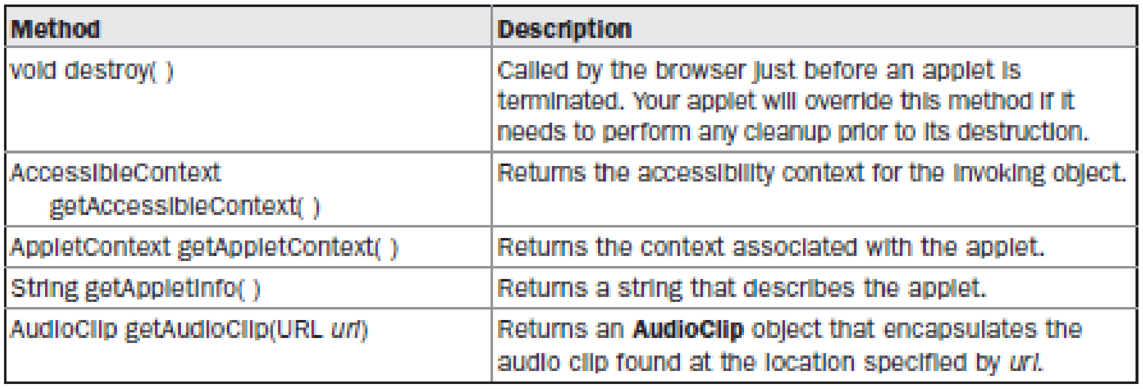
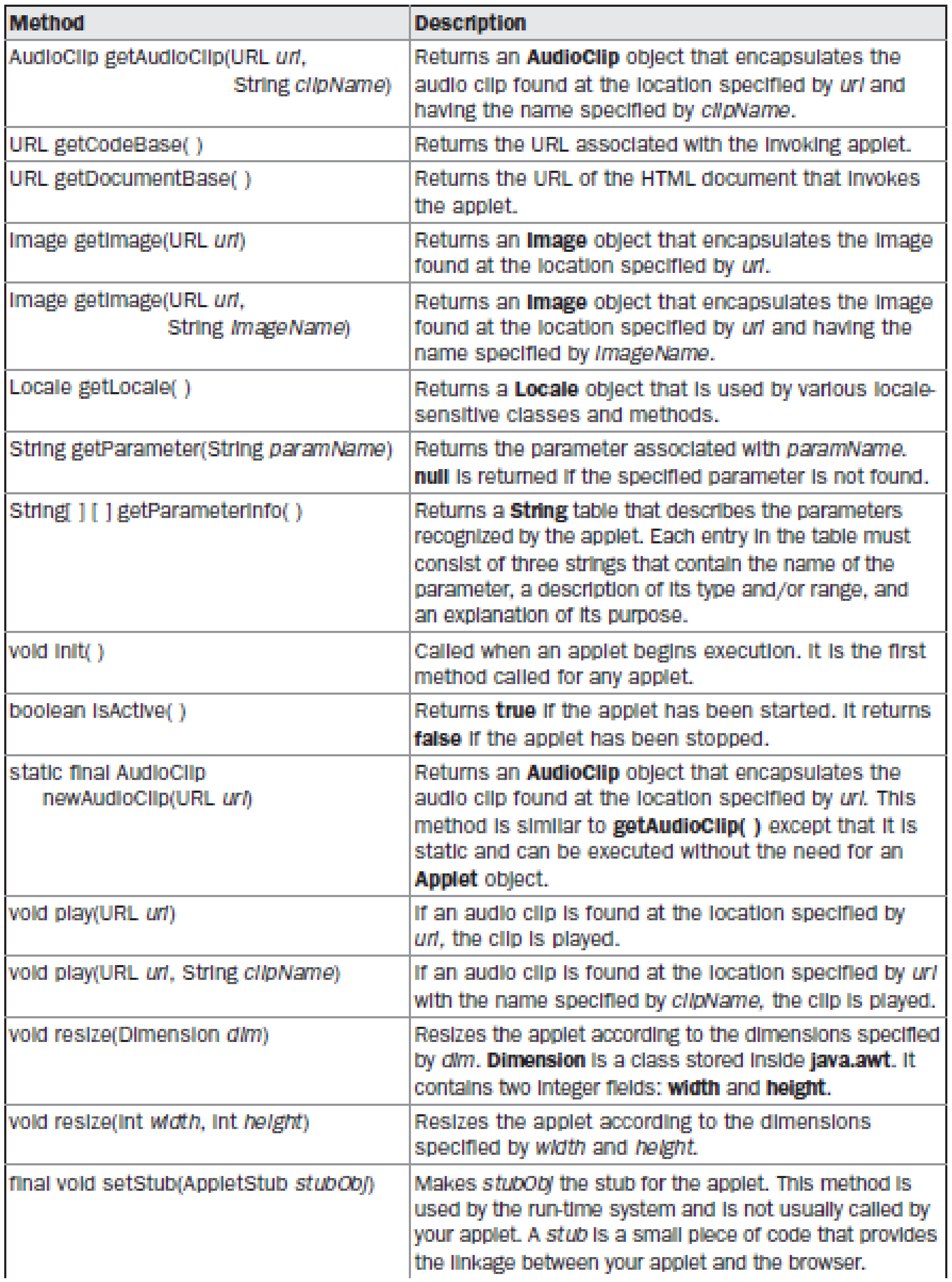
Applets

