2

Web Applications

CERTIFICATION OBJECTIVES

- File and Directory Structure
- Deployment Descriptor Elements
- WAR Files

✓ Two-Minute Drill

Q&A Self Test

In Chapter 1 we saw how J2EE gave us a coding framework for our web applications. But J2EE also has plenty to say about how to package and deploy applications—nothing is left to chance! In this chapter you'll see how to design the correct directory structure for a J2EE web app, and which files go where. You'll learn about the "deployment descriptor" file, web.xml, which tells any self-respecting J2EE web container all it needs to know about your web app. Finally, you'll get to bundle up all the files making up your application into a single "web archive" or WAR file, making deployment that much easier.

Do not think there is anything optional about these topics, either in the real world or in the exam! You can't wash your hands of your web app as soon as the code is written—you have to package and deploy with the best of them. The good news is that any decent J2EE IDE does practically all the packaging and deployment as you code. But because you need to understand what you're doing, we'll keep things explicit and hands-on in this chapter.

CERTIFICATION OBJECTIVE

File and Directory Structure (Exam Objective 2.1)

Construct the file and directory structure of a Web Application that may contain (a) static content, (b) JSP pages, (c) servlet classes, (d) the deployment descriptor, (e) tag libraries, (f) JAR files and (g) Java class files; and describe how to protect resource files from HTTP access.

One of the many goals outlined in the J2EE specification is the "portable deployment of J2EE applications into any J2EE product." A logical consequence of this goal is that any one J2EE application should have a structure broadly identical to any other J2EE application. Sun has standards for this. Although a J2EE web container is free to impose its own structure, it rarely makes sense for it to do so. And more to the point, the standard structure is something you are expected to know for the exam.

A Place for Everything and Everything in Its Place

Generally, servers (with J2EE web containers) have a preferred location for web applications—sometimes more than one. You should generally abide by preferred locations, but most servers provide a facility to specify any directory whatsoever as a home for web application contexts. The Tomcat server prefers <TOMCAT

INSTALLATION DIRECTORY>/webapps—you'll see this in action in the chapter exercises.

Whether or not there is a preferred location, each web application needs its own home directory, which generally means a directory immediately beneath the preferred location. All the resources a web application needs go inside this home directory, or subdirectories beneath it. Once the web application is placed there, how can we get at those resources? Typically, we point toward them with a URL, usually (but not exclusively) entered in the address line of a browser, such as

```
http://host:port/webappcontext/resourceIneed
```

Let's look at what different parts of the URL equate to (always assuming that a J2EE web application is the target):

- **host:port**—directly or indirectly, this identifies a running instance of an application server hosting a J2EE web container.
- **webappcontext**—this part of the URL uniquely identifies a particular web application running within the server. This is the "context root," and it identifies the home directory for the web application.
- **resourceIneed**—any resource available in the web application. This could be a simple static web page or a servlet returning complex dynamic content; the request mechanism (resource name in URL) is the same.

In a production environment, the host:port part of the URL rarely points directly to an application server running a web container. More usually, the host:port combination identifies an industrial-strength web server, such as Apache. The web server works out—from the rest of the URL—which requests are appropriate to hand off to the web container.

For web applications running under the same server to be distinguished from one another, the context root must be unique. There's no requirement that the context name match the home directory name, though this is often the case—for one thing, it keeps organization simple, and for another, many deployment tools actively encourage this behavior.

HTTP Accessible Resources

Any files directly within the context root are meant to be available to users of your web application: They are there for the requesting. Here's a nonexhaustive list of resources we might expect to find there:

- Static HTML files
- Dynamic JavaServer Page (JSP) files
- Images
- Media clips
- Stylesheets
- Java applets (and their supporting classes and JARs)

Nothing in the rules says you can't define your own directories off the context root. So if you want a directory to store JPEGs and GIFs, create an "images" directory. If you want another directory to store stylesheet files, create one called "style." The process is no different from defining a typical directory structure for a regular web site.

