Swing

Unit Outcomes

2a. Differentiate between AWT and Swing on the given aspect

2b. Develop Graphical User Interface (GUI) programs using swing components for the given problem.

2c. Use the given type of Button in the Java based GUI.

2d. Develop Graphical User Interface (GUI) programs using advanced swing components for the given problem.

Origin of Swing

- The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.
- AWT translates its various visual components into their corresponding, platform-specific equivalents, or *peers*.
- This means that the look and feel of a component is defined by the platform, not by Java.
- Because the AWT components use native code resources, they are referred to as *heavyweight*.

Problems arising due to native peers

- Because of variations between operating systems, a component might look, or even act, differently on different platforms. This potential variability threatened the overarching philosophy of Java: **write once, run anywhere.**
- The look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed.
- The use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component was always opaque.

Swing is built on AWT

- Swing eliminates a number of the limitations inherent in the AWT, Swing *does not* replace it.
- Instead, Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java.
- Swing also uses the same event handling mechanism as the AWT.
- Hence a basic understanding of the AWT and of event handling is required to use Swing.

Swing Features

- Lightweight components
- Pluggable look and feel.
- Together they provide an elegant, yet easy-to-use solution to the problems of the AWT.

Lightweight Components

- They are **written entirely in Java** and do not map directly to platform-specific peers.
- Lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system.
- As a result, each component will work in a consistent manner across all platforms.

Pluggable Look and Feel (PLAF)

- Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing.
- This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does.
- Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects.
- Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing.
- This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does.
- Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects.
- It is possible to define a look and feel that is consistent across all platforms.
- Conversely, it is possible to create a look and feel that acts like a specific platform.
- For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel.
- It is also possible to design a custom look and feel.
- Finally, the look and feel can be changed dynamically at run time.

Model-View and Controller (MVC)

A visual component is a composite of three distinct aspects:

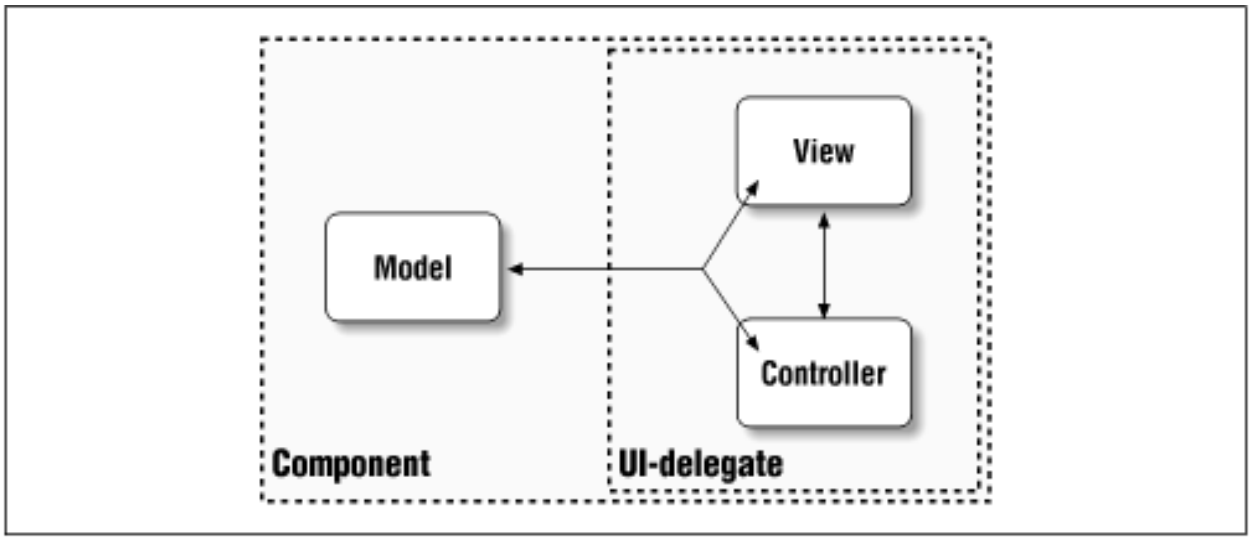
• The way that the component looks when rendered on the screen

• The way that the component reacts to the user.

• The state information associated with the component

- 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.
- Therefore, although Swing's component architecture is based on MVC, it does not use a classical implementation of it.



Model- Delegate

- This design combines the view and the controller object into a single element that draws the component to the screen and handles GUI events known as the *UI delegate* .
- Bundling graphics capabilities and event handling is somewhat easy in Java, since much of the event handling is taken care of in AWT.
- The communication between the model and the UI delegate then becomes a two-way street.
- Each Swing component contains a model and a UI delegate.
- The model is responsible for maintaining information about the component's state.
- The UI delegate is responsible for maintaining information about how to draw the component on the screen.
- The UI delegate (in conjunction with AWT) reacts to various events that propagate through the component.

