Core JAVA UNIT 3

AWT:

Introduction

•Java AWT (Abstract Window Toolkit) is an API to develop Graphical User Interface (GUI) or windows-based applications in Java.

•Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system.

•AWT is heavy weight i.e. its components are using the resources of underlying operating system (OS).

• The java.awt package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Why AWT is platform independent?

•Java AWT calls the native platform calls the native platform (operating systems) subroutine for creating API components like TextField, ChechBox, button, etc.

• For example, an AWT GUI with components like TextField, label and button will have different look and feel for the different platforms like Windows, MAC OS, and Unix.

• The reason for this is the platforms have different view for their native components and AWT directly calls the native subroutine that creates those components.

•In simple words, an AWT application will look like a windows application in Windows OS whereas it will look like a Mac application in the MAC OS.

Java AWT Hierarchy

• The hierarchy of Java AWT classes are given below.

Components

• All the elements like the button, text fields, scroll bars, etc. are called components. In Java AWT, there are classes for each component as shown in above diagram. In order to place every component in a particular position on a screen, we need to add them to a container.

Container

• The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

• It is basically a screen where the where the components are placed at their specific locations. Thus it contains and controls the layout of components.

Types of containers:

• There are four types of containers in Java AWT:

• Window

• Panel

• Frame

• Dialog

1. Window

• The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window. We need to create an instance of Window class to create this container.

2. Panel

• The Panel is the container that doesn't contain title bar, border or menu bar. It is generic

container for holding the components. It can have other components like button, text field etc. An instance of Panel class creates a container, in which we can add components.

3. 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.

Event-Delegation-Model

• The Delegation Event model is defined to handle events in GUI programming languages.

• The GUI stands for Graphical User Interface, where a user graphically/visually interacts with the system.

• The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user.

• This is known as event handling.Event Processing in Java

•Java support event processing since Java 1.0. It provides support for AWT (Abstract Window

Toolkit), which is an API used to develop the Desktop application.

•In Java 1.0, the AWT was based on inheritance. To catch and process GUI events for a program, it should hold subclass GUI components and override action() or handleEvent() methods.

• The image demonstrates the event processing.

• But, the modern approach for event processing is based on the Delegation Model. It defines a standard and compatible mechanism to generate and process events. In this model, a source generates an event and forwards it to one or more listeners.

• The listener waits until it receives an event. Once it receives the event, it is processed by the listener and returns it. The UI elements are able to delegate the processing of an event to a separate function.

• The key advantage of the Delegation Event Model is that the application logic is completely separated from the interface logic.

•In this model, the listener must be connected with a source to receive the event notifications. Thus, the events will only be received by the listeners who wish to receive them. So, this approach is more convenient than the inheritance-based event model (in Java 1.0).

• In the older model, an event was propagated up the containment until a component

was handled. This needed components to receive events that were not processed, and it took lots of time. The Delegation Event model overcame this issue.

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

1. Events

2. Events Sources

3. Events Listeners

Events

• The Events are the objects that define state change in a source. An event can be generated as a reaction of a user while interacting with GUI elements. Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key, selecting an item from the list, and so on. We can also consider many other user operations as events.

• The Events may also occur that may be not related to user interaction, such as a timer expires, counter exceeded, system failures, or a task is completed, etc. We can define events for any of the applied actions.

Event Sources

• A source is an object that causes and generates an event. It generates an event when the internal state of the object is changed. The sources are allowed to generate several different types of events.

• A source must register a listener to receive notifications for a specific event. Each event contains its registration method. Below is an example:

public void addTypeListener (TypeListener e1)

• From the above syntax, the Type is the name of the event, and e1 is a reference to the event listener. For example, for a keyboard event listener, the method will be called as addKeyListener().

• For the mouse event listener, the method will be called as addMouseMotionListener(). When an event is triggered using the respected source, all the events will be notified to registered listeners and receive the event object.

• This process is known as event multicasting. In few cases, the event notification will only be sent to listeners that register to receive them.

• Some listeners allow only one listener to register. Below is an example:

public void addTypeListener(TypeListener e2) throws

java.util.TooManyListenersException

• From the above syntax, the Type is the name of the event, and e2 is the event listener's reference. When the specified event occurs, it will be notified to the registered listener. This process is known as unicasting events.

• A source should contain a method that unregisters a specific type of event from the listener if not needed. Below is an example of the method that will remove the event from the listener.

public void removeTypeListener(TypeListener e2?)

• From the above syntax, the Type is an event name, and e2 is the reference of the listener. For example, to remove the keyboard listener, the removeKeyListener() method will be called.

• The source provides the methods to add or remove listeners that generate the events. For example, the Component class contains the methods to operate on the different types of events, such as adding or removing them from the listener.

Event Listeners

• An event listener is an object that is invoked when an event triggers. The listeners require two things; first, it must be registered with a source; however, it can be registered with several resources to receive notification about the events. Second, it must implement the methods to receive and process the received notifications.

• The methods that deal with the events are defined in a set of interfaces. These interfaces can be found in the java.awt.event package.

• For example, the MouseMotionListener interface provides two methods when the mouse is dragged and moved. Any object can receive and process these events if it implements the MouseMotionListener interface.

Types of Events

• The events are categories into the following two categories:

1. The Foreground Events:

• The foreground events are those events that require direct interaction of the user. These types of events are generated as a result of user interaction with the GUI component. For example, clicking on a button, mouse movement, pressing a keyboard key, selecting an option from the list, etc.

2. The Background Events :

• The Background events are those events that result from the interaction of the end-user. For example, an Operating system interrupts system failure (Hardware or Software).

• To handle these events, we need an event handling mechanism that provides control over the events and responses.