- Applet is a small program that
  - can be placed on a web page
  - will be executed by the web browser
  - give web pages "dynamic content".
  - Java Applets enable user interaction with GUI elements
  - Applets are Java programs that can be embedded in HTML documents
  - When browser loads Web page containing applet,Applet downloads into **Web browser** and begins execution or applets can be executed in **appletviewer.**
  - Applets are not stand alone programs.
  - Applets are specified in html document by using applet tag
  - /* <applet code="MyApplet" width=200 height=100> </applet> */
  - [thus applet will be executed in java enabled web browser when it encounters applet tag within the html file.]
- **The Applet class**
  - Applet class provides all necessary methods to start and stop the applet program.
  - It also provides methods to load and display images, and play audio clips.
  - Applet extends the AWT class Panel.
  - Panel extends Container which extends Component.



Method	Description
void destroy( )	Called by the browser just before an applet is terminated. Your applet will override this method if it needs to perform any cleanup prior to its destruction.
AccessibleContext getAccessibleContext( )	Returns the accessibility context for the invoking object.
AppletContext getAppletContext( )	Returns the context associated with the applet.
String getAppletInfo( )	Returns a string that describes the applet.
AudioClip getAudioClip(URL *url*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url*.



Method	**Description**
AudioClip getAudioClip(URL *url*, String *clipName*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url* and having the name specified by *clipName*.
URL getCodeBase( )	Returns the URL associated with the invoking applet.
URL getDocumentBase( )	Returns the URL of the HTML document that invokes the applet.
Image getImage(URL *url*)	Returns an **Image** object that encapsulates the image found at the location specified by *url*.
Image getImage(URL *url*, String *imageName*)	Returns an **Image** object that encapsulates the image found at the location specified by *url* and having the name specified by *imageName*.
Locale getLocale( )	Returns a **Locale** object that is used by various locale-sensitive classes and methods.
String getParameter(String *paramName*)	Returns the parameter associated with *paramName*. **null** is returned if the specified parameter is not found.
String[ ] [ ] getParameterInfo( )	Returns a **String** table that describes the parameters recognized by the applet. Each entry in the table must consist of three strings that contain the name of the parameter, a description of its type and/or range, and an explanation of its purpose.
void init( )	Called when an applet begins execution. It is the first method called for any applet.
boolean isActive( )	Returns **true** if the applet has been started. It returns **false** if the applet has been stopped.
static final AudioClip newAudioClip(URL *url*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url*. This method is similar to **getAudioClip( )** except that it is static and can be executed without the need for an **Applet** object.
void play(URL *url*)	If an audio clip is found at the location specified by *url*, the clip is played.
void play(URL *url*, String *clipName*)	If an audio clip is found at the location specified by *url* with the name specified by *clipName*, the clip is played.
void resize(Dimension *dim*)	Resizes the applet according to the dimensions specified by *dim*. **Dimension** is a class stored inside **java.awt**. It contains two integer fields: **width** and **height**.
void resize(int *width*, int *height*)	Resizes the applet according to the dimensions specified by *width* and *height*.
final void setStub(AppletStub *stubObj*)	Makes *stubObj* the stub for the applet. This method is used by the run-time system and is not usually called by your applet. A *stub* is a small piece of code that provides the linkage between your applet and the browser.

Applet Architecture

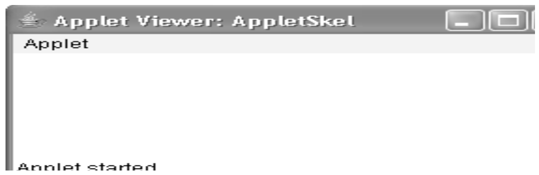
- An applet is a window-based program. As such, its architecture is different from the console-based programs.
- Applets are event driven.
- An applet waits until an event occurs.
- The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet.
- Once this happens, the applet must take appropriate action and then quickly return.
- Applet must perform specific actions in response to events and then return control to the run-time system.
- In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), an additional thread of execution must be started.
- The user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond.
- For example, when the user clicks the mouse inside the applet's window, a mouse-clicked event is generated.
- If the user presses a key while the applet's window has input focus, a keypress event is generated.
- Applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

An Applet Skeleton

- It defines init(),start(),stop(),destroy() methods.
- AWT-based applets will override paint() method defined by AWT Component class. This method is called when applet's output must be redisplayed.

```
import java.awt.*;
 import java.applet.*;
/*
<applet code=”AppletSkel”       width=300 height=200>
</applet>
*/
 public class AppletSkel extends Applet
 {
        // Called first.
        public void init()
        {
                // initialization
        }
        /* Called second, after init(). Also called whenever the applet is restarted. */
        public void start()
       {
                // start or resume execution
       }
        // Called when the applet is stopped.
        public void stop()
        {
                // suspends execution
        }
        /* Called when applet is terminated. This is the last method executed. */
        public void destroy()
        {
                // perform shutdown activities
        }
        // Called when an applet's window must be restored.
        public void paint(Graphics g)
        {
                // redisplay contents of window
       }
        }
```

When run, it generates the following window when viewed with an Appletviewer



.

