CHAPTER - 10

APPLETS

10. Introduction

Java programs are generally classified into two ways as Applications programs and Applet programs.

***Application Programs*:** Application programs are those programs normally created, compiled and executed as similar to the other languages. Each Application program contains one main() method. Programs are created in a local computer. Programs are compiled with javac compiler and executed with java interpreter.

***Applet Programs*:** Applet programs are small programs that are primarily used in Internet programming. These programs are either developed in local systems or in remote systems and are executed by either a java compatible "Web browser" or "Appletviewer".

An applet developed locally and stored in a local system is known as a ***Local Applet.*** When a Web page is trying to find a local applet, it does not need to use the Internet and therefore the local system does not require the Internet connection. It simply searches the directories in the local system and locates and loads the specified applet.

An applet developed by someone else and stored on a remote computer is known as Remote Applet. If our system is connected to the Internet, we download the remote applet onto our system via the Internet and run it. In order to locate the remote applet, we must know the applet's address on the Web. This address is known as Uniform Resource Locator (URL).

10.1 Applet Life Cycle

Every java applet inherits a set of default behaviors from the Applet class defined in java.applet package. When an applet is loaded, it undergoes a series of changes in its states. The important states of the

Applet Life Cycle states are:

Born or Initialization State

Running State

Idle State

Dead or Destroyed State

Initialization State: Applet enters the initialization state when it is first loaded. This is achieved by calling the **init()** method of **Applet** class. The applet is born. The initialization occurs only once in the applet's life cycle. Generally all the initialization variables are to be placed in the init() method.

Syntax: public void init()

```
{
        ....
        ....    (Action)
}
```

Running State: Applet enters the running state when the system calls the start() method of **Applet** class. This occurs automatically after the applet is initialized. Starting can also occurs if the applet is already in "Stopped State". The start() method may be called more than once.

Syntax: public void start()

```
{
        ....
        ....    (Action)
}
```

Idle or Stopped State: An applet becomes idle when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. This can also done by calling the stop() method explicitly.

Syntax: public void stop()

```
{
        ....
        ....    (Action)
}
```

Dead State: An applet is said to be dead when it is removed from memory. This occurs automatically by invoking the **destroy()** method when we quit the browser. Destroying stage occurs only once in the applet life cycle.

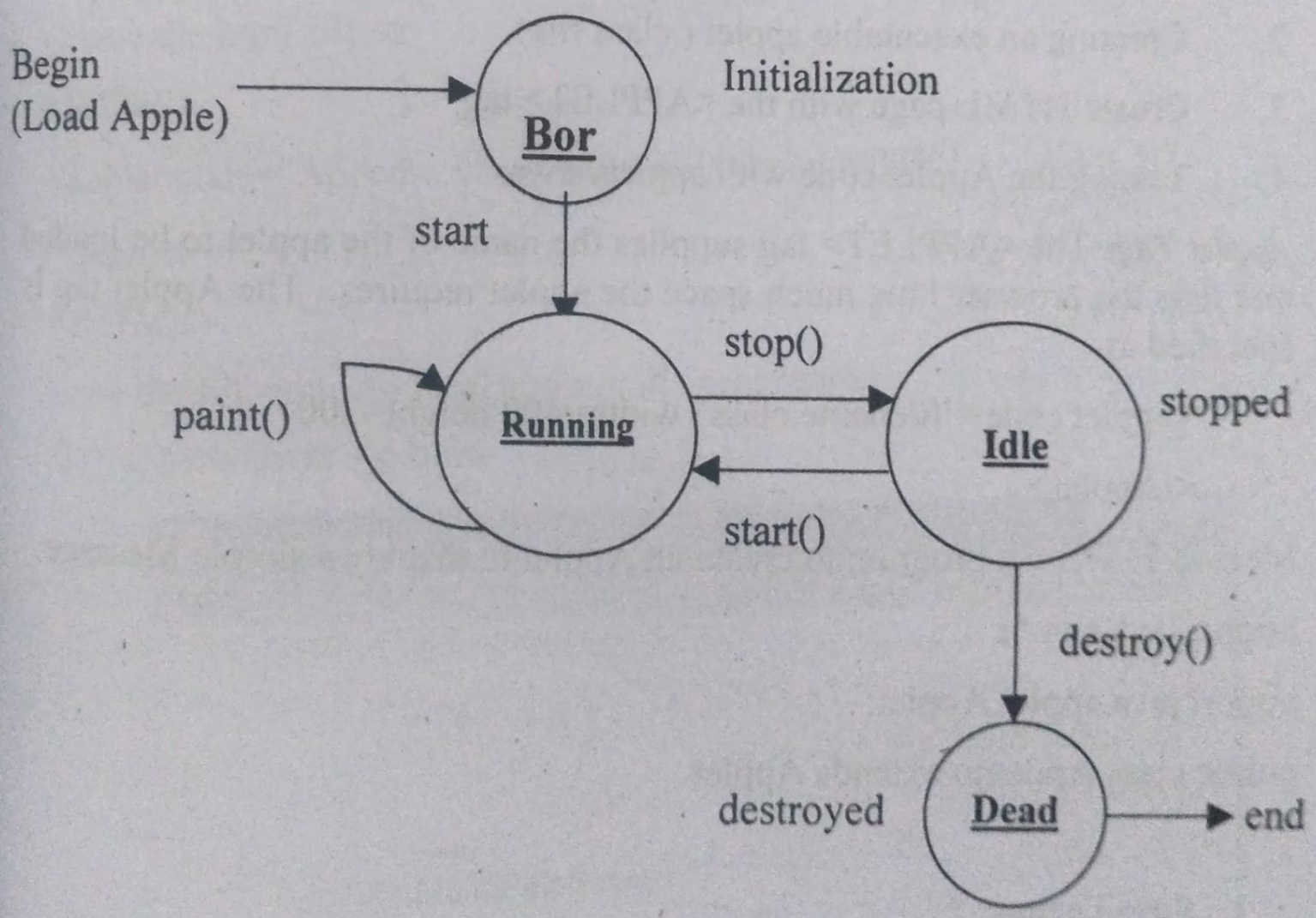
Syntax: public void destroy()

```
{
        ....
        ....    (Action)
}
```

Display State: Display state is useful to display the information on the output screen. This happens immediately after the applet enters into the running state. The paint() is called to accomplish this task. Almost every applet will have a paint() method.