The Delegation Model

• The Delegation Model is available in Java since Java 1.1. it provides a new

delegation-based event model using AWT to resolve the event problems. It provides a convenient mechanism to support complex Java programs.

Design Goals

• The design goals of the event delegation model are as following:

• It is easy to learn and implement

• It supports a clean separation between application and GUI code.

• It provides robust event handling program code which is less error-prone (strong compile-time checking)

• It is Flexible, can enable different types of application models for event flow and propagation.

• It enables run-time discovery of both the component-generated events as well as observable events.

• It provides support for the backward binary compatibility with the previous model.Event and Listener (Java Event Handling)

• Changing the state of an object is known as an event. For example, click on button, dragging mouse etc.

• The java.awt.event package provides many event classes and Listener interfaces for event handling.Java Event classes and Listener interfaces

Event Classes Listener Interfaces

ActionEvent ActionListener

MouseEvent MouseListener and

MouseMotionListener

MouseWheelEvent MouseWheelListener

KeyEvent KeyListener

ItemEvent ItemListener

TextEvent TextListener

AdjustmentEvent AdjustmentListener

WindowEvent WindowListener

ComponentEvent ComponentListener

ContainerEvent ContainerListener

FocusEvent FocusListener

Registration Methods

• For registering the component with the Listener, many classes provide the registration methods. For example:

1) Button: public void addActionListener(ActionListener a){}

2) MenuItem: public void addActionListener(ActionListener a){}

3) TextField: public void addActionListener(ActionListener a){}

public void addTextListener(TextListener a){}

4) TextArea: public void addTextListener(TextListener a){}

5) Checkbox: public void addItemListener(ItemListener a){}

6) Choice: public void addItemListener(ItemListener a){}

7) List: public void addActionListener(ActionListener a){}

public void addItemListener(ItemListener a){}

Layouts

• The LayoutManagers are used to arrange components in a particular manner. The Java LayoutManagers facilitates us to control the positioning and size of the components in GUI forms.

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

1. java.awt.BorderLayout 2. java.awt.FlowLayout 3. java.awt.GridLayout

4. java.awt.CardLayout 5. java.awt.GridBagLayout 6. javax.swing.BoxLayout

7. javax.swing.GroupLayout 8. javax.swing.ScrollPaneLayout

9. javax.swing.SpringLayout etc.

Java BorderLayout

• The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the default layout of a frame or window. The BorderLayout provides five constants for each region:

public static final int NORTH

public static final int SOUTH

public static final int EAST

public static final int WEST

public static final int CENTER

Constructors of BorderLayout class:

1. BorderLayout(): creates a border layout but with no gaps between the components.

2. BorderLayout(int hgap, int vgap): creates a border layout with the given horizontal and vertical gaps between the components.

• Example of BorderLayout class: Using

BorderLayout() constructor

```
import java.awt.*;

import javax.swing.*;

public class Border

{

JFrame f;

Border()

{

 f = new JFrame();

 JButton b1 = new JButton("NORTH");;

 JButton b2 = new JButton("SOUTH");;

 JButton b3 = new JButton("EAST");;

 JButton b4 = new JButton("WEST");;
```

```
JButton b5 = new JButton("CENTER");;
f.add(b1, BorderLayout.NORTH);
f.add(b2, BorderLayout.SOUTH);
f.add(b3, BorderLayout.EAST);
f.add(b4, BorderLayout.WEST);
f.add(b5, BorderLayout.CENTER);
f.setSize(300, 300);
f.setVisible(true); }
public static void main(String[] args) {
new Border();
}
}
```

Java GridLayout

• The Java GridLayout class is used to arrange the components in a

rectangular grid. One component is displayed in each rectangle.

Constructors of GridLayout class

1. GridLayout(): creates a grid layout with one column per component in a row.

2. GridLayout(int rows, int columns): creates a grid layout with the given rows and columns but no gaps between the components.

3. GridLayout(int rows, int columns, int hgap, int vgap): creates a grid layout with the given rows and columns along with given horizontal and vertical gaps.

• Example of GridLayout class: Using GridLayout() Constructor

• The GridLayout() constructor creates only one row. The following example shows the usage of the parameterless constructor.

```
import java.awt.*;
import javax.swing.*;
public class GridLayoutExample
{
JFrame frameObj;
GridLayoutExample()
```

```
{
frameObj = new JFrame();
JButton btn1 = new JButton("1");
JButton btn2 = new JButton("2");
JButton btn3 = new JButton("3");
JButton btn4 = new JButton("4");
JButton btn5 = new JButton("5");
JButton btn6 = new JButton("6");
JButton btn7 = new JButton("7");
JButton btn8 = new JButton("8");
JButton btn9 = new JButton("9");
frameObj.add(btn1); frameObj.add(btn2); frameObj.add(btn3)
;
frameObj.add(btn4); frameObj.add(btn5); frameObj.add(btn6)
;
frameObj.add(btn7); frameObj.add(btn8); frameObj.add(btn9)
;
// setting the grid layout using the parameterless constructor
frameObj.setLayout(new GridLayout());
frameObj.setSize(300, 300);
frameObj.setVisible(true);
}
public static void main(String argvs[])
{ new GridLayoutExample(); }
}
```

• Example of GridLayout class:

Using GridLayout(int rows, int columns) Constructor

```
import java.awt.*;
import javax.swing.*;
```

```
public class MyGridLayout{
JFrame f;
MyGridLayout(){
 f=new JFrame();
 JButton b1=new JButton("1");
 JButton b2=new JButton("2");
 JButton b3=new JButton("3");
 JButton b4=new JButton("4");
 JButton b5=new JButton("5");
 JButton b6=new JButton("6");
 JButton b7=new JButton("7");
 JButton b8=new JButton("8");
 JButton b9=new JButton("9");
 // adding buttons to the frame
 f.add(b1); f.add(b2); f.add(b3);
 f.add(b4); f.add(b5); f.add(b6);
 f.add(b7); f.add(b8); f.add(b9);
 // setting grid layout of 3 rows and 3 columns
 f.setLayout(new GridLayout(3,3));
 f.setSize(300,300);
 f.setVisible(true);
}
public static void main(String[] args) {
 new MyGridLayout(); }
}
```

Java FlowLayout

• The Java FlowLayout class is used to arrange the components in a line, one after another (in a flow). It is the default layout of the applet or panel.

