mood-book

UNIT IV- APPLETS

What is an Applet?

According to Sun "An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application....The Applet class provides a standard interface between applets and their environment."

Four definitions of applet:

- A small application
- A secure program that runs inside a web browser
- A subclass of java.applet.Applet
- An instance of a subclass of java.applet.Applet

public class Applet extends Panel

```
java.lang.Object
  |
  + ---java.awt.Component
        |
        + ---java.awt.Container
              |
              + ---java.awt.Panel
                    |
                    + ---java.applet.Applet
```

Hello World: The Applet

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

public class HelloWorldApplet extends Applet {

  public void paint(Graphics g) {
    g.drawString("Hello world!", 50, 25);  }}
```

The applet version of HelloWorld is a little more complicated than the HelloWorld application, and it will take a little more effort to run it as well.

First type in the source code and save it into file called HelloWorldApplet.java. Compile this file in the usual way. If all is well a file called HelloWorldApplet.class will be created. Now you need to create an HTML file that will include your applet. The following simple HTML file will do.

```
<HTML>
<HEAD>
<TITLE> Hello World </TITLE>
</HEAD>

<BODY>
        <applet code="HelloWorldApplet.class" width="150" height="50">
        </applet>
</BODY>
</HTML>
```

Save this file as HelloWorldApplet.html in the same directory as the HelloWorldApplet.class file. When you've done that, load the HTML file into a Java enabled Web Browser..

The APPLET HTML Tag

Applets are embedded in web pages using the <APPLET> and </APPLET> tags. The <APPLET> tag is similar to the <IMG> tag. Like <IMG> <APPLET> references a source file that is not part of the HTML page on which it is embedded. IMG elements do this with the SRC attribute. APPLET elements do this with the CODE attribute. The CODE attribute tells the browser where to look for the compiled .class file. It is relative to the location of the source document.

The CODEBASE attribute is a URL that points at the directory where the .class file is. The CODE attribute is the name of the .class file itself. For instance if on the HTML page of the previous section you had written

```
<APPLET
        CODE="HelloWorldApplet.class"
```

```
        CODEBASE="classes"
        WIDTH=200 HEIGHT=200>
</APPLET>
```

then the browser would have tried to find HelloWorldApplet.class in the classes directory in the same directory as the HTML page that included the applet.

The HEIGHT and WIDTH attributes work exactly as they do with IMG, specifying how big a rectangle the browser should set aside for the applet. These numbers are specified in pixels and are required.

Spacing Preferences

The <APPLET> tag has several attributes to define how it is positioned on the page.

The ALIGN attribute defines how the applet's rectangle is placed on the page relative to other elements. Possible values include LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM and ABSBOTTOM. This attribute is optional.

You can specify an HSPACE and a VSPACE in pixels to set the amount of blank space between an applet and the surrounding text. The HSPACE and VSPACE attributes are optional.

```
<APPLET
        code="HelloWorldApplet.class"
        CODEBASE="http://www.foo.bar.com/classes"
        width=200
        height=200
        ALIGN=RIGHT
        HSPACE=5
        VSPACE=10>
</APPLET>
```

The ALIGN, HSPACE, and VSPACE attributes are identical to the attributes of the same name used by the <IMG> tag.

The <APPLET> has an ALT attribute. An ALT attribute is used by a browser that understands the APPLET tag but for some reason cannot play the applet. , then the browser should display the ALT text.

```
<applet code="HelloWorldApplet.class"
        CODEBASE="http://www.foo.bar.com/classes" width=200 height=200
        ALIGN=RIGHT HSPACE=5 VSPACE=10
        ALT="Hello World!">
</APPLET>
```

Naming Applets

You can give an applet a name by using the NAME attribute of the APPLET tag. This allows communication between different applets on the same Web page.

```
<APPLET
        code="HelloWorldApplet.class"
        Name=Applet1
        CODEBASE="http://www.foo.bar.com/classes"
        width=200
        height=200
        ALIGN=RIGHT HSPACE=5 VSPACE=10
        ALT="Hello World!">
        Hello World!<P>
</APPLET>
```

JAR Archives

HTTP 1.0 uses a separate connection for each request. When you're downloading many small files, the time required to set up and tear down the connections can be a significant fraction of the total amount of time needed to load a page. It would be better if you could load all the HTML documents, images, applets, and sounds a page needed in one connection.

One way to do this without changing the HTTP protocol, is to pack all those different files into a single archive file, perhaps a zip archive, and just download that.

You can pack all the images, sounds, and .class files an applet needs into one JAR archive and load that instead of the individual files. Applet classes do not have to be loaded directly. They can also be stored in JAR archives. To do this you use the ARCHIVES attribute of the APPLET tag

```
<APPLET
        CODE=HelloWorldApplet
```

```
        WIDTH=200 HEIGHT=100
        ARCHIVES="HelloWorld.jar">
        <hr>
                Hello World!
        <hr>
</APPLET>
```

In this example, the applet class is still HelloWorldApplet. However, there is no HelloWorldApplet.class file to be downloaded. Instead the class is stored inside the archive file HelloWorld.jar.

The OBJECT Tag

HTML 4.0 deprecates the <APPLET> tag. Instead you are supposed to use the <OBJECT> tag. For the purposes of ewbedding applets, the <OBJECT> tag is used almost exactly like the <APPLET> tag except that the class attribute becomes the classid attribute. For example,

```
<OBJECT classid="MyApplet.class"
        CODEBASE="http://www.foo.bar.com/classes" width=200 height=200
        ALIGN=RIGHT HSPACE=5 VSPACE=10>
</OBJECT>
```

Finding an Applet's Size

When running inside a web browser the size of an applet is set by the height and width attributes and cannot be changed by the applet. Many applets need to know their own size. After all you don't want to draw outside the lines. :-)

Retrieving the applet size is straightforward with the getSize() method. java.applet.Applet inherits this method from java.awt.Component. getSize() returns a java.awt.Dimension object. A Dimension object has two public int fields, height and width. Below is a simple applet that prints its own dimensions.

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

