Servlet – Basic Questions

Constructor in Servlets

http://javarevisited.blogspot.in/2015/02/constructor-vs-init-method-in-servlet.html

Can we define Constructor in Servlet?

Short answer of this question, Yes, Servlet implementation classes can have constructor but they should be using init() method to initialize Servlet because of two reasons, first you cannot declare constructors on interface in Java, which means you cannot enforce this requirement to any class which implements Servlet interface and second, Servlet require ServletConfig object for initialization which is created by container as it also has reference of ServletContext object, which is also created by container.

`Servlet` is an interface defined in `javax.servlet` package and `HttpServlet` is a class and like any other class in Java they can have constructor, but you cannot declare constructor inside interface in Java. If you don't provide an explicit constructor than compiler will add a default no argument constructor in any Servlet implementation class. Another reason that you should not initialize Servlet using constructor because Servlets are not directly instantiated by Java code, instead container create there instance and keep them in pool. Since containers from web servers like Tomcat and Jetty uses Java Reflection for creating instance of Servlet, presence of no argument constructor is must. So, by any chance if you provide a parametric constructor and forget to write a no argument constructor, web container will not be able to create instance of your Servlet, since there is no default constructor. Remember Java compiler doesn't add default no argument constructor, if there is a parametric constructor present in class. That's why it's not advised to provide constructor in Servlet class. Now let's see some difference between Constructor and init method in Java Servlet

Difference between Constructor and init method in Servlet

In real world application, you better use `init()` method for initialization, because `init()` method receives a `ServletConfig` parameter, which may contain any initialization parameters for that Servlet from web.xml file. Since web.xml provides useful information to web container e.g. name of Servlet to instantiate, ServletConfig instance is used to supply initialization parameter to Servlets. You can configure your Servlet based upon settings provided in `ServletConfig` object e.g. you can also provide environment specific settings e.g. path of temp directory, database connection parameters (by the way for that you should better leverage JNDI connection pool) and any other configuration parameters. You can simply deploy your web application with different settings in web.xml file on each environment. Remember, `init()` method is not chained like constructor, where super class constructor is called before sub class constructor executes, also known as constructor chaining.

http://way2java.com/servlets/java-made-clear-can-we-write-constructor-in-servlet-example/

Reasons for not preferring to write a Servlet Constructor

1. **init()** is one of the life cycle methods called implicitly after container creates a Servlet object.
2. The overloaded **init()** method takes an object of **ServletConfig** as parameter. This ServletConfig object represents the specific servlet in which **init()** it written. You can use **super.init(config)** to call super class init() method of **HttpServlet**. Creation of an object of ServletConfig, calling init() method, passing ServletConfig object are done automatically by the container.
   Constructor does not have this facility of accessing ServletConfig object. Or, constructor does not know what a ServletConfig is.
3. You can throw **ServletException** with init() but not with a constructor.
4. You cannot have parameterized constructor in objects created dynamically by some software as in Servlets.
5. If you implement Servlet interface to write a Servlet (instead of extending HttpServlet), you cannot have constructor (in interface).
6. If you write a constructor to open some database handles, you must have your own destructor also. Java does not support destructors. **destroy()** is one of the life cycle methods called automatically where you can close the handles. Moreover, API is consistent with init() and destroy() (like in Applets).
7. To create a Servlet object, anyhow, Servlet requires a constructor. This you may create or container create.
8. init() is equivalent to constructor. So better avoid constructor and use init() to get the advantage of life cycle management.

Let us dig a little bit.

9. **Constructor is called before init() method.** Servlet's initialization data, <init-param>, is available with init() method by the support of ServletConfig passed as parameter. If the constructor requires any initialization information to do with the Servlet, it cannot have and may throw NullPointerException sometimes by the container.
10. As long as you do not require any initialization that do not require ServletConfig or ServletContext, you can use as well constructor also.

According to the servlets (2.3) specification, the servlets are instantiated by the servlet engine by invoking the no-arg constructor.

The init() method is typically used to perform servlet initialization--creating or loading objects that are used by the servlet in the handling of its requests. **Why not use a constructor instead?** Well, in JDK 1.0 (for which servlets were originally written), constructors for dynamically loaded Java classes (such as servlets) couldn't accept arguments.