Syntax: public void paint(Graphics g)

```
{
        ....
        ....    (Display Statements)
}
```

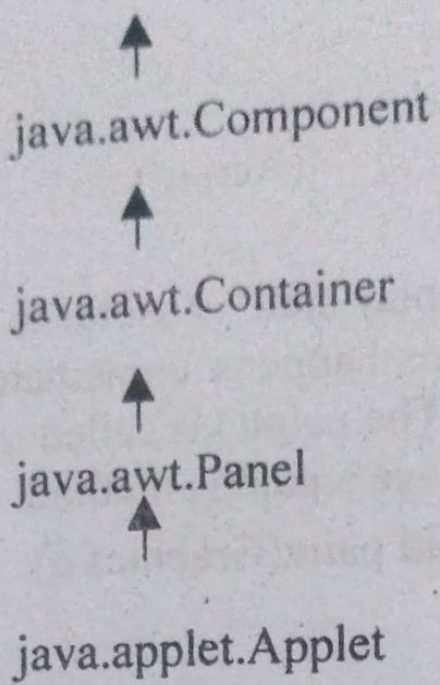


Exit of Browser

Note:1. All the methods are automatically to called when any applet begin execution.

2. The applet execution follows the above order.
3. In every applet it doesn't need to place all the methods.
4. class hierarchy of the applet is

java.lang.Object



The steps involved in developing and testing an applet are:

1. Building an applet code (.java file)
2. Creating an executable applet (.class file)
3. Create HTML page with the <APPLET> tag
4. Testing the Applet code with appletviewer

Applet Tag: The <APPLET> tag supplies the name of the applet to be loaded and tells the browser how much space the applet requires. The Applet tag is specified as

```
<applet code="filename.class" width=400 height=300>
</Applet>
```

Method 1: Write a program to create an Applet to display a simple Message.

```
import java.awt.*;
import java.applet.Applet;
public class Apdemo extends Applet
{
    String msg=" ";
    public void init()
    {
    msg=msg+"Init";
    }
    public void start()
```

```
    {
    msg=msg+"Start";
    }
    public void paint(Graphics g)
    {
    msg=msg+"Paint";
    g.drawString(msg,50,100);
    }
}
```

Save the file with Apdemo.java and compile as

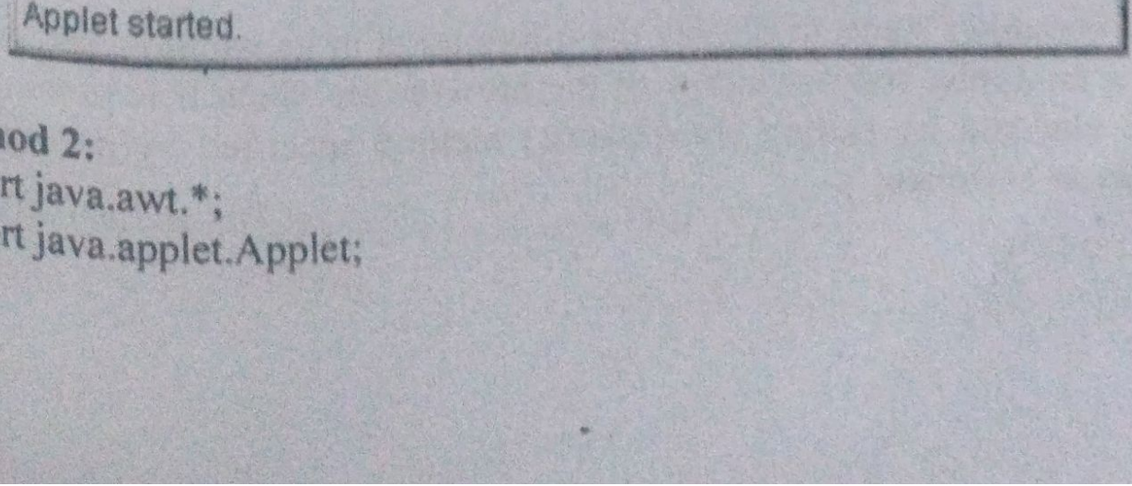
C:> javac Apdemo.java

Create the html file as

```
<HTML>
<applet code="Apdemo.class" width=400 height=350>
</Applet>
</HTML>
```

Save the file with Ap.html and run the program as

C:> appletviewer Ap.html

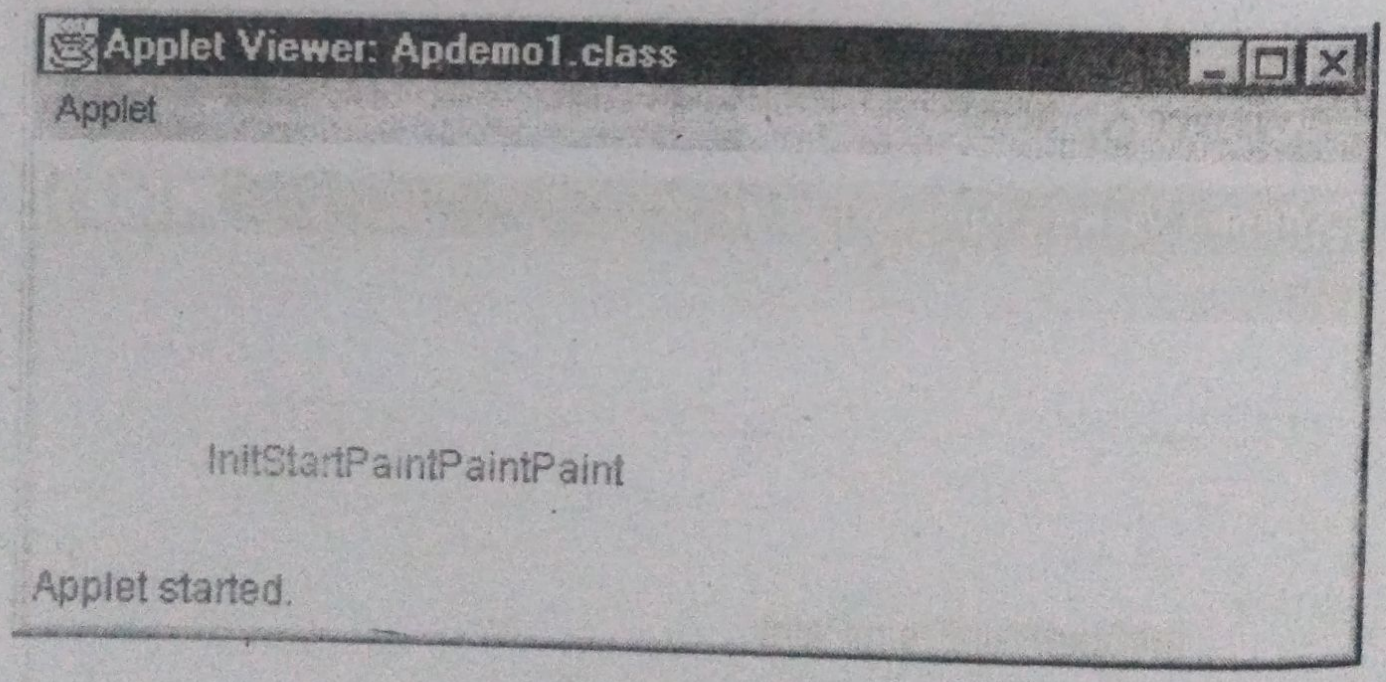


Method 2:

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

```
/* <applet code="Apdemo1.class" width=400 height=350>
</Applet> */
public class Apdemo1 extends Applet
{
        String msg=" ";
        public void init()
        {
        msg=msg+"Init";
        }
        public void start()
        {
        msg=msg+"Start";
        }
        public void paint(Graphics g)
        {
        msg=msg+"Paint";
        g.drawString(msg,50,100);
        }
}
```

Save the file with Apdemo1.java compile and run the program as
C:> javac Apdemo1.java
C:> appletviewer Apdemo1.java



Status Window: Apart to display the information in its window, applet also produce an applet status window of the browser on which it is running. It can be changed by calling showStatus() method specified by passing the string as an argument.

repaint(): If the programmer needs to call the paint(), a call is made to the **Component** class repaint() method. Method repaint request a call to the component class update() method. As soon as possible to clear the components background of any previous status then update calls paints directly. The repaint() method is frequently called by the programmer to force paint() operation.

