Lecture 17 - Graphic User Interfaces

The ***AWT*** (Qu: What is it? Ans: ***Abstract Windowing Toolkit***) is a platform independent collection of classes the support GUIs (Qu: What is it? Ans: **look and feel**). It works on any system (which is one reason why Java is so popular, since others such as C don't have a GUI library, or are platform-specific such as VB, or are a scripting language such as Perl/Tk), but the look and feel of windows and buttons can vary form system to system.

There's a nice series of videos explaining basic computer graphics on YouTube.

There are many GUI toolkits available: Qt, TK, Gtk+, SWT, FLTK, ...) only some of which are portable.

Among third party toolkits, SWT is developed and maintained by the Eclipse foundation and is thus well supported. Gtk+ and Qt are popular for Unix and Linux systems, although Gtk+ hasn't advanced as quickly as Qt and lags behind on features and portability. Major commercial vendors had largely standardized around Gtk+ because its permissive licensing made it a less expensive choice for proprietary software vendors. Qt was originally created by Trolltech, a commercial software vendor that offered the toolkit under a dual-licensing model.

After struggling for several years with the limitations of Gtk+, Nokia acquired Trolltech in 2008 with the aim of adopting Qt as the standard unifying development toolkit across its Symbian and Linux-based mobile platforms. Nokia relicensed Qt under the LGPL in 2009, eliminating the commercial licensing barrier that had previously impeded broader adoption. Companies that already use Qt for cross-platform development include Google, Amazon, Skype, Adobe, Canonical (Ubuntu Linux), and others.

The GUI code in Java has changed dramatically with each release: **Java 1.0 used a *hierarchical*** method (based on containment / inheritance) to signal events (e.g., mouse click). Java 1.1 uses a much better method that uses registration and call-backs, called the ***delegation* model**. Java 1.2 (Java 2) uses this same stuff as Java 1.1, but replaced many parts of the AWT with newer, better pieces. The new stuff is called ***swing***. (That's not an acronym; the Java mascot is called "Duke" after Duke Ellington, a famous swing musician.)

The basic idea is that **GUI *components* are created and added to *containers*, which are also components**. In AWT when a GUI component such as a `Button` is created the **JRE automatically creates a *peer*** button object.

A ***peer*** is a native GUI component: on Windows it is a Win32 button, on Macintosh it is a Macintosh button, etc. (Show `java.awt.Toolkit.createXXX()`, point out you never use these directly.) When the user clicks on the button on the screen with the mouse, the OS tells the JVM, which tells your `Button` object it was clicked on.

Every different platform implements the JRE differently. The use of peer objects is why windows and buttons look like other windows and buttons on that platform. This is often a good thing!

In addition to AWT and swing, Oracle is pushing (2012) JavaFX, a JRE designed for *rich Internet applications*, or RIAs. JavaFX works more like CSS and HTML. JavaFX *may* someday become more popular than swing.

Heavyweight vs. *Lightweight* Components

A "heavyweight" component is an opaque component that has a native *peer*. It is like a sheet of paper (often called a *canvas*) you can draw on. The original AWT only had heavyweight components, but v1.1 added some lightweight ones (you can extend `Component` and `Container`). Top-level containers such as `Frame`s must always be heavyweight.

A "lightweight" component is a "virtual" component **with no native peer** (and thus no *canvas*) of its own. The painted pixels of a lightweight component draw on the canvas of its "closest" heavyweight ancestor. This can get confusing when heavyweight and lightweight components overlap each other!

Lightweight components support transparency. If a lightweight component leaves some of its associated bits unpainted, the underlying component will "show through". (**Show `HeavyLight.java`.**)

Swing vs. AWT

Swing doesn't use peer objects. Nearly every swing component is lightweight (Not `JFrame`). (However swing components have a `setOpaque()` method to prevent transparency, simulating heavyweight components.)