Special Directories Beneath the Context Root

What do we do, though, with all those files that support the operation of the web application but that have no business being the direct target of a user request? These might include (but are not limited to)

- Servlet class files
- Other class files that support the work of your servlets
- Whole libraries of support code in JAR files
- Configuration files—your own, or those mandated by the servlet specification

The servlet specification strongly recommends that you have a directory called WEB-INF for these files. The directory is called WEB-INF: capitalized WEB, hyphen, capitalized INF. Other variants don't count (Web-Inf, WEB_INF, web-inf—none of these will do). There are a standard set of directories defined within WEB-INF:

- /WEB-INF/classes—for classes that exist as separate Java classes (*not* packaged within JAR files). These might be servlets or other support classes. Of course, because your classes are likely to live in packages, the directory structure should normally reflect that: for example, /WEB-INF/classes/webcert/chapter2/WellLocatedServlet.class.
- /WEB-INF/lib—for JAR files. These can contain anything at all—the main servlets for your application, supporting classes that connect to databases—whatever.
- /WEB-INF itself is the home for an absolutely crucial file called web.xml, the deployment descriptor file. You'll learn much more about this in the "Deployment Descriptor Elements" section of the chapter.

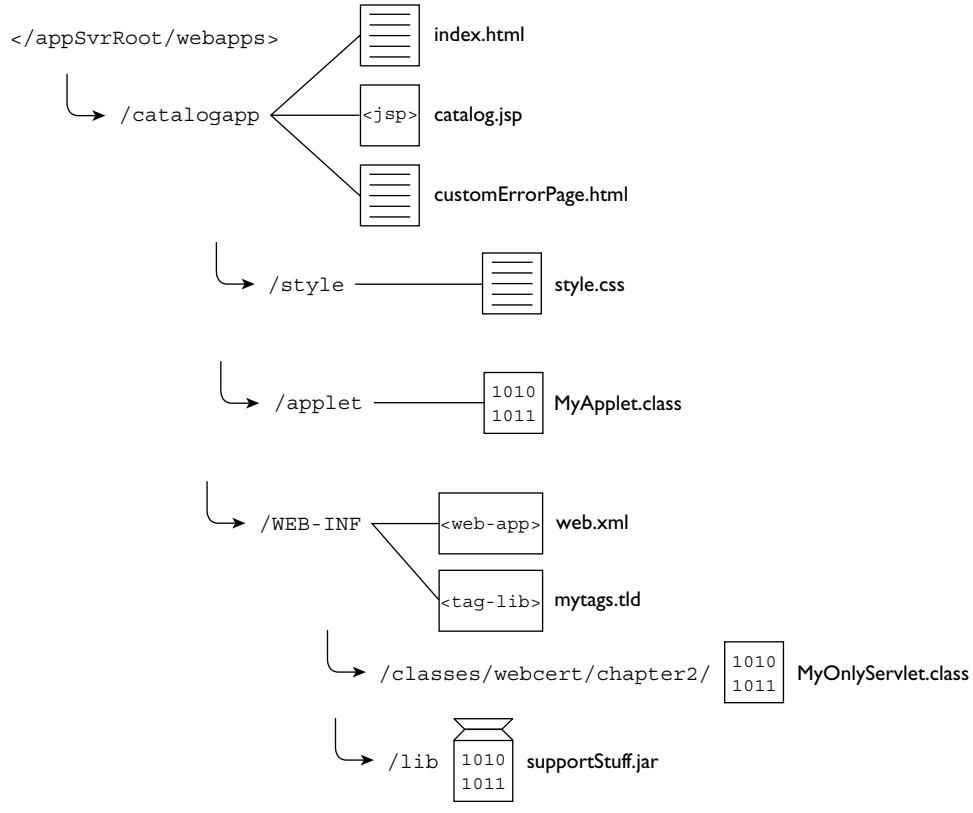


FIGURE 2-1

A Web Application Directory Structure with Typical Contents

Again, there is nothing wrong with defining your own directories (or directory structure) under WEB-INF. You might have, perhaps, a /WEB-INF/xml directory to house a bunch of XML configuration files that support your application. The point—especially for exam purposes—is that you can distinguish between the Sun-specified directories and any other sort.

Two special rules apply to files within the /WEB-INF directory. One is that direct client access should be disallowed with an HTTP 404 (file not found) error. The second regards the order of class loading. Java classes in the /WEB-INF/classes directory should be loaded before classes resident in JAR files in the /WEB-INF/lib directory.