Components and Containers

- A *component* is an independent visual control, such as a push button or slider.
- 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.
- Thus, all Swing GUIs will have at least one container.

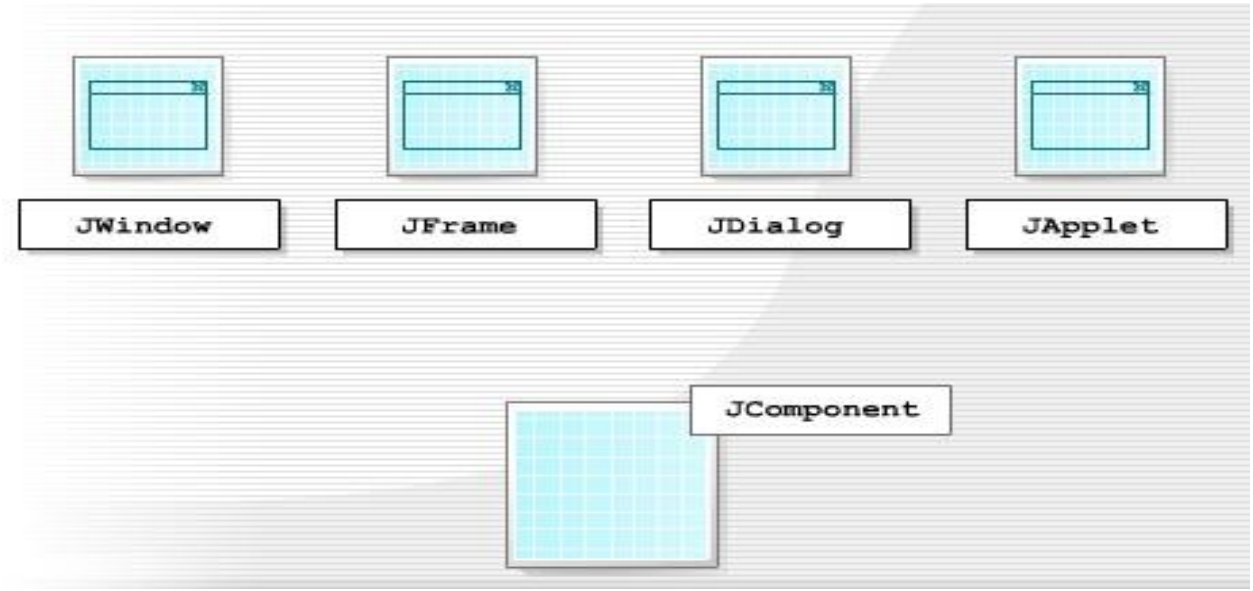
- Because containers are components, a container can also hold other containers.

Components

- In general, Swing components are derived from the **JComponent** class. (Except top level containers)
- **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.

Containers

- Swing defines two types of 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**.
- Hence, the top-level containers are **heavyweight**.
- 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.



JWindow

- JWindow is a top-level window that doesn't have any trimmings and can be displayed anywhere on a desktop.
- JWindow is a heavyweight component. It is used to create pop-up windows and "splash" screens.

- JWindow extends AWT's Window class.

JFrame

- JFrame is a top-level window that can contain borders and menu bars.
- JFrame is a subclass of JWindow and is thus a heavyweight component.
- JFrame is placed on a JWindow.
- JFrame extends AWT's Frame class.

