

Introduction to Swing

Introduction to UIs

Before you start to learn Swing, you must address the true beginner's question: What is a UI? Well, the beginner's answer is a "user interface."

So, I'll pose the question again: What's a UI? Well, you could define it by saying it's the buttons you press, the address bar you type in, and the windows you open and close, which are all elements of a UI, but there's more to it than just things you see on the screen. The mouse, keyboard, volume of the music, colors on the screen, fonts used, and the position of an object compared to another object are all included in the UI. Basically, any object that plays a role in the interaction between the computer and the user is part of the UI. That seems simple enough, but you'd be surprised how many people and huge corporations have screwed this up over the years. In fact, there are now college majors whose sole coursework is studying this interaction.

Swing's Role

Swing is the Java platform's UI -- it acts as the software to handle all the interaction between a user and the computer. It essentially serves as the middleman between the user and the guts of the computer. How exactly does Swing do this? It provides mechanisms to handle the UI aspects described in the previous panel:

Keyboard: Swing provides a way to capture user input.

Colors: Swing provides a way to change the colors you see on the screen.

The address bar you type into: Swing provides text components that handle all the mundane tasks.

The volume of the music: Well ... Swing's not perfect.

In any case, Swing gives you all the tools you need to create your own UI.

MVC

Swing even goes a step further and puts a common design pattern on top of the basic UI principles. This design pattern is called Model-View-Controller (MVC) and seeks to "separate the roles." MVC keeps the code responsible for how something looks separate from the code to handle the data separate from the code that reacts to interaction and drives changes.

Confused? It's easier if I give you a non-technical example of this design pattern in the real world. Think about a fashion show. Consider this your UI and pretend that the clothes are the data, the computer information you present to your user. Now, imagine that this fashion show has only one person in it. This person designed the clothes, modified the clothes, and walked them down the runway all at the same time. That doesn't seem like a well-constructed or efficient design.

Now, consider this same fashion show using the MVC design pattern. Instead of one person doing everything, the roles are divided up. The fashion models (not to be confused with the model in the acronym

MVC of course) present the clothes. They act as the view. They know the proper way to display the clothes (data), but have no knowledge at all about how to create or design the clothes. On the other hand, the clothing designer works behind the scenes, making changes to the clothes as necessary. The designer acts as the controller. This person has no concept of how to walk a runway but can create and manipulate the clothes. Both the fashion models and the designer work independently with the clothes, and both have an area of expertise.

That is the concept behind the MVC design pattern: Let each aspect of the UI deal with what it's good at. If you're still confused, the examples in the rest of the document will hopefully alleviate that -- but keep the basic principle in mind as you continue: Visual components display data, and other classes manipulate it.

JComponent

The basic building block of the entire visual component library of Swing is the JComponent. It's the super class of every component. It's an abstract class, so you can't actually create a JComponent, but it contains literally hundreds of functions every component in Swing can use as a result of the class hierarchy.

JComponent is the base class not only for the Swing components but also for custom components as well.

It provides the painting infrastructure for all components -- something that comes in handy for custom components

It knows how to handle all keyboard presses. Subclasses then only need to listen for specific keys.


It contains the `add()` method that lets you add other `JComponents`. Looking at this another way, you can seemingly add any Swing component to any other Swing component to build nested components (for example, a `JPanel` containing a `JButton`, or even weirder combinations such as a `JMenu` containing a `JButton`).

Simple Swing Widgets

JLabel

The most basic component in the Swing library is the `JLabel`. It does exactly what you'd expect: It sits there and looks pretty and describes other components. The image below shows the `JLabel` in action:

The JLabel

A screenshot of a Java Swing application window. Inside the window, there is a single component, a JLabel, which is a light gray rectangular box with the text "This is a label" centered inside it in a bold, black, sans-serif font.

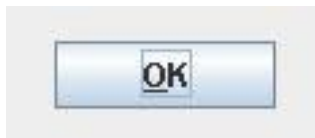
Not very exciting, but still useful. In fact, you use `JLabels` throughout applications not only as text descriptions, but also as picture descriptions. Any time you see a picture in a Swing application, chances are it's a `JLabel`. `JLabel` doesn't have many methods for a Swing beginner outside of what you might expect. The basic methods involve setting the text, image, alignment, and other components the label describes:

- **get/setText():** Gets/sets the text in the label.
- **get/setIcon():** Gets/sets the image in the label.
- **get/setHorizontalAlignment():** Gets/sets the horizontal position of the text.
- **get/setVerticalAlignment():** Gets/sets the vertical position of the text.
- **get/setDisplayedMnemonic():** Gets/sets the mnemonic (the underlined character) for the label.
- **get/setLabelFor():** Gets/sets the component this label is attached to; so when a user presses Alt+mnemonic, the focus goes to the specified component.

JButton

The basic action component in Swing, a JButton, is the push button you see with the OK and Cancel in every window; it does exactly what you expect a button to do -- you click it and something happens. What exactly happens? Well, you have to define that (see [Events](#) for more information). A JButton in action looks like this:

The JButton



The methods you use to change the JButton properties are similar to the JLabel methods (and you'll find they're similar across most Swing components). They control the text, the images, and the orientation:

- **get/setText():** Gets/sets the text in the button.
- **get/setIcon():** Gets/sets the image in the button.
- **get/setHorizontalAlignment():** Gets/sets the horizontal position of the text.
- **get/setVerticalAlignment():** Gets/sets the vertical position of the text.
- **get/setDisplayedMnemonic():** Gets/sets the mnemonic (the underlined character) that when combined with the Alt button, causes the button to click.