Before doing the exercise at the end of this section, take a look at Figure 2-1, which shows a web application directory structure. Although sparsely populated, you can see a selection of different sorts of files and where they belong.

exam Watch

Having a directory called WEB-INF is a strong recommendation, but not an absolute obligation. Look out for questions that ask you to say whether a web application must have particular directories. The correct answer is "no"! More usually, though, questions will be phrased to allow for this loophole in the specification. So if you see a question along these lines—"Should a servlet class live in the WEB-INF/classes directory?"—you are safe to answer "yes." The expectation is that files normally do live in the recommended file structure.

on the Job

If you are designing a web container, or working with one that allows flexibility over the name of the WEB-INF directory, you should still support the usual naming convention. You may not like the standard, but practically all of the Java web application universe abides by it. So you're very much on your own if you go your own way.

EXERCISE 2-1

Using a Servlet to Look at the Context Path

In this exercise we are going to deploy a small web application containing a single servlet. By running the servlet, we will see details of the web application's home directory and context.

Install and Deploy

1. Start the Tomcat server.
2. In the book CD, find file sourcecode/ch02/ex0201.war.
3. Copy this file to <Tomcat Installation Directory>/webapps.
4. Observe the messages on the Tomcat console—make sure that it finds ex0201.war and installs it without error messages.

Explore Directories and Run Servlet

5. Use your file system facilities to confirm that a new directory has been created: <Tomcat Installation Directory>/webapps/ex0201.

6. Check under this directory for files in the right places: a couple of JSPs in the context root and a servlet class file under /WEB-INF/classes.
7. Now run the servlet. The default URL will be http://localhost:8080/ex0201/ShowContext. Study the output carefully. Look at the source code (which is listed to the web page, but you may find the original file easier to work with—look under /WEB-INF/src). Work out which parts of the code produce which parts of the web page.

exam Watch

You are very likely to get questions that require you to spot a misplaced file, such as a jsp in the WEB-INF directory or a class file in a subdirectory that isn't WEB-INF/classes.

CERTIFICATION OBJECTIVES

Deployment Descriptor Elements (Exam Objectives 2.2 and 2.3)

Describe the purpose and semantics of the following deployment descriptor elements: error-page, init-param, mime-mapping, servlet, servlet-class, servlet-mapping, servlet-name, and welcome-file.

Construct the correct structure for each of the following deployment descriptor elements: error-page, init-param, mime-mapping, servlet, servlet-class, servlet-mapping, servlet-name, and welcome-file.

Proper file structure is an important part of web application packaging, but the story does not end there. As we have seen, each web application contains a deployment descriptor file called web.xml in the WEB-INF directory. We need to start exploring the semantics of this file in gory detail. Again, your IDE will do you no favors. Chances are your IDE builds the WEB-INF file as you build each web component. As the IDE fills in deployment descriptor details from wizards or through intelligent guesswork, you are left in happy ignorance of web.xml detail. The exam, though, expects you to have developed deployment descriptors from childhood. You need to memorize the elements, the elements that go inside the elements, and sometimes the element order. Time to put aside that IDE and go to your text editor!

This section of the book is the first (but not the last) to examine web.xml elements. We'll start with some of the more fundamental ones, pertaining to servlets or the web application as a whole. As we look at different facets of web applications throughout the rest of the book, their associated deployment descriptor elements will be introduced.

Overall Structure of the Deployment Descriptor

The first thing to note about the deployment descriptor file is that it's an XML file. Given that the name is web.xml, you were probably ahead of me on that one. You might find it reassuring (or disappointing!) to know that next to no XML knowledge is required for the exam, but we will start this part of the chapter by giving you enough knowledge to tackle deployment descriptor semantics. With that foundation, we can go on to explore the seven specific deployment descriptor elements mentioned in the exam objective above.

The Least You Need to Know about XML

If you're reading this book, it's hard to believe that you have not been exposed to XML at some point in your development career. However, it can't do any harm to pin down some essentials about XML format that assume no previous knowledge and that will aid your efforts in deconstructing deployment descriptor files.