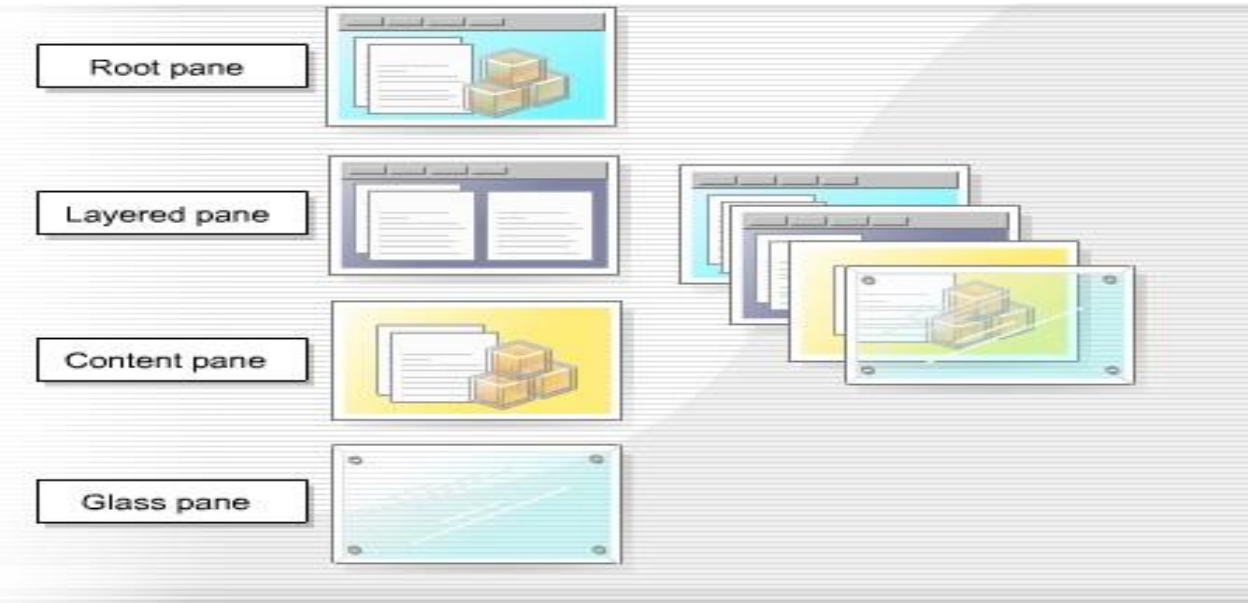
JDialog

- JDialog is a lightweight component that you use to create dialog windows.
- Dialog windows can be placed on JFrame or JApplet.
- JDialog extends AWT's Dialog class.

JApplet

- JApplet is a container that provides the basis for applets that run within web browsers.
- JApplet is a lightweight component that can contain other graphical user interface (GUI) components.
- JApplet extends AWT's Appletclass.

Panes in top level containers



Root Pane

- The root pane is an intermediate container that manages the layered pane, content pane, and glass pane.
- It can also manage an optional menu bar.
- Root pane is used to paint over multiple components or to catch input events.

Layered pane

- The layered pane contains the content pane and the optional menu bar.
- It can also contain other components, which it arranges so that they overlap each other.
- This enables you to add pop-up menus to applications.
- The layered pane provides six functional layers in which the components that you add can be placed.
- Each of these functional layers is used for a specific function.

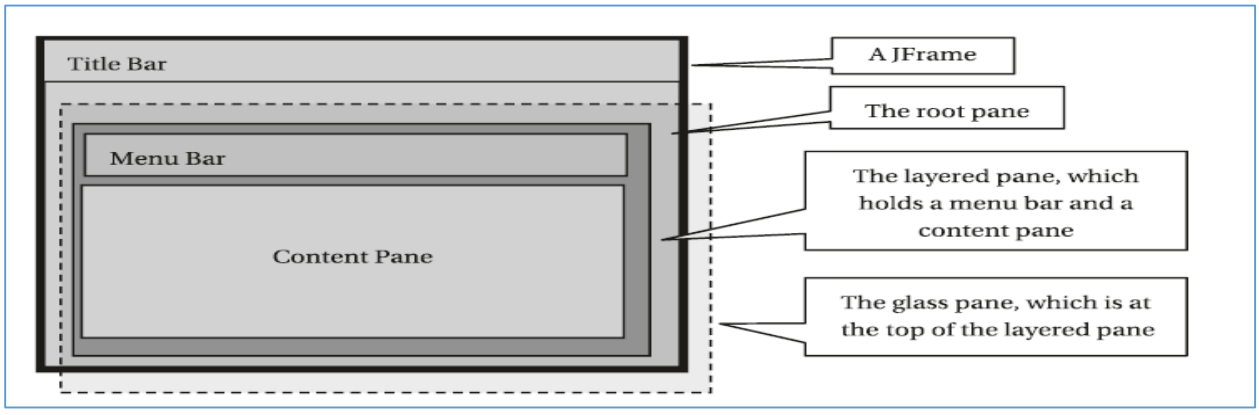
Content pane

- The content pane holds all the visible components of the root pane, except the menu bar.
- It covers the visible section of the JFrame or JWindow and you use it to add components to the display area.
- Java automatically creates a content pane when you create a JFrame or JWindow but you can create your own content pane, which has to be opaque.

Glass pane

- The glass pane is invisible by default but you can make it visible.
- When it is visible, it covers the components of the content pane, blocks all input events from reading these components, and can paint over an existing area containing one or more components.

JFrame



Example

- Suppose JFrame is a picture frame.
- Glass cover of picture frame corresponds to Glass Pane.
- Behind glass cover, picture is placed. That is the layered pane.
- Multiple pictures can be placed in one picture frame.
- Each picture will make up one layer behind the glass cover. As long as the pictures don't overlap each other, you can view them.
- Add pictures taken together in different layers form the layered pane of the picture frame.
- The picture layer farthest from the glass cover is the content pane.
- Normally picture frame contains only one picture. So the layered pane contains only one picture and hence only one content pane.