So, in order to provide a new servlet any information about itself and its environment, a server had to call a servlet's init() method and pass along an object that implements the ServletConfig interface.

Also, Java doesn't allow interfaces to declare constructors. This means that the javax.servlet.Servlet interface cannot declare a constructor that accepts a ServletConfig parameter. It has to declare another method, like init(). It's still possible, of course, for you to define constructors for your servlets, but in the constructor you don't have access to the ServletConfig object or the ability to throw a ServletException.

Important Http Methods

HEAD Method

The HEAD method is functionally similar to GET, except that the server replies with a response line and headers, but no entity-body. The HEAD method is identical to the GET method but it does not return a message body. The Http HEAD method is used to retrieve the meta-information about a resource.

The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response. The metainformation contained in the HTTP headers in response to a HEAD request SHOULD be identical to the information sent in response to a GET request. This method can be used for obtaining metainformation about the entity implied by the request without transferring the entity-body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.

OPTIONS Method

The OPTIONS method represents a request for information about the communication options available on the request/response chain identified by the Request-URI. This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval. Responses to this method are not cacheable.

The **HTTP OPTIONS method** is used to describe the communication options for the target resource. The client can specify a specific URL for the OPTIONS method, or an asterisk (*) to refer to the entire server. **It is always Idempotent**.

Example

OPTIONS http://example.org/

Request body : *

Response

```
HTTP/1.1 200 OK
Allow: OPTIONS, GET, HEAD, POST
Cache-Control: max-age=604800
Date: Thu, 13 Oct 2016 11:45:00 GMT
Expires: Thu, 20 Oct 2016 11:45:00 GMT
Server: EOS (lax004/2813)
x-ec-custom-error: 1
Content-Length: 0
```

Using HttpClient

```
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
System.out.println(conn.getRequestMethod()); // GET
conn.setRequestMethod("OPTIONS");
System.out.println(conn.getHeaderField("Allow"));
```

TRACE Method

TRACE: This method simply echoes back to the client whatever string has been sent to the server, and is used mainly for debugging purposes. This method, originally assumed harmless, can be used to mount an attack known as Cross Site Tracing, which has been discovered by Jeremiah Grossman.

User-Agent : In http request "User-Agent" describes the type of client. It means "User-Agent" will tell you whether request is coming from computer browser or mobile browser etc.

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
 if(request.getHeader("User-Agent").indexOf("Mobile") != -1) {
   //you're in mobile land
 } else {
   //nope, this is probably a desktop
 }
}
```

If-Modified-Since : The "If-Modified-Since" HTTP header is sent from an agent (browser/bot) to the web server in order to know if the requested page has been changed since its last visit. The server responds with a 200 code (*Ok*) if the page has been modified or with a 304 code (*Not modified*) if the page has not been modified.

Syntax

```
If-Modified-Since: <day-name>, <day> <month> <year> <hour>:<minute>:<second> GMT
```

```
GET /index.html HTTP/1.0
User-Agent: Mozilla/5.0
From: something.somewhere.net
Accept: text/html,text/plain,application/*
Host: www.example.com
If-Modified-Since: Wed, 19 Oct 2005 10:50:00 GMT
```

Cookies : Cookies are small text passed between client and server.

Difference between sendRedirect and forward in Servlet

In case of Response.sendRedirect(""), you have to specify the complete url or servlet path so that the web page displays the redirected web page in the browser. Any user can also see the redirected page url in the browser. In other words, the servlet sends a message telling the browser to get the resource from elsewhere.

In case of Request.getRequestDispatcher("/forwardedServletPath").forward(request,response), the redirection happens inside the server. The client, browser or user does not know what happens in side the server.

The complete code details is given below for sendRedirect and forward.

Response.sendRedirect()

Java Code

SendRedirectServlet.java

```
package com.ddlab.rnd.servlets;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class SendRedirectServlet extends HttpServlet {

	private static final long serialVersionUID = 7303819158566415 77L;

	@Override
	protected void doGet(HttpServletRequest request, HttpServletResponse response)
			throws ServletException, IOException {
		response.setContentType("text/html");
		response.sendRedirect("http://localhost:8090/basic-servlets/redirectedServlet");
	}
}
```

RedirectedServlet.java

