UNIT-5

Daemon Threads:

- Any Java thread can be a *daemon* thread.
- Daemon threads are service providers for other threads running in the same process as the daemon thread.
- The run() method for a daemon thread is typically an infinite loop that waits for a service request. When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when only daemon threads remain, there is no other thread for which a daemon thread can provide a service.
- To specify that a thread is a daemon thread, call the setDaemon method with the argument true. To determine if a thread is a daemon thread, use the accessor method isDaemon.

Concepts of Applets

- *Applets* are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a Web document.

- After an applet arrives on the client, it has limited access to resources, so that it can produce an arbitrary multimedia user interface and run complex computations without introducing the risk of viruses or breaching data integrity.
- applets – Java program that runs within a Java-enabled browser, invoked through a —applet|| reference on a web page, dynamically downloaded to the client computer

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
public void paint(Graphics g) {
g.drawString("A Simple Applet", 20, 20);
}
}
```

There are two ways to run an applet:

1. Executing the applet within a Java-compatible Web browser, such as NetscapeNavigator.
2. Using an applet viewer, such as the standard JDK tool, **appletviewer**.

- An appletviewer executes your applet in a window. This is generally the fastest and easiest way to test an applet.
- To execute an applet in a Web browser, you need to write a short HTML text file that contains the appropriate APPLET tag.

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

Differences between applets and applications

- Java can be used to create two types of programs: applications and applets.
- An *application* is a program that runs on your computer, under the operating system of that Computer(i.e an application created by Java is more or less like one created using C or C++).
- When used to create applications, Java is not much different from any other computer language.
- An *applet* is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser.
- An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image, sound file, or video clip.
- The important difference is that an applet is an *intelligent program*, not just an animation or media file(i.e an applet is a program that can react to user input and dynamically change—not just run the same animation or sound over and over
- Applications require main method to execute.
- Applets do not require main method.
- Java's console input is quite limited
- Applets are graphical and window-based.

. **Life cycle of an applet**

- Applets life cycle includes the following methods

1.init( )

2.start( )

3.paint( )

4.stop( )

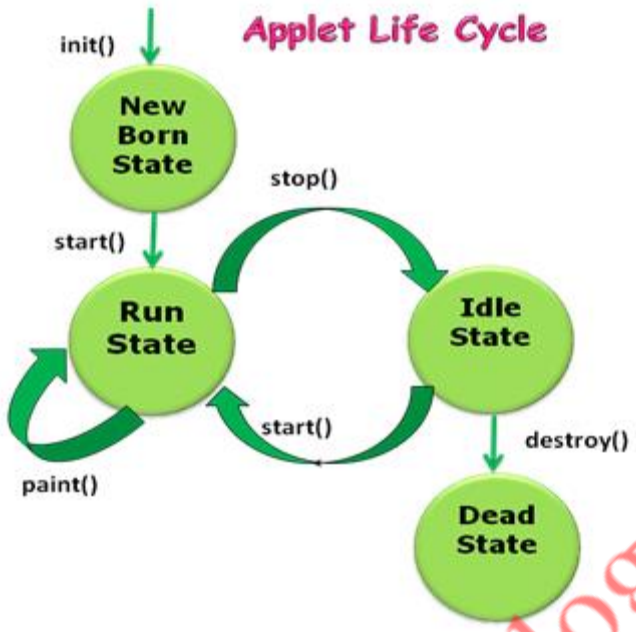
5.destroy( )

- When an applet begins, the AWT calls the following methods, in this sequence:

init( )
start( )
paint( )

- When an applet is terminated, the following sequence of method calls takes place:

stop( )
destroy( )



- **init( ):** The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.
- **start( ):** The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.
- **paint( ):** The **paint( )** method is called each time applet's output must be redrawn. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

- **stop( ):** The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. Applet uses **stop( )** to suspend threads that don't need to run when the applet is not visible. To restart **start( )** is called if the user returns to the page.
- **destroy( ):** The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. The **stop( )** method is always called before **destroy( )**.

public void paint(Graphics g)

- Needed if you do any drawing or painting other than just using standard GUI Components
- Any painting you want to do should be done here, or in a method you call from here
- Painting that you do in other methods may *or may not* happen*Never call* paint(*Graphics*), *call* repaint( )

repaint( ):

- Call repaint( ) when you have changed something and want your changes to show up on the screen
- You do *not* need to call repaint() when something in Java's own components (Buttons, TextFields, etc.)
- You *do* need to call repaint() after drawing commands (drawRect(...), fillRect(...), drawString(...), etc.)
- repaint( ) is a *request*--it might not happen
- When you call repaint( ), Java schedules a call to update(Graphics g)

update( )

- When you call repaint( ), Java schedules a call to update(Graphics g)
- Here's what update does:

