UNIT 1: ABSTRACT WINDOW TOOLKIT

Unit Structure

1.0 Learning Objectives

After learning this unit, you will be able to:

- Define window fundamentals.
- Explain working with graphics and controls.
- State layout managers and event handling.
- Describe adapter classes, inner classes and anonymous inner classes.
- Discuss applet fundamentals and applet lifecycle.

1.1 Introduction

The Abstract Window Toolkit (AWT) is Java's original platform-independent windowing, graphics and user-interface widget toolkit. The AWT is now part of the Java Foundation Classes (JFC) the standard API for providing a graphical user interface (GUI) for a Java program.

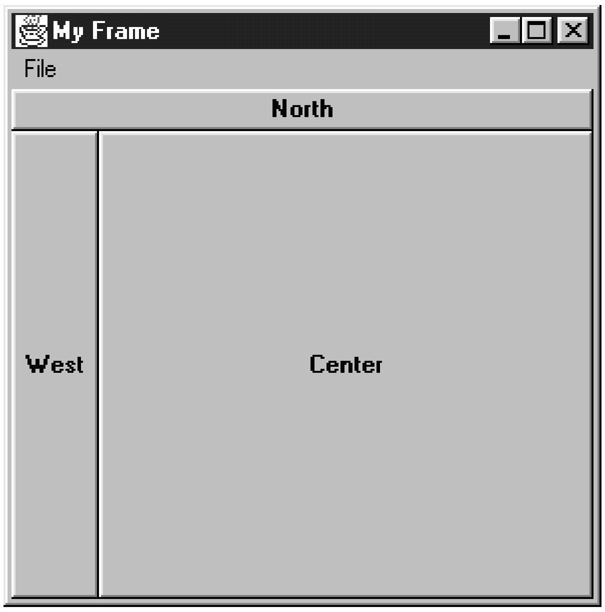
AWT is also the GUI toolkit for a number of Java ME profiles. For example, Connected Device Configuration profiles require Java runtimes on mobile telephones to support AWT.

1.2 Window Fundamentals

A top level window on the screen with no borders or menu bar is the provision of a window. Among various other provisions, it gives a way to implement pop-up messages. The default layout for a Window is Border Layout.

Working with Frames:

A window which contains all the enhancements of the window manager such as borders, window title and window minimize/maximize/close functionality is called a frame. A frame can include a menu bar as well. The default layout of a frame is the Border Layout since it subclasses a window. The very basic building block of a screen-oriented application is provided by frames allowing you to change the mouse cursor, have menus and set icon images. The figure below is an example of a Frame.



A frame

A frame can be created with the help of a program in the Java AWT package. A frame in Java is similar to the main window into which components can be added or joined in order to develop an application. The Frame class represents the top-level windows in Java AWT. The feel and adornments of a frame are supported by Java. You must provide standalone for creating java application at the same time GUI to the user.

Generally, in order to create a frame the most familiar method used is the single string argument constructor of the frame class that contains a single string argument with just the title of the window frame. Later, a user interface can be added by constructing and joining various components to the container.

The program illustrated below shall construct a label on the message frame saying “—you are Welcome to the world of Java.—“The Label. CENTER defines the center alignment of the label. After creating the frame it needs to be visualized by set Visible (true) method as it is initially invisible.

add(lbl):

This method has been used to add the label to the frame. Method add() adds a component to it's container.

setSize (width, height):

This is the method of the Frame class that sets the size of the frame or window. This method takes two arguments width (int), height (int).

setVisible(boolean):
This method of the Frame class sets the visibility of the frame. The frame will be invisible if you pass the boolean value false otherwise the frame will be visible.

Here is the code of the progam :

```
import java.awt.*

public class AwtFrame{
public static void main(String[] args){

Frame frm = new Frame("Java AWT Frame")

Label lbl = new Label("—you are Welcome to the world of Java.--
",Label.CENTER)

frm.add(lbl)
frm.setSize(400,400)
frm.setVisible(true)
}
}
```

Check your progress 1

1. What is a frame?
2. Write the use of addlbl().

....

....

....

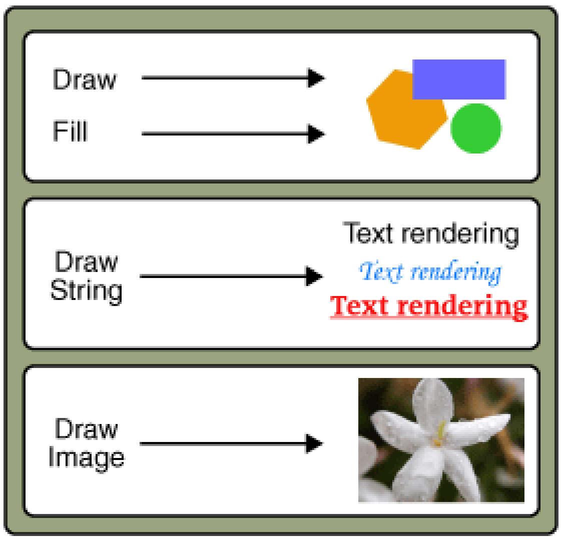
....

....

....

....

1.3 Working with Graphics



Java provides numerous primitives for drawing lines, squares, circles, polygons and images. The Font, Font Metrics, Color and System Color classes provide the ability to alter the displayed output. With the Font class, you adjust how the displayed text will appear. With Font Metrics, you can find out how large the output will be for the specific system the user is using. You could use the Color class to set the color of the text and graphics.

Whereas AWT also includes a number of classes that support more complex graphics manipulations: displaying images, generating images in memory and transforming images.

The Graphics class provides the means to access various graphics devises and is an abstract class. This class enables you to display images and draw on the screen. Since working with graphics requires a comprehensive knowledge of the platform that the program is run on it makes graphics an abstract class. Concrete classes that are closely tied to a particular platform are the ones which do the actual work. These concrete classes are provided by your Java Virtual Machine vendor.

You can be confident that the platform specific classes are going to work accurately whenever you run your program, since you can call all the methods of the Graphics class once you have a graphics object, leaving the worry about platform specific classes.

You rarely need to create a Graphics object yourself; its constructor is protected and is only called by the subclasses that extend Graphics. How then do you get a Graphics object to work with? The sole parameter of the Component.paint () and Component.update() methods is the current graphics context. Therefore, a Graphics object is always available when you override a component’s paint () and update() methods.