In addition to these methods, I'll introduce another group of methods the JButton contains. These methods take advantage of all the different states of a button. A state is a property that describes a component, usually in a true/false setting. In the case of a JButton, it contains the following possible states: active/inactive, selected/not selected, mouse-over/mouse-off, pressed/unpressed. In addition, you can combine states, so that, for example, a button can be selected with a mouse-over. Now you might be asking yourself what the heck you're supposed to do with all these states. As an example, go up to the Back button on your browser. Notice how the image changes when you mouse over it, and how it changes when you press it. This button takes advantage of the various states. Using different images with each state is a popular and effective way to indicate to a user that interaction is taking place. The state methods on a JButton are:

- `get/setDisabledIcon()`
- `get/setDisabledSelectedIcon()`
- `get/setIcon()`
- `get/setPressedIcon()`
- `get/setRolloverIcon()`
- `get/setRolloverSelectedIcon()`
- `get/setSelectedIcon()`

JTextField

The basic text component in Swing is the `JTextField`, and it allows a user to enter text into the UI. I'm sure you're familiar with a text field. You enter text, delete text, highlight text, and move the caret around -- and Swing takes care of all of that for you. As a UI developer, there's really little you need to do to take advantage of the `JTextField`.

In any case, this what a `JTextField` looks like in action:

The JTextField

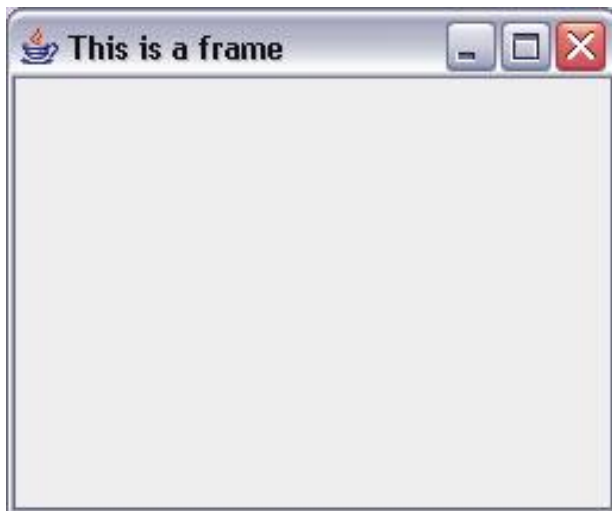


You need to concern yourself with only one method when you deal with a `JTextField` -- and that should be obvious -- the one that sets the text: `getText()`, which gets/sets the text inside the `JTextField`.

JFrame

So far I've talked about three basic building blocks of Swing, the label, button, and text field; but now you need somewhere to put them. They can't just float around on the screen, hoping the user knows how to deal with them. The JFrame class does just that -- it's a container that lets you add other components to it in order to organize them and present them to the user. It contains many other bonuses, but I think it's easiest to see a picture of it first:

The JFrame



A JFrame actually does more than let you place components on it and present it to the user. For all its apparent simplicity, it's actually one of the most complex components in the Swing packages. To greatly simplify why, the JFrame acts as bridge between the OS-independent Swing parts and the actual OS it runs on. The JFrame registers as a window in the native OS and by doing so gets many of the familiar OS window features: minimize/maximize, resizing, and movement. For the

purpose of this introductory document though, it is quite enough to think of the JFrame as the palette you place the components on. Some of the methods you can call on a JFrame to change its properties are:

- **getTitle/setTitle():** Gets/sets the title of the frame.
- **getState/setState():** Gets/sets the frame to be minimized, maximized, etc.
- **isVisible/setVisible():** Gets/sets the frame to be visible, in other words, appear on the screen.
- **getLocation/setLocation():** Gets/sets the location on the screen where the frame should appear.
- **getSize/setSize():** Gets/sets the size of the frame.
- **add():** Adds components to the frame.

A Simple Application

Like all "Introduction to x" tutorials, this one has the requisite HelloWorld demonstration. This example, however, is useful not only to see how a Swing app works but also to ensure your setup is correct. Once you get this simple app to work, every example after this one will work as well. The image below shows the completed example:

HelloWorld Example



Your first step is to create the class. A Swing application that places components on a JFrame needs to subclass the JFrameclass, like this:

```
public class HelloWorld extends JFrame
```