```
public class SizeApplet extends Applet {

  public void paint(Graphics g) {
    Dimension appletSize = this.getSize();
    int appletHeight = appletSize.height;
    int appletWidth = appletSize.width;

    g.drawString("This applet is " + appletHeight +
      " pixels high by " + appletWidth + " pixels wide.",
      15, appletHeight/2);
  } }
```

Passing Parameters to Applets

Parameters are passed to applets in NAME=VALUE pairs in <PARAM> tags between the opening and closing APPLET tags. Inside the applet, you read the values passed through the PARAM tags with the getParameter() method of the java.applet.Applet class.

The program below demonstrates this with a generic string drawing applet. The applet parameter "Message" is the string to be drawn.

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

public class DrawStringApplet extends Applet {

  private String defaultMessage = "Hello!";

  public void paint(Graphics g) {
    String inputFromPage = this.getParameter("Message");
    if (inputFromPage == null) inputFromPage = defaultMessage;
    g.drawString(inputFromPage, 50, 25);
  }

}
```

You also need an HTML file that references your applet. The following simple HTML file will do:

```
<HTML>
<HEAD>
```

```
<TITLE> Draw String </TITLE>
</HEAD>

<BODY>
        This is the applet:<P>
        <APPLET code="DrawStringApplet.class" width="300" height="50">
                <PARAM name="Message" value="Howdy, there!">
        </APPLET>
</BODY>
</HTML>
```

You pass getParameter() a string that names the parameter you want. This string should match the name of a <PARAM> tag in the HTML page. getParameter() returns the value of the parameter. All values are passed as strings.

The Basic Applet Life Cycle

1. The browser reads the HTML page and finds any <APPLET> tags.
2. The browser parses the <APPLET> tag to find the CODE and possibly CODEBASE attribute.
3. The browser downloads the .class file for the applet from the URL found in the last step.
4. The browser converts the raw bytes downloaded into a Java class, that is a java.lang.Class object.
5. The browser instantiates the applet class to form an applet object. This requires the applet to have a noargs constructor.
6. The browser calls the applet's init() method.
7. The browser calls the applet's start() method.
8. While the applet is running, the browser passes any events intended for the applet, e.g. mouse clicks, key presses, etc., to the applet's handleEvent() method. Update events are used to tell the applet that it needs to repaint itself.
9. The browser calls the applet's stop() method.
10. The browser calls the applet's destroy() method.

All applets have the following four methods:

```
public void init();
```

public void start();

public void stop();

public void destroy();

They have these methods because their superclass, java.applet.Applet, has these methods

In the superclass, these are simply do-nothing methods. For example,

public void init() {}

Subclasses may override these methods to accomplish certain tasks at certain times. For instance, the init() method is a good place to read parameters that were passed to the applet via <PARAM> tags because it's called exactly once when the applet starts up

The start() method is called at least once in an applet's life, when the applet is started or restarted. In some cases it may be called more than once. Many applets you write will not have explicit start()methods and will merely inherit one from their superclass. A start() method is often used to start any threads the applet will need while it runs.

The stop() method is called at least once in an applet's life, when the browser leaves the page in which the applet is embedded. The applet's start() method will be called if at some later point the browser returns to the page containing the applet. In some cases the stop() method may be called multiple times in an applet's life. Many applets you write will not have explicit stop()methods and will merely inherit one from their superclass.

The destroy() method is called exactly once in an applet's life, just before the browser unloads the applet. This method is generally used to perform any final clean-up.

Graphics Objects

In Java all drawing takes place via a Graphics object. This is an instance of the class java.awt.Graphics. Initially the Graphics object you use will be the one passed as an argument to an applet's paint() method.

Drawing Lines

Drawing straight lines with Java is easy. Just call

g.drawLine(x1, y1, x2, y2)

where (x1, y1) and (x2, y2) are the endpoints of your lines and g is the Graphics object you're drawing with.

This program draws a line diagonally across the applet.

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

public class SimpleLine extends Applet {

 public void paint(Graphics g) {
    g.drawLine(0, 0, this.getSize().width, this.getSize().height);
 }

}
```

Drawing Rectangles

Drawing rectangles is simple. Start with a Graphics object g and call its drawRect() method:

public void drawRect(int x, int y, int width, int height)

As the variable names suggest, the first int is the left hand side of the rectangle, the second is the top of the rectangle, the third is the width and the fourth is the height. This is in contrast to some APIs where the four sides of the rectangle are given.

This uses drawRect() to draw a rectangle around the sides of an applet.

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

public class RectangleApplet extends Applet {

  public void paint(Graphics g) {
    g.drawRect(0, 0, this.getSize().width - 1, this.getSize().height - 1);
  } }
```

Remember that getSize().width is the width of the applet and getSize().height is its height.

Clearing Rectangles

It is also possible to clear a rectangle that you've drawn. The syntax is exactly what you'd expect:

public abstract void clearRect(int x, int y, int width, int height)

This program uses clearRect() to blink a rectangle on the screen.