Let's start by looking at a minimalist deployment descriptor file, just for the purpose of picking out the XML features. The lines are numbered for ease of reference, as they might appear in some text editors, but the numbers are not part of the syntax.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <web-app version="2.4" xmlns=http://java.sun.com/xml/ns/j2ee
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
05  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
06
07   <welcome-file-list>
08     <welcome-file>index.jsp</welcome-file>
09   </welcome-file-list>
10
11 </web-app>
```

Line 1 This line tells us that the file is an XML file, which version of XML is used, and what character set is used for encoding the file contents.

Lines 2–5 and Line 11 These lines define the start and end tags of the root element `<web-app>`, together with a number of attributes that define aspects of the entire document. Every XML document must have a root element to enclose its contents. So in the case of web.xml, the start tag `<web-app>` is closed off with `</web-app>` on line 11. The start tag contains some attributes (name/value pairs). For example, the version attribute (*version*="*2.4*") marks the document version being used and matches the version number of the servlet specification. (A short digression on versions: The exam you are studying for pertains to the J2EE 1.4 standard. However, J2EE 1.4 embraces a whole range of technologies, each with an associated version number. So J2EE 1.4 has embraced version 2.4 of the servlet specification, which is what's reflected in the deployment descriptor document.) The remaining attributes all have to do with defining an associated "schema" document. The software, which reads an XML file (an XML Parser), has the option of validating the document contents against rules defined in the schema document.

Schemas are not the only mechanism by which XML documents can be validated. Up to and including J2EE 1.3 and version 2.3 of the servlet standard, web.xml was validated against something called a DTD—or document type definition. The heading material of the XML file looks slightly different, as you can see:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
```

However, the intention is the same—to have a means of validating the deployment descriptor. If you have worked on web applications prior to J2EE 1.4, the good news is that the actual rules for element validation have changed very little, even if the mechanism has. We'll examine the few subtle version differences as we encounter them in the discussion.

Lines 7 and 9 These lines define the start and end tags for `<welcome-file-list>`, one of the many immediate children of the root tag `<web-app>`. XML works by having pairs of tags nested inside each other, to any depth you like. We'll defer talking about what `<welcome-file-list>` actually does for you until later in the chapter; at the moment, we'll stick with XML syntax features.

Line 8 This line defines the start and end tag for `<welcome-file>`, child of `<welcome-file-list>` and grandchild of the root element `<web-app>.` The start and end tags enclose some character data: `index.jsp`.

In terms of content, this very brief deployment descriptor tells us that there is one welcome file, called index.jsp, defined in the welcome file list for this web application. Of course, we have many more options available to us for the information we record about our web application in the deployment descriptor. The next section opens up the possibilities.

Anatomy of web.xml

The deployment descriptor has, in fact, 27 top-level elements, shown in Figure 2-2. These top-level elements each contain — on average — a half dozen or so elements, at various levels of nesting, ranging from the very simple (e.g., `<distributable>`, with no nested elements) to `<servlet>` (with 9 nested elements, some containing more nested elements).

