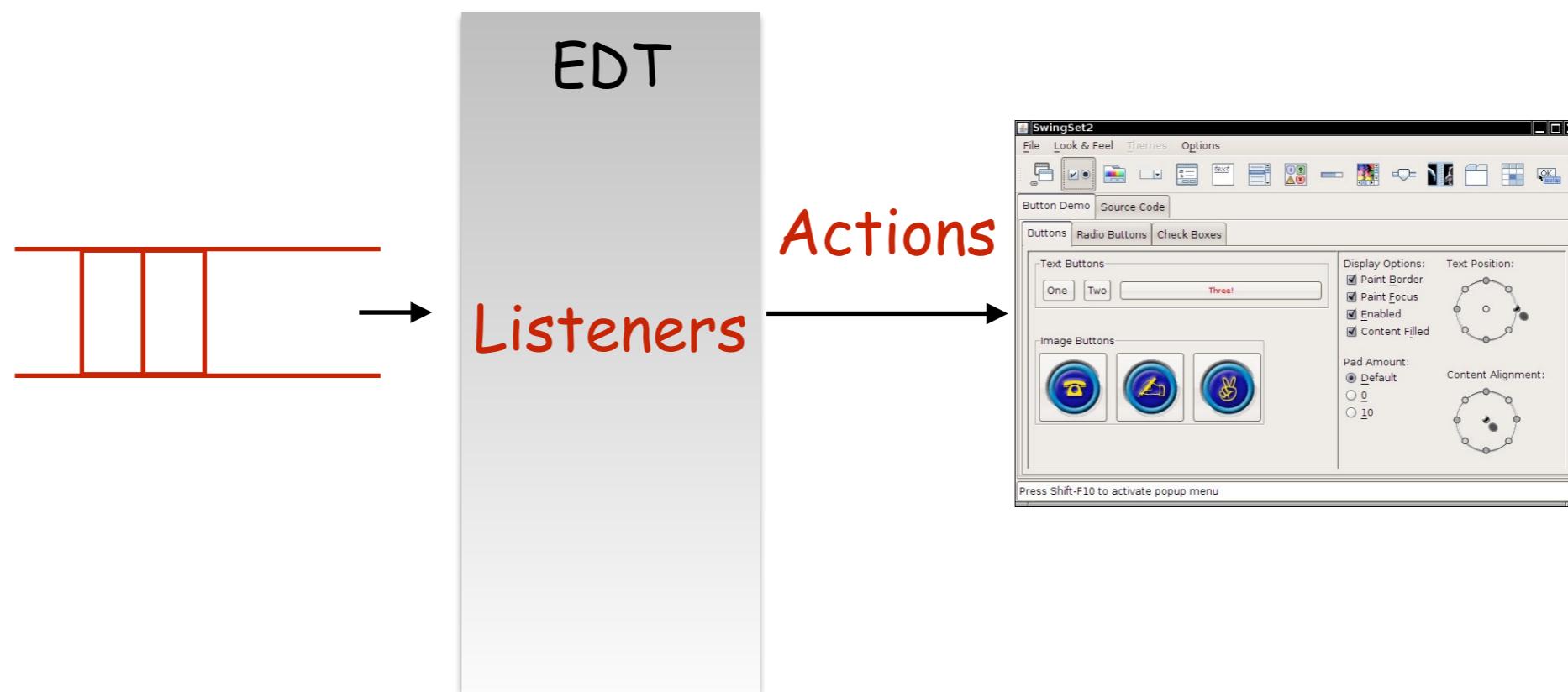
- Sollicités par l'événement reçu, les listeners s'exécutent dans le thread EDT



- Un listener exécute une Action (code exécuté en réponse à l'événement)
 - e.g. changer la couleur de fond en réponse au clic d'un bouton

Listeners et Actions

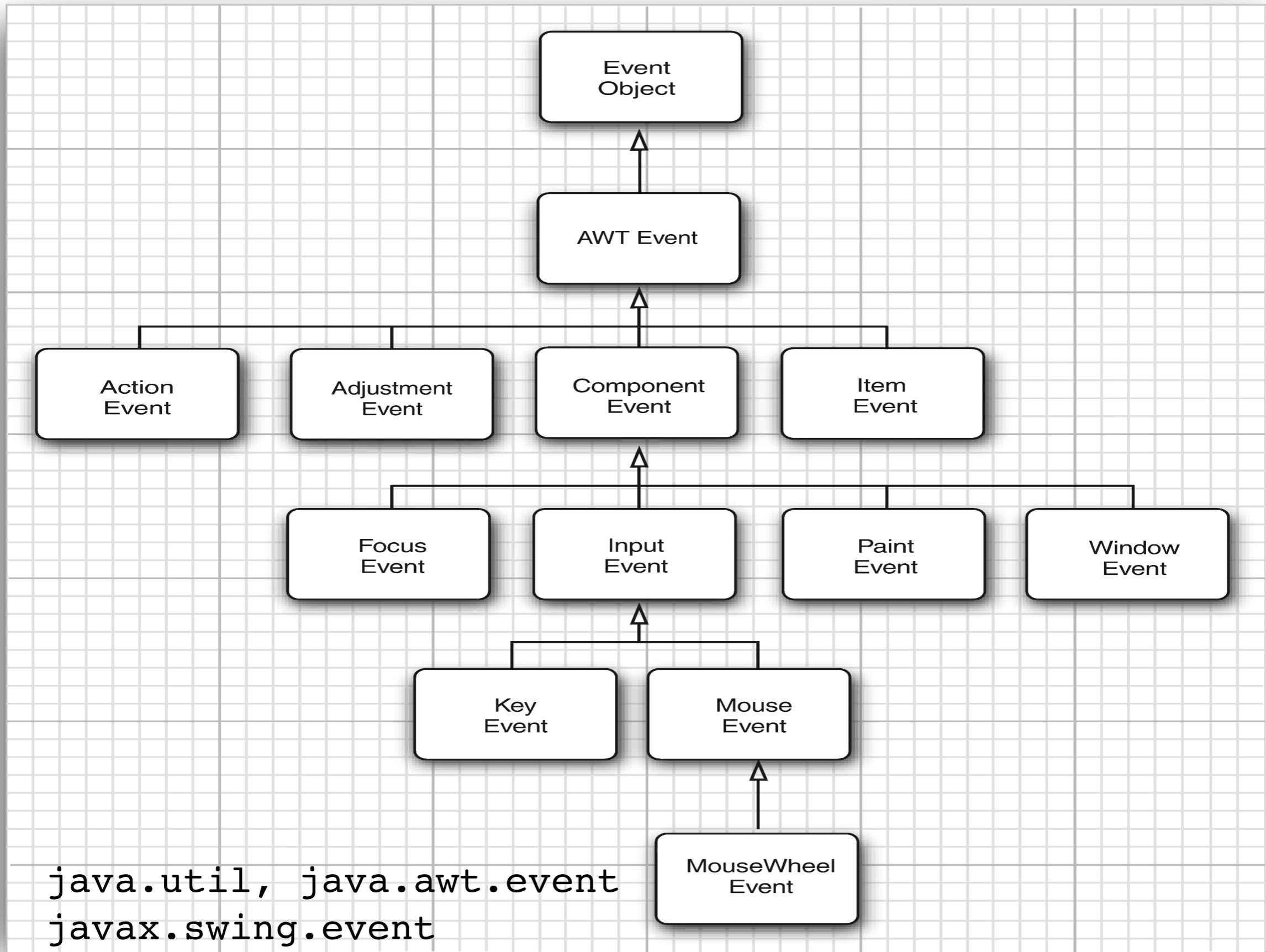
- Puisque les listeners s'exécutent dans le thread EDT, il est safe pour un listener de manipuler directement l'interface graphique



