Intro to Java (Page 7)

JUMP June 2019

Draft: 07/19/2019

Table of Contents

Swing

Swing – Intro

- For the longest of time, Java made it possible to create Graphical User Interface (GUI) apps in 3 ways.
    - ~~**Applets**~~
    - **Swing**
    - **Java FX**
- ~~Applets~~ have been deprecated in Java since version JDK 9. Many browsers no longer support the execution of Java applets, thus the reason for why this is no longer available.

Swing – Intro

- All of the programs writing in Java in this training session have been using the console as output.
- This means that they do not make use of a graphical user interface (GUI).
- Console-based programs are excellent for teaching the basics of Java and for some types of programs, such as server-side code, but most real-world applications will be GUI-based. At the time of this writing, the most widely used Java GUI is Swing.
- Swing defines a collection of classes and interfaces that support a rich set of visual components, such as buttons, text fields, scroll panes, check boxes, trees, and tables, to name a few.
- Collectively, these controls can be used to construct powerful, yet easy-to-use graphical interfaces.
- Swing is something with which all Java programmers should be familiar as it is still extremely popular.

Swing – Intro

- Swing is a very large topic that requires an entire book of its own.
- We only cover the basics.
- The material presented here will give you a general understanding of Swing, including its history, basic concepts, and design philosophy.
- We introduce a few of commonly used Swing components:
    - label
    - push button
    - text field
    - check box
    - list
    - grid (via gridlayout)
- At the end of this session you will be able to begin writing simple GUI-based programs.
- You will also have a foundation upon which to continue your study of Swing.

Swing – Intro

- Swing did not exist when Java was first released in 01/1996.
- It was introduced in 04/1997 as part of the Java Foundation Classes (JFC).
- Java's original GUI subsystem the Abstract Window Toolkit (AWT) is flawed as it is limited to the look and feel of an operating system.
- GUI components behavior is dictated by the OS and as such implies they will not be consistent.
- This goes against the overarching philosophy of Java which is **write once** and **run anywhere** as your GUI doesn't look identical across platforms.
- Swing addresses this issue and was initially available for use with Java 1.1 as a separate library.
- Since Java 1.2, Swing (and the rest of JFC) is fully integrated into Java.

Swing – Intro

- Swing addresses the limitations associated with the AWT’s components through the use of two key features:
  - **lightweight components and**
  - **a pluggable look and feel.**
- Largely transparent to the programmer, these two features are at the foundation of Swing’s design philosophy and the reason for much of its power and flexibility.
- With very few exceptions, Swing components are lightweight.
- This means that a component is written entirely in Java.
- They do not rely on platform-specific peers.
- Lightweight components have some important advantages, including efficiency and flexibility.
- As lightweight components do not translate into platform-specific peers, the look and feel of each component is determined by Swing, not by the underlying operating system.
- This means that each component can work in a consistent manner across all platforms.

Swing – Intro

- Swing components are rendered by Java code allowing the separation of the look and feel of a component from the logic of the component.
- The implication is that is it possible to change the way that a component is rendered without affecting any of its other aspects.
- In other words, it is possible to “plug in” a new look and feel for any given component without creating any side effects in the code that uses that component.
- Java provides look-and-feels, such as metal and Nimbus, that are available to all Swing users.
- The metal look and feel is also called the Java look and feel.
- It is a platform-independent look and feel that is available in all Java execution environments.
- It is also the default look and feel.
- Swing’s pluggable look and feel is made possible because Swing uses a **modified version** of the classic **model-view-controller (MVC)** architecture. In MVC terminology, the model corresponds to the state information associated with the component.

Swing – Intro

- An the case of a check box:
  - the **model** contains a field that indicates if the box is checked or unchecked.
  - the **view** determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model.
  - the **controller** determines how the component reacts to the user.
- When the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked).
- This results in the view being updated. By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two.
- Different view implementations can render the same component in different ways without affecting the model or the controller.