You can ask for the graphics context of a Component by calling Component.getGraphics(). However, many components do not have a drawable graphics context. Canvas and Container objects return a valid Graphics object; whether or not any other component has a drawable graphics context depends on the run-time environment.

Check your progress 2

1. Why graphics is considered an abstract class?
2. Give the use of canvas and container objects.

....

....

....

....

....

....

....

....

....

....

....

....

....

1.4 Controls

Controls

Under this section you shall be informed about some components of Java AWT which are available in the Java AWT package for developing the user interface for your program.

1. **Labels -** the simplest component of the Java Abstract Window Toolkit is a label. Label does not perform any type of actions and is basically used to show a text or string in your application. Syntax for defining the label only and with justification:

   Label label_name = new Label ("This is the label text for the day")

   Above code simply represents the text for the label.

   Label label_name = new Label ("This is the label text for the day"

   Label.CENTER);

   The Justification of label can be right or left, centered. The Above declaration used the center justification of the label using the Label. CENTER.

2. **Buttons -** are utilized in generating actions and various other events required for an application. These are components of the Java Abstract Window Toolkit. The syntax of defining the button is as follows:

   Button button_name = new Button ("See this is the label of the button.")

   The Button.setLabel (String) and Button.getLabel() method can be used to change the Button’s label or text. Buttons are added to its container using the add (button_name) method.

3. **Check Boxes -** allow you to create check boxes in applications. The syntax of the definition of Checkbox is as follows:

   CheckBox checkbox_name = new Checkbox ("Optional check box 1", false)

   The above code constructs the unchecked Checkbox by passing the boolean valued argument false with the Checkbox label through the Check box() constructor.

   Defined Checkbox is added to it's container using add (checkbox_name) method. You can change and get the checkbox's label using the setLabel (String) and getLabel() method. You can also set and get the state of the checkbox using the setState(boolean) and getState() method provided by the Checkbox class.

4. **Radio Button -** proves to be a special case of the Java AWT packages under the checkbox component where just a single checkbox from a group of checkbox can be selected at a time.

   Syntax for creating radio buttons is as follows :

   CheckboxGroup chkgp = new CheckboxGroup()

   add (new Checkbox ("1 One", chkgp, false)

   add (new Checkbox ("2 Two", chkgp, false)

   add (new Checkbox ("3 Three",chkgp, false)

   In the above code we are making three check boxes with the label "1 One", "2 Two" and "3 Three". If you mention more than one true valued for checkboxes then your program takes the last true and shows the last check box as checked.

5. **Text Area -** is the text container component of the Java AWT package. The Text Area contains plain text. TextArea can be declared as follows:

   TextArea txtArea_name = new TextArea()

   Now you can make the Text Area very much editable or not using the setEditable (boolean) method. If you pass the Boolean valued argument false then the text area will be non-editable otherwise it will be editable. The text area is by default in an editable mode. Text are set in the text area using the setText(string) method of the TextArea class.

6. **Text Field:** being a text container component of the Java AWT package, the text field component has single line and limited text information. This is declared as follows:

   TextField txtfield = new TextField(20)

   By specifying the number in the constructor you will be able to fix the number of columns in the text field. In the above code we have fixed the number of columns to 20.

Check your progress 3

1. Write a note on button.
2. Explain the text area component of the AWT package.

....

....

....

....

....

....

....

....

....

1.5 Understanding Layout Managers

All Containers, by default, have a layout manager; an object that implements the Layout Manager interface. If a Container's default layout manager doesn't go well with your needs, you can easily replace it with another one. The AWT provide/ supplies layout managers that range from the very simple (FlowLayout and GridLayout) to the special purpose (BorderLayout and CardLayout) to the ultra-flexible (GridBagLayout).

General Rules for Using Layout Managers

You have to openly tell a container not to use a layout manager, it is linked with its own occasion of a layout manager. This layout manager is automatically/by default consulted every time the Container might need to change its appearance. Most layout managers do not require programs to directly call their respective methods.

How to Choose a Layout Manager /Way

Layout managers provided by AWT inherit various strengths and weaknesses. Under this section, common layout scenarios have been discussed and exactly which AWT layout managers might work for each situation and scenario has also been given. You can feel free to use layout managers contributed to the net, such as Packer Layout if you fail to feel that none of the AWT layout managers is suitable for your situation.

How to Create a Layout Manager and Associate It with a Container

A default layout manager is associated with each container. Panels including Applets are initialized to use a Flow Layout whereas windows (apart from special purpose ones such as File Dialog) are initialized to use a Border Layout.

You don’t have to worry about doing a thing to use a Container’s default layout manager since each Container’s constructor creates a layout manager instance and initializes the Container to use it.

You must create an instance (example) of the layout manager class that you require and tell the Container to use it in order to use a non default layout manager. A basic code that does this is mentioned below. This code creates a Card Layout manager and sets it up as the layout manager for a Container.

Container. set Layout (new Card Layout ())

Rules Of Thumb for Using Layout Managers

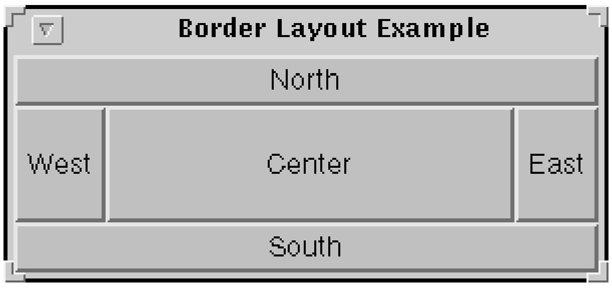
The Container methods that result in calls to the Container's layout manager are add (), remove(), removeAll(), layout(), preferredSize() and minimumSize(). The add (), remove() and removeAll() methods add and remove Components from a Container; you can call them at any time. The layout() method, is called as the result of any paint request to a Container, requests that the Container place and size itself and the components it contains; you don't usually call it directly.

The preferredSize() and minimumSize() methods return the Container's ideal size and minimum size, respectively. Until your program does not enforce these sizes the values returned are merely hints.

You must take special care when calling a Container's preferred Size() and minimum Size() methods. Unless the container and its components have just peer objects the values returned by the methods are meaningless.