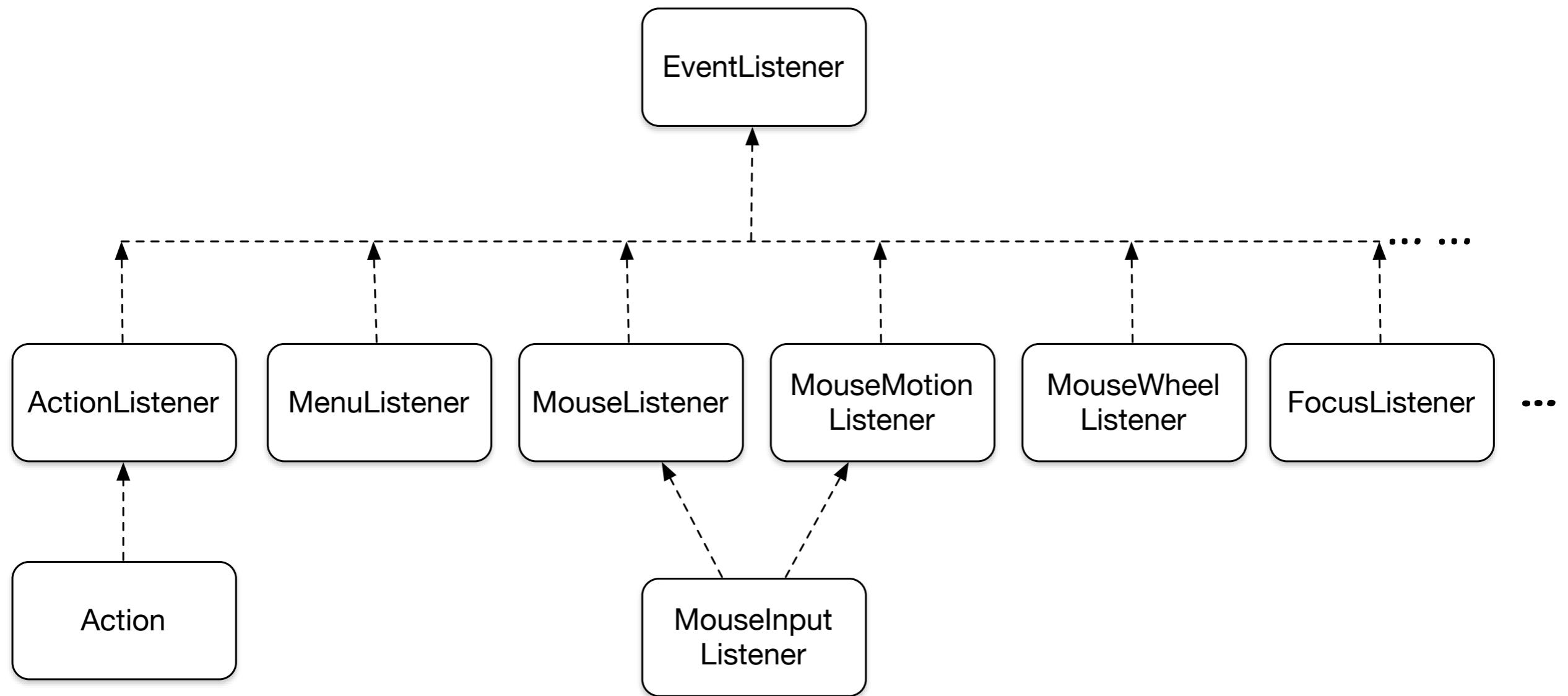
Gestion des événements en AWT

- En Java composants, événements et listeners correspondent à autant de hierarchies de classes / interfaces
 - Un **composant graphique** est un objet d'une sous-classe de [Component](#)
 - Un **événement** est un objet dont la classe hérite de [AWTEvent](#)
 - Un **event listener** est un objet dont la classe implémente une interface qui hérite de [EventListener](#)
 - les méthodes abstraites de ces interfaces sont utilisés pour implementer les actions du listener en réponse à l'événement déclencheur

Hiérarchie des classes événement AWT



Un aperçu de la hiérarchie des interfaces listeners



(`java.util, java.awt.event`)

Exemples d'interfaces listeners

```
public interface ActionListener extends EventListener{  
    void actionPerformed (ActionEvent e);  
}
```

```
public interface MouseListener extends EventListener {  
    void mouseClicked(MouseEvent e);  
    void mouseEntered(MouseEvent e);  
    void mouseExited(MouseEvent e);  
    void mousePressed(MouseEvent e);  
    void mouseReleased(MouseEvent e);  
}
```

Programmation event-driven d' IG avec AWT

Génération des événements par les composants graphiques

- Chaque composant graphique peut générer des événements
 - E.g. un `JButton`, quand il est cliqué, génère un `ActionEvent` (transmis par l'OS dans la boucle d'événements)
 - ainsi qu'un `JTextfield`, quand du texte est rentré dans le champs
 - un `Component`, quand la souris entre/sort/est pressée/relâchée sur ce composant, genre un `MouseEvent`
 - etc ...

