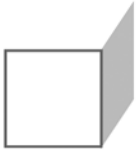
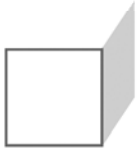
Chapter 2

Servlets Architecture





OBJECTIVES

After completing "Servlets Architecture," you will be able to:

- **Identify the various packages that constitute the Servlets API.**
- **Understand the types in the javax.servlet.http package, and how they are useful in constructing Servlets.**
  - Why do we need **HttpServlet**, a concrete implementation of the generic **Servlet** interface?
  - What is the role played by **HttpServletRequest** and **HttpServletResponse** objects?
  - What are servlet attributes?
- **Understand the generic servlet structure.**
- **Understand and appreciate the role of web containers in providing the runtime environment for servlets.**
- **Gain some insights into the life cycle of servlets – from its birth to destruction – within the runtime environment.**

Understanding the DateTime Servlet

- In last chapter, we worked with some simple servlet examples. Let us take our first example and try to understand the code, bit by bit.
  - Look in **Examples\DateTime\Step0**.
  - See **src\cc\datetime\DateTime.java**.

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

public class DateTimeServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request,
      HttpServletResponse response)
    throws IOException, ServletException
  {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println ("<html>");
    out.println ("<head>");
    out.println ("  <title>Date and Time
                              Example</title>");
    out.println ("</head>");
    ...
    out.println ("  <p>Today's date and time is " +
                         new Date () + ".</p>");
    ...
  }
}
```

Imported Packages

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

- **The two most important packages that are imported in almost all servlet source files are javax.servlet and javax.servlet.http.**
  - From the previous chapter we can recollect that most of the servlet classes and interfaces belonged to one of these two packages.
  - Additional packages might be required, depending on what we are trying to do within the servlet.
  - In the present example, we need the **java.io** package to write the HTML output back to the HTTP stream.
  - We also imported **java.util** package for the **Date** class we are using.

HttpServlet Class

```
public class DateTimeServlet extends HttpServlet
{
```

- **You should be wondering why we are 'extending' the javax.servlet.http. HttpServlet class instead of 'implementing' the basic javax.servlet.Servlet interface.**
- **Remember that the protocol we need to speak is HTTP, and that the Servlets API is not tied to HTTP.**
  - **javax.servlet. Servlet** is only a basic interface. But **javax.servlet.http. HttpServlet** is an abstract class that implements the basic **Servlet** interface for talking HTTP.
  - By extending **javax.servlet.http.HttpServlet**, we need not worry about the nuances of the HTTP protocol and instead can concentrate on business logic (e.g. displaying the date).
  - For web application servlets, **javax.servlet.http.HttpServlet** is the only class that needs to be extended.
  - In case you want to talk custom protocols (say WAP), you may write your own implementations of the **Servlet** interface.

HttpServlet Methods

- **HttpServlet** provides several key methods, any or all of which can be overridden to handle a certain sort of HTTP request:

```
public class javax.servlet.http.HttpServlet
  implements javax.servlet.Servlet
{
  ...
  /* This is not a comprehensive listing */
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response);
  protected void doPost(HttpServletRequest request,
                       HttpServletResponse response);
  protected void doPut(HttpServletRequest request,
                       HttpServletResponse response);
  ...
}
```

- The **service** implementation in the base class dispatches the request to one of these methods, based on the HTTP verb.
- The most useful methods in the above group are **doGet** and **doPost**.
  - When you are expecting your clients to invoke the servlet through an HTTP GET request, you can implement **doGet**. Otherwise use **doPost**.
  - If you are expecting both types of requests, implement both!
  - A common approach is to have one delegate to the other, since their signatures are the same.

Request and Response Objects

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

- **The doXXX methods have two important arguments, of types HttpServletRequest and HttpServletResponse.**
  - These interfaces provide methods to access the underlying input and output streams associated with the client connection.
  - Objects are created by the servlet container and passed to **doXXX** methods for further processing by applications.
  - The **request** object encapsulates the request made by the client to the servlet; **response** encapsulates the response returned by the servlet to the client.
  - In the current example, we do not expect any inputs from the calling client applications; hence we may not need access to the **request** object.
  - However, since we need to return an HTML stream, containing the date value, we will be making use of **response**.

HttpServletRequest Methods

- Here are some of the most commonly used methods on **ServletRequest**:

```
public interface javax.servlet.ServletRequest
{
  public String getParameter (String key);
  public Enumeration getParameterNames ();
  public String[] getParameterValues (String key);

  public String getContentType ();
  public ServletInputStream getInputStream ();
}
```

- **getParameter** and related methods provide access to parameters included in the HTTP request.
  - These may have been encoded in the URL in CGI syntax, in the case of HTTP GET:

```
http://my.com/MyServlet?name=Gokul&course=Servlets
```

  - For HTTP POST, parameters are passed as part of the request body.
  - In either case, parameters are name-value pairs. For the above request, **getParameter ("course")** would return a Java String with the value "Servlets".
  - Names are typically unique, but need not be, and in some cases multiple values can be grouped under one name.

HttpServletResponse Methods

- Having explored the request object, let us get to know HttpServletResponse object, which implements the **javax.servlet.ServletResponse** interface:

```
public interface javax.servlet.ServletResponse
{
  public void setContentType(String type);
  public void setContentLength(int size);
  public ServletOutputStream getOutputStream();
  public PrintWriter getWriter( );
  public void sendRedirect (String URL);
}
```

  - In our example, we have used two of these methods:

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

  - The content type is a MIME type identifier – the most common type will be "text/html", but "text/plain" and "text/xml" are also common.
  - **getWriter** gives us a handy reference to the response output stream as a **java.io.PrintWriter**, which can then be used to produce character output that is handed to the client browser.

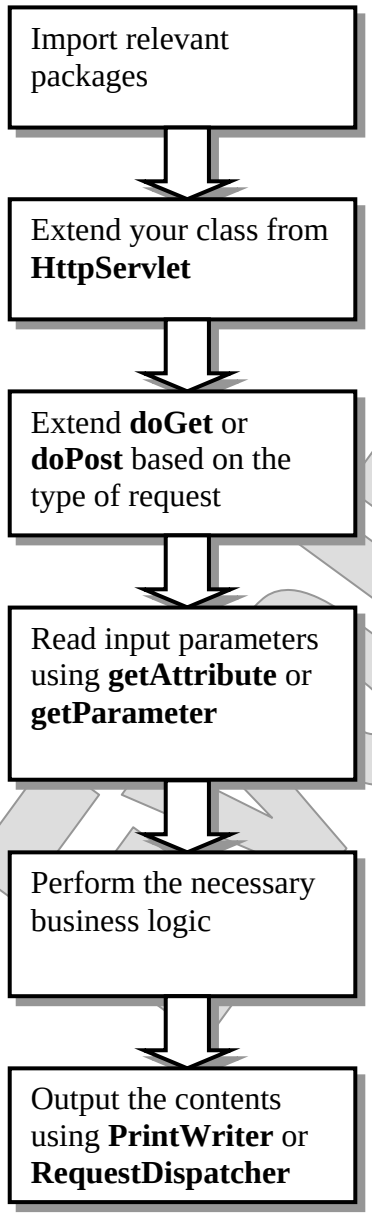
Writing the HTML Response

```
out.println ("<html>");
out.println ("<head>");
out.println ("  <title>Date and Time
                          Example</title>");
out.println ("</head>");
...
out.println ("  <p>Today's date and time is " +
                    new Date () + ".</p>");
...
```

- **The last lines of DateTime.java simply write out a few HTML lines to the standard PrintWriter object out.**
  - One of the biggest shortcomings of servlets is hidden in these lines …
  - When you want to change something in the outgoing HTML, **you must edit the servlet code directly**, recompile the servlet and re-deploy!
  - This could be a painstaking process, and it must be undertaken to implement changes not only in **dynamic functions**, but also in basic look-and-feel or **page design**.
  - We will be seeing a few ways out of this bind: template parsers and forwarding to "view" handlers after processing.
- **A cleaner and more efficient way to output HTML to the client is to use JSP.**
  - JavaServer Pages or JSPs are another breed of server-side Java technology. With JSPs, it is much easier to control the way output is directed to clients.

Generic Servlet Structure

- **With what we have discussed so far, let us prepare a small mind-map of how to construct servlets:**




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


public class DateTimeServlet
    extends HttpServlet

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

String Input1 =
  request.getParameter("Input1");


...


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

Redirecting the Web Server

- **sendRedirect is another useful method.**
  - This method is best used for redirecting the client to a static HTML page – "Thank you for your input" or "Sorry, we are unable to process your request".