-Applet Initialization and Termination

- Applet methods are called in the following order
- When an applet begins, the following methods are called, in this sequence:
  - 1. init( )
  - 2. start( )
  - 3. paint( )
- When an applet is terminated, the following sequence of method calls takes place
  - 1. stop( )
  - 2. destroy( )

- **init( ):** The init( ) method is the first method to be called. Initialization of variables is done here. This method is called only once during the run time of applet.
- **start( ):** The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. Whereas init( ) is called once—the first time an applet is loaded. start( ) is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start().
- **paint( ):**
  - The paint( ) method is called each time when applet's output must be redrawn.
  - For example, the window in which the applet is running may be overwritten by another window and then uncovered.
  - Or the applet window may be minimized and then restored.
  - paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called.
  - The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running.
- **stop( ):**
  - The stop( ) method is called when a web browser leaves the HTML document containing the applet—when it goes to another page
  - For example. stop( ) is called,when the applet is probably running. stop( ) is used to suspend threads that don't need to run when the applet is not visible.
- **destroy( )**
  - The destroy( ) method is called when the environment determines that the applet needs to be removed completely from memory.
  - We should free up any resources the applet may be using.
  - The stop( ) method is always called before destroy( ).

Overriding update( )

- AWT, defines a method called update( ). This method is called when applet has requested that a portion of its window be redrawn.
- update( ) method simply calls paint( ).

Simple Applet Display Methods

- AWT-based applets use AWT to perform input and output.
- to output a string to an applet, drawString( ) method is used, which is a member of the Graphics class. Typically, it is called from within either update( ) or paint().
- It has the following general form:
  void drawString(String message, int x, int y) Here, message is the string to be displayed at x,y location. In a Java window, the upper- left corner is location 0,0.
- To set the background color of an applet's window, setBackground( ) method is used.
- To set the foreground color setForeground( ) method is used.
- These methods are defined by Component, and they have the following general forms:
  - void setBackground(Color newColor)
  - void setForeground(Color newColor)
  - Here, newColor specifies the new color.

The class Color defines the constants shown here that can be used to specify colors:

Color.black	Color.magenta	Color.blue	Color.orange	Color.cyan
Color.pink	Color.darkGray	Color.red	Color.gray	Color.white
Color.green	Color.yellow	Color.lightGray		

- The following example sets the background color to green and the text color to red:
  setBackground(Color.green);
  setForeground(Color.red);
- A good place to set the foreground and background colors is in the init( ) method.
- We can obtain the current settings for the background and foreground colors by calling getBackground( ) and getForeground( ), respectively.
- They are also defined by Component ,they are:
  Color getBackground( ) Color getForeground( )
- Here is a very simple applet that sets the background color to cyan, the foreground color to red, and displays a message that illustrates the order in which the init( ), start( ), and paint( ) methods are called when an applet starts up:

```
/* A simple applet that sets the foreground and background colors and outputs a string. */
 import java.awt.*;
import java.applet.*;
 /*<applet code=”Sample” width=300 height=200>

</applet> */
public class Sample extends Applet
{
        String msg;
        // set the foreground and background colors. public void init()
        {
           setBackground(Color.cyan); setForeground(Color.red);
                msg = "Inside init( ) --";
        }
        // Initialize the string to be displayed. public void start()
        {
                msg += " Inside start( ) --";0
        }
        // Display msg in applet window. public void
        paint(Graphics g)
        {
                msg += " Inside paint( )."; g.drawString(msg, 10, 30);
        }
}
```

Requesting Repainting

- As a general rule, an applet writes to its window only when its update( ) or paint( ) method is called by the AWT.
- An applet must quickly return control to the run-time system.(constraint)
- It cannot create a loop inside paint( ) that repeatedly scrolls the banner. This would prevent control from passing back to the AWT.
- If applet needs to update the information displayed in the window, it simply calls repaint( ).
- The repaint( ) method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet’s update( ) method, which by default, calls paint( ).
- The AWT will then execute a call to paint( ), which can display the stored information.
- The repaint( ) method has four forms.
  The simplest version of repaint( ) is shown here:
      void repaint( )
      This version causes the entire window to be repainted.

- The following version specifies a region that will be repainted: void repaint(int left, int top, int width, int height)
- These dimensions are specified in pixels.
- Update may not be called immediately if system is slow or busy. In this case the following forms of repaint( ) are used:
  void repaint(long maxDelay)
  void repaint(long maxDelay, int x, int y, int width, int height)
- Here, maxDelay specifies the maximum number of milliseconds that can elapse before update( ) is called.
- It is possible for a method other than paint( ) or update( ) to output to applet window,to do so it must obtain a graphics context by calling getGraphics( ) (defined by Component) and then use this context to output to the window.