Programmation event-driven d' IG avec AWT

Génération des événements par les composants graphiques

- Exemple

```
class FenetreCompteur extends JFrame{  
    private static final int largeur = 300;  
    private static final int hauteur = 200;  
    private JLabel message = new JLabel(" Rien reçu ");  
    private int cmp = 0;  
    FenetreCompteur () {  
        JButton bouton = new JButton("Clicker ici");  
        // va générer un ActionEvent à chaque click  
        // on va l'utiliser pour afficher le nombre de clicks  
        ...  
        add(bouton, BorderLayout.NORTH);  
        add(message);  
        setSize(largeur, hauteur);  
        setTitle("Bouton qui compte");  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
}
```

Programmation event-driven d' IG avec AWT

Définition des listeners

- Chaque objet peut être un event listener, pour cela sa classe doit **implémenter l'interface correspondante aux événements qu'il veut écouter**, et en implémenter les méthodes de réponse

```
class Compteur implements ActionListener {  
    public void actionPerformed (ActionEvent event) {  
        //modifier le texte de message  
    }  
}
```

Programmation event-driven d' IG avec AWT

Définition des listeners (suite)

- De plus il doit s'enregistrer auprès du composant qui génère les événements qu'il veut écouter

```
FenetreCompteur () {  
    JButton bouton = new JButton("Clicker ici");  
    bouton.addActionListener( new Compteur() );  
    add(bouton, BorderLayout.NORTH);  
    add(message);  
    setSize(largeur, hauteur);setTitle("Bouton qui  
        compte");  
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
}
```

Programmation event-driven d' IG avec AWT

Définition des listeners (suite)

- Une classe listener peut la plupart du temps être interne à la classe qui en crée des instances
 - e.g. Compteur interne à FenetreCompteur
- Elle peut même être locale au bloc de code qui crée les listeners (et possiblement anonyme)

```
FenetreCompteur () {  
    JButton bouton = new JButton("Clicker ici");  
    bouton.addActionListener( new ActionListener(){...});  
    add(bouton, BorderLayout.NORTH);  
    add(message);  
    setSize(largeur, hauteur);setTitle("Bouton qui  
        compte");  
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
}
```

- De fait elle n'est utilisée que dans ce bloc de code

Programmation event-driven d' IG avec AWT

Définition des listeners (suite)

- Cela a un autre avantage : le listener a accès aux autres champs de la classe englobante, qu'il peut manipuler directement

```
class FenetreCompteur extends JFrame{  
    private int cmp = 0; private JLabel message; ...  
    FenetreCompteur () {  
        JButton bouton = new JButton("Clicker ici");  
        bouton.addActionListener( new ActionListener() {  
            public void actionPerformed (ActionEvent e) {  
                message.setText("clické "+ (++cmp)+ " fois");  
            }  
        });  
        add(bouton, BorderLayout.NORTH);  
        add(message);  
        setSize(largeur, hauteur);setTitle("Bouton qui  
        compte");  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
}
```

Programmation event-driven d' IG avec AWT

Définition des listeners (suite)

- Depuis Java 8 ceci peut s'écrire de façon beaucoup plus concise et intuitive, avec les expressions Lambda !

```
class FenetreCompteur extends JFrame{  
    private int cmp = 0; private JLabel message; ...  
    FenetreCompteur () {  
        JButton bouton = new JButton("Clicker ici");  
        bouton.addActionListener ( event ->  
            {  
                message.setText("clické "+ (++cmp)+ " fois");  
            }  
        );  
        add(bouton, BorderLayout.NORTH);  
        add(message);  
        setSize(largeur, hauteur);setTitle("Bouton qui compte");  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
}
```

>! poo/gui/FenetreCompteurTest.java

Programmation event-driven d' IG avec AWT

Attention :

- Les listeners seront définis comme classes locales non-anonymes (ou comme classes externes) si en revanche plusieurs instances du même listener sont utilisées par l'IG
- Un exemple plus loin

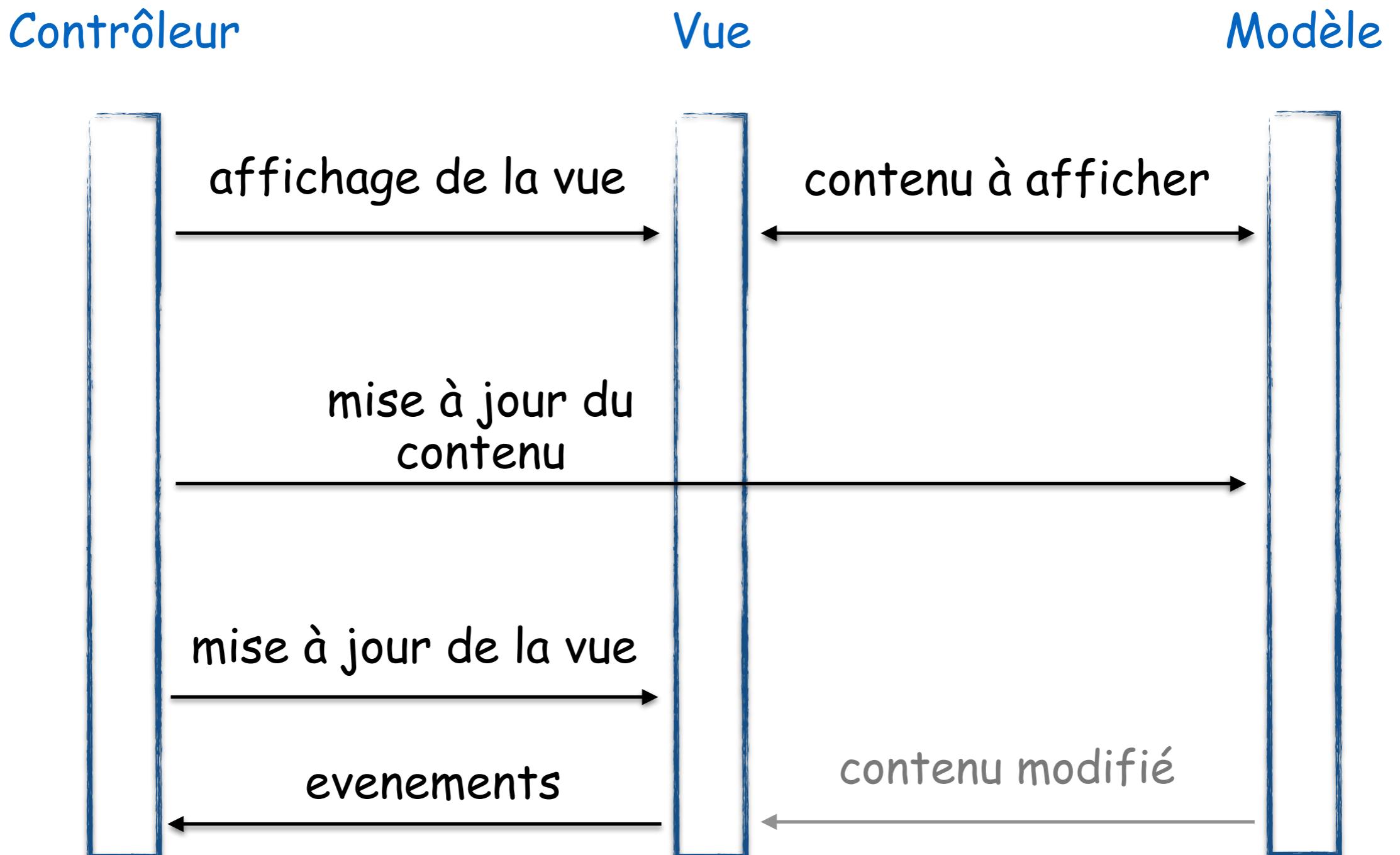
Design patterns et Model-View-Controller (MVC)