Syntax: public void repaint()

public void update(Graphics g)

update() method takes a Graphics object as an argument which is supplied automatically by the system when update() is called.

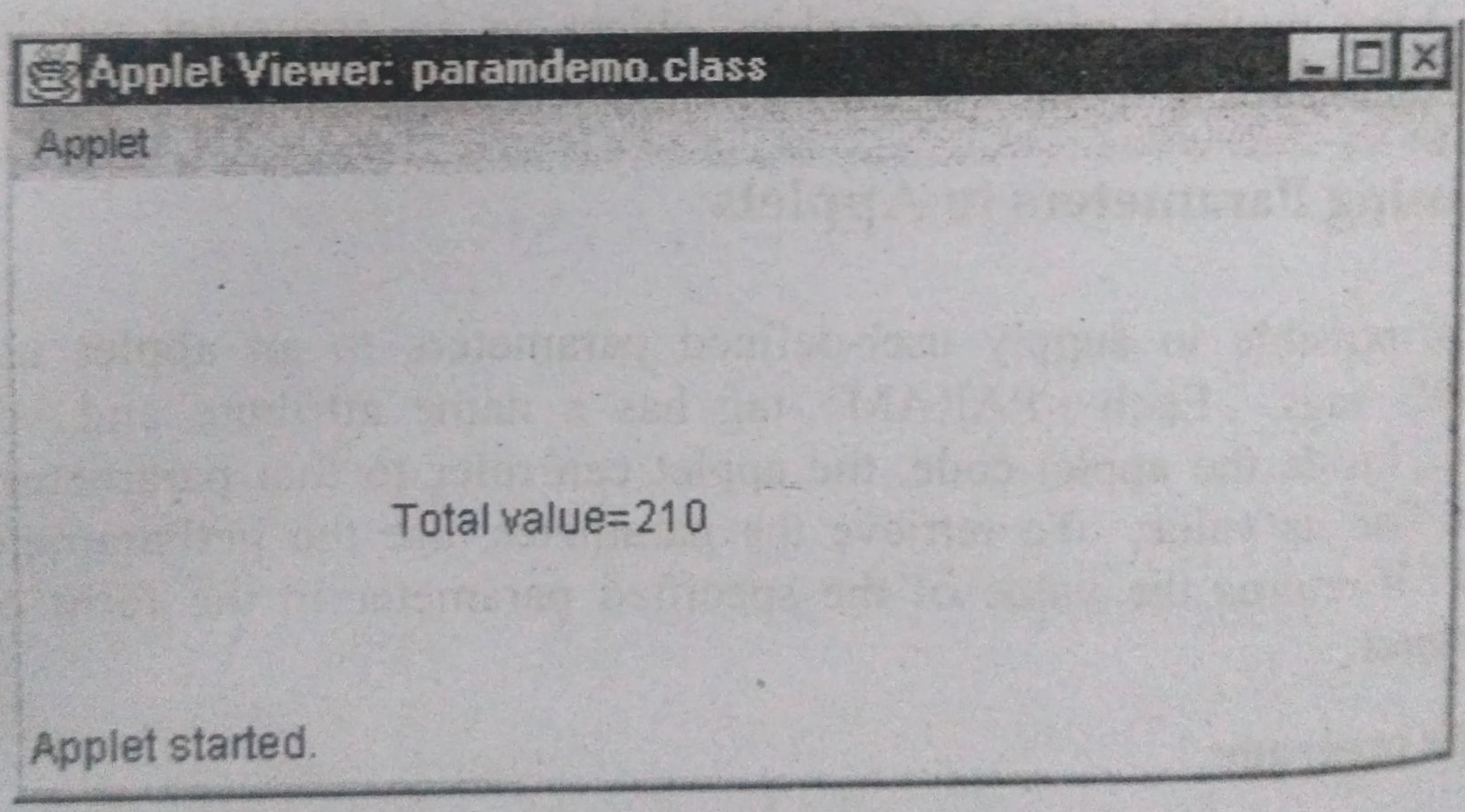
10.2 Passing Parameters to Applets

It is also possible to supply user-defined parameters to an applet using <PARAM> tags. Each <PARAM> tag has a name attribute and value attribute. Inside the applet code, the applet can refer to that parameter by name to find its value. To retrieve the parameter, use the getParameter() method. It returns the value of the specified parameter in the form of a String object.

Example program:

```
import java.awt.*;
import java.applet.Applet;
/* <applet code="paramdemo.class" width=400 height=350>
<param name=first value=50>
<param name=second value=160>
</Applet> */
public class paramdemo extends Applet
{
    String first,second;
    int x,y,sum;
    public void paint(Graphics g)
    {
```

```
        x=Integer.parseInt(getParameter("first"));
        y=Integer.parseInt(getParameter("second"));
        sum=x+y;
        g.drawString("Total value="+sum,100,100);
        }
}
```



CHAPTER - 11

ABSTRACT WINDOW TOOL KIT

11.1 AWT Classes

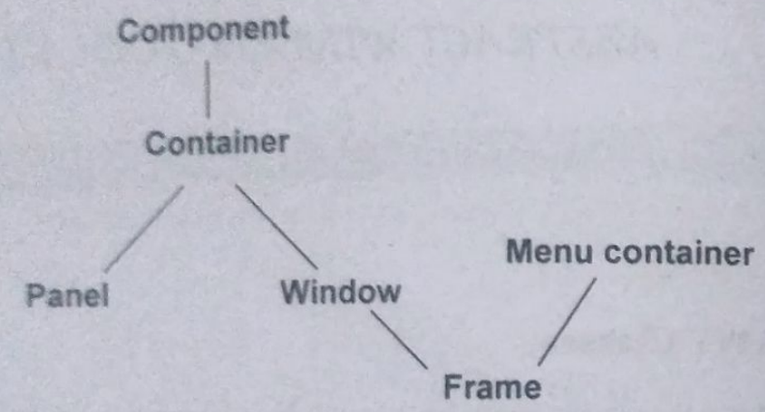
The AWT classes are contained in the java.awt package. It is one of Java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Table lists some of the many AWT classes.