Border Layout:

The default layout manager for all Windows is Border Layout, including Frames and Dialogs. North, south, east, west and center are the five areas it uses to hold components. . All extra space is placed in the center area. Given below is an applet that places one button in each area.



As the above applet shows, a Border Layout has five areas: north, south, east, west and center. You see that the center area gets the maximum of the new space that is available (likely) to it on enlarging the window. Whereas, with the other areas it is different since they expand only up to that space which is necessary to fill it up.

Below is the code that creates the Border Layout and the components it manages.

The program runs either within an applet, with the help of the Applet Button, or as an application. The first line shown below is actually unnecessary (not required) for this example, because it's in a Window subclass and each Window already has an associated Border Layout instance. However, the first line would be necessary if the code were in a Panel instead of a Window.

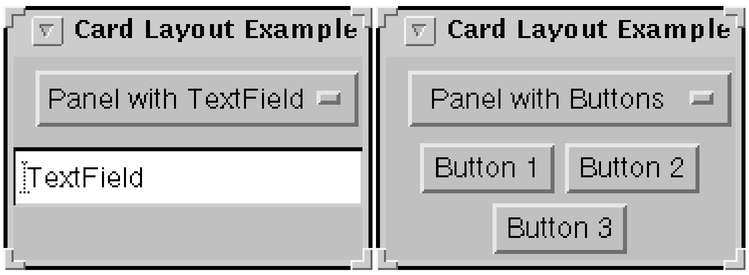
```
setLayout(new BorderLayout())

setFont(new Font("Helvetica", Font.PLAIN, 14))

add("-North-", new Button("-North-"))

add("-South-", new Button("-South-"))

add("-East-", new Button("-East-"))

add("-West-", new Button("-West-"))

add("-Cente-r", new Button("-Center-"))
```

By default, a Border Layout puts no space between the components it manages. In the applet above, any apparent gaps (space) are the result of the Buttons reserving extra space around their apparent display area. You can specify gaps (in pixels) using the following constructor:

```
Public BorderLayout(int horizontalGap, int verticalGap)
```

Card Layout:

Let us Use the Card Layout class when you have an area that can contain different components at different times. Card Layouts are often controlled by Choices, with the state of the option (choice) determining which Panel (group of components) the Card Layout displays. Here's an applet that uses a Choice (option) and Card Layout in this way.



As illustrated in the applet above, the Card Layout class helps in managing two or more components (usually Panel instances) sharing the same display space. Each component managed by a Card Layout is hypothetically like a playing card or a trading card in a stack, with only the top card being visible at any one time. The card being displayed in any of the following manners can be chosen:

- By means of asking for either the first or last card, in the order they were added to the container.
- By means of flipping through the deck backwards or forwards.
- By means of specifying a card with a specific name.

The example program uses this scheme. The user can specifically choose (opt) a card (component) through selecting a name from a pop up list.

Below is the code that creates the Card Layout and the components it manages.

The program runs either within an applet, with the help of Applet Button, or as an application.)

```
//Where instance variables are declared:

Panel cards;

final static String BUTTONPANEL = "-Panel with Buttons-"

final static String TEXTPANEL = "-Panel with TextField-"


//Where the container is initialised

cards = new Panel();

cards.setLayout (new CardLayout())


...//-reate a Panel named p1. Put buttons in it.
```

.../-Create a Panel named p2. Put a text field in it.

cards.add(BUTTONPANEL, p1)

cards.add(TEXTPANEL, p2);

Adding a component to a container managed by a CardLayout uses the two-argument form of the Container add() method: add(String name, Component comp). Any string identifying the component that is added can be the first argument (case).

Below are all the Card Layout methods that let you choose(opt) a component. For each method, the first argument(case) is the container for which the Card Layout is the layout manager (the container of the cards the Card Layout controls).

public void first(-Container parent-)

public void next(-Container parent-)

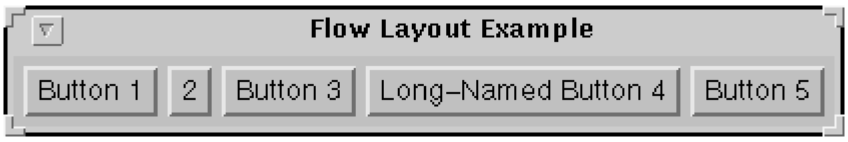
public void previous(-Container parent-)

public void last(-Container parent-)

public void show(-Container parent, String name-)

Flow Layout:

Flow Layout is the default (in any case) layout manager for all Panels. It simply lays out components from left to right, starting new rows if necessary.



Above shown the applet shows, Flow Layout puts components in a row, sized at their preferred size. If the horizontal space (gap) in the container is too small to put all the components in one row, Flow Layout uses multiple rows. Within each row, components are centered (the default), left-aligned, or right-aligned as specified when the Flow Layout is created.

You can see below is the code that creates the Flow Layout and the components it manages. The program runs either within an applet, with the help of Applet Button, or as an application.)

```
Set Layout (new Flow Layout ())

Set Font (new Font ("Helvetica", Font. PLAIN, 14))

Add (new Button ("Button -1-"))

Add (new Button ("2"))

Add (new Button ("Button -3-"))

Add (new Button ("Long-Named Button 4"))

Add (new Button ("Button 5"))
```

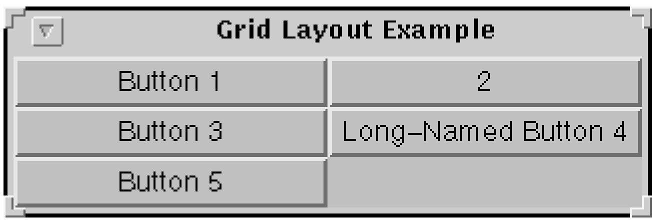
The Flow Layout class has three constructors:

```
public Flow Layout()

public Flow Layout(int alignment)

public Flow Layout(int alignment, int horizontal Gap, int vertical Gap)
```

The alignment argument must have the value Flow Layout. LEFT, Flow Layout. CENTER, or Flow Layout. RIGHT. The horizontal Gap and vertical Gap arguments specify the number of pixels to put between components. If you don't specify a gap value, Flow Layout acts as if you specified 5 for the gap value.