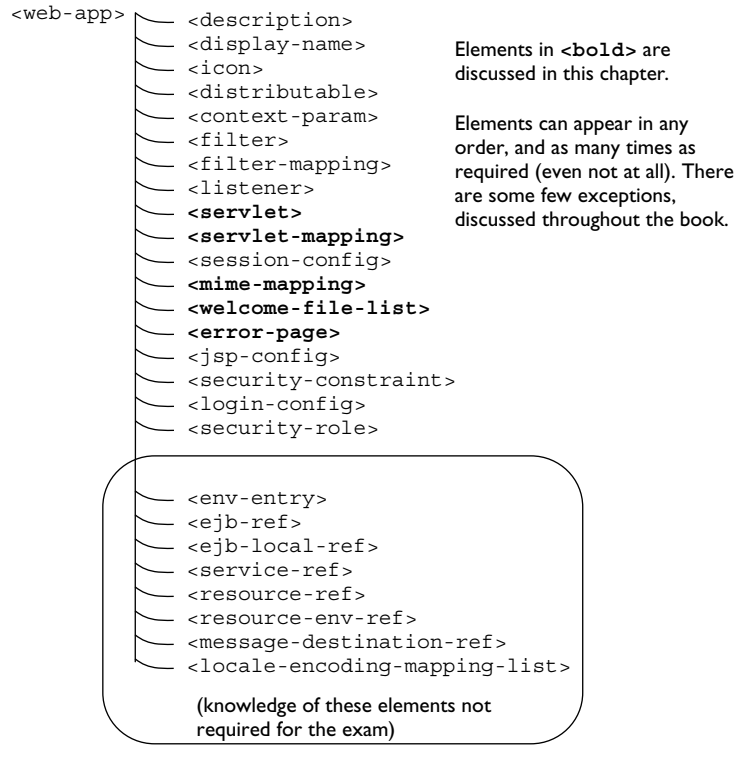


FIGURE 2-2

The Full List of Top-Level Elements in the Deployment Descriptor web.xml

exam Watch

A traditional type of "gotcha" exam question asks you to identify which of a list of elements within root element web-app are mandatory. None of them are! A web.xml file that contains only an empty root element (<web-app></web-app>) is perfectly legal.

Differences from Previous Servlet Specification

Good news for examinees: Top-level elements can now appear in any order! For an earlier version of the exam, I spent a lot of time memorizing sequence. Having said that, you are probably well advised to follow the order shown in Figure 2-2. People are very used to the order from previous versions of the servlet specification, so they might be thrown by placement of elements radically different from the established norm.

exam Watch

Sequence still matters within the elements inside the top-level elements—<servlet>, for example.

Some elements that used to be optional or appear once are now optional and can (according to schema validation) appear many times. This doesn't make a lot of sense in some circumstances. For example, the `<distributable>` element need only appear once for the application to be marked distributable—repeating the element ten times doesn't make the application any more distributable! The servlet specification tells you what containers are supposed to do when there is more than one occurrence of an element that formerly could only appear once, and we'll draw attention to the rules in subsequent chapters.

Deployment Descriptor Elements for Servlets

One of the top-level elements in the deployment descriptor is `<servlet>`. This is a "complex type" of element because it contains several other elements within itself. For the exam and for real-life development, it isn't sufficient just to know the top-level elements—you'll need to know the contents as well.

There are several characteristics you can define for a servlet, beyond a simple name. We'll need to give the deployment descriptor the fully qualified name of the actual Java class for a start. Other optional subelements control whether the servlet is loaded when the web container starts, and any security rules in force.

And we'll find that <servlet> is a slight misnomer. Not only can you use the element to define plain servlets; you can also use it to define a reference to a JSP (JavaServer Page). (However, as we'll see later, JSPs are converted to servlets by the web container, so perhaps the element name isn't such a misnomer after all.)

A separate element called <servlet-mapping> gives you flexibility over which URLs are used to access servlet resources, and we'll talk about that too.

<servlet> and Its Important Subelements

Figure 2-3 shows the <servlet> element expanded so that you can see how the subelements nest within one another, which ones are optional, and how many times each can appear. You can see that only the <servlet-name> and <servlet-class>/<jsp-file> elements are mandatory and that these elements can appear only once within the <servlet> element. Other elements, such as <load-on-startup>, are entirely optional. You don't have to have any <init-param> elements, or you can have as many as you like. The figure indicates that if you do include an <init-param>, it must house one <param-name> and one <param-value>.

The ordering of subelements within <servlet> is crucial. We've already noted that the top-level elements within the root element <web-app> can come in any order. However, the deployment descriptor schema validates that when it comes to the elements in <servlet>, then (for example) <servlet-name> must come before <servlet-class> (and after <icon>, if you choose to include an <icon> element).