Class	Description
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 indpendent

11.2 Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common

windows are those derived from Panel, which is used by applets, and those derived from Frame, which creates a standard window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding. Figure shows the class hierarchy for Panel and Frame. Let's look at each of these classes now.



Component

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 over a hundred 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.

Container

The **Container** class is a subclass of **Component**. It has additional methods that allow other **Component** objects to be nested within it. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers

Panel

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**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object. In essence, a **Panel** is a window that does not

contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border.

Other components can be added to a **Panel** object by its **add( )** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation( ), setSize( )**, or **setBounds( )** methods defined by Component.

Window

The **Window** class creates a top-level window. A *top-level window* is not contained within any other object; it sits directly on the desktop. Generally, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame.**

Frame

Frame encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners. If you create a **Frame** object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. (An applet that could masquerade as a host-based application could be used to obtain passwords and other sensitive information without the user's knowledge.) When a **Frame** window is created by a program rather than an applet, a normal window is created

Canvas

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas**. **Canvas** encapsulates a blank window upon which you can draw

11.3 Working with Frame Windows

After the applet, the type of window you will most often create is derived from **Frame**. You will use it to create child windows within applets, and top-level or child windows for applications. As mentioned, it creates a standard-style window.

Here are two of **Frame**'s constructors:

Frame( )

Frame(String *title*)

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by *title*. Notice that you

cannot specify the dimensions of the window. Instead, you must set the size of the window after it has been created.

There are several methods you will use when working with **Frame** windows. They are examined here.

11.3.1 Setting the Window's Dimensions

The **setSize( )** method is used to set the dimensions of the window. Its signature is shown here:

void setSize(int *newWidth*, int *newHeight*)

void setSize(Dimension *newSize*)

The new size of the window is specified by *newWidth* and *newHeight*, or by the **width** and **height** fields of the **Dimension** object passed in *newSize*. The dimensions are specified in terms of pixels.

The **getSize( )** method is used to obtain the current size of a window. Its signature is shown here:

Dimension getSize( )

This method returns the current size of the window contained within the **width** and **height** fields of a **Dimension** object.

11.3.2 Hiding and Showing a Window

After a frame window has been created, it will not be visible until you call setVisible( ). Its signature is shown here: void setVisible(boolean *visibleFlag*)

The component is visible if the argument to this method is **true**. Otherwise, it is hidden.

11.3.3 Setting a Window's Title

You can change the title in a frame window using **setTitle( )**, which has this general form:

void setTitle(String *newTitle*)

Here, *newTitle* is the new title for the window.

11.3.4 Closing a Frame Window

When using a frame window, your program must remove that window from the screen when it is closed, by calling **setVisible(false)**. To intercept a window-close event, you must implement the **windowClosing( )** method of the **WindowListener** interface. Inside **windowClosing( )**, you must remove the window from the screen.

The example in the next section illustrates this technique.

11.4 Creating a Frame Window in an Applet

Creating a new frame window from within an applet is actually quite easy. First, create a subclass of **Frame**. Next, override any of the standard window methods, such as **init( )**, **start( )**, **stop( )**, and **paint( )**. Finally, implement the **windowClosing( )** method of the **WindowListener** interface, calling **setVisible(false)** when the window is closed.

Once you have defined a **Frame** subclass, you can create an object of that class. This causes a frame window to come into existence, but it will not be initially visible.

You make it visible by calling **setVisible( )**. When created, the window is given a default height and width. You can set the size of the window explicitly by calling the **setSize( )** method.

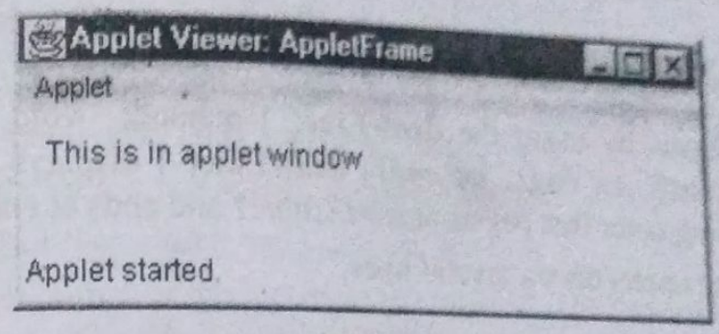
The following applet creates a subclass of **Frame** called **SampleFrame**. A window of this subclass is instantiated within the **init( )** method of **AppletFrame.** Notice that **SampleFrame** calls **Frame**'s constructor. This causes a standard frame window to be created with the title passed in **title**. This example overrides the applet window's **start( )** and **stop( )** methods so that they show and hide the child window, respectively.

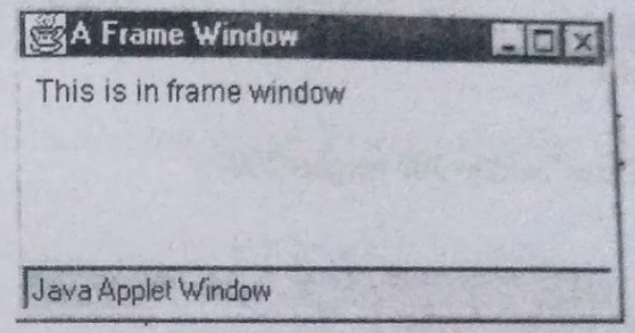
This causes the window to be removed automatically when you terminate the applet, when you close the window, or, if using a browser, when you move to another page.

It also causes the child window to be shown when the browser returns to the applet.

```
// Create a child frame window from within an applet.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AptFrame" width=400 height=70>
</applet>
*/
// Create a subclass of Frame.
class SampleFrame extends Frame {
SampleFrame(String title) {
super(title);
// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
```

```
addWindowListener(adapter);
}
public void paint(Graphics g) {
g.drawString("This is in frame window", 10, 40);
}
}
class MyWindowAdapter extends WindowAdapter {
SampleFrame sampleFrame;
public MyWindowAdapter(SampleFrame sampleFrame) {
this.sampleFrame = sampleFrame;
}
public void windowClosing(WindowEvent we) {
sampleFrame.setVisible(false);
}
}
// Create frame window.
public class AptFrame extends Applet {
Frame f;
public void init() {
f = new SampleFrame("A Frame Window");
f.setSize(250, 250);
f.setVisible(true);
}
public void start() {
f.setVisible(true);
}
public void stop() {
f.setVisible(false);
}
public void paint(Graphics g) {
g.drawString("This is in applet window", 10, 20);
}
}
```



11.5 Working with Graphics

The AWT supports all of graphics methods. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. The origin of each window is at the top-left corner and is 0,0. Coordinates are specified in pixels. All output to a window takes place through a graphics context. A *graphics context* is encapsulated by the **Graphics** class and is obtained in two ways:

- 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**.

