COMPUTER SCIENCE

B. Sc. III

Semester-VI

2023-2024

6S : Advanced Java & VB.net

Unit-III : Event Handling & AWT

Prof. V. V. Agarkar

Assistant Professor & Head

Department of Computer Science

Shri. D. M. Burungale Science & Arts College, Shegaon, Dist. Buldana

Unit III

Event Handling and AWT: Introduction, Event delegation model, Java AWT event description, sources of event, Event listener interfaces, Adapter classes, Inner classes. AWT (Abstract Window Toolkit): Introduction, AWT Controls Label, Button, Checkboxes, Lists, ScrollBar, TextField, TextArea, Layout manager.

Introduction

Any application in Java uses graphical user interface (GUI), such as Windows based application or web based application, are event driven. Thus these types of application must be written with event handling. Most events to which the program will respond are generated when the user interacts with a GUI-based program. Events are supported in Java by a number of packages, including `java.util`**,** `java.awt`, and `java.awt.event`.

Event Handling

The event-driven programming paradigm is the most important aspect of GUI programming. Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as ***event handler*** that is executed when an event occurs. The older approach in Java to handle event is inheritance-based event model (in Java 1.0). Old methods are still supported, but deprecated and hence not recommended for new programs.

But, the modern approach for event handling in Java is now based on the Delegation Model. This model defines the standard mechanism to generate and handle the events.

Event Delegation Model

The Delegation Event model is used in Java to handle events. This is the modern approach for event handling and it defines a standard and compatible mechanism to generate and process events.

The concept of delegation model is simple: “a **source** generates an event and sends it to one or more **listeners**. The **listener** simply waits until it receives an event. Once received, the listener processes the event and then returns”. But, a listener must be registered with a source in order to receive an event. The main advantage of the Delegation Event Model is that the *application logic* (to process events) is completely separated from the *interface logic* (that generates events).

Basically, an Event Delegation Model is based on the following three components:

- Events
- Events Sources
- Events Listeners

- **Events**

The Events are the objects that define state change in a source. An event in Java is an object that is created when something changes within a graphical user interface. If a user clicks on a button, clicks on a combo box, or types characters into a text field, moving the mouse pointer, pressing the keyboard key, selecting an item from the list, etc., then an event triggers, creating the relevant event object. The events which are generated due to interaction by the user on components in GUI are called *Foreground Events*.

Some events may be generated without user's interaction, called *Background Events*. Examples of these events are a timer expires, operating system failures/interrupts, operation completion, etc.

- **Event Sources**

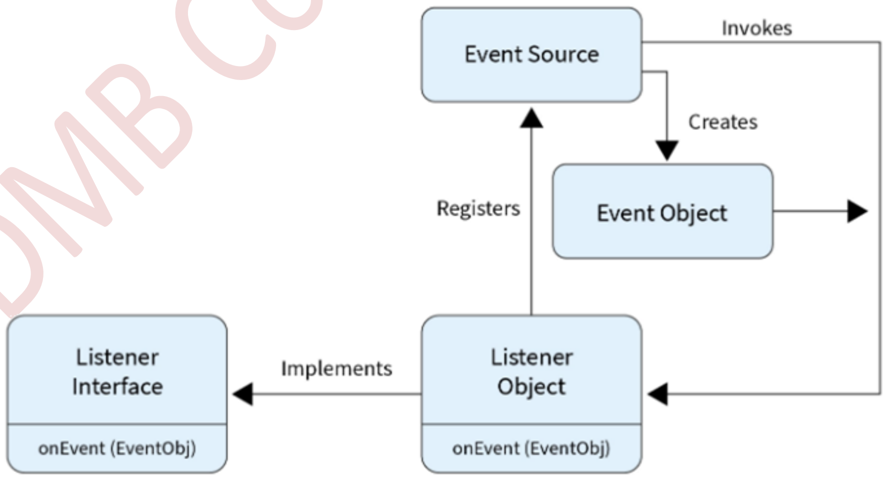
An event source is an object that generates an event. It is typically a graphical user interface (GUI) component, such as a button, a text field, or a menu item, but it can also be other types of objects that generate events, such as timers or sockets.

When an event is triggered on an event source, the event is encapsulated in an event object and passed to all the registered event listeners or handlers. The event source is responsible for notifying the event listeners or handlers of the event and providing them with the information they need to handle the event.