Grid Layout:

Grid Layouts simply make a bunch of Components have equal size, displaying them in the requested number of rows and columns. Here's an applet that uses a Grid Layout to control the display of five buttons.



Illustrated in the above applet is a Grid Layout placing its components in a grid of cells. Each component takes all the available space within its cell and each cell is exactly the same size. Given the space available to the container, you'll notice that the Grid Layout will change the size of cells so that they are as large as possible once you resize the Grid Layout window.

Below is the code that creates the Grid Layout and the components it manages.

The constructor tells the Grid Layout class to create an instance that has two columns and as many rows as necessary. It's one of two constructors for Grid Layout. Here are the declarations for both constructors:

public Grid Layout(int rows, int columns)

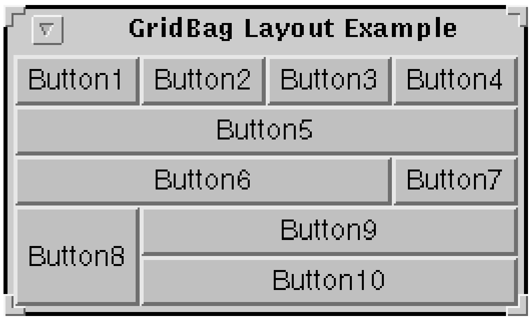
public Grid Layout(int rows, int columns,

int horizontalGap, int verticalGap)

Among all the arguments at least one must be non zero. The numbers of pixels among the cells are specified with the help of the horizontal Gap and vertical Gap arguments. The values shall default to zero if the gaps are not specified. (In the applet above, any apparent gaps are the result of the Buttons reserving extra space around their apparent display area.)

Grid Bag Layout:

Grid Bag Layout is the most sophisticated, flexible layout manager the AWT provides. It aligns components by placing them within a grid of cells, allowing some components to span more than one cell. The rows in the grid are not necessarily all the same height; similarly, grid columns can have different widths. Here is an applet that uses a Grid Bag Layout to manage ten buttons in a panel.



This layout is the most flexible yet complex layout manager that the AWT provides. .The Grid Bag Layout, as illustrated in the applet above places the components in a grid of rows and columns, allowing certain components to span multiple rows or columns. . It is not necessary for each row to have the same height just as each column does not have the same width. Essentially, Grid Bag Layout places components in squares (cells) in a grid and then uses the components' preferred sizes to determine how big the cells should be.

You notice that once you enlarge the window brought up by the applet, it results in the last row getting all the new available vertical space whereas the horizontal space gets evenly distributed among all the columns. The weight that the applet assigns to individual components in the Grid Bag Layout determines the resizing behavior. Notice that each component takes the maximum space it can and this is also specified by the applet.

By specifying the constraints for each component an applet specifies their size and characteristics. To specify constraints, you set instance variables in a Grid Bag Constraints object and tell the Grid Bag Layout (with the set Constraints() method) to associate the constraints with the component.

Check your progress 4

1. Give the general rules for using Layout Manager.
2. Explain Card Layout.

....

....

....

....

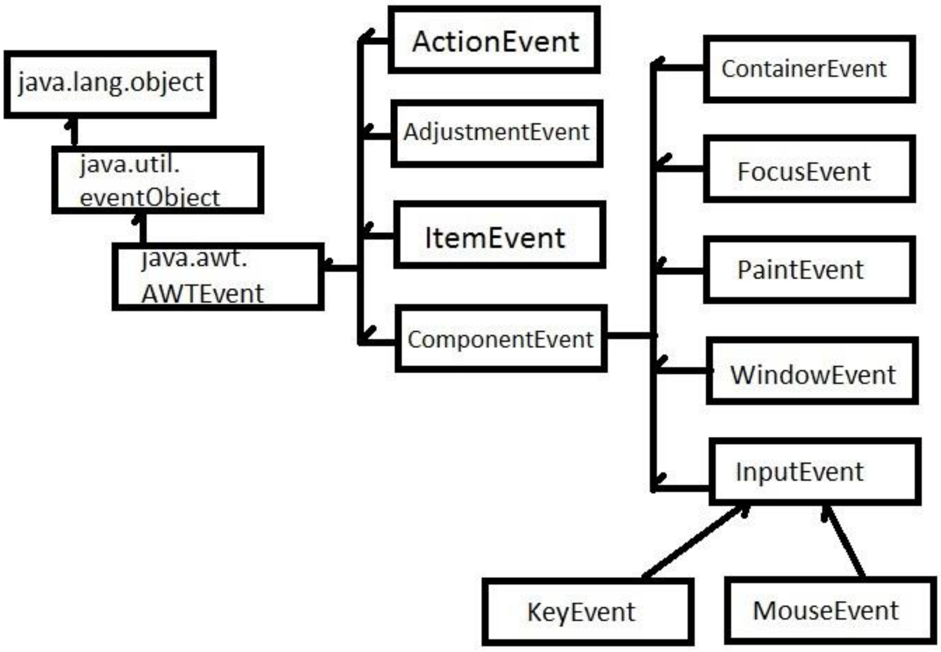
....

....

....

....

1.6 Event Handling



This section helps you learn handling events in Java AWT. Events are referred to as very intrinsic segments of the Java platform. The example below gives you a view of the concepts related to event handling and the methods which can be used to implement an application driven by events.

The objects register themselves as listeners for any event to occur. No event takes place if there is no listener, that is, nothing happens when an event takes place if there is no listener. Every listener has the capability of processing an event irrespective of how many listeners there are. For example, a Simple Button Event applet registers itself as the listener for the button's action events that creates a Button instance.

Any class including an applet can implement an Action Listener. You must always keep in mind that all the listeners are always notified. Moreover, in case you don’t want any further processing of an event you can call AWT Event. consume () method. There is another method which is used by a listener to check for the consumption. The method is Consumed () method.

With the consumption of the events by the system once the listener is notified, the events stop being processed. Consumption only works for Input Event and its subclasses. You can use the consume () methods for the Key Event just in case you do not require any input by the user through the use of a keyboard.

