

Servlet Basics

- Evaluating servlets vs. other technologies
- Understanding the role of servlets
- Building Web pages dynamically
- Looking at servlet code
- A servlet that generates HTML
- The basic structure of servlets
- The servlet life cycle
- How to deal with multithreading problems
- In-Class Exercise

The Advantages of Servlets Over “Traditional” CGI

- Java servlets are more efficient, easier to use, more powerful, more portable, safer, and cheaper than traditional CGI and many alternative CGI-like technologies.

1. Efficient

- With traditional CGI, a new process is started for each HTTP request. If the CGI program itself is relatively short, the overhead of starting the process can dominate the execution time.
- *With servlets, the Java virtual machine stays running and handles each request with a lightweight Java thread, not a heavyweight operating system process.*
- **Similarly, in traditional CGI, if there are N requests to the same CGI program, the code for the CGI program is loaded into memory N times.**
- *With servlets, however, there would be N threads, but only a single copy of the servlet class would be loaded. This approach reduces server memory requirements and saves time by instantiating fewer objects.*
- **Finally, when a CGI program finishes handling a request, the program terminates. This approach makes it difficult to cache computations, keep database connections open, and perform other optimizations that rely on persistent data.**
- *Servlets, however, remain in memory even after they complete a response, so it is straightforward to store arbitrarily complex data between client requests.*

2. Convenient

- Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such high-level utilities.
- In CGI, you have to do much of this yourself.
- Besides, if you already know the Java programming language, why learn Perl too? You're already convinced that Java technology makes for more reliable and reusable code than does Visual Basic, VBScript, or C++. Why go back to those languages for server-side programming?

3. Powerful

- Servlets support several capabilities that are difficult or impossible to accomplish with regular CGI.
- Servlets can talk directly to the Web server, whereas regular CGI programs cannot, at least not without using a server-specific API.
- Communicating with the Web server makes it easier to translate relative URLs into concrete path names, for instance.
- Multiple servlets can also share data, making it easy to implement database connection pooling and similar resource-sharing optimizations.
- Servlets can also maintain information from request to request, simplifying techniques like session tracking and caching of previous computations.

4. Portable

- Servlets are written in the Java programming language and follow a standard API.
- *Servlets are supported directly or by a plugin on virtually every major Web server.*
- *Consequently*, servlets written for, say, Macromedia JRun can run virtually unchanged on
 - Apache Tomcat,
 - Microsoft Internet Information Server (with a separate plugin),
 - IBM WebSphere
 - iPlanet Enterprise Server,
 - Oracle9i AS,
 - StarNine WebStar.
- They are part of the Java 2 Platform, Enterprise Edition (J2EE), so industry support for servlets is becoming even more pervasive.

5. Inexpensive

- A number of free or very inexpensive Web servers are good for development use or deployment of low- or medium-volume Web sites.
- Thus, with servlets and JSP you can start with a free or inexpensive server and migrate to more expensive servers with high-performance capabilities or advanced administration utilities only after your project meets initial success.
- This is in contrast to many of the other CGI alternatives, which require a significant initial investment for the purchase of a proprietary package

6. Secure

- One of the main sources of vulnerabilities in traditional CGI stems from the fact that the programs are often executed by general-purpose operating system shells.
- So, the CGI programmer must be careful to filter out characters such as backquotes and semicolons that are treated specially by the shell. Implementing this precaution is harder than one might think, and weaknesses stemming from this problem are constantly being uncovered in widely used CGI libraries.
- A second source of problems is the fact that some CGI programs are processed by languages that do not automatically check array or string bounds. For example, in C and C++ it is perfectly legal to allocate a 100-element array and then write into the 999th “element,” which is really some random part of program memory.
- So, *programmers who forget to perform this check open up their system to deliberate or accidental buffer overflow attacks.*
- Servlets suffer from neither of these problems. Even if a servlet executes a system call (e.g., with `Runtime.exec` or `JNI`) to invoke a program on the local operating system, it does not use a shell to do so. And, of course, array bounds checking and other memory protection features are a central part of the Java programming language.

7. Mainstream

- There are a lot of good technologies out there.
- But if vendors don't support them and developers don't know how to use them, what good are they?
- Servlet and JSP technology is supported by servers from Apache, Oracle, IBM, Sybase, BEA, Macromedia, Caucho, Sun/iPlanet, New Atlanta, ATG, Fujitsu, Lutris, Silverstream, the World Wide Web Consortium (W3C), and many others. Several low-cost plugins add support to Microsoft IIS and Zeus as well.
- They run on Windows, Unix/Linux, MacOS, VMS, and IBM mainframe operating systems.
- They are arguably the most popular choice for developing medium to large Web applications.
- They are used by the airline industry (most United Airlines and Delta Airlines Web sites), e-commerce (ofoto.com), online banking (First USA Bank, CitiBank), Web search engines/portals (excite.com), large financial sites (American Century Investments), and hundreds of other sites that you visit every day.

A Servlet's Job

Servlets are Java programs that run on Web or application servers, acting as a middle layer between requests coming from Web browsers or other HTTP clients and databases or applications on the HTTP server. Their job is to perform the following tasks, as illustrated in Figure 1-1.

