Fundamentals of Java™ Servlets: The Java Servlet API

MageLang Institute

A servlet is a Java™ component that can be plugged into a Java-enabled web server to provide custom services. These services can include:

- New features
- Runtime changes to content
- Runtime changes to presentation
- New standard protocols (such as FTP)
- New custom protocols

Servlets are designed to work within a request/response processing model. In a request/response model, a client sends a request message to a server and the server responds by sending back a reply message. Requests can come in the form of an

```
HTTP
URL,
FTP,
URL,
```

or a custom protocol.

The request and the corresponding response reflect the state of the client and the server at the time of the request. Normally, the state of the client/server connection cannot be maintained across different request/response pairs. However, session information is maintainable with servlets through means to be described later.

The Java Servlet API includes several Java interfaces and fully defines the link between a hosting server and servlets. The Servlet API is defined as an extension to the standard JDK. JDK extensions are packaged under `javax`--the root of the Java extension library tree. The Java Servlet API contains the following packages:

- Package `javax.servlet`
- Package `javax.servlet.http`

Servlets are a powerful addition to the Java environment. They are fast, safe, reliable, and 100% pure Java. Because servlets plug into an existing server, they leverage a lot of existing code and technology. The server handles the network connections, protocol negotiation, class loading, and more; all of this work does not need to be replicated! And, because servlets are located at the middle tier, they are positioned to add a lot of value and flexibility to a system.

In this course you will learn about the Servlet API and you will get a brief tour of the types of features servlets can implement.

Architectural Roles for Servlets

Because of their power and flexibility, servlets can play a significant role in a system architecture. They can perform the application processing assigned to the middle tier, act as a proxy for a client, and even augment the features of the middle tier by adding support for new protocols or other features. A middle tier acts as the application server in so called three-tier client/server systems, positioning itself between a lightweight client like a web browser and a data source.

Middle-Tier Process

In many systems a middle tier serves as a link between clients and back-end services. By using a middle tier a lot of processing can be off-loaded from both clients (making them lighter and faster) and servers (allowing them to focus on their mission).

One advantage of middle tier processing is simply connection management. A set of servlets could handle connections with hundreds of clients, if not thousands, while recycling a pool of expensive connections to database servers.

Other middle tier roles include:

- Business rule enforcement
- Transaction management
- Mapping clients to a redundant set of servers
- Supporting different types of clients such as pure HTML and Java capable clients

Proxy Servers

When used to support applets, servlets can act as their proxies. This can be important because Java security allows applets only to make connections back to the server from which they were loaded.

If an applet needs to connect to a database server located on a different machine, a servlet can make this connection on behalf of the applet.

Protocol Support

The Servlet API provides a tight link between a server and servlets. This allows servlets to add new protocol support to a server. (You will see how HTTP support is provided for you in the API packages.) Essentially, any protocol that follows a request/response computing model can be implemented by a servlet. This could include:

- SMTP
- POP
- FTP

Servlet support is currently available in several web servers, and will probably start appearing in other types of application servers in the near future. You will use a web server to host the servlets in this class and only deal with the HTTP protocol.

Because HTTP is one of the most common protocols, and because HTML can provide such a rich presentation of information, servlets probably contribute the most to building HTTP based systems.

HTML Support

HTML can provide a rich presentation of information because of its flexibility and the range of content that it can support. Servlets can play a role in creating HTML content. In fact, servlet support for HTML is so common, the `javax.servlet.http` package is dedicated to supporting HTTP protocol and HTML generation.

Complex web sites often need to provide HTML pages that are tailored for each visitor, or even for each hit. Servlets can be written to process HTML pages and customize them as they are sent to a client. This can be as simple as on the fly substitutions or it can be as complex as compiling a grammar-based description of a page and generating custom HTML.

Inline HTML Generation

Some web servers, such as the Java Web Server™ (JWS), allow servlet tags to be embedded directly into HTML files. When the server encounters such a tag, it calls the servlet while it is sending the

HTML file to the client. This allows a servlet to insert its contribution directly into the outgoing HTML stream.

Server-Side Includes

Another example is on the fly tag processing known as *server-side includes* (SSI). With SSI, an HTML page can contain special commands that are processed each time a page is requested. Usually a web server requires HTML files that incorporate SSI to use a unique extension, such as `.shtml`. As an example, if an HTML page (with an `.shtml` extension) includes the following: `<!--#include virtual="/includes/page.html"-->`

it would be detected by the web server as a request to perform an inline file include. While server side includes are supported by most web servers, the SSI tags are not standardized.

Servlets are a great way to add server side include processing to a web server. With more and more web servers supporting servlets, it would be possible to write a standard SSI processing servlet and use it on different web servers.

Replacing CGI Scripts

An HTTP servlet is a direct replacement for Common Gateway Interface (CGI) scripts. HTTP servlets are accessed by the user entering a URL in a browser or as the target of an HTML form action. For example, if a user enters the following URL into a browser address field, the browser requests a servlet to send an HTML page with the current time:
`http://localhost/servlet/DateTimeServlet` The `DateTimeServlet` responds to this request by sending an HTML page to the browser.

Note that these servlets are not restricted to generating web pages; they can perform any other function, such as storing and fetching database information, or opening a socket to another machine.

Installing Servlets

Servlets are not *run* in the same sense as applets and applications. Servlets provide functionality that extends a server. In order to test a servlet, two steps are required:

1. Install the servlet in a hosting server
2. Request a servlet's service via a client request

There are many web servers that support servlets. It is beyond the scope of this course to cover the different ways to install servlets in each server. This course examines the JSDK's `servletrunner` utility and the JWS.