By doing this, you get all the JFrame properties outlined above, most importantly native OS support for the window. The next step is to place the components on the screen. In this example, you use a null layout. You will learn more about layouts and layout managers later. For this example though, the numbers indicate the pixel position on the JFrame:

```
public HelloWorld()  
{  
    super();  
    this.setSize(300, 200);  
    this.getContentPane().setLayout(null);  
    this.add(getJLabel(), null);  
    this.add(getJTextField(), null);  
    this.add(getJButton(), null);  
    this.setTitle("HelloWorld");  
}
```

```
private javax.swing.JLabel getJLabel() {  
    if(jLabel == null) {  
        jLabel = new javax.swing.JLabel();  
        jLabel.setBounds(34, 49, 53, 18);  
        jLabel.setText("Name:");  
    }  
    return jLabel;  
}
```

```
private javax.swing.JTextField getJTextField() {  
    if(jTextField == null) {  
        jTextField = new javax.swing.JTextField();  
        jTextField.setBounds(96, 49, 160, 20);  
    }  
    return jTextField;  
}
```

```
private javax.swing.JButton getJButton() {  
    if(jButton == null) {  
        jButton = new javax.swing.JButton();  
        jButton.setBounds(103, 110, 71, 27);  
        jButton.setText("OK");  
    }  
    return jButton;  
}
```

Now that the components are laid out on the JFrame, you need the JFrame to show up on the screen and make your application runnable. As in all Java applications, you must add a main method to make a

Swing application runnable. Inside this main method, you simply need to create your HelloWorld application object and then call setVisible() on it:

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

And you're done! That's all there is to creating the application.

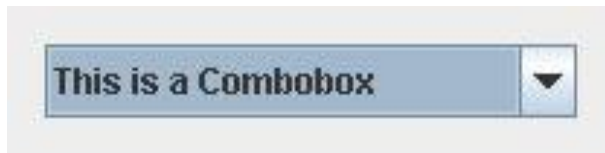
Additional Swing Widgets

JComboBox

In this section we'll cover all the other components in the Swing library, how to use them, and what they look like, which should give you a better idea of the power Swing gives you as a UI developer.

We'll start with the JComboBox. A combo box is the familiar drop-down selection, where users can either select none or one (and only one) item from the list. In some versions of the combo box, you can type in your own choice. A good example is the address bar in your browser; that is a combo box that lets you type in your own choice. Here's what the JComboBox looks like in Swing:

The JComboBox



The important functions with a JComboBox involve the data it contains. You need a way to set the data in the JComboBox, change it, and get the users' choice once they've made a selection. You can use the following JComboBox methods:

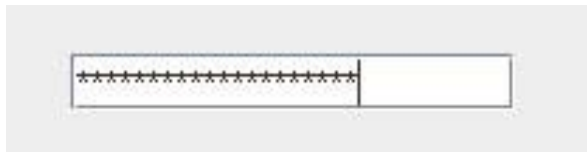
- **addItem():** Adds an item to the JComboBox.
- **get/setSelectedIndex():** Gets/sets the index of the selected item in JComboBox.

- **get/setSelectedItem():** Gets/sets the selected object.
- **removeAllItems():** Removes all the objects from the JComboBox.
- **removeItem():** Removes a specific object from the JComboBox.

JPasswordField

A slight variation on the JTextField is the JPasswordField, which lets you hide all the characters displayed in the text field area. After all, what good is a password everyone can read as you type it in? Probably not very good one at all, and in this day and age when your private data is susceptible, you need all the help you can get. Here's how a JPasswordField looks in Swing:

The JPasswordField



The additional "security" methods on a JPasswordField change the behavior of a JTextField slightly so you can't read the text:

- **get/setEchoChar():** Gets/sets the character that appears in the JPasswordField every time a character is entered. The "echo" is not returned when you get the password; the actual character is returned instead.

- **getText():** You should *not* use this function, as it poses possible security problems (for those interested, the String would be kept in memory, and a possible heap dump could reveal the password).
- **getPassword():** This is the proper method to get the password from the JPasswordField, as it returns a char[] containing the password. To ensure proper security, the array should be cleared to 0 to ensure it does not remain in memory.

JCheckBox/JRadioButton

The JCheckBox and JRadioButton components present options to a user, usually in a multiple-choice format. What's the difference? From a practical standpoint, they aren't that different. They behave in the same way. However, in common UI practices, they have a subtle difference: JRadioButtons are usually grouped together to present to the user a question with a mandatory answer, and these answers are exclusive (meaning there can be only one answer to the question). The JRadioButton's behavior enforces this use. Once you select a JRadioButton, you cannot deselect it unless you select another radio button in the group. This, in effect, makes the choices unique and mandatory. The JCheckBox differs by letting you select/deselect at random, and allowing you to select multiple answers to the question.

Here's an example. The question "Are you a guy or a girl?" leads to two unique answer choices "Guy" or "Girl." The user must select one and cannot select both. On the other hand, the question "What are your hobbies?" with the answers "Running," "Sleeping," or "Reading" should not allow only one answer, because people can have more than one hobby.

The class that ties groups of these JCheckBoxes or JRadioButtons together is the ButtonGroup class. It allows you to group choices together (such as "Guy" and "Girl") so that when one is selected, the other one is automatically deselected.

Here's what JCheckBox and JRadioButton look like in Swing:

JCheckBox and JRadioButton



The important ButtonGroup methods to remember are:

- **add():** Adds a JCheckBox or JRadioButton to the ButtonGroup.
- **getElements():** Gets all the components in the ButtonGroup, allowing you to iterate through them to find the one selected.