JFrame/JApplet

- A JFrame contains a JRootPane as its only child. The content pane provided by the root pane should, as a rule, contain all the non-menu components displayed by the JFrame.
- frame.add(child);
- The child will be added to the contentPane.
- The default content pane will have a BorderLayout manager set on it.
- Same is applicable for JApplet.

Difference between AWT and Swing

AWT	SWING
Components are heavyweight	Components are lightweight
Platform dependent	Platform independent
Does not support pluggable look and feel	Supports pluggable look and feel
Does not support MVC pattern	Supports MVC pattern
AWT components require and occupy larger memory space.	Swing components do not occupy as much memory space as AWT components.
AWT components are slower than swing components in terms of performance.	Swing in Java is faster than AWT in Java.
AWT components are thread safe	Swing components are not thread safe

Thread

A thread is a:

- Facility to allow multiple activities within a single process.
- Referred as lightweight process.
- A thread is a series of executed statements.
- Each thread has its own program counter, stack and local variables.
- A thread is a nested sequence of method calls.
- Its shares memory, files and per-process state.

Need of a Thread

- To perform asynchronous or background processing
- Increases the responsiveness of GUI applications
- Take advantage of multiprocessor systems
- Simplify program logic when there are multiple independent entities

What happens a Thread is invoked?

- When a thread is invoked, there will be two paths of execution.
- One path will execute the thread
- The other path will follow the statement after the thread invocation.
- There will be a separate stack and memory space for each thread.

Risk Factor

- Proper co-ordination is required between threads accessing common variables [use of synchronized and volatile] for consistence view of data
- Overuse of java threads can be hazardous to program's performance and its maintainability.

Deadlock

- Deadlock in java is a part of multithreading.
- Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread.

- Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

Thread safe

- A class **is thread-safe** if it behaves correctly when accessed from multiple **threads**, regardless of the scheduling or interleaving of the execution of those **threads** by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.
- A thread is an execution path of a program.
- A single threaded program will only have one thread and so this problem doesn't arise.
- Virtually all GUI programs have multiple execution paths and hence threads - there are at least two, one for processing the display of the GUI and handing user input, and at least one other for actually performing the operations of the program.
- This is done so that the UI is still responsive while the program is working by offloading any long running process to any non-UI threads.
- These threads may be created once and exist for the lifetime of the program, or just get created when needed and destroyed when they've finished.

Why AWT is Thread safe?

- The AWT is based on the OS's WIndowing System's peer objects which are inherently thread safe.
- That is why AWT is thread safe.

Why swing is not Thread safe?

- The Core part of Swing is made up of different GUI component e.g. JLable, JPanel, JCombobox, etc.
- All these components are **not thread-safe**, which means you can not call methods of these components like the JLable.setText("new title") from any thread, other than **Event Dispatcher Thread(EDT)**.

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.
- The AWT uses a single-threaded model in which all screen updates must be performed from a single thread. The event dispatching thread is the only valid thread to update the visual state of visible user interface components.

- When you run a GUI applet or application, the code in your main() method creates a GUI and sets up event handling.
- It is a Thread automatically created by JVM when it detects that it is working with event handling.
- The JVM uses this thread to execute the component's event handlers.
- If the user had to wait for the main() method to finish whatever it is doing (say calculating pi up to 10000 decimal places)before they could interact with the GUI, then the UI is of no use.
- So the AWT library implements its own thread to watch GUI interaction. This thread is essentially a little loop that checks the system event queue for mouse clicks, key presses and other system-level events.
- The AWT thread (aka the "Event Dispatch" thread) grabs a system event off the queue and determines what to do with it.
- *The AWT thread also handles repainting of your GUI. Anytime you call repaint(), a "refresh" request is placed in the event queue. Whenever the AWT thread sees a "refresh" request, it examines the GUI, then calls the appropriate methods to layout the GUI and paint any components that require painting.*

Solution? SwingUtilities class

- In Java, after swing components displayed on the screen, they can be operated by only one thread called **Event Handling Thread**.
- We can write our code in a separate block and can give this block reference to **Event Handling thread**.
- The **SwingUtilities** class has two important static methods, **invokeAndWait()** and **invokeLater()** to use to put references to blocks of code onto the event **queue**.

Syntax

- **public static void invokeAndWait (Runnable target) throws InterruptedException, InvocationTargetException**
- **public static void invokeLater (Runnable target)**

invokeLater()	**invokeAndWait()**
. InvokeLater() method in swing is asynchronous	invokeAndWait() method in swing is synchronous
It posts an action event to the event queue and returns immediately. It will not wait for the task to complete	It blocks until Runnable task is complete
invokeLater() keeps the code on event thread and runs the rest of code in the thread.	invokeAndWait() keeps the code on to event thread and waits till the execution of run method is completed
Asynchronous: Does not block the program from the code execution	Synchronous: Code execution waits before continuing

Solution?