Temporary versus Permanent Servlets

Servlets can be started and stopped for each client request, or they can be started as the web server is started and kept alive until the server is shut down. Temporary servlets are loaded on demand and offer a good way to conserve resources in the server for less-used functions.

Permanent servlets are loaded when the server is started, and live until the server is shutdown. Servlets are installed as permanent extensions to a server when their start-up costs are very high (such as establishing a connection with a DBMS), when they offer permanent server-side functionality (such as an RMI service), or when they must respond as fast as possible to client requests.

There is no special code necessary to make a servlet temporary or permanent; this is a function of the server configuration.

Because servlets can be loaded when a web server starts, they can use this auto-loading mechanism to provide easier loading of server-side Java programs. These programs can then provide functionality that is totally unique and independent of the web server. For example, a servlet could provide R-based services (`rlogin`, `rsh`, ...) through TCP/IP ports while using the servlet request/response protocol to present and process HTML pages used to manage the servlet.

Using `servletrunner`

For both JDK 1.1 and the Java 2 platform, you need to install the Java Servlet Development Kit (JSDK). To use `servletrunner`, make sure your `PATH` environment variable points to its directory. For the JSDK 2.0 installed with all default options, that location is: `c:\jsdk2.0\bin` on a Windows platform.

To make sure that `servletrunner` has access to the Java servlet packages, check that your `CLASSPATH` environment variable is pointing to the correct JAR file, `c:\jsdk2.0\lib\jsdk.jar` on a Windows platform. With the Java 2 platform, instead of modifying the `CLASSPATH`, it is easier to just copy the JAR file to the `ext` directory under the Java runtine environment. This treats the servlet packages as standard extensions.

With the environment set up, run the `servletrunner` program from the command line. The parameters are:

```
Usage: servletrunner [options]
Options:
  -p port     the port number to listen on
  -b backlog  the listen backlog
  -m max      maximum number of connection handlers
  -t timeout  connection timeout in milliseconds
  -d dir      servlet directory
  -s filename servlet property file name
```

The most common way to run this utility is to move to the directory that contains your servlets and run `servletrunner` from that location. However, that doesn't automatically configure the tool to load the servlets from the current directory.

Magercise

1. Hosting with `servletrunner`

Using Java Web Server

Sun's Java Web Server (JWS) is a full featured product. For servlet developers, a nice feature is its ability to detect when a servlet has been updated. It detects when new class files have been copied to the appropriate servlet directory and, if necessary, automatically reloads any running servlets.

The JWS can be installed as a service under Windows NT. While this makes it convenient for running a production server, it is not recommended for servlet development work. Under Windows 95, there are no OS services, so the command line start-up is your only option.

To run JWS from the `c:\JavaWebServer1.1\bin` directory, type in the `httpd` command. This starts the server in a console window. No further display is shown in the console unless a servlet executes a `System.out.println()` statement.

Servlets are installed by moving them to the `c:\JavaWebServer1.1\servlets` directory. As mentioned, JWS detects when servlets have been added to this directory. Although you can use the JWS management applet to tailor the servlet installation, this is generally not advised except for production server installations.

To shut down the JWS, press <Control>+C in the command window. The server prints a message to the console when it has finished shutting down.

Magercise

2. Hosting with the Java Web Server

Servlet API

The Java Servlet API defines the interface between servlets and servers. This API is packaged as a standard extension to the JDK under `javax`:

- Package `javax.servlet`
- Package `javax.servlet.http`

The API provides support in four categories:

- Servlet life cycle management
- Access to servlet context
- Utility classes
- HTTP-specific support classes

The Servlet Life Cycle

Servlets run on the web server platform as part of the same process as the web server itself. The web server is responsible for initializing, invoking, and destroying each servlet instance.

A web server communicates with a servlet through a simple interface, `javax.servlet.Servlet`. This interface consists of three main methods:

- `init()`
- `service()`
- `destroy()`

and two ancillary methods:

- `getServletConfig()`
- `getServletInfo()`

You may notice a similarity between this interface and that of Java applets. This is by design! Servlets are to web servers what applets are to web browsers.An applet runs in a web *browser*, performing actions it requests through a specific interface. A servlet does the same, running in the web *server*.

The `init()` Method

When a servlet is first loaded, its `init()` method is invoked. This allows the servlet to per form any setup processing such as opening files or establishing connections to their servers. If a servlet has been permanently installed in a server, it loads when the server starts to run. Otherwise, the server activates a servlet when it receives the first client request for the services provided by the servlet.

The `init()` method is guaranteed to finish before any other calls are made to the servlet--such as a call to the `service()` method. Note that `init()` will only be called once; it will not be called again unless the servlet has been unloaded and then reloaded by the server.

The `init()` method takes one argument, a reference to a `ServletConfig` object which provides initialization arguments for the servlet. This object has a method `getServletContext()` that returns a `ServletContext` object containing information about the servlet's environment (see the discussion on Servlet Initialization Context below).

The `service()` Method

The `service()` method is the heart of the servlet. Each request message from a client results in a single call to the servlet's `service()` method. The `service()` method reads the request and produces the response message from its two parameters:

- A `ServletRequest` object with data from the client. The data consists of name/value pairs of parameters and an `InputStream`. Several methods are provided that return the client's parameter information. The `InputStream` from the client can be obtained via the `getInputStream()` method. This method returns a `ServletInputStream`, which can be used to get additional data from the client. If you are interested in processing character-level data instead of byte-level data, you can get a `BufferedReader` instead with `getReader()`.

- A `ServletResponse` represents the servlet's reply back to the client. When preparing a response, the method `setContentType()` is called first to set the MIME type of the reply. Next, the method `getOutputStream()` or `getWriter()` can be used to obtain a `ServletOutputStream` or `PrintWriter`, respectively, to send data back to the client.