With swing, the programmer has complete control of the look (and feel) of buttons, windows, etc. Your program will look exactly the same regardless of which platform it runs on. Swing is more powerful and has many convenience methods and supports many effects not possible (at least not easily!) in AWT. However this means a swing GUI may not match the rest of the user's desktop.

You can change the default look and feel (called "*metal*") or even create your own custom look and feel with swing. (Sun calls this ***Pluggable Look And Feel*, or *PLAF***.) To support this, swing includes an entire windowing system, which doubled the size of the previous JRE. (Show `IntCalc.java` swing+PLAF.) PLAF is discussed further, below ("Swing GUIs").

Swing has problems with multi-threaded programs, and is more complicated (to take advantage of the newer features, you must know ***model-view-controller***). Also, all event handling with either AWT or swing is done using AWT events.

Whether to use AWT or swing (or some other GUI toolkit) is up to you. You create GUI programs in almost the same way no matter which you chose. Most modern textbooks only show swing. (This is unfortunate since swing is built on top of AWT and you must know AWT to use swing effectively.) I will use AWT in class but will show swing too.

Warning: Don't mix swing and AWT components in the same application; stick to one or another toolkit. (With Java 7, some mixing is possible.)

Event-Driven programs

All GUI toolkits have a main event loop, the ***AWT Event (handling) Thread***:

```
while ( true )
{   wait_for_event;
     create_event_object evnt;
    dispatch_to_proper_obj;  // obj.dispatch( evnt
)
}
Component Dispatch:
for ( each registered listener obj )
```

```
{   obj.method( evnt );  }
```

Note this thread keeps applications running even after the main thread terminates.

Demo simpleGUI.java window:

```
class simpleGUI
{  public static void main (String[] args)
   {  Frame myWin = new Frame("My Window");
      myWin.setSize(300, 200);
      myWin.setVisible( true );// or use show
}  }
```

Discus ***focus***: If the user clicks on a focusable `Component` **a** of an inactive `Frame` **b**, the following events will be dispatched and handled in order:

1. b will receive a `WINDOW_ACTIVATED` event.
2. Next, b will receive a `WINDOW_GAINED_FOCUS` event.
3. Finally, a will receive a `FOCUS_GAINED` event.

If the user later clicks on a focusable `Component` **c** of another `Frame` **d**, the following events will be dispatched and handled in order:

1. a will receive a `FOCUS_LOST` event.
2. b will receive a `WINDOW_LOST_FOCUS` event.
3. b will receive a `WINDOW_DEACTIVATED` event.
4. d will receive a `WINDOW_ACTIVATED` event.
5. d will receive a `WINDOW_GAINED_FOCUS` event.
6. c will receive a `FOCUS_GAINED` event.

Default keys to traverse forward to the next `Component`:

`TextArea`s: CTRL-TAB on KEY_PRESSED
All others: TAB on KEY_PRESSED and CTRL-TAB on KEY_PRESSED

Traverse backward to the previous `Component`:

`TextArea`s: CTRL-SHIFT-TAB on KEY_PRESSED
All others: SHIFT-TAB on KEY_PRESSED and CTRL-SHIFT-TAB on KEY_PRESSED

Focus is more complex than this. When you type a keystroke (shortcut key) to trigger an (say) a menu action, the current component won't know about it. There is a hierarchical event processing model, so on a keystroke event, the focused component, all its parent containers (in order), and then all visible and enabled components in the same window, are checked for a handler.

Note clicking the close button in `SimpleGUI1` does nothing. Why? Discuss **`setVisible`** (for all components, including Windows, Frames, Panels, and Applets), `Window.`**`toFront`** (forces layout and brings to top if already visible). Note methods `toBack`, `Frame.setExtendedState(Frame.NORMAL|ICONIFIED).`

Note the easy window control in available in swing (Show in Java docs.):

```
JFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOS
E)
```

Convert SimpleGUI3, 6 to swing: *component* →J*component*, Pre-Java5: add(X) →**getContentPane()**.add(X). Changed `JLabel` to `JButton`, note use of `final` local variable, add event handling to change `JButton` color (`setBackground`(..)). **Show [J]`Frame.pack`**`()`.