Swing – Intro

- While an MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller was not beneficial for Swing components.
- Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the **UI delegate**.
- For this reason, Swing's approach is called either the **model-delegate architecture** or the **separable model architecture**.
- Swing is based on MVC but does not use the classical implementation of it.
- Swing's ease of use is its most important advantage.
- Swing makes manageable the often difficult task of developing your program's user interface.
- This lets you concentrate on the GUI itself, rather than on implementation details.

Swing – Components and Containers

- Swing GUI consists of two key items:
    - **components**
    - **containers**
- This distinction is mostly conceptual because all containers are also components.
- The difference between the two is found in their intended purpose:
    - **a component** is an independent visual control, such as a push button or text field.
    - **a container** holds a group of components.
- A container is a special type of component that is designed to hold other components.
- In order for a component to be displayed, it must be held within a container.
- All Swing GUI components must have at least one container.
- Because containers are components, a container can also hold other containers.
- This enables Swing to define what is called a containment hierarchy, at the top of which must be a top-level container.

Swing – Components and Containers

Components

- In general, Swing components are derived from the JComponent class. *(The only exceptions to this are the four top-level containers, to be reviewed in the "Top Level Containers" slide.)*
- **JComponent** provides the functionality that is common to all components.
- For example, **JComponent** supports the pluggable look and feel.
- **JComponent** inherits the **AWT** classes **Container and Component**.
- Thus, a Swing component is built on and compatible with an AWT component.

Swing – Components and Containers

Components

- All of Swing's components are represented by classes defined within the package javax.swing.
- The following table shows the class names for Swing components (including those used as containers):

~~JApplet~~	JDialog	JLayeredPane	JPopupMenu	JSlider	JTogglebutton
JButton	JEditorPane	JList	JProgressBar	JSpinner	JToolBar
JCheckBox	JFileChooser	JMenu	JRadioButton	JSplitPane	JToolTip
JCheckBoxMenuItem	JFormattedTextField	JMenuBar	JRadioButtonMenuItem	JTabbedPane	JTree
JColorChooser	JFrame	JMenuItem	JRootPane	JTable	JViewport
JComboBox	JInternalFrame	JOptionPane	JScrollBar	JTextArea	JWindow
JComponent	JLabel	JPanel	JScrollPane	JTextField	
JDesktopPane	JLayer	JPasswordField	JSeparator	JTextPane	

Note: all Swing classes start with the letter "J" – Swing ~~JApplet~~ are Applets which are deprecated (unsupported) in most browsers.

Swing – Components and Containers

Containers

Swing defines two types of containers.

- Top-Level Containers
- Lightweight Containers

Swing – Components and Containers

Top-Level Containers

- The first are top-level containers: **JFrame, ~~JApplet~~, JWindow, and JDialog.**
- These containers **do not inherit** JComponent.
- They do, however, inherit the AWT classes Component and Container.
- Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight.
- This makes the top-level containers a special case in the Swing component library.
- As the name implies, a top-level container must be at the top of a containment hierarchy.
- A top-level container is not contained within any other container.
- Furthermore, every containment hierarchy must begin with a top-level container.
- The one most commonly used for applications is JFrame.

Swing – Components and Containers

Lightweight Containers

- The second type of container supported by Swing is the lightweight container.
- Lightweight containers **do inherit** JComponent.
- Examples of lightweight containers are **JPanel, JScrollPane, and JRootPane**.
- Lightweight containers are often used to collectively organize and manage groups of related components because a lightweight container can be contained within another container.
- Thus, you can use lightweight containers to create subgroups of related controls that are contained within an outer container.

Swing – Components and Containers

Top Level Container Panes

- Each top-level container defines a set of panes.
- At the top of the hierarchy is an instance of JRootPane.
- JRootPane is a lightweight container whose purpose is to manage the other panes.
- It also helps manage the optional menu bar.
- The panes that compose the root pane are called the glass pane, the content pane, and the layered pane.
- The glass pane is the top-level pane.
- It sits above and completely covers all other panes.
- The glass pane enables you to manage mouse events that affect the entire container (rather than an individual control) or to paint over any other component, for example.