For the remainder of the examples in this chapter, we will be demonstrating graphics in the main applet window. However, the same techniques will apply to any other window.

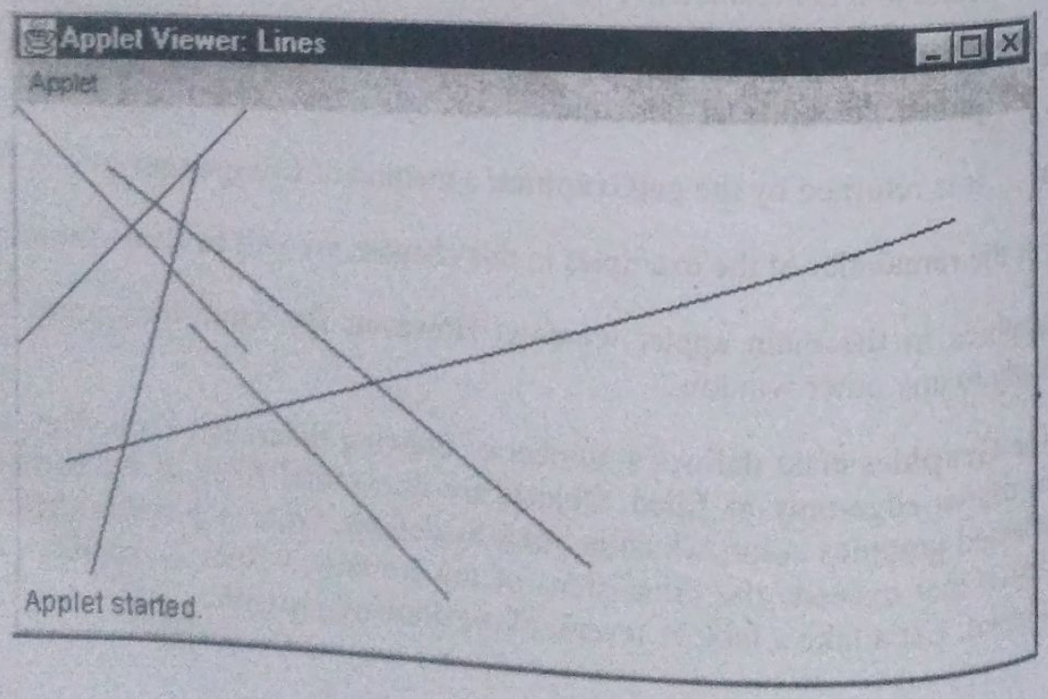
The **Graphics** class defines a number of drawing functions. Each shape can be drawn edge-only or filled. Objects are drawn and filled in the currently selected graphics color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped. Let's take a look at several of the drawing methods.

11.5.1 Drawing Lines

Lines are drawn by using the **drawLine( )** method, void drawLine(int *startX*, int *startY*, int *endX*, int *endY*) **drawLine( )** displays a line in the current drawing color that begins at *startX*,*startY* and ends at *endX*,*endY*.

The following applet draws several lines:

```
// Draw lines
import java.awt.*;
import java.applet.*;
/*
<applet code="Lines" width=300 height=200>
</applet>
*/
public class Lines extends Applet {
public void paint(Graphics g) {
g.drawLine(0, 0, 100, 100);
g.drawLine(0, 100, 100, 0);
g.drawLine(40, 25, 250, 180);
g.drawLine(75, 90, 400, 400);
g.drawLine(20, 150, 400, 40);
g.drawLine(5, 290, 80, 19);
}
}
```



11.5.2 Drawing Rectangles

The **drawRect( )** and **fillRect( )** methods display an outlined and filled rectangle, respectively. They are shown here:

void drawRect(int *top*, int *left*, int *width*, int *height*)

void fillRect(int *top*, int *left*, int *width*, int *height*)

The upper-left corner of the rectangle is at *top*,*left*. The dimensions of the rectangle are specified by *width* and *height*.

To draw a rounded rectangle, use **drawRoundRect( )** or **fillRoundRect( )**, both shown here:

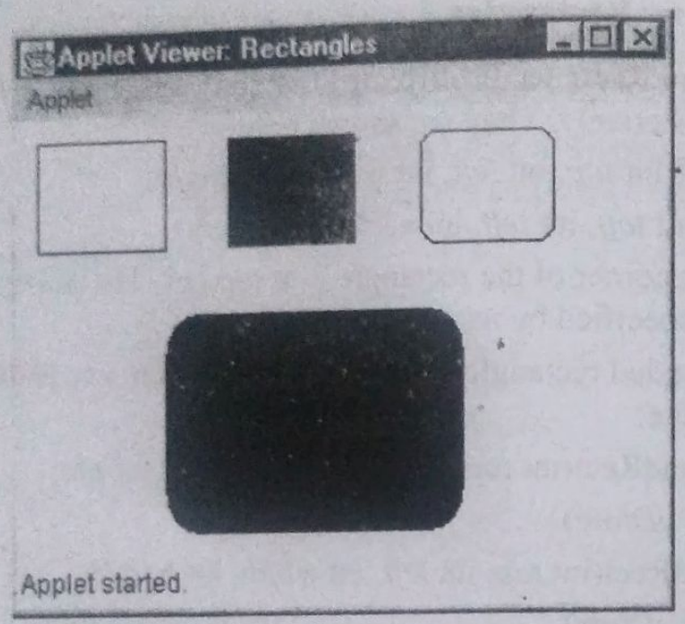
void drawRoundRect(int *top*, int *left*, int *width*, int *height*,
int *xDiam*, int *yDiam*)

void fillRoundRect(int *top*, int *left*, int *width*, int *height*,
int *xDiam*, int *yDiam*)

A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at *top*,*left*. The dimensions of the rectangle are specified by *width* and *height*. The diameter of the rounding arc along the X axis is specified by *xDiam*. The diameter of the rounding arc along the Y axis is specified by *yDiam*.