```
public void update(Graphics g) {
  // Fills applet with background color, then
  paint(g);
}
```

Simple example of Applet by appletviewer tool:

- To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{

public void paint(Graphics g){
g.drawString("welcome to applet",150,150);
}

}
/*
<applet code="First.class" width="300" height="300">
</applet> */
```

To execute the applet by appletviewer tool, write in command prompt:

```
c:\>javac First.java
c:\>appletviewer First.java
```

Displaying Graphics in Applet

- java.awt.Graphics class provides many methods for graphics programming.

Commonly used methods of Graphics class:

- **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
- **public void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
- **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
- **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
- **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.
- **public abstract void drawLine(int x1, int y1, int x2, int y2**): is used to draw line between the points(x1, y1) and (x2, y2).
- **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.
- **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
- **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
- **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.
- **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

Example of Graphics in applet:

```
import java.applet.Applet;
import java.awt.*;

public class GraphicsDemo extends Applet{

public void paint(Graphics g){
g.setColor(Color.red);
g.drawString("Welcome",50, 50);
g.drawLine(20,30,20,300);
g.drawRect(70,100,30,30);
g.fillRect(170,100,30,30);
```

```
g.drawOval(70,200,30,30);

g.setColor(Color.pink);
g.fillOval(170,200,30,30);
g.drawArc(90,150,30,30,30,270);
g.fillArc(270,150,30,30,0,180);

}
}
```

myapplet.html

```
<html>
<body>
<applet code="GraphicsDemo.class" width="300" height="300">
</applet>
</body>
</html>
```

Two Types of Applets:

- It is important to state at the outset that there are two varieties of applets. The first are those based directly on the Applet class described in this chapter. These applets use the Abstract Window Toolkit (AWT) to provide the graphic user interface (or use no GUI at all). This style of applet has been available since Java was first created.
- The second type of applets are those based on the Swing class JApplet. Swing applets use the Swing classes to provide the GUI. Swing offers a richer and often easier-to-use user interface than does the AWT. Thus, Swing-based applets are now the most popular. However, traditional AWT-based applets are still used, especially when only a very simple user interface is required.

Thus, both AWT- and Swing-based applets are valid.

- Because JApplet inherits Applet, all the features of Applet are also available in JApplet, and most of the information in this chapter applies to both types of applets. Therefore, even if you are interested in only Swing applets, the information in this chapter is still relevant and necessary. Understand, however, that when creating Swing-based applets, some additional constraints apply and these are described later in this topic, when Swing is covered

Event handling

- For the user to interact with a GUI, the underlying operating system must support event handling.
- Operating systems constantly monitor events such as keystrokes, mouse clicks, voice command, etc.
- operating systems sort out these events and report them to the appropriate application programs each application program then decides what to do in response to these events

Events

- An event is an object that describes a state change in a source.
- It can be generated as a consequence of a person interacting with the elements in a graphical user interface.
- Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.
- Events may also occur that are not directly caused by interactions with a user interface.
- For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.
- Events can be defined as needed and appropriate by application.

Event sources

- A source is an object that generates an event.
- This occurs when the internal state of that object changes in some way.
- Sources may generate more than one type of event.
- source must register listeners in order for the listeners to receive notifications about a specific type of event.
- Each type of event has its own registration method.
- General form is:
    - public void addTypeListener(TypeListener el)

Here, Type is the name of the event and el is a reference to the event listener.

For example,

- The method that registers a keyboard event listener is called addKeyListener ().
- The method that registers a mouse motion listener is called addMouseMotionListener ( ).

- When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event.
- In all cases, notifications are sent only to listeners that register to receive them.
- Some sources may allow only one listener to register. The general form is:
    - public void addTypeListener(TypeListener el) throws java.util.TooManyListenersException

  Here Type is the name of the event and el is a reference to the event listener.
- When such an event occurs, the registered listener is notified. This is known as unicasting the event.
- A source must also provide a method that allows a listener to unregister an interest in a specific type of event.
- The general form is:
    - public void removeTypeListener(TypeListener el)

Here, Type is the name of the event and el is a reference to the event listener.