The step by step procedure of Event handling is as follows:

1. The component generates subclasses of an AWT Event when anything interesting or intriguing takes place.
2. As permitted by the Event sources, any class can act like a Listener. For example, add Action Listener () method is used for any action to be performed, where Action is the event type. There is another method by which you can remove the listener class which is remove XXX Listener() method, where XXX is the event type.
3. A listener type such as an Action Listener must be implemented for handling an event.
4. Some special listener types require the implementation of multiple methods such as key Events. The key release, key typed and key press, are the three methods which are the requisites of implementation and registration of Key events. A few special classes exist referred to as adapters which are used in implementing listener interfaces and in stubbing out all the methods. Such adapter classes can be subclassed and necessary methods can be overridden.

AWT Event:

Most of the times every event-type has Listener interface as Events subclass the AWT Event class. However, Paint Event and Input Event do not have the Listener interface because only the paint() method can be overriden with Paint Event etc.

Low-level Events:

Low level events represent a low level input or window. Key press, mouse movement, window opening, etc. are the types of low level events.

For example, on typing the letter ‘A’, the three events one for pressing, one for releasing and one for typing are generated. Given below are the various types of low-level events and operations generated by each event.

Table 9.1: Events and the operations that generate them

Focus Event	Used for Getting/losing focus.
Mouse Event	Used for entering, exiting, clicking, dragging, moving, pressing, or releasing.
Container Event	Used for Adding/removing component.
Key Event	Used for releasing, pressing, or typing (both) a key.
Window Event	Used for opening, deactivating, closing, Iconifying, deiconifying, really closed.
Component Event	Used for moving, resizing, hiding, showing.

Semantic Events:

The interaction with GUI component is represented by the Semantic events like changing the text of a text field, selecting a button etc**.** The different events generated by different components is shown below:

Table 9.2: Events generated by different components

Item Event	Used for state changed.
Action Event	Used for do the command.
Text Event	Used for text changed.
Adjustment Event	Used for value adjusted.

Event Sources:

If a component is an event source for something then the same happens with its subclasses. The different event sources are represented by the following table.

Table 9.3: Event sources

Low-Level Events	
Window	Window Listener
Container	Container Listener
Component	Component Listener Focus Listener Key Listener Mouse Listener Mouse Motion Listener

Semantic Events	
Scrollbar	Adjustment Listener
Text Area Text Field	Text Listener
Button List Menu Item Text Field	Action Listener
Choice Checkbox Checkbox Check box Menu Item List	Item Listener

Event Listeners:

Every listener interface has at least one event type. Moreover, it also contains a method for each type of event the event class incorporates. For example as discussed earlier, the Key Listener has three methods, one for each type of event that the Key Event has: keyTyped(), keyPressed() and keyReleased().

The listener interfaces and their methods are as follow:

Table 9.4: Listener interfaces and their methods

Interface	**Methods**
Window Listener	Window Activated(Window Event e)
	Window Deiconified(Window Event e)
	Window Opened(Window Event e)
	Window Closed(Window Event e)
	Window Closing(Window Event e)
	Window Iconified(Window Event e)
	Window Deactivated(Window Event e)
Action Listener	Action Performed(Action Event e)
Adjustment Listener	Adjustment Value Changed(Adjustment Event e)
Mouse Listener	Mouse Clicked(Mouse Event e)
	Mouse Entered(Mouse Event e)
	Mouse Exited(Mouse Event e)
	Mouse Pressed(Mouse Event e)
	Mouse Released(Mouse Event e)

Focus Listener	Focus Gained(Focus Event e)
	Focus Lost(Focus Event e)
Item Listener	Item State Changed(Item Event e)
Key Listener	Key Released(Key Event e)
	Key Typed(Key Event e)
	Key Pressed(Key Event e)
Component Listener	Component Hidden(Component Event e)
	Component Moved(Component Event e)
	Component Shown(Component Event e)
	Component Resized(Component Event e)
Mouse Motion Listener	Mouse Moved(Mouse Event e)
	Mouse Dragged(Mouse Event e)
Text Listener	Text Value Changed(Text Event e)
Container Listener	Component Added(Container Event e)
	Component Removed(Container Event e)

Check your progress 5

1. Write down the procedure for event handling.
2. Explain semantic events

....

....

....

....

....

....

....

....

....

1.7 Adapter Classes

There are some event listeners that have multiple methods to implement. That is some of the listener interfaces contain more than one method. For instance, the Mouse Listener interface contains five methods such as mouse Clicked, mouse Pressed, mouse Released etc. If you want to use only one method out of these then also you will have to implement all of them. Hence, the methods which we do not want to care about can have empty (clear) bodies. To avoid such thing, we have adapter class.

Now it is fact that Adapter classes help us in avoiding the implementation (execution of) of the empty method bodies. Usually (Generally) an adapter class is there for each listener interface having more than one method.

For example, the Mouse Adapter class implements the Mouse Listener interface**.** An adapter class can be used by creating (making) a subclass of the same and then overriding the methods which are of use only. Hence, it avoids the implementation of all the methods of the listener interface. The following example shows the implementation of a listener interface directly.

```
public class MyClass implements MouseListener {
...
someObject.addMouseListener(this);
...
/* -Empty method definition- ok. -*/
public void mouseEntered(MouseEvent e) {
}

/*- Empty method definition-ok -. */
public void mouseExited(MouseEvent e) {
}
/*- Empty method definition-ok- */
public void mousePressed(MouseEvent e) {
}

}
}
```

In the above program code, the adapter class has been used (untilled). This class has been used as an anonymous (unknown) inner class to draw a rectangle within an applet**.** This example demonstrates the functionality of the mouse press. That is on every click of the mouse from top left corner, we are going to get a rectangle on the release of the bottom right.

Check your progress 6

1. What are adapter classes?
2. Give the use of adapter classes

....

....

....

....

....

....

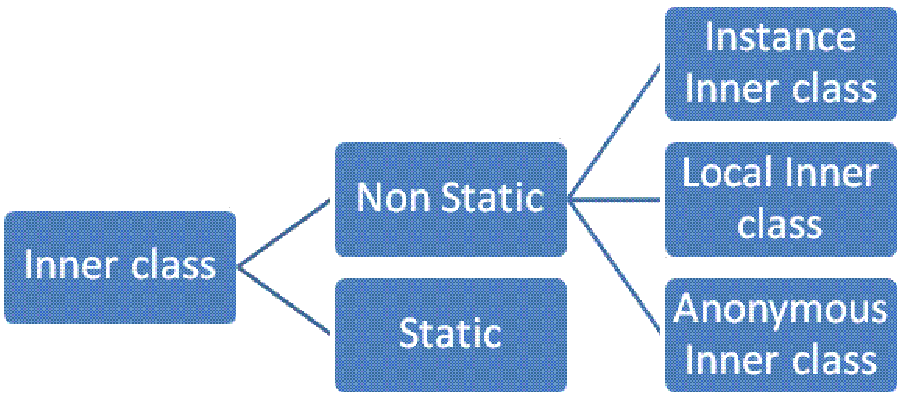
....