The following applet draws several rectangles:

```
// Draw rectangles
import java.awt.*;
import java.applet.*;\
/*
<applet code="Rectangles" width=300 height=200>
</applet>
*/
public class Rectangles extends Applet {
public void paint(Graphics g) {
g.drawRect(10, 10, 60, 50);
g.fillRect(100, 10, 60, 50);
g.drawRoundRect(190, 10, 60, 50, 15, 15);
g.fillRoundRect(70, 90, 140, 100, 30, 40);
}
}
```



11.5.3 Drawing Ellipses and Circles

To draw an ellipse, use **drawOval( ).** To fill an ellipse, use **fillOval( ).** These methods are shown here:

void drawOval(int *top*, int *left*, int *width*, int *height*)

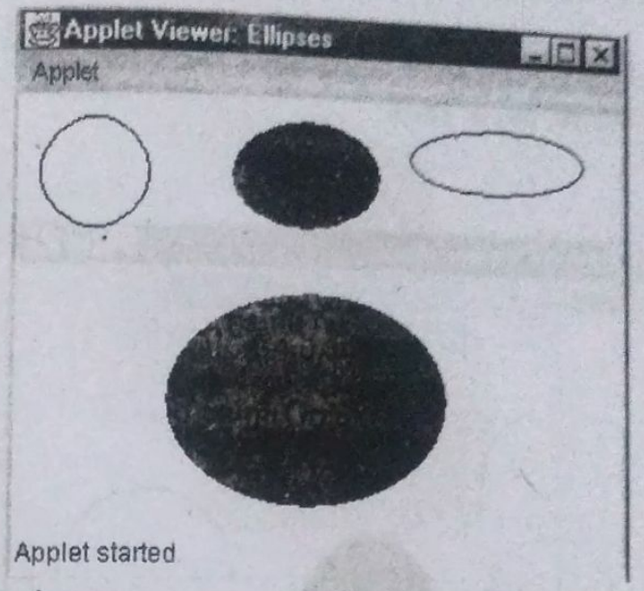
void fillOval(int *top*, int *left*, int *width*, int *height*)

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by *top*,*left* and whose width and height are specified by *width* and *height*.

To draw a circle, specify a square as the bounding rectangle.

The following program draws several ellipses:

```
// Draw Ellipses
import java.awt.*;
import java.applet.*;
/*
<applet code="Ellipses" width=300 height=200>
</applet>
*/
public class Ellipses extends Applet {
public void paint(Graphics g) {
g.drawOval(10, 10, 50, 50);
g.fillOval(100, 10, 75, 50);
g.drawOval(190, 10, 90, 30);
g.fillOval(70, 90, 140, 100);
}
}
```

11.5.4 Drawing Arcs

Arcs can be drawn with ***drawArc( )*** and ***fillArc( ),*** shown here:

void drawArc(int *top*, int *left*, int *width*, int *height*, int *startAngle*, int *sweepAngle*)

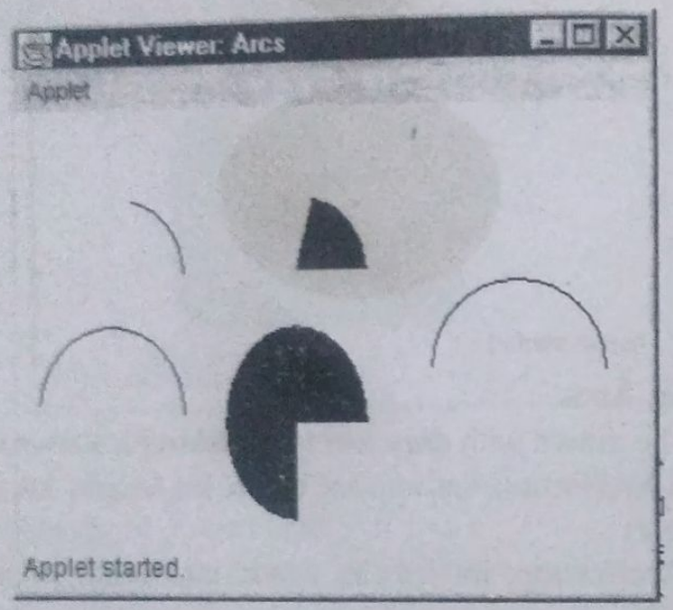
void fillArc(int *top*, int *left*, int *width*, int *height*, int *startAngle*, int *sweepAngle*)

The arc is bounded by the rectangle whose upper-left corner is specified by *top*,*left* and whose width and height are specified by *width* and *height*. The arc is drawn from startAngle through the angular distance specified by sweepAngle. Angles are specified in degrees. Zero degrees is on the horizontal, at the three o'clock position. The arc is drawn counterclockwise if *sweepAngle* is positive, and clockwise if *sweepAngle* is negative. Therefore, to draw an arc from twelve o'clock to six o'clock, the start angle would be 90 and the sweep angle 180.

The following applet draws several arcs:

```
// Draw Arcs
import java.awt.*;
import java.applet.*;
/*
<applet code="Arcs" width=300 height=200>
</applet>
*/
public class Arcs extends Applet {
public void paint(Graphics g) {
g.drawArc(10, 40, 70, 70, 0, 75);
g.fillArc(100, 40, 70, 70, 0, 75);
g.drawArc(10, 100, 70, 80, 0, 175);
```

```
g.fillArc(100, 100, 70, 90, 0, 270);
g.drawArc(200, 80, 80, 80, 0, 180);
}
}
```



11.5.5 Drawing Polygons

It is possible to draw arbitrarily shaped figures using **drawPolygon( )** and **fillPolygon( )**, shown here:

void drawPolygon(int *x*[ ], int *y*[ ], int *numPoints*)

void fillPolygon(int *x*[ ], int *y*[ ], int *numPoints*)

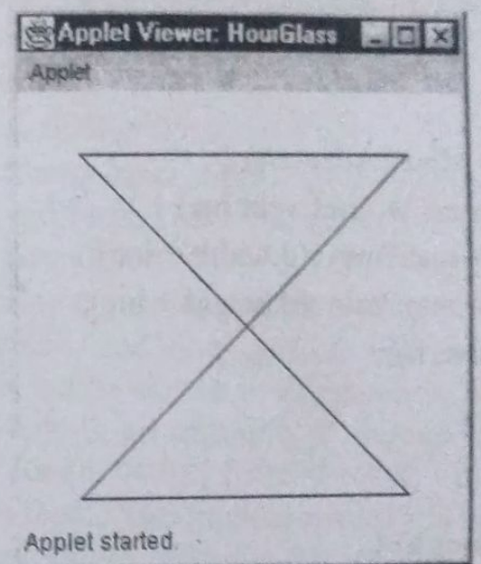
The polygon's endpoints are specified by the coordinate pairs contained within the *x* and *y* arrays. The number of points defined by *x* and *y* is specified by *numPoints*. There are alternative forms of these methods in which the polygon is specified by a **Polygon** object.