Swing – Components and Containers

Top Level Container Panes

- In most cases, you won’t need to use the glass pane directly.
- The layered pane allows components to be given a depth value.
- This value determines which component overlays another.
- (Thus, the layered pane lets you specify a Z-order for a component, although this is not something that you will usually need to do.)
- The layered pane holds the content pane and the (optional) menu bar.
- Although the glass pane and the layered panes are integral to the operation of a top-level container and serve important purposes, much of what they provide occurs behind the scene.
- The pane with which your application will interact the most is the content pane, because this is the pane to which you will add visual components. In other words, when you add a component, such as a button, to a top-level container, you will add it to the content pane.
- Therefore, the content pane holds the components that the user interacts with.
- For example, a JFrame cannot just be added layouts. We first need to **getContentPane();**

Swing – Layout Managers

- The layout manager controls the position of components within a container.
- Java offers several layout managers.
- Most are provided by the AWT (within java.awt), but Swing adds a few of its own.
- All layout managers are instances of a class that implements the LayoutManager interface.
- (Some will also implement the LayoutManager2 interface.)
- List of some of the layout managers available to the Swing programmer:

Layout Type	Description
FlowLayout	A simple layout that positions components left-to-right, top-to-bottom. (Positions components right-to-left for some cultural settings.)
BorderLayout *	Positions components within the center or the borders of the container.
GridLayout	Lays out components within a grid.
GridBagLayout	Lays out different size components within a flexible grid.
BoxLayout	Lays out components vertically or horizontally within a box.
SpringLayout	Lays out components subject to a set of constraints.

** This is the default layout manager for a content pane.*

Swing – Layout Managers

In this session we use two layout managers—**BorderLayout** and **FlowLayout**.

BorderLayout

- The default layout manager for the content pane. It implements a layout style that defines five locations to which a component can be added. The first is the **center**. The other four are the sides (i.e., borders), which are called **north, south, east, and west**.
- By default, when you add a component to the content pane, you are adding the component to the **center**. To add a component to one of the other regions, specify its name.

FlowLayout

- A flow layout lays out components one row at a time, top to bottom.
- When one row is full, layout advances to the next row.
- Although this scheme gives you little control over the placement of components, it is quite simple to use.
- Be aware that if you ***resize the frame***, the position of the components will change.

Swing – First Simple Swing Program

- Swing programs differ from the console-based programs shown earlier in this book.

- Swing programs use the Swing component set to handle user interaction, and also have special requirements that relate to threading.

- There are two types of Java programs in which Swing is typically used.

- The first is a desktop application.
- The second is the ~~applet~~.

- This section shows how to create a Swing application.

Note: We do not cover ~~Swing Applet~~ creation as they are meant to run in browsers, which no longer support these types of applications.

Swing – First Simple Swing Program

- The following short POC demonstrates the basic skeleton of a Swing application.
- It demonstrates several key features of Swing.
- Uses two Swing components: **JFrame** and **JLabel**.
    - **JFrame** is the top-level container that is commonly used for Swing applications.
    - **JLabel** is a Swing component that creates a label; a component that displays information.
- The label is Swing’s simplest component because it is passive.
  This implies that a label does not respond to user input. It only displays output.
- The program uses a JFrame container to hold an instance of a JLabel.
- The label displays a short text message.

See live code in STS “BasicSwingDemo”

Swing – First Simple Swing Program

SwingDemo.java

```
import javax.swing.*;

class SwingDemo {
   private JFrame jfrm;
   SwingDemo(String title) {
      jfrm = new JFrame(title);
   }
   public JFrame getFrame() {
      return jfrm;
   }
}
```

Main.java

```
import javax.swing.SwingUtilities;
import javax.swing.*;
public class Main {

   public static JFrame setFrameStuff(JFrame frm, int width, int height, int onExit, String label) {
      frm.setSize(width, height);
      frm.setDefaultCloseOperation(onExit);
      JLabel lab = new JLabel(label);
      frm.add(lab);
      return frm;
   }

   public static void main(String args[]) {

      SwingUtilities.invokeLater(new Runnable() {
         SwingDemo demo;
         JFrame frm;
         public void run() {
            demo = new SwingDemo("Demo 1");
            frm = demo.getFrame();
            frm = setFrameStuff(frm, 375, 100, JFrame.EXIT_ON_CLOSE, "My label");
            frm.setVisible(true);
         }
      });
   }
}
```