```
package com.ddlab.rnd.servlets;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class RedirectedServlet extends HttpServlet {
	private static final long serialVersionUID = 3597561920098701037L;

	@Override
	protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
		PrintWriter pw = response.getWriter();
		pw.println("<h1>This is the redirected page</h1>");
	}
}
```

We should keep in mind a couple of important points about the sendRedirect()method. We cannot call this method if the response is committed—that is, if the response header has already been sent to the browser. If we do, the method will throw a java.lang.IllegalStateException. For example, the following code will generate an IllegalStateException:

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
{
	PrintWriter pw = res.getWriter();
	pw.println("<html><body>Hello World!</body></html>");
	pw.flush();
	res.sendRedirect("http://www.cnn.com");
}
```

Request.getRequestDispatcher(“”).forward(request,response)

ForwardServlet.java

```
package com.ddlab.rnd.servlets;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ForwardServlet extends HttpServlet {

	private static final long serialVersionUID = -2671410483116378502L;

	@Override
	protected void doGet(HttpServletRequest request, HttpServletResponse response)
			throws ServletException, IOException {
		response.setContentType("text/html");
		request.getRequestDispatcher("/forwardedServlet").forward(request, response);

	}
}
```

ForwardedServlet.java

```
package com.ddlab.rnd.servlets;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ForwardedServlet extends HttpServlet {

	private static final long serialVersionUID = 1493313193754039559L;

	@Override
	protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
		PrintWriter pw = response.getWriter();
		pw.println("<h1>This is the forwarded response</h1>");
	}
}
```

Web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         version="2.5"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <display-name>Archetype Created Web Application Basic Servlet</display-name>

  <servlet>
      <servlet-name>sendredirect</servlet-name>
      <servlet-class>com.ddlab.rnd.servlets.SendRedirectServlet</servlet-class>
  </servlet>
  <servlet-mapping>
      <servlet-name>sendredirect</servlet-name>
      <url-pattern>/sendredirect</url-pattern>
  </servlet-mapping>

  <servlet>
      <servlet-name>redirectedServlet</servlet-name>
      <servlet-class>com.ddlab.rnd.servlets.RedirectedServlet</servlet-class>
  </servlet>
  <servlet-mapping>
      <servlet-name>redirectedServlet</servlet-name>
      <url-pattern>/redirectedServlet</url-pattern>
  </servlet-mapping>

  <servlet>
      <servlet-name>forwardServlet</servlet-name>
      <servlet-class>com.ddlab.rnd.servlets.ForwardServlet</servlet-class>
  </servlet>
  <servlet-mapping>
      <servlet-name>forwardServlet</servlet-name>
      <url-pattern>/forwardServlet</url-pattern>
  </servlet-mapping>

  <servlet>
      <servlet-name>forwardedServlet</servlet-name>
      <servlet-class>com.ddlab.rnd.servlets.ForwardedServlet</servlet-class>
  </servlet>
  <servlet-mapping>
      <servlet-name>forwardedServlet</servlet-name>
      <url-pattern>/forwardedServlet</url-pattern>
  </servlet-mapping>

</web-app>
```

URL to test : http://localhost:8090/basic-servlets/sendredirect

Servlet A decides it needs to make use of Servlet B n a separate web application, it can do using RequestDispatcher mechanism.

In case of forwarding, the request information (parameters and attributes) are preserved where as in case of sendRedirect it is again a new request, which user has to initiate. In this case we can say that forwarding is better than sendRedirect.

When you try to call the forward() or include() methods when a response has been committed to the client, you will get an IllegalStateException.

SingleThreadModel interface is a marker interface and it is a sign to the servlet container to ensure that any one instance of the servlet has only one request accessing it at a time. However SingleThreadModel has been deprecated in 2.4 version of servlet specification.

Servlet init() method

The init() method must complete successfully before the servlet container will allow any request to be processed by the servlet. The init method is a convenience method in GenericServlet class. The servlet container actually calls ***Servlet.inti(ServletConfig config),*** passing in a servlet configuration object. The init() method will be invoked once in servlet life cycle. It means if a user makes a web request, the init method will be invoked and upon subsequent request, init() method will not be invoked and other methods like either service() or doXXX() methods will be invoked. A simple example is given below.