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

public class Blink extends Applet {
  public void paint(Graphics g) {
    int appletHeight = this.getSize().height;
    int  appletWidth  =  this.getSize().width;
    int rectHeight  = appletHeight/3;
    int rectWidth   = appletWidth/3;
    int rectTop     = (appletHeight - rectHeight)/2;
    int rectLeft    = (appletWidth - rectWidth)/2;

    for (int i=0; i < 1000; i++) {
      g.fillRect(rectLeft, rectTop, rectWidth-1, rectHeight-1);
      g.clearRect(rectLeft, rectTop, rectWidth-1, rectHeight-1);
    }    }  }
```

OVAL methods:

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

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

Java also has methods to draw outlined and filled arcs. They're similar to drawOval() and fillOval() but you must also specify a starting and ending angle for the arc. Angles are given in degrees. The signatures are:

public void drawArc(int left, int top, int width, int height, int startangle, int stopangle)

public void fillArc(int left, int top, int width, int height, int startangle, int stopangle)

Polygons:

Polygons are defined by their corners. No assumptions are made about them except that they lie in a 2-D plane. The basic constructor for the Polygon class is

public Polygon(int[] xpoints, int[] ypoints, int npoints)

```
int[] xpoints = {0, 3, 0};
int[] ypoints = {0, 0, 4};
Polygon myTriangle = new Polygon(xpoints, ypoints, 3);
```

Loading Images

Images in Java are bitmapped GIF or JPEG files that can contain pictures of just about anything. You can use any program at all to create them as long as that program can save in GIF or JPEG format. If you know the exact URL for the image you wish to load, you can load it with the getImage() method:

```
URL imageURL = new URL("http://www.prenhall.com/logo.gif");
java.awt.Image img = this.getImage(imageURL);
```

Drawing Images at Actual Size

Once the image is loaded draw it in the paint() method using the drawImage() method like this

```
g.drawImage(img, x, y, io)
```

img is a member of the Image class which you should have already loaded in your init() method.
x is the x coordinate of the upper left hand corner of the image.
y is the y coordinate of the upper left hand corner of the image.
io is a member of a class which implements the ImageObserver interface.

The ImageObserver interface is how Java handles the asynchronous updating of an Image when it's loaded from a remote web site rather than directly from the hard drive. java.applet.Applet implements ImageObserver so for now just pass the keyword this to drawImage() to indicate that the current applet is the ImageObserver that should be used.

A paint() method that does nothing more than draw an Image starting at the upper left hand corner of the applet may look like this

```
public void paint(Graphics g) {
  g.drawImage(img, 0, 0, this);
}
```

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

public class MagnifyImage extends Applet {

  private Image image;
  private int scaleFactor;

  public void init() {
    String filename  = this.getParameter("imagefile");
    this.image       = this.getImage(this.getDocumentBase(), filename);
    this.scaleFactor = Integer.parseInt(this.getParameter("scalefactor"));
  }

  public void paint (Graphics g) {
    int  width  =  this.image.getWidth(this);
    int  height  =  this.image.getHeight(this);
    scaledWidth = width * this.scaleFactor;
    scaledHeight = height * this.scaleFactor;
    g.drawImage(this.image, 0, 0, scaledWidth, scaledHeight, this);
  }   }
```

Color: color is a part of the Graphics object that does the drawing.

```
g.setColor(Color.pink);
g.drawString("This String is pink!", 50, 25);
g.setColor(Color.green);
g.drawString("This String is green!", 50, 50);
```

Components

Components are graphical user interface (GUI) widgets like checkboxes, menus, windows, buttons, text fields, applets, and more. In Java all components are subclasses of java.awt.Component. Subclasses of Component include

- Canvas
- TextField
- TextArea
- Label
- List
- Button
- Choice
- Checkbox
- Frame
- JButton
- JLabel
- JComboBox
- JMenu

Three Steps to Adding a Component

```
public void init() {
  Label l;
  l = new Label("Hello Container");
  this.add(l);
 }
```

The key thing to remember about adding components to the applet is the three steps:

1. Declare the component
2. Initialize the component
3. Add the component to the layout.

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

public class HelloContainer extends Applet {
  public void init() {
```

```
    Label l;
    l = new Label("Hello Container");
    this.add(l);
  }
}
```

Here's a very simple applet with a Button:

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

public class FirstButton extends Applet {

  public void init () {
      this.add(new Button("My First Button"));
  }     }
```

Button Actions

Unlike labels, buttons do things when you press them. When the mouse is clicked on a Button, the Button fires an ActionEvent. To be ready to respond to this event you must register an *ActionListener* with the Button. For example,

Button beep = new Button("Beep");

add(beep); // add the button to the layout

beep.addActionListener(myActionListener); // assign the button a listener

Here myActionListener is a reference to an object which implements the java.awt.event.ActionListener interface. This interface specifies a single method, actionPerformed():

```
public abstract void actionPerformed(ActionEvent e)
```

The ActionListener object does something as a result of the ActionEvent the button press fired. For example, the following class beeps when it gets an ActionEvent:

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

public class BeepAction implements ActionListener {
```

```
  public void actionPerformed(ActionEvent e) {
    Toolkit.getDefaultToolkit().beep();
  } }
```

java.awt.Canvas

The java.awt.Canvas class is a rectangular area on which you can draw using the methods of java.awt.Graphics. The Canvas class has only three methods:

public Canvas()

public void addNotify()

public void paint(Graphics g)

You generally won't instantiate a canvas directly. Instead you'll subclass it and override the paint() method in your subclass to draw the picture you want.

For example the following Canvas draws a big red oval you can add to your applet.

```
import java.awt.*;
public class RedOval extends Canvas {
public void paint(Graphics g) {
   Dimension d = this.getSize();
   g.setColor(Color.red);
   g.fillOval(0, 0, d.width, d.height);  }