- For example, to remove a keyboard listener, you would call removeKeyListener( ).
  - The methods that add or remove listeners are provided by the source that generates events.
- For example, the Component class provides methods to add and remove keyboard and mouse event listeners.

Event classes

- The Event classes that represent events are at the core of Java's event handling mechanism.
- Super class of the Java event class hierarchy is EventObject, which is in java.util. for all events.
- Constructor is :
  - EventObject(Object src)
    - Here, src is the object that generates this event.
- EventObject contains two methods: getSource( ) and toString( ).
  - The getSource( ) method returns the source of the event. General form is : Object getSource( )
  - The toString( ) returns the string equivalent of the event.
- EventObject is a superclass of all events.
- AWTEvent is a superclass of all AWT events that are handled by the delegation event model.
- The package java.awt.event defines several types of events that are generated by various user interface elements.
- Event Classes in java.awt.event
  - 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 super class 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.
  - 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.

Event Listeners

- A listener is an object that is notified when an event occurs.
- Event has two major requirements.
  - It must have been registered with one or more sources to receive notifications about specific types of events.
  - It must implement methods to receive and process these notifications.
- The methods that receive and process events are defined in a set of interfaces found in java.awt.event.
- For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved.
- Any object may receive and process one or both of these events if it provides an implementation of this interface.

Delegation event model

- The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events.
- Its concept is quite simple: a source generates an event and sends it to one or more listeners.
- In this scheme, the listener simply waits until it receives an event.
- Once received, the listener processes the event and then returns.
- The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.
- A user interface element is able to "delegate“ the processing of an event to a separate piece of code.
- In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.
- This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component.
- This required components to receive events that they did not process, and it wasted valuable time.The delegation event model eliminates this overhead.
- Note:
  - Java also allows you to process events without using the delegation event model.
  - This can be done by extending an AWT component.

Handling mouse events

- mouse events can be handled by implementing the MouseListener and the MouseMotionListener interfaces.
- MouseListener Interface defines five methods. The general forms of these methods are:
  - void mouseClicked(MouseEvent me)
  - void mouseEntered(MouseEvent me)
  - void mouseExited(MouseEvent me)
  - void mousePressed(MouseEvent me)
  - void mouseReleased(MouseEvent me)
- MouseMotionListener Interface. This interface defines two methods. Their general forms are :
  - void mouseDragged(MouseEvent me)
  - void mouseMoved(MouseEvent me)

Handling keyboard events

- Keyboard events, can be handled by implementing the **KeyListener** interface.
- **KeyListner** interface defines three methods. The general forms of these methods are :
  - void keyPressed(KeyEvent ke)
  - void keyReleased(KeyEvent ke)
  - void keyTyped(KeyEvent ke)
- To implement keyboard events implementation to the above methods is needed.

Adapter classes

- Java provides a special feature, called an adapter class that can simplify the creation of event handlers.
- 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.
- Adapter classes in java.awt.event are.

Adapter Class	**Listener Interface**
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener

MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Inner classes

- Inner classes, which allow one class to be defined within another.
- An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.
- An inner class is fully within the scope of its enclosing class.
- an inner class has access to all of the members of its enclosing class, but the reverse is not true.
- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class

The AWT class hierarchy

- The AWT classes are contained in the java.awt package. It is one of Java's largest packages. some of the AWT classes.
- AWT Classes
  - AWTEvent:Encapsulates AWT events.
  - AWTEventMulticaster: Dispatches events to multiple listeners.
  - BorderLayout: The border layout manager. Border layouts use five components: North, South, East, West, and Center.
  - Button: Creates a push button control.
  - Canvas: A blank, semantics-free window.
  - CardLayout: The card layout manager. Card layouts emulate index cards. Only the one on top is showing.
  - Checkbox: Creates a check box control.
  - CheckboxGroup: Creates a group of check box controls.
  - CheckboxMenuItem: Creates an on/off menu item.
  - Choice: Creates a pop-up list.
  - Color: Manages colors in a portable, platform-independent fashion.
  - Component: An abstract super class for various AWT components.
  - Container: A subclass of Component that can hold other components.
  - Cursor: Encapsulates a bitmapped cursor.
  - Dialog: Creates a dialog window.