....

....

....

1.8 Inner Classes



Inner Classes

Inner classes cannot have (will not have) static members, only static final variables. Interfaces are never be inner. Hence Static classes are not inner classes.

Inner classes may inherit static members that are not compile-time constants even though they may not declare them.

Nested classes that are not inner classes may declare static members freely, in accordance with the usual rules of the Java programming language. Member interfaces are always implicitly static so they are never considered to be inner classes. A statement (agreement) or expression occurs in a static context if and only if the innermost method, constructor, instance initialiser, static initialiser, field initialiser, or explicit constructor invocation (spell) statement enclosing the statement or expression is a static method, a static initialiser, the variable initialiser of a static variable or an explicit constructor invocation (spell) statement.

We should not that a blank final field of a lexically enclosing class may not be assigned within an inner class.

For Example:

```
class HasStatic {

static int j = 100

}


class Outer{


final int z=10


class Inner extends HasStatic {

static final int x = 3

static int y = 4

}

static class Inner2 {
```

```
public static int size=130

}


interface InnerInteface {

public static int size=100;

}

}


public class InnerClassDemo {


public static void main(String[] args) {


Outer outer=new Outer();

System.out.println(outer.new Inner().z)

System.out.println(outer.new Inner().x)

System.out.println(outer.new Inner().j)


System.out.println(Outer.Inner2.size)


System.out.println(Outer.InnerInteface.size)

}

}
```

Hence, in this case it gives compilation problems, as z cannot be used in inner class "Inner".

Also note that a Method parameter names may not be redeclared as it is a local variables of the method, or you may say as exception parameters of catch clauses in a try statement of the method or constructor. However, a parameter of a method or constructor may be shadowed anyplace inside a class declaration nested within that method or constructor. Such a nested class declaration (statement) could declare either a local class or an anonymous class.

Now coming to Section Nested Inner Classes:

Consider the below program:

```
class WithDeepNesting {
boolean toBe

WithDeepNesting(boolean b) { toBe = b;}

class Nested {
boolean theQuestion

class DeeplyNested {

DeeplyNested(){
theQuestion = toBe || !toBe
}
}
}

public static void main(String[] args) {

WithDeepNesting withDeepNesting=new WithDeepNesting(true)
WithDeepNesting.Nested nested=withDeepNesting.new Nested()
nested.new DeeplyNested()
System.out.println(nested.theQuestion)
}
}
```

We may also note and understand that the Inner classes whose declarations do not occur in a static context may freely refer to the instance variables of their enclosing (attached) class. We need to understand an instance variable is always defined (clear) with respect to an instance. Now in the case of instance variables of an enclosing class, the instance variable must be defined with respect to an enclosing instance of that class. So, we can consider the example, the class Local above has an enclosing instance of class Outer. As a further example we can say that:

In this every instance of With Deep Nesting. Nested. Deeply Nested has an enclosing instance of class With Deep Nesting. Nested (its immediately enclosing instance) and an enclosing instance of class With Deep Nesting (its 2nd lexically enclosing instance). Hence, it is going to prints: true.

Check your progress 7

1. Write a note on inner classes.
2. What could be declared a local or anonymous class?

....

....

....

....

....

....

....

....

....

....

....

....

1.9 Anonymous Inner Classes

An Inner classes have made it possible to code with less lines, coding became easy task and in the process to obscure the code to the unenlightened but make it wonderfully well-designed (elegant) to those who understand. A good friend of mine and coworker told me "we may say that One good Java programmer is better than comparatively about ten bad Java programmers" which I heartily agreed with. He then went on to say "And five bad Java programmers are better than ten bad Java programmers". Hence Inner classes have widened this divide. We can say that the typical way of using anonymous inner classes is for writing GUI event handlers, let us take for example,

```
button.addActionListener (new ActionListener () {
 // This is how we define an anonymous inner class
 public void actionPerformed(ActionEvent e) {
System.out.println ("-ok the button was pressed!-")
}
});
```

The amazing thing is that we are actually defining a new class (!) while calling another method. You can virtually make new classes in all sorts of places in Java.

```
new Thread(new Runnable() {
public void run() {
try {
while (true) {
sleep(1000); System.out.print(".")
}
}
catch(InterruptedException ex) { }
}
}).start()
```

If we try to look at the definition of a Thread, we can see that it takes a Runnable (to work) as a parameter so it makes sense to create an anonymous inner class from Runnable (to work) and stick (fix in to) that into the parameter. However, after looking more carefully inside Thread we notice that the run () method defined in Thread, calls the run() method defined in Runnable, if a Runnable has been passed into the Thread constructor. Instead, it would be more efficient to do the following, because we would have one less object on the heap and one less method call per Thread creation:

```
new Thread() {
public void run() {
try {
while (true) {
sleep(1000); System.out.print(".")
}
}
 catch(InterruptedException ex) { }
}
}).start()
```

This way Thread by itself is made into an anonymous inner class and we override the run() method so as an alternative (in place of) of Thread.run() having to check that a Runnable exists (present) we can just execute the code in run().