- Create the frame on the event dispatching thread.
- Swing uses concurrency behind the scene
- Without the ability to thread Swing and Java could not create usable applications which uses GUI resources.
- This is a standard way of ensuring our app works.
- The “invokeLater” method will ensure the thread is scheduled at some point in Java’s scheduler.
- The class Main and its public static void main are also threads but a public static void main is a implicit call the GUI is meant to work concurrently with other thread(s).
- The “invokeLater” method ensure this thread is managed in a way that behind the scene the Java developer is not burdened with dealing in GUI related operations at the OS level as opposed to using a command prompt output which is sequential in nature.
- GUIs are meant to respond to events and as such asynchronous.

See live code in STS “BasicSwingDemo”

Swing – Event Dispatching Thread

The **event dispatching thread** (EDT) is a background **thread** used in **Java** to process events from the Abstract Window Toolkit (AWT) graphical user interface **event** queue. ... Updating visible components from other **threads** is the source of many common bugs in **Java** programs that use Swing.

Event Dispatching Thread for Swing

The demo creates a SwingDemo object on the **event-dispatching thread** rather than on the main thread of the application.

The Rationale

- Swing programs are event-driven. When a user interacts with a component, an event is generated.
- An event is passed to the application by calling an event handler defined by the application.
- The handler is executed on the event-dispatching thread provided by Swing and not on the main thread of the application.
- Event handlers are defined by your program, but they are called on a thread that was not created by your program.
- To avoid problems (such as two different threads trying to update the same component at the same time), all Swing GUI components must be created and updated from the event-dispatching thread, not the main thread of the application.
- Main( ) is executed on the main thread. It **cannot directly instantiate** a SwingDemo object.
- **It must create a Runnable object** that executes on the event-dispatching thread, and have this object create the GUI.
- To enable the GUI code to be created on the event-dispatching thread, you must use one of two methods that are defined by the SwingUtilities class. These methods are invokeLater( ) and invokeAndWait( ).

  - `static void invokeLater(Runnable obj)`
  - `static void invokeAndWait(Runnable obj) throws InterruptedException, InvocationTargetException`

Swing – Event Dispatching Thread

The **event dispatching thread** (EDT) is a background **thread** used in **Java** to process events from the Abstract Window Toolkit (AWT) graphical user interface **event** queue. ... Updating visible components from other **threads** is the source of many common bugs in **Java** programs that use Swing.

Event Dispatching Thread for Swing

- **`static void invokeLater(Runnable obj)`**
- **`static void invokeAndWait(Runnable obj) throws InterruptedException, InvocationTargetException`**

Difference between invokeLater and invokeAndWait:

- **invokeLater** is used to perform a task asynchronously in AWT Event dispatcher thread. **InvokeLater** is non-blocking call
- **InvokeAndWait** is used to perform task synchronously. **InvokeAndWait** will block until the task is completed.

Best Practices

- Use **invokeLater** when you are trying to get **many operations** to **execute concurrently.**

- Use **invoteandWait** when you need a **GUI operation** to be **fully done before anything else** executes.

Swing – Second Simple Swing Program

The next POC will build on the first one where we are centering the **"JFrame"** across the user's screen. This will require the use of 2 classes both part of the **Abstract Window Toolkit (AWT)**.

- **Toolkit**: **`java.awt.Toolkit`**

  The Toolkit class is the abstract superclass of every implementation in the Abstract Window Toolkit (AWT).

  Subclasses of Toolkit are used to bind various components.

  - **getDefaultToolkit()** returns an instance of the toolkit
  - **getScreenSize()** is one of the many methods available

Dimension: `java.awt.Dimension`

- Dimension class is a part of Java AWT.
- It contains the height and width of a component in integer as well as double precision.
- The use of Dimension class is that many functions of Java AWT and Swing return dimension object.