  public Dimension getMinimumSize() {    return new Dimension(50, 100);    }

  public Dimension getPreferredSize() {    return new Dimension(150, 300);    }

  public Dimension getMaximumSize() {    return new Dimension(200, 400);    } }
```

Any applet that uses components should not also override paint(). Doing so will have unexpected effects because of the way Java arranges components. Instead, create a Canvas object and do your drawing in its paint() method.

Java Servlets

- Servlets are used primarily with web servers, where they provide a Java-based replacement for CGI scripts. They can be used to provide dynamic web content like CGI scripts.
- Advantages of servlets over CGI scripts:
    1. Servlets are persistent between invocations, which dramatically improves performance relative to CGI programs.
    2. Servlets are portable among operating systems and among servers.
    3. Servlets have access to all the APIs of the Java platform (e.g. a servlet can interact with a database using JDBC API).

Servlets are a natural fit if you are using the web for enterprise computing. Web browsers then function as universally available thin clients; the web server becomes middleware responsible for running applications for these clients.

Thus the user makes a request of the web server, the server invokes a servlet designed to handle the request, and the result is returned to the user in the web browser. The servlet can use JNDI, Java IDL, JDBC, and other enterprise APIs to perform whatever task is necessary to fulfill the request.

Servlets can be used when collaboration is needed between people. A servlet can handle multiple requests concurrently, and can synchronize requests. So servlets can support on-line conferencing. Servlets can forward requests to other servers and servlets. Thus, servlets can be used to balance load among several servers that mirror the same content, and to partition a single logical service over several servers, according to task type or organizational boundaries.

The Servlet Life Cycle

When a client (web browser) makes a request involving a servlet, the web server loads and executes the appropriate Java classes. Those classes generate content (e.g. HTML), and the server sends the contents back to the client. From the web browser's perspective, this isn't any different from requesting a page generated by a CGI script, or standard HTML. On the server side there is one important difference is ***persistence.*** Instead of shutting down at the end of each request, the servlet remains loaded, ready to handle the subsequent requests. Each request is

handled by a separate thread. These threads share code and data (instance vars). So try to avoid using instance vars, else be sure to use them in synchronized blocks.

The request processing time for a servlet can vary, but is typically quite fast when compared to a similar CGI program. The advantage in the servlet is that you incur the most of the startup overhead only once.

When a servlet loads, its *init()* method is called. You can use *init()* to create I/O intensive resources, such as database connections, for use across multiple invocations. If you have a high-traffic site, the performance benefits can be quite dramatic.

The servlet's *destroy()* method can clean up resources when the server shuts down.

Since a servlet remains active, it can perform other tasks when it is not servicing client request, such as running a background thread (where clients connect to the servlet to view the result) or even acting as an RMI host, enabling a single servlet to handle connections from multiple types of clients.

Servlet Basics

The Servlet API consists of two packages, *javax.servlet*, and *javax.servlet.http*.

The javax is there because servlets are a standard extension to Java, rather than a mandatory part of the API. Thus JVM developers are not required to include classes for them in their Java development and execution environments. Sun has kept the distribution of the servlets API separate from the Java 2 platform because the Servlet API is evolving much faster than the core Java SDK. You can find the Java Servlet Development Kit (JSDK) at http://java.sun.com/products/servlet/. The JSDK includes the necessary servlet classes and a small servlet runner application for development and testing.

HTTP Servlets

The *HttpServlet* class is an extension of *GenericServlet* class that includes methods for handling HTTP specific data. *HttpServlet* defines a number of methods, such as *doGet(),* and *doPost(),* to handle particular types of HTTP requests (GET, POST, etc.). These methods are called by the default implementation of the *service()* method, which figures out the kind of request being made and then invokes the appropriate method. (*service()* is defined in *GenericServlet* class).

Example 1:

HelloServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.print("<html><body>");
        out.print("<h3>Hello Servlet</h3>");
        out.print("</body></html>");
   }
}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>

   <servlet>

         <servlet-name>HelloServlet</servlet-name>
         <servlet-class>HelloServlet</servlet-class>

   </servlet>

   <servlet-mapping>

         <servlet-name>HelloServlet</servlet-name>
         <url-pattern>/HelloServlet</url-pattern>

   </servlet-mapping>

</web-app>
```

The *doGet()* method is called whenever anyone requests a URL that points to this servlet. The servlet is installed in the *servlets* directory and its URL is http://localhost:8080/FirstServlet/HelloServlet. The *doGet()* method is actually called by the default *service()* method of *HttpServlet*. The *service()* method is called by the web server when a request is made of *HelloServlet*; the method determines what kind of HTTP request is being made and dispatches the request to the appropriate *doXXX()* method (in this case, *doGet()*). *doGet()* is passed two objects, *HttpServletRequest* ,and *HttpServletResponse*, that contain information about the request and provide a mechanism for the servlet to provide a response, respectively.

Example 2:

This example deals with forms and dynamic HTML.

The HTML form that calls the servlet using a GET request is as follows:

MyHtml.html

```
<!DOCTYPE html>
<html>
        <body>
                <form action="HelloUserServlet">

                        <label for="user">Name:</label>
                        <input type="text" id="user" name="user">

                        <input type="submit" value="go">
                </form>
        </body>
</html>
```

The form submits a variable named username to the URL /SecondServlet/HelloUserServlet.

The servletis as follows:

HelloUserServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class HelloUserServlet extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
                throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String name=request.getParameter("user");
        out.print("Hello "+name);

        out.close();
    }
}
```

The getParameter() method of HttpServletRequest is used to retrieve the value of the form variable. When a server calls a servlet, it can also pass a set of request parameters.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app >

	<servlet>
		<servlet-name>HelloUserServlet</servlet-name>
		<servlet-class>HelloUserServlet</servlet-class>
	</servlet>

	<servlet-mapping>
		<servlet-name>HelloUserServlet</servlet-name>
		<url-pattern>/HelloUserServlet</url-pattern>
	</servlet-mapping>

	<welcome-file-list>
		<welcome-file>MyHtml.html</welcome-file>
	</welcome-file-list>

</web-app>
```

The POST request

The POST request is designed for posting information to the server, although in practice it is also used for long parameterized requests and larger forms, to get around limitations on the length of URLs. The *doPost()* method is the corresponding method for POST requests. If your servlet is performing database updates, charging a credit card, or doing anything that takes an explicit client action, you should make sure that this activity is happening in a *doPost()* method.

This is because POST requests are not idempotent, which means that they are not safely repeatable, and web browsers treat them specially. E.g. a browser cannot bookmark them. GET requests are idempotent, so they can safely be bookmarked, and a browser is free to issue the request repeatedly (say to get some information from the web server) without necessarily consulting the user. Hence it is not appropriate to charge a credit card in a GET method!