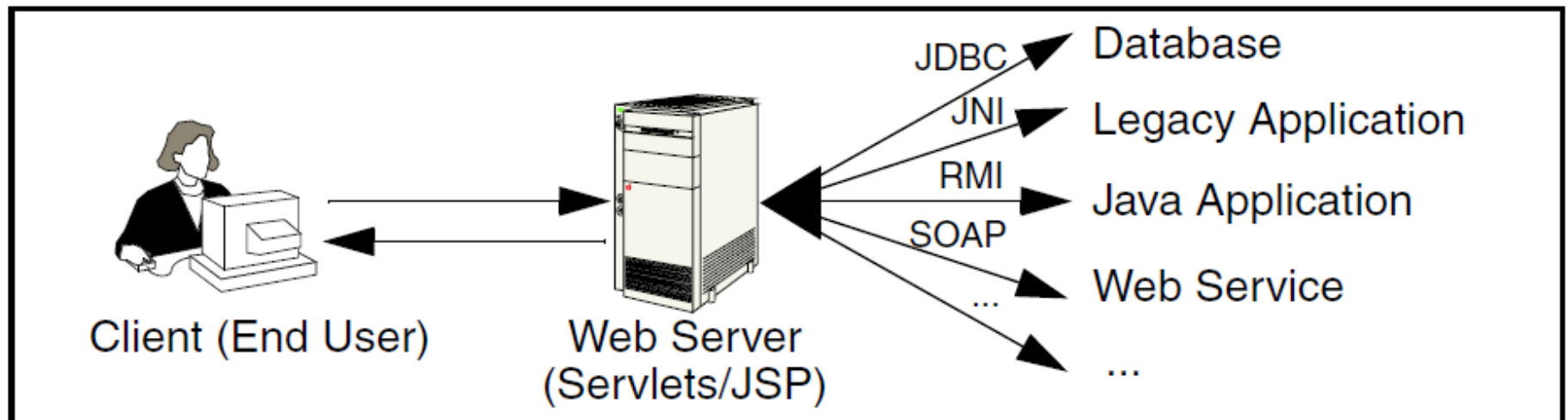


Figure 1-1 The role of Web middleware.

1. Read the explicit data sent by the client.

- The end user normally enters this data in an HTML form on a Web page.

2. Read the implicit HTTP request data sent by the browser.

- Figure 1–1 shows a single arrow going from the client to the Web server (the layer where servlets and JSP execute), but there are really *two varieties of data*:
 - *the explicit data that the end user enters in a form and*
 - *the behind-the-scenes HTTP information.*
- Both varieties are critical.
- The HTTP information includes cookies, information about media types and compression schemes the browser understands, and so forth; it is discussed later.

3. Generate the results.

- This process may require talking to a database, executing an RMI or EJB call, invoking a Web service, or computing the response directly.
- Your real data may be in a relational database.
- Fine. But your database probably doesn't speak HTTP or return results in HTML, so the Web browser can't talk directly to the database.
- Even if it could, for security reasons, you probably would not want it to.
- The same argument applies to most other applications. You need the Web middle layer to extract the incoming data from the HTTP stream, talk to the application, and embed the results inside a document.

4. Send the explicit data (i.e., the document) to the client.

- This document can be sent in a variety of formats, including text (HTML or XML), PDF, binary (GIF images), or even a compressed format like gzip that is layered on top of some other underlying format.
- But, HTML is by far the most common format, so an important servlet/JSP task is to wrap the results inside of HTML.

5. Send the implicit HTTP response data.

- Figure 1–1 shows a single arrow going from the Web middle layer (the servlet or JSP page) to the client.
- But, there are really *two varieties of* data sent:
 - the document itself and
 - the behind-the-scenes HTTP information.
- Again, both varieties are critical to effective development.
- Sending HTTP response data involves telling the browser or other client what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.
- *These tasks are discussed in week 4*

Why Build Web Pages Dynamically?

- **The Web page is based on data sent by the client.**
For instance, the results page from search engines and order-confirmation pages at online stores are specific to particular user requests. You don't know what to display until you read the data that the user submits. Just remember that the user submits two kinds of data: explicit (i.e., HTML form data) and implicit (i.e., HTTP request headers). Either kind of input can be used to build the output page. In particular, it is quite common to build a user-specific page based on a cookie value.
- **The Web page is derived from data that changes frequently.**
If the page changes for every request, then you certainly need to build the response at request time. If it changes only periodically, however, you could do it two ways: you could periodically build a new Web page on the server (independently of client requests), or you could wait and only build the page when the user requests it. The right approach depends on the situation, but sometimes it is more convenient to do the latter: wait for the user request. For example, a weather report or news headlines site might build the pages dynamically, perhaps returning a previously built page if that page is still up to date.
- **The Web page uses information from corporate databases or other server-side sources.**
If the information is in a database, you need server-side processing even if the client is using dynamic Web content such as an applet. Imagine using an applet by itself for a search engine site:

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.println("<HTML>\n" +
            "<HEAD><TITLE>Hello</TITLE></HEAD>\n" +
            "<BODY BGCOLOR='#FDF5E6'>\n" +
            "<H1>Hello</H1>\n" +
            "</BODY></HTML>");
    }
}
```

A Quick Peek at Servlet Code

Now, this is hardly the time to delve into the depths of servlet syntax. Don't worry, you'll get plenty of that throughout the book. But it is worthwhile to take a quick look at a simple servlet, just to get a feel for the basic level of complexity.

Listing 1.1 shows a simple servlet that outputs a small HTML page to the client. Figure 1–2 shows the result.

The code is explained in detail in Chapter 3 (Servlet Basics), but for now, just notice four points:

- **It is regular Java code.** There are new APIs, but no new syntax.
- **It has unfamiliar import statements.** The servlet and JSP APIs are not part of the Java 2 Platform, Standard Edition (J2SE); they are a separate specification (and are also part of the Java 2 Platform, Enterprise Edition—J2EE).
- **It extends a standard class (`HttpServlet`).** Servlets provide a rich infrastructure for dealing with HTTP.
- **It overrides the `doGet` method.** Servlets have different methods to respond to different types of HTTP commands.

The Servlet Life Cycle

- Only a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate.
- When the servlet is first created, its init method is invoked, so init is where you put one-time setup code. After this, each user request results in a thread that calls the service method of the previously created instance.
- Multiple concurrent requests normally result in multiple threads calling service simultaneously, although your servlet can implement a special interface (SingleThreadModel) that stipulates that only a single thread is permitted to run at any one time.
- The service method then calls doGet, doPost, or another doXxx *method*, depending on the type of HTTP request it received.
- Finally, if the server decides to unload a servlet, it first calls the servlet's destroy method.

The service Method

- Each time the server receives a request for a servlet, the server spawns a new thread and calls service.
- The service method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc., as appropriate.
- A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified.
- A POST request results from an HTML form that specifically lists POST as the METHOD.
- Other HTTP requests are generated only by custom clients.
- 99% of the time, you only care about GET or POST requests, so you override doGet and/or doPost.

Handling both POST and GET requests identically

Now, if you have a servlet that needs to handle both POST and GET requests identically, you may be tempted to override `service` directly rather than implementing both `doGet` and `doPost`. This is not a good idea. Instead, just have `doPost` call `doGet` (or vice versa), as below.

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet code
}

public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
```

Although this approach takes a couple of extra lines of code, it has several advantages over directly overriding `service`. First, you can later add support for other HTTP request methods by adding `doPut`, `doTrace`, etc., perhaps in a subclass. Overriding `service` directly precludes this possibility. Second, you can add support for modification dates by adding a `getLastModified` method

The `init` Method

Most of the time, your servlets deal only with per-request data, and `doGet` or `doPost` are the only life-cycle methods you need. Occasionally, however, you want to perform complex setup tasks when the servlet is first loaded, but not repeat those tasks for each request. The `init` method is designed for this case; it is called when the servlet is first created, and *not* called again for each user request. So, it is used for one-time initializations, just as with the `init` method of applets. The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started

The `init` method definition looks like this:

```
public void init() throws ServletException {  
    // Initialization code...  
}
```

The `init` method performs two varieties of initializations: general initializations and initializations controlled by initialization parameters.

The destroy Method

- The server may decide to remove a previously loaded servlet instance, perhaps because it is explicitly asked to do so by the server administrator or perhaps because the servlet is idle for a long time.
- Before it does, however, it calls the servlet's destroy method.
- This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.
- Be aware, however, that it is possible for the Web server to crash (remember those Massachusetts power outages?). So, don't count on destroy as the *only mechanism for saving state to disk*.
- *If your servlet performs* activities like counting hits or accumulating lists of cookie values that indicate special access, you should also proactively write the data to disk periodically.

