COMP 310

# Java GUI Programming Primer

Current Home | Java Resources | Eclipse Resources

One of the big advances that Java made when it was introduced in 1995 was the inclusion of standardized "graphical user interface" ("GUI") libraries. This was a tremendous leap forward over the bewildering array of incompatible third-party libraries that were the only avenues previously open to GUI developers. The Java GUI library was also one of the first major software packages to explicitly describe its architecture in terms of the then-brand-new language of Design Patterns.

## Main Packages

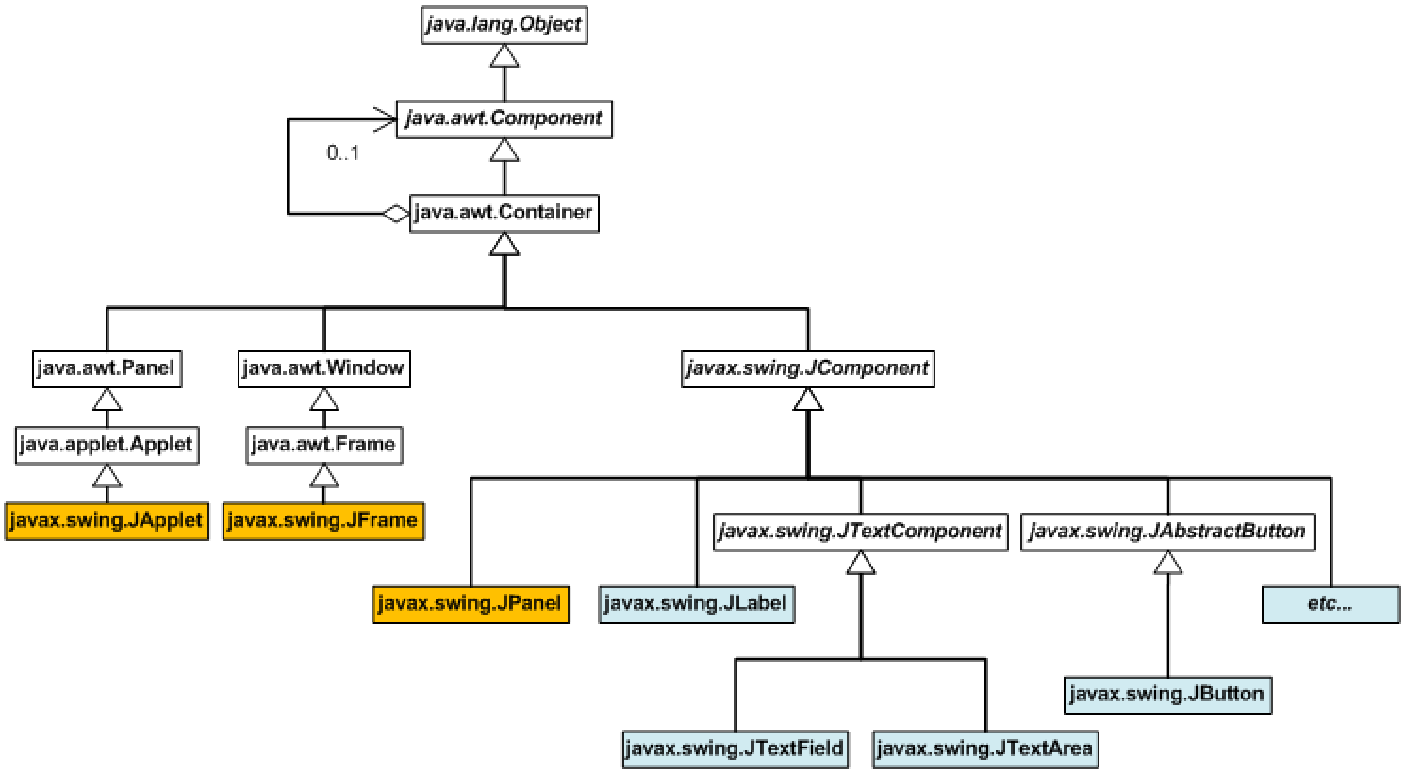
One confusing aspect of the Java GUI system however is a legacy issue where the system underwent a large-scale upgrade in 1997, so some of the classes involved are spread over the (relatively) older `java.awt` packages and the newer `javax.swing` packages. In general, if the class name starts with "`J`", then it is in the Swing package. Since some functionality appears to be duplicated in the Swing packages, such as frames and buttons, always use the Swing component over the older AWT components when there is a choice.