To create a servlet that can handle POST requests, you need to override the default doPost() method from HttpServlet and implement the necessary functionality.

Servlet Responses

In the case of an Http servlet, the response can include three components: a status code, any number of HTTP headers, and a response body.

You use setContentType() method of the response object passed into the servlet to set the type of the response. Examples include “text/html” for text, “image/gif” for returning a GIF file from the database, and “application/pdf” for Adobe Acrobat files.

ServletResponse and *HttpServletResponse* each define two methods for producing output streams, *getOutputStream()* and *getWriter().* The former returns a *ServletOutputStream* that can be used for text or binary data. The latter returns a *java.io.PrintWriter* object used for text data.

You can use *setStatus()* or *sendError()* method to specify the status code sent back to the server. Examples include, 200 (“OK”), 404 (“Not Found”) etc. The *sendRedirect()* method allows you to issue a page redirect. Calling this sets the Location header to the specified location and uses the appropriate status code for a redirect.

Servlet Requests

When a servlet is asked to handle a request, it typically needs specific information about the request so that it can process the request appropriately. For example, a servlet may need to find out about the actual user who is accessing the servlet, for authentication purposes.

ServletRequest and *ServletRequest* provide these methods. Examples include *getProtocol()* (protocol used by request), *getRemoteHost()* (client host name), *getServerName()* (name of the web server), *getServerPort()* (port number the web server listens at), *getParameter()* (access to request parameters as form variables), and *getParameterValues()* (returns an array of strings that contains all the values for a particular parameter).

Error Handling

If the error is part of a servlet's normal operation, such as when a user forgets to fill in a required form field, then write an error message to the servlet's output stream.

If the error is a standard HTTP error, use the *sendError()* method of *HttpServletResponse* to tell the server to send a standard error status code.

E.g. if a file was not found,

resp.sendError(HttpServletResponse.SC_NOT_FOUND);

Example 3:

This servlet serves HTML files.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class FileServlet extends HttpServlet {
 public void doGet(HttpServletRequest req, HttpServletResponse resp) throws
      ServletException, IOException {

    File r;
    FileReader fr;
    BufferedReader br;

      try {

          r = new File(req.getParameter("filename");

          fr = new FileReader(r);
          br = new BufferedReader(fr);

          if (!r.isFile()) {
```

```
                resp.sendError(resp.SC_NOT_FOUND);
                return;
            }
        }
        catch (FileNotFoundException e) {
            resp.sendError(resp.SC_NOT_FOUND);
            return;
        }
                catch (SecurityException se) {    //Be unavailable permanently
                throw(new UnavailableException(this, "Servlet lacks appropriate
privileges"));
        }

        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        String text;
        while ((text = br.readLine()) != null)
            out.println(text);
        br.close();
    }
  }
```

Servlet Initialization: When a server loads a servlet for the first time, it calls the servlet's *init()* method. In its default implementation, *init()* handles some basic housekeeping, but a servlet can override the method to perform other tasks. This includes performing I/O intensive tasks such as opening a database connection. You can also create threads in *init()* to perform other tasks such as pinging other machines on the network to monitor the status of these machines. When an actual request occurs, the service methods can use the resources created in *init()*. The default *init()* is not a do-nothing method. You must call always call *super.init()* as the first action in your own init() routines.

The server passes the *init()* method a *ServletConfig* object. This object encapsulates the servlet initialization parameters, which are accessed via the *getInitParameter()* and

getInitParameterNames() methods. *GenericServlet* and *HttpServlet* both implement the *ServletConfig* interface, so these methods are always available in a servlet. Consult your server documentation on how to set the initialization parameters.

Each servlet also has a *destroy()* method that can be overwritten. This method is called when a server wants to unload a servlet. You can use this method to free important resources.

Server-side includes can be a powerful tool but are not part of the standard Servlet API, and therefore some servlet implementations may not support them. JavaServer Pages (JSP) is another technology for accessing server-side Java components directly in HTML pages. This is similar to Active Server Pages.

The Session Tracking API

Using sessions in servlets is straightforward and involves looking up the session object associated with the current request, creating a new session object when necessary, looking up information associated with a session, storing information in a session, and discarding completed or abandoned sessions. Finally, if you return any URLs to the clients that reference your site and URL-rewriting is being used, you need to attach the session information to the URLs.

Looking Up the HttpSession Object Associated with the Current Request

You look up the HttpSession object by calling the getSession method of HttpServletRequest. Behind the scenes, the system extracts a user ID from a cookie or attached URL data, then uses that as a key into a table of previously created HttpSession objects. But this is all done transparently to the programmer: you just call getSession. If getSession returns null, this means that the user is not already participating in a session, so you can create a new session. Creating a new session in this case is so commonly done that there is an option to automatically create a new session if one doesn't already exist. Just pass true to getSession. Thus, your first step usually looks like this:

HttpSession session = request.getSession(true);

Looking Up Information Associated with a Session

HttpSession objects live on the server; they're just automatically associated with the client by a behind-the-scenes mechanism like cookies or URL-rewriting. These session objects have a built-in data structure that lets you store any number of keys and associated values. In version 2.1 and earlier of the servlet API, you use session.getValue("attribute") to look up a previously stored value. The return type is Object, so you have to do a typecast to whatever more specific type of data was associated with that attribute name in the session. The return value is null if there is no such attribute, so you need to check for null before calling methods on objects associated with sessions. In version 2.2 of the servlet API, getValue is deprecated in favor of get-Attribute because of the better naming match with setAttribute (in version 2.1 the match for getValue is putValue, not setValue). Nevertheless, since not all commercial servlet engines yet support version 2.2, I'll use getValue in my examples. Here's a representative example, assuming ShoppingCart is some class you've defined to store information on items being purchased