The SingleThreadModel Interface

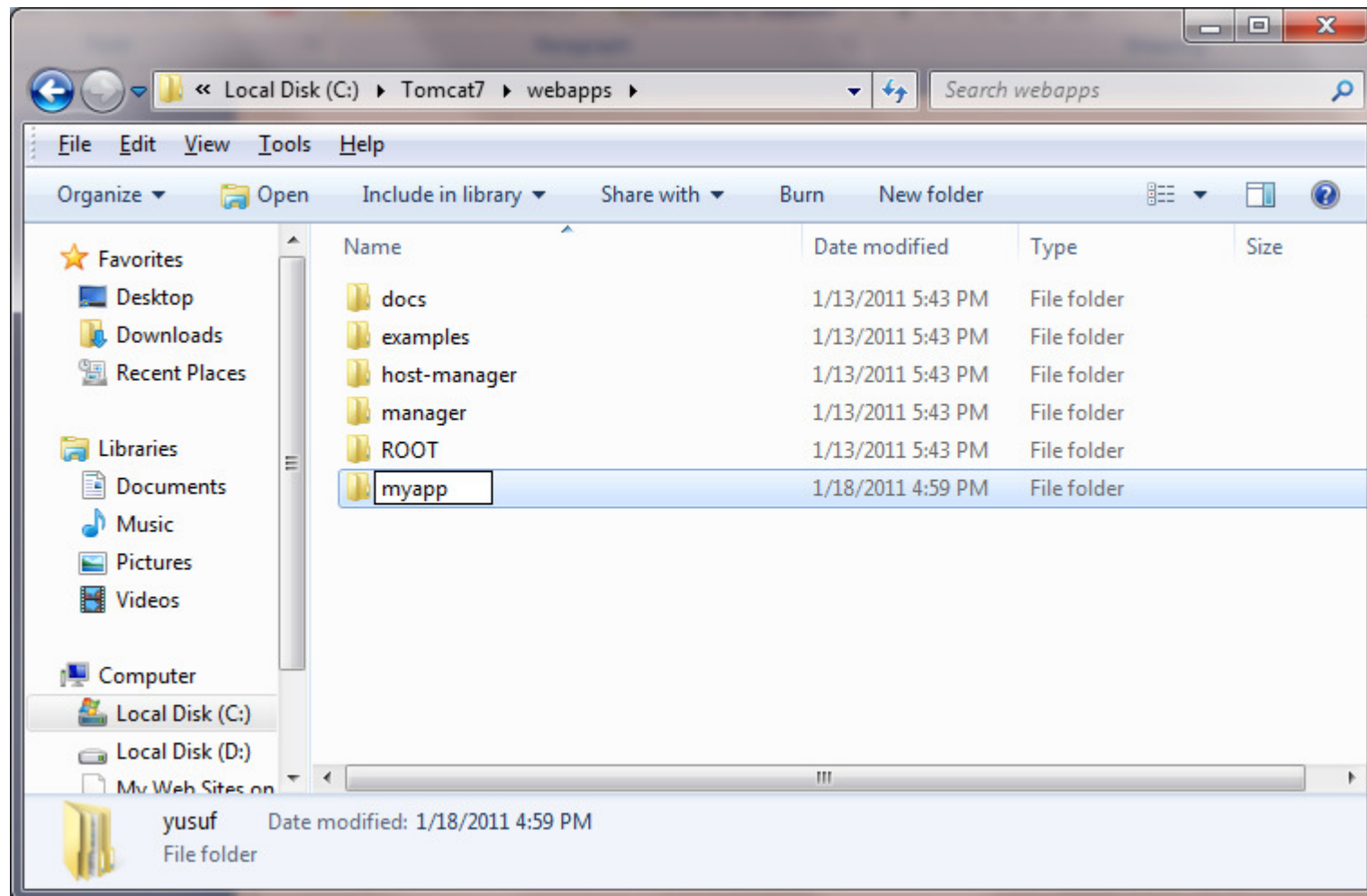
- Normally, the system makes a single instance of your servlet and then creates a new thread for each user request. This means that if a new request comes in while a previous request is still executing, multiple threads can concurrently be accessing the same servlet object.
- Consequently, your doGet and doPost methods must be careful to synchronize access to fields and other shared data (if any) since multiple threads may access the data simultaneously.
- Note that local variables are not shared by multiple threads, and thus need no special protection. In principle, you can prevent multithreaded access by having your servlet implement the SingleThreadModel interface, as below.

```
public class YourServlet extends HttpServlet implements SingleThreadModel