A Simple Banner Applet

- To demonstrate repaint( ), a simple banner applet is developed. This applet scrolls a message, from right to left, across the applet's window.
- The scrolling of the message is a repetitive task, it is performed by a separate thread, created by the applet when it is initialized.

```
import java.awt.*;
import java.applet.*;
    /*<applet code=SimpleBanner width=300 height=200>
   </applet> */
   public class SimpleBanner extends Applet implements Runnable
    {
          String msg = " A Simple Moving Banner.";
          Thread t = null;
          int state;
          boolean stopFlag;
         // Set colors and initialize thread.
         public void init()
          {
                setBackground(Color.cyan);
                setForeground(Color.red);
          }
         // Start thread
         public void start()
          {
                 t = new Thread(this);
                 stopFlag = false;
                 t.start();

          }
```

```
        // Entry point for the thread that runs the banner.
        public void run()
        {
                char ch;
                // Display banner
                for( ; ; )
                {
                        try
                        {
                                repaint();
                                Thread.sleep(250);
                                ch = msg.charAt(0);
                                msg = msg.substring(1, msg.length());
                                msg += ch;
                                if(stopFlag)
                                break;
                        } catch(InterruptedException e) {}
                }
        }
        // Pause the banner.
        public void stop()
        {
                stopFlag = true;
                t = null;
        }
        // Display the banner.
        public void paint(Graphics g)
        {
                g.drawString(msg, 50, 30);
        }
}
```

- SimpleBanner extends Applet and implements Runnable Interface.
- It is necessary to implement Runnable interface since the applet will be creating a second thread of execution that will be used to scroll the banner.
- Inside init( ), the foreground and background colors of the applet are set.
- After initialization, the run-time system calls start( ) to start the applet running.
- Inside start( ), a new thread of execution is created and assigned to the Thread variable t.
- Then, the boolean variable stopFlag, which controls the execution of the applet, is set to false.
- the thread is started by a call to t.start( ).
- t.start( ) calls run( ) to begin executing.
- Inside run( ), the characters in the string contained in msg are repeatedly rotated left.
- Between each rotation, a call to repaint( ) is made. This eventually causes the paint( )

method to be called, and the current contents of msg are displayed.

- Between each iteration, run( ) sleeps for a quarter of a second.
- The stopFlag variable is checked on each iteration. When it is true, the run( ) method terminates.
- If a browser is displaying the applet when a new page is viewed, the stop( ) method is called, which sets stopFlag to true, causing run( ) to terminate.

Using the Status Window

- An applet can also output a message to the status window of the browser or applet viewer on which it is running.
- showStatus( ) method displays the msg in the status window which is passed as a parameter to it.
- The following applet demonstrates showStatus( ):

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/* <applet code StatusWindow width=300 height=100>
</applet>*/
public class StatusWindow extends Applet
        {
                public void init()
                {
                setBackground(Color.cyan);
                }
// Display msg in applet window.
                public void paint(Graphics g)
                {
                g.drawString("This is in the applet window.", 10, 20);
                 showStatus("This is shown in the status window.");
                }
        }
```

Sample output from this program is shown here

The HTML APPLET Tag

- The APPLET tag can be used to start an applet from both an HTML document and from an applet viewer.
- An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers will allow many applets on a single page.
- Bracketed items are optional.

```
 < APPLET
[CODEBASE = codebaseURL]
 CODE = appletFile
 [ALT = alternateText]
 [NAME = appletInstanceName]
WIDTH = pixels HEIGHT = pixels
[ALIGN = alignment]
 [VSPACE = pixels]
[HSPACE = pixels] >
[< PARAM NAME = AttributeName VALUE = AttributeValue>] [< PARAM NAME =
AttributeName2 VALUE =AttributeValue>]

 [HTML Displayed in the absence of Java]
</APPLET>
```

- CODEBASE
  CODEBASE is an optional attribute that specifies the base URL of the applet code
- The HTML document's URL directory is used as the CODEBASE if this attribute is not specified.
- CODE
  CODE is a required attribute that gives the name of the file containing your applet's compiled .class file.
- ALT
  The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLET tag but can't currently run Java applets.
- NAME
  NAME is an optional attribute used to specify a name for the applet instance.
  To obtain an applet by name, getApplet( ) methos is ised, which is defined by the AppletContext interface.
- WIDTH and HEIGHT
  WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.
- ALIGN
  ALIGN is an optional attribute that specifies the alignment of the applet.with values:

LEFT, RIGHT, TOP, BOTTOM, MIDDLE,
BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

- VSPACE and HSPACE
  These attributes are optional.
  VSPACE specifies the space, in pixels, above and below the applet.
  HSPACE specifies the space, in pixels, on each side of the applet.
- PARAM NAME and VALUE
  The PARAM specifies applet-specific arguments in an HTML page. Applets access their attributes with the getParameter( ) method.

Passing Parameters to Applets

- The APPLET tag in HTML allows to pass parameters to applet.
- To retrieve a parameter, getParameter( ) method is used.
- It returns the value of the specified parameter in the form of a String object.
- Thus, for numeric and boolean values,its need to convert their string representations into their internal formats.
- Here is an example that demonstrates passing parameters:

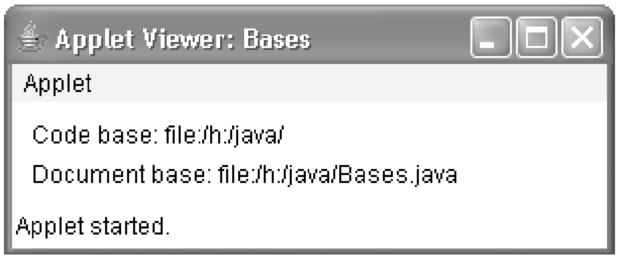
```
 // Use Parameters
import java.awt.*;
import java.applet.*;
/* <applet code=”ParamDemo” width=300 height=200>
</applet> */
 public class ParamDemo extends Applet
{
          String fontName;
          int  fontSize;
          float leading;
          boolean active;
        // Initialize the string to be displayed.
        public void start()
         {
                    String param;
                     fontName = getParameter("fontName");
                    if(fontName == null)
                     fontName = "Not Found";
                    param = getParameter("fontSize");
                    try
                    {
                            if(param != null) // if not found
                                    fontSize = Integer.parseInt(param);
                            else
                            fontSize = 0;
                    } catch(NumberFormatException e) {
                            fontSize = -1;
                            }
                    param = getParameter("leading");
                    try {
                            if(param != null) // if not found
                            leading = Float.valueOf(param).floatValue();
                    else
                            leading = 0;
                    } catch(NumberFormatException e) {
                            leading = -1;
                    }
                    param = getParameter("accountEnabled");
                    if(param != null)
                            active = Boolean.valueOf(param).booleanValue();
                    }
                    // Display parameters.
                    public void paint(Graphics g)
                    {
                            g.drawString("Font name: " + fontName, 0, 10);
                            g.drawString("Font size: " + fontSize, 0, 26);
                            g.drawString("Leading: " + leading, 0, 42);
                            g.drawString("Account Active: " + active, 0, 58);
                    }
        }
```

getDocumentBase( ) and getCodeBase( )

- Java allows applet to load data from the directory holding the HTML file that started the applet (the document base)
- and the directory from which the applet's class file was loaded (the code base).
- These directories are returned as URL objects by getDocumentBase( ) and getCodeBase( ).
- To actually load another file, will use the showDocument( ) method defined by the AppletContext interface.