- **Event Listeners**

It is also called as *event handler*. An event listener is an object that is registered to an event source and is responsible for handling events that are generated by that source. When an event occurs on the event source, the event listener is notified and performs the necessary actions to respond to the event.

An event listener in Java is typically implemented as a class that implements a specific event listener interface, such as ActionListener, MouseListener, or KeyListener. Each interface specifies a set of methods that the event listener must implement to handle the corresponding type of event. For example, the ActionListener interface requires the implementation of a single method called actionPerformed, which is called when an action event occurs on the event source.



[Fig.1 : Event Delegation Model]

Sources of Events

Following **Table-1** lists some of the user interface components that can generate the events. In addition to these graphical user interface elements, any class derived from

`Component`, such as `Frame`, can generate events. For example, user can receive key and mouse events from an instance of `Frame`.

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

[**Table-1**: Event Source Examples]

Event Listener Interfaces

The delegation event model has two parts: sources and listeners. Listeners are created by implementing one or more of the interfaces defined by the **`java.awt.event`** package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. **Table-2** lists several commonly used listener interfaces and provide a brief description of the methods that they define.

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustrnentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.

WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

[**Table-2**: Commonly Used Event Listener Interfaces]

Event Classes

The classes that represent events are at the core of Java's event handling mechanism. Java defines several types of events. The most widely used events are defined by the `AWT` and by `Swing`.

The package `java.awt.event` defines many types of events that are generated by various user interface elements. **Table-3** shows several commonly used event classes and provides a brief description of when they are generated.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

[**Table-3**: Commonly used event classes in `java.awt.event`]

Commonly used constructors and methods in each of the above class are described in the following section.

The ActionEvent Class

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: `ALT_MASK`, `CTRL_MASK`, `META_MASK`, and `SHIFT_MASK`. In addition, there is an integer constant, `ACTION_PERFORMED`, which can be used to identify action events.

ActionEvent has these three constructors:

```
ActionEvent(Object src, int type, String cmd)
ActionEvent(Object src, int type, String cmd, int modifiers)
ActionEvent(Object src, int type, String cmd, long when, int modifiers)
```

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*, and its command string is *cmd*. The argument *modifiers* indicates which modifier keys (`alt`, `ctrl`, `meta`, and/or `shift`) were pressed when the event was generated. The *when* parameter specifies when the event occurred.

You can obtain the command name for the invoking **ActionEvent** object by using the `getActionCommand()` method, shown here:

```
String getActionCommand()
```

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

The `getModifiers()` method returns a value that indicates which modifier keys (`ALT`, `CTRL`, `META`, and/or `SHIFT`) were pressed when the event was generated. Its form is shown here:

```
int getModifiers()
```

The method `getWhen()` returns the time at which the event took place. This is called the event's *timestamp*. The `getWhen()` method is shown here:

```
long getWhen()
```

Using the Delegation Event Model

To use event delegation model, just follow these two steps:

1. Implement the appropriate interface in the listener so that it can receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events. In all cases, an event handler must return quickly i.e. an event handler must not retain control for an extended period of time.

Adapter Classes

Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

For example, the `MouseMotionAdapter` class has two methods, `mouseDragged()` and `mouseMoved()`, which are the methods defined by the `MouseMotionListener` interface. If you were interested in only mouse drag events, then you could simply extend `MouseMotionAdapter` and override `mouseDragged()`. The empty implementation of `mouseMoved()` would handle the mouse motion events for you.

Table-4 lists several commonly used adapter classes in `java.awt.event` and notes the interface that each implements.

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener, MouseMotionListener, and MouseWheelListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener, WindowFocusListener, and WindowStateListener

[**Table-4**: Commonly Used Listener Interfaces Implemented by Adapter Classes]

Inner Classes

An inner class in Java is defined as a class that is declared inside another class. Inner classes are often used to create nested data structures, such as a linked list.