- Dimension: Specifies the dimensions of an object. The width is stored in width, and the height is stored in height.
- Event: Encapsulates events.
- EventQueue: Queues events.
- FileDialog: Creates a window from which a file can be selected.
- FlowLayout: The flow layout manager. Flow layout positions components left to right, top to bottom.
- Font: Encapsulates a type font.
- FontMetrics: Encapsulates various information related to a font. This information helps you display text in a window.
- Frame: Creates a standard window that has a title bar, resize corners, and a menu bar.
- Graphics: Encapsulates the graphics context. This context is used by various output methods to display output in a window.
- GraphicsDevice: Describes a graphics device such as a screen or printer.
- GraphicsEnvironment: Describes the collection of available Font and GraphicsDevice objects.
- GridBagConstraints: Defines various constraints relating to the GridBagLayout class.
- GridBagLayout: The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by GridBagConstraints.
- GridLayout: The grid layout manager. Grid layout displays components i n a two-dimensional grid.
- Scrollbar: Creates a scroll bar control.
- ScrollPane: A container that provides horizontal and/or vertical scrollbars for another component.
- SystemColor: Contains the colors of GUI widgets such as windows, scrollbars, text, and others.
- TextArea: Creates a multiline edit control.
- TextComponent: A super class for TextArea and TextField.
- TextField: Creates a single-line edit control.
- Toolkit: Abstract class implemented by the AWT.
- Window: Creates a window with no frame, no menu bar, and no title.

User interface components

Labels:

- Creates a label that displays a string.
- A label is an object of type Label, and it contains a string, which it displays.
- Labels are passive controls that do not support any interaction with the user.
- Label defines the following constructors:

- Label( )-- creates a blank label.
- Label(String str)-- creates a label that contains the string specified by str. This string is left-justified
- Label (String str, int how) -- 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

- methods are shown here:
  - void setText(String str)-- specifies the new label
  - String getText( )- -the current label is returned.
  - void setAlignment(int how) set the alignment of the string within the label
  - int getAlignment( )-- obtain the current alignment
- Label creation: Label one = new Label("One");

Button

- The most widely used control is the push button.
- A push button is a component that contains a label and that generates an event when it is pressed.
- Push buttons are objects of type Button. Button defines these two constructors:
- Button( )-- creates an empty button
- Button(String str)-- creates a button that contains str as a label
- methods are as follows:
  - void setLabel(String str)-- set its new label
  - String getLabel( )-- retrieve its label
- Button creation: Button yes = new Button("Yes");

Canvas

- It is not part of the hierarchy for applet or frame windows
- Canvas encapsulates a blank window upon which you can draw.
- Canvas creation:
  - Canvas c = new Canvas();
- Image test = c.createImage(200, 100);-- to actually make an Image object. that image is blank.

Scrollbars

- Scrollbar generates adjustment events when the scroll bar is manipulated.
- Scrollbar creates a scroll bar control.
- Scroll bars are used to select continuous values between a specified minimum and maximum.
- Scroll bars may be oriented horizontally or vertically.
- A scroll bar is actually a composite of several individual parts.
- Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow.

- 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.
- Scrollbar defines the following constructors:
  - Scrollbar( )-- creates a vertical scroll bar
  - Scrollbar(int style)-- to specify the orientation of the scroll bar.
  - Scrollbar(int style, int initialValue, int thumbSize, int min, int max)-- to specify the orientation of the scroll bar.
    - If style is Scrollbar.VERTICAL, a vertical scroll bar is created.
    - If style is Scrollbar.HORIZONTAL, the scroll bar is horizontal
    - The 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.
- vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, height);
- horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, width);

Text

- Text is created by Using a TextField class
- The TextField class implements a single-line text-entry area, usually called an edit control.
- Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.
- TextField is a subclass of TextComponent.
- TextField defines the following constructors:
  - TextField( )-- creates a default text field.
  - TextField(int numChars)-- creates a text field that is numChars characters wide.
  - TextField(String str)- form initializes the text field with the string contained in str.
  - TextField(String str, int numChars)- initializes a text field and sets its width.
- Methods
  - String getText( )- obtain the string currently contained in the text field
  - void setText(String str)- To set the text. str is the new string.

Components

- At the top of the AWT hierarchy is the Component class.
- Component is an abstract class that encapsulates all of the attributes of a visual component.