JMenu/JMenuItem/JMenuBar

The JMenu, JMenuItem, and JMenuBar components are the main building blocks to developing the menu system on your JFrame. The base of any menu system is the JMenuBar. It's plain and boring, but it's required because every JMenu and JMenuItem builds off it. You use the setJMenuBar() method to attach the JMenuBar to the JFrame. Once it's anchored onto the JFrame, you can add all the menus, submenus, and menu items you want.

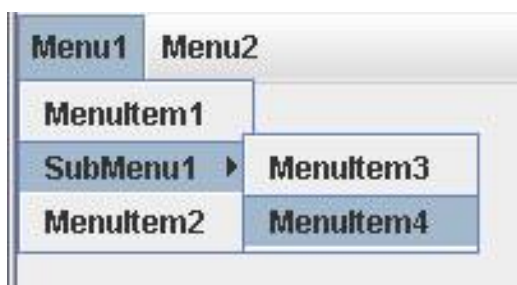
The JMenu/JMenuItem difference might seem obvious, but is in fact underneath the covers and isn't what it appears to be. If you look at the class hierarchy, JMenu is a subclass of JMenuItem. However, on the surface, they have a difference: You use JMenu to contain other JMenuItem and JMenus; JMenuItem, when chosen, trigger actions.

The JMenuItem also supports the notion of a shortcut key. As in most applications you've used, Swing applications allow you to press Ctrl+(a key) to trigger an action as if the menu item itself was selected. Think of the Ctrl+X and Ctrl+V you use to cut and paste.

In addition, both JMenu and JMenuItem support mnemonics. You use the Alt key in association with a letter to mimic the selection of the menu itself (for example, pressing Alt+F then Alt+x closes an application in Windows).

Here's what a JMenuBar with JMenus and JMenuItem looks like in Swing:

JMenuBar, JMenu, and JMenuItem



The important methods you need for these classes are:

- **JMenuItem and JMenu:**
 - **get/setAccelerator():** Gets/sets the Ctrl+key you use for shortcuts.

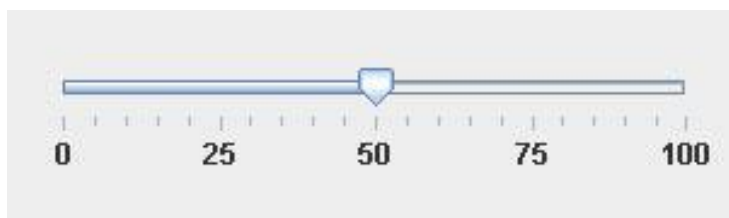
- **get/setText():** Gets/sets the text for the menu.
- **get/setIcon():** Gets/sets the image used in the menu.
- **JMenu only:**
 - **add():** adds another JMenu or JMenuItem to the JMenu (creating a nested menu).

JSlider

You use the JSlider in applications to allow for a change in a numerical value. It's a quick and easy way to let users visually get feedback on not only their current choice, but also their range of acceptable values. If you think about it, you could provide a text field and allow your user to enter a value, but then you'd have the added hassle of ensuring that the value is a number and also that it fits in the required numerical range. As an example, if you have a financial Web site, and it asks what percent you'd like to invest in stocks, you'd have to check the values typed into a text field to ensure they are numbers and are between 0 and 100. If you use a JSlider instead, you are guaranteed that the selection is a number within the required range.

In Swing, a JSlider looks like this:

The JSlider



The important methods in a JSlider are:

- **get/setMinimum():** Gets/sets the minimum value you can select.
- **get/setMaximum():** Gets/sets the maximum value you can select.
- **get/setOrientation():** Gets/sets the JSlider to be an up/down or left/right slider.
- **get/setValue():** Gets/sets the initial value of the JSlider.

JSpinner

Much like the JSlider, you can use the JSpinner to allow a user to select an integer value. One major advantage of the JSlider is its compact space compared to the JSlider. Its disadvantage, though, is that you cannot easily set its bounds.

However, the comparison between the two components ends there. The JSpinner is much more flexible and can be used to choose between any group of values. Besides choosing between numbers, it can be used to choose between dates, names, colors, anything. This makes the JSpinner extremely powerful by allowing you to provide a component that contains only predetermined choices. In this way, it is similar to JComboBox, although their use shouldn't be interchanged. You should use a JSpinner only for logically consecutive choices -- numbers and dates being the most logical choices. A JComboBox, on the other hand, is a better choice to present seemingly random choices that have no connection between one choice and the next.

A JSpinner looks like this:

The JSpinner



The important methods are:

- **get/setValue():** Gets/sets the initial value of the JSpinner, which in the basic instance, needs to be an integer.
- **getNextValue():** Gets the next value that will be selected after pressing the up-arrow button.
- **getPreviousValue():** Gets the previous value that will be selected after pressing the down-arrow button.

JToolBar

The JToolBar acts as the palette for other components (JButtons, JComboBoxes, etc.) that together form the toolbars you are familiar with in most applications. The toolbar allows a program to place commonly used commands in a quick-to-find location, and group them together in groups of common commands. Often times, but not always, the toolbar buttons have a matching command in the menu bar. Although this is not required, it has become common practice and you should attempt to do that as well.

The JToolBar also offers another function you have seen in other toolbars, the ability to "float" (that is, become a separate frame on top of the main frame).