As you can see, there are two ways for a client to send information to a servlet. The first is to send parameter values and the second is to send information via the `InputStream` (or `Reader`). Parameter values can be embedded into a URL. How this is done is discussed below. How the parameter values are read by the servlet is discussed later.

The `service()` method's job is conceptually simple--it creates a response for each client request sent to it from the host server.

However, it is important to realize that there can be multiple service requests being processed at once. If your service method requires any outside resources, such as files, databases, or some external data, you must ensure that resource access is thread-safe. Making your servlets thread-safe is discussed in a later section of this course.

The `destroy()` Method

The `destroy()` method is called to allow your servlet to clean up any resources (such as open files or database connections) before the servlet is unloaded. If you do not require any clean-up operations, this can be an empty method.

The server waits to call the `destroy()` method until either all service calls are complete, or a certain amount of time has passed. This means that the `destroy()` method *can* be called while some longer-running `service()` methods are still running. It is important that you write your `destroy()` method to avoid closing any necessary resources until all `service()` calls have completed.

Sample Servlet

The code below implements a simple servlet that returns a static HTML page to a browser. This example fully implements the `Servlet` interface.

```
import java.io.*;
import javax.servlet.*;
public SampleServlet implements Servlet {
  private ServletConfig config;

  public void init (ServletConfig config)
    throws ServletException {
    this.config = config;
  }

  public void destroy() {} // do nothing

  public ServletConfig getServletConfig() {
    return config;
  }

  public String getServletInfo() {
    return "A Simple Servlet";
  }

  public void service (ServletRequest req,
    ServletResponse res
  ) throws ServletException, IOException  {
    res.setContentType( "text/html" );
    PrintWriter out = res.getWriter();
    out.println( "<html>" );
    out.println( "<head> );
```

```
    out.println( "<title>A Sample Servlet</title>" );
    out.println( "</head>" );
    out.println( "<body>" );
    out.println( "<h1>A Sample Servlet</h1>" );
    out.println( "</body>" );
    out.println( "</html>" );
    out.close();
  }
}
```

Servlet Context

A servlet lives and dies within the bounds of the server process. To understand its operating environment, a servlet can get information about its environment at different times. Servlet initialization information is available during servlet start-up; information about the hosting server is available at any time; and each service request can contain specific contextual information.

Servlet Initialization Information

Initialization information is passed to the servlet via the `ServletConfig` parameter of the `init()` method. Each web server provides its own way to pass initialization information to a servlet. With the JWS, if a servlet class `DatePrintServlet` takes an initialization argument `timezone`, you would define the following properties in a `servlets.properties` file:

```
servlet.dateprinter.code=DatePrinterServlet
servlet.dateprinter.timezone=PST
```

or this information could be supplied through a GUI administration tool.

The timezone information would be accessed by the servlet with the following code:

```
String timezone;
public void init(ServletConfig config) {
  timeZone = config.getInitParameter("timezone");
}
```

An `Enumeration` of all initialization parameters is available to the servlet via the `getInitParameterNames()` method.

Server Context Information

Server context information is available at any time through the `ServletContext` object. A servlet can obtain this object by calling the `getServletContext()` method on the `ServletConfig` object. Remember that this was passed to the servlet during the initialization phase. A well written `init()` method saves the reference in a private variable.

The `ServletContext` interface defines several methods. These are outlined below.

`getAttribute()`	An extensible way to get information about a server via attribute name/value pairs. This is server specific.
`getMimeType()`	Returns the MIME type of a given file.
`getRealPath()`	This method translates a relative or virtual path to a new path relative to the server's HTML documentation root location.
`getServerInfo()`	Returns the name and version of the network service under which the servlet is running.
`getServlet()`	Returns a `Servlet` object of a given name. Useful when you want to access the services of other servlets.
`getServletNames()`	Returns an enumeration of servlet names available in the current namespace.
`log()`	Writes information to a servlet log file. The log file name and format are server specific.

The following example code shows how a servlet uses the host server to write a message to a servlet log when it initializes:

```
private ServletConfig config;
public void init(ServletConfig config) {
  // Store config in an instance variable
  this.config = config;
  ServletContext sc = config.getServletContext();
  sc.log( "Started OK!" );
}
```

Servlet Context During a Service Request

Each service request can contain information in the form of name/value parameter pairs, as a `ServletInputStream`, or a `BufferedReader`. This information is available from the `ServletRequest` object that is passed to the `service()` method.

The following code shows how to get service-time information:

```
BufferedReader reader;
```

```
String         param1;
String         param2;
public void service (
    ServletRequest  req,
    ServletResponse res) {

  reader = req.getReader();
  param1 = req.getParameter("First");
  param2 = req.getParameter("Second");
}
```

There are additional pieces of information available to the servlet through `ServletRequest`. These are shown in the following table.

`getAttribute()`	Returns value of a named attribute for this request.
`getContentLength()`	Size of request, if known.
`getContentType()`	Returns MIME type of the request message body.
`getInputStream()`	Returns an `InputStream` for reading binary data from the body of the request message.
`getParameterNames()`	Returns an array of strings with the names of all parameters.
`getParameterValues()`	Returns an array of strings for a specific parameter name.
`getProtocol()`	Returns the protocol and version for the request as a string of the form <protocol>/<major version>.<minor version>.
`getReader()`	Returns a `BufferedReader` to get the text from the body of the request message.
`getRealPath()`	Returns actual path for a specified virtual path.
`getRemoteAddr()`	IP address of the client machine sending this request.
`getRemoteHost()`	Host name of the client machine that sent this request.
`getScheme()`	Returns the scheme used in the URL for this request (for example, https, http, ftp, etc.).
`getServerName()`	Name of the host server that received this request.
`getServerPort()`	Returns the port number used to receive this request.