Demo PopUp.java. Discuss ***Mouse, MouseMotion****, and MouseWheel events*. (MouseAdapter implements all three.) Note a `Window` is a simplified `Frame`, or, more accurately, a `Frame` is a specialized `Window`. Point out that the system double-click interval is 200 milliseconds.

Pages 485–487 in the book (7th ed) lists events and listeners. (**Show EventChart.htm**). Point out should know about all event types even the ones not covered in class. Point out what generates **`ActionEvents`** and **`KeyEvents`**.

Show sketcher. (To discuss Mouse Events some more, paint and update, and **insets**.)

MouseEvent handling since 1.4: `getModifiers()` has problems telling the difference between say `ALT` and `BUTTON3`. `getModifiers()` returns which buttons/modifiers changed (note the use of *bit-wise operators* and *`int` masks*). To see exactly what buttons and modifiers are down at the time of the event, use:

```
if((e.getModifiersEx()&e.BUTTON3_MASK)==e.BUTTON3_M
ASK)
```

```
if ( e.getButton() == MouseEvent.BUTTON3 ) // New
way
```

Some systems (such as Linux) support multiple desktops. On such systems, GUI elements may not appear on the currently showing desktop by default. On such systems, you can try code like this:

```
JFrame f = new JFrame(
  GraphicsEnvironment.getLocalGraphicsEnvironment()

.getDefaultScreenDevice().getDefaultConfiguration()
);
JOptionPane.showMessageDialog(f, "Test");
f.dispose();
```

Menus

`Menu`s are `MenuItem`s and contain other `MenuItem`s. (So you can add a `Menu` to a `Menu`.) `Menu`s are added to a `Menubar`, and a container can have a `Menubar` set for it. A `Frame` is the only AWT container that supports a `Menubar`; swing allows any container to have `Menu`s. (Note in AWT, other containers such as `Applet`s can have `PopupMenu`s, usually triggered by a right-click.) Selecting a `MenuItem` generates an `ActionEvent`. So you need to add an `ActionListener` for each. You can add a separator (horizontal line) with either `Menu.addSeperator()` or by adding a `Menuitem` of `"-"`).

There are different types of `MenuItem`s, such as checkbox, images, etc. A `MenuItem` can have a *mnemonic key* defined (shows as underlined) used with ALT to activate. You can also set an *accelerator* key (**show MenuDemo**).

Rarely useful, you can use the older hierarchical event handling system to add a listener for action events on the menu itself, rather than on each menu item.

More useful is the swing support for `Action` objects, which can be used from menus, buttons, or a toolbar.

Painting Graphics [*From* http://java.sun.com/products/jfc/tsc/articles/painting/]

In AWT, there are two kinds of painting operations: system-triggered painting and application-triggered painting. ***System-triggered* painting occurs when a component is first made visible, is resized, or the component has changed its appearance.** For example something that was covering the component has moved and a previously obscured portion of the component has become exposed. In this case a ***paint event*** is placed on the event queue, which eventually invokes the component's `paint()` method.

With an *application-triggered* painting operation, the component decides it needs to update its contents because its internal state has changed. (For example a button detects that a mouse button has been pressed and determines that it needs to paint a "depressed" button visual). This can also be triggered by invoking the `repaint` method. In this case a ***repaint event*** (this is a name I give to application-triggered paint events; this is not a standard term) is placed on the event queue, with consecutive repaint events collapsed into one. This eventually invokes the component's `update` method.

> The difference between paint and repaint events is that paint events call the `paint()` method directly; repaint events call `update()` which then calls `paint()`. Also multiple consecutive repaint events may be collapsed into a single event.

The default `update` method for heavyweight components clears the component (with `g.clearRect()`) and then calls `paint`. The `update` method of lightweight components simply invokes paint (no `clearRect`.) In some cases this can lead to ***flicker***. **If you don't want a heavyweight component cleared, you should override `update()` to simply invoke `paint()`.** This is called *incremental painting*.