- Swing provides a mechanism to deal with the issues of threading –
- Use SwingUtilities class to update any Swing component from a thread other thread Event Dispatcher Thread.
- javax.swing.SwingUtilities.invokeLater(Runnable ...);
- javax.swing.SwingUtilities.invokeAndWait(Runnable ...);

Swing Components

JApplet	JSeparator
JLabel	JTabbedPane
JTextField	JTable
JButton	JTextArea
JCombobox	JTextField
JRadioButton	JToolTip
JCheckBox	JToggleButton
JScrollPane	JTree

JLabel

- JLabel is used to display a short string or an image icon.
- JLabel can display text, image or both .
- JLabel is only a display of text or image and it cannot get focus .
- JLabel is inactive to input events such a mouse focus or keyboard focus.
- By default labels are vertically centered but the user can change the alignment of label.

Constructors

- **JLabel() :** Creates a blank label with no text or image in it.
- **JLabel(String s) :** Creates a new label with the string specified.
- **JLabel(Icon i) :** Creates a new label with a image on it.
- **JLabel(String s, Icon i, int align) :** Creates a new label with a string, an image and a specified horizontal alignment

Methods:

Method	**Description**
String getText()	Returns the text string that a label displays.
void setText(String text)	Defines the single line of text this component will display.
Icon getIcon()	Returns the graphic image that the label displays.
void setIcon(Icon icon)	Defines the icon this component will display.
public Image getScaledInstance(int width, int height, int hints)	Creates a scaled version of this image. hints - flags to indicate the type of algorithm to use for image resampling.

Closing the Frame

- The setDefaultCloseOperation() method is used to specify one of several options for the close button. Use one of the following constants to specify your choice:
- JFrame.EXIT_ON_CLOSE — Exit the application.
- JFrame.HIDE_ON_CLOSE — Hide the frame, but keep the application running.

- JFrame.DISPOSE_ON_CLOSE — Dispose of the frame object, but keep the application running.
- JFrame.DO_NOTHING_ON_CLOSE — Ignore the click.
- If you forget to call setDefaultCloseOperation() you will get JFrame.HIDE_ON_CLOSE by default.
- This can be frustrating, because it looks like you have "killed" the program, but it keeps on running, and you see no frame.

SwingUtilities

- Consider the sequence

```
SwingUtilities.invokeLater(new Runnable()
{
public void run()
{
new swinguti();
}
}
```

- This sequence causes a new swinguti object to be created on the event dispatching thread and not on main thread.
- To avoid deadlock, all swing GUI components must be created and updated from the event dispatching thread.
- main() is executed from main thread.
- Thus main() cannot instantiate a swinguti object.
- Hence it creates a Runnable object that executes on the event dispatching thread and have this object create GUI

JButton

- Swing defines four types of buttons:

JButton, **JToggleButton**, **JCheckBox**, and **JRadioButton**.

- All are subclasses of the **AbstractButton** class, which extends **Jcomponent**.
- Different icons that are displayed for the button when it is disabled, pressed, or selected. Another icon can be used as a *rollover* icon, which is displayed

when the mouse is positioned over a button. The following methods set these icons:

void setDisabledIcon(Icon *di*)

void setPressedIcon(Icon *pi*)

void setSelectedIcon(Icon *si*)

void setRolloverIcon(Icon *ri*)

- Here, *di, pi, si*, and *ri* are the icons to be used for the indicated purpose.
- The **JButton** class provides the functionality of a push button.
- **JButton** allows an icon, a string, or both to be associated with the push button.
- Constructors

JButton(Icon *icon*)

JButton(String *str*)

JButton(String *str*, Icon *icon*)

- *str* and *icon* are the string and icon used for the button.

Method	**Description**
void setText(String s)	It is used to set specified text on button
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.

void setHorizontalTextPosition(int textPosition)	Sets the button message on the LEFT/RIGHT of its icon or image.
void setVerticalTextPosition(int textPosition)	Sets the button message on the TOP/BOTTOM of its icon or image.

JToggleButton

- A JToggleButton is a two-state button.
- The two states are selected and unselected.
- The *JRadioButton* and *JCheckBox* classes are subclasses of this class.
- When the user presses the toggle button, it toggles between being pressed or unpressed.
- JToggleButton is used to select a choice from a list of possible choices.

JToggleButton : Constructors

- JToggleButton(): Creates an initially unselected toggle button without setting the text or image.
- JToggleButton(Icon icon): Creates an initially unselected toggle button with the specified image but no text.
- JToggleButton(Icon icon, boolean selected): Creates a toggle button with the specified image and selection state, but no text.
- JToggleButton(String text): Creates an unselected toggle button with the specified text.
- **JToggleButton(String text, boolean selected):** Creates a toggle button with the specified text and selection state.
- **JToggleButton(String text, Icon icon):** Creates a toggle button that has the specified text and image, and that is initially unselected.
- **JToggleButton(String text, Icon icon, boolean selected):** Creates a toggle button with the specified text, image, and selection state.
- ActionListener as well as ItemListener can be used to handle events.