Inner classes can be either static or non-static. A static inner class is one that is declared with the static keyword. A non-static inner class is one that is not declared with the static keyword.To access the inner class, create an object of the outer class, and then create an object of the inner class:

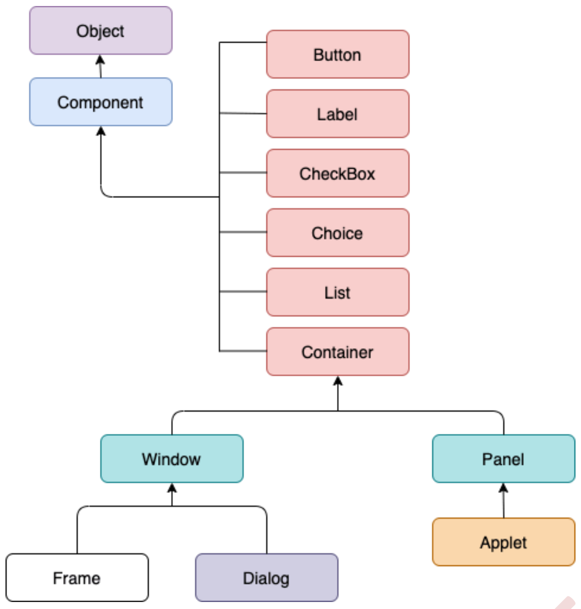
Abstract Window Toolkit (AWT)

Java AWT (Abstract Window Toolkit) is set of Application Program Interfaces (APIs) to develop Graphical User Interface (GUI) or window-based applications in java. It is used to create GUI objects such as buttons, scroll bars, windows etc. Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

(*A more recent set of GUI interfaces called* **Swing** *extends the AWT so that the programmer can create generalized GUI objects that are independent of a specific OS's windowing system.*)

Java AWT Hierarchy

The hierarchy of Java AWT classes are shown in Fig.2 on next page:



[**Fig.2** : Java AWT Hierarchy]

Components:

Component class is at the top of AWT hierarchy. AWT provides various components such as buttons, labels, text fields, checkboxes, etc used for creating GUI elements for Java Applications.

Container

Container is a component in AWT that contains another component like button, text field, tables etc. Container is a subclass of component class. AWT provides containers like panels, frames, and dialogues to organize and group components in the Application. There are four types of containers in Java AWT: 1) Window 2) Panel 3) Frame and 4) Dialog.

Window

Window is a top-level container that represents a graphical window or dialog box. The window is the container that have no borders and menu bars. The Window class extends the Container class, which means it can contain other components, such as buttons, labels, and text fields.

Panel

Panel is a container class in Java. It is a lightweight container that can be used for grouping other components like button, textfield etc. together within a window or a frame. The Panel does not contain title bar, border or menu bar.

Frame

The Frame is the container that contain title bar and border and can have menu bars. It can have other components like button, text field, scrollbar etc. Frame is most widely used container while developing an AWT application.

Dialog

The Dialog class provides a special type of display window that is normally used for pop-up messages or input from the user.

Controls

Java AWT controls are the controls that are used to design graphical user interfaces or web applications. The controls are components that allow a user to interact with the applications in various ways. To make an effective GUI, Java provides `java.awt` package that supports various AWT controls like Label, Button, CheckBox, List, Text Field, Text Area, etc that creates or draws various components on web and manages the GUI based application. A *layout manager* automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them. It is also possible to manually position components within a window, but which is quite tedious. The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Lists
- Choice lists
- Scroll bars
- Text Editing

All AWT controls are subclasses of **Component**. Although the set of controls provided by the AWT is not particularly rich, it is sufficient for simple applications and also quite useful for the basic concepts and techniques related to handling events in controls.

• *Adding and Removing Controls*

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling `add()`, which is defined by `Container`. The `add()` method has several forms, the following is one of its form:

```
Component add(Component compRef)
```

Here, *compRef* is a reference to an instance of the control that you want to add. A reference to the object is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call `remove()`. This method is also defined by **Container**. Here is one of its forms:

```
void remove(Component compRef)
```

Here, *compRef* is a reference to the control you want to remove. You can remove all controls by calling **removeAll( )**.

Labels

A Label is a GUI control which can be used to display static uneditable text. The Label contains a string and is an object of type Label. Labels are generally limited to single-line messages. Labels are usually used to identify components. Labels are passive controls that do not support any interaction with the user and do not fire any events. Label can be created using the *Label* class.

Label Constructors:

Label defines the following constructors:

```
Label() throws HeadlessException
Label(String str) throws HeadlessException
Label(String str, int how) throws HeadlessException
```

The first version creates a blank label. The second version creates a label that contains the string specified by *str*. This string is left-justified. The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: `Label.LEFT`, `Label.RIGHT`, or `Label.CENTER`.

Label Methods:

1) `setText()` method

This method is used to set or change the text in a label. The syntax of the method is:

```
void setText(String str)
```

Where, `str` specifies the new label to set.

2) `getText()` method

This method is used to obtain the current label. The syntax of the method is:

```
String getText()
```

The current label is returned.