The image below shows a non-floating JToolBar:

Non-floating JToolBar



The important method to remember with a JToolBar is: `is/setFloatable()`, which gets/sets whether the JToolBar can float.

JToolTip

You've probably seen JToolTips everywhere but never knew what they were called. They're kind of like the plastic parts at the end of your shoelaces -- they're everywhere, but you don't know the proper name (they're called *aglets*, in case you're wondering). JToolTips are the little "bubbles" that pop up when you hold your mouse over something.

They can be quite useful in applications, providing help for difficult-to-use items, extending information, or even showing the complete text of an item in a crowded UI. They are triggered in Swing by leaving the mouse over a component for a set amount of time; they usually appear about a second after the mouse becomes inactive. They stay visible as long as the mouse remains over that component.

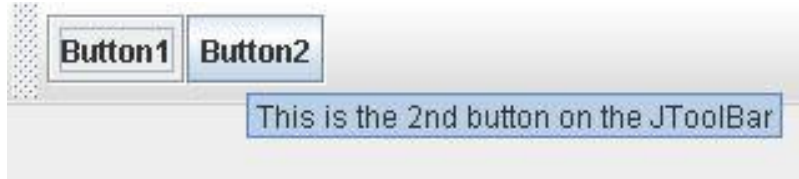
The great part about the JToolTip is its ease of use.

The `setToolTip()` method is a method in the `JComponent` class, meaning every Swing component can have a tool tip associated with it. Although the JToolTip is a Swing class itself, it really provides no additional functionality for your needs at this time, and shouldn't be created itself.

You can access and use it by calling the `setToolTip()` function in `JComponent`.

Here's what a `JToolTip` looks like:

A `JToolTip`

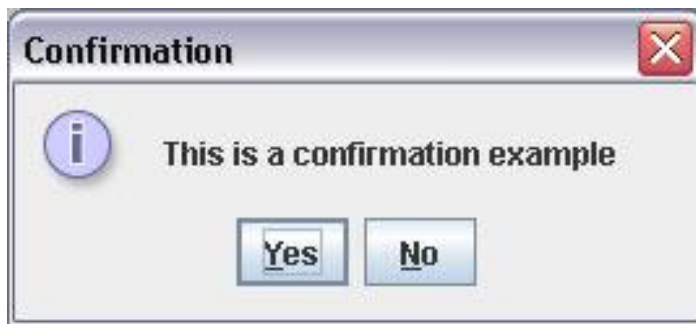


`JOptionPane`

The `JOptionPane` is something of a "shortcut" class in Swing. Often times as a UI developer you'd like to present a quick message to your users, letting them know about an error or some information. You might even be trying to get some quick data, such as a name or a number. In Swing, the `JOptionPane` class provides a shortcut for these rather mundane tasks. Rather than make every developer recreate the wheel, Swing has provided this basic but useful class to give UI developers an easy way to get and receive simple messages.

Here's a `JOptionPane`:

A JOptionPane



The somewhat tricky part of working with JOptionPane is all the possible options you can use. While simple, it still provides numerous options that can cause confusion. One of the best ways to learn JOptionPane is to play around with it; code it and see what pops up. The component lets you change nearly every aspect of it: the title of the frame, the message itself, the icon displayed, the button choices, and whether or not a text response is necessary. There are far too many possibilities to list and your best bet is to visit the JOptionPane API page to see its many possibilities.

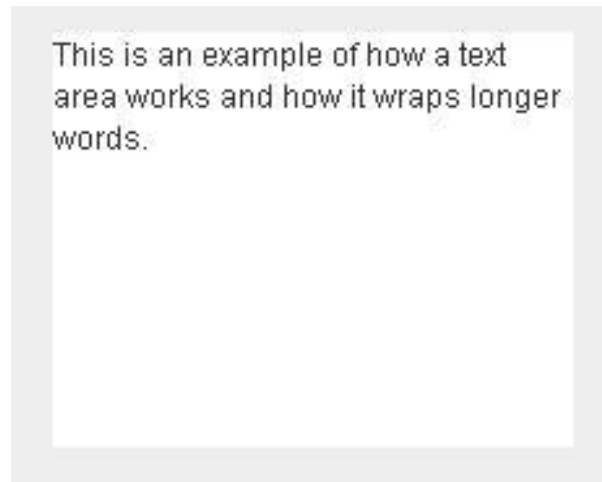
JTextArea

The JTextArea takes the JTextField a step further. While the JTextField is limited to one line of text, the JTextArea extends that capability by allowing for multiple rows of text. Think of it as an empty page allowing you to type anywhere in it. As you can probably guess, the JTextArea contains many of the same functions as the JTextField; after all, they are practically the exact same component. However, the JTextArea offers a few additional important functions that set it apart. These features include the ability to word wrap (that is, wrap a long word to the next line instead of cutting it off mid-word) and the ability to wrap

the text (that is, move long lines of text to the next line instead of creating a very long line that would require a horizontal scroll bar).

A JTextArea in Swing looks like you'd probably expect:

A JTextArea



The important methods to enable line wrapping and word wrapping are:

- **is/setLineWrap():** Sets whether the line should wrap when it gets too long.
- **is/setWrapStyleWord():** Sets whether a long word should be moved to the next line when it is too long.