• Fields of FlowLayout class

public static final int LEFT

public static final int RIGHT

public static final int CENTER

public static final int LEADING

public static final int TRAILING

Constructors of FlowLayout class

1. FlowLayout(): creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.

2. FlowLayout(int align): creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.

3. FlowLayout(int align, int hgap, int vgap): creates a flow layout with the given alignment and the given horizontal and vertical gap.

• Example of FlowLayout class: Using FlowLayout()

constructor

```
import java.awt.*;

import javax.swing.*;

public class FlowLayoutExample

{

JFrame frameObj;

FlowLayoutExample()

{

 frameObj = new JFrame();

 JButton b1 = new JButton("1");

 JButton b2 = new JButton("2");

 JButton b3 = new JButton("3");

 JButton b4 = new JButton("4");

 JButton b5 = new JButton("5");

 JButton b6 = new JButton("6");

 JButton b7 = new JButton("7");

 JButton b8 = new JButton("8");
```

```
 JButton b9 = new JButton("9");
 JButton b10 = new JButton("10");
frameObj.add(b1); frameObj.add(b2); frameObj.add(b3);
frameObj.add(b4);
frameObj.add(b5); frameObj.add(b6); frameObj.add(b7);
frameObj.add(b8);
 frameObj.add(b9); frameObj.add(b10);
 frameObj.setLayout(new FlowLayout());
 frameObj.setSize(300, 300);
 frameObj.setVisible(true);
}
public static void main(String argvs[])
{ new FlowLayoutExample(); }
}
```

Individual components: Label

• The object of the Label class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by a programmer but a user cannot edit it directly.

• It is called a passive control as it does not create any event when it is accessed. To create a label, we need to create the object of Label class.

AWT Label Class Declaration

public class Label extends Component implements Accessible

AWT Label Fields

1. static int LEFT: It specifies that the label should be left justified.

2. static int RIGHT: It specifies that the label should be right justified.

3. static int CENTER: It specifies that the label should be placed in center.

• Label class Constructors • Label Class Methods Constructor Description

Label() It constructs an empty label.

Label(String text) It constructs a label with the given string (left justified by default).

Label(String text, int alignement)

It constructs a label with the specified string and the specified alignment.

Method name Description

void setText(String text)

It sets the texts for label with the specified text.

void setAlignment(int alignment)

It sets the alignment for label with the specified alignment.

String getText() It gets the text of the label

int getAlignment() It gets the current alignment of the label.

void addNotify() It creates the peer for the label.

AccessibleContext

getAccessibleContext()

It gets the Accessible Context associated with the label.protected String

paramString()

It returns the string the state of the label.

Button

• A button is basically a control component with a label that generates an event when pushed. The Button class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

• When we press a button and release it, AWT sends an instance of ActionEvent to that button by calling processEvent on the button.

• The processEvent method of the button receives the all the events, then it passes an action event by calling its own method processActionEvent. This method passes the action event on to action listeners that are interested in the action events generated by the button.

• To perform an action on a button being pressed and released, the ActionListener interface needs to be implemented. The registered new listener can receive events from the button by calling addActionListener method of the button.

• The Java application can use the button's action command as a messaging protocol.

AWT Button Class Declaration

public class Button extends Component implements Accessible

Button Class Constructors

• Following table shows the types of Button class constructors

Constructor Description

Button( ) It constructs a new button with an empty string i.e. it has no label.

Button (String text) It constructs a new button with given string as its label.

- **Button Class Methods**

Method Description

void setText (String text) It sets the string message on the button

String getText() It fetches the String message on the button.

void setLabel (String label) It sets the label of button with the specified string.

String getLabel() It fetches the label of the button.

void addNotify() It creates the peer of the button.

AccessibleContext getAccessibleContext() It fetched the accessible context associated with the button.

void addActionListener(ActionListener l) It adds the specified action listener to get the action events from the button.

String getActionCommand() It returns the command name of the action event fired by the button.

ActionListener[ ] getActionListeners() It returns an array of all the action listeners registered on the button.

T[ ] getListeners(Class listenerType) It returns an array of all the objects currently registered as FooListeners upon this Button.

protected String paramString() It returns the string which represents the state of button.

protected void processActionEvent (ActionEvent e) It process the action events on the button by dispatching them to a registered ActionListener object.

protected void processEvent (AWTEvent e) It process the events on the button

void removeActionListener (ActionListener l) It removes the specified action listener so that it no longer receives action events from the button.

void setActionCommand(String command) It sets the command name for the action event given by the button.

CheckBox

• The Checkbox class is used to create a checkbox. 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".

AWT Checkbox Class Declaration

public class Checkbox extends Component

implements ItemSelectable, Accessible

• **Checkbox Class Constructors**

Constructor Description

Checkbox() It constructs a checkbox with no string as the label.

Checkbox(String label) It constructs a checkbox with the given label.

Checkbox(String label, boolean state) It constructs a checkbox with the given label and sets the given state.

Checkbox(String label, boolean state, CheckboxGroup group)

It constructs a checkbox with the given label, set the given state in the specified checkbox group.

Checkbox(String label, CheckboxGroup group, boolean state)

It constructs a checkbox with the given label, in the given checkbox group and set to the specified state.