- `java.awt` -- Contains the main superclasses for the GUI components plus a number of utility type classes, such as `Color` and `Point`.
- `java.awt.event` -- Contains the classes and interfaces for managing events from the GUI components, e.g. button clicks and mouse movements.
- `javax.swing` -- Contains most of the visible GUI components that are used such as buttons, text fields, frames and panels.

For more complete information, see All of Java's Classes

## Class Hierarchy

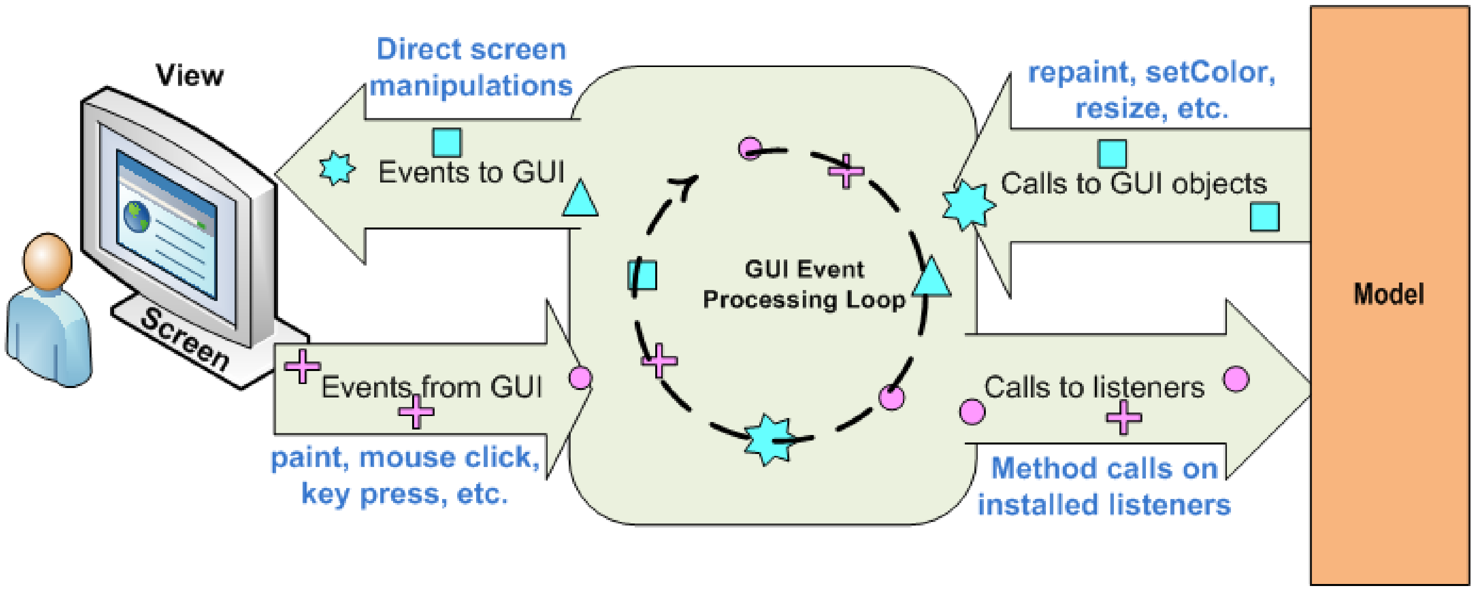
Java uses the Composite Design Pattern to create GUI components that can also serve as containers to hold more GUI components. Here is a UML diagram showing the class relations between a few of the more commonly used GUI components:



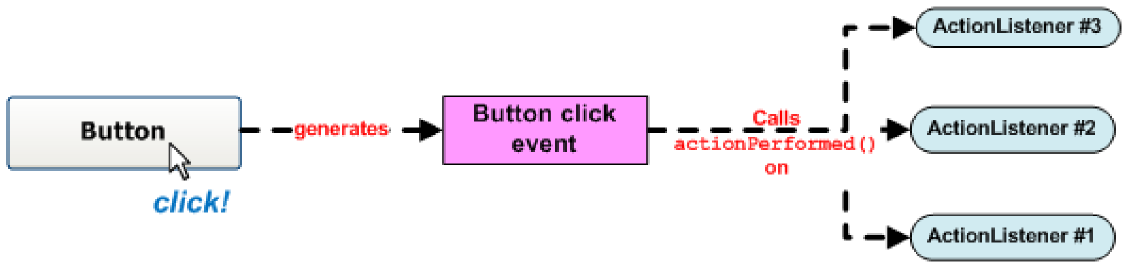
Above, the commonly used container components are in orange while the non-container components are in blue. Note that technically, all `javax.swing` components are capable of holding other `Components`, but in practice, only `JFrame`, `JApplet`, and `JPanel`, plus a few not shown above (`JSplitPane`, `JTabbedPane` and `JScrollPane`) are commonly used for that purpose.

## The Java GUI Subsystem

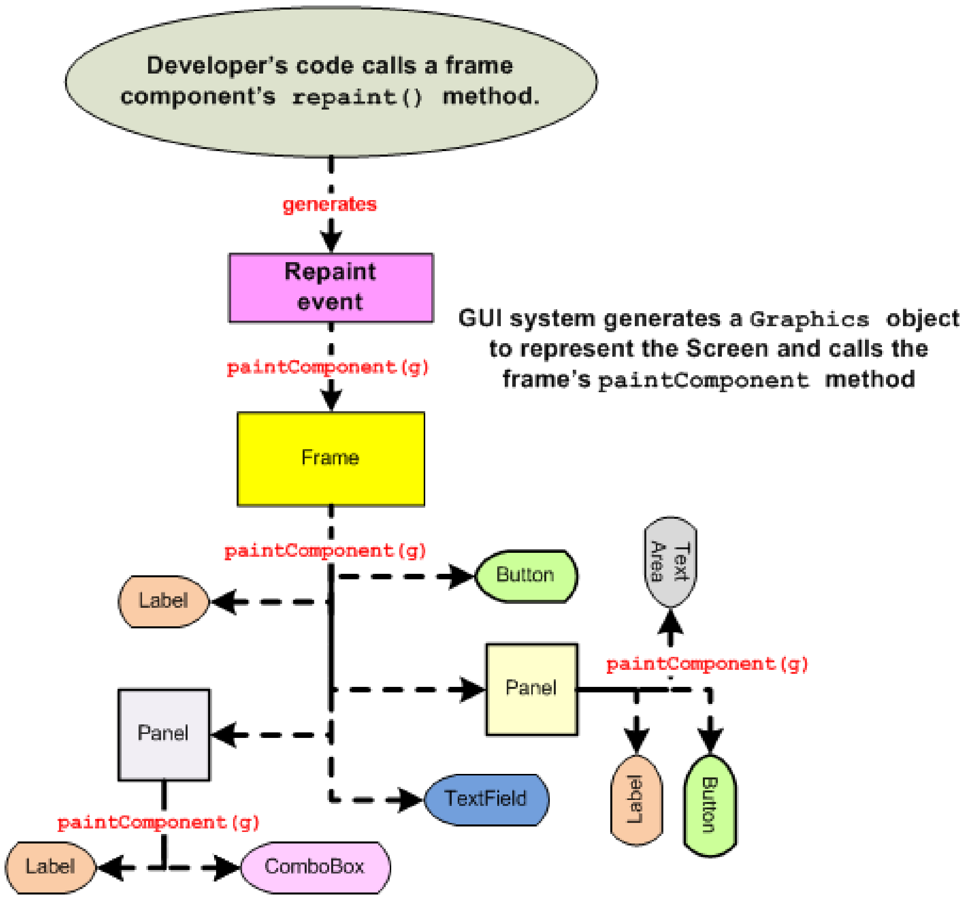
The Java GUI subsystem consists of a separate, automous task execution thread called the "event loop". Every action that affects the GUI, e.g. calls to repaint the screen or to manipulate the properties of a GUI component, or is a result of something happening to the GUI, e.g. the user clicks the mouse or hits a key, is encapsulated in the form of an "event" that is placed into a queue for the event loop to process. The result of processing an event may be a manipulation of the bits of color on the screen or it may result in calls to methods in the developer's code.