JScrollPane

Building off of the example above, suppose that the JTextArea contains too much text to contain in the given space. Then what? If you think that scroll bars will appear automatically, unfortunately, you are wrong. The JScrollPane fills that gap, though, providing a Swing component to handle all scroll bar-related actions. So while it might be a slight pain to

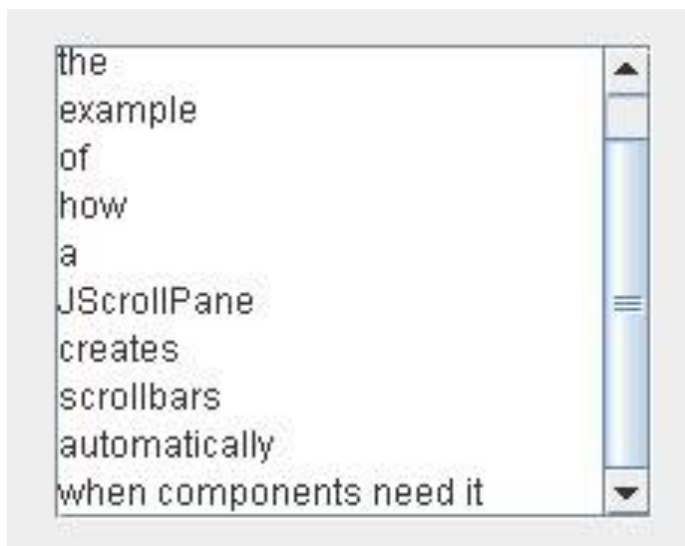
provide a scroll pane for every component that could need it, once you add it, it handles everything automatically, including hiding/showing the scroll bars when needed.

You don't have to deal with the JScrollPane directly, outside of creating it using the component to be wrapped. Building off the above example, by calling the JScrollPane constructor with the JTextArea, you create the ability for the JTextArea to scroll when the text gets too long:

```
JScrollPane scroll = new JScrollPane(getTextArea());  
  
add(scroll);
```

This updated example looks like this:

JScrollPane Example



The JScrollPane also exposes the two JScrollBars it will create. These JScrollBar components also contain methods you can use to change their behavior.

The methods you need to work with JScrollPane are:

- **getHorizontalScrollBar():** Returns the horizontal JScrollBar component.
- **getVerticalScrollBar():** Returns the vertical JScrollBar component.
- **get/setHorizontalScrollBarPolicy():** This "policy" can be one of three things: Always, Never, or As Needed.
- **get/setVerticalScrollBarPolicy():** The same as the horizontal function.

JList

The JList is a useful component for presenting many choices to a user. You can think of it as an extension to the JComboBox. JList provides more choices and adds the capability for multiple choices. The choice between a JList and JComboBox often comes down to these two features: If you require multiple choices or if the options include more than 15 choices (although that number is not a general rule), you should always choose a JList.

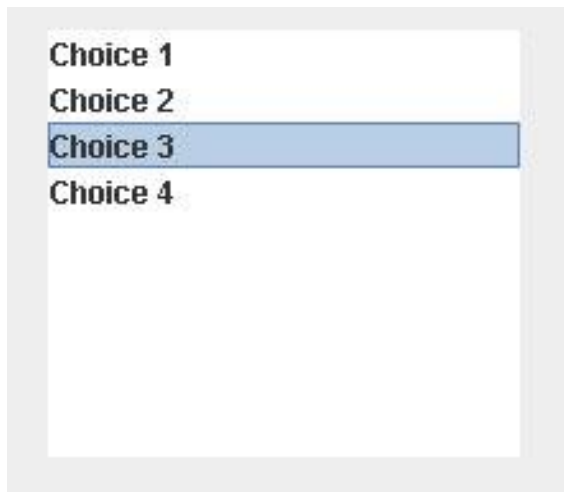
You should use the JList in conjunction with the JScrollPane, as demonstrated above, because it can present more options than its space can contain.

The JList contains the notion of a selection model (also seen in JTables), where you can set your JList to accept different types of choices. These types are the single selection, where you can only select one choice, the single interval selection, where you can only select contiguous choices, but as many as desired, or the multiple interval selection, where you can select any number of choices in any combination.

The JList is the first of what I call the "complex components," which also include the JTable and the JTree that allow a large amount of custom changes, including changing the way the UI looks and how it deals with data. Because this introductory document only strives to cover the basics, I won't get into these more advanced functions, but it's something to remember when working with these components -- they present a more difficult challenge than those components I've introduced up to this point.

The JList appears like this in Swing:

The JList



There are many functions in the JList to deal with the data, and as I said, these just touch the surface of everything required to work in detail with the JList. Here are the basic methods:

- **get/setSelectedIndex():** Gets/sets the selected row of the list; in the case of multiple-selection lists, an int[] is returned.

- **get/setSelectionMode():** As explained above, gets/sets the selection mode to be either single, single interval, or multiple interval.
- **setListData():** Sets the data to be used in the JList.
- **get/setSelectedValue():** Gets the selected object (as opposed to the selected row number).