```
HttpSession session = request.getSession(true);
ShoppingCart cart = (ShoppingCart)session.getValue("shoppingCart");
if (cart == null) {
  // No cart already in session
  cart = new ShoppingCart();
  session.putValue("shoppingCart", cart);
}
doSomethingWith(cart);
```

In most cases, you have a specific attribute name in mind and want to find the value (if any) already associated with that name. However, you can also discover all the attribute names in a given session by calling getValueNames, which returns an array of strings. This method is your only option for finding attribute names in version 2.1, but in servlet engines supporting version 2.2 of the servlet specification, you can use getAttributeNames. That method is more consistent in that it returns an Enumeration, just like the getHeader- Names and getParameterNames methods of HttpServletRequest. Although the data that was explicitly associated with a session is the part you care most about, there are some other pieces of information that are sometimes useful as well. Here is a summary of the methods available in the HttpSession class.

public Object getValue(String name)

public Object getAttribute(String name)

These methods extract a previously stored value from a session object. They return null if there is no value associated with the given name. Use getValue in version 2.1 of the servlet API. Version 2.2 supports both methods, but getAttribute is preferred and getValue is deprecated.

public void putValue(String name, Object value)
public void setAttribute(String name, Object value)

These methods associate a value with a name. Use putValue with version 2.1 servlets and either setAttribute (preferred) or putValue (deprecated) with version 2.2 servlets. If the object supplied to putValue or setAttribute implements the HttpSessionBindingListener interface, the object's valueBound method is called after it is stored in the session. Similarly, if the previous value implements HttpSessionBindingListener, its valueUnbound method is called.

public void removeValue(String name)
public void removeAttribute(String name)

These methods remove any values associated with the designated name.If the value being removed implements HttpSessionBindingListener, its valueUnbound method is called. With version 2.1 servlets,use removeValue. In version 2.2, removeAttribute is preferred, butremoveValue is still supported (albeit deprecated) for backward compatibility.

public String[] getValueNames()
public Enumeration getAttributeNames()

These methods return the names of all attributes in the session. Use getValueNames in version 2.1 of the servlet specification. In version 2.2, getValueNames is supported but deprecated; use getAttributeNames instead.

public String getId()

This method returns the unique identifier generated for each session. It is sometimes used as the key name when only a single value is associated with a session, or when information about sessions is being logged.

public boolean isNew()

This method returns true if the client (browser) has never seen the session, usually because it was just created rather than being referenced by an incoming client request. It returns false for preexisting sessions.

public long getCreationTime()

This method returns the time in milliseconds since midnight, January 1, 1970 (GMT) at which the session was first built. To get a value useful for printing out, pass the value to the Date constructor or the setTimeInMillis method of GregorianCalendar.

public long getLastAccessedTime()

This method returns the time in milliseconds since midnight, January 1, 1970 (GMT) at which the session was last sent from the client.

Associating Information with a Session

```
HttpSession session = request.getSession(true);
session.putValue("referringPage", request.getHeader("Referer"));
ShoppingCart cart = (ShoppingCart)session.getValue("previousItems");
if (cart == null) {
   // No cart already in session
   cart = new ShoppingCart();
   session.putValue("previousItems", cart);
}
String itemID = request.getParameter("itemID");
if (itemID != null) {
   cart.addItem(Catalog.getItem(itemID));
}
```

Terminating Sessions

Sessions will automatically become inactive when the amount of time between client accesses exceeds the interval specified by getMaxInactiveInterval. When this happens, any objects bound to the HttpSession object automatically get unbound. When this happens, your attached objects will automatically be notified if they implement the HttpSessionBindingListener interface. Rather than waiting for sessions to time out, you can explicitly deactivate a session through the use of the session's invalidate method.

A Servlet Showing Per-Client Access Counts

The following program presents a simple servlet that shows basic information about the client's session. When the client connects, the servlet uses request.getSession(true) to either retrieve the existing session or, if there was no session, to create a new one. The servlet then

looks for an attribute of type Integer called accessCount. If it cannot find such an attribute, it uses 0 as the number of previous accesses. This value is then incremented and associated with the session by putValue. Finally, the servlet prints a small HTML table showing information about the session.

Example:

MySession.java

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class MySession extends HttpServlet {

        public void doGet(HttpServletRequest request, HttpServletResponse response)
                                                                        throws
ServletException, IOException {

                // Create a session object if it is already not created
                HttpSession session = request.getSession(true);

                // Get session creation time.
                Date createTime = new Date(session.getCreationTime());

                // Get last access time of this web page.
                Date lastAccessTime = new Date(session.getLastAccessedTime());

                Integer visitCount = new Integer(0);
                String visitCountKey = new String("visitCount");

                String userID = new String("MNC");
                String userIDKey = new String("userID");


                // Check if this is new comer on your web page.
                if (session.isNew()) {
                        session.setAttribute(userIDKey, userID);
                }
                else {
                        visitCount = (Integer)session.getAttribute(visitCountKey);
                        visitCount = visitCount + 1;
                }
                session.setAttribute(visitCountKey, visitCount);
```

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();

out.println(
        "<!doctype html>" + "<html>"
        + "<body>"
        + "<h2 align = \"center\">My Session Information</h2>"
        + "<table border = \"1\" align = \"center\">"
        +

        "<tr>"
        + " <th>Session Info</th>"
        + "<th>Value</th>"
        + "</tr>"
        +

        "<tr>"
        + " <td>Id</td>"
        + " <td>" + session.getId() + "</td>"
        + "</tr>"
        +

        "<tr>"
        + " <td>Creation Time</td>"
        + " <td>" + createTime + " </td>"
        + "</tr>"
        +

        "<tr>"
        + " <td>Time of Last Access</td>"
        + " <td>" + lastAccessTime + "</td>"
        + "</tr>"
        +

        "<tr>"
        + " <td>User ID</td>"
        + " <td>" + userID + "</td>"
        + "</tr>"
        +

        "<tr>"
        + " <td>Number of Visits</td>"
        + " <td>" + visitCount + "</td>"
        + "</tr>"
        + "</table>"
        + "</body>"
```