```
package com.ddlab.rnd.servlets;
import java.io.IOException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class InitServlet extends HttpServlet {

	private static final long serialVersionUID = 3449078750941207887L;

	@Override
	public void init(ServletConfig config) throws ServletException {
		String param1 = config.getInitParameter("param1");
		String param2 = config.getInitParameter("param2");
		System.out.println("Param1------>"+param1);
		System.out.println("Param2------>"+param2);
	}

	@Override
	protected void service(HttpServletRequest request, HttpServletResponse response)
			throws ServletException, IOException {
		System.out.println("***************** service() ********************");
		super.service(request, response);
	}
}
```

Contents in the web.xml

```
<servlet>
		<servlet-name>initServlet</servlet-name>
		<servlet-class>com.ddlab.rnd.servlets.InitServlet</servlet-class>
		<init-param>
			<param-name>param1</param-name>
			<param-value>parameter value 1</param-value>
		</init-param>
		<init-param>
			<param-name>param2</param-name>
			<param-value>parameter value 2</param-value>
		</init-param>
	</servlet>
	<servlet-mapping>
		<servlet-name>initServlet</servlet-name>
		<url-pattern>/initServlet</url-pattern>
	</servlet-mapping>
```

Servlet service() method

This method is called by the servlet container in response to a client request. There is an overloaded version of the method in HttpServlet that dispatches to the appropriate doXXX() methods. The overloaded version accepts an HttpServletRequest and HttpServletResponse as parameters.

If you override service method, any request using any of the http methods like GET, POST, PUT, TRACE will be received by service() method. Service method is a generic method which can receive any kind of http request whereas doXXX() method will receive only specific request with http method like Get, POST, Trace etc.

The source code of service() method is given below.

```
String method = req.getMethod();
if (method.equals(METHOD_GET)) {

---

} else if (method.equals(METHOD_HEAD)) {
            long lastModified = getLastModified(req);
            maybeSetLastModified(resp, lastModified);
            doHead(req, resp);

        } else if (method.equals(METHOD_POST)) {
            doPost(req, resp);

        } else if (method.equals(METHOD_PUT)) {
```

So service method internally calls the appropriate doXXX() methods.

HttpServlet is an abstract class which extends GenericServlet class.

```
public abstract class HttpServlet extends GenericServlet
```

If both the methods service() and doGet() methods are there in a servlet, which method will be invoked upon a web request ? Ans : It is always service() method, as service() method is always a generic method for all kinds of web request to a servlet.

Servlet destroy() method

When servlet instance is taken out of service, the destroy() method is called. It is an opportunity to reclaim all the expensive resources that may have been setup in init() method. The destroy() method is never called if init() fails.

<load-on-startup> in web.xml

- A servlet with a lower number will be loaded before a servlet with a higher number.
- Two or more servlets can have the same number. If both the servlets have the same number, it is upto the container which one to load first. <Load-on-startup> can be negative, again it is up to the container about how to load.

How to handle error in servlet

One way to handle the error is given below. Use **Response.sendError()** and **Response.setStatus()**.

```
@Override
	protected void service(HttpServletRequest request, HttpServletResponse response)
			throws ServletException, IOException {

		//Imagine that there is an error here
		response.sendError(HttpServletResponse.SC_NOT_FOUND, "My Server error message");
//		response.setStatus(sc, sm); // Deprecated
		response.setStatus(HttpServletResponse.SC_NOT_FOUND);
	}
```

Another way to handle is, use a custom error page to handle specific kinds of exception. The code is given below.

Inside the servlet, the code is given below.

```
@Override
	protected void service(HttpServletRequest request, HttpServletResponse response)
			throws ServletException, IOException {

		throw new ServletException("My custom servlet exception");
	}
```

The code snippet for web.xml is given below.