JTable

Think of an Excel spreadsheet when you think of a JTable and that should give you a clear picture of what the JTable does in Swing. It shares many of the same characteristics: cells, rows, columns, moving columns, and hiding columns. The JTable takes the idea of a JList a step further. Instead of displaying data in one column, it displays it in multiple columns. Let's use a person as an example. A JList would only be able to display one property of a person -- his or her name for instance. A JTable, however, would be able to display multiple properties -- a name, an age, an address, etc. The JTable is the Swing component that allows you to provide the most information about your data.

Unfortunately, as a trade-off, it is also notoriously the most difficult Swing component to tackle. Many UI developers have gotten headaches trying to learn every detail of a JTable. I hope to save you from that here, and just get the ball rolling with your JTable knowledge.

Many of the same concepts in JLists extend to JTables as well, including the idea of different selection intervals, for example. But the one-row idea of a JList changes to the cell structure of a JTable. This means you have different ways to make these selections in JTables, as columns,

rows, or individual cells.

In Swing, a JTable looks like this:

A JTable

A	B	C	D	E
0x0	0x1	0x2	0x3	0x4
1x0	1x1	1x2	1x3	1x4
2x0	2x1	2x2	2x3	2x4
3x0	3x1	3x2	3x3	3x4
4x0	4x1	4x2	4x3	4x4

JTree

The JTree is another complex component that is not as difficult to use as the JTable but isn't as easy as the JList either. The tricky part of working with the JTree is the required data models.

The JTree takes its functionality from the concept of a tree with branches and leaves. You might be familiar with this concept from working with Internet Explorer in Windows -- you can expand and collapse a branch to show the different leaves you can select and deselect.

You will most likely find that the tree is not as useful in an application as a table or a list, so there aren't as many helpful examples on the Internet. In fact, like the JTable, the JTree does not have any beginner-level functions. If you decide to use JTree, you will immediately be at the intermediate level and must learn the concepts that go with it. However, there are times when a tree is the logical UI component for your needs. File/directory systems are one example, as in Internet Explorer, and the JTree is the best component in the case where data takes a hierarchical structure -- in other words, when the data is in the form of a tree.

In Swing, a JTree looks like this:

A JTree



Swing concepts

Layouts, Models and Events

Now that you know about most (but definitely not all) of the components you can use to make a UI, you have to actually do something with them. You can't just place them randomly on the screen and expect them to instantly work. You must place them in specific spots, react to interaction with them, update them based on this interaction, and populate them with data. More learning is necessary to fill in the gaps of your UI knowledge with the other important parts of a UI.

Therefore, let's examine:

- **Layouts:** Swing includes a lot of layouts, which are classes that handle where a component is placed on the application and what should happen to them when the application is resized or components are deleted or added.
- **Events:** You need to respond to the button presses, the mouse clicks, and everything else a user can do to a UI. Think about what would happen if you didn't -- users would click and nothing would change.
- **Models:** For the more advanced components (Lists, Tables, Trees), and even some easier ones such as the JComboBox, models are the most efficient way to deal with the data. They remove most of the work of handling the data from the actual component itself (think back to the earlier MVC discussion) and provide a wrapper for common data object classes (such as Vector and ArrayList).

Easy Layouts

As mentioned in the last section, a layout handles the placement of components on the application for you. Your first question might be "why can't I just tell it where to go by using pixels?" Well, you can, but then you'd immediately be in trouble if the window was resized, or worse, when users changed their screen resolutions, or even when someone tried it on another OS. Layout managers take all those worries away. Not everyone uses the same settings, so layout managers work to create "relative" layouts, letting you specify how things should get resized relative to how the other components are laid out. Here's the good part: it's easier than it sounds. You simply call `setLayout(yourLayout)` to set up the layout manager. Subsequent calls to `add()` add the component to the container and let the layout manager take care of placing it where it belongs.

Numerous layouts are included in Swing nowadays; it seems there's a new one every release that serves another purpose. However, some tried-and-true layouts that have been around forever, and by forever, I mean forever -- since the first release of the Java language back in 1995. These layouts are the `FlowLayout`, `GridLayout`, and `BorderLayout`.

The `FlowLayout` lays out components from left to right. When it runs out of space, it moves down to the next line. It is the simplest layout to use, and conversely, also the least powerful layout:

```
setLayout(new FlowLayout());  
  
add(new JButton("Button1"));  
  
add(new JButton("Button2"));
```

```
add(new JButton("Button3"));
```

The FlowLayout at work



The GridLayout does exactly what you'd think: it lets you specify the number of rows and the number of columns and then places components in these cells as they are added:

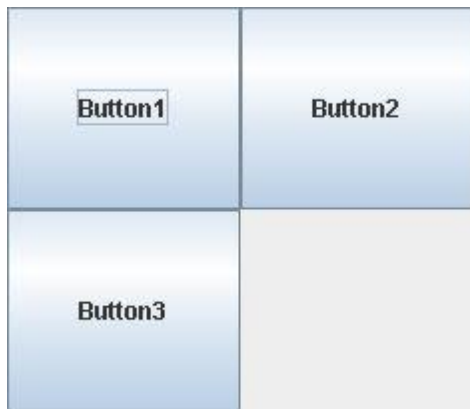
```
setLayout(new GridLayout(1,2));
```

```
add(new JButton("Button1"));
```

```
add(new JButton("Button2"));
```

```
add(new JButton("Button3"));
```