- Le système d'interface graphique Swing a été développé en suivant un "design pattern" connu sous le nom de Model-View-Controller
 - **Design pattern** (paradigme de conception) : un ensemble de règles de conception du code
 - donnent une solution propre à de problèmes récurrents de conception du code

Model-View-Controller (MVC)

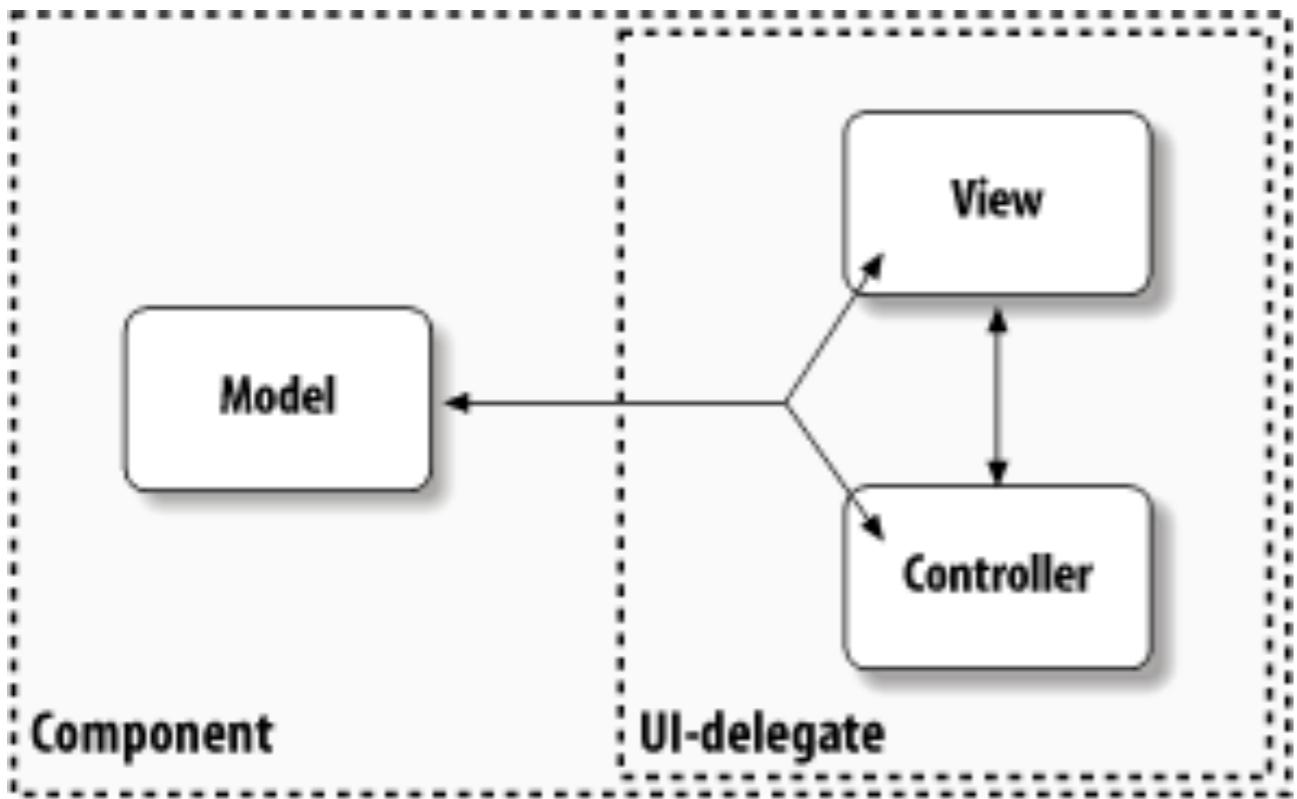
- MVC : un design pattern très en vogue, particulièrement propre
 - Consiste en **distinguer** et faire **interagir** trois parties principales du code :
 - Le **modèle** : le contenu (les données)
 - La **vue** : l'affichage
 - Le **contrôleur** : le comportement
 - Les trois parties doivent être bien **distinguées**
 - e.g. les données indépendantes de l'affichage
 - le comportement distinct des données et de l'affichage
- Mais elles ont des **interactions fortes**
 - le contrôleur reçoit des input d'utilisateur à travers la vue
 - détermine le comportement à suivre
 - modifie les données et/ou la vue

Interactions MVC



Swing et MVC

- Chaque composant Swing est conçu selon MVC



E.g Un `Jtextfield`

- modèle (contenu) : le texte
- vue :
- contrôleur : coordonne modèle vue et utilisateur, e.g
 - utilisateur écrit un caractère
 - contrôleur notifié, demande au modèle et à la vue de se mettre à jour



- Dans Swing classes différentes s'occupent des différentes parties, e.g.
 - classe ButtonModel (pour le contenu d'un bouton)
 - classe ComponentUI classe déléguée UI de tout JComponent (vue et contrôleur)
 - ...