3) `setAlignment()` method

This method sets the alignment for the label to the specified alignment. The syntax of the method is:

```
void setAlignment(int how)
```

Here, `how` must be one of the alignment constants.

4) `getAlignment()` method

This method is used to get the current alignment of the label. The syntax of the method is:

```
int getAlignment()
```

Returns the current alignment of the label i.e. LEFT, RIGHT or CENTER.

Buttons

Button is a control component has a text on the face of the button is called *button label* and when the user presses this button, it produces an event. It is a regular push button and whenever a push button is clicked, it generates an event and executes the code of the specified listener.

If an application wants to perform some action based on a button being pressed and released, it should implement `ActionListener` and register the new listener to receive events from this button, by calling the button's `addActionListener` method.

When a button is pressed and released, AWT will send an instance of `ActionEvent` to the button, by calling `processEvent` on the button. Button's `processEvent` method receives all events for the button and passes an action event along by calling its own `processActionEvent` method. Latter method passes the action event on to any action listeners have interest in action events generated by this button.

- **Button Constructors:**

Button defines the following two constructors:

```
Button() throws HeadlessException
Button(String str) throws HeadlessException
```

The first version creates an empty button. The second creates a button that contains *str* as a label.

- **Button Methods:**

After a button has been created, you can set its label by calling `setLabel()`. You can retrieve its label by calling `getLabel()`. These methods are as follows:

```
void setLabel(String str)
String getLabel()
```

Here, *str* becomes the new label for the button.

- **Handling Buttons:**

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed()** method, which is called when an event occurs. An **ActionEvent** object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button. By default, the action command string is the label of the button. Either the button reference or the action command string can be used to identify the button.

Checkboxes

A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. It is usually used to display a set of options which can be selected independently by the user, or allow multiple selections.There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class. **Checkbox** supports these constructors:

Constructor	Description
Checkbox()	Creates a checkbox with no label i.e. blank label, this checkbox is unchecked by default.
Checkbox(String *str*)	Creates a checkbox whose label is specified by ***str***, this checkbox is unchecked by default.

Checkbox(String str, boolean on)	Creates a checkbox whose label is specified by ***str***, this checkbox is checked or unchecked depending on the boolean value.
Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*)	The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be **null**. The value of *on* determines the initial state of the check box.
Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*)	

Check Box methods

Methods	Description
void setLabel(String *str*)	Sets a String *str* as Checkbox's label.
String getLabel()	Gets the label of the Checkbox.
void setState(boolean b)	Sets a state of check box to the specified state.
boolean getState()	Gets the state of Checkbox whether it is in on or off state.

Handling Check Boxes

Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the `ItemListener` interface. That interface defines the `itemStateChanged()` method. An `ItemEvent` object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).

CheckboxGroup

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called ***radio buttons***. To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group.

Which check box in a group is currently selected is determined by calling `getSelectedCheckbox()`. You can set a check box by calling `setSelectedCheckbox()`. These methods are as follows:

```
Checkbox getSelectedCheckbox()
void setSelectedCheckbox(Checkbox which)
```

Here, *`which`* is the check box that you want to be selected. The previously selected check box will be turned off.

Lists

The `java.awt.List` component, known as a list or listbox, is similar to the Choice component, except it shows multiple items at a time and user is allowed to select either one or multiple items at a time. When the numbers of items in the list exceed the available space, the scrollbar will be displayed automatically. A list can be created by instantiating a List class.

The List is a GUI component used to display a list of text items. It contains a set of String values that the user can choose from. It is a 'list' that allows the user to select one or more options. The programmer has the choice to configure 'single select' or 'multiple select' option of a list. An object of List class generates an ItemEvent object when an item is selected from the list. This event can be handled by implementing the ItemListener interface. Any class implementing this interface can interact with the List object at runtime and handle selecting and unselecting of items from a list. List provides following constructors:

Constructor	Description
`List()`	Creates a new scrolling list and allows only one item to be selected at any one time.
`List(int numRows)`	Creates a new scrolling list and the value of *numRows* specifies the number of entries (lines) in the list that will always be visible (others can be scrolled into view as needed).
`List(int numRows, boolean multipleSelect)`	If *multipleSelect* is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.