```
<error-page>
	<exception-type>javax.servlet.ServletException</exception-type>
	<location>/customErrorPage.html</location>
</error-page>
```

```
In this case, any servlet having ServletException, this html page will come up in the browser.
```

ServletContext

ServletContext represents the whole web application itself.

Parameters for servlet context in web.xml

```
<web-app>
	<context-param>
		<param-name>param1</param-name>
		<param-value>parameter value 1</param-value>
	</context-param>
	<context-param>
		<param-name>secretParamFile</param-name>
		<param-value>/WEB-INF/xml/somefile.xml</param-value>
	</context-param>
</web-app>
```

How to access servlet context parameter details
ServletContext context = getServletContext();
String value = context.getInitParameterName("paramName");

On startup of the container, a servlet context is created for each web application.

Scope

Request scope is represented by an instance of a request.

The request attributes are thread-safe in a web application but session attributes are not officially.

Servlet Filter

Servlet filter has its own life cycle using the following methods.

Init(FilterConfig config)

doFilter(HttpServletRequest request, HttpServletResponse response)

destroy()

Should one call .close() on HttpServletResponse.getOutputStream()/.getWriter()?

Normally you should not close the stream. The servlet container will automatically close the stream after the servlet is finished running as part of the servlet request life-cycle. If it's not you that's opening the stream, you should not close it. The stream is opened by the container so the responsibility for closing lies with it.

Session Details

You can handle session timeout in the following manner.

```
<web-app>
        <session-config>
                <session-timeout>60<session-timeout> In minutes
        </session-config>
</web-app>
```

The session timeout is in minutes. A value of 0 or negative value indicates that the session never expires.

Cookies

The preferred way for a web application to pass session IDs back to client is via cookie in the response.

URL Rewriting

To do

The **contextInitialized**() method of a **ServletContextListner** is a great place to read in parameters from initialization files that are fundamental to the operation of your web application. It is a better alternative than relying on the **init**() method in a servlet that loads on startup. Although you can configure your servlet to be the first one that loads n the application, that's vulnerable to later configuration changes. But you can guarantee that the **contextInitialized**() method will be the first piece of your code run on startup of the web application.

Big Picture of Servlet Container

The browser sends requests to the web server. If the target is an HTML file, the server handles it directly. If the target is a servlet, the server delegates the request to the servlet container, which in turn forwards it to the servlet.

Web server consists of two major modules, one is web server and other is servlet container. Servlet container runs as a plugin inside the web server.

The Model 1 architecture

In Model 1 architecture, the target of every request is a JSP page. This page is completely responsible for doing all the tasks required for fulfilling the request. This includes authenticating the client, using JavaBeans to access the data, managing the state of the user, and so forth.

The Model 2 architecture

This architecture follows the Model-View-Controller (MVC) design pattern (which we will discuss in chapter 18, "Design patterns."). In this architecture, the targets of all the requests are servlets that act as the controller for the application. They analyze the request and collect the data required to generate a response into JavaBeans objects,
which act as the model for the application. Finally, the controller servlets dispatch the request to JSP pages. These pages use the data stored in the JavaBeans to generate a response.

The biggest advantage of this model is the ease of maintenance that results from the separation of responsibilities. The Controller presents a single point of entry into the application, providing a cleaner means of implementing security and state management; these components can be reused as needed.

Active vs Passive Resources

One way of categorizing web resources is that they are either *passive* or *active*. A resource is passive when it does not have any processing of its own; active objects have their own processing capabilities. For example, when a browser sends a request for www.myserver.com/myfile.html, the web server at myserver.com looks for the myfile.html file, a passive resource, and returns it to the browser. Similarly, when a browser sends a request
for www.myserver.com/reportServlet, the web server at myserver.com forwards the request to reportServlet, an active resource. The servlet generates the HTML text on the fly and gives it to the web server. The web server, in turn, forwards it to the browser. A passive resource is also called a static resource, since its contents do not change with requests. A web application is usually a mixture of active and passive resources.

URI, URL, URN

- *Uniform Resource Identifier*—A URI is a string that identifies any resource. Identifying the resource may not necessarily mean that we can retrieve it. URI is a superset of URL and URN.
- *Uniform Resource Locator*—URIs that specify common Internet protocols such as HTTP, FTP, and mailto are also called URLs. URL is an informal term and is not used in technical specifications.
- *Uniform Resource Name*—A URN is an identifier that uniquely identifies a resource but does not specify how to access the resource. URNs are standardized by official institutions to maintain the uniqueness of a resource.

Here are some examples:

• `files/sales/report.html` is a URI, because it identifies some resource. However, it is not a URL because it does not specify how to retrieve the resource. It is not a URN either, because it does not identify the resource uniquely.

• `http://www.manning.com/files/sales/report.html` is a URL because it also specifies how to retrieve the resource.

• `ISBN:1-930110-59-6` is a URN because it uniquely identifies this book, but it is not a URL because it does not indicate how to retrieve the book.

Usage of ServletOutputStream

If we want to send a binary file, for example a JAR file, to the client, we will need an OutputStream instead of a PrintWriter. ServletResponse provides the getOutputStream() method that returns an object of class javax.servlet.ServletOutputStream. An example is given below.

Web.xml snippet