> Components that render complex output should invoke `repaint()` with arguments defining the rectangular region that requires updating. The rest of the component (outside this rectangle) is not affected by the call to update or paint, reducing flicker and improving rendering time. A common mistake is to always invoke the no-arg version, which causes a repaint of the entire component, often resulting in unnecessary paint processing.

Normally components don't overlap much except for a container and its added components. **When the container needs to be refreshed** (that is a system-triggered paint event occurs or the layout manager causes an application-triggered paint event), its paint method recursively triggers paint or repaint events on the affected components in that container.

Things get more complicated when components overlap. What happens depends on the type of component (heavyweight or lightweight) and the type of paint event (application or system triggered).

Heavyweight components are easier to understand and work with. **When two or more heavyweight components overlap, they are drawn (or *rendered*) in a back-to-front order.** The last component added is *under* the previous ones (where they overlap). (**Show OverlapHeavy.java.**) Either paint or update method is invoked, depending on the type of paint event.

Lightweight components are more complex. Lightweight components can never cause system-triggered paint events since the "system" doesn't even know about them. So any paint events that come from lightweight components behave as application-triggered, even if it seems they were system-triggered. This means `update()` is always called for lightweight components for any paint event. For this reason the default implementation of `update()` will not clear the background if the component is lightweight.

Another issue is that lightweight components aren't "real". You can't draw one by itself. **Lightweight components are always attached to the top-most heavyweight component.** As each heavyweight component is drawn, the lightweight ones on top of it are drawn next.

When a lightweight component is the source of a paint event, the underlying heavyweight component's `paint` method is called. That in turn must call `super.paint` to invoke the lightweight components' `paint` methods. (The default `Container.paint()` method handles this, so if you override that be sure to invoke `super.paint()` or lightweight components won't get redrawn!) (S**how PaintDemo;** comment out the `super.paint` call, move it to the top of `paint`.**)**

Painting in Swing

Swing components have multiple parts to them called ***panes***, especially containers. They may have borders, a *content pane* (that may have multiple layers to support fancy graphic and animation effects), and children components. They are nearly all lightweight components which further complicates painting. A swing component's `paint` method has a lot of work to do! If you override `paint()` be sure to invoke `super.paint()!`

Mostly in swing you don't need or want to repaint the whole component, just the content pane (or you could cause flicker). So swing components have a `paintComponent` method you use like you use paint in AWT. (With lightweight components in content panes (almost always the case with swing) you still need to call `super.paintComponent()`.) Note that

`JFrames` are still heavyweight components in swing so you use `paint` for them as normal.

Discuss mixing of **adding components and overriding `paint` or `paintComponent` (must invoke `super.paint` to have added components show**). If `super.paint` is first, then later `g.drawXXX()` output will appear on top of lightweight components ("DEMO VERSION"); if last then components on top (e.g., wallpaper effect).

Swing containers have a special ***glass pane*** that covers the whole container; You can draw (lightweight) on that to cover a (part of a) window, with a splash screen for instance.

In a multi-screen environment, the class `java.awt.GraphicsConfiguration` can be used to render components on multiple screens. It has methods to list all screens and printers available.

A note about using colors: We have already covered the `java.awt.Color` class, but you should also know about `SystemColor` class. This class defines `Color` constants for the system UI: the current color for windows, titlebars, menus, etc.

Fonts

Discuss `font f = new Font("SansSerif", Font.BOLD, 18)`

public Font(String *name*, int *style*, int *size*)

Creates a new Font from the specified name, style and point size.

name - The font name. This can be a logical font name or a font face (*physical* font) name. (**Demo UnicodeSymbols.java to show how to list available fonts: `java.awt.GraphicEnvironment`**.)

Discuss *proportional* versus *monospaced* fonts, *serif* versus *sans serif* fonts. (Demo Times, Arial, and `Courier` using "Now is the time 1 2 3" in 24 point, in Notepad.)