Interfaces graphiques et MVC

- Le paradigme MVC est utile également pour concevoir la globalité de l'interface graphique
- Exemple : jeu d'échecs utilisateur - ordinateur, en gros
 - **Vue** : affichage de l'échiquier et des pions
 - **Modèle** : le tableau qui stocke les positions des pions dans l'échiquier (et pas seulement...)
 - **Contrôleur** : la logique du jeu, e.g.
 - notifié par la vue, lit le coup de l'utilisateur, calcule la réponse, modifie les données de façon correspondante, met à jour la vue (l'échiquier) pour montrer son déplacement
 - etc...
- Le code des trois parties doit être le plus possible séparé (différentes classes qui interagissent)

Look and feel

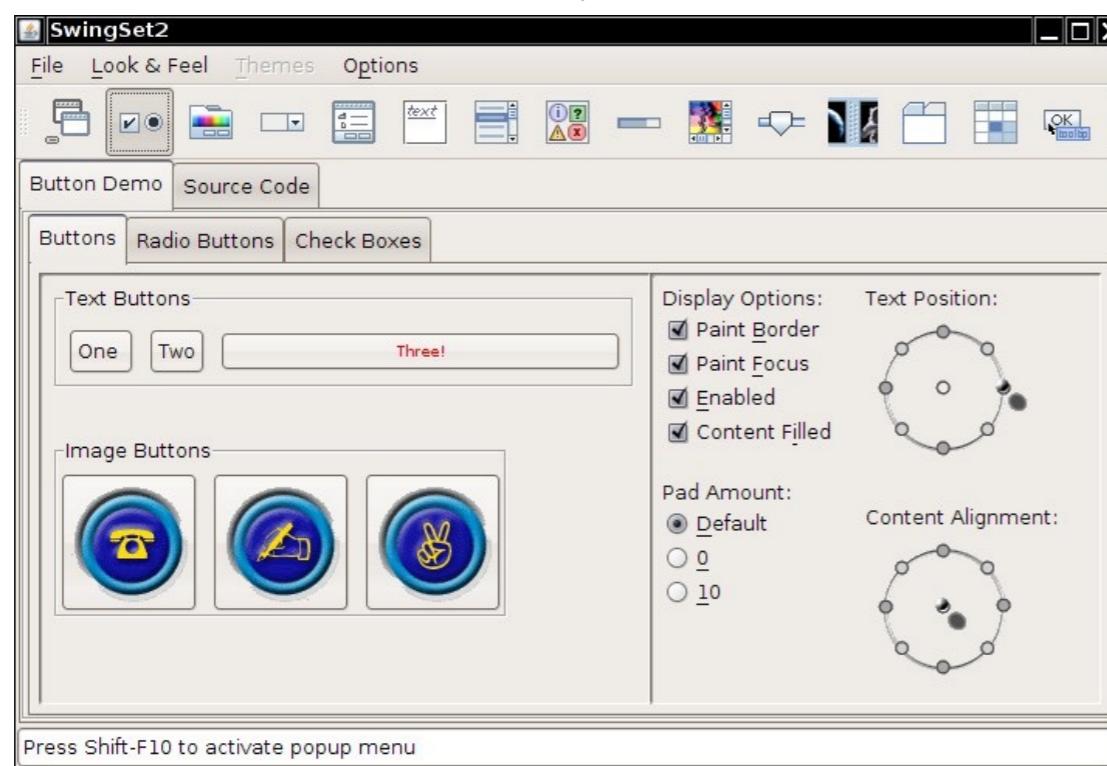
- La séparation de la vue du reste (modèle + contrôleur), permet de gérer facilement **de multiples affichages du même contenu**
- **Look and feel**: possibilité de choisir l'apparence (style) de l'interface graphique

Quelques look&feel

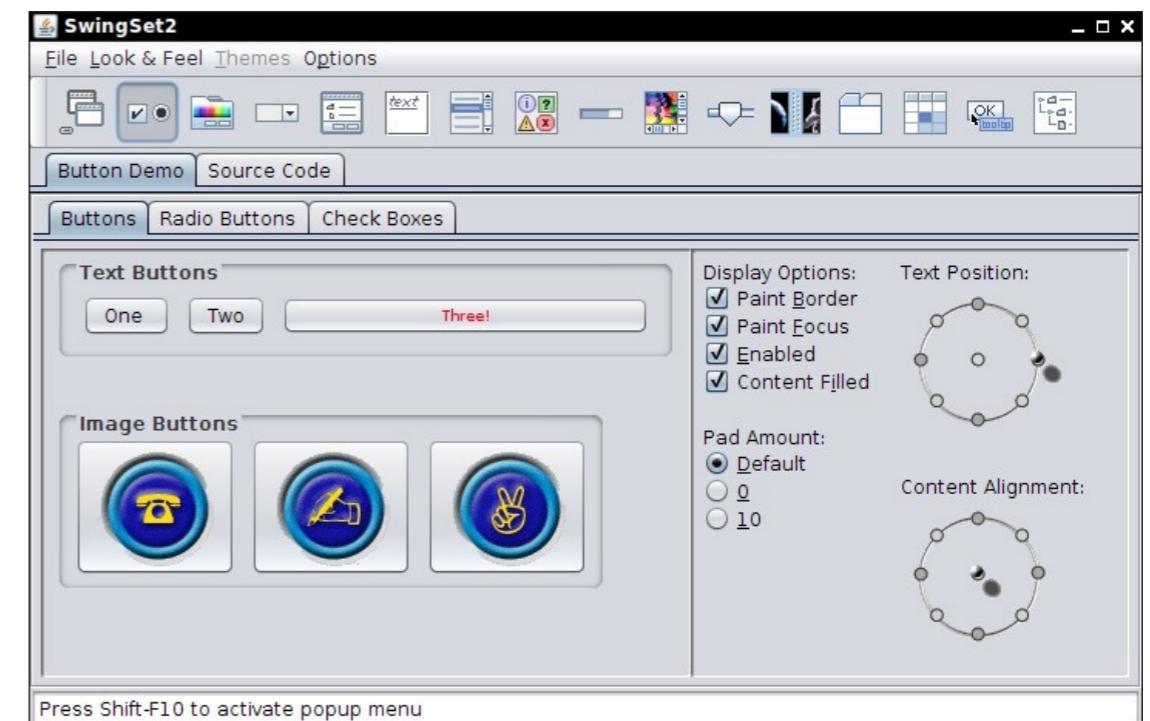
Metal



GTK



Nimbus



Look and feel

- Les composants Swing ont un look&feel par default, mais on peut le changer dynamiquement
 - UIManager gère l'apparence de l'interface
 - (on peut même créer son propre look&feel !)

```
public static void main (String args[ ]) {  
    EventQueue.invokeLater(() -> {  
        //on fixe le look&feel  
        String laf =  
            UIManager.getCrossPlatformLookAndFeelClassName();  
        try { UIManager.setLookAndFeel(laf); }  
        catch (Exception e) { e.printStackTrace(); }  
        FenetreLaf frame = new FenetreLaf();  
        frame.setVisible(true);  
    });  
}
```

code/poo/gui/Laf.java
> !