The following Magercise shows you how to extract parameters from a service request.

Magercise

3. Accessing Servlet Service-Time Parameters

Utility Classes

There are several utilities provided in the Servlet API. The first is the interface `javax.servlet.SingleThreadModel` that can make it easier to write simple servlets. If a servlet implements this marker interface, the hosting server knows that it should never call the servlet's `service()` method while it is processing a request. That is, the server processes all service requests within a single thread.

While this makes it easier to write a servlet, this can impede performance. A full discussion of this issue is located later in this course.

Two exception classes are included in the Servlet API. The exception `javax.servlet.ServletException` can be used when there is a general failure in the servlet. This notifies the hosting server that there is a problem.

The exception `javax.servlet.UnavailableException` indicates that a servlet is unavailable. Servlets can report this exception at any time. There are two types of unavailability:

- **Permanent**. The servlet is unable to function until an administrator takes some action. In this state, a servlet should write a log entry with a problem report and possible resolutions.

- **Temporary**. The servlet encountered a (potentially) temporary problem, such as a full disk, failed server, etc. The problem can correct itself with time or may require operator intervention.

HTTP Support

Servlets that use the HTTP protocol are very common. It should not be a surprise that there is specific help for servlet developers who write them. Support for handling the HTTP protocol is provided in the package `javax.servlet.http`. Before looking at this package, take a look at the HTTP protocol itself.

HTTP stands for the HyperText Transfer Protocol. It defines a protocol used by web browsers and servers to communicate with each other. The protocol defines a set of text-based request messages called *HTTP methods*. (Note: The HTTP specification calls these *HTTP methods*; do not confuse this term with Java methods. Think of *HTTP methods* as messages requesting a certain type of response). The HTTP methods include:

- `GET`
- `HEAD`
- `POST`
- `PUT`
- `DELETE`
- `TRACE`
- `CONNECT`
- `OPTIONS`

For this course, you will only need to look at only three of these methods: `GET`, `HEAD`, and `POST`.

The HTTP `GET` Method

The HTTP `GET` method requests information from a web server. This information could be a file, output from a device on the server, or output from a program (such as a servlet or CGI script).

An HTTP `GET` request takes the form:

```
GET URL <http version>
Host: <target host>
```

in addition to several other lines of information.

For example, the following HTTP `GET` message is requesting the home page from the MageLang web site:

```
GET / HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/4.0 (
 compatible;
 MSIE 4.01;
 Windows NT)
Host: www.magelang.com
Accept: image/gif, image/x-xbitmap,
 image/jpeg, image/pjpeg
```

On most web servers, servlets are accessed via URLs that start with `/servlet/`. The following HTTP `GET` method is requesting the servlet `MyServlet` on the host `www.magelang.com`:

```
GET /servlet/MyServlet?name=Scott&
  company=MageLang%20Institute HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/4.0 (
 compatible;
```

```
 MSIE 4.01;
 Windows NT)
Host: www.magelang.com
Accept: image/gif, image/x-xbitmap,
 image/jpeg, image/pjpeg
```

The URL in this `GET` request invokes the servlet called `MyServlet` and contains two parameters, `name` and `company`. Each parameter is a name/value pair following the format `name=value`. The parameters are specified by following the servlet name with a question mark ('?'), with each parameter separated by an ampersand ('&').

Note the use of `%20` in the company's value. A space would signal the end of the URL in the `GET` request line, so it must be "URL encoded", or replaced with `%20` instead. As you will see later, servlet developers do not need to worry about this encoding as it will be automatically decoded by the `HttpServletRequest` class.

HTTP `GET` requests have an important limitation. Most web servers limit how much data can be passed as part of the URL name (usually a few hundred bytes.) If more data must be passed between the client and the server, the HTTP `POST` method should be used instead.

It is important to note that the server's handling of a `GET` method is expected to be *safe* and *idempotent*. This means that a `GET` method will not cause any side effects and that it can be executed repeatedly.

When a server replies to an HTTP `GET` request, it sends an HTTP response message back. The header of an HTTP response looks like the following:

```
HTTP/1.1 200 Document follows
Date: Tue, 14 Apr 1997 09:25:19 PST
Server: JWS/1.1
Last-modified: Mon, 17 Jun 1996 21:53:08 GMT
Content-type: text/html
Content-length: 4435

<4435 bytes worth of data -- the document body>
```

The `HEAD` Method

The HTTP `HEAD` method is very similar to the HTTP `GET` method. The request looks exactly the same as the `GET` request (except the word `HEAD` is used instead of `GET`), but the server *only returns the header information*.

`HEAD` is often used to check the following:

- The last-modified date of a document on the server for caching purposes

- The size of a document before downloading (so the browser can present progress information)
- The server type, allowing the client to customize requests for that server
- The type of the requested document, so the client can be sure it supports it

Note that `HEAD`, like `GET`, is expected to be *safe* and *idempotent*.

The `POST` Method

An HTTP `POST` request allows a client to send data to the server. This can be used for several purposes, such as

- Posting information to a newsgroup
- Adding entries to a web site's guest book
- Passing more information than a `GET` request allows

Pay special attention to the third bullet above. The HTTP `GET` request passes all its arguments as part of the URL. Many web servers have a limit to how much data they can accept as part of the URL. The `POST` method passes all of its parameter data in an input stream, removing this limit.

A typical `POST` request might be as follows:

```
POST /servlet/MyServlet HTTP/1.1
User-Agent: Mozilla/4.0 (
 compatible;
 MSIE 4.01;
 Windows NT)
Host: www.magelang.com
Accept: image/gif, image/x-xbitmap,
 image/jpeg, image/pjpeg, */
Content-type: application/x-www-form-urlencoded
Content-length: 39

name=Scott&company=MageLang%20Institute
```

Note the blank line--this signals the end of the `POST` request header and the beginning of the extended information.