See live code in STS "BasicSwingDemo2"

Swing – Third Simple Swing Program

The next POC will build on the previous one where we are adding a label to all 5 regions of a Border Layout (default layout)

The centered frame will be 400 pixels in width by 400 pixels in height and will be centered across the screen.

We will do this by overloading the add() method with a object location from BorderLayout class.

```
void add(Component comp, Object loc)
```

- BorderLayout.CENTER
- BorderLayout.EAST
- BorderLayout.NORTH
- BorderLayout.SOUTH
- BorderLayout.WEST

See live code in STS "BasicSwingDemo3"

Swing – Fourth Simple Swing Program

The next POC will build on the previous one we are centering all labels in their layouts both horizontally and vertically.

- **`jl.setHorizontalAlignment(int alignment);`**
- **`jl.setVerticalAlignment(int alignment);`**

Some int alignment constants:

- **`JLabel.CENTER;`**
- **`JLabel.TOP;`**
- **`JLabel.LEFT;`**
- **`JLabel.RIGHT;`**
- **`JLabel.BOTTOM;`**

See live code in STS "BasicSwingDemo4"

Swing – Swing Configuration 101

The entire concept of configuring nearly every aspect of every swing components is very simplistic:

1. Create an instance of a Swing component
2. Apply configuration via API
3. Add the component

Note: Once a component instance is available, unless otherwise stated in documentation, as long as you can refer to it programmatically, you can use the API associated with the component to modify it on-the-fly while in your app

Swing - JButton

- In this session, your first component which provides the ability to interact to user event and one of the most commonly used Swing controls is the push button.
- A push button is an instance of **`JButton`**.
- **`JButton`** inherits the abstract class **`AbstractButton`**, which defines the functionality common to all buttons.
- Swing push buttons can contain text, an image, or both, we use text in our sample code.
- JButton supplies several constructors. The one used here is:
  `JButton(String label)`
- Here, **label** specifies the string that will be displayed inside the button.
- We will also look at adding a few buttons across a few layouts in a JFrame.

Swing - JButton

- Here, `label` specifies the string that will be displayed inside the button.
- When a push button is pressed, it generates an **ActionEvent**. **ActionEvent** is defined by the AWT and also used by Swing. **JButton** provides the following methods, which are used to add or remove an action listener:
  - `void addActionListener(ActionListener al)`
  - `void removeActionListener(ActionListener al)`
- Here, **al** specifies an object that will receive event notifications.
- This object must be an instance of a class that implements the ActionListener interface.
- The ActionListener interface defines only one method: **actionPerformed( ).** It is shown here:
  - `void actionPerformed(ActionEvent ae)`
- This method is called when a button is pressed.
- In other words, it is the event handler that is called when a button press event has occurred.

Note: while GUIs can be 'multithreaded', eventually, all code must still execute in sequence within a thread, therefore as a general rule of thumb your implementation of **actionPerformed( )** must quickly respond to that event and return. Event handlers must not engage in long operations, because doing so will slow down the entire application. If a time-consuming procedure must be performed, then a separate thread should be created for that purpose.

Swing - JButton

- Using the **ActionEvent** object passed to **actionPerformed( )**, you can obtain several useful pieces of information relating to the button-press event.
- The one used in our sample code is the action command string associated with the button.
- By default, this is the string displayed inside the button.
- The action command is obtained by calling **getActionCommand( )** on the event object. It is declared like this:
  - **`String getActionCommand( )`**
- The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.
- The following sample code demonstrates how to create a push button and respond to button-press events.

See live code in STS “BasicSwingJButtonDemo”

Swing - JTextField

- JTextField enables the user to enter a line of text.
- JTextField inherits the abstract class JTextComponent, which is the superclass of all text components.
- JTextField defines several constructors. Here's a sample
  - **`JTextField(int cols)`**
- Here, cols specifies the width of the text field in columns.
- You can enter a string that is longer than the number of columns, cols is merely the physical size of the text field in your layout.
- When you press enter when inputting into a text field, an ActionEvent is generated.
- Therefore, JTextField provides the addActionListener( ) and removeActionListener( ) methods.
- To handle action events, you must implement the actionPerformed( ) method defined by the ActionListener interface.
- The process is similar to handling action events generated by a button, as described earlier.