Interfaces graphiques et tâches longues

- Les listeners d'événements sont exécutés dans l'EDT
 - => ils ne doivent executer que des tâches "rapides"
 - en effet puisque toutes les tâches graphiques s'exécutent dans le même thread (EDT), le reste de l'interface graphique est en attente pendant l'exécution de chaque tâche
- Une tâche longue ne doit pas être exécutée dans l'EDT

```
//listener  
event ->  
{  
    // operation longue  
}
```

=> "freeze" de l'interface graphique !

code/poo/gui/Freeze.java
>!

Interfaces graphiques et "worker threads"

- Solution : lancer un nouveau thread ("worker thread") pour l'exécution de la tâche longue
- Il s'exécutera comme tâche de fond en même temps que l'EDT
 - => interface graphique active pendant la tâche de fond !
- Une fois terminé, le worker thread pourra ajouter une tâche à exécuter dans l'EDT (i.e. afficher les résultats calculés)
 - (des tâches peuvent être ajoutées à l'EDT également pendant l'exécution de la tâche de fond...)

Interfaces graphiques et “worker threads”

- Worker thread : extension (typiquement locale) de la classe SwingWorker

```
class TacheLongue extends SwingWorker <T, V> {  
    @Override  
    public T doInBackground () {  
        //tâche longue  
    }  
    @Override  
    public void done () {  
        //tâche à executer dans l'EDT quand doInBackground termine  
    }  
}
```

- Listener : utilise execute() pour lancer un nouveau worker thread

```
//listener  
event -> new TacheLongue().execute();
```

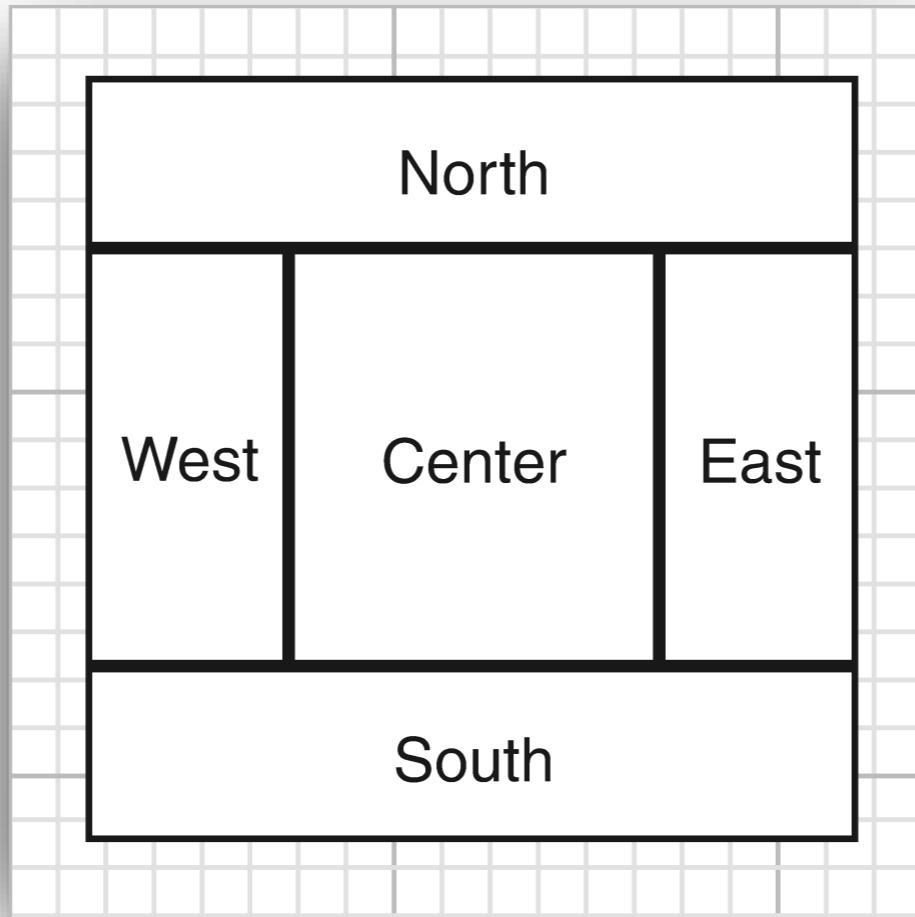
code/poo/gui/Freeze.java
>!

Layout Manager

- Java permet d'associer à chaque container graphique un "layout manager"
 - spécifie comment vont se positionner le composants que l'on ajoute dans le container
 - chaque type de container a son layout manager par default, e.g.
 - JPanel : Flow layout manager
 - JFrame : Border Layout manager
 - ...
 - mais cela peut être changé :

```
JPanel p = new JPanel();
p.setLayout(new BorderLayout());
```

Border layout



- Chaque zone occupe tout l'espace si les autres zones sont vides
- `add(component)` ajoute par default à la zone CENTER (les composants se superposent)
- `add(component, BorderLayout.EAST)` pour ajouter à Est

Border layout

- Les composants sont maximisés en taille pour occuper toute la zone dans laquelle ils sont placés



Un bouton ajouté à la fenêtre dans BorderLayout.SOUTH

Flow layout

- Les composants gardent leur taille et sont placés l'un à côté de l'autre
- va à la ligne quand il n'y a plus de place sur la même ligne