```
 import java.awt.*;
import java.applet.*;
import java.net.*;
 /*<applet code="Bases" width=300 height=200> </applet>*/
 public class Bases extends Applet
 {
        // Display code and document bases.
        public void paint(Graphics g)
        {
            String msg;
             URL url = getCodeBase();
            // get code base
            msg = "Code base: " + url.toString();
            g.drawString(msg, 10, 20);
            url = getDocumentBase();
            // get document base
             msg = "Document base: " + url.toString();
             g.drawString(msg, 10, 40);
        }
    }
```

Sample output from this program is shown here:



```
Code base: file:/h:/java/
Document base: file:/h:/java/Bases.java
Applet started.
```

AppletContext and showDocument( )

- One application of Java is to use active images and animation to provide a graphical means of navigating the Web that is more interesting than simple text-based links.
- To allow applet to transfer control to another URL, we use showDocument( ) method defined by the AppletContext interfac
- The context of the currently executing applet is obtained by a call to the getAppletContext( ) method defined by Applet.
- This method has no return value and throws no exception if it fails.
- There are two showDocument( ) methods.
- The method showDocument(URL) displays the document at the specified URL.
- The method showDocument(URL, String) displays the specified document at the specified location within the browser window.
- The methods defined by AppletContext are shown in

Method	Description
Applet getApplet(String appletName)	Returns the applet specified by appletName if it is within the current applet context. Otherwise, null is returned.
Enumeration<Applet> getApplets( )	Returns an enumeration that contains all of the applets within the current applet context.
AudioClip getAudioClip(URL url)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.
Image getImage(URL url)	Returns an Image object that encapsulates the image found at the location specified by url.
InputStream getStream(String key)	Returns the stream linked to key. Keys are linked to streams by using the setStream( ) method. A null reference is returned if no stream is linked to key.Iterator<String> getStreamKeys( ) Returns an iterator for the keys associated with the invoking object. The keys are linked to streams. See getStream( ) and setStream( ).
void setStream(String key, InputStream strm)	Links the stream specified by strm to the key passed in key. The key is deleted from the invoking object if strm is null.
void showDocument(URL url)	Brings the document at the URL specified by url into view. This method may not be supported by applet viewers.
void showDocument(URL url, String where)	Brings the document at the URL specified by url into view. This method may not be supported by applet viewers. The placement of the document is specified by where as described in the text.

void showStatus(String str)	Displays str in the status window.

- Upon execution, it obtains the current applet context and uses that context to transfer control to a file called Test.html. This file must be in the same directory as the applet.