Unlike the `GET` method, `POST` is *not* expected to be *safe* nor *idempotent*; it can perform modifications to data, and it is not required to be repeatable.

HTTP Support Classes

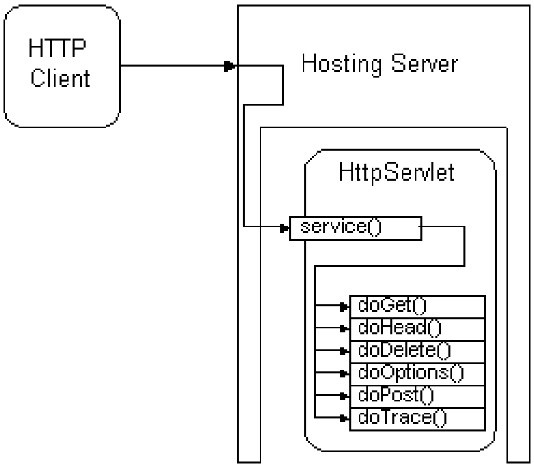
Now that you have been introduced to the HTTP protocol, consider how the `javax.servlet.http` package helps you write HTTP servlets. The abstract class `javax.servlet.http.HttpServlet` provides an implementation of the `javax.servlet.Servlet` interface and includes a lot of helpful default functionality. The easiest way to write an HTTP servlet is to extend `HttpServlet` and add your own custom processing.

The class `HttpServlet` provides an implementation of the `service()` method that dispatches the HTTP messages to one of several special methods. These methods are:

- `doGet()`
- `doHead()`
- `doDelete()`
- `doOptions()`
- `doPost()`
- `doTrace()`

and correspond directly with the HTTP protocol methods.

As shown in the following diagram, the `service()` method interprets each HTTP method and determines if it is an HTTP `GET`, HTTP `POST`, HTTP `HEAD`, or other HTTP protocol method:



The class `HttpServlet` is actually rather intelligent. Not only does it dispatch HTTP requests, it detects which methods are overridden in a subclass and can report back to a client on the capabilities of the server. (Simply by overriding the `doGet()` method causes the class to respond to an HTTP `OPTIONS` method with information that `GET`, `HEAD`, `TRACE`, and `OPTIONS` are all supported. These capabilities are in fact all supported by the class's code).

In another example of the support `HttpServlet` provides, if the `doGet()` method is overridden, there is an automatic response generated for the HTTP `HEAD` method. (Since the response to an

HTTP `HEAD` method is identical to an HTTP `GET` method--minus the body of the message--the `HttpServlet` class can generate an appropriate response to an HTTP `HEAD` request from the reply sent back from the `doGet()` method). As you might expect, if you need more precise control, you can always override the `doHead()` method and provide a custom response.

Using the HTTP Support Classes

When using the HTTP support classes, you generally create a new servlet that extends `HttpServlet` and overrides either `doGet()` or `doPost()`, or possibly both. Other methods can be overridden to get more fine-grained control.

The HTTP processing methods are passed two parameters, an `HttpServletRequest` object and an `HttpServletResponse` object. The `HttpServletRequest` class has several convenience methods to help parse the request, or you can parse it yourself by simply reading the text of the request.

A servlet's `doGet()` method should

- Read request data, such as input parameters
- Set response headers (length, type, and encoding)
- Write the response data

It is important to note that the handling of a `GET` method is expected to be *safe* and *idempotent*.

- Handing is considered *safe* if it does not have *any* side effects for which users are held responsible, such as charging them for the access or storing data.
- Handling is considered *idempotent* if it can safely be repeated. This allows a client to repeat a `GET` request without penalty.

Think of it this way: `GET` should be "looking without touching." If you require processing that has side effects, you should use another HTTP method, such as `POST`.

A servlet's `doPost()` method should be overridden when you need to process an HTML form posting or to handle a large amount of data being sent by a client. HTTP `POST` method handling is discussed in detail later.

`HEAD` requests are processed by using the `doGet()` method of an `HttpServlet`. You could simply implement `doGet()` and be done with it; any document data that you write to the response output stream will *not* be returned to the client. A more efficient implementation, however, would check to see if the request was a `GET` or `HEAD` request, and if a `HEAD` request, not write the data to the response output stream.

Summary

The Java Servlet API is a *standard extension*. This means that there is an explicit definition of servlet interfaces, but it is not part of the Java Development Kit (JDK) 1.1 or the Java 2 platform. Instead, the servlet classes are delivered with the Java Servlet Development Kit (JSDK) version 2.0 from Sun (http://java.sun.com/products/servlet/). This JSDK version is intended for use with both JDK 1.1 and the Java 2 platform. There are a few significant differences between JSDK 2.0 and JSDK 1.0. See below for details. If you are using a version of JSDK earlier than 2.0, it is recommended that you upgrade to JSDK 2.0.

Servlet support currently spans two packages:

javax.servlet: General Servlet Support

`Servlet`	An interface that defines communication between a web server and a servlet. This interface defines the `init()`, `service()`, and `destroy()` methods (and a few others).
`ServletConfig`	An interface that describes the configuration parameters for a servlet. This is passed to the servlet when the web server calls its `init()` method. Note that the servlet should save the reference to the `ServletConfig` object, and define a `getServletConfig()` method to return it when asked. This interface defines how to get the initialization parameters for the servlet and the context under which the servlet is running.
`ServletContext`	An interface that describes how a servlet can get information about the server in which it is running. It can be retrieved via the `getServletContext()` method of the `ServletConfig` object.
`ServletRequest`	An interface that describes how to get information about a client request.