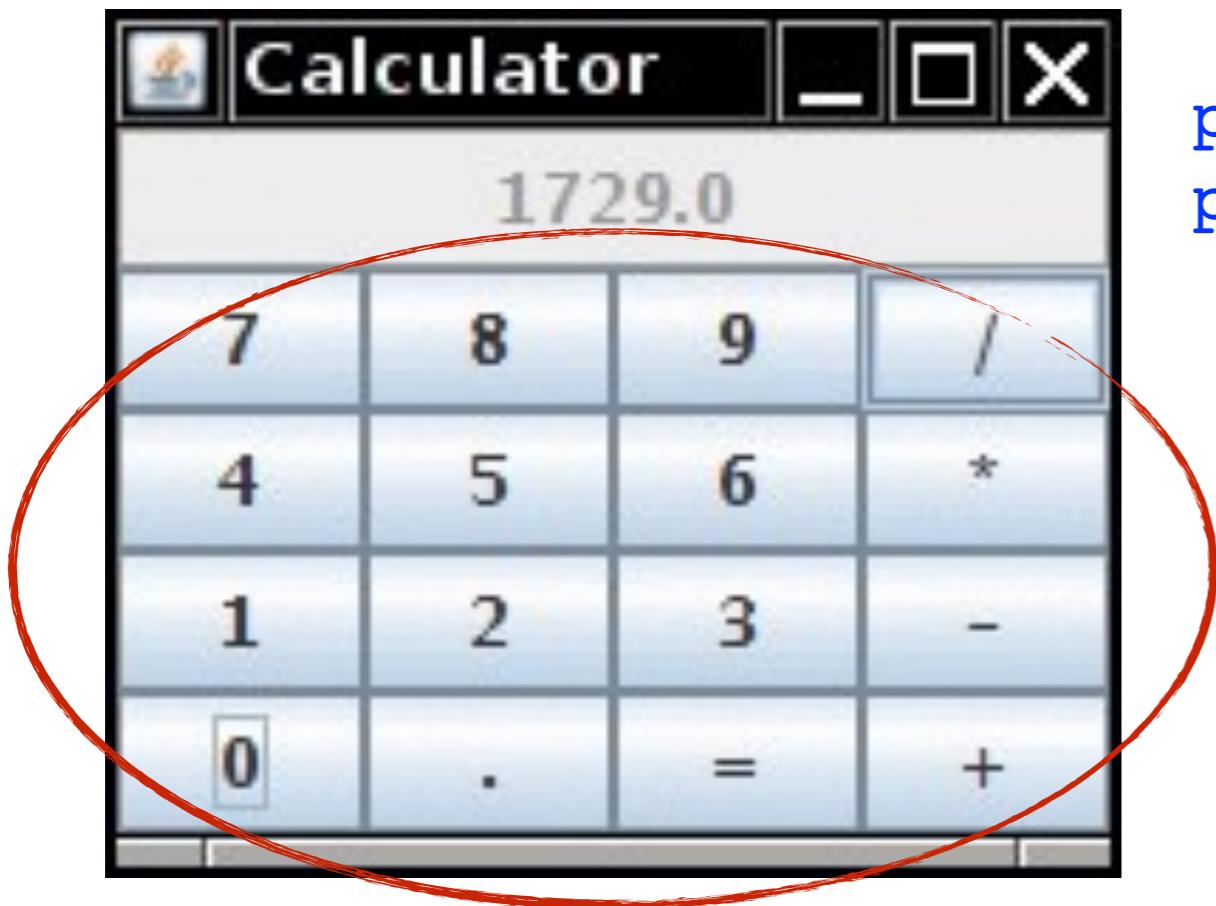
Flow layout

- Le contenu se reorganise quand la taille du container change (e.g. en redimensionnant la fenêtre)



Grid Layout

- Les composants sont disposés dans une grille fixée

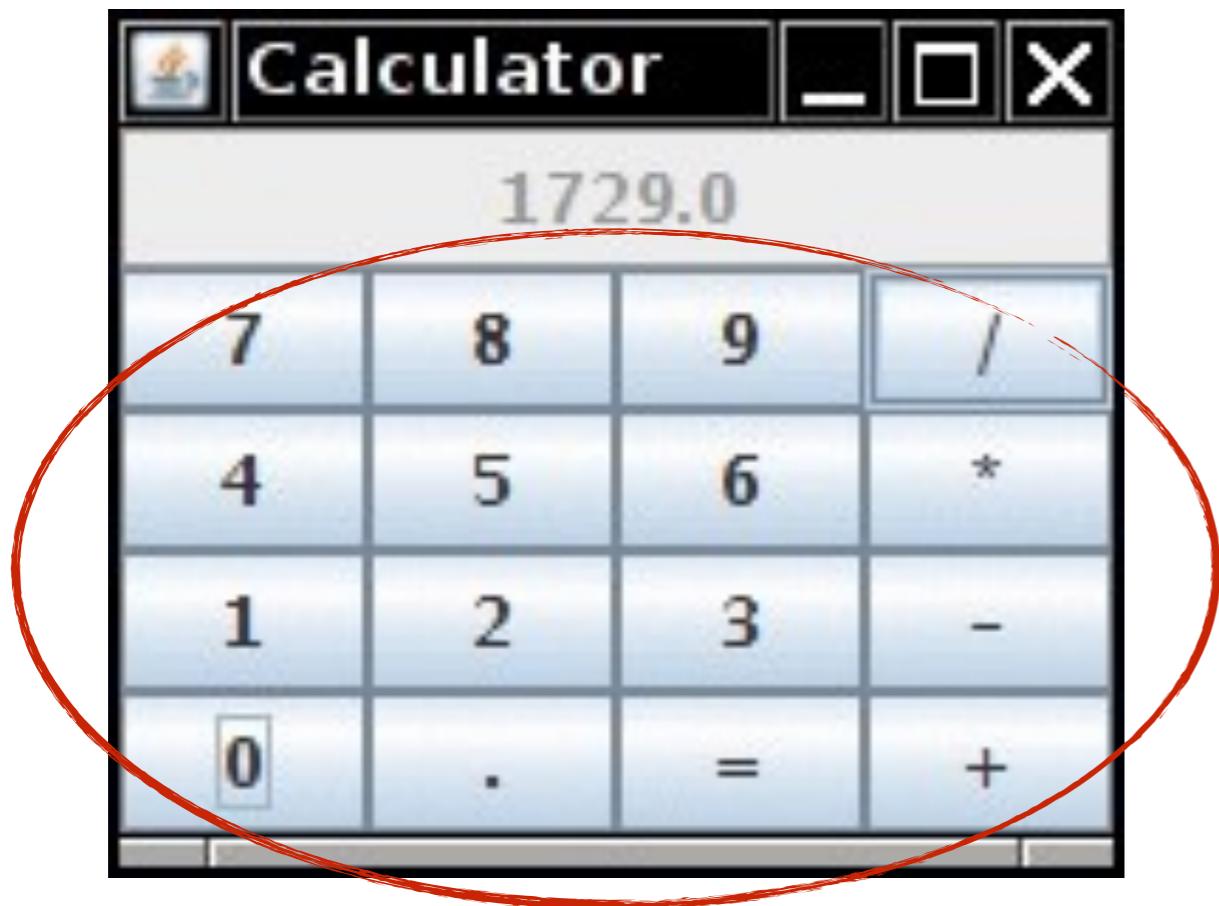


```
panel = new JPanel();
panel.setLayout(
    new GridLayout(4, 4));
```