Method inherited by Checkbox

• The methods of Checkbox class are inherited by following classes:

1. java.awt.Component

2. java.lang.Object

Method name Description

void addItemListener(ItemListener IL) It adds the given item listener to get the item events from the checkbox.

AccessibleContext getAccessibleContext() It fetches the accessible context of checkbox.

void addNotify() It creates the peer of checkbox.

CheckboxGroup getCheckboxGroup() It determines the group of checkbox.

ItemListener[] getItemListeners() It returns an array of the item listeners registered on checkbox.

String getLabel() It fetched the label of checkbox.

T[] getListeners(Class listenerType) It returns an array of all the objects registered as

FooListeners.

Object[] getSelectedObjects() It returns an array (size 1) containing checkbox label and returns null if checkbox is not selected.

Method name Description

boolean getState() It returns true if the checkbox is on, else returns off.

protected String paramString() It returns a string representing the state of checkbox.

protected void processEvent(AWTEvent e) It processes the event on checkbox.

protected void processItemEvent(ItemEvent e) It process the item events occurring in the checkbox by dispatching them to registered ItemListener object. void removeItemListener(ItemListener l) It removes the specified item listener so that the item

listener doesn't receive item events from the checkbox anymore.void setCheckboxGroup(CheckboxGroup g) It sets the checkbox's group to the given checkbox.

void setLabel(String label) It sets the checkbox's label to the string argument.

void setState(boolean state) It sets the state of checkbox to the specified state.

Choice

• The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

AWT Choice Class Declaration

public class Choice extends Component

implements ItemSelectable, Accessible

Choice Class constructor

Constructor Description

Choice() It constructs a new choice menu.

Methods inherited by class

• The methods of Choice class are inherited by following classes:

1. java.awt.Component

2. java.lang.Object

Method name Description

void add(String item) It adds an item to the choice menu.

void addItemListener(ItemListener l) It adds the item listener that receives item events from the choice menu.

void addNotify() It creates the peer of choice.

AccessibleContext getAccessibleContext() It gets the accessbile context related to the choice.

String getItem(int index) It gets the item (string) at the given index position in the

choice menu.

int getItemCount() It returns the number of items of the choice menu.

ItemListener[] getItemListeners() It returns an array of all item listeners registered on choice.

Method name Description

T[] getListeners(Class listenerType) Returns an array of all the objects currently registered as FooListeners upon this

Choice.

int getSelectedIndex() Returns the index of the currently selected item.

String getSelectedItem() Gets a representation of the current choice as a string.

Object[] getSelectedObjects() Returns an array (length 1) containing the currently selected item.

void insert(String item, int index) Inserts the item into this choice at the specified position.

protected String paramString() Returns a string representing the state of this Choice menu.

protected void processEvent(AWTEvent e) It processes the event on the choice.

protected void processItemEvent (ItemEvent e) Processes item events occurring on this Choice menu by dispatching them to any registered ItemListener objects.

void remove(int position) It removes an item from the choice menu at the given index position.

void remove(String item) It removes the first occurrence of the item from choice menu.

void removeAll() It removes all the items from the choice menu.

void removeItemListener (ItemListener l) It removes the mentioned item listener. Thus is doesn't receive item events from

the choice menu anymore.

void select(int pos) It changes / sets the selected item in the choice menu to the item at given index position.

void select(String str) It changes / sets the selected item in the choice menu to the item whose string value is equal to string specified in the argument.

List

• The object of List class represents a list of text items. With the help of the List class, user can choose either one item or multiple items. It inherits the Component class.

AWT List class Declaration

public class List extends Component

implements ItemSelectable, Accessible

AWT List Class Constructors

Constructor Description

List() It constructs a new scrolling list.

List(int row_num) It constructs a new scrolling list initialized with the given

number of rows visible.

List(int row_num, Boolean multipleMode) It constructs a new scrolling list initialized which displays the given number of rows.

Methods Inherited by the List Class

• The List class methods are inherited by following classes:

1. java.awt.Component

2. java.lang.Object

Method name Description

void add(String item) It adds the specified item into the end of scrolling list.

void add(String item, int index) It adds the specified item into list at the given index position.

void addActionListener(ActionListener l) It adds the specified action listener to receive action events from list.

void addItemListener(ItemListener l) It adds specified item listener to receive item events from list.

void addNotify() It creates peer of list.

void deselect(int index) It deselects the item at given index position.

AccessibleContext

getAccessibleContext() It fetches the accessible context related to the list.

Method name Description

ActionListener[] getActionListeners() It returns an array of action listeners registered on the list.

String getItem(int index) It fetches the item related to given index position.

int getItemCount() It gets the count/number of items in the list.

ItemListener[] getItemListeners() It returns an array of item listeners registered on the list.

String[] getItems() It fetched the items from the list.

Dimension getMinimumSize() It gets the minimum size of a scrolling list.

Dimension getMinimumSize(int rows) It gets the minimum size of a list with given number of rows.

Dimension getPreferredSize() It gets the preferred size of list.

Dimension getPreferredSize(int rows) It gets the preferred size of list with given number of rows.

int getRows() It fetches the count of visible rows in the list.

int getSelectedIndex() It fetches the index of selected item of list.

int[] getSelectedIndexes() It gets the selected indices of the list.

String getSelectedItem() It gets the selected item on the list.

String[] getSelectedItems() It gets the selected items on the list.

Object[] getSelectedObjects() It gets the selected items on scrolling list in array of objects.

int getVisibleIndex() It gets the index of an item which was made visible by method

makeVisible()

void makeVisible(int index) It makes the item at given index visible.

boolean isIndexSelected(int index) It returns true if given item in the list is selected.

boolean isMultipleMode() I t returns the true if list allows multiple selections.