```
                + "</html>");
        }
}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>

<servlet>
  <servlet-name>MySession</servlet-name>
  <servlet-class>MySession</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>MySession</servlet-name>
  <url-pattern>/MySession</url-pattern>
</servlet-mapping>

</web-app>
```

The Servlet Cookie API

To send cookies to the client, a servlet should create one or more cookies with designated names and values with new Cookie(name, value), set any optional attributes with cookie.set*Xxx* (readable later by cookie.get*Xxx*), and insert the cookies into the response headers with response.addCookie(cookie). To read incoming cookies, a servlet should call request.getCookies, which returns an array of Cookie objects corresponding to the cookies the browser has associated with your site (this is null if there are no cookies in the request). In most cases, the servlet loops down this array until it finds the one whose name (getName) matches the name it had in mind, then calls getValue on that Cookie to see the value associated with that name. Each of these topics is discussed in more detail in the following sections.

Creating Cookies

You create a cookie by calling the Cookie constructor, which takes two strings: the cookie name and the cookie value. Neither the name nor the value should contain white space or any of the following characters: [ ] ( ) = , " / ? @ : ;

Cookie Attributes

Before adding the cookie to the outgoing headers, you can set various characteristics of the cookie by using one of the following set*Xxx* methods, where *Xxx* is the name of the attribute you want to specify. Each set*Xxx* method has a corresponding get*Xxx* method to retrieve the attribute value. Except for name and value, the cookie attributes apply only to *outgoing* cookies from the server to the client; they aren't set on cookies that come *from* the browser to the server. See Appendix A (Servlet and JSP Quick Reference) for a summarized version of this information.

public String getComment()
public void setComment(String comment)

These methods look up or specify a comment associated with the cookie. With version 0 cookies (see the upcoming subsection on getVersion and setVersion), the comment is used purely for informational purposes on the server; it is not sent to the client.

public String getDomain()
public void setDomain(String domainPattern)

These methods get or set the domain to which the cookie applies. Normally, the browser only returns cookies to the exact same hostname that sent them. You can use setDomain method to instruct the browser to return them to other hosts within the same domain. To prevent servers setting cookies that apply to hosts outside their domain, the domain specified is required to start with a dot (e.g.,.prenhall.com), and must contain two dots for noncountry domains like .com, .edu and .gov; and three dots for country domains like .co.uk and .edu.es. For instance, cookies sent from a servlet at bali.vacations.com would not normally get sent by the browser to pages at mexico.vacations.com. If the site wanted this to happen, the servlets could specify

cookie.setDomain(".vacations.com").

public int getMaxAge()

public void setMaxAge(int lifetime)

These methods tell how much time (in seconds) should elapse before the cookie expires. A negative value, which is the default, indicates that the cookie will last only for the current session (i.e., until the user quits the browser) and will not be stored on disk. See the LongLivedCookie class (Listing 8.4), which defines a subclass of Cookie with a maximum age automatically set one year in the future. Specifying a value of 0instructs the browser to delete the cookie.

public String getName()

public void setName(String cookieName)

This pair of methods gets or sets the name of the cookie. The name and the value are the two pieces you virtually *always* care about. However, since the name is supplied to the Cookie constructor, you rarely need to call setName. On the other hand, getName is used on almost every cookie received on the server. Since the getCookies method of HttpServletRequest returns an array of Cookie objects, it is common to loop down this array, calling getName until you have a particular name, then check the value with getValue.

public String getPath()

public void setPath(String path)

These methods get or set the path to which the cookie applies. If you don't specify a path, the browser returns the cookie only to URLs in or below the directory containing the page that sent the cookie. The setPath method can be used to specify something more general. For example, someCookie.setPath("/") specifies that *all* pages on the server should receive the cookie. The path specified must include the current page; that is, you may specify a more general path than the default, but not a more specific one.

public boolean getSecure()

public void setSecure(boolean secureFlag)

This pair of methods gets or sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e., SSL) connections. The default is false; the cookie should apply to all connections.

public String getValue()

public void setValue(String cookieValue)

The getValue method looks up the value associated with the cookie; the setValue method

specifies it. Again, the name and the value are the two parts of a cookie that you almost *always* care about, although in a few cases, a name is used as a boolean flag and its value is ignored (i.e., the existence of a cookie with the designated name is all that matters).

public int getVersion()

public void setVersion(int version)

These methods get/set the cookie protocol version the cookie complies with. Version 0, the default,

Placing Cookies in the Response Headers

The cookie is inserted into a Set-Cookie HTTP response header by means of the addCookie method of HttpServletResponse. The method is called addCookie, not setCookie, because any previously specified SetCookie headers are left alone and a new header is set. Here's an example:

```
Cookie userCookie = new Cookie("user", "uid1234");
userCookie.setMaxAge(60*60*24*365); // 1 year
response.addCookie(userCookie);
```

Reading Cookies from the Client

To send cookies *to* the client, you create a Cookie, then use addCookie to send a Set-Cookie HTTP response header. To read the cookies that come back *from* the client, you call getCookies on the HttpServletRequest. This call returns an array of Cookie objects corresponding to the values that came in on the Cookie HTTP request header. If there are no cookies in the request, getCookies returns null. Once you have this array, you typically loop down it, calling getName on each Cookie until you find one matching the name you have in mind. You then call getValue on the matching Cookie and finish with some processing specific to the resultant value. This is such a common process that Section 8.5 presents two utilities that simplify retrieving a cookie or cookie value that matches a designated cookie name.

Examples of Setting and Reading Cookies

The following program shows the SetCookies servlet, a servlet that sets six cookies. Three have the default expiration date, meaning that they should apply only until the user next restarts the browser. The other three use setMaxAge to stipulate that they should apply for the next hour, regardless of whether the user restarts the browser or reboots the computer to initiate a new browsing session.

Example:

CookieServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class CookieServlet extends HttpServlet {
  public void doPost(HttpServletRequest request, HttpServletResponse response)
                                                        throws ServletException, IOException
{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String name=request.getParameter("user");
        out.print("Hello "+ name);
        Cookie ck=new Cookie("uname",name);          //creating cookie object
        response.addCookie(ck);                      //adding cookie in the response
        out.close();
    }
}
```

Java Server Pages

JavaServer Pages (JSP) technology enables you to mix regular, static HTML with dynamically generated content from servlets. You simply write the regular HTML in the normal manner, using familiar Web-page-building tools. You then enclose the code for the dynamic parts in special tags, most of which start with <% and end with %>.