```
 import java.awt.*;
 import java.applet.*;
 import java.net.*;
/*<applet code=”ACDemo” width=300 height=200>
</applet> */
public class ACDemo extends Applet
        {
        public void start()
        {
                AppletContext ac = getAppletContext();
                URL url = getCodeBase();
                // get url of this applet
                try
                {
                 ac.showDocument(new URL(url+"Test.html"));
                 }
                catch(MalformedURLException e)
                {
                showStatus("URL not found");
                }
        }
       }
```

The AudioClip Interface

- The AudioClip interface defines these methods: play( ) (play a clip from the beginning), stop( ) (stop playing the clip), and loop( ) (play the loop continuously).
- After its loaded ,using getAudioClip( ), we can use these methods to play it.

The AppletStub Interface

- The AppletStub interface provides the means by which an applet and the browser (or applet viewer) communicate.

Swings

- Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT.
- In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables.
- Even familiar components such as buttons have more capabilities in Swing. For example, a button may have both an image and a text string associated with it. Also, the image can be changed as the state of the button changes.
- Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and, therefore, are platform-independent. The term *lightweight* is used to describe such elements.
- Swing are built on AWT.

Explain two key features of Swing.

1. Swing components are light weight

   They are entirely written in java they does not map to native platform specific code. More flexible and more efficient. Not in rectangular shapes.

2. Swing supports a pluggable look and feel.

   It becomes possible to change the that component is rendered with out affecting any of its other aspects. Possible to create new look and feel for any given component with out side effects. Look and feel is simply plugged in.

Briefly explain Container and Component of Swing.

- A Swing GUI consists of two key items: Components and Container
- A term component is an independent visual control such as push button or slider.
- A container holds group of components. Thus container is special kind of component that holds that is designed to hold other components. Container are also called components so container can hold other container.

Components

Swing components are derived from JComponent Class. Supports pluggable look and feel. It inherits Component and Container of AWT.

Swing Components : JButton, JCheckBox, JComboBox ,JTree ,JLabel, JTable ,JPanel …etc

Container

-Swing defines two types of heavy weight container. JFrame , JApplet

-Others are light weight containers.

-Top level containers should be declared first like JFrame and JApplet.

-Light weight containers example if JPanel. This is used to manage group of related components

-JPanel is used to create subgroups of related components that are contained with in another container.

Examples of containers

JPanel is Swing's version of the AWT class Panel and uses the same default layout,FlowLayout. JPanel is descended directly from JComponent.

JFrame is Swing's version of Frame and is descended directly from that class. The components added to the frame are referred to as its contents; these are managed by the contentPane. To add a component to a JFrame, we must use its contentPane instead.

JWindow is Swing's version of Window and is descended directly from that class. LikeWindow, it uses BorderLayout by default.

JDialog is Swing's version of Dialog and is descended directly from that class. Like Dialog, it uses BorderLayout by default. Like JFrame and JWindow,

Explain MVC architecture of swing:

- In general a visual component is composite of three distinct aspects The way that the component looks when rendered on the screen The way the component reacts to the user.
- The state information associated with the component
- The architecture has proven itself to be effective is MVC
- MVC mean Model View Controller.
- Model corresponds to the state information associated with the component. For example in case of check Box model contained a field that indicates if check box ix checked or unchecked.
- View determines how the component is displayed on the screen
- Controller determines how the component react to the user after that result in the view is updated.
- By separating model , view , and controller the specific implementation of one model can

be changed with out affecting other model.

- The MVC architecture sounds good but in swing separating view and controller is not beneficial.
- Swing uses modified version of MVC called UI delegate. For this reason swings approach is called **Model delegate** architecture.
- Swings pluggable look and feel is possible by its model delegate architecture.
- Because view and controller are separate look and feel can be changed without affecting the component.

A Simple Swing Application

```
// A simple Swing application.
import javax.swing.*;
class SwingDemo
 {
        SwingDemo()
        {
                // Create a new JFrame container.
                JFrame jfrm = new JFrame("A Simple Swing Application");
                // Give the frame an initial size.
                jfrm.setSize(275, 100);
                // Terminate the program when the user closes the application.
                jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                // Create a text-based label.
                JLabel jlab = new JLabel(" Swing means powerful GUIs.");
                // Add the label to the content pane.
                jfrm.add(jlab);
                // Display the frame.
                jfrm.setVisible(true);
        }
        public static void main(String args[])
        {
                // Create the frame on the event dispatching thread.
                SwingUtilities.invokeLater(new Runnable()
                {
                        public void run()
                        {
                                new SwingDemo();
                        }
                });
        }
}
```

Class	Description
AbstractButton	Abstract super class for Swing buttons.
ButtonGroup	Encapsulates a mutually exclusive set of buttons.
Image	Icon encapsulates an icon.
JApplet	The Swing version of Applet.
JButton	The Swing push button class.
JCheckBox	The Swing check box class.
JComboBox	Encapsulates a combo box (an combination of a drop-down listand text field).
JLabel	The Swing version of a label.
JRadioButton	The Swing version of a radio button.
JScrollPane	Encapsulates a scrollable window.
JTabbedPane	Encapsulates a tabbed window.
JTable	Encapsulates a table-based control.
JTextField	The Swing version of a text field.
JTree	Encapsulates a tree-based control.

The Swing-related classes are contained in **javax.swing**

JLabel: Swing labels are instances of the **JLabel** class, which extends **JComponent**. It can display text and/or an icon. Some of its constructors are shown here:

JLabel(Icon i)

Label(String s)

JLabel(String s, Icon i, int align)

Here, *s* and *i* are the text and icon used for the label.

The *align* argument is either **LEFT**,**RIGHT**, or **CENTER**

The text associated with the label can be read and written by the following

methods: String getText( ) ,void setText(String s)

Example Program

import javax.swing.*;

```
import javax.swing.*;
class SwingDemo
{
SwingDemo()
{
        JFrame jfrm=new JFrame("A simple Swing Application");
        jfrm.setSize(275,100);
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel jlab=new JLabel("Swing means power ful GUI");
        jfrm.add(jlab);
        jfrm.setVisible(true);
}
public static void main(String args[]) {
 ob= new SwingDemo();
}
}
```

Text Fields

The Swing text field is encapsulated by the **JTextComponent** class, which extends **JComponent**.

It provides functionality that is common to Swing text components.

One of its subclasses is **JTextField**, which allows you to edit one line of text. Some of its constructors are shown here:

```
JTextField( )
JTextField(int cols)
JTextField(String s, int cols)
JTextField(String s)
```

Here, *s* is the string to be presented, and *cols* is the number of columns in the text field. The following example illustrates how to create a text field. The applet begins by getting its content pane, and then a flow layout is assigned as its layout manager. Next, a **JTextField** object is created and is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
```

```
</applet>
*/
public class JTextFieldDemo extends JApplet {
JTextField jtf;
public void init() {
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
jtf = new JTextField(15);
contentPane.add(jtf);
}
}
```

JButton

The **JButton** class provides the functionality of a push button. **JButton** allows an icon, a string, or both to be associated with the push button. Some of its constructors are shown here:

JButton(Icon i)

JButton(String s)

JButton(String s, Icon i)

Here, *s* and *i* are the string and icon used for the button.