ServletResponse	An interface that describes how to pass information back to the client.
GenericServlet	A base servlet implementation. It takes care of saving the `ServletConfig` object reference, and provides several methods that delegate their functionality to the `ServletConfig` object. It also provides a dummy implementation for `init()` and `destroy()`.
ServletInputStream	A subclass of `InputStream` used for reading the data part of a client's request. It adds a `readLine()` method for convenience.
ServletOutputStream	An `OutputStream` to which responses for the client are written.
ServletException	Should be thrown when a servlet problem is encountered.
UnavailableException	Should be thrown when the servlet is unavailable for some reason.

javax.servlet.http: Support for HTTP Servlets

HttpServletRequest	A subclass of `ServletRequest` that defines several methods that parse HTTP request headers.
HttpServletResponse	A subclass of `ServletResponse` that provides access and interpretation of HTTP status codes and header information.
HttpServlet	A subclass of `GenericServlet` that provides automatic separation of HTTP request by method type. For example, an HTTP `GET` request will be processed by the `service()` method and passed to a `doGet()` method.
HttpUtils	A class that provides assistance for parsing HTTP `GET` and `POST` requests.

Servlet Examples

Now for an in-depth look at several servlets. These examples include:

- Generating Inline Content
- Processing HTTP Post Requests
- Using Cookies
- Maintaining Session Information
- Connecting to Databases

Generating Inline Content

Sometimes a web page needs only a small piece of information that is customized at runtime. The remainder of a page can be static information. To substitute only small amounts of information, some web servers support a concept known as server-side includes, or SSI.

If it supports SSI, the web server designates a special file extension (usually `.shtml`) which tells the server that it should look for SSI tags in the requested file. The JWS defines a special SSI tag called the `<servlet>` tag, for example:

```
<servlet code="DatePrintServlet">
  <param name=timezone value=pst>
</servlet>
```

This tag causes the invoking of a servlet named `DatePrintServlet` to generate some in-line content.

SSI and servlets allow an HTML page designer to write a skeleton for a page, using servlets to fill in sections of it, rather than require the servlet to generate the entire page. This is very useful for features like page-hit counters and other small pieces of functionality.

The `DatePrintServlet` servlet works just like a regular servlet except that it is designed to provide a very small response and not a complete HTML page. The output MIME type gets set to "text/plain" and not "text/html".

Keep in mind that the syntax of server-side includes, if they are even supported, may vary greatly from one web server to another.

In the following Magercise you create the `DatePrintServlet` and see how to use it in an HTML page.

Magercise

4. Generating In-line Content

Processing HTTP Post Requests

HTTP `POST` method processing differs from HTTP `GET` method processing in several ways. First, because `POST` is expected to modify data on the server, there can be a need to safely handle updates coming from multiple clients at the same time. Second, because the size of the information stream sent by the client can be very large, the `doPost()` method must open an `InputStream` (or `Reader`) from the client to get any of the information. HTTP `POST` does *not* support sending parameters encoded inside of the URL as does the HTTP `GET` method.

The problem of supporting simultaneous updates from multiple clients has been solved by database systems (DBMSs); unfortunately the HTTP protocol does not work well with database systems. This is because DBMSs need to maintain a persistent connection between a client and the DBMS to determine which client is trying to update the data.

The HTTP protocol does not support this type of a connection as it is a message based, stateless protocol. Solving this problem is not easy and never elegant! Fortunately, the Servlet API defines a means to track client/server sessions. This is covered in the Maintaining Session Information section later in this course. Without session management tracking, one can resort to several different strategies. They all involve writing data to the client in hidden fields which is then sent back to the server. The simplest way to handle updates is to use an optimistic locking scheme based on date/time stamps. One can use a single date/time stamp for a whole form of data, or one could use separate date/time stamps for each "row" of information.

Once the update strategy has been selected, capturing the data sent to the server via an HTTP `POST` method is straightforward. Information from the HTML form is sent as a series of parameters (name/value pairs) in the `InputStream` object. The `HttpUtils` class contains a method `parsePostData()` that accepts the raw `InputStream` from the client and return a `Hashtable` with the parameter information already processed. A really nice feature is that if a parameter of a given name has multiple values (such is the case for a column name with multiple rows), then this information can be retrieved from the `Hashtable` as an array of type `String`.

In the following Magercise, you will be given skeleton code that implements a pair of servlets that display data in a browser as an editable HTML form. The structure of the data is kept separate from the actual data. This makes it easy to modify this code to run against arbitrary tables from a JDBC-connected database.

Magercise

5. Posting and Processing HTML Forms

Using Cookies

For those unfamiliar with cookies, a cookie is a named piece of data maintained by a browser, normally for session management. Since HTTP connections are stateless, you can use a cookie to store persistent information accross multiple HTTP connections. The `Cookie` class is where all the "magic" is done. The `HttpSession` class, described next, is actually easier to use. However, it doesn't support retaining the information across multiple browser sessions.

To save cookie information you need to create a `Cookie`, set the content type of the `HttpServletResponse` response, add the cookie to the response, and then send the output. You must add the cookie after setting the content type, but before sending the output, as the cookie is sent back as part of the HTTP response header.

```
private static final String SUM_KEY = "sum";
...
int sum = ...; // get old value and add to it
Cookie theCookie =
  new Cookie (SUM_KEY, Integer.toString(sum));
response.setContentType("text/html");
response.addCookie(theCookie);
```

It is necessary to remember that all cookie data are strings. You must convert information like `int` data to a String object. By default, the cookie lives for the life of the browser session. To enable a cookie to live longer, you must call the `setMaxAge(interval)` method. When positive, this allows you to set the number of seconds a cookie exists. A negative setting is the default and destroys the cookie when the browser exits. A zero setting immediately deletes the cookie.