An application (use) of anonymous inner class is to pass an array as a parameter to a method, which is shown below statements :

```
String[] temp_names = new String[4]
temp_names[0] = "Ashok"
temp_names[1] = "Zameer"
temp_names[2] = "Monica"
temp_names[3] = "Shakila"
universityRegistration.addNames(temp_names)
```

or, alternatively

```
String[] temp_names = {"Ashok","Zameer" , "Monica","Shakila" }

universityRegistration.addNames(temp_names)
```

Know we would not need to have a temporary variable which is bad. We could thus say that:

```
universityRegistration.addNames(

new String[]({"Ashok","Zameer" , "Monica","Shakila" })
```

If you wanted to pass in a Collection instead of an array it would look as follows:

```
Collection temp_names = new Vector(4)

temp_names.add("Ashok ")

temp_names.add("Zameer ")

temp_names.add("Monica ")

temp_names.add("Shakila ")

universityRegistration.addNames(temp_names);
```

Here the ability to avoid local temporary variables with an arrays was always (every time) a strong deciding (decision making) factor in defining interfaces to my classes because we could get away with(replace with) one line of code instead of five and the less lines of code the better hence, with anonymous inner classes we can get the same effect seen above but with collections:

```
universityRegistration.addNames (new Vector(4)

  {{ add Ashok","Zameer" , "Monica","Shakila" }})
```

Hence the call to the super constructor always takes place first, so we could re-write MyVector as follows without changing the functionality in any way:

```
public class MyVector extends Vector {

  { // initialiser block

    add ("Ashok","Zameer" , "Monica","Shakila");

  }

  public MyVector() {

    super(4); // to initialise it with a size of 4
  }
}
```

In case if we want to make an instance of an anonymous inner class we can pass the parameters directly to the super class via the parameter list of the constructor of the anonymous class. In addition to this, any init block denoted by { } is done AFTER the call to the super class constructor is completed, so the class MyVector could look like this:

```
Vector myVector =
  new Vector(4) { // defining anonymous inner class
{
 add;
}
};
```

From here the step to addNames(new Vector(4) {{ add("Ashok "); add("Zameer "); add("Monica ") add("Shakila ");}});

Check your progress 8

1. Define an anonymous inner class.
2. Explain how thread itself is made into an anonymous inner class.

....

....

....

....

....

....

....

....

....

1.10 Applet Fundamentals

Applet can be defined as a small program that is intended to be embedded (fixed) inside another or in a different application such as a browser. The JApplet class must be the superclass of any applet that is to be embedded (fixed) in a Web page or viewed by the Java Applet Viewer (appletviewer.exe). The JApplet class provides (gives) a standard interface between applets and their environment. JApplet hierarchy is mentioned as follows:

java.lang.Object

java.awt.Container

java.awt.Panel

java.applet.Applet

javax.swing.JApplet

An Applet is a java program that can be embedded (fixed) into HTML pages. Java applets run on the java enables web browsers such as mozilla and internet explorer. Applet is designed for the purpose of running remotely on the client browser, so there are some restrictions (limitation) on it. Applet can't access system resources on the local computer (PC). Applets are used to make the web site more dynamic and entertaining supporting for the changes in day to day life as per the new requirement.

Applets Structure

- Both (two) methods that are called automatically- init() and paint()
- Also there is no main method.
- The init method initializes (begin with) variables and objects; then in case, if you do not have one you will inherit one from the JApplet class.
- Use paint to draw screen required.
- There are lot of methods exist in JApplet class so the "extends" keyword inherits everything that the class has. In the above example, JApplet is parent class and shellapplet is the subclass so we have to use the keyword "extends" to create inheritance.
- We have to import JApplet and java.awt.Graphics (abstract windowing toolkit) to get Graphics to paint.

- All applets should have inherit JApplet
- The use of the "super" keyword in subclass to call up method in the superclass.
- super.paint(g); this says that we should use the paint method from JApplet in my paint class

Now the Other applet methods

- repaint() going to repaints when the window which the applet resides re-gains focus
- stop () is called when applet is no longer visible. Signature: public void stop (). This Stop is called by the browser or applet viewer to inform this applet that it must stop its execution. It is also called when the Web page that contains this applet has been replaced (changed by) by another page and also just prior to the applet is to be cracked. A subclass of JApplet should override this method if it has any action or process that it wants to do each time the Web page containing (having) it is no longer visible. The implementation of stop method provided (gives) by the JApplet class does nothing.
- start() function is called after the applet is visible. The signature for the start method is as follows: public void start(). Start is called by the browser or applet viewer to tell this applet that it must start its execution. It is called after the init method and each time the applet is revisited in a Web page. A subclass of JApplet must override this method in case if it has any action that it wants to perform each time the Web page containing it is visited. The implementation (induction) of start method provided by the JApplet class does nothing.
- destroy() when you are hosting window is closed (exit). The signature is as follows: public void destroy(). Destroy method is called (defined) by the browser or applet viewer to inform this applet that it is being cultivated (refined) and that it should destroy any resources that it has allocated. The stop method will at all times be called before destroy. A subclass of JApplet should override this method if it has any action that it wants to perform before it is destroyed. The implementation of destroy method provided by the JApplet class does nothing.

Running an Applet:

We need to use html code to run applets. The minimum html required to run applet with a browser (java host) is as follows:

```
<HTML>
<HEAD>
<TITLE>TitleName</TITLE>
</HEAD>
<BODY>
<APPLET>
   CODE = Classname.class
   CODEBASE = . directory of class file
   WIDTH = 50 width of window in pixels
   HEIGHT = 50 height of window in pixels
</APPLET>
</BODY>
</HTML>
```

Note that in the applet tag, you include the .class bytecode file and not the .java.

Advantages of Applet

- Applets are cross platform and can run on Windows, Mac OS and Linux platform.
- Applets can work on all the version of Java Plugin.
- Applets run in a sandbox, so the user does not need to trust the code, so it can work without security approval.
- Applets are supported by most web browsers.
- Applets are cached in most web browsers, so will be quick to load when returning to a web page.
- User can also have full access to the machine if user allows.

Disadvantages of Java Applet:

- Java plug-in is required to run applet.
- Java applet requires JVM so first time it takes significant startup time.

- If applet is not already cached in the machine, it will be downloaded from internet and will take time.

Its difficult to desing and build good user interface in applets compared to HTML technology.

Check your progress 9

1. Give the structure of applets.
2. Write a note on the other applet methods.

....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....

1.11 Applet Life-Cycle

Let us learn about Applet Lifecycle. Applet runs in the browser and their lifecycle methods are called by JVM after it is loaded and destroyed. You can see here the lifecycle methods of an Applet:

init(): method of this kind is called to initialised an applet.

start(): method of this kind is called after the initialization of the applet.

stop(): This method would be called multiple times in the life cycle of an Applet.

destroy(): This method is called once in the life cycle of the applet when applet is destroyed.

init () method: Now we need to understand that the life cycle of an applet is going to begin when there a time the applet is first loaded into the browser and then called the init() method. The init() method is called only one time in the life cycle on an applet. The init() method is mostly called to read the PARAM tag in the html file. The init () method retrieves the passed parameter through the PARAM tag of html file using get Parameter() method. The various initialization such as initialization of variables and the objects like image, sound file are loaded in the init () method thereafter the initialization of the init () method then user can interact with the Applet and generally applet contains the init() method.