A ***logical name*** must be one of these:

- **Serif**
- **SansSerif**
- **Monospaced**
- **Dialog**

- **DialogInput**
- **Symbol**

The mapping from logical to physical font names is complex; each logical fonts is actually comprised of several physical fonts “stitched together”. This allows more Unicode code points to be available than are supported in any one physical font. The mapping is defined in …`/jre/lib/fontconfig.properties.src`.

You can count on “Lucida” font family present as part of JRE, since 1.2. (Show …`\jre\lib\fonts`.)

> You can also download and convert TrueType, OpenType, and PostScript type 1 fonts, into Java `Font` objects you can use:
>
> ```
> InputStream fStream = new
> URL("...").openStream();
> Font f = Font.createFont(Font.TRUETYPE,
> fStream);
> f = f.deriveFont(Font.BOLD, 12F);
> fStream.close();
> ```

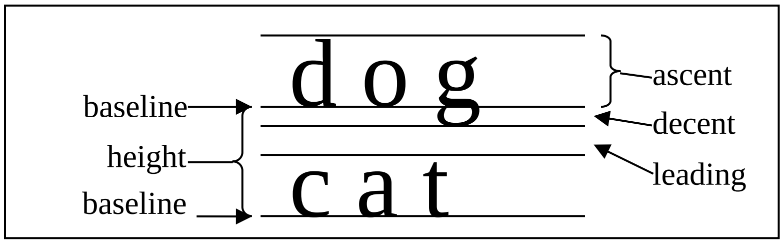
style - The style constant for the Font. The style argument is an integer *bitmask* that may be **PLAIN**, or a bitwise union of **BOLD** and/or **ITALIC** (for example, ITALIC or BOLD|ITALIC). Can also use “BOLD+ITALICS”. Any other bits set in the style parameter are ignored. If the style argument does not conform to one of the expected integer bitmasks then the style is set to PLAIN.

> Note a missing italics version style of some font will get rendered as a slanted version of the plain or bold fonts.

size - The *point size* of the Font. (Discuss the difference between ***points*** (~1/72 in.; 1 point = .013837 in. so 72 pt. = 0.996264 in.), ***picas*** (12 points = 1 pica), and ***pixels*** (Originally same as a point, but on modern screens may have many more pixels to the inch.)

Since ***bit-mapped*** fonts are actually a collection of graphics, they will shrink if you use a higher resolution (more dpi). In theory, ***true-type*** and similar fonts will scale appropriately but in practice many systems report to the JRE an incorrect (or no) dpi value, and will render such fonts as if 1 pixel = 1 point, the same as for bit-mapped fonts. In general you should use logical fonts (and variants).

[Adv: typography terms: ***leading***: space between elements;
X-height: typical or average height of non-capital letters (different fonts with the same point size but different X-heights will appear as different sizes); ***baseline***: the bottom edge of a line of text (***descenders*** go below that, e.g., "y").



em (horizontal measure equal to the point size, which is the width of a capital 'M'); ***en*** (the width of 'N'); *dashes* (em, en, hyphen, minus); ***ligatures*** ("fi" = ﬁ, "fl" = ﬂ, etc.)

]

Discuss **FontMetrics**. Discuss "**`stringWidth`**" method to align columns.

[Adv. Show **Tempconv2.java**. (If you need more accuracy, use a **`LineMetrics`** object instead.) You can also get (and the draw if you want) the bounding box for a string of text `s`, from a `Font f`, and a `Graphics2D g2`:

```
FontRenderContext frc =
g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds( s, frc );
int width = (int)Math.round(bounds.getWidth());
LineMetrics lm = f.getLineMetrics( s, frc );
```

*See **Font Concepts web resource** for more background information (not on exam).*

]

Remind students **the exam covers** window, mouse, menu and button (`Action`), and key events, (you should know which listener is used for each, but only details for action events), and the components: `Frame`s, `Panel`s, `Button`s, `Label`s, `Menu`s, `TextField`s, and `TextArea`s. Be generally familiar with `Font`s (including logical font names), and the Flow-, Border-, and Grid- Layout Managers.