{ ... }
```

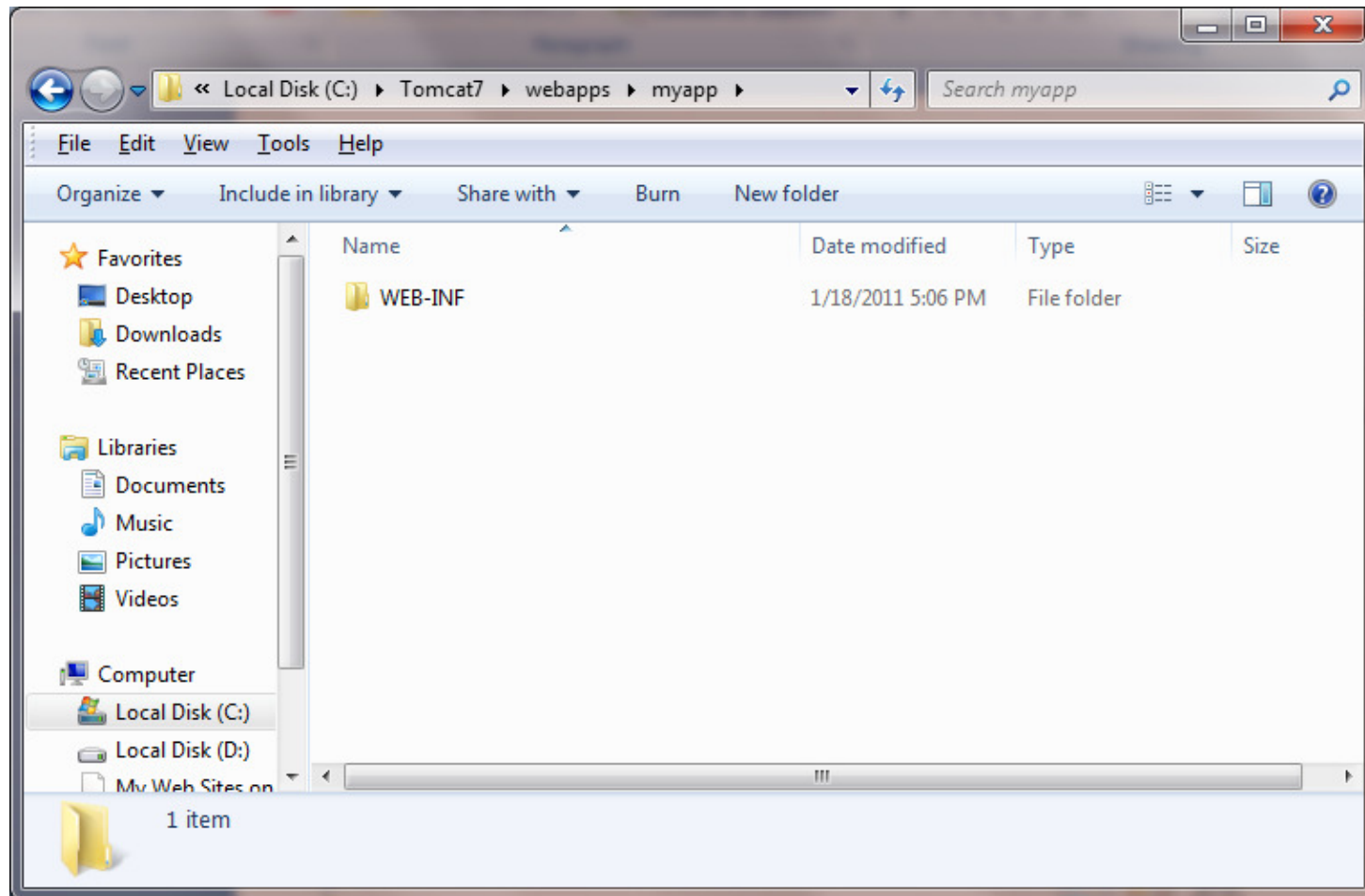
- If you implement this interface, the system guarantees that there is never more than one request thread accessing a single instance of your servlet.
- Although SingleThreadModel prevents concurrent access in principle, in practice there are two reasons why it is usually a poor choice.
 1. *First, synchronous access to your servlets can significantly hurt performance (latency) if your servlet is accessed frequently.*
 2. *The second problem with SingleThreadModel stems from the fact that the specification permits servers to use pools of instances instead of queueing up the requests to a single instance. As long as each instance handles only one request at a time, the pool-of-instances approach satisfies the requirements of the specification. But, it is a bad idea.*