Swing - JTextField

- Like a JButton, a JTextField has an action command string associated with it.
- By default, the action command is the current content of the text field.
- We will instead set the action command to a fixed value of our own choosing by calling the setActionCommand( ) method, shown here:
  - **`void setActionCommand(String cmd)`**
- The string passed in cmd becomes the new action command.
- The text in the text field is unaffected.
- Once you set the action command string, it remains the same no matter what is entered into the text field.
- One reason that you might want to explicitly set the action command is to provide a way to recognize the text field as the source of an action event.
- This is especially important when another control in the same frame also generates action events and you want to use the same event handler to process both events.
- Setting the action command gives you a way to tell them apart.
- Also, if you don’t set the action command associated with a text field, then the contents of the text field might accidentally match the action command of another component.

See live code in STS “BasicSwingJTextFieldDemo”

Swing - JCheckBox

- After the push button, The check box is a widely used control in the creation of interactive GUIs.
- In Swing, a check box is an object of type **JCheckBox**. **JCheckBox** inherits **AbstractButton** and **JToggleButton**.
- Thus, a check box is, essentially, a special type of button.
- **JCheckBox** defines several constructors. Typical one is **`JCheckBox(String str)`**
- It creates a check box that has the text specified by **str** as a label.
- When a check box is selected or deselected (that is, checked or unchecked), an item event is generated.
- Item events are represented by the **ItemEvent** class.
- Item events are handled by classes that implement the **ItemListener** interface. This interface specifies only one method: **itemStateChanged( )**, which is shown here:
  - **`void itemStateChanged(ItemEvent ie)`**
- The item event is received in **ie**.

Swing - JCheckBox

- To obtain a reference to the item that changed, call **getItem( )** on the **ItemEvent** object. This method is shown here:
  - `Object getItem( )`
- The reference returned must be cast to the component class being handled, which in this case is **JCheckBox**.
- You can obtain the text associated with a check box by calling **getText( )**. You can set the text after a check box is created by calling **setText( )**. These methods work the same as they do for **JButton**, described earlier.
- The easiest way to determine the state of a check box is to call the **isSelected( )** method. It is shown here:
  - `boolean isSelected( )`
- It returns **true** if the check box is **selected** and **false otherwise**.

See live code in STS “BasicSwingJCheckBoxDemo”

Swing - JList

- **JList** is Swing's basic list class. It supports the selection of one or more items from a list.
- Although often the list consists of strings, it is possible to create a list of just about any object that can be displayed.
- **JList** is so widely used in Java that it is highly unlikely that you have not seen one before.
- In the past, the items in a JList were represented as Object references.
- Beginning with **JDK 7**, **JList** was made generic, and it is now declared like this:
  - **`class JList<E>`**
- Here, **E** represents the type of the items in the list. As a result, **JList is now type-safe**.
- **JList** provides several constructors. Here is a sample: **`JList(E[ ] items)`**
- This creates a **JList** that contains the items in the array specified by items.

Swing - JList

- Although a **JList** will work properly by itself, most of the time you will wrap a **JList** inside a **JScrollPane**, which is a container that automatically provides scrolling for its contents. Here is the constructor that we will use:
  - `JScrollPane(Component comp)`
- Here, comp specifies the component to be scrolled, which in this case will be a **JList**. When you wrap a **JList** in a **JScrollPane**, long lists will **automatically be scrollable**.
- Regardless of the amount of entries you have, you never worry about changing the size of the **JList** component.
- A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection.
- This event is also generated when the user deselects an item. It is handled by implementing **ListSelectionListener**. This listener specifies only one method, called valueChanged( ), which is shown here:
  - `void valueChanged(ListSelectionEvent le)`
- **le** is a reference to the object that generated the event. Although **ListSelectionEvent** does provide some methods of its own, often you will interrogate the **JList** object itself to determine what has occurred.
- By default, a **JList** allows the user to select multiple ranges of items within the list, but you can change this behavior by calling **setSelectionMode( )**, which is defined by JList. Here's the method:
  - `void setSelectionMode(int mode)`