JCheckBox

- The **JCheckBox** class provides the functionality of a check box. Its immediate superclass is **JToggleButton.**
- It is used to turn an option on (true) or off (false).
- Clicking on a CheckBox changes its state from "on" to "off" or from "off" to "on”.

JCheckBox: Constructor

- **JCheckBox()** : creates a new checkbox with no text or icon
- **JCheckBox(Icon i)** : creates a new checkbox with the icon specified.
- **JCheckBox(Icon icon, boolean s)** : creates a new checkbox with the icon specified and the boolean value specifies whether it is selected or not.
- **JCheckBox(String t)** :creates a new checkbox with the string specified.
- **JCheckBox(String text, boolean selected)** :creates a new checkbox with the string specified and the boolean value specifies whether it is selected or not.
- **JCheckBox(String text, Icon icon)** :creates a new checkbox with the string and the icon specified.
- **JCheckBox(String text, Icon icon, boolean selected)**: creates a new checkbox with the string and the icon specified and the boolean value specifies whether it is selected or not.

JRadioButton

- Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time.
- They are supported by the **JRadioButton** class, which extends **JToggleButton**.
- In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group.
- Only one of the buttons in the group can be selected at any time.
- A button group is created by the **ButtonGroup** class.
- Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:
- void add(AbstractButton *ab*)
- Here, *ab* is a reference to the button to be added to the group.

JRadioButton: Constructor

- **JRadioButton():** Creates an initially unselected radio button with no set text.
- **JRadioButton(Icon icon):** Creates an initially unselected radio button with the specified image but no text.
- **JRadioButton(Icon icon, boolean selected):** Creates a radio button with the specified image and selection state, but no text.
- **JRadioButton(String text, boolean selected):** Creates a radio button with the specified text and selection state.
- **JRadioButton(String text, Icon icon):** Creates a radio button that has the specified text and image, and which is initially unselected.
- **JRadioButton(String text, Icon icon, boolean selected):** Creates a radio button that has the specified text, image, and selection state.

JComboBox

- JComboBox inherits JComponent class .
- JComboBox shows a popup menu that shows a list and the user can select a option from that specified list .
- JComboBox can be editable or read- only depending on the choice of the programmer .

JComboBox: Constructor

- **JComboBox()** : creates a new empty JComboBox .
- **JComboBox(Object[] items):** Creates a JComboBox that contains the elements in the specified array.
- **JComboBox(Vector<?> items):** Creates a JComboBox that contains the elements in the specified Vector.

Method	Description
void addItem(Object anObject)	t is used to add an item to the item list.
void removeItem(Object anObject)	It is used to delete an item to the item list.
void removeAllItems()	It is used to remove all the items from the list.
void setEditable(boolean b)	It is used to determine whether the JComboBox is editable.

JTextField

- The class JTextField is a component that allows editing of a single line of text.
- JTextField inherits the JTextComponent class and uses the interface SwingConstants.

JTextField: Constructor

- JTextField() : constructor that creates a new TextField
- JTextField(int columns) : constructor that creates a new empty TextField with specified number of columns.
- JTextField(String text) : constructor that creates a new empty text field initialized with the given string.
- JTextField(String text, int columns) : constructor that creates a new empty textField with the given string and a specified number of columns .

JScrollPane

- JScrollPane is a lightweight container that automatically handles the scrolling of another component.
- The component being scrolled can be either an individual component, such as a table, or a group of components contained within another lightweight container, such as a JPanel.
- In either case, if the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane.
- The viewable area of a scroll pane is called the *viewport*.

- It is a window in which the component being scrolled is displayed.
- The viewport displays the visible portion of the component being scrolled.
- The scroll bars scroll the component through the viewport.
- In its default behavior, a **JScrollPane** will dynamically add or remove a scroll bar as needed.

JScrollPane : Constructors

- JScrollPane(): Creates an empty (no viewport view) JScrollPane where both horizontal and vertical scrollbars appear when needed.
- JScrollPane(Component view) : Creates a JScrollPane that displays the contents of the specified component, where both horizontal and vertical scrollbars appear whenever the component's contents are larger than the view.
- JScrollPane(Component view, int vsb, int hsb): Creates a JScrollPane that displays the view component in a viewport whose view position can be controlled with a pair of scrollbars.
- JScrollPane(int vsb, int hsb): Creates an empty (no viewport view) JScrollPane with specified scrollbar policies.
- vsb and hsb are integer constants that define when vertical and horizontal scrollbars for this scrollpane are shown.