List provides following methods:

Methods	Description
`Add(String` *name*`)`	Adds the specified item *name* to the end of scrolling list.
`Add(String` *name*`, int` *Index*`)`	Adds the specified item to the scrolling list at the position indicated by the *index*. Indexing begins at zero. You can specify –1 to add the item to the end of the list.
`String getSelectedItem()`	For lists that allow only single selection, you can determine which item is currently selected by calling `getSelectedItem()`. This method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, null is returned.
`int getSelectedIndex()`	For lists that allow only single selection, you can determine which item is currently selected by calling `getSelectedIndex()`. This returns index of the item. The first item is at index 0. If more than one item is selected, or if no selection has made, –1 is returned.
`String[] getSelectedItems()`	For lists that allow multiple selection, you can use `getSelectedItems()`. This returns an array containing the names of the currently selected items.

`int[] getSelectedIndexes()`	For lists that allow multiple selection, you can use `getSelectedIndexes()`. The returns an array containing the indexes of the currently selected items.
`int getItemCount()`	To obtain the number of items in the list.
`void select(int index)`	To set the currently selected item by using the `select()` method.
`getItem(int` *`index`*`)`	By giving an index, you can obtain the name associated with the item at that index.

Handling Lists

To process list events, you will need to implement the `ActionListener` interface. Each time a **List** item is double-clicked, an `ActionEvent` object is generated. Its `getActionCommand()` method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an `ItemEvent` object is generated. Its `getStateChange()` method can be used to determine whether a selection or deselection triggered this event. `getItemSelectable()` returns a reference to the object that triggered this event.

ScrollBar

Scrollbar class is used to create a horizontal and vertical Scrollbar. A Scrollbar can be added to a top-level container like Frame or a component like Panel. The *Scroll bars* are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scrollbar consists of several individual parts: arrows (the buttons at each end of the scrollbar), a slider box or thumb (scrollable box you slide) and a track (part of the scrollbar you slide the thumb in). The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the **Scrollbar** class.

Scrollbar provides following constructors:

Constructor	Description
`Scrollbar()`	Constructs a new vertical scroll bar.
`Scrollbar(int` *`style`*`)`	Constructs a new scroll bar with the specified orientation by *style*. If *style* is `Scrollbar.VERTICAL`, a vertical scroll bar is created. If *style* is `Scrollbar.HORIZONTAL`, the scroll bar is horizontal.
`Scrollbar(int` *`style`*`, int` *`initialValue`*`, int` *`thumbSize`*`, int` *`min`*`, int` *`max`*`)`	Initial value of the scroll bar is passed in *initialValue*. The number of units represented by the height of the thumb is passed in *thumbSize*. The minimum and maximum values for the scroll bar are specified by *min* and *max*.

Scrollbar provides following methods:

Methods	Description
`void setValues(int value, int visible, int minimum, int maximum)`	Sets the values of four properties for this scroll bar: *value*, *visibleAmount*, *minimum*, and *maximum*.
`void setValue(int newValue)`	Sets the *value* of this scroll bar to the specified value.
`int getValue()`	Gets the current value of this scroll bar.
`int getMaximum()`	Gets the maximum value of this scroll bar.
`int getMinimum()`	Gets the minimum value of this scroll bar.
`int getOrientation()`	Returns the orientation of this scroll bar.
`void setOrientation(int orientation)`	Sets the orientation for this scroll bar.

Handling Scroll Bars

To process scroll bar events, you need to implement the `AdjustmentListener` interface. Each time a user interacts with a scroll bar, an `AdjustmentEvent` object is generated. Its `getAdjustmentType()` method can be used to determine type of adjustment.

TextField

A `TextField` is a component used for displaying, inputting and editing a single line of plain text. The `TextField` can be created by creating an instance of `TextField` class. It is usually called an edit control. Text fields allow the user to edit the text using the arrow keys, cut and paste keys, and mouse selections.

Whenever a key is pressed in a **TextField**, the AWT creates events. It could be either a key pressed event, a key released event, or a key typed event. A *KeyEvent* is passed to the registered *KeyListener*. A **TextField** can also generate an *ActionEvent* whenever the *'enter'* key is pressed. Any class interested in this event needs to implement the *ActionListener* interface.

TextField defines the following constructors:

Constructor	Description
`TextField()`	Constructs a new text field.
`TextField(int numChars)`	Creates a TextField with a specified width.
`TextField(String str)`	Creates a TextField with a specified default text.
`TextField(String str, int numChars)`	Constructs a new text field initialized with the specified text to be displayed, and wide enough to hold the specified number of characters.