In-Class Exercise

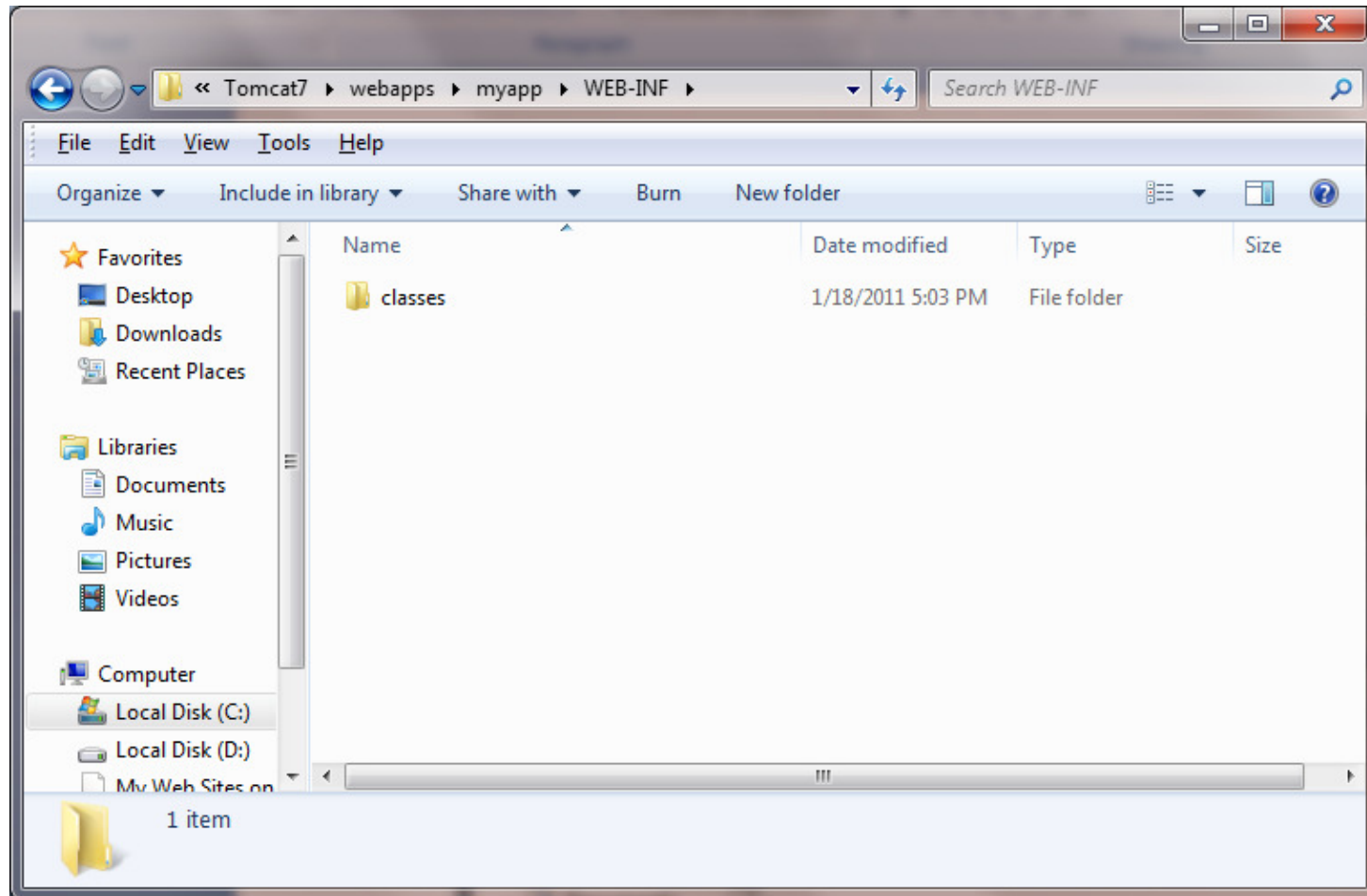
1. Create a new app under C:\yourTomcatInstallation\webapps