`ListSelectionModel.SINGLE_SELECTION`
single item selection

`ListSelectionModel.SINGLE_INTERVAL_SELECTION`
continuous range selection of items

`ListSelectionModel.MULTIPLE_INTERVAL_SELECTION`
disjointed range selection of items in list

You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling `getSelectedIndex( )`.

```
int getSelectedIndex( )
```

Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, –1 is returned.

You can obtain an array containing all selected items by calling `getSelectedIndices( ).`

```
int[ ] getSelectedIndices( )
```

In the returned array, the indices are ordered from smallest to largest. If a zero-length array is returned, it means that no items are selected.

See live code in STS “BasicSwingJListDemo”

Swing - Anonymous Inner Classes and Lambda Expressions to Handle Events

- So far the programs in this session have used a simple, straightforward approach to handling events in which the main class of the application has implemented the listener interface itself and all events are sent to an instance of that class.
- We've also seen implementation where we separated listeners and classes such as the first 4 basic swing demos.
- There are 2 others ways we can add event listeners:
  - **anonymous inner classes**
  - **lambda expression**

Swing - Anonymous Inner Classes and Lambda Expressions to Handle Events

Anonymous Inner Classes

- Anonymous inner classes are inner classes that don’t have a name.
- Instead, an instance of the class is simply generated **“on the fly”** as needed.
- Anonymous inner classes make implementing some types of event handlers much easier.
- For example, given a **JButton** called **jbtn**, you could implement an action listener for it like this:

```
jbtn.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent ae) {
    // Handle action event here.
 }
});
```

- An anonymous inner class is created that implements the **ActionListener** interface.
- Pay special attention to the syntax.
- The body of the inner class begins after the **{** that follows **new ActionListener( )**.
- Also notice that the call to **addActionListener( )** ends with a **})** and a **;** just like normal.
- The same basic syntax and approach is used to create an anonymous inner class for any event handler.
- Of course, for different events, you specify different event listeners and implement different methods.
- One advantage to using an anonymous inner class is that the component that invokes the class’ methods is already known.
- For instance, in the preceding example, there is no need to call **getActionCommand( )** to determine what component generated the event, because this implementation of **actionPerformed( )** will only be called by events generated by **jbtn**.

Swing - Anonymous Inner Classes and Lambda Expressions to Handle Events

Lamba Expression

- In the case of an event whose listener defines a functional interface, you can handle the event by use of a lambda expression.
- Action events can be handled with a lambda expression because **ActionListener** defines only one abstract method, **actionPerformed( )**.
- Using a lambda expression to implement **ActionListener** provides a compact alternative to explicitly declaring an anonymous inner class.
- For example, again assuming a **JButton** called **jbtn**, you could implement the action listener like this:

```
jbtn.addActionListener((ae) -> {
 // Handle action event here.
});
```

- Just as with the anonymous inner class approach, the object that generates the event is known. In this case, the lambda expression applies only to the **jbtn** button.
- When an event can be handled by use of a single expression, use an expression lambda.

```
jbtnUp.addActionListener((ae) -> jlab.setText("You pressed Up."));
```

- In general, use a lambda expression to handle an event when its listener defines a functional interface.
- **ItemListener** is also a functional interface.

Swing – Grid Layout

Grid Layout allows for the arrangement of information in a row/column layout.

The constructor new GridLayout(int rows, int cols) defined the layout of a component.

The “add()” method will add items row by row, from left to right.

See live code in STS “BasicSwingGridLayoutDemo”

Swing – Sample App

A Swing-Based File Comparison App

Case Study: Given what we've learned so far, we will be creating an app which compares the content of 2 files.

See live code in STS “BasicSwingCaseStudy”