```
<servlet>
        <servlet-name>downloadServlet</servlet-name>
        <servlet-class>com.ddlab.rnd.servlets.JarDownloadServlet</servlet-class>
</servlet>
<servlet-mapping>
        <servlet-name>downloadServlet</servlet-name>
        <url-pattern>/downloadServlet</url-pattern>
</servlet-mapping>
```

JarDownloadServlet.java

```
package com.ddlab.rnd.servlets;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.OutputStream;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class JarDownloadServlet extends HttpServlet {

   @Override
   protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

      response.setContentType("application/jar");

      File file = new File("E:/netbeans-workspace/basic-servlets/sampleJar/abcd.jar");
      byte[] buffer = new byte[(int) file.length()];
      FileInputStream fin = new FileInputStream(file);
      fin.read(buffer);
      OutputStream os = response.getOutputStream();
      os.write(buffer);
      os.flush();

   }
}
```

NOTE
An important point to note about the getWriter() and getOutput-Stream() methods is that you can call only one of them on an instance of ServletResponse. For example, if you have already called the get-Writer() method on a ServletResponse object, you cannot call the getOutputStream() method on the same ServletResponse object. If you do, the getOutputStream() method will throw an Illegal-StateException. You can call the same method multiple times, though.

SERVLET LIFE CYCLE

Before a servlet can service the client requests, a servlet container must take certain steps in order to bring the servlet to a state in which it is ready to service the requests. The first step is loading and instantiating the servlet class; the servlet is now considered to be in the loaded state. The second step is initializing the servlet instance. Once the servlet is in the initialized state, the container can invoke its service() method whenever it receives a request from the client. There may be times when the container

will call the destroy() method on the servlet instance to put it in the destroyed state. Finally, when the servlet container shuts down, it must unload the servlet instance.

Loading and instantiating a servlet

Each web application has its own deployment descriptor file, web.xml, which includes an entry for each of the servlets it uses. An entry specifies the name of the servlet and a Java class name for the servlet. The servlet container creates an instance of the given servlet class using the method **Class.forName(className).newInstance().** However, to do this the servlet class must have a public constructor with no arguments. Typically, we do not define any constructor in the servlet class. We let the Java compiler add the default constructor. At this time, the servlet is *loaded.* It is entirely possible that we will want to initialize the servlet with some data when it is instantiated. It is exactly for this reason that once the container creates the servlet instance, it calls the init(ServletConfig) method on this newly created instance. The ServletConfig object contains all the initialization parameters that we specify in the deployment descriptor of the web application. If you override the init(ServletConfig config) method, you will have to include a call to super.init(config) in the method so that the GenericServlet can store a reference to the config object for future use.

Load-on-Startup

Usually, a servlet container does not initialize the servlets as soon as it starts up. It initializes a servlet when it receives a request for that servlet for the first time. This is called lazy loading. Although this process greatly improves the startup time of the servlet container, it has a drawback. If the servlet performs many tasks at the time of initialization, such as caching static data from a database on initialization, the client that sends the first request will have a poor response time. In many cases, this is unacceptable. The servlet specification defines the <load-on-startup> element, which can be specified in the deployment descriptor to make the servlet container load and initialize the servlet as soon as it starts up. This process of loading a servlet before any request comes in is called preloading, or preinitializing, a servlet.

Mapping URL to Servlet