```
response.sendRedirect ("Confirmation.html");
```

  - This may be done after the servlet has done relevant processing; in fact the "static" file may have just been created, although this is not all that common.
- **One can also redirect to another dynamic resource.**
  - It is also possible to invoke another servlet or Java class directly in order to reuse functionality while retaining control over the final output stream.

The RequestInfo Servlet

LAB 2A

In this lab you will write a servlet called **HttpRequestInfo**, compile it and deploy it to Tomcat. You can have the **DateTime** servlet you deployed in the earlier chapter, as a reference – which you can modify to write the **HttpRequestInfo s**ervlet.

This exercise allows you to understand the intricacies of writing a servlet and deploying it in a standard manner – the same procedure is applicable for all servlets you'll be writing in the future.

Detailed instructions are contained in the Lab 2A write-up at the end of the chapter.

Suggested time: 30 minutes

The Redirector Servlet

LAB 2B

In this lab you will write another servlet called **HttpRedirector**. The job of this servlet is to redirect the request to a static HTML page. You'll use the **sendRedirect** method of the response object to achieve this.

This exercise will familiarize you with the usage of various APIs available with request and response objects. Also, **sendRedirect** is a pretty useful method for practical web applications, when one needs to redirect a response to a static HTML page.

Detailed instructions are contained in the Lab 2B write-up at the end of the chapter.

Suggested time: 30 minutes

Attributes and Scopes

- **A major difference between J2SE applications and servlet programming lies in how state is managed.**
- **Java classes typically store transient state in fields.**
- **Due to pooling of servlets and other container behaviors intended to assure scalability, this is not a sound strategy for servlets.**
- **Instead, the container provides several means of storing transient state as named attributes.**
- **Attributes are managed by the application using four maps provided by the container. Each of these is associated with a different scope – these are listed here from most general to most specific:**
  - **Application** scope is shared by all components in the same web application, and attributes placed here live as long as the application does – often from deployment to undeployment.
  - **Session** scope is shared by components as they handle requests from a common client, and attributes placed here live until the client disconnects or the session "times out."
  - **Request** scope is shared by components over the duration of a single request/response cycle. It allows a servlet to pass information along to a component that will take over request processing later in the cycle.
  - **Page** scope is not shared by other components; it is something like local storage for handling one request.

Attribute Methods

- The various scopes are implemented in different places in the Servlets API.
- The first that we'll study is available on the request object. This is a natural place to implement the request-scope attribute map:

```
public interface javax.servlet.ServletRequest
{
  public Object getAttribute (String name);
  public Enumeration getAttributeNames ();
  public void setAttribute (String name, Object o);
  public void removeAttribute (String name);
}
```

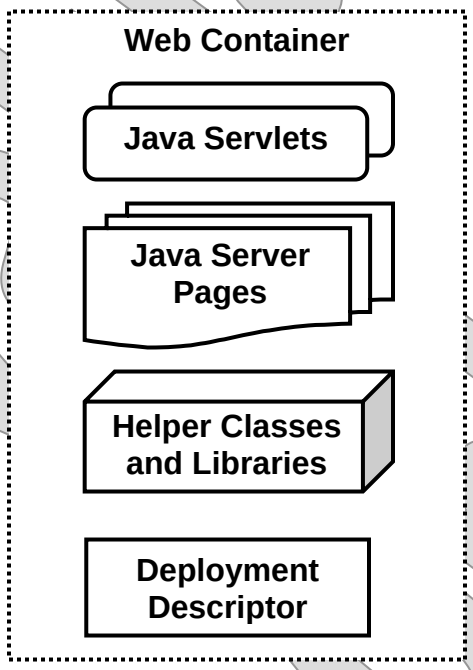
- At first, it's easy to confuse **attributes** and **parameters**.
  - Request **parameters** are those values provided as part of the **HTTP request** – either CGI-encoded in the HTTP GET URL or listed one per text line in the body of an HTTP POST.
  - Request **attributes** are generated and used entirely on the **server side** of the cycle, and provide a means for multiple components handling a request to share information.
  - Until we start using multiple components to handle requests, this is not likely to seem very useful!
  - An architecture known as "Model 2" dictates that almost every request will be handled by a chain of components, at least one **controller** and one **view**. In this architecture, sharing attributes and forwarding is a key technique.

The Servlet Container

- **Java Servlets defines the behavior of containers.**
  - Though many different flavors of containers are currently available in the market, from different vendors, their basic behavior is the same.
  - This makes it possible to program servlets that are vendor-independent, or **portable**.
  - In other words, the servlets we write for one container can be ported to a different vendor's container – with very little or no code change!
  - For example, it is a very common practice to develop servlets in Tomcat (which is free) and then host them in a commercial servlet container such as WebLogic or WebSphere.
  - If everything has been done to the specification – that is, to the contract – then none of the Java code and very little of the XML in the deployment descriptor will have to change when this porting is done.

Web Containers

- **Servlet containers are now known by a more generic name – "web containers."**
  - Since the introduction of **JavaServer Pages** technology, many Java web applications combine a host of JSPs, servlets, supporting helper classes, libraries and static resources such as XML.
  - The J2EE specifications define standard ways to package and deploy such web applications to a container.
  - The servlet containers were modified and expanded suitably to handle these features and were given their new name.



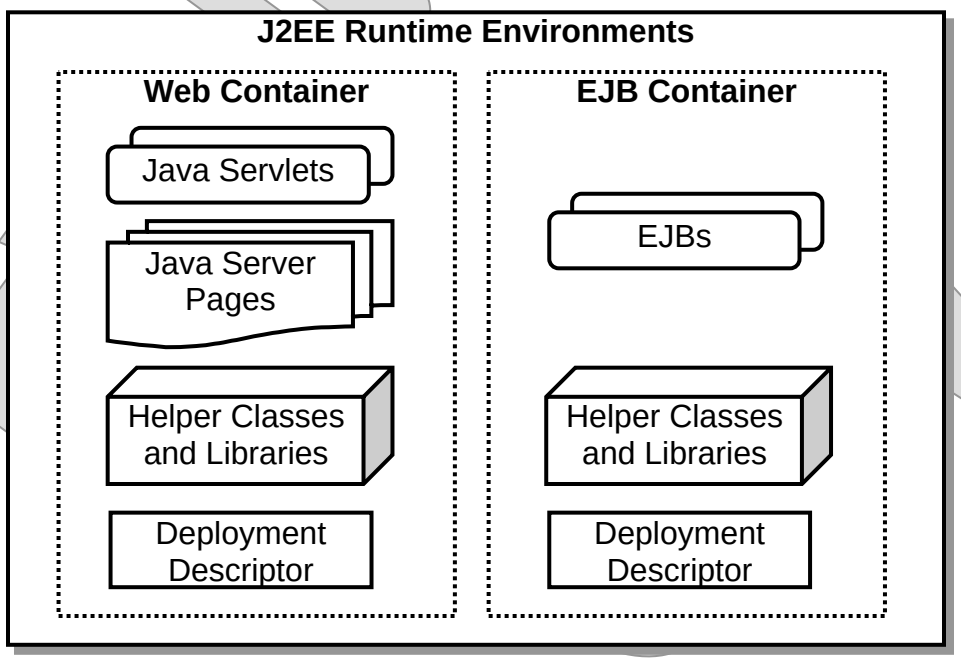
  - You might be wondering why JSP didn't require a separate container for its own purposes – can you think of any reasons?

JSPs and Servlets in Web Containers

- **Beneath the script-heavy surface, JSPs are nothing but Java servlets!**
  - In fact, all JSPs are translated to servlets – often just before being served.
  - Thus it was needless to have separate containers to serve JSPs. Servlet containers were enhanced to handle JSPs and understand their deployment descriptors – thus becoming "web containers."