View
Screen
Direct screen manipulations
Events to GUI
Events from GUI
paint, mouse click, key press, etc.
GUI Event Processing Loop
repaint, setColor, resize, etc.
Calls to GUI objects
Calls to listeners
Method calls on installed listeners
Model

If we look at the process to handle the clicking of a button, what we see is that the user's mouse click sets off a chain of events. The button object responds to the mouse click by creating a button click event that is placed into the event queue. The event loop, when it is free to do so, picks up that event and processes it. The processing of a button click event involves calling methods on specially registered objects called "listeners" who are "listening" for the click event. The listeners for a button click are objects implementing the `java.awt.ActionListener` interface and the button click event processing involves calling the listener's `actionPerformed` method, which the developer has implemented to do whatever is needed when that particular button is clicked. Note that multiple listeners may be "added" to any given button and the button click processing will call each `ActionListener`'s `actionPerformed` method in turn.



In the other direction, if we look at the painting process, supposed the developer's code wishes to call repaint a frame component on the screen. The developer's code will call the `repaint()` method of the frame, but all that does is to generate a repaint event that is placed into the event queue along with all the other events. When the event loop is able, it processes that repaint event and in doing so, the GUI subsystem will generate a special `java.awt.Graphics` instance that is used to represent the physical screen of the computer. This `Graphics` object is passed as an input parameter to a call to the `paintComponent` method of the frame. (Technically, the system calls the frame's `paint()` method which in turn calls `paintComponent()`, but we don't need to worry about that.) Since the frame is a container, it as any good Composite Design pattern implementation will do, in turn calls the `paintComponent` method of all of its sub-components. If one of those components is itself a container, then the painting process cascades down the composite tree until every component has had a chance to paint itself upon the screen (the `Graphics` object). The developer can insert custom painting operations by simply overriding the `paintComponent` method of the desired component.


Developer's code calls a frame component's repaint() method.
generates
Repaint event
GUI system generates a Graphics object to represent the Screen and calls the frame's paintComponent method
paintComponent(g)
Frame
paintComponent(g)
Label
Button
Text Area
Panel
paintComponent(g)
Panel
TextField
Label
Button
paintComponent(g)
Label
ComboBox

## Commonly Used Classes

### Simple Components

**javax.swing.JLabel** -- A piece of non-editable text on the screen.

**javax.swing.JButton** -- A button that can be clicked. Clicking the button will cause its `actionPerformed` event to be fired which will call all installed `ActionListener`'s `actionPerformed` methods.

**javax.swing.JTextField** -- A single line of user-editable text.

**javax.swing.JTextArea** -- A box with multiple lines of text displayed. Can be set for editable or non-editable.

**javax.swing.JRadioButton**, **javax.swing.JCheckBox** -- Small round or square boxes that can be clicked on to indicate the selection of something, such as an option of some sort. Clicking the radio button or check box will fire a particular type of event. Radio buttons can be grouped together using an invisible (to the user) class called `javax.swing.ButtonGroup` to create a set where only one button can be set at a time.

**javax.swing.JComboBox** -- A drop-list of items. A `JComboBox` holds a set of `Objects`, not just `Strings`. The displayed texts are the return values of the item's `toString()` method. Thus a `JComboBox` can be used to hold arbitrary entities without the GUI knowing what those entities are. Events are fired only when the selected item changes, though the currently selected item and its index can always be retrieved.