ScrollPaneConstants

Constant	Description
HORIZONTAL_SCROLLBAR_ALWAYS	Always provide horizontal scrollbar
HORIZONTAL_SCROLLBAR_AS_NEEDED	Provide horizontal scrollbar, if needed
VERTICAL_SCROLLBAR_ALWAYS	Always provide vertical scrollbar
VERTICAL_SCROLLBAR_AS_NEEDED	Provide vertical scrollbar, if needed

Steps to follow to use a scroll pane:

1. Create the component(JComponent object) to be scrolled.
2. Create an instance of **JScrollPane(The argument to the constructor specify the component and the policies for vertical and horizontal scroll bars)**, passing to it the object to scroll.
3. Add the scroll pane to the content pane.

JList and Constructors

- JList is a component that displays a set of Objects and allows the user to select one or more items.
- JList inherits JComponent class.
- Constructors

1. **JList()**: creates an empty blank list.
2. **JList(Vector l)** : creates a new list with the elements of the vector
3. JList(E[ ] *items*)

- This creates a **JList** that contains the items in the array specified by *items*. **E** represents the type of the items in the list.
- **JList** is based on two models.
- The first is **ListModel**. This interface defines how access to the list data is achieved.
- The second model is the **ListSelectionModel** interface, which defines methods that determine what list item or items are selected.

Method	Description
int getSelectedIndex()	It is used to return the smallest selected cell index.
Object getSelectedItem()	Returns the selected object

JTree

- A tree is a component that presents a hierarchical view of data.
- The user has the ability to expand or collapse individual subtrees in this display.
- Trees are implemented in Swing by the JTree class.

JTree: Constructors

- JTree(Object obj [ ]): The tree is constructed from the elements in the array *obj*.
- JTree(Vector<?> v): The tree is constructed from the elements of vector *v*.
- JTree(TreeNode tn): The tree whose root node is specified by *tn* specifies the tree.
- Jtree(Hashtable ht): The tree is constructed in which each element of the hashtable ht is the child node.

- **Although JTree is packaged in javax.swing, its support classes and interfaces are packaged in javax.swing.tree.**
- **This is because the number of classes and interfaces needed to support JTree is quite large.**

TreeNode

- The TreeNode interface declares methods that obtain information about a tree node.
- The MutableTreeNode interface extends TreeNode.
- It declares methods that can insert and remove child nodes or change the parent node.
- The DefaultMutableTreeNode class implements the MutableTreeNode interface. It represents a node in a tree. One of its constructors is shown here:
- DefaultMutableTreeNode(Object *obj*)
- Here, *obj* is the object to be enclosed in this tree node. The new tree node doesn't have a parent or children.
- To create a hierarchy of tree nodes, the **add( )** method of **DefaultMutableTreeNode** can be used. Its signature is shown here:
- void add(MutableTreeNode *child*)
- Here, *child* is a mutable tree node that is to be added as a child to the current node.

- **JTree** does not provide any scrolling capabilities of its own. Instead, a **JTree** is typically placed within a **JScrollPane**.
- This way, a large tree can be scrolled through a smaller viewport.

JTree

- The getPathFor Location() method is used to translate a mouse click on a specific point of the tree to a tree path.
- Syntax:

TreePath getPathFor Location(int x, int y)

- x and y are the coordinates at which the mouse is clicked.

The TreePath class encapsulates information about a path to a particular node in a tree.

Here are the steps to follow to use a tree:

- 1. Create an instance of **JTree**.
- 2. Create a **JScrollPane** and specify the tree as the object to be scrolled.
- 3. Add the tree to the scroll pane.
- 4. Add the scroll pane to the content pane.

JTable

- **JTable** is a component that displays rows and columns of data.
- You can drag the cursor on column boundaries to resize columns.
- You can also drag a column to a new position.
- Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell.
- At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table.
- **JTable** does not provide any scrolling capabilities of its own. Instead, you will normally wrap a **JTable** inside a **JScrollPane**.
- Constructor:

JTable(Object *data*[ ][ ], Object *colHeads*[ ])

- *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.
- JTable(Object *data*[ ][ ], Object *colHeads*[ ])

  data is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.
- **JTable():** A table is created with empty cells.
- **JTable(int rows, int cols):** Creates a table of size rows * cols.
- **JTable** relies on three models. The first is the table model, which is defined by the **TableModel** interface. This model defines those things related to displaying data in a two-dimensional format.
- The second is the table column model, which is represented by **TableColumnModel**. **JTable** is defined in terms of columns, and it is **TableColumnModel** that specifies the characteristics of a column. These two models are packaged in **javax.swing.table**.
- The third model determines how items are selected, and it is specified by the **ListSelectionModel**, which was described when **JList** was discussed.

Steps to create JTable

- Create an instance of **JTable**.
- Create a **JScrollPane** object, specifying the table as the object to scroll.
- Add the table to the scroll pane.
- Add the scroll pane to the content pane.