  - But things have started changing as the JSP 2.0 specifications have taken shape, with JSP evolving in many ways – independent of servlets.
  - The day may not be far off when JSPs do indeed need to be hosted in exclusive containers!

EJB Containers

- **It's interesting to note that Enterprise JavaBeans (or EJBs, if you are Java-savvy) do need to be hosted in their own independent containers.**
  - Enterprise JavaBeans is another integral technology defined under the J2EE umbrella.
  - As a group they have a dramatically different purpose, different life cycle needs, and much more complex deployment descriptions and features than JSPs and servlets.
  - Today, in a comprehensive J2EE application server, only these two containers dominate the landscape: web containers and EJB Containers.

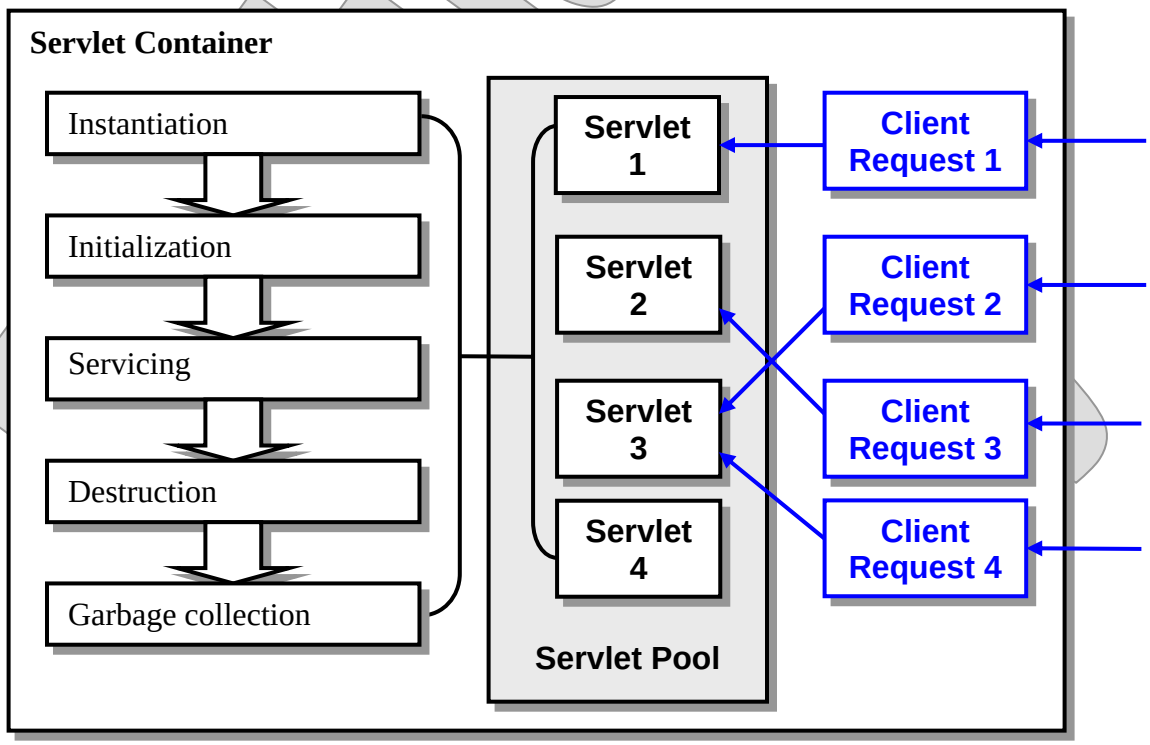


Servlet Life Cycle and Containers

- **The servlet container controls a servlet's life cycle!**
  - This is in striking contrast to J2SE applications, which are instantiated and destroyed by the user.
  - There is no need to use the constructor or **main** method to instantiate and invoke servlet objects.
  - The reason is that the **servlet container** takes responsibility for instantiating and making the object available for use as well as destroying it when it is not needed anymore.
- **Why is life cycle management left to containers?**
  - Container management of J2EE components allows a number of **enterprise features** to be implemented in the container, rather than the component: **security**, **scalability**, and transaction control are some common examples.
  - These features can be **implemented generically** by the container, with just a little declarative help from the **deployment descriptor**.
  - In this way, the programmer can concentrate on developing business logic, instead of writing plumbing code to do low-level system management.
  - One important feature enabled by this management relationship is **pooling**: the ability of the container to keep a limited-size set or **pool** of servlet instances to serve any number of incoming requests. This cuts the costs of object creation and destruction, and is the key to scalability in servlet container implementations.

Servlet Life Cycle

- A servlet's life cycle follows through several stages –from its instantiation or **birth** to destruction or **death**:
  - **Instantiation** via an ordinary **new** operation – but invoked by the container, not another application class
  - **Initialization** via a lifecycle method **init**
  - **Service** (most time spent here, handling any number of HTTP requests from any number of clients)
  - **Destruction** via a lifecycle method **destroy**
  - **Garbage collection** as with any Java object



The Life Cycle Methods

- **As the servlet passes through different stages of its life cycle, the container invokes some of the methods defined in javax.servlet.Servlet interface:**

