The Applet Class

- An applet is a program written in the Java programming language that can be included in an HTML page, much in the same way an image is included in a page.
- When you use a Java technology-enabled browser to view a page that contains an applet, the applet's code is transferred to your system and executed by the browser's Java Virtual Machine (JVM).
- *Applets* are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document.
- After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run complex computations without introducing the risk of viruses or breaching data integrity.
- The **Applet** class is contained in the **java.applet** package.
- **Applet** contains several methods that give you detailed control over the execution of your applet.
- In addition, **java.applet** also defines three interfaces: **AppletContext**, **AudioClip**, and **AppletStub**.

Two Types of Applets

- It is important to state at the outset that there are two varieties of applets.
- The first are those based directly on the **Applet** class.
- These applets use the Abstract Window Toolkit (AWT) to provide the graphic user interface.
- This style of applet has been available since Java was first created.
- The second type of applets are those based on the Swing class **JApplet**.
- Swing applets use the Swing classes to provide the GUI.
- Swing offers a richer and often easier-to-use user interface than does the AWT.
- Thus, Swing-based applets are now the most popular.
- However, traditional AWT-based applets are still used, especially when only a very simple user interface is required.
- Because **JApplet** inherits **Applet**, all the features of **Applet** are also available in **JApplet**.

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

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

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

- ***paint()*** method is defined by the AWT and must be overridden by the applet.
- **paint( )** is called each time that the applet must redisplay its output.
- This situation can occur for several reasons.
- For example, the window in which the applet is running can be overwritten by another window and then uncovered.
- Or, the applet window can 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 contains the graphics context, which describes the graphics environment in which the applet is running.
- This context is used whenever output to the applet is required.

- Inside **paint( )** is a call to **drawString( )**, which is a member of the **Graphics** class.
- This method outputs a string beginning at the specified X,Y location.
- It has the following general form:
  void drawString(String *message*, int *x*, int *y*)
- Here, *message* is the string to be output beginning at *x,y*.
- In a Java window, the upper-left corner is location 0,0.
- The call to **drawString( )** in the applet causes the message “A Simple Applet” to be displayed beginning at location 20,20.
- Notice that the applet does not have a **main( )** method.
- Unlike Java programs, applets do not begin execution at **main( )**.
- Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

- After you enter the source code for **SimpleApplet**, compile in the same way that you have been compiling programs.

- However, running **SimpleApplet** involves a different process.

- In fact, there are two ways in which you can run an applet:
  - Executing the applet within a Java-compatible web browser.
  - Using an applet viewer, such as the standard tool, **appletviewer**. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

The Applet Class

- The **Applet** class defines various methods (Refer text book).

- **Applet** provides all necessary support for applet execution, such as starting and stopping.

- It also provides methods that load and display images, and methods that load and play audio clips.

- **Applet** extends the AWT class **Panel**.

- In turn, **Panel** extends **Container**, which extends **Component**.

- These classes provide support for Java's window-based, graphical interface.

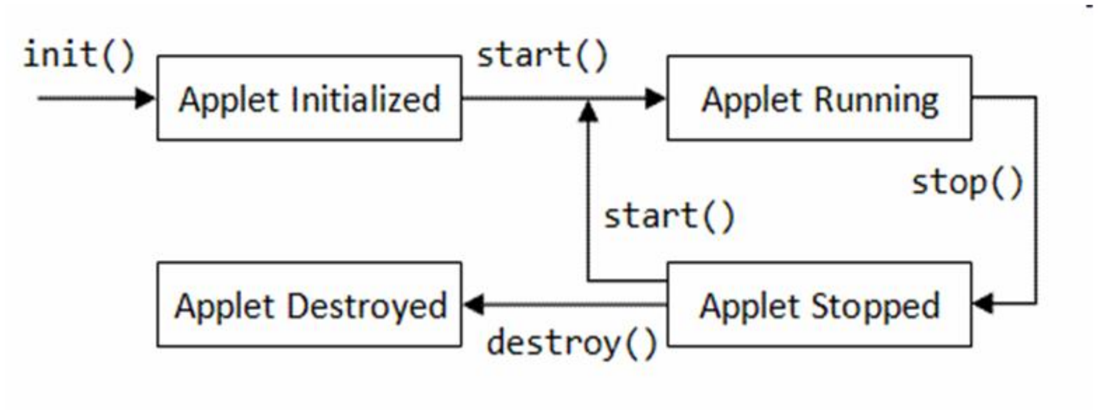
- Thus, **Applet** provides all of the necessary support for window-based activities.