- All user interface elements that are displayed on the screen and that interact with the user are subclasses of Component.
- It defines public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting.
- A Component object is responsible for remembering the current foreground and background colors and the currently selected text font.
- Component add(Component compObj) -- To add components
  - Here, *compObj* is an instance of the control that you want to add.
  - A reference to *compObj* is returned.
- Once a control has been added, it will automatically be visible whenever its parent window is displayed.
- void remove(Component obj) -To remove a control from a window when the control is no longer needed
  - Here, *obj* is a reference to the control you want to remove.
  - You can remove all controls by calling **removeAll( )**.

Check box

- 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.
- There is a label associated with each check box that describes what option the box represents.
- You can change the state of a check box by clicking on it.
- Check boxes can be used individually or as part of a group.
- Checkboxes are objects of the Checkbox class.

- Checkbox supports these constructors:
  - Checkbox( )-
    - creates a check box whose label is initially blank.
    - The state of the check box is unchecked.
  - Checkbox(String str)- creates a check box whose label is specified by str. The state of the check
  - Checkbox(String str, boolean on)-
    - allows you to set the initial state of the check box.
    - If on is true, the check box is initially checked; otherwise, it is cleared.
  - Checkbox(String str, boolean on, CheckboxGroup cbGroup)
  - Checkbox(String str, CheckboxGroup cbGroup, boolean on)

        - 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.
- methods are as follows:
    - boolean getState( )- To retrieve the current state of a check box
    - void setState(boolean on)- To set its state
        - Here, if on is true, the box is checked. If it is false, the box is cleared.
    - String getLabel( )-- To retrieve the current label
    - void setLabel(String str)-- To set the label
- Checkbox creation:
    - CheckBox Win98 = new Checkbox("Windows 98", null, true);

Check box groups

- 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.
- methods
    - Checkbox getSelectedCheckbox( ) - determine which check box in a group is currently selected
    - void setSelectedCheckbox(Checkbox which) - set a check box
        - Here, which is the check box that you want to be selected.
        - The previously selected checkbox will be turned off.
- CheckboxGroup cbg = new CheckboxGroup();
- Win98 = new Checkbox("Windows 98", cbg, true);
- winNT = new Checkbox("Windows NT", cbg, false);

Choices

- The Choice class is used to create a pop-up list of items from which the user may choose.
- A Choice control is a form of menu.
- Choice only defines the default constructor, which creates an empty list.
- Methods:
    - void addItem(String name)- add a selection to the list
    - void add(String name)- add a selection to the list
    - Here, name is the name of the item being added.

- String getSelectedItem( )- determine which item is currently selected,
- int getSelectedIndex( ) determine which item is currently selected,

Lists

- The List class provides a compact, multiple-choice, scrolling selection list.
- List object can be constructed to show any number of choices in the visible window.
- It can also be created to allow multiple selections.
- List provides these constructors:
  - List( )
  - List(int numRows)
  - List(int numRows, boolean multipleSelect)
- Methods
  - void add(String name) - add a selection to the list
  - void add(String name, int index) - add a selection to the list
  - •Ex: List os = new List(4, true);

Panels

- Panel is a window that does not contain a title bar, menu bar, or border.
- The Panel class is a concrete subclass of Container.
- It doesn't add any new methods; it simply implements Container.
- A Panel may be thought of as a recursively nestable, concrete screen component.
- Panel is the superclass for Applet.
- Methods
- When screen output is directed to an applet, it is drawn on the surface of a Panel object.
- Methods(inherited)
  - add( ) --Components can be added to a Panel object.
  - setLocation( ), setSize( ), or setBounds( )-- position and resize them
- Ex: Panel osCards = new Panel();
- CardLayout cardLO = new CardLayout();
- osCards.setLayout(cardLO);

Scroll pane

- A scroll pane is a component that presents a rectangular area in which a component may be viewed.
- Horizontal and/or vertical scroll bars may be provided if necessary.
- Constants are defined by the ScrollPaneConstants interface.
  - HORIZONTAL_SCROLLBAR_ALWAYS
  - HORIZONTAL_SCROLLBAR_AS_NEEDED
  - VERTICAL_SCROLLBAR_ALWAYS
  - VERTICAL_SCROLLBAR_AS_NEEDED