TextField defines the following methods:

Methods	Description
`void setText(String str)`	Sets a String *str* on the TextField.
`String getText()`	Gets the string currently contained in the TextField.
`void setEditable(boolean b)`	Sets a Sets a TextField to editable or uneditable.
`void setFont(Font f)`	Sets a font type to the TextField
`setForeground(Color c)`	Sets a foreground color, color of text in TextField.
`setEchoChar()`	Set echo character for the text field.

TextArea

The `TextArea` control in `AWT` provides us multiline editor area. The user can type here as much as he wants. Initially the scroll bar is invisible, when the text in the `TextArea` becomes larger than the viewable area the scroll bar is automatically appears which help us to scroll the text up & down and right & left.

TextArea defines the following constructors:

Constructor	Description
`TextArea()`	Constructs a new text area with the empty string as text.
`TextArea(int numLines, int numChars)`	Constructs a new text area with the specified height (number of lines) and width (number of characters) and the empty string as text.
`TextArea(String str, int numLines, int numChars)`	Constructs a new text area with the specified text, and with the specified number of lines and width.
`TextArea(String str, int numLines, int numChars, int sBars)`	Constructs a new text area with the specified text, and with the lines, width, and scroll bar visibility as specified. `SCROLLBARS_BOTH`, `SCROLLBARS_NONE`, `SCROLLBARS_HORIZONTAL_ONLY`, `SCROLLBARS_VERTICAL_ONLY`

TextArea defines the following methods:

Methods	Description
`void setText(String text)`	Sets a String message on the TextArea.
`String getText()`	Gets a String message of TextArea.
`void append(String text)`	Appends the text to the TextArea.
`setRows(int n)`	Specifies the number of lines of text.
`int getRows()`	Gets the total number of lines in TextArea.
`setColumns()`	Specifies the number of columns of text.
`int getColumns()`	Gets the total number of columns in TextArea.

Layout Manager

Layout Managers are used to arrange different components (Button, Label, TextField etc) within different types of windows (Panel, Applet, Frame etc.). They control the size and position of components within container. The layout manager automatically positions all the components within the container. If we do not use layout manager then also the components are positioned by the default layout manager. It is possible to layout the controls manually but it is very tedious.

Java provides various layout manager to position the controls. The properties like size, shape and arrangement varies from one layout manager to other layout manager. When the size of the applet or the application window changes the size, shape and arrangement of the components also changes in response i.e. the layout managers adapt to the dimensions of appletviewer or the application window.

Layout Manager is an interface that is implemented by all the classes of layout managers. There are following classes that represent the layout managers:

Layout Manager	Description
`BorderLayout`	The borderlayout arranges the components to fit in the five regions: east, west, north, south and center.
`CardLayout`	The CardLayout object treats each component in the container as a card. Only one card is visible at a time.
`FlowLayout`	The FlowLayout is the default layout.It layouts the components in a directional flow.
`GridLayout`	The GridLayout manages the components in form of a rectangular grid.
`GridBagLayout`	This is the most flexible layout manager class.The object of GridBagLayout aligns the component vertically,horizontally or along their baseline without requiring the components of same size.

■ ■ ■ ■ ■

Sant Gadge Baba Amravati University, Amravati
B. Sc. Part THREE (Semester – VI) Examination
Questions Asked in Previous University Exams

- **Summer-2022 (AY-2272)**

6. A) List the different AWT controls and explain how to add and remove controls. 6
   B) Explain Event Delegation model in detail. 6

OR

7. A) List the Event Listener Interfaces with description. 6
   B) Explain Checkboxes and Label AWT control with example. 6

- **Winter 2022 (AC-2131)**

6. A) List the different types of controls supported by AWT. Explain. 6
   B) Explain Adapter Classes with suitable example. 6

OR

7. A) What is event? Explain sources of events with example. 6
   B) Explain List and Labels AWT controls with example. 6

- **Summer 2023 (AD-1909)**

6. A) Explain Java AWT Checkboxes with example. 6
   B) Explain Event Listener Interface. 6

OR

7. A) Explain Delegation model. 6
   B) Explain Java AWT Hierarchy. 6

- **Winter 2023 (AE-1827)**

6. A) Explain event delegation model in detail. 6
   B) Explain the following AWT controls :
   (i) Button
   (ii) List 6

OR

7. A) Explain Adapter Classes with example. 6
   B) List the different AWT controls and explain the steps of adding and removing controls. 6

■ ■ ■ ■ ■