Retrieving cookie data is a little awkward. You cannot ask for the cookie with a specific key. You must ask for all cookies and find the specific one you are interested in. And, it is possible that multiple cookies could have the same name, so just finding the first setting is not always sufficient. The following code finds the setting of a single-valued cookie:

```
int sum = 0;
Cookie theCookie = null;
Cookie cookies[] = request.getCookies();
if (cookies != null) {
  for(int i=0, n=cookies.length; i < n; i++) {
    theCookie = cookies[i];
    if (theCookie.getName().equals(SUM_KEY)) {
      try {
        sum = Integer.parseInt(theCookie.getValue());
```

```
        } catch (NumberFormatException ignored) {
          sum = 0;
        }
        break;
      }
    }
  }
```

The complete code example shown above is available for testing.

Maintaining Session Information

A session is a continuous connection from the same browser over a fixed period of time. (This time is usually configurable from the web server. For the JWS, the default is 30 minutes.) Through the implicit use of browser cookies, HTTP servlets allow you to maintain session information with the `HttpSession` class. The `HttpServletRequest` provides the current session with the `getSession(boolean)` method. If the boolean parameter is `true`, a new session will be created when a new session is detected. This is, normally, the desired behavior. In the event the parameter is `false`, then the method returns `null` if a new session is detected.

```
public void doGet (HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
  HttpSession session = request.getSession(true);
  // ...
```

Once you have access to an `HttpSession`, you can maintain a collection of key-value-paired information, for storing any sort of session-specific data. You automatically have access to the creation time of the session with `getCreationTime()` and the last accessed time with `getLastAccessedTime()`, which describes the time the last servlet request was sent for this session.

To store session-specific information, you use the `putValue(key, value)` method. To retrieve the information, you ask the session with `getValue(key)`. The following example demonstrates this, by continually summing up the integer value of the *Addend* parameter. In the event the value is not an integer, the number of errors are also counted.

```
private static final String SUM_KEY =
  "session.sum";
private static final String ERROR_KEY =
  "session.errors";
Integer sum = (Integer) session.getValue(SUM_KEY);
int ival = 0;
if (sum != null) {
  ival = sum.intValue();
}
try {
```

```
    String addendString =
    request.getParameter("Addend");
    int addend = Integer.parseInt (addendString);
    sum = new Integer(ival + addend);
    session.putValue (SUM_KEY, sum);
  } catch (NumberFormatException e) {
    Integer errorCount =
      (Integer)session.getValue(ERROR_KEY);
    if (errorCount == null) {
      errorCount = new Integer(1);
    } else {
      errorCount = new Integer(errorCount.intValue()+1);
    }
    session.putValue (ERROR_KEY, errorCount);
  }
```

As with all servlets, once you've performed the necessary operations, you need to generate some output. If you are using sessions, it is necessary to request the session with `HttpServletRequest.getSession()` before generating any output.

```
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<html>" +
    "<head><title>Session Information</title></head>" +
    "<body bgcolor=\"#FFFFFF\">" +
    "<h1>Session Information</h1><table>");
  out.println ("<tr><td>Identifier</td>");
  out.println ("<td>" + session.getId() + "</td></tr>");
  out.println ("<tr><td>Created</td>");
  out.println ("<td>" + new Date(
    session.getCreationTime()) + "</td></tr>");
  out.println ("<tr><td>Last Accessed</td>");
  out.println ("<td>" + new Date(
    session.getLastAccessedTime()) + "</td></tr>");
  out.println ("<tr><td>New Session?</td>");
  out.println ("<td>" + session.isNew() + "</td></tr>");
  String names[] = session.getValueNames();
  for (int i=0, n=names.length; i<n; i++) {
    out.println ("<tr><td>" + names[i] + "</td>");
    out.println ("<td>" + session.getValue (names[i])
      + "</td></tr>");
  }
  out.println("</table></center></body></html>");
  out.close();
```

The complete code example shown above is available for testing. One thing not demonstrated in the example is the ability to *end* a session, where the next call to `request.getSession(true)` returns a different session. This is done with a call to `invalidate()`.

In the event a user has browser cookies disabled, you can encode the session ID within the `HttpServletResponse` by calling its `encodeUrl()` method.

Connecting to Databases

It is very common to have servlets connect to databases through JDBC. This allows you to better control access to the database by only permitting the middle-tier to communicate with the database. If your database server includes sufficient simultanious connection licenses, you can even setup database connections once, when the servlet is initialized, and pool the connections between all the different service requests.

The following demonstrates sharing a single `Connection` between all service requests. To find out how many simultaneous connections the driver supports, you can ask its `DatabaseMetaData` and then create a pool of `Connection` objects to share between service requests.

- In the `init()` method connect to the database.

```
Connection con = null;
public void init (ServletConfig cfg)
  throws ServletException {
  super.init (cfg);
  // Load driver
  String name = cfg.getInitParameter("driver");
  Class.forName(name);
  // Get Connection
  con = DriverManager.getConnection (urlString);
}
```

- In the `doGet()` method retrieve database information.

```
public void doGet (HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
  response.setContentType("text/html");

  // Have browser ignore cache - force reload
  response.setHeader ("Expires",
    "Mon, 01 Jan 1990 00:00:00 GMT");

  Statement stmt = null;
  ResultSet result = null;

  try {
    // Submit query
    stmt = con.createStatement();
    result = stmt.executeQuery (
      "SELECT programmer, cups " +
      "FROM JoltData ORDER BY cups DESC;");

    // Create output
    PrintWriter out = response.getWriter();
    while(result.next()) {
      // Generate output from ResultSet
    }
  } finally {
    if (result != null) {
      result.close();
    }
    if (stmt != null) {
```

```
        stmt.close();
      }
    }
    out.flush();
    out.close();
  }
```

- In the `destroy()` method disconnect from the database.

```
public void destroy() {
  super.destroy();
  con.close();
}
```

It is not good practice to leave a database connection permanently open, so this servlet should not be installed as a permanent servlet. Having it as a temporary servlet that closes itself down after a predefined period of inactivity allows the sharing of the database connection with requests that coincide, reducing the cost of each request.