Thanks for ordering <I><%= request.getParameter("title") %></I>

Separating the static HTML from the dynamic content provides a number of benefits over servlets alone, and the approach used in JavaServer Pages offers several advantages over competing technologies such as ASP, PHP, or ColdFusion. Section 1.4 (The Advantages of JSP) gives some details on these advantages, but they basically boil down to two facts: that JSP is widely supported and thus doesn't lock you into a particular operating system or Web server and that JSP gives you full access to servlet and Java technology for the dynamic part, rather than requiring you to use an unfamiliar and weaker special-purpose language. The process of making JavaServer Pages accessible on the Web is much simpler than that for servlets. Assuming you have a Web server that supports JSP, you give your file a .jsp extension and simply install it in any place you could put a normal Web page: no compiling, no packages, and no user CLASSPATH settings. However, although your personal *environment* doesn't need any special settings, the *server* still has to be set up with access to the servlet and JSP class files and the Java compiler.

Aside from the regular HTML, there are three main types of JSP constructs that you embed in a page: *scripting elements*, *directives*, and *actions*.

Scripting elements let you specify Java code that will become part of the resultant servlet,

directives let you control the overall structure of the servlet, and

actions let you specify existing components that should be used and otherwise control the

behavior of the JSP engine. To simplify the scripting elements, you have access to a number of predefined variables, such as request in the code snippet just shown. Scripting elements are covered in this chapter, and directives and actions are explained in the following chapters.

Scripting Elements

JSP scripting elements let you insert code into the servlet that will be generated from the JSP page. There are three forms:

1. *Expressions* of the form <%= expression %>, which are evaluated and inserted into the servlet's output
2. *Scriptlets* of the form <% code %>, which are inserted into the servlet's _jspService method (called by service)
3. *Declarations* of the form <%! code %>, which are inserted into the body of the servlet class, outside of any existing methods

JSP Expressions

A JSP expression is used to insert values directly into the output. It has the following form:

<%= *Java Expression* %>

The expression is evaluated, converted to a string, and inserted in the page. This evaluation is performed at run time (when the page is requested) and thus has full access to information about the request. For example, the following shows the date/time that the page was requested:

Current time: <%= new java.util.Date() %>

JSP Scriptlets

If you want to do something more complex than insert a simple expression, JSP scriptlets let you insert arbitrary code into the servlet's _jspService method (which is called by service). Scriptlets have the following form:

<% *Java Code* %>

Scriptlets have access to the same automatically defined variables as expressions (request, response, session, out, etc). So, for example, if you want output to appear in the resultant page, you would use the out variable, as in the following example.

<%

```
String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData);
%>
```

In this particular instance, you could have accomplished the same effect more easily by using the following JSP expression:

```
Attached GET data: <%= request.getQueryString() %>
```

In general,however, scriptlets can perform a number of tasks that cannot be accomplished with expressions alone. These tasks include setting response headers and status codes, invoking side effects such as writing to the server log or updating a database, or executing code that contains loops, conditionals, or other complex constructs. For instance, the following snippet specifies that the current page is sent to the client as plain text, not as HTML (which is the default).

```
<% response.setContentType("text/plain"); %>
```

As an example of executing code that is too complex for a JSP expression the listing presents a JSP page that uses the bgColor request parameter to set the background color of the page. Some results are shown in Figures

The import Attribute

The import attribute of the page directive lets you specify the packages that should be imported by the servlet into which the JSP page gets translated. If you don't explicitly specify any classes to import, the servlet imports java.lang.*, javax.servlet.*, javax.servlet.jsp.*, javax.servlet.
http.*, and possibly some number of server-specific entries. Never write JSP code that relies on any server-specific classes being imported automatically. Use of the import attribute takes one of the following two forms:

```
<%@ page import="package.class" %>
<%@ page import="package.class1,...,package.classN" %>
```

For example, the following directive signifies that all classes in the java.util package should be available to use without explicit package identifiers.

<%@ page import="java.util.*" %>

The import attribute is the only page attribute that is allowed to appear multiple times within the same document. Although page directives can appear anywhere within the document, it is traditional to place import statements either near the top of the document or just before the first place that the referenced package is used.

The contentType Attribute

The contentType attribute sets the Content-Type response header, indicating the MIME type of the document being sent to the client. For more information on MIME types.

Use of the contentType attribute takes one of the following two forms:

<%@ page contentType="MIME-Type" %>

<%@ page contentType="MIME-Type; charset=Character-Set" %>

For example, the directive

<%@ page contentType="text/plain" %>

has the same effect as the scriptlet

<% response.setContentType("text/plain"); %>

Unlike regular servlets, where the default MIME type is text/plain, the default for JSP pages is text/html (with a default character set of ISO-8859-1).

Example:

MyJSP.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
	<body>
		<%
			String name=request.getParameter("user");
			out.print("Hello "+name);
		%>
	</body>
</html>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app >

	<welcome-file-list>
		<welcome-file>MyHtml.html</welcome-file>
	</welcome-file-list>

</web-app>
```

MyHtml.html

```
<!DOCTYPE html>
<html>
	<body>
		<form action="MyJSP.jsp">
			<label for="user">Name:</label>
			<input type="text" id="user" name="user">

			<input type="submit" value="go">
		</form>
	</body>
</html>
```