The following program displays a button on swing frame and displays string “SayHello”

```
import javax.swing.*;
import java.awt.*;
public class First {
        JFrame jf;
        public First()
        {
        jf=new JFrame("My  window");
        JButton btn= new JButton("say Hello");
        jf.add(btn);
        jf.setLayout(new FlowLayout());
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setSize(400,400);
        jf.setVisible(true);
        }
        public static void main(String[] args) {
                new First();
        }

}
```

Check Boxes

The **JCheckBox** class, which provides the functionality of a check box, is a concrete implementation of **AbstractButton**. Some of its constructors are shown here:

JCheckBox(Icon i)

JCheckBox(Icon i, boolean state)

JCheckBox(String s)

JCheckBox(String s, boolean state)

JCheckBox(String s, Icon i)

JCheckBox(String s, Icon i, boolean state)

Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the check box is initially selected. Otherwise, it is not. The state of the check box can be changed via the following method: void setSelected(boolean state) Here, *state* is **true** if the check box should be checked.

1.The following example illustrates how to create an applet that displays four check boxes and a text field. When a check box is pressed, its text is displayed in the text field.

2.) flow layout is assigned as its layout manager.

3.).four check boxes are added to the content pane, and icons are assigned for the normal, rollover, and selected states. The applet is then registered to receive item events.

Finally, a text field is added to the JFrame. When a check box is selected or deselected, an item event is generated. This is handled by **itemStateChanged( )**. Inside **itemStateChanged( )**, the **getItem( )** method gets the **JCheckBox** object that generated the event. The **getText( )** method gets the text for that check box and uses it to set the text inside the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo extends JFrame
implements ItemListener {
JTextField jtf;
JCheckBoxDemo()
{
        setLayout(new FlowLayout());


        JCheckBox cb = new JCheckBox("C");
        cb.addItemListener(this);
        add(cb);

        cb = new JCheckBox("C++");
        cb.addItemListener(this);
        add(cb);

        cb = new JCheckBox("Java");
        addItemListener(this);
        add(cb);

        cb = new JCheckBox("Perl", normal);
        addItemListener(this);
        add(cb);

        jtf = new JTextField(15);
        add(jtf);
}

public void itemStateChanged(ItemEvent ie) {
        JCheckBox cb = (JCheckBox)ie.getItem();
        jtf.setText(cb.getText());
}
Public static void main(String args[])
        {
        new JCheckBoxDemo();
        }
}
```

Combo Boxes

1. Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**.
2. A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. You can also type your selection into the text field.
3. Two of **JComboBox**'s

constructors are shown here:

JComboBox( )

JComboBox(array a)

Here, *a* is a array that initializes the combo box.

4. Items are added to the list of choices via the **addItem( )** method, whose signature is shown here: void addItem(Object obj)

Here, *obj* is the object to be added to the combo box.

The following example contains a combo box and a label. The label displays an icon. The combo box contains entries for "France", "Germany", "Italy", and "Japan". When a country is selected, the label is updated to display the flag for that country.

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

public class JComboBoxDemo extends JFrame
implements ItemListener {
JLabel jl;
ImageIcon france, germany, italy, japan;

public void init() {
setLayout(new FlowLayout());

JComboBox jc = new JComboBox();
jc.addItem("France");
jc.addItem("Germany");
jc.addItem("Italy");
jc.addItem("Japan");
jc.addItemListener(this);
add(jc);
jl = new JLabel();
add(jl);
}
public void itemStateChanged(ItemEvent ie) {
String s = (String)ie.getItem();
jl.setText(s);
}
}
```

Radio Buttons

Radio buttons are supported by the **JRadioButton** class, which is a concrete implementation of **AbstractButton**. Some of its constructors are shown here:

```
JRadioButton(Icon i)
JRadioButton(Icon i, boolean state)
JRadioButton(String s)
JRadioButton(String s, boolean state)
JRadioButton(String s, Icon i)
JRadioButton(String s, Icon i, boolean state)
```

Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at any time.

For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected.