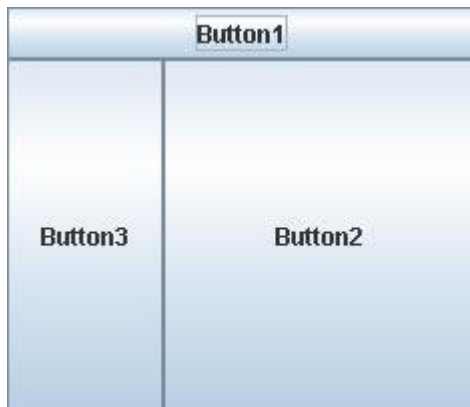
The GridLayout at work



The BorderLayout is still a very useful layout manager, even with all the other new ones added to Swing. Even experienced UI developers use the BorderLayout often. It uses the notions of North, South, East, West, and Center to place components on the screen:

```
setLayout(new BorderLayout());  
add(new JButton("Button1"), "North");  
add(new JButton("Button2"), "Center");  
add(new JButton("Button3"), "West");
```

The BorderLayout at work



GridBagLayout

While the examples from above are good for easy layouts, more advanced UIs need a more advanced layout manager. That's where the GridBagLayout comes into play. Unfortunately, it is extremely confusing and difficult to work with, and anyone who has worked with it will agree. I can't disagree either; but despite its difficulties, it's probably the best way to create a clean-looking UI with the layout managers built into Swing.

Here's my first bit of advice: In the newest versions of Eclipse, there's a built-in visual builder that automatically generates the required GridBagLayout code needed for each screen. Use it! It will save countless hours of fiddling around with the numbers to make it just right. So while I could use this slide to explain how GridBagLayout works and how to tweak it to make it work best, I'll just offer my advice to find a visual builder and generate the code. It will you save hours of work.

Events

Finally, we get to one of the most important parts of Swing: dealing with events and reacting to interaction with the UI. Swing handles events by using the event/listener model. This model works by allowing certain classes to register for events from a component. This class that registers for events is called a listener, because it waits for events to occur from the component and then takes an action when that happens. The component itself knows how to "fire" events (that is, it knows the types of interaction it can generate and how to let the listeners know when that interaction happens). It communicates this interaction with events, classes that contain information about the interaction.

With the technical babble aside, let's look at some examples of how events work in Swing. I'll start with the simplest example, a JButton and printing out "Hello" on the console when it is pressed.

The JButton knows when it is pressed; this is handled internally, and there's no code needed to handle that. However, the listener needs to register to receive that event from the JButton so you can print out "Hello." The listener class does this by implementing the listener interface and then calling `addActionListener()` on the JButton:

```
// Create the JButton  
  
JButton b = new JButton("Button");  
  
// Register as a listener  
  
b.addActionListener(new HelloListener());
```

```
class HelloListener implements ActionListener
{
    // The interface method to receive button clicks
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("Hello");
    }
}
```

A JList works in a similar way. When someone selects something on a JList, you want to print out what object is selected to the console:

```
// myList is a JList populate with data
myList.addListSelectionListener(new ListSelectionListener()
{
    public void valueChanged(ListSelectionEvent e)
    {
        Object o = myList.getSelectedItemAt();
        System.out.println(o.toString());
    }
}
```

);

From these two examples, you should be able to understand how the event/listener model works in Swing. In fact, every interaction in Swing is handled this way, so by understanding this model, you can instantly understand how every event is handled in Swing and react to any possible interaction a user might throw at you.

Models

Now, you should know about the Java Collections, a set of Java classes that handle data. These classes include an ArrayList, a HashMap, and a Set. Most applications use these classes ubiquitously as they shuttle data back and forth. However, one limitation arises when you need to use these data classes in a UI. A UI doesn't know how to display them. Think about it for a minute. If you have a JList and an ArrayList of some data object (such as a Person object), how does the JList know what to display? Does it display the first name or both the first name and the last name?

That's where the idea of a model comes in. While the term model refers to the larger scope, in this tutorial's examples I use the term UI model to describe the classes that components use to display data.

Every component that deals with a collection of data in Swing uses the concept of a model, and it is the preferable way to use and manipulate data. It clearly separates the UI work from the underlying data (think back to the MVC example). The model works by describing to the component how to display the collection of data. What do I mean by describing? Each component requires a slightly different description:

JComboBox requires its model to tell it what text to display as a choice and how many choices exist.

JSpinner requires its model to tell it what text to display, and also what the previous and next choices are.

JList also requires its model to tell it what text to display as a choice and how many choices exist.

JTable requires much more: It requires the model to tell it how many columns and rows exist, the column names, the class of each column, and what text to display in each cell.

JTree requires its model to tell it the root node, the parents, and the children for the entire tree.

Why do all this work, you might ask. Why do you need to separate all this functionality? Imagine this scenario: You have a complicated JTable with many columns of data, and you use this table in many different screens. If you suddenly decide to get rid of one of the columns, what would be easier, changing the code in every single JTable instance you used, or changing it in one model class you created to use with every JTable instance? Obviously, changing fewer classes is better.