Swing GUIs

Swing is a sophisticated GUI toolkit, built on top of the AWT toolkit but with many extra features and abilities. However the power of swing comes at a price: it is much more complex to use (***model-view-controller*** or **MVC** paradigm, actually in swing more of a *document-user interface* or M-UI paradigm). When the user interacts with the UI, your code catches the events and updates the document.

Working with swing's PLAF is easy, at least in Java 6 and newer. To make swing use the native system's look and feel, add this to the top of your program:

```
   try {
     UIManager.setLookAndFeel(
       UIManager.getSystemLookAndFeelClassName());
   } catch ( Exception e) {}
```

You can view the Java docs for the `javax.swing.UIManager` class to see additional methods exists for discovering which UI (*user interfaces*) are available on your system.

The PLAFs available vary by vendor. Oracle has added a nifty new one in Java 7 called nimbus. Unlike most PLAFs, nimbus uses vector graphics, so it should look smooth at any resolution. A third-party PLAF called napkin shows the UI with a "hand-drawn" look. That could be useful to show clients a prototype UI.

Swing is not *thread-safe* the way AWT is. This means that if you update the GUI from multiple threads, the result may not look right (or even work at all). Of course most of the time it works fine, but why take needless risks?

The correct way to update a swing GUI is to run the code from the AWT Event Handling Thread (a.k.a. the *Event Dispatch Thread*, or EDT). This can be done immediately with `SwingUtilities.invokeAndWait`, or just added to the event queue with `SwingUtilities.invokeLater`. (***Show swing image demo, Smile3.java***.):

```
SwingUtilities.invokeAndWait( new Runnable()
   {  public void run () {  createGUI();  }
   } );
```

(Since Java 1.3, these two methods just invoke the similarly named methods from `java.awt.EventQueue`. You can use that directly if you wish.)

A consideration (applies to any GUI toolkit) is that if you need to create a complex GUI, say a custom animation by downloading images, you will have a "stalled" program if this is done from the `main` Thread. But if you do this from the EDT, the GUI will appear unresponsive. Using `SwingUtilities` doesn't help here; you need to use a background thread ("*worker*" thread) to deal with time-consuming GUI tasks. And that thread must still be careful to update the GUI only from the EDT.

> The `JOptionPane.show`*`XXX`* methods and some of the *`JXXX`*`Chooser.show`*`XXX`* (e.g. `JFileChooser`) methods apparently don't need to be invoked on the EDT. Some Swing components (and some of their methods) are marked "thread-safe" in the docs; those can be invoked from any thread.

A utility class called **`SwingWorker`** has been used to make it simpler to do this. It has many features and methods. See the Java docs for examples and more information (also the uiswing/concurrency section of the on-line Java tutorial).

To make a custom component in swing, extend `JComponent` and override its `paintComponent` method; often that starts or ends with a call to `super.paint.Component`. You can also extend `JPanel`, but that is opaque by default, so you must call `super.paintComponent` first, to draw the proper background.

> Since `JFrame`s have convenient methods to add stuff to the content pane, it is easy to forget about it and run into trouble. To set the background color in swing, you can use:
>
> ```
>  getContentPane().setBackground( Color.GREEN )
> ```

SWT and JFace (Eclipse GUI Toolkit)

Standard Widget Toolkit is what the AWT should have evolved into. SWT provides a low-level API (like AWT) of GUI components called *widgets*. (X Window calls them this too; MS calls them *controls*.)

SWT uses a more complete API than AWT and supports a "superset" of the functionality of the various platforms. When possible a native implementation is used, and missing features are emulated. The result is a rich, high performance GUI API that provides a native look-and-feel.

JFaces is a higher-level API, implemented on top of SWT. It provides an easy to use, high-level of abstraction API and provides higher-level widgets such as trees, tables, etc.