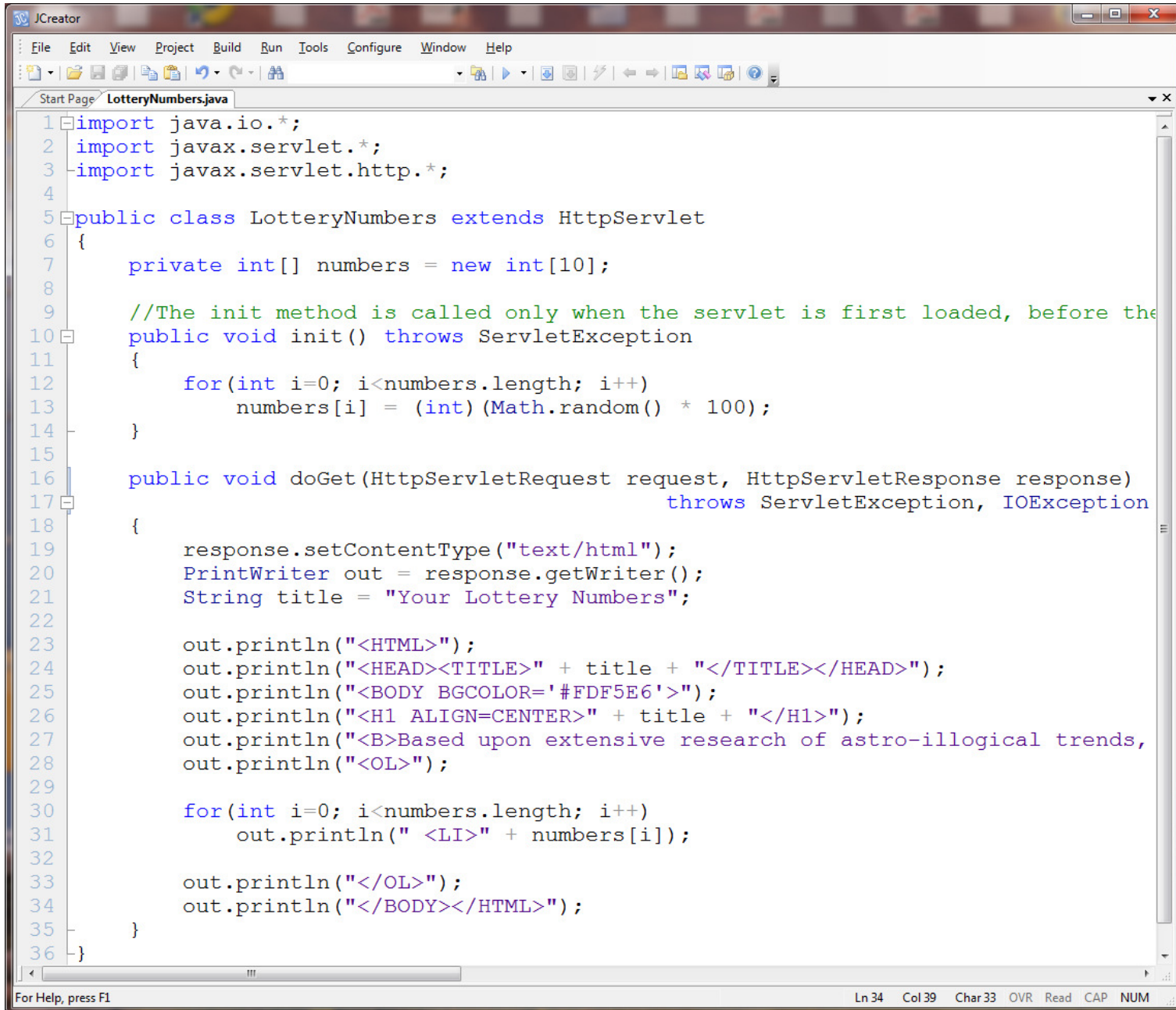
2. Create the WEB-INF directory



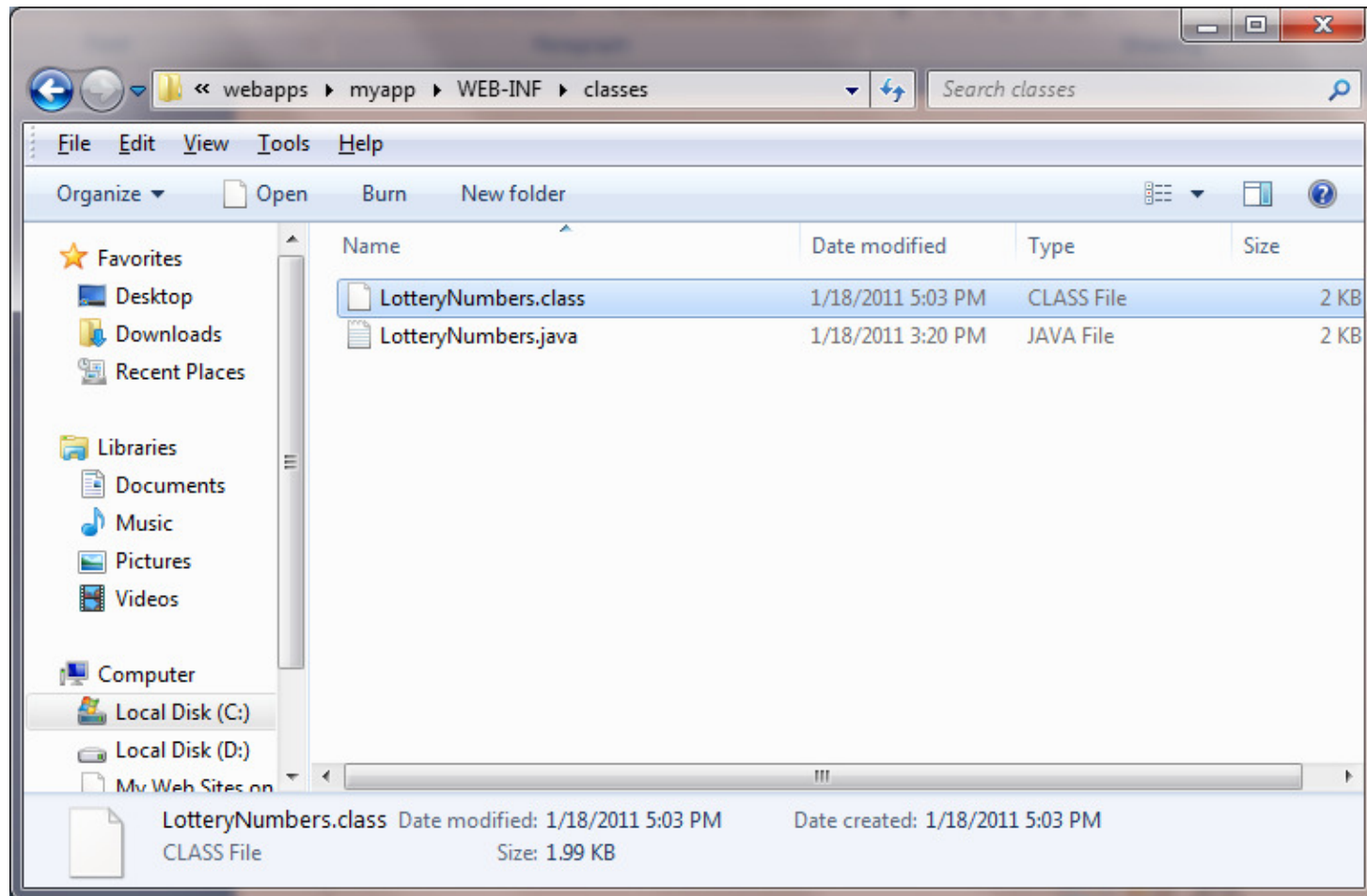
3. Create classes directory under WEB-INF



4. Create and Compile Your Servlet

The image shows a screenshot of the JCreator IDE. The main window displays the source code for a Java Servlet named 'LotteryNumbers.java'. The code includes imports for java.io.*, javax.servlet.*, and javax.servlet.http.*. It defines a public class 'LotteryNumbers' that extends 'HttpServlet'. Inside the class, there is a private integer array 'numbers' of size 10. An 'init()' method is implemented, which uses a for loop to populate the 'numbers' array with random values between 0 and 100. The 'doGet()' method is also implemented, which sets the content type to 'text/html', gets a 'PrintWriter' from the 'response', and writes an HTML document. The HTML document has a title 'Your Lottery Numbers' and a body with a background color of '#FDF5E6'. It contains an h1 heading 'Based upon extensive research of astro-illogical trends,' followed by an ordered list of the numbers in the 'numbers' array. The status bar at the bottom indicates 'Ln 34 Col 39 Char 33 OVR Read CAP NUM'.