Start () method: In this method an applet is called after the initialization method init(). Hence this method may be called multiple times when the Applet needs to be started or restarted. To understand this we take an example, in case of the user wants to return to the Applet, then in this situation the start Method() of an Applet will be called by the web browser and the user will be back on the applet. In the start method user can interact within the applet.

Stop () method: this stop () method can be called many times in the life cycle of applet like the start () method or otherwise should be called at least one time. There is only minor difference between the start() method and stop () method. Take for example, the stop() method is called by the web browser on that time or occasion when the user leaves one applet to go to another applet and the start() method is called on that time, when the user wants to go back into the first program or Applet.

destroy() method: this destroy() method is called only one(once) time in the life cycle of Applet like init() method. This method is going to be called only on that time when the browser needs to Shut down.

Check your progress 10

1. Write a note on init() method.
2. When is the destroy() method called?

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

1.12 Let Us Sum Up

This Unit made us learn about Abstract Window Toolkit which is nothing but is Java's original platform-independent windowing, graphics and user-interface widget toolkit. The AWT is now part of the Java Foundation Classes (JFC) the standard API for providing a graphical user interface (GUI) for a Java program. A Window provides a top-level window on the screen, with no borders or menu bar. It provides a way to implement pop-up messages, among other things. The default layout for a Window is Border Layout.

Working with Frames: A Frame is a Window with all the window manager's adornments (window title, borders, window minimize/maximize/close functionality) added. It may also include a menu bar. Since Frame subclasses Window, its default layout is Border Layout. Frame provides the basic building block for screen-oriented applications.

As long as working with Graphics is concern Java provides numerous primitives for drawing lines, squares, circles, polygons and images. The figure shows a simple drawing. The Font, Font Metrics, Color and System Color classes provide the ability to alter the displayed output. With the Font class, you adjust how displayed text will appear. With Font Metrics, you can find out how large the output will be, for the specific system the user is using. You could use the Color class to set the color of text and graphics.

We have learned about Controls where in Labels, Buttons, Check Boxes, Radio Button, Text Area, and Text Field covered. After this we learned about Layout Manager. In this All Containers, by default, has a layout manager; an object that implements the Layout Manager interface. If a Container's default layout manager doesn't go well with your needs, you can easily replace it with another one. We further came to know about Rule those can be said as you have to openly tell a container not to use a layout manager, it is linked with its own occasion of a layout manager. This layout manager is automatically/by default consulted every time the Container might need to change its appearance. Most layout managers do not require programs to directly call their respective methods.

There is also learning about how to choose A Loyout Manager /Way and D How ti Reate A reate a Layout Manager and Associate it with a container.

We also seen and understood about rules for Layout Manager like Flow layout, Grid Layout and Grid bag layout. Further we have covered event handling.

Events are the integral part of the java platform. You can see the concepts related to the event handling through the example and use methods through which you can implement the event driven application.

There was also learning related to the step by step procedure of Event handling and even Events and the operations that generate them and Events generated by different components.

Further we have also learned about there are some event listeners that have multiple methods to implement. That is some of the listener interfaces contain more than one method. For instance, the Mouse Listener interface contains five methods such as mouse Clicked, mouse Pressed, mouse Released etc. There is also mention about Inner classes cannot have static members, only static final variables. 1. Interfaces are never inner. 2. Static classes are not inner classes. We have also learned about anonymous inner classes. There is in detail learning about applets, applets life cycle

1.13 Suggested Answer for Check Your Progress

Check your progress 1

Answers: See Section 1.2

Check your progress 2

Answers: See Section 1.3

Check your progress 3

Answers: See Section 1.4

Check your progress 4

Answers: See Section 1.5

Check your progress 5

Answers: See Section 1.6

Check your progress 6

Answers: See Section 1.7

Check your progress 7

Answers: See Section 1.8

Check your progress 8

Answers: See Section 1.9

Check your progress 9

Answers: See Section 1.10

Check your progress 10

Answers: See Section 1.11

1.14 Glossary

1. **Applet -** An apple is a small program that is intended to be embedded inside another application such as a browser.
2. **Frame -** A Frame is a Window with all the window manager’s adornments (window title, borders, window minimize/maximize/close functionality) added.
3. **Inner classes** - Inner classes may inherit static members that are not compile-time constants even though they may not declare them.
4. **Adapter class -** methods which you do not want to care about can have empty bodies. To avoid such thing, we have adapter class.

1.15 Assignment

1. Write a program to create Grid Layout.
2. Write the rules of thumb for using Layout managers.

1.16 Activities

Write programs to illustrate the use of controls?

1.17 Case Study

A case study on Java Applet: You have to write a Java applet to calculate the average of a list of integers and their count. Now the input data is stored in a text file, to this each line contains a single integer. The input file is chosen via Open File Windows-like Dialog box. In this the average and count (number of integers in a file) are displayed inside the applet. Now you provide for error checking of the file name, i.e. the program must display a meaningful message when the input file name is incorrect, and it should ignore (non consideration) any lines in the input file that are other than a single integer.

1.18 Further Reading

1. Core Java 2, 2 volumes, Cay S. Horstmann, Gary Cornell, The Sun Microsystems Press, 1999, Indian reprint 2000
2. Java 2, The Complete Reference, Patrick Naughton and Herbert Schildt, Tata McGraw Hill, 1999
3. Programming with Java, Ed. 2,E. Balagurusamy, Tata McGraw Hill, 1998, reprint, 2000
4. The Java Tutorial, Ed. 2, 2 volumes, Mary Campione and Kathy Walrath, Addison Wesley Longmans, 1998
5. The Java Language Specification, Ed. 2, James Gosling, Bill Joy, Guy Steele & Gilad Bracha, (Ed. 1 1996, Ed. 2 2000), Sun Microsystems, 2000
6. Using Java 2, Joseph L. Weber, Prentice Hall, Eastern Economy Edition, 2000