The Applet Architecture

The Applet Skeleton

- Most of the applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution.
- Four of these methods, **init( )**, **start( )**, **stop( )**, and **destroy()**, apply to all applets and are defined by **Applet**.
- Default implementations for all of these methods are provided.
- Applets do not need to override those methods they do not use.
- However, only very simple applets will not need to define all of them.

The life cycle of an applet can be depicted using following diagram.



- AWT-based applets will also override the **paint()** method, which is defined by the AWT **Component** class.
- This method is called when the applet's output must be redisplayed.
- These five methods can be assembled into the skeleton shown here:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet
{
    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
}
}
```

Applet Initialization and Termination

- It is important to understand the order in which the various methods shown in the skeleton are called.

- When an applet begins, the following methods are called, in this sequence:
  - **init( )**
  - **start( )**
  - **paint( )**

- When an applet is terminated, the following sequence of method calls takes place:
  - **stop( )**
  - **destroy( )**

- Each of the method is explained in detail:

init( )

- The **init( )** method is the first method to be called.
- This is where you should initialize variables.
- This method is called only once during the run time of your applet.

start( )

- The **start( )** method is called after **init( )**.
- It is also called to restart an applet after it has been stopped.
- Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen.
- So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

paint( )

- The **paint( )** method is called each time your applet's output must be redrawn.
- This situation can occur for several reasons.
- 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.
- The **paint( )** method has one parameter of type **Graphics**.
- This parameter will contain the graphics context, which describes the graphics environment in which the applet is running.
- This context is used whenever output to the applet is required.

stop( )

- The **stop()** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example.
- When **stop()** is called, the applet is probably running.
- You should use **stop()** to suspend threads that don't need to run when the applet is not visible.
- You can restart them when **start()** is called if the user returns to the page.

destroy()

- The **destroy()** method is called when the environment determines that your applet needs to be removed completely from memory.
- At this point, you should free up any resources the applet may be using.
- The **stop()** method is always called before **destroy()**.

Overriding update()

- In some situations, your applet may need to override another method defined by the AWT, called **update()**.
- This method is called when your applet has requested that a portion of its window be redrawn.
- The default version of **update()** simply calls **paint()**.
- However, you can override the **update()** method so that it performs more subtle repainting.
- In general, overriding **update()** is a specialized technique that is not applicable to all applets.

Simple Applet Display Methods

- Applets are displayed in a window, and AWT-based applets use the AWT to perform input and output.
- To output a string to an applet, use **drawString( ),** which is usually 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 output beginning at *x,y*.
- In a Java window, the upper-left corner is location 0,0.
- The **drawString( )** method will not recognize newline characters.
- If you want to start a line of text on another line, you must do so manually, specifying the precise X,Y location where you want the line to begin.

- To set the background color of an applet's window, use **setBackground( )**.
- To set the foreground color (the color in which text is shown, for example), use **setForeground( )**.
- 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.gray	Color.blue	Color.orange
Color.cyan	Color.pink	Color.white	Color.darkGray	Color.red
Color.green	Color.yellow	Color.lightGray		

- A good place to set the foreground and background colors is in the **init( )** method.
- You can change these colors as often as necessary during the execution of your applet.
- You can obtain the current settings for the background and foreground colors by calling **getBackground( )** and **getForeground( )**, respectively.
- They are also defined by **Component** and are shown here:
  - Color getBackground( )
  - Color getForeground( )

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

/*
<applet code="Sample" width=300 height=100>
</applet>
*/

public class Sample extends Applet
{
    String msg1, msg2, msg3;

    public void init()
    {
          setBackground(Color.cyan);
          setForeground(Color.red);
          msg1 = "Inside init( )";
    }

    public void start()
    {
          msg2 = "Inside start( ) ";
    }
```

```
public void paint(Graphics g)
{
          msg3 = "Inside paint( ).";
          g.drawString(msg1, 10, 30);
          g.drawString(msg2, 10, 50);
          g.drawString(msg3, 10, 70);
 }
}
```