The following applet draws an hourglass shape:

```
// Draw Polygon
import java.awt.*;
import java.applet.*;
/*
<applet code="HourGlass" width=230 height=210>
</applet>
*/
public class HourGlass extends Applet {
public void paint(Graphics g) {
```



```
int xpoints[] = {30, 200, 30, 200, 30};
int ypoints[] = {30, 30, 200, 200, 30};
int num = 5;
g.drawPolygon(xpoints, ypoints, num);
}
```



11.5.6 Sizing Graphics

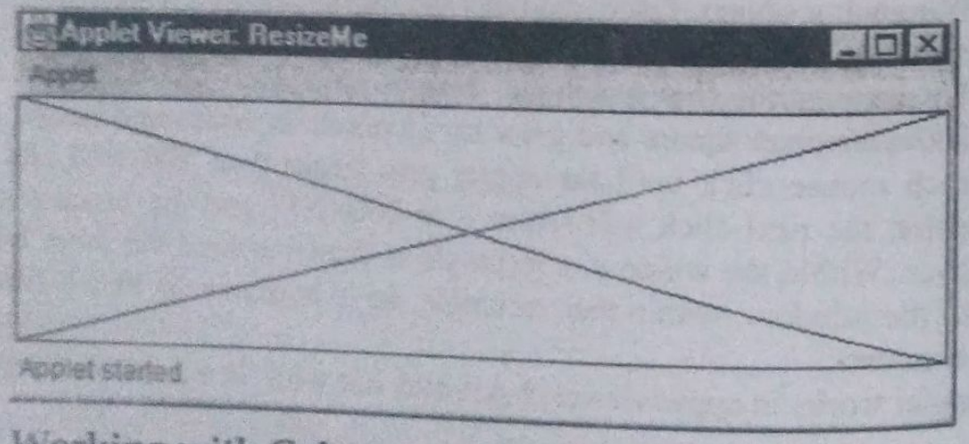
Often, you will want to size a graphics object to fit the current size of the window in which it is drawn. To do so, first obtain the current dimensions of the window by calling **getSize( )** on the window object. It returns the dimensions of the window encapsulated within a **Dimension** object. Once you have the current size of the window, you can scale your graphical output accordingly.

To demonstrate this technique, here is an applet that will start as a 200×200-pixel square and grow by 25 pixels in width and height with each mouse click until the applet gets larger than 500×500. At that point, the next click will return it to 200×200, and the process starts over. Within the window, a rectangle is drawn around the inner border of the window; within that rectangle, an *X* is drawn so that it fills the window.

This applet works in appletviewer, but it may not work in a browser window.

```
// Resizing output to fit the current size of a window.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
```

```
<applet code="ResizeMe" width=200 height=200>
</applet>
*/public class ResizeMe extends Applet {
final int inc = 25;
int max = 500;
int min = 200;
Dimension d;
public ResizeMe() {
addMouseListener(new MouseAdapter() {
public void mouseReleased(MouseEvent me) {
int w = (d.width + inc) > max?min :(d.width + inc);
int h = (d.height + inc) > max?min :(d.height + inc);
setSize(new Dimension(w, h));
}
});
}
public void paint(Graphics g) {
d = getSize();
g.drawLine(0, 0, d.width-1, d.height-1);
g.drawLine(0, d.height-1, d.width-1, 0);
g.drawRect(0, 0, d.width-1, d.height-1);
}
}
```



11.6 Working with Color

The AWT color systemallows you to specify any color you want. It then finds the best match for that color,given the limits of the display hardware currently executing your program or applet.

Thus, your code does not need to be concerned with the differences in the way color is supported by various hardware devices. Color is encapsulated by the **Color** class. **Color** defines several constants (for example, **Color.black**) to specify a number of common colors. You can also create your own colors, using one of the color constructors. The most commonly used forms are shown here:

Color(int *red*, int *green*, int *blue*)

Color(int *rgbValue*)

Color(float *red*, float *green*, float *blue*)

The first constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example: new Color(255, 100, 100); // light red.

The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor: int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00); Color darkRed = new Color(newRed); The final constructor, **Color(float, float, float)**, takes three float values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue.

Once you have created a color, you can use it to set the foreground and/or background color by using the **setForeground( )** and **setBackground( )** methods You can also select it as the current drawing color.

11.6.1 Color Methods

The **Color** class defines several methods that help manipulate colors. They are examined here.

11.6.1 .1 Using Hue, Saturation, and Brightness

The *hue-saturation-brightness* **(HSB)** color model is an alternative to red-green-blue (RGB) for specifying particular colors. Figuratively, *hue* is a wheel of color. The hue is specified with a number between 0.0 and 1.0 (the colors are approximately: red, orange, yellow, green, blue, indigo, and violet). **Saturation** is another scale ranging from 0.0 to 1.0, representing light pastels to intense hues. *Brightness* values also range from 0.0 to 1.0, where 1 is bright white and 0 is black. **Color** supplies two methods that let you convert between RGB and HSB. They are shown here:

static int **HSBtoRGB**(float *hue*, float *saturation*, float *brightness*)

static float[ ] **RGBtoHSB**(int *red*, int *green*, int *blue*, float *values*[ ])

HSBtoRGB( ) returns a packed **RGB** value compatible with the **Color(int)** constructor.

RGBtoHSB( ) returns a float array of HSB values corresponding to RGB integers. If *values* is not **null**, then this array is given the HSB values and returned. Otherwise, a new array is created and the HSB values are returned in it. In either case, the array contains the hue at index 0, saturation at index 1, and brightness at index 2.

11.6.1.2 getRed( ), getGreen( ), getBlue( )

You can obtain the red, green, and blue components of a color independently using **getRed( ), getGreen( ), and getBlue( ),** shown here:

int getRed( )
int getGreen( )
int getBlue( )

Each of these methods returns the RGB color component found in the invoking **Color** object in the lower 8 bits of an integer.

11.6.1 .3 getRGB( ),

To obtain a packed, RGB representation of a color, use **getRGB( ),** shown here: int getRGB( )

The return value is organized as described earlier.

11.6.2 Setting the Current Graphics Color

By default, graphics objects are drawn in the current foreground color. You can change this color by calling the **Graphics** method **setColor()**:

void setColor(Color *newColor*)

Here, *newColor* specifies the new drawing color.

You can obtain the current color by calling **getColor( ),** shown here:

Color getColor( )