un JPanel avec GridLayout

Grid Layout

- Les composants sont ajoutés à la grille par lignes de gauche à droite



un JPanel avec GridLayout

```
panel = new JPanel();
panel.setLayout(
    new GridLayout(4, 4));

panel.add(new JButton("7"));
panel.add(new JButton("8"));
panel.add(new JButton("9"));
panel.add(new JButton("/"));

panel.add(new JButton("4"));
panel.add(new JButton("5"));
panel.add(new JButton("6"));
panel.add(new JButton("*"));

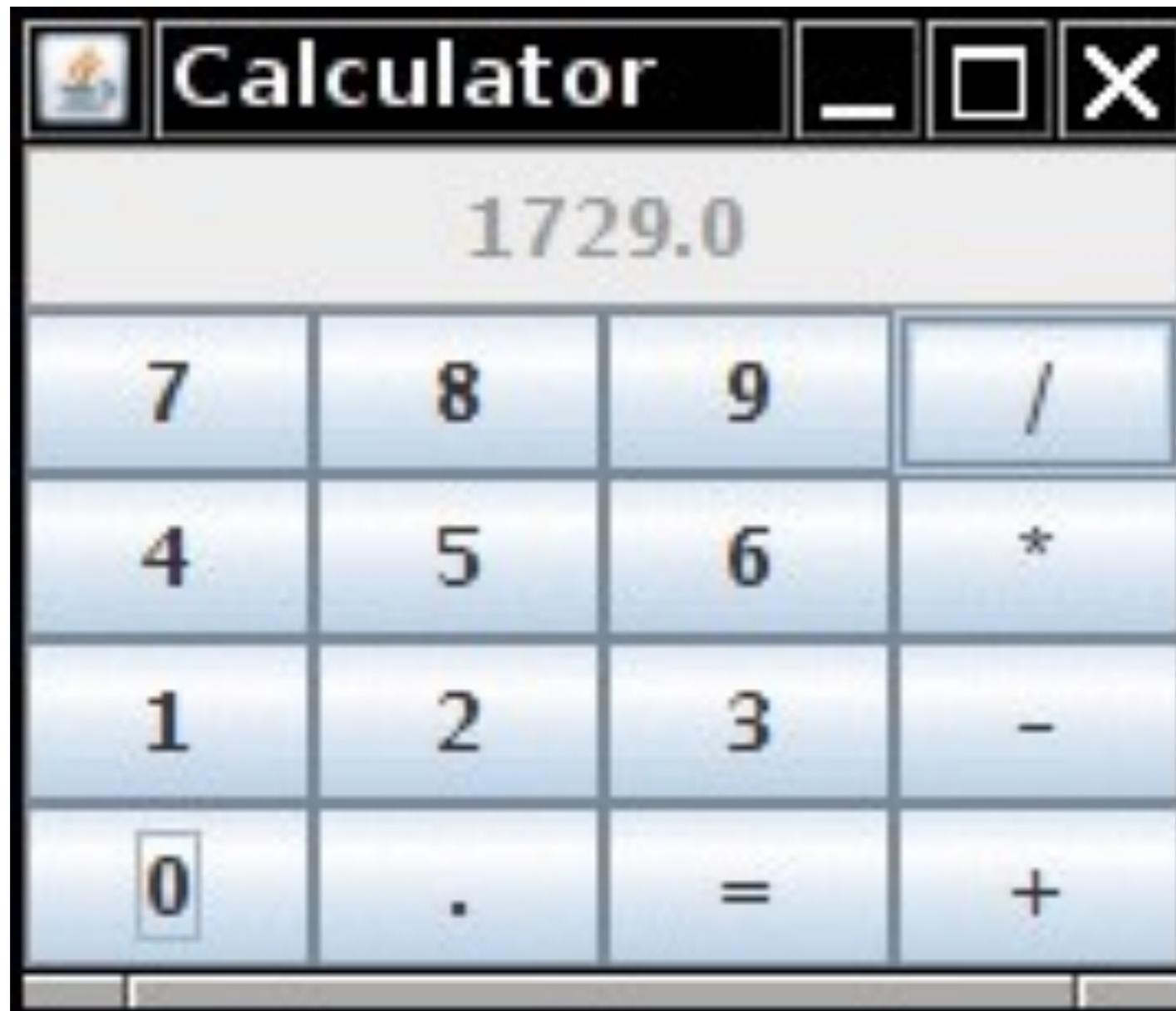
panel.add(new JButton("1"));
panel.add(new JButton("2"));
panel.add(new JButton("3"));
panel.add(new JButton("-"));

panel.add(new JButton("0"));
panel.add(new JButton("."));
panel.add(new JButton("="));
panel.add(new JButton("+"));

...
```

Grid Layout

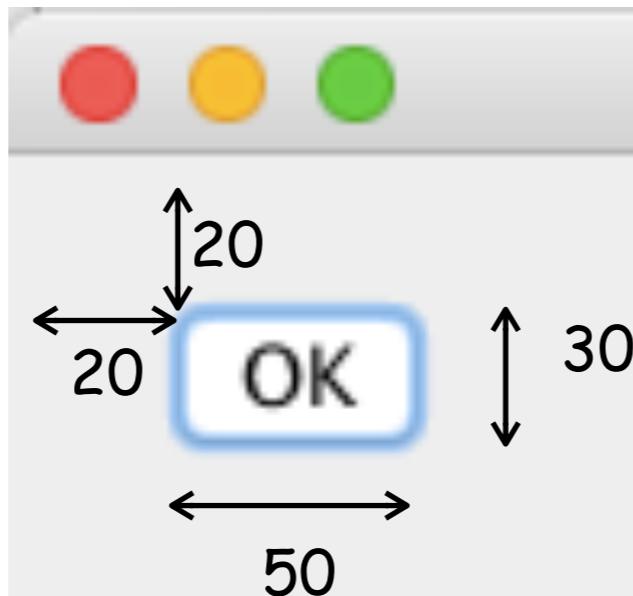
- les composantes changent en taille mais ne se déplacent pas, quand le container change de taille



Positionner sans layout

- Quand nécessaire on peut positionner les composants dans un container sans layout manager (positionnement absolu)

```
fenetre.setLayout(null);
JButton bouton = new JButton("OK");
fenetre.add(bouton);
bouton.setBounds(20, 20, 50, 30);
// cordonné x, cordonné y, largeur, hauteur
```



Design du layout

- Des Layout managers plus flexibles et complexes existent dans Swing
- Les IDE permettent de "dessiner" l'interface graphique et générer le code automatiquement
 - **Swing GUI Builder (anciennement Matisse)** de NetBeans est bien fait !
 - mais il est souvent nécessaire de peaufiner le résultat