## Container Components

**`javax.swing.JFrame`** -- a stand-alone "window" or "frame" with a title and the usual abilities to be moved, resized, minimized, maximized and closed. For most people, this is the top-level container in their system. Thus a `JFrame` cannot hold another `JFrame`. While a `JFrame` can hold any type of component, typically, a `JFrame` holds `JPanels`, which are arranged to group together sets of the components used in the window (frame). The default layout manager of a `JFrame` is the `BorderLayout`. A `JFrame` has a couple of unique and important methods that are worth pointing out:

- **`void getContentPane().add(Component c)`** -- adds a component to the frame. Note that a `JFrame` acts slightly differently than a `JPanel` here because `JFrames` have a little-used feature of multiple layers (panes) of components that can be quickly switched in and out of visibility. There are overridden forms of this method inherited from `java.awt.Container` that allow parameters for the layout manager to be passed along, such as specifying the component's position in a `BorderLayout`.
- **`void setDefaultCloseOperation(int option)`** -- sets the behavior of the frame when it is closed. See the web page on Setting the Behavior When Closing a JFrame.

**`javax.swing.JApplet`** -- Allows a Java program to be run as a component in a web page. Note that an applet and a frame are not the same thing and are not interchangeable, though there are some work-arounds for that which allow the same code to run as either a stand-alone application or an applet in a web page.

**`javax.swing.JPanel`** -- The basic building block for laying components out in a frame or inside of other panels. A JAva GUI consists of a frame with panels holding panels holding panels, etc. Note that "panels" here could also be scroll panes, split panes and/or tabbed panes (see below). The default layout manager for a `JPanel` is the `FlowLayout`.

**`javax.swing.JScrollPane`** -- Allows you to display a single panel with scroll bars on the sides, allowing more or larger components to be displayed than will fit on the screen.

**`javax.swing.JSplitPane`** -- Allows two panels to be displayed with either a vertical or horizonatal, user-moveable bar separating them.

**`javax.swing.JTabbedPane`** -- Allows multiple panels to be displayed in a "tabbed" format.

### Commonly Used Methods in GUI Containers

**`void add(Component c)`** -- adds another component to the container to be laid out as per the currently installed layout manager. There are overridden forms of this method inherited from `java.awt.Container` that allow parameters for the layout manager to be passed along, such as specifying the component's position in a `BorderLayout`.

**`void setLayout(LayoutManager lm)`** -- installs a new layout manager into the container.

## Commonly Used Methods in All GUI Components

**`void setPreferredSize(int width, int height)`** -- sets the size of a component to be used when the layout manager is able to use that size, e.g. there is enough room. Some layout managers always ignore the size setting of a component because the size is under other constraints..

**`String getText()`**, **`void setText(String s)`** -- accessor methods for the text of labels, buttons, text fields and text areas.

**`void setBackground(Color c)`** -- sets the background color of a component.

### Utility Classes

**`java.awt.Graphics`** --This abstract class is rarely instantiated by the developer's code. A machine-dependent instance of this class is handed to the `paintComponent` method of a visible GUI component during the screen painting process -- *it is NOT instantiated by the developer's code but rather by the Java GUI sub-system*. The developer can writed code that will use that supplied `Graphics` object instance to draw lines, shapes and images onto the screen. For advanced graphics work, it should be noted that the supplied `Graphics` object can always be safely downcast to the more capable java.awt.Graphcs2D class.

**NEVER CALL `paintComponent` DIRECTLY FROM YOUR OWN CODE.** ***Likewise, NEVER INSTANTIATE A*** **`Graphics`** ***OBJECT***--the Java GUI system will automatically hand your code the `Graphics` object that it needs.

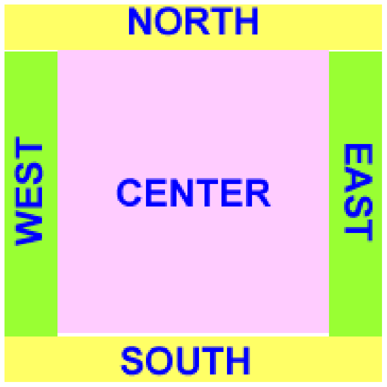
**`java.awt.Color`** -- Represents a color. Has static fields predefined to common colors plus can be expressed in a number of different formats, such as 3-bit RGB values.

**`java.awt.Point`** -- Represents an point on 2-dimensional plane as specified by integer-valued Cartesian coordinates.. Has fields to retrieve the `x` and `y` coordinates plus methods for operations such as calculating the distance between two points. For *floating point-valued* Cartesian coordinates, use the `java.awt.geom.Point2D` class.