```
public interface javax.servlet.Servlet
{
  public void init (ServletConfig config);
  public void service (ServletRequest request,
                       ServletResponse response);
  public void destroy ();
  public String getServletInfo ();
}
```

  - As the container invokes **init**, it also passes the **ServletConfig** object as a parameter. This allows a servlet to access its **ServletConfig** object and read its configuration data. **init** is called only once, throughout the lifecycle of a servlet.
  - As we saw earlier, for an instance of **HttpServlet**, the base **service** method delegates to **doXXX** methods, based on the kind of HTTP request.
  - The container calls the **destroy** method before marking the servlet for garbage collection. Once this method is completed, the servlet is not available anymore for servicing the requests; a new instance would have to be created and added to the container's pool.

Initialization and Destruction Tasks

- **During servlet initialization (i.e. within the init method), we could …**
  - **Initialize static fields** from our application's configuration files – but beware of this, as we'll explain later
  - **Read configuration** parameters using the **ServletConfig** object that is passed by the container to **init**
  - **Register a JDBC driver**
  - Initialize **database connections** and connection pools, if any
  - Initialize logging or other services for our application
- **As we move on to Chapter 6 – which deals with servlet configuration – we will start coding inside the init method.**
- **During destruction, one usually just cleans up any initializations.**
  - Many initialized object references can simply be let go – that is, normal Java **garbage collection** is sufficient.
  - Some however require **explicit cleanup** – for instance one must **close** a database connection or other external resource.

SUMMARY

- The Servlets API comprises two main packages: **javax.servlet** and **javax.servlet.http**.
- All HTTP servlets extend the abstract **HttpServlet** class and use one of the **doXXX** methods, as appropriate to the HTTP verbs that are expected.
- **HttpServletRequest** and **HttpServletResponse** objects are most useful for reading input and writing output.
- Web containers provide runtime environments for Java web applications – which include Servlets, JSPs, HTML, XML and other libraries.
- A servlet's life cycle is controlled by the web container, and not by the application code or an interactive user.
- A servlet's life cycle consists of five distinct states: instantiation, initialization, service, destruction and garbage collection.

The RequestInfo Servlet

LAB 2A

Introduction

In this lab you will write a servlet called **HttpRequestInfo**, compile it and deploy it to Tomcat. You can have the **DateTime** servlet you deployed in the earlier chapter, as a reference – which you can modify to write the **HttpRequestInfo** servlet.

This exercise allows you to understand the intricacies of writing a servlet and deploying it in a standard manner – the same procedure is applicable for all servlets you'll be writing in the future.

Suggested Time: 30 minutes

Directories:	**Labs\Lab2A**	(Starter files)
	Examples\DateTime\Step3	(Source code template)
	Examples\Request\Step1	(Backup of starter files)
	Examples\Request\Step2	(Answer)
Files:	**src\cc\httprequestinfo\HttpRequestInfo.java**	
	docroot\post.html	

Instructions

1. In this exercise, we will be writing a servlet to accept an HTTP GET or POST request from the browser-based client – and to print all the information regarding the incoming request as its response. To be more specific, we will be printing out the following information:
   - Method of Request (GET or POST)
   - Requested URI
   - Protocol
   - Path Info
   - Remote Address
   - Content Type
2. Review the code in **Examples\DateTime\Step3\cc\datetime\DateTime.java**, which we completed in the previous chapter's labs. This should serve as a reference for you to write your own servlets. Also, refer the "Generic Servlet Structure" diagram in this chapter and the comments you'll see in the starter source code.

3. Open the starter source code in **HttpRequestInfo.java**, and prepare the skeletal code first – putting the import packages, **HttpRequestInfo** class extended from **HttpServlet**, and a **doGet** method.
4. Make use of the following methods available with **HttpServletRequest** object, to extract all the information you need about the request:

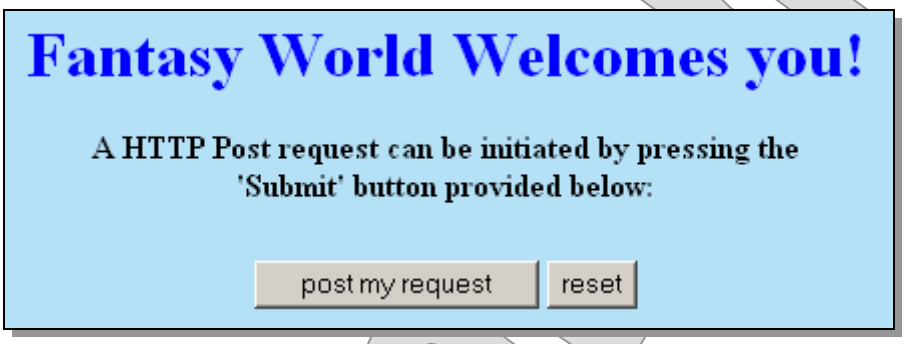
```
HttpServletRequest.getMethod()
HttpServletRequest.getRequestURI()
HttpServletRequest.getProtocol()
HttpServletRequest.getPathInfo()
HttpServletRequest.getRemoteAddr()
HttpServletRequest.getContentType()
```

5. You can either store the results of this method to some string variables – or, perhaps better, print them directly to the HTML stream!
6. Make use of the **response** object's **setContentType** method and **PrintWriter** object to produce the output. For sample code, refer to **DateTime.java**.
7. Compile the completed servlet class, and deploy it using **ant**. Test at the following URL:

`http://localhost:8080/HttpRequestInfo/HttpRequestInfo`

8. That covers GET requests. To handle POSTs, un-comment the implementation of **doPost** in the source file – this just delegates to **doGet**.
9. To see if the request is working, you have to initiate a POST request from a browser. A static HTML page is already available at the following URL for this purpose:

```
http://localhost:8080/HttpRequestInfo/post.html
```



10. Press the **post my request** button and observe the results. Compare these results with those you saw earlier. What are the significant differences?

The Redirector Servlet

LAB 2B

Introduction

In this lab you will write another servlet called **HttpRedirector**. The job of this servlet is to redirect the request to a static HTML page. You'll use the **sendRedirect** method of the response object to achieve this.

This exercise will familiarize you with the usage of various APIs available with request and response objects. Also, **sendRedirect** is a pretty useful method for practical web applications, when one needs to redirect a response to a static HTML page.

Suggested Time: 30 minutes

Directories:	**Labs\Lab2B**	(Starter files)
	Examples\Redirect\Step1	(Backup of starter files)
	Examples\Redirect\Step2	(Answer)
	Examples\Redirect\Step3	(Optional final answer)
Files:	**src\cc\httpredirector\HttpRedirector.java**	

Instructions

1. Start out by building the basic servlet infrastructure; see comments in the starter file **HttpRedirector.java** for reminders of what you need.
2. Now implement **doGet**. We don't need to check the request at all, and response is very simple: call the **sendRedirect** method, available on the **HttpServletResponse** object. The URL to which you will be redirecting the control is:

```
http://localhost:8080/HttpRedirector/redirect.html
```

3. Build and deploy using **ant**.

4. Try to invoke the servlet, using the following URL.

`http://localhost:8080/HttpRedirector/HttpRedirector`

(This is the **Step2** answer version.)

Optional Steps:

5. Note that you've redirected to a hard-coded URL – does this worry you at all? What if the entire application were relocated: how would your code perform? Try it: edit **%CATALINA_HOME%\conf\Catalina\localhost\HttpRedirector.xml**, changing the context URL:

```
<Context
  path="/Relocated"
  docBase="C:\Capstone\Servlets\Examples\Redirect\Step2/build"
  reloadable="true"
  antiJARLocking="true"
  antiResourceLocking="true"
/>
```

6. Shut down your browser, restart it, and clear its cache.
7. Wait for Tomcat to pick up the change, and try the new location:

`http://localhost:8080/Relocated/HttpRedirector`

8. You'll get an error 404, because the servlet, which was found without any trouble, redirected Tomcat to "http://localhost:8080/HttpRedirector/redirect.html" – which no longer resolves to a file on disk!
9. To make the servlet more robust in the face of application relocation, change the URL in the redirect call to a **relative** URL: call **request.getContextPath** and then concatenate the string "/redirect.html". This will find the HTML file relative to the request for the servlet itself. (Also, if you don't want to have to clean out the Tomcat **webapps** directory again, you can change the **war-file** and **webapps-subdir** values in **build.properties** before rebuilding.)
10. Rebuild and re-deploy, and test in the new location. You should now find that redirection works at any application location. (This is the optional **Step3** answer.)