5. Compiled servlet needs to be under the classes directory

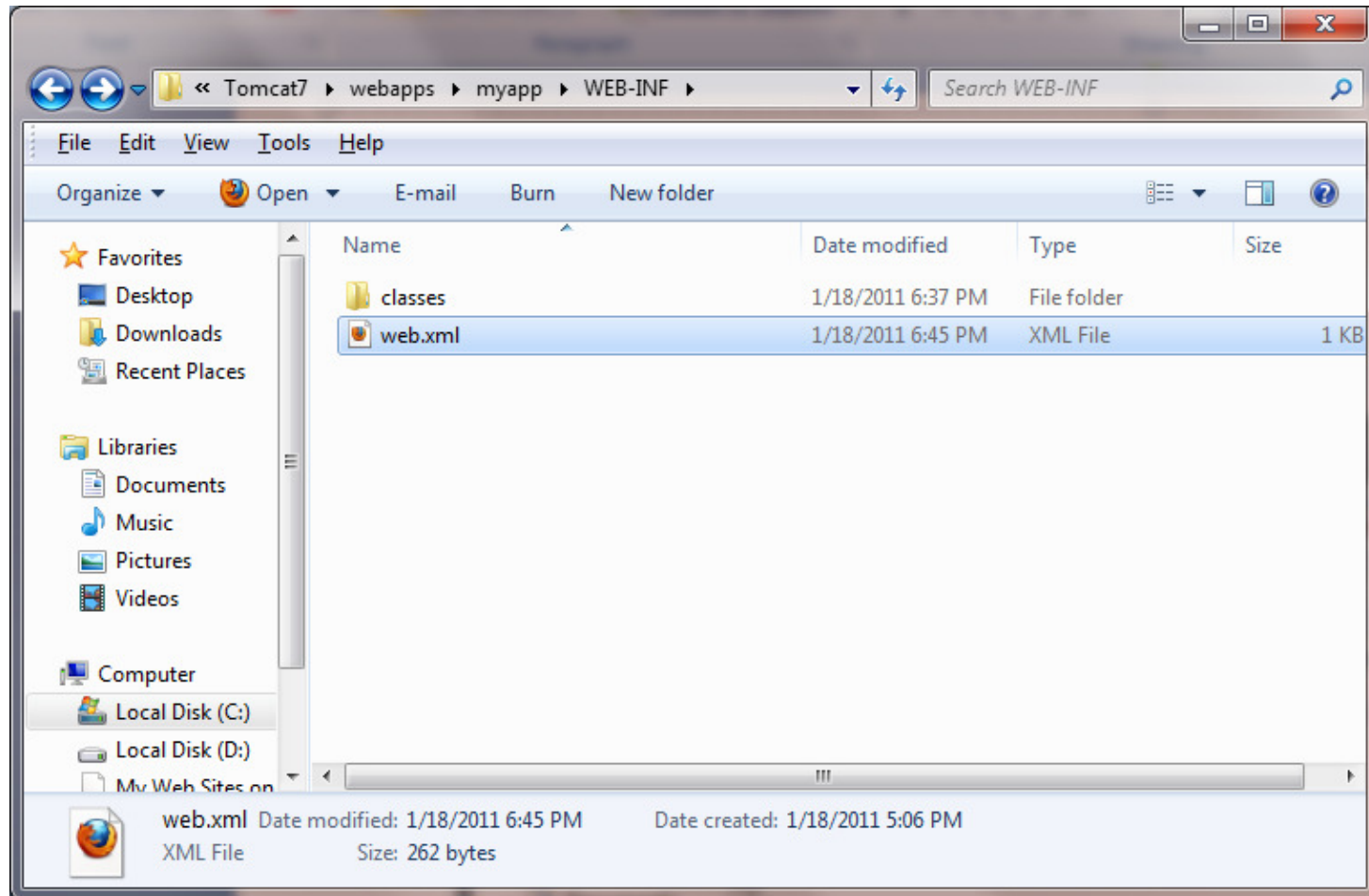


6. Create deployment descriptor web.xml

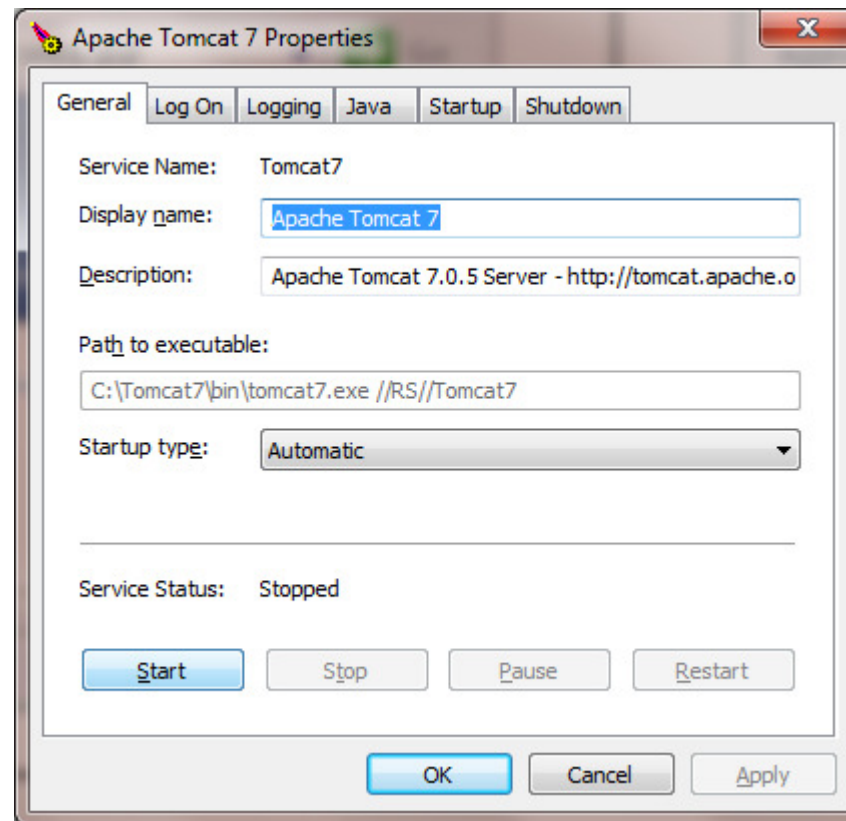
```
<web-app>
  <servlet>
    <servlet-name>lottery</servlet-name>
    <servlet-class>LotteryNumbers</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>lottery</servlet-name>
    <url-pattern>/lottery.do</url-pattern>
  </servlet-mapping>
</web-app>
```

7. Save the web.xml under WEB-INF folder



8. Restart Tomcat



9. Run your web application