Requesting Repainting

- As a general rule, an applet writes to its window only when its **update( )** or **paint( )** method is called by the AWT.
- So, the question arises: how an applet window can update itself when the information within it changes?
- One of the fundamental architectural constraints imposed on an applet is that it must quickly return control to the run-time system.
- It cannot create a loop inside **paint( )** that repeatedly to update the information.
- This would prevent control from passing back to the AWT.
- Given this constraint, whenever your applet needs to update the information displayed in its 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, in its default implementation, calls **paint( )**.
- Thus, for another part of your applet to output to its window, simply store the output and then call **repaint( )**.
- The AWT will then execute a call to **paint( )**, which can display the stored information.
- The **repaint( )** method has four forms:
  - void repaint( ) : causes the entire window to be repainted.
  - void repaint(int *left*, int *top*, int *width*, int *height*) : specifies a region that will be repainted. Here, the coordinates of the upper-left corner of the region are specified by *left* and *top*, and the width and height of the region are passed in *width* and *height*. These dimensions are specified in pixels.
- You save time by specifying a region to repaint.
- Window updates are costly in terms of time.
- If you need to update only a small portion of the window, it is more efficient to repaint only that region.

- Calling **repaint( )** is essentially a request that your applet be repainted sometime soon.
- However, if your system is slow or busy, **update( )** might not be called immediately.
- Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that **update( )** is only called at irregular intervals.
- This can be a problem in many situations, including animation, in which a consistent update time is necessary.
- One solution to this problem is to use the following forms of **repaint( )**:
  - 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.
- If the time elapses before **update( )** can be called, it isn't called. There's no return value or exception thrown, so you must be careful.

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

/*
<applet code="SimpleBanner" width=300 height=50>
</applet>
*/

public class SimpleBanner extends Applet implements Runnable
{
    String msg = " A Simple Moving Banner.";
    Thread t = null;
    int state;
    boolean stopFlag;

    public void init()
    {       setBackground(Color.cyan);
            setForeground(Color.red);
    }

    public void start()             //Applet start() method
    {
            t = new Thread(this);
            stopFlag = false;
            t.start();              //Thread start() method
    }
public void run()
{
    char ch;
    for( ; ; )
    {       try
            {
                    repaint();
                    Thread.sleep(250);
                    ch = msg.charAt(0);
                    msg = msg.substring(1, msg.length());
                    msg += ch;

                    if(stopFlag)
                            break;
            } catch(InterruptedException e) {}
    }
}

  public void stop()
  {        stopFlag = true;
           t = null;
  }
  public void paint(Graphics g)
  {        g.drawString(msg, 50, 30);
  }
}
```

- 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**.
- Next, the thread is started by a call to **t.start( )**.
- Remember that **t.start( )** calls a method defined by **Thread**, which causes **run( )** to begin executing.
- It does not cause a call to the version of **start( )** defined by **Applet**. These are two separate methods.

- 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 net effect of **run( )** is that the contents of **msg** are scrolled right to left in a constantly moving display.
- 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.
- This is the mechanism used to stop the thread when its page is no longer in view.
- When the applet is brought back into view, **start( )** is once again called, which starts a new thread to execute the banner.

Using the Status Window

- In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running.
- To do so, call **showStatus( )** with the string that you want displayed.
- The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors.
- The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet.

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

/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/

public class StatusWindow extends Applet
{
    public void init()
    {
          setBackground(Color.cyan);
    }

    public void paint(Graphics g)
    {
          g.drawString("This is in the applet window.", 10, 20);
          showStatus("This is shown in the status window.");
    }
}
```

The HTML APPLET Tag

- The APPLET tag 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.
- So far, we have been using only a simplified form of the APPLET tag.
- The syntax for a fuller form of the APPLET tag is shown here (Items within square-brackets 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**
  is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.