SWT also uses a different event model than AWT (or swing).

SWT and JFaces are used to produce Eclipse. You can download the jar(s) for this, add them to the extensions (“`ext`”) directory, and start using this GUI system instead of AWT or swing.

New AWT features in Java 7

Class `Window` has a new method `public void setOpacity(float opacity)`, where *opacity* is between 0.0 (fully transparent) and 1.0 (fully opaque). You can also use the *alpha* channel of a background color for this.

New methods to position components, e.g., `Window.setLocationRelativeTo`. (With an argument of `null`, this will center the window on the screen.)

`Window`s don’t have to be rectangular anymore. Use `Window.setShape(Shape shape).`

Lecture 18 — Layouts

Containers (which extend Component) support *Layout Managers*. These allow components in a window to adjust automatically if the window is resized, or if additional components are dynamically added to the window. The developer (you) can easily center components without pre-calculating their positions. Components can also be automatically resized. Some (not all) layout managers ask a component about its ***preferred size*** or its ***minimum size***, and stretch or shrink the component as needed. (In AWT, you must extend a class to override these; in swing, there are set methods you can use.)

> When creating custom components you should provide `getPreferredSize` and `getMinimumSize` methods for the layout manager to use.

If you don't want all this, you can turn off the layout manager then position and size all components yourself manually:

```
setLayout( null );
```

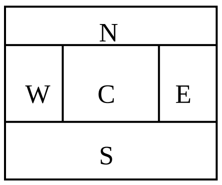
Use **`setSize`**, **`setLocation`**, or **`setBounds`** to size and position the added components (defaults to 0 width, 0 height at point 0,0). Note that each container requires its own layout manager object, even if two containers are using the same type of layout.

If using a layout manager than manually setting size or location will have no effect. Also when a component is added, removed, or changed, it is ***invalidated***. A Container has a method **validate** (swing **revalidate**) to re-layout and re-paint components if needed.

BorderLayout (default for Frames) `BorderLayout(`*`hGap`*`, `*`vGap`*`)` or `()`

(Example: `setLayout( new BorderLayout() );`)

This is my favorite of the basic layout managers. The constraint when adding a component to a container with BorderLayout is "`East`", "`West`", "`North`", or "`South`", or `BorderLayout.EAST`, etc. A container that uses a BorderLayout is subdivided into five areas as shown:



Each area should contain only a single component; however that component could be a Panel. The components in the North and South are drawn first and with their preferred height, but are stretched horizontally the full width of the container. Next the East and West components are drawn, with their preferred width but are stretched vertically to fit between the North and South components. Finally, the component in the Center is drawn, and it is stretched (or shrunk) to fill any remaining space. Note it is quite possible that the height or width of the Center is zero, and the component drawn there may not show up!

FlowLayout (default for Panels)

`new FlowLayout(`*`align, hGap, vGap`*`)` or `(`*`align`*`)` or `()`

align is one of `FlowLayout.RIGHT`, `.LEFT`, or `.CENTER`. Default gap is 5 pixels between components. With `FlowLayout`, components are added in rows starting at the top, from left to right. When attempting to add a component at the end of a row that won't fit, the layout manager starts a new row. Using `FlowLayout` components are always drawn with their preferred sizes. (The only layout manger where this is true.)

Aside: To achieve a visual **effect**, it is often necessary to use `Panel`s, a container used to group and align other components. For example, adding a `Button` to a container with `BorderLayout` will stretch the Button. However, you can `add` a `Button` to a `Panel` (with `FlowLayout`), then add that `Panel` to the container with `BorderLayout`. The (invisible) `Panel` will stretch, but the `Button` will be drawn at its preferred size. To align many components, it is sometimes necessary to have `Panel`s within `Panel`s (within `Panel`s). Sometimes it is easier to turn off the Layout Manager altogether, and position and size components manually. Note Java contains more sophisticated layout managers then the ones we cover in this course, and you can create a custom layout manager.