Dialogs

- Dialog class creates a dialog window.
- constructors are :
  - Dialog(Frame parentWindow, boolean mode)

- Dialog(Frame parentWindow, String title, boolean mode)

- The dialog box allows you to choose a method that should be invoked when the button is clicked.
- Ex: Font f = new Font("Dialog", Font.PLAIN, 12);

Menu bar

- Menu Bar class creates a menu bar.
- A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu.
- To create a menu bar, first create an instance of Menu Bar. By its default constructor.
- Create instances of Menu that will define the selections displayed on the bar.
- Following are the constructors for Menu:
  - Menu( )
  - Menu(String optionName)
  - Menu(String optionName, boolean removable)
- Methods
  - MenuItem add(MenuItem item) -- add the item to a Menu
  - Here, item is the item being added. Items are added to a menu in the order in which the calls to add( ) take place.
  - Menu add(Menu menu) -- add menu object to the menu bar

Graphics

- The AWT supports a rich assortment of graphics methods.
- All graphics are drawn relative to a window.
- A graphics context is encapsulated by the Graphics class
- It is passed to an applet when one of its various methods, such as paint( ) or update( ), is called.
- It is returned by the getGraphics( ) method of Component.
- The Graphics class defines a number of drawing functions. Each shape can be drawn edge-only or filled.
- Objects are drawn and filled in the currently selected graphics color, which is black by default.
- When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped
- Ex:

```
Public void paint(Graphics g)
{
G.drawString("welcome",20,20);
}
```

Layout manager

- A layout manager automatically arranges your controls within a window by using some type of algorithm.

- It is very tedious to manually lay out a large number of components and sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized.
- Each **Container** object has a layout manager associated with it.
- A layout manager is an instance of any class that implements the **LayoutManager** interface.
- The layout manager is set by the **setLayout( )** method. If no call to **setLayout ( )** is made, then the default layout manager is used.
- Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.
- Different types of layout managers
    - Border Layout
    - Grid Layout
    - Flow Layout
    - Card Layout
    - GridBag Layout

Border layout

- The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center.
- The four sides are referred to as north, south, east, and west. The middle area is called the center.
- The constructors defined by **BorderLayout**:
    - BorderLayout( )
    - BorderLayout(int horz, int vert)
- **BorderLayout** defines the following constants that specify the regions:
    - BorderLayout.CENTER
    - B orderLayout.SOUTH
    - BorderLayout. EAST
    - BorderLayout.WEST
    - BorderLayout. NORTH
- void add(Component compObj, Object region)—to add the components

Grid layout

- **GridLayout** lays out components in a two-dimensional grid. When you instantiate a
- **GridLayout**, you define the number of rows and columns.
- The constructors are
    - GridLayout( ) - creates a single-column grid layout.
    - GridLayout(int numRows, int numColumns)- creates a grid layout with the specified number of rows and columns
    - GridLayout(int numRows, int numColumns, int horz, int vert)
        - *horz* and *vert*, specifies the horizontal and vertical space left between components

- Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-lengthrows.

Flow layout

- **FlowLayout** is the default layout manager.
- Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right.
- The constructors are
- FlowLayout( ) - creates the default layout, which centers components and leaves five pixels of space between each component.
- FlowLayout(int *how)* –
  - *how* specifies that how each line is aligned.
  - Valid values for *how* are:
    - FlowLayout.LEFT
    - FlowLayout.CENTER
    - FlowLayout.RIGHT
- FlowLayout(int *how*, int *horz,* int *vert*)-specifies the horizontal and vertical space left between components in *horz* and *vert,* respectively

Card layout

- The **CardLayout** class is unique among the other layout managers in that it stores several different layouts.
- Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time.
- **CardLayout** provides these two constructors:
  - CardLayout( )
  - CardLayout(int horz, int vert)
- The cards are held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager.
- void add(Component panelObj, Object name); -- Cards are added to panel
- methods defined by **CardLayout**:
  - void first(Container deck)
  - void last(Container deck)
  - void next(Container deck)
  - void previous(Container deck)
  - void show(Container deck, String cardName)

GridBag Layout

- The Grid bag layout displays components subject to the constraints specified by GridBagConstraints.
- **GridLayout** lays out components in a two-dimensional grid.
- The constructors are
  - GridLayout( )
  - GridLayout(int numRows, int numColumns)

- GridLayout(int numRows, int numColumns, int horz, int vert)