- **CODE**
  is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

- **ALT**
  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. This is distinct from the alternate HTML you provide for browsers that don't support applets.

- **NAME**
  is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use **getApplet( )**, which is defined by the **AppletContext** interface.

- **WIDTH and HEIGHT**
  are required attributes that give the size (in pixels) of the applet display area.

- **ALIGN**
  is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

- **VSPACE and HSPACE**
  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. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

- **PARAM NAME and VALUE**
  The PARAM tag allows you to specify 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 you to pass parameters to your applet.

- To retrieve a parameter, use the **getParameter()** method.

- It returns the value of the specified parameter in the form of a **String** object.

- Thus, for numeric and **boolean** values, you will need to convert their string representations into their internal formats.

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

/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/

public class ParamDemo extends Applet
{
    String font;
    int Size;
    float leading;
    boolean active;
```

```
public void start()
{
    String param;
    font = getParameter("fontName");

    if(font == null)
        font = "Not Found";

    param = getParameter("fontSize");

    try
    {
        if(param != null)
            Size = Integer.parseInt(param);
        else
            Size = 0;
    } catch(NumberFormatException e)
    {
        Size = -1;
    }

    param = getParameter("leading");
try
{
  if(param != null)
    leading = Float.valueOf(param).floatValue();
  else
    leading = 0;
 } catch(NumberFormatException e)
 {
           leading = -1;
  }

  param = getParameter("accountEnabled");

   if(param != null)
      active = Boolean.valueOf(param).booleanValue();
 }

 public void paint(Graphics g)
 {
           g.drawString("Font name: " + font, 0, 10);
           g.drawString("Font size: " + Size, 0, 26);
           g.drawString("Leading: " + leading, 0, 42);
           g.drawString("Account Active: " + active, 0, 58);
  }
}
```

Event Handling

- Event handling is fundamental to Java programming because it is integral to the creation of applets and other types of GUI-based programs.
- Applets are event-driven programs that use a graphical user interface to interact with the user.
- Furthermore, any program that uses a graphical user interface, such as a Java application written for Windows, is event driven.
- Events are supported by a number of packages, including **java.util**, **java.awt**, and **java.awt.event**.
- Most events are generated when the user interacts with a GUI-based program.
- They are passed to the program through the specific method dependent upon the actual event.
- There are several types of events, including those generated by the mouse, the keyboard, and various GUI controls, such as a button, text box, check box etc.

Two Event Handling Mechanisms

- Before beginning our discussion of event handling, an important point must be made:
- The way in which events are handled changed significantly between the original version of Java (1.0) and modern versions of Java, beginning with version 1.1.
- The 1.0 method of event handling is still supported, but it is not recommended for new programs.
- Also, many of the methods that support the old 1.0 event model have been deprecated.
- The modern approach is the way that events should be handled by all new programs.

The Delegation Event Model

- The modern approach to handling events is based on the ***delegation event model***, which defines standard and consistent mechanisms to generate and process events.
- Its concept is quite simple: a ***source*** generates an event and sends it to one or more ***listeners***.
- In this scheme, the listener simply waits until it receives an event.
- Once an event is received, the listener processes the event and then returns.
- The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.
- A user interface element is able to “delegate” the processing of an event to a separate piece of code.
- In the delegation event model, listeners must register with a source in order to receive an event notification.
- This provides an important benefit: notifications are sent only to listeners that want to receive them.

Events

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

Event Sources

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

  public void add*Type*Listener(*Type*Listener *el*)
- Here, *Type* is the name of the event, and *el* is a reference to the event listener.

- For example, the method that registers a keyboard event listener is called **addKeyListener( )**.
- The method that registers a mouse motion listener is called **addMouseMotionListener( )**.
- When an event occurs, all registered listeners are notified and receive a copy of the event object.
- This is known as *multicasting* the event.
- In all cases, notifications are sent only to listeners that register to receive them.