Method name Description

protected String paramString() It returns parameter string representing state of the scrolling list.

protected void processActionEvent(ActionEvent e)It process the action events occurring on list by dispatching them to a

registered ActionListener object.

protected void processEvent(AWTEvent e) It process the events on scrolling list.

protected void processItemEvent(ItemEvent e) It process the item events occurring on list by dispatching them to a registered ItemListener object.

void removeActionListener(ActionListener l) It removes specified action listener. Thus it doesn't receive further action events from the list.

void removeItemListener(ItemListener l) It removes specified item listener. Thus it doesn't receive further action events from the list.

void remove(int position) It removes the item at given index position from the list.

void remove(String item) It removes the first occurrence of an item from list.

void removeAll() It removes all the items from the list.

void replaceItem(String newVal, int index) It replaces the item at the given index in list with the new string specified.

void select(int index) It selects the item at given index in the list.

void setMultipleMode(boolean b) It sets the flag which determines whether the list will allow multiple selection or not.

void removeNotify() It removes the peer of list.

Menu

• The object of MenuItem class adds a simple labeled menu item on menu.

The items used in a menu must belong to the MenuItem or any of its subclass.

• The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

AWT MenuItem class declaration

public class MenuItem extends MenuComponent implements Accessible

AWT Menu class declaration

public class Menu extends MenuItem implements MenuContainer, Accessible

Text Field

• The object of a TextField class is a text component that allows a user to

enter a single line text and edit it. It inherits TextComponent class, which

further inherits Component class.

• When we enter a key in the text field (like key pressed, key released or key typed), the event is sent to TextField. Then the KeyEvent is passed to the registered KeyListener.

• It can also be done using ActionEvent; if the ActionEvent is enabled on the text field, then the ActionEvent may be fired by pressing return key. The event is handled by the ActionListener interface.

AWT TextField Class Declaration

public class TextField extends TextComponent

• TextField Class constructors

Constructor Description

TextField() It constructs a new text field component.

TextField(String text) It constructs a new text field initialized with the given string text to be displayed.

TextField(int columns) It constructs a new textfield (empty) with given

number of columns.

TextField(String text, int columns) It constructs a new text field with the given text and

given number of columns (width).

• **TextField Class Methods**

Method name Description

void addNotify() It creates the peer of text field.

boolean echoCharIsSet() It tells whether text field has character set for echoing or not.

void addActionListener(ActionListener l) It adds the specified action listener to receive action events from the text field.

ActionListener[] getActionListeners() It returns array of all action listeners registered on text field.

AccessibleContext getAccessibleContext() It fetches the accessible context related to the text field.

int getColumns() It fetches the number of columns in text field.

char getEchoChar() It fetches the character that is used for echoing.

protected String paramString() It returns a string representing state of the text field.

protected void processActionEvent(ActionEvent e) It processes action events occurring in the text field by dispatching them to a registered ActionListener object.

protected void processEvent(AWTEvent e) It processes the event on text field.

void removeActionListener(ActionListener l) It removes specified action listener so that it doesn't receive action events anymore.

void setColumns(int columns) It sets the number of columns in text field.

void setEchoChar(char c) It sets the echo character for text field.

void setText(String t) It sets the text presented by this text component to the specified text.

Text Area

• The object of a TextArea class is a multiline region that displays text. It allows the editing of multiple line text. It inherits TextComponent class.

• The text area allows us to type as much text as we want. When the text in the text area becomes larger than the viewable area, the scroll bar appears automatically which helps us to scroll the text up and down, or right and left.

AWT TextArea Class Declaration

public class TextArea extends TextComponent

• **Fields of TextArea Class**

1. static int SCROLLBARS_BOTH - It creates and displays both horizontal and vertical scrollbars.

2. static int SCROLLBARS_HORIZONTAL_ONLY - It creates and displays only the horizontal scrollbar.

3. static int SCROLLBARS_VERTICAL_ONLY - It creates and displays only the vertical scrollbar.

4. static int SCROLLBARS_NONE - It doesn't create or display any scrollbar in the text area.

• Class constructors:

Constructor Description

TextArea() It constructs a new and empty text area with no text in it.

TextArea (int row, int column) It constructs a new text area with specified number of rows and columns and empty string as text.

TextArea (String text) It constructs a new text area and displays the specified text in it.

TextArea (String text, int row, int column) It constructs a new text area with the specified text in the text area and specified number of rows and columns.

TextArea (String text, int row, int column, int scrollbars)

It construcst a new text area with specified text in text area and specified number of rows and columns and visibility.

• **TextArea Class Methods**

Method name Description

void addNotify() It creates a peer of text area.

void append(String str) It appends the specified text to the current text of text area.

AccessibleContext getAccessibleContext() It returns the accessible context related to the text area

int getColumns() It returns the number of columns of text area.

Dimension getMinimumSize() It determines the minimum size of a text area.

Dimension getMinimumSize(int rows, int columns) It determines the minimum size of a text area with the given number of rows and columns.

Dimension getPreferredSize() It determines the preferred size of a text area.

Dimension preferredSize(int rows, int columns) It determines the preferred size of a text area with given number of rows and columns.

int getRows() It returns the number of rows of text area.

int getScrollbarVisibility() It returns an enumerated value that indicates which scroll bars the text area uses.

void insert(String str, int pos) It inserts the specified text at the specified position in this text area.

protected String paramString() It returns a string representing the state of this TextArea.

void replaceRange(String str, int start, int end) It replaces text between the indicated start and end positions with the specified

replacement text.

void setColumns(int columns) It sets the number of columns for this text area.

void setRows(int rows) It sets the number of rows for this text area.