The **ButtonGroup** class is instantiated to create a button group. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

```
void add(AbstractButton ab)
```

Here, *ab* is a reference to the button

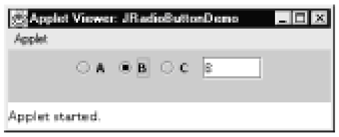
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JRadioButtonDemo extends JFrame
implements ActionListener
{
JTextField tf;
JRadioButtonDemo()
setLayout(new FlowLayout());
JRadioButton b1 = new JRadioButton("A");
b1.addActionListener(this);
add(b1);
JRadioButton b2 = new JRadioButton("B");
b2.addActionListener(this);
add(b2);
JRadioButton b3 = new JRadioButton("C");
b3.addActionListener(this);
add(b3);

// Define a button group
ButtonGroup bg = new ButtonGroup();
bg.add(b1);
bg.add(b2);
bg.add(b3);

// Create a text field and add it
// to the frame
tf = new JTextField(5);

add(tf);
}
public void actionPerformed(ActionEvent ae) {
tf.setText(ae.getActionCommand());
}
}
```



Tabbed Panes

1. A *tabbed pane* is a component that appears as a group of folders in a file cabinet. Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options.
2. Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent**. We will use its default constructor. Tabs are defined via the following method: void addTab(String str, Component comp)

Here, *str* is the title for the tab, and *comp* is the component that should be added to the tab. Typically, a **JPanel** or a subclass of it is added. The general procedure to use a tabbed pane in an applet is outlined here:

1. Create a **JTabbedPane** object.
2. Call **addTab( )** to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.)
3. Repeat step 2 for each tab.

4. Add the tabbed pane to the content pane of the applet. The following example illustrates how to create a tabbed pane.

```
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/


public class JTabbedPaneDemo extends JApplet {
public void init() {
JTabbedPane jtp = new JTabbedPane();
jtp.addTab("Cities", new CitiesPanel());
jtp.addTab("Colors", new ColorsPanel());
jtp.addTab("Flavors", new FlavorsPanel());
getContentPane().add(jtp);
}
}


class CitiesPanel extends JPanel {
public CitiesPanel() {
JButton b1 = new JButton("New York");
add(b1);
JButton b2 = new JButton("London");
add(b2);
JButton b3 = new JButton("Hong Kong");
add(b3);
JButton b4 = new JButton("Tokyo");
add(b4);
}
}

class ColorsPanel extends JPanel { public ColorsPanel() {
JCheckBox cb1 = new JCheckBox("Red"); add(cb1);
JCheckBox cb2 = new JCheckBox("Green"); add(cb2);
JCheckBox cb3 = new JCheckBox("Blue"); add(cb3);
}
}

class FlavorsPanel extends JPanel { public FlavorsPanel() {
JComboBox jcb = new JComboBox();
jcb.addItem("Vanilla"); jcb.addItem("Chocolate");
jcb.addItem("Strawberry");
add(jcb);
}
 }
```

Scroll Panes

1. A *scroll pane* is a component that presents a rectangular area in which a component may be viewed. Horizontal and/or vertical scroll bars may be provided if necessary.
2. Scroll panes are implemented in Swing by the **JScrollPane** class, which extends **JComponent**. Some of its constructors are shown here:

   JScrollPane(Component comp)

Here are the steps that you should follow to use a scroll pane in an applet:
1. Create a **JComponent** object.

2. Create a **JScrollPane** object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.)

3. Add the scroll pane to the content pane of the applet.

The following example illustrates a scroll pane. First, the content pane of the **JApplet** object is obtained and a border layout is assigned as its layout manager. Next, a **JPanel** object is created and four hundred buttons are added to it, arranged into twenty columns. The panel is then added to a scroll pane, and the scroll pane is added to the content pane. This causes vertical and horizontal scroll bars to appear. You can use the scroll bars to scroll the buttons into view.

```
import javax.swing.*;

import java.awt.BorderLayout;
import java.awt.GridLayout;


public class JScrollPaneDemo extends JApplet{

        public void init()
        {
                JPanel jp=new JPanel();
                jp.setLayout(new GridLayout(20,20));
        add(jp);

            int b=0;
            for(int i=0;i<20;i++)
                for(int j=0;j<20;j++)
                {
                jp.add(new JButton("Button"+b));
                    ++b;
                }
            JScrollPane jsp=new JScrollPane(jp);
            add(jsp,BorderLayout.CENTER);

        }


    }
```

Tables

A *table* is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Tables are implemented by the **JTable** class, which extends **JComponent**. One of its constructors is shown here:

JTable(Object data[ ][ ], Object colHeads[ ])

Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

Here are the steps for using a table in an applet:

- Create a **JTable** object.
- Create a **JScrollPane** object.
- Add the table to the scroll pane.
- Add the scroll pane to the content pane of the applet.

The following example illustrates how to create and use a table. array of strings is created for the column headings. This table has three columns. A two-dimensional array of strings is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane and then the scroll pane is added to the content pane.

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

public class JTableDemo extends JFrame {

        JTable jtbl;
        final String[] colHeads = { "Name", "Phone", "Fax" };

final Object[][] data = {{ "Gail", "4567", "8675" },{ "Ken", "7566", "5555" },{ "Viviane", "5634",
"5887" },
{ "Melanie", "7345", "9222" },{ "Anne", "1237", "3333" },{ "John", "5656", "3144" },{ "Matt", "5672",
"2176" },
{ "Claire", "6741", "4244" },{ "Erwin", "9023", "5159" },{ "Ellen", "1134", "5332" },{ "Jennifer",
"5689", "1212" },{ "Ed", "9030", "1313" },{ "Helen", "6751", "1415" }};

        JTableDemo()
        {
                setTitle("MY window");
```

```
            setSize(400,400);
            setLayout(new FlowLayout());
            jtbl=new JTable(data,col);
            add(jtbl);
            setVisible(true);
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
        public static void main(String[] args) {
            // TODO Auto-generated method stub
         new    JTableDemo();

        }

    }
```