TableModel

- The JTable is designed around the concepts of MVC (Model/View/Control), the table model controls how the data is stored.
- The **TableModel** interface specifies the methods the JTable will use to interrogate a tabular data model.

Method	Description
int **getColumnCount**()	Returns the number of columns managed by the data source object.
String getColumnName(int columnIndex)	Returns the name of the column at *columnIndex*.
int **getRowCount**()	Returns the number of records managed by the data source object.
Object getValueAt(int rowIndex, int columnIndex)	Returns an attribute value for the cell at *columnIndex* and *rowIndex*.
boolean isCellEditable(int rowIndex, int columnIndex)	Returns true if the cell at *rowIndex* and *columnIndex* is editable.
void **setValueAt**(Object aValue, int rowIndex, int columnIndex)	Sets an attribute value for the record in the cell at *columnIndex* and *rowIndex*.

JTabbedPane

- **JTabbedPane** encapsulates a *tabbed pane.* It manages a set of components by linking them with tabs.
- Selecting a tab causes the component associated with that tab to come to the forefront.

JTabbedPane: Constructors

- JTabbedPane(): Creates an empty TabbedPane with a default tab placement of JTabbedPane.TOP
- JTabbedPane(int tabPlacement): Creates an empty TabbedPane with the specified tab placement of either: JTabbedPane.TOP, JTabbedPane.BOTTOM, JTabbedPane.LEFT, or JTabbedPane.RIGHT.
- JTabbedPane(int tabPlacement, int tabLayoutPolicy): Creates an empty TabbedPane with the specified tab placement and tab layout policy.
- Tabs are added by calling **addTab( )**. Here is one of its forms:
- void addTab(String *name*, Component *comp*)

- *name* is the name for the tab, and *comp* is the component that should be added to the tab. Often, the component added to a tab is a **JPanel** that contains a group of related components.
- This technique allows a tab to hold a set of components.

Steps to create JTabbedPane

- 1. Create an instance of **JTabbedPane**.
- 2. Add each tab by calling **addTab( )**.
- 3. Add the tabbed pane to the content pane.

JProgressBar

- The JProgressBar class is used to display the progress of the task. It inherits JComponent class.

JProgressBar: Constructors

- JProgressBar(): It is used to create a horizontal progress bar but no string text.
- JProgressBar(int min, int max): It is used to create a horizontal progress bar with the specified minimum and maximum value.
- JProgressBar(int orient): It is used to create a progress bar with the specified orientation, it can be either Vertical or Horizontal by using SwingConstants.VERTICAL and SwingConstants.HORIZONTAL constants.
- JProgressBar(int orient, int min, int max): It is used to create a progress bar with the specified orientation, minimum and maximum value.

Method	Description
int getMaximum()	Returns the progress bar's maximum value.
int getMinimum()	Returns the progress bar's minimum value.
String getString()	Get the progress bar's string representation of current value.
void setMaximum(int n)	Sets the progress bar's maximum value to the value n.
void setMinimum(int n)	Sets the progress bar's minimum value to the value n.
void setValue(int n)	Set Progress bar's current value to the value n.

void setString(String s)	Set the value of the progress String to the String s.

JTooltip

- JTooltip text can be added to almost all the components of Java Swing by using the following method setToolTipText(String s).
- This method sets the tooltip of the component to the specified string s .
- When the cursor enters the boundary of that component a popup appears and text is displayed .

Method	**Description**
String getToolTipText()	Returns the tooltip text for that component.
void setToolTipText(String s)	Returns the progress bar's minimum value.
String getString()	Get the progress bar's string representation of current value.
void setMaximum(int n)	Sets the progress bar's maximum value to the value n.
void setMinimum(int n)	Sets the progress bar's minimum value to the value n.
void setValue(int n)	Set Progress bar's current value to the value n.
void setString(String s)	Set the value of the progress String to the String s.

JPasswordField

- The class JPasswordField is a component that allows editing of a single line of text where the view indicates that something was typed by does not show the actual characters.
- JPasswordField inherits the JTextField class in javax.swing package.

JPasswordField: Constructors

- **JPasswordField()**: constructor that creates a new PasswordField.
- **JPasswordField(int columns)** : constructor that creates a new empty PasswordField with specified number of columns.
- **JPasswordField(String Password)** : constructor that creates a new empty Password field initialized with the given string.
- **JPasswordField(String Password, int columns)** : constructor that creates a new empty PasswordField with the given string and a specified number of columns .
- **JPasswordField(Document doc, String Password, int columns)** : constructor that creates a Passwordfield that uses the given text storage model and the given number of columns.

Method	Description
char getEchoChar()	Returns the character used for echoing in JPasswordField.
void setEchoChar(char c)	Set the echo character for JPasswordField.
String getPassword()	Returns the text contained in JPasswordField.
String getText()	Returns the text contained in JPasswordField.