Inner Classes:

Introduction

• In Java, inner class refers to the class that is declared inside class or interface which were mainly introduced, to sum up, same logically relatable classes as Java is purely object-oriented so bringing it closer to the real world.

There are certain advantages associated with inner classes are as follows:

• Making code clean and readable.

• Private methods of the outer class can be accessed, so bringing a new dimension and making it closer to the real world.

• Optimizing the code module.

Syntax of Inner class

```
class Java_Outer_class{
 //code
```

```
class Java_Inner_class{
//code
}
}
```

Advantage of Java inner classes

1. Nested classes represent a particular type of relationship that is it can access all the members (data members and methods) of the outer class, including private.

2. Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.

3. Code Optimization: It requires less code to write.Need of Java Inner class

• Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

• If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

Types of Nested classes

• There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

Type	**Description**
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing an interface or extending class. The java compiler decides its name.
Local Inner Class	A class was created within the method.
Static Nested Class	A static class was created within the class.
Nested Interface	An interface created within class or interface.

Member inner class

• A non-static class that is created inside a class but outside a method is called member inner class. It is also known as a regular inner class. It can be declared with access modifiers like public, default, private, and protected.

Syntax:

```
class Outer{
//code
```

```
 class Inner{
 //code
 }
}
```

Java Member Inner Class Example

• In this example, we are creating a msg() method in the member inner class that is accessing the private data member of the outer class.

```
class TestMemberOuter1{
 private int data=30;
 class Inner{
 void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
 TestMemberOuter1 obj=new TestMemberOuter1();
 TestMemberOuter1.Inner in=obj.new Inner();
 in.msg();
 }
}
```

How to instantiate Member Inner class in Java?

•An object or instance of a member's inner class always exists within an object of its outer class. The new operator is used to create the object of member inner class with slightly different syntax.

• The general form of syntax to create an object of the member inner class is as follows:

Syntax:

OuterClassReference.new MemberInnerClassConstructor();

Example:

obj.new Inner();

•Here, OuterClassReference is the reference of the outer class followed by a dot which is followed by the new operator.

Anonymous inner class

•Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

•In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

1. Class (may be abstract or concrete).

2. Interface

• Java anonymous inner class example using class

```
abstract class Person{
 abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
 Person p=new Person(){
 void eat(){System.out.println("nice fruits");}
 };
 p.eat(); }
}
```

Internal working of given code

```
Person p=new Person(){
void eat(){System.out.println("nice fruits");}
};
```

1. A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.

2. An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type.

Local inner class

•A class i.e., created inside a method, is called local inner class in java. Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause.

• Local Inner classes are not a member of any enclosing classes. They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them.

•However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it.

•If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

Java local inner class example

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
 class Local{
 void msg(){System.out.println(data);}
 }
 Local l=new Local();
 l.msg();
 }
 public static void main(String args[]){
 localInner1 obj=new localInner1();
 obj.display();
 }
}
```

Rules: Local variables can't be private, public, or protected.

1) Local inner class cannot be invoked from outside the method.

2) Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in the local inner class.

Static inner class

•A static class is a class that is created inside a class, is called a static nested class in Java. It cannot access non-static data members and methods. It can be accessed by outer class name.

1. It can access static data members of the outer class, including private.

2. The static nested class cannot access non-static (instance) data members

Java static nested class example with instance method

```
class TestOuter1{
 static int data=30;
 static class Inner{
 void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
 TestOuter1.Inner obj=new TestOuter1.Inner();
 obj.msg();
 }
}
```

• In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of the Outer class because

the nested class is static and static properties, methods, or classes can be accessed without an object.

Collection Framework:

Introduction

• The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

• Java Collections can achieve all the operations that you perform on a data such as searching,

sorting, insertion, manipulation, and deletion.

• Java Collection means a single unit of objects. Java Collection framework provides many interfaces

(Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet,

LinkedHashSet, TreeSet).

What is Collection in Java

• A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java

• It provides readymade architecture.

• It represents a set of classes and interfaces.

• It is optional.

util Package interfaces

• It contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).Hierarchy of Collection Framework

• The java.util package contains all the classes and interfaces for the Collection framework.

List

• List interface in Java is a sub-interface of the Java collections interface. It contains the index-based methods to insert, update, delete, and search the elements.

• It can have duplicate elements also. We can also store the null elements in the list. List preserves the insertion order, it allows positional access and insertion of elements.

• It found in the java.util package. Let's consider an example for a better understanding where you will see how you can add elements by using a list interface in java.

Example:

```
import java.util.*;
public class GFG {
 public static void main(String args[])
 {
 List<String> al = new ArrayList<>();
 al.add("mango");
 al.add("orange");
 al.add("Grapes");
 for (String fruit : al)
 System.out.println(fruit);
 }
}
```

Set

•The Set follows the unordered way and it found in java.util package and extends the collection interface in java.

•Duplicate item will be ignored in Set and it will not print in the final output. Let's consider

an example for a better understanding where you will see how you can add elements

by using a set interface in java. Let's have a look.

Example:

```
import java.util.*;
public class SetExample {
 public static void main(String[] args)
 {
 Set<String> Set = new HashSet<String>();
 Set.add("one");
 Set.add("two");
 Set.add("three");
 Set.add("four");
 Set.add("five");
 System.out.println(Set);
 }
}
```

Map

• The Java Map interface, java.util.Map represents a mapping between a key and a value.

• More specifically, a Java Map can store pairs of keys and values. Each key is linked to a specific value.

• Once stored in a Map, you can later look up the value using just the key.

• Let's consider an example for a better understanding where you will see how you can add elements by using the Map interface in java. Let's have a look.

Example:

```
import java.util.*;
class MapExample {
 public static void main(String args[])
 {
 Map<Integer, String> map = new
HashMap<Integer, String>();
```