Best Way To Layout

Learning how to layout components is really a matter of practice. There is no one "right answer" on how to position components. The best approach is probably to draw rough sketches of what the screen(s) should look like. Try several different layouts and pick the most pleasing (and functional).

Even with a specific appearance in mind there is no one best way to actually achieve that. You can achieve very similar appearances using `Box`es, `BorderLayout`, or other layout managers, and then `Panel`s within `Panel`s.

In-class program: Scientific Personality quiz. Show solution, and development solution where every `Panel` has a different `Color`.

`GridLayout` `(rows, cols)` or `(rows, cols, hGap, vGap)`

Either `rows` or `cols` may be zero; the layout manager will calculate the missing value from the other and the number of components added. The container's area is divided into a grid of identically sized cells. Components added to the container fill the grid from top to bottom, left to right. **Show Tic-Tac-Toe**.

`GridBag`

This is the single most powerful layout manager, and it is available in Java 1.1. However it is very complex and won't be discussed in detail in this course. A container with `GridBag` layout is divided into cells laid out into rows and columns, like graph paper. When adding any `Component`, you specify which cells the component will use and where within that block the component is to be drawn. There are so many parameters to set that you must use a helper object, called a `GridBagConstraint`. First you set the fields in the `GidBagConstraint` object, than add the component using that object.

> The model solution for Project 3 (**TxtCrypt.java**) uses a `GridBag` layout manager, so don't try to reproduce this layout exactly.

`CardLayout` (Legacy layout manager for AWT; use `JTabbedPanel` instead)

With this layout manager, each component added is drawn the full size of the container. Only one component is visible at a time however. The `CardLayout` layout manager object has methods to allow you to show the first, last, previous, or next component. When adding components, you can use a name for the constraint. Then you can show a specific component by requesting the layout manager show that component.

Unlike other layout managers, this layout manger requires your code to invoke methods on it later, so the correct way to use it is:

```
CardLayout cl = new CardLayout();
setLayout( cl );
...
cl.next();
```

`Box` **(Part of Swing)**

This is a very useful layout manager! It lays out components in a single row or column. Actually, `Box` is a container that uses the `BoxLayoutManager`). You can specify how adjacent components are to be spaced. To easily create complex layout effects, you can create `Box`es of `Box`es. (Imagine two *row* `Box`es of components, added to a *column* `Box`.)

`SpringLayout` (Part of Swing)

A very flexible layout manager, designed to be used by GUI tools such as NetBeans. It would be very difficult to use directly (but not impossible).

`GroupLayout` (Part of Swing)

Like SpringLayout, GroupLayout was designed for use by GUI builder tools. However it is much simpler to use by humans and can be very powerful too. This layout supports complex hierarchical (nested) panes.

Other Components

Add a **`Label`**. (**Show UIDemo. Show DrawIt.**)

Note swing **`JLables`** can be text, icons (images), or both. All `Frame`s and `Window`s are ***`Container`***s, which are specialized **`Components`** that contain other `Components`. In addition to `Frame`s, `Window`s, and `DialogWindow`s, other containers are **`Panels`**. All container components support a method **`add(`*`component`*`)`** and **`add(`*`component, constraint`*`)`** to associate a component with a container. (Note some book examples show this the other way around.) Adding a `Component` to a `Container` is the only way to visually show a `Component`. In some cases all the `Component`s in a `Container` are treated as a group. For example, if a `Container` becomes invisible, all its `Component`s also become invisible.

Next add a **`TextField`**. Show API and inheritance again. Note that `TextFields` and **`TextAreas`** inherit from **`TextComponent`**. Add a `TextArea`.

Add a **`Menubar`** and a **`Menu`**. Note only `Frame`s can have menus in AWT. This is fixed in swing (any container may have a `Menubar`).

A **`Canvas`** can be extended to create a *custom heavyweight component*. In your custom component you override methods such as `paint` and `getPreferredSize` and can even handle events internally.

Show `$JAVA_HOME/demo/applets.html` (especially `swingSet*`).