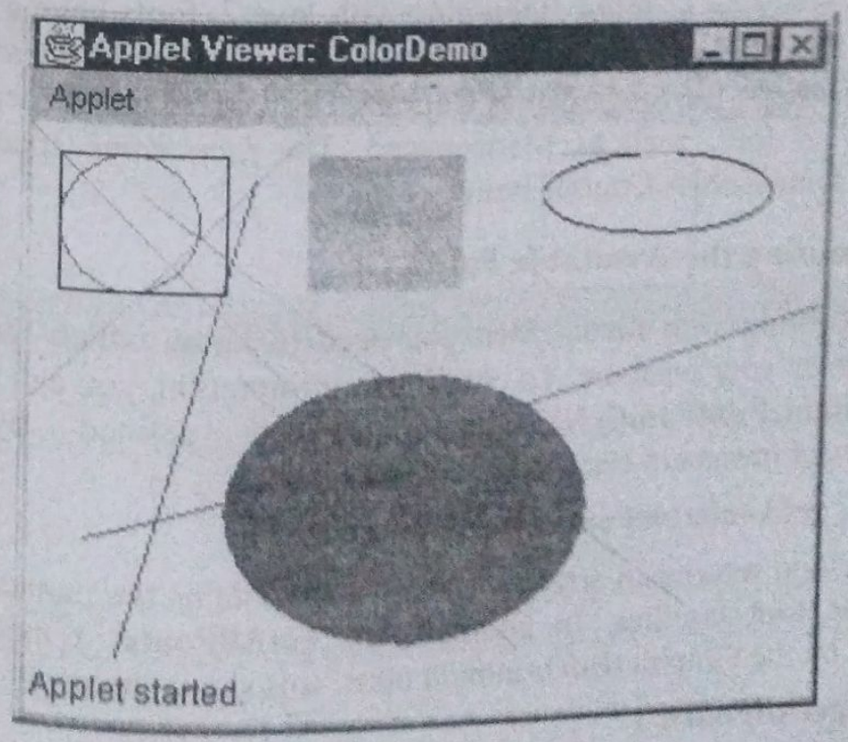
A Color Demonstration Applet

The following applet constructs several colors and draws various objects using these colors:

```
// Demonstrate color.
import java.awt.*;
import java.applet.*;
/*
<applet code="ColorDemo" width=300 height=200>
</applet>
*/
public class ColorDemo extends Applet {
// draw lines
public void paint(Graphics g) {
Color c1 = new Color(255, 100, 100);
```



```
Color c2 = new Color(100, 255, 100);
Color c3 = new Color(100, 100, 255);
g.setColor(c1);
g.drawLine(0, 0, 100, 100);
g.drawLine(0, 100, 100, 0);
g.setColor(c2);
g.drawLine(40, 25, 250, 180);
g.drawLine(75, 90, 400, 400);
g.setColor(c3);
g.drawLine(20, 150, 400, 40);
g.drawLine(5, 290, 80, 19);
g.setColor(Color.red);
g.drawOval(10, 10, 50, 50);
g.fillOval(70, 90, 140, 100);
g.setColor(Color.blue);
g.drawOval(190, 10, 90, 30);
g.drawRect(10, 10, 60, 50);
g.setColor(Color.cyan);
g.fillRect(100, 10, 60, 50);
g.drawRoundRect(190, 10, 60, 50, 15, 15);
}
}
```



11.6.3 Setting the Paint Mode

The *paint mode* determines how objects are drawn in a window. By default, new output to a window overwrites any preexisting contents. However, it is possible to have new objects XORed onto the window by using setXORMode( ), as follows:

void setXORMode(Color *xorColor*)

Here, *xorColor* specifies the color that will be XORed to the window when an object is drawn. The advantage of XOR mode is that the new object is always guaranteed to be visible no matter what color the object is drawn over.

To return to overwrite mode, call setPaintMode( ), shown here:

void setPaintMode( )

In general, you will want to use overwrite mode for normal output, and XOR mode for special purposes. For example, the following program displays cross hairs that track the mouse pointer. The cross hairs are XORed onto the window and are always visible,

11.7 Working with Fonts

The AWT supports multiple type fonts. Fonts have emerged from the domain of traditional typesetting to become an important part of computer-generated documents and displays. The AWT provides flexibility by abstracting font-manipulation operations and allowing for dynamic selection of fonts. Beginning with Java 2, fonts have a family name, a logical font name, and a face name. The *family name* is the general name of the font, such as Courier. The *logical name* specifies a category of font, such as Monospaced. The *face name* specifies a specific font, such as Courier Italic.

11.7.1 Determining the Available Fonts

When working with fonts, often you need to know which fonts are available on your machine. To obtain this information, you can use the **getAvailableFontFamilyNames( )** method defined by the GraphicsEnvironment class. It is shown here:

String[ ] **getAvailableFontFamilyNames( )**

This method returns an array of strings that contains the names of the available font families. In addition, the **getAllFonts( )** method is defined by the GraphicsEnvironment class. It is shown here:

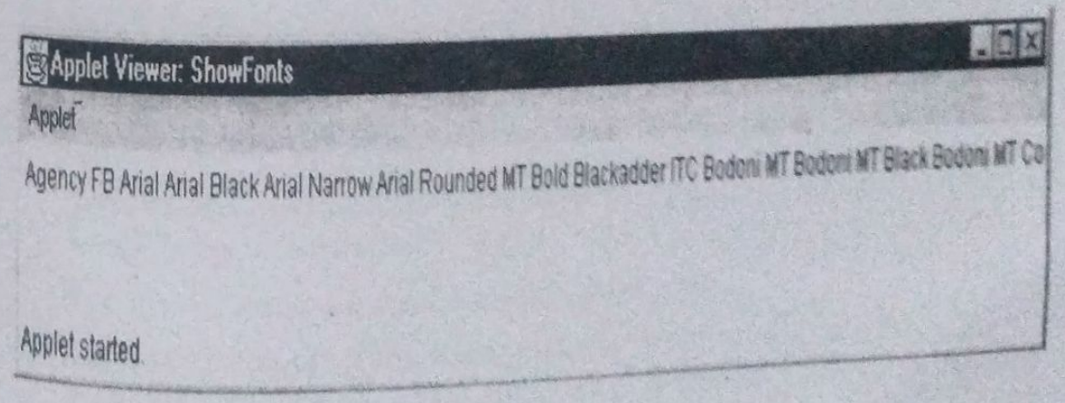
Font[ ] **getAllFonts( )**

This method returns an array of **Font** objects for all of the available fonts. Since these methods are members of **GraphicsEnvironment**, you need a **GraphicsEnvironment** reference to call them. You can obtain this reference by using the **getLocalGraphicsEnvironment( )** static method, which is defined by **GraphicsEnvironment**. It is shown here:

static GraphicsEnvironment getLocalGraphicsEnvironment( )

Here is an applet that shows how to obtain the names of the available font families:

```
// Display Fonts
/*
<applet code="ShowFonts" width=550 height=60>
</applet>
*/
import java.applet.*;
import java.awt.*;
public class ShowFonts extends Applet {
public void paint(Graphics g) {
String msg = "";
String FontList[];
GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment();
FontList = ge.getAvailableFontFamilyNames();
for(int i = 0; i < FontList.length; i++)
msg += FontList[i] + " ";
g.drawString(msg, 4, 16);
}
```



11.7.2 Creating and Selecting a Font

select a new font, you must first construct a **Font** object that describes that font. **Font** constructor has this general form:

Font(String *fontName*, int *fontStyle*, int *pointSize*)

Here, *fontName*--→ specifies the name of the desired font. The name can be specified using either the logical or face name.

Example fonts: Dialog, DialogInput, Sans Serif, Serif, Monospaced, and Symbol.

fontStyle -→ The style of the font is specified by *fontStyle*.

Font.PLAIN, **Font.BOLD**, and **Font.ITALIC**.

pointSize.-→ The size, in points, of the font is specified by *pointSize*.

To use a font that you have created, you must select it using **setFont( )**, which is defined by **Component**. It has this general form:

void setFont(Font *fontObj*)

Here, *fontObj* is the object that contains the desired font.