You can also save some information in the `HttpSession` to possible page through the result set.

Security Issues

As with Java applets, Java servlets have security issues to worry about, too.

The Servlet Sandbox

A servlet can originate from several sources. A webmaster may have written it; a user may have written it; it may have been bought as part of a third-party package or downloaded from another web site.

Based on the source of the servlet, a certain level of trust should be associated with that servlet. Some web servers provide a means to associate different levels of trust with different servlets. This concept is similar to how web browsers control applets, and is known as "sandboxing".

A servlet sandbox is an area where servlets are given restricted authority on the server. They may not have access to the file system or network, or they may have been *granted* a more trusted status. It is up to the web server administrator to decide which servlets are granted this status. Note that a fully trusted servlet has full access to the server's file system and networking capabilities. It could even perform a `System.exit()`, stopping the web server...

Access Control Lists (ACLs)

Many web servers allow you to restrict access to certain web pages and servlets via access control lists (ACLs). An ACL is a list of users who are allowed to perform a specific function in the server. The list specifies:

- What kind of access is allowed
- What object the access applies to
- Which users are granted access

Each web server has its own means of specifying an ACL, but in general, a list of users is registered on the server, and those user names are used in an ACL. Some servers also allow you to add users to logical groups, so you can grant access to a group of users without specifying all of them explicitly in the ACL.

ACLs are extremely important, as some servlets can present or modify sensitive data and should be tightly controlled, while others only present public knowledge and do not need to be controlled.

Threading Issues

A web server can call a servlet's `service()` method for several requests at once. This brings up the issue of thread safety in servlets.

But first consider what you *do not* need to worry about: a servlet's `init()` method. The `init()` method will *only* be called *once* for the duration of the time that a servlet is loaded. The web server calls `init()` when loading, and will not call it again unless the servlet has been unloaded and reloaded. In addition, the `service()` method or `destroy()` method *will not* be called until the `init()` method has completed its processing.

Things get more interesting when you consider the `service()` method. The `service()` method can be called by the web server for multiple clients at the same time. (With the JSDK 2.0, you can tag a servlet with the `SingleThreadModel` interface. This results in each call to `service()` being handled serially. Shared resources, such as files and databases, can still have concurrency issues to handle.)

If your `service()` method uses outside resources, such as instance data from the servlet object, files, or databases, you need to carefully examine what might happen if multiple calls are made to `service()` at the same time. For example, suppose you had defined a counter in your servlet class that keeps track of how many `service()` method invocations are currently running:

```
private int counter = 0;
```

Next, suppose that your `service()` method contained the following code:

```
int myNumber = counter + 1;   // line 1
counter = myNumber;           // line 2

// rest of the code in the service() method

counter = counter - 1;
```

What would happen if two `service()` methods were running at the same time, and both executed line 1 before either executed line 2? Both would have the same value for `myNumber`, and the `counter` would not be properly updated.

For this situation, the answer might be to synchronize the access to the counter variable:

```
synchronized(this) {
  myNumber = counter + 1;
  counter = myNumber;
}

// rest of code in the service() method

synchronized(this) {
  counter = counter - 1 ;
}
```

This ensures that the counter access code is executed only one thread at a time.

There are several issues that can arise with multi-threaded execution, such as deadlocks and coordinated interactions. There are several good sources of information on threads, including Doug Lea's book *Concurrent Programming in Java*.

JSDK 1.0 and JSDK 2.0

The Java Servlet Development Kit (JSDK) provides servlet support for JDK 1.1 and Java 2 platform developers.

JSDK 1.0 was the initial release of the development kit. Everything worked fine, but there were some minor areas that needed improvement. The JSDK 2.0 release incorporates these improvements. The changes between JSDK 1.0 and JSDK 2.0 are primarily the addition of new classes. In addition, there is also one deprecated methods. Because some web servers still provide servlet support that complies with the JSDK 1.0 API definitions, you need to be careful about upgrading to the new JSDK.

New Servlet Features in JSDK 2.0

JSDK 2.0 adds the following servlet support:

- The interface `SingleThreadModel` indicates to the server that only one thread can call the `service()` method at a time.
- `Reader` and `Writer` access from `ServletRequest` and `ServletResponse`
- Several HTTP session classes that can be used to provide state information that persists over multiple connections and requests between an HTTP client and an HTTP server.
- Cookie support is now part of the standard servlet extension.
- Several new HTTP response constants have been added to `HttpServletResponse`
- Delegation of `DELETE`, `OPTIONS`, `PUT`, and `TRACE` to appropriate methods in `HttpServlet`

JSDK 2.0 deprecated one method:

- `getServlets()` --you should use `getServletNames()` instead

For More Information

Interest and support for servlets is exploding. Here are some links to help you keep up to date:

- Main servlet support page at Sun http://java.sun.com/products/servlet/
- List of environments supporting servlets, maintained by Sun, http://java.sun.com/products/servlet/runners.html
- Information on the HTTP protocol at the World Wide Web Consortium's protocol page http://www.w3.org/Protocols
- Servlet Central web magazine http://www.servletcentral.com
- JRun Magazine http://www.jrunmag.com/
- *Java Servlets* book from Karl Moss; McGraw-Hill - ISBN 0-07-913779-2
- *Java Servlet Programming* book from Jason Hunter and William Crawford; O'Reilly - ISBN 1-56592-391-X