```
map.put(100, "Amit");
map.put(101, "Vijay");
map.put(102, "Rahul");
// Elements can traverse in any order
for (Map.Entry m : map.entrySet()) {
System.out.println(m.getKey() + " "
+ m.getValue());
}
}
}
```

List **Set** **Map**

The list interface allows duplicate elements Set does not allow duplicate elements.

The map does not allow duplicate elements

The list maintains insertion order. Set do not maintain any insertion order.

The map also does not maintain any insertion order.

We can add any number of null values. But in set almost only one null value. The map allows a single null key at most and any number of null values.

List implementation classes are Array

List, LinkedList.

Set implementation classes

are HashSet, LinkedHashSet,

and TreeSet.

Map implementation classes

are HashMap, HashTable, TreeMap,

ConcurrentHashMap, and Linked

HashMap.

The list provides get() method to get the element at a specified index.Set does not provide get method to get the elements at a specified index The map does not provide get

method to get the elements at a specified index If you need to access the elements

frequently by using the index then we can use the list If you want to create a collection of

unique elements then we can use set If you want to store the data in the form of key/value pair then we can use the map.To traverse the list elements by using ListIterator.

Iterator can be used traverse the set elements Through keyset, value, and entry set.

List interface & its classes

• The List interface in Java provides a way to store the ordered collection. It is a child interface of Collection. It is an ordered collection of objects in which duplicate values can be stored.

Since List preserves the insertion order, it allows positional access and insertion of elements.

• The List interface is found in java.util package and inherits the Collection interface. It is a factory of the ListIterator interface.

• Through the ListIterator, we can iterate the list in forward and backward directions. The

implementation classes of the List interface are ArrayList, LinkedList, Stack, and Vector.

• ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

Declaration of Java List Interface

public interface List<E> extends Collection<E> ;

Syntax of Java List

• This type of safelist can be defined as:

List<Obj> list = new ArrayList<Obj> ();

Example of Java List

```
import java.util.*;
class GFG {
public static void main(String[] args)
 {
 List<Integer> l1 = new ArrayList<Integer>();
 l1.add(0, 1);
 l1.add(1, 2);
 System.out.println(l1);
 List<Integer> l2 = new ArrayList<Integer>();
 l2.add(1);
 l2.add(2);
```

```
l2.add(3);
l1.addAll(1, l2);
System.out.println(l1);
l1.remove(1);
System.out.println(l1);
System.out.println(l1.get(3));
l1.set(0, 5);
System.out.println(l1);
}
}
```

Operations in a Java List Interface

• Since List is an interface, it can be used only with a class that implements this interface. Now, let's see how to perform a few frequently used operations on the List.

Operation 1: Adding elements to List class using add() method

Operation 2: Updating elements in List class using set() method

Operation 3: Searching for elements using indexOf(), lastIndexOf methods

Operation 4: Removing elements using remove() method

Operation 5: Accessing Elements in List class using get() method

Operation 6: Checking if an element is present in the List class using

contains() method

Methods of the List Interface

Method Description

add(int index, element) This method is used with Java List Interface to add an element at a particular index in the

list. When a single parameter is passed, it simply adds the element at the end of the list.

addAll(int index, Collection collection)

This method is used with List Interface in Java to add all the elements in the given

collection to the list. When a single parameter is passed, it adds all the elements of the

given collection at the end of the list.

size() This method is used with Java List Interface to return the size of the list.

clear() This method is used to remove all the elements in the list. However, the reference of the list created is still stored.

remove(int index) This method removes an element from the specified index. It shifts subsequent

elements(if any) to left and decreases their indexes by 1.remove(element) This method is used with Java List Interface to remove the first occurrence of the given element in the list.

get(int index) This method returns elements at the specified index.

set(int index, element) This method replaces elements at a given index with the new element. This function returns the element which was just replaced by a new element.

Method Description

indexOf(element) This method returns the first occurrence of the given element or -1 if the element is not present in the list.

lastIndexOf(element) This method returns the last occurrence of the given element or -1 if the element is not present in the list.equals(element) This method is used with Java List Interface to compare the equality of the given element with the elements of the list.

hashCode() This method is used with List Interface in Java to return the hashcode value of the given list.

isEmpty() This method is used with Java List Interface to check if the list is empty or not. It returns true if the list is empty, else false.contains(element) This method is used with List Interface in Java to check if the list contains the given element or not. It returns true if the list contains the element.

containsAll(Collection collection)

This method is used with Java List Interface to check if the list contains all the collection of

elements.

sort(Comparator comp) This method is used with List Interface in Java to sort the elements of the list on the basis of the given comparator.

Set interface & its classes

• The set interface is present in java.util package and extends the Collection interface.

• It is an unordered collection of objects in which duplicate values cannot be stored.

• It is an interface that implements the mathematical set.

• This interface contains the methods inherited from the Collection interface and adds a feature that restricts the insertion of the duplicate elements.

• There are two interfaces that extend the set implementation namely SortedSet and NavigableSet.

•In the image, the navigable set extends the sorted set interface.

• Since a set doesn't retain the insertion order, the navigable set interface provides the implementation to navigate through the Set.

• The class which implements the navigable set is a TreeSet which is an implementation of a self-balancing tree.

• Therefore, this interface provides us with a way to navigate through this tree.

The Set interface is declared as:

public interface Set extends Collection

Creating Set Objects

• Since Set is an interface, objects cannot be created of the typeset. We always need a class that extends this list in order to create an object.

•And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Set. This type-safe set can be defined as:

// Obj is the type of the object to be stored in Set

Set<Obj> set = new HashSet<Obj> ();

• Let us discuss methods present in the Set interface provided below in a tabular format below as follows:

Method Description

add(element)

This method is used to add a specific element to the set. The function adds the element only if the specified element is not already present in the set else the function returns False if the element is already present in the Set.

addAll(collection)

This method is used to append all of the elements from the mentioned collection to the existing set. The elements are added randomly without following any specific order.

clear()

This method is used to remove all the elements from the set but not delete the set. The reference for the set still exists.

contains(element)

This method is used to check whether a specific element is present in the Set or not.

containsAll(collection)

This method is used to check whether the set contains all the elements present in the given

collection or not. This method returns true if the set contains all the elements and returns false if any of the elements are missing.

hashCode()

This method is used to get the hashCode value for this instance of the Set. It returns an integer value which is the hashCode value for this instance of the Set.

Method Description

isEmpty() This method is used to check whether the set is empty or not.

iterator() This method is used to return the iterator of the set. The elements from the set are returned in a random order.

remove(element) This method is used to remove the given element from the set. This method returns True if the specified element is present in the Set otherwise it returns False.

removeAll(collection) This method is used to remove all the elements from the collection which are present in the set. This method returns true if this set changed as a result of the call.

retainAll(collection) This method is used to retain all the elements from the set which are mentioned in the given collection. This method returns true if this set changed as a result of the call.

size() This method is used to get the size of the set. This returns an integer value which signifies the number of elements.

toArray() This method is used to form an array of the same elements as that of the Set.

Example:

```
import java.util.*;
 public class GFG {
public static void main(String[] args)
 {
 Set<String> hash_Set = new HashSet<String>();
 hash_Set.add("Geeks");
 hash_Set.add("For");
 hash_Set.add("Geeks");
 hash_Set.add("Example");
```

```
 hash_Set.add("Set");
 // Printing elements of HashSet object
 System.out.println(hash_Set);
 }
}
```

Operations on the Set Interface

• The set interface allows the users to perform the basic mathematical operation on the set. Let's take two arrays to understand these basic operations. Let set1 = [1, 3, 2, 4, 8, 9, 0] and set2 = [1, 3, 7, 5, 4, 0, 7, 5]. Then the possible operations on the sets are:

1. Intersection: This operation returns all the common elements from the given two sets. For the above two sets, the intersection would be:

Intersection = [0, 1, 3, 4]

2. Union: This operation adds all the elements in one set with the other. For the above two sets, the union would be:

Union = [0, 1, 2, 3, 4, 5, 7, 8, 9]

3. Difference: This operation removes all the values present in one set from the other set. For the above two sets, the difference would be:

Difference = [2, 8, 9]

Map interface & its classes

• In Java, Map Interface is present in java.util package represents a mapping between a key and a value. Java Map interface is not a subtype of the Collection interface.

Therefore it behaves a bit differently from the rest of the collection types. A map contains unique keys.

• Maps are perfect to use for key-value association mapping such as dictionaries. The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys. Some common scenarios are as follows:

1. A map of error codes and their descriptions.

2. A map of zip codes and cities.

3. A map of managers and employees. Each manager (key) is

associated with a list of employees (value) he manages.

4. A map of classes and students. Each class (key) is associated

with a list of students (value).

Creating Map Objects

• Since Map is an interface, objects cannot be created of the type map. We always need a class that extends this map in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Map.

Syntax: Defining Type-safe Map

Map hm = new HashMap();

// Obj is the type of the object to be stored in Map

Characteristics of a Map Interface

• A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null values like the HashMap and LinkedHashMap, but some do not like the TreeMap.

• The order of a map depends on the specific implementations. For

example, TreeMap and LinkedHashMap have predictable orders, while HashMap does not.

• There are two interfaces for implementing Map in Java. They are Map and SortedMap,

and three classes: HashMap, TreeMap, and LinkedHashMap.

Methods in Java Map Interface

Method Action Performed

clear() This method is used in Java Map Interface to clear and remove all of the elements or mappings from a specified Map collection.

containsKey(Object)

This method is used in Map Interface in Java to check whether a particular key is being mapped into the Map or not. It takes the key element as a parameter and returns True if that element is mapped in the map.

containsValue(Object)

This method is used in Map Interface to check whether a particular value is being mapped by a single or more than one key in the Map. It takes the value as a parameter and returns True if that value is mapped by any of the keys in the map.

isEmpty() This method is used to check if a map is having any entry for key and value pairs. If no mapping exists, then this returns true.

equals(Object) This method is used in Java Map Interface to check for equality between two maps. It verifies whether the elements of one map passed as a parameter is equal to the elements of this map or not.

get(Object) This method is used to retrieve or fetch the value mapped by a particular key mentioned in the parameter. It returns NULL when the map contains no such mapping for the key.

put(Object, Object) This method is used in Java Map Interface to associate the specified value with the specified key in this map.

putAll(Map) This method is used in Map Interface in Java to copy all of the mappings from the specified map to this map.

hashCode() This method is used in Map Interface to generate a hashCode for the given map containing keys and

values.

• Classes that implement the Map interface are depicted in the below media and described later as follows:

keySet() This method is used in Map Interface to return a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.

remove(Object) This method is used in Map Interface to remove the mapping for a key from this map if it is present in the map.

size() This method is used to return the number of key/value pairs available in the map.

values() This method is used in Java Map Interface to create a collection out of the values of the map. It basically returns a Collection view of the values in the HashMap.

Example:

```
import java.util.*;

public class GFG {

public static void main(String[] args)

 {
```

```
Map<String, Integer> map = new HashMap<>();
map.put("vishal", 10);
map.put("sachin", 30);
map.put("vaibhav", 20);
// Iterating over Map
for (Map.Entry<String, Integer> e : map.entrySet())
// Printing key-value pairs
System.out.println(e.getKey() + " "
+ e.getValue());
}
}
```