## Layout Managers

Java containers use the Strategy Design Pattern to implement automatic layout management of GUI components on the screen. The management of the size and positions of the contained components is delegated to a "layout manager", which can be set at design time or dynamically changed. Here are a couple of common layouts:

**`java.awt.BorderLayout`** -- Separates the component into 5 distinct areas, 4 non-resizeable areas on the edges ("`North`", "`East`", "`South`", and "`West`") and one re-sizeable area in the center ("`Center`"). *Each area can only contain 1 component, so typically, panels are used in each area to hold multiple components.*



A typical statement that adds a component to a container that is known to have a `BorderLayout` installed would be

```
myContainer.add(myComponent, BorderLayout.NORTH);
```

**`java.awt.FlowLayout`** -- Holds multiple components in a horizontally arranged line that will wrap around if the width of the container is too small. A `FlowLayout` does not require any additional parameters when a component is added to the container -- the components will display left-to-right in the order in which they were added.

**`javax.swing.BoxLayout`** -- Holds components in either a vertical or horizontal arrangement that is fixed, independent of the size of container. The width and height of the contained components are all identical and are automatically set tocompletely fill the container.

## Event Handling in Java GUIs

Java uses the Observer-Observable Design Pattern to handle events from GUI components, e.g. button clicks and mouse movements. Components supply methods to add various types of listeners, e.g. "`void addActionListener(ActionListener al)`" and "`void addMouseMotionListener(MouseMotionListener mml)`". Multiple listeners can be added and all will be called when the associated event occurs, though there is no guarantee as to the order in which the installed listeners will be called.

**`java.awt.event.ActionListener`** -- Interface used to detect a simple action on a component, such as clicking a button or hitting `Enter` on a text field.

**`java.awt.event.MouseListener`** -- Interface used for event listeners watching for mouse click events. This event allows more detailed capture of the mouse click than an `ActionListener` event, for instance, distinguishing between a mouse button pressed *vs*. a mouse button released events. A convenience class, `MouseAdapter`, is also available that no-ops all the methods, freeing the developer to only write the code that overrides the desired methods.

**`java.awt.event.MouseMotionListener`** -- Interface used for event listeners watching for mouse motion events, e.g. dragging. A convenience class, `MouseMotionAdapter`, is also available that no-ops all the methods, freeing the developer to only write the code that overrides the desired methods.

## How to start a GUI-based program

In general, a GUI-based program will start whenever the first GUI component is shown (visibility set to true). Typically that would be one of the frames in the system, so one would have a line of code like such:

```
	myFrame.setVisible(true);
```

This a quickie start for a GUI program would have a `main` method like such:

```
	public static void main(String[] args) {
		java.awt.EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					(new MyFrameClass("The title")).setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}
```

(Note:  The above code correctly instantiates the frame on the GUI thread as specified by the official Java requirements.   The code can be simplified a little bit by using a lambda function instead of the anonymous inner class implemenation of a `Runnable`.)

In Model-View-Controller architectures however, it is **not** recommended that the view, i.e. the main frame, be started by simply setting its visibility to `true`.  It is much better to define a "`start()`" method on the frame or main view class, where all final initializations and checks are performed and whose last line of code is the `setVisible(true)` statement.

A typical controller in an MVC will instantiate the model and the view, connecting them together with their respective adapters.  After all that is done, the controller's *last* act will be to call the view's `start` method.

NEVER START THE GUI FROM THE CONSTRUCTOR OF THE FRAME!   This is simply asking for trouble as many of the other components in the system are probably not fulling instantiated or intialized yet.  But as soon as the first frame becomes visible, events will start firing, and those listeners will expect fully operational objects to be present but won't find them, causing, at best, a flurry of null pointer errors.

## Additional Resources

- Animations in Java
- Dynamically modifying GUIs
- Cross-thread invocation issues with GUIs
- Tutorials from Sun/Oracle:
  - The Java Swing Tutorial
  - Java 2D Graphics Tutorial
- Java Swing GUI Tutorial from Chee Yap at NYU
- An interesting collection of tutorials with short descriptions from John Russell  (Niagra College, retired).
-