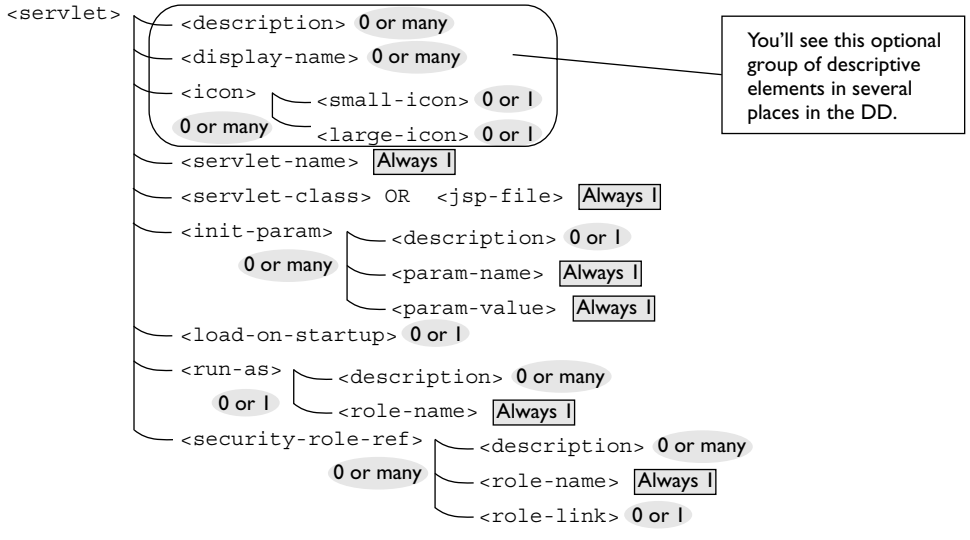


FIGURE 2-3 The Servlet Element Expanded

FIGURE 2-4 An Example <servlet> Declaration in the Deployment Descriptor

```
<web-app>
  <servlet>
    <description>A servlet for predicting the future</description>
    <description xml:lang="fr">une servlette pour prédire l'avenir</description>
    <display-name>Future Predictor</display-name>
    <display-name xml:lang="fr">Pour Prédire L'avenir</display-name>
    <icon>
      <small-icon>/images/futurep.gif</small-icon>
    </icon>
    <servlet-name>FutureServlet</servlet-name>
    <servlet-class>com.osborne.c02.FutureServlet</servlet-class>
    <init-param>
      <description>The number of months ahead to predict: default value</description>
      <param-name>months</param-name>
      <param-value>3</param-value>
    </init-param>
    <init-param>
      <description>How wild to make the prediction: default adjective</description>
      <param-name>wildness</param-name>
      <param-value>exaggerated</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
</web-app>
```

Fully qualified name of class—NB: don't add.class to the end!

run-as and security-role-ref omitted—these make a comeback in Chapter 5, on web security.

Figure 2-4 completes the picture by showing servlet deployment definitions in practice: a mostly complete servlet definition for a web.xml file. The more important start and end tags are in boldface. Just note for now that the servlet is called Future Servlet and that it maps to the Java class com.osborne.c02.FutureServlet. There are two initialization parameters, one called *months* and the other *wildness*, with values of 3 and "exaggerated," respectively.

Let's take a look at the more important subelements in a bit more detail.

<servlet-name> This subelement defines a logical name for the servlet. There aren't many rules for this element, but you should know them. The name must be unique within the web application. Any string for the name will do, provided it's at least one character long. There's no obligation to make this name the same as the Java class to which it relates (though people often do use the Java class name stripped of the fully qualified package parts).

Why do we want to name a servlet anyway? There are many other possible resources in a web application that don't boast a specially defined name in the deployment descriptor. However, here are at least a couple of reasons:

- Servlets are normally a protected resource, kept in the WEB-INF/classes directory, so that direct URL access to the servlet won't work. The servlet name is part

of the mechanism by which controlled access to servlets is allowed (you'll learn more about this when we look at `<servlet-mapping>`).

- A logical, unique name for the servlet is a deal less cumbersome than always referring to a servlet, say, by its fully qualified Java class name. You'll find that you can reference a servlet name at various points from elsewhere in the deployment descriptor.

Should you want to access the servlet name within your own code, you can. There is a `getServletName()` method defined in the ServletConfig interface (which is implemented by GenericServlet, so any inheriting servlet will have the method available). Here's a code listing showing how a servlet's `doGet()` method can print the servlet name on the server console:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
  System.out.println(this.getServletName());
}
```

exam Watch

You may think I'm very pedantic in pointing out which interface defines the `getServletName()` method (ServletConfig). You might encounter questions that ask you to distinguish whether a method originates from, say, the ServletConfig or the Servlet interface. Helpful hint: Three of the four methods on ServletConfig have to do with extracting information from the deployment descriptor (all of which you meet in this section); Servlet has no methods of this kind; its methods are mainly to do with servlet life cycle, which we meet in Chapter 3. However you do it, though, you need the methods of Servlet memorized for the exam, together with the interfaces from which they originate.

`<servlet-class>` This subelement defines the fully qualified name of a Java servlet class. You'll want this class to be a descendant of GenericServlet or HttpServlet. Your XML validation software won't tick you off if you violate this rule, but your web container will choke when it tries to run a nonservlet defined as a servlet class here. Separate parts of the package name should be separated with dots (nothing unusual there). On no account put ".class" at the end of the value you enter.

Although the names of servlets (defined in `<servlet-name>`) have to be unique, there is no such constraint on servlet classes. You can define the same

servlet class against two or more names, as shown in the deployment descriptor extract below:

```
<servlet>
 <servlet-name>MyServletHere</servlet-name>
 <servlet-class>webcert.chapter2.MyServlet</servlet-class>
</servlet>
<servlet>
 <servlet-name>MyServletThere</servlet-name>
 <servlet-class>webcert.chapter2.MyServlet</servlet-class>
</servlet>
```

Why would you want to do this? Although not shown, it's the only way of supplying separate sets of initialization parameters to the servlet, for starters. It's also a way of ensuring a separate running instance of the servlet. Normally, a web container would deal with any number of requests for the servlet logically known as MyServletHere by instantiating only one object of the MyServlet type. However, as soon as the web container received a request for MyServletThere, it would be forced to instantiate another separate object of MyServlet type. If it's important, you can use the `getServletName()` method to determine which instance you are running.

`...or <jsp-file>` We don't meet JSPs (JavaServer Pages) for another few chapters. When we do, you'll see that—unlike servlets—they are normally located in the HTTP-accessible regions of a web application. So users typically request a JSP directly from the web application context root (or a suitable subdirectory).

However, suppose we want a JSP that is not directly accessible, which we keep in some directory of WEB-INF—for example,

```
/WEB-INF/secure/concealed.jsp
```

The direct approach—say, http://localhost:8080/mywebcontext/WEB-INF/secure/concealed.jsp—rightly results in an HTTP 404 (page not found) error. However, the following deployment descriptor entries give a means of access:

```
<servlet>
  <servlet-name>ConcealedJSP</servlet-name>
  <jsp-file>/WEB-INF/secure/concealed.jsp</jsp-file>
</servlet>
<servlet-mapping>
  <servlet-name>ConcealedJSP</servlet-name>
  <url-pattern>/allIsRevealed</url-pattern>
</servlet-mapping>
```

The full chapter and verse on <servlet-mapping> will follow very soon. Suffice to say for the moment that the user can now type http://localhost:8080/mywebcontext/allIsRevealed into her browser and have the JSP returned. One possible reason for doing this is to conceal the usage of JSPs in the application: The URL gives nothing away. Another is to support existing links that are converted from static pages to JSPs (so . . . /index.html finds a JSP file). There is no necessity to keep the JSPs under WEB-INF for this purpose; just set up an appropriate servlet mapping.

<init-param> We saw in Chapter 1 how we can get parameters to a servlet from the web page <form>, which has the servlet as the subject of its action. <init-param> gives another means of priming a servlet, but this time the information is recorded directly in the deployment descriptor file. We saw in Figure 2-4 the full deployment descriptor details for the FuturePredictor servlet. Here's the deployment descriptor for the initialization parameters alone:

```
<init-param>
  <description>The number of months ahead to predict: default value</description>
  <param-name>months</param-name>
  <param-value>3</param-value>
</init-param>
<init-param>
  <description>How wild to make the prediction: default adjective</description>
  <param-name>wildness</param-name>
  <param-value>exaggerated</param-value>
</init-param>
```

The <init-param> envelope can repeat as many times as you want parameters for the servlet. Inside the envelope, we always find two subelements: <param-name> and <param-value>. These are mandatory, and they represent a key/value pairing: You use the key of the name to return the value. There can only be one <param-name>/<param-value> pairing for each <init-param> (if you want another pairing, use a fresh <init-param>). Should you wish, you can place a <description> before <param-name>.

All we need now is the means of retrieving the information, which is very easy. The code below uses two servlet methods (originating from the ServletConfig interface, implemented in the GenericServlet class) to get at the initialization information. `getInitParameterNames()` returns an Enumeration of all the parameter names available to the servlet. Armed with a parameter name, you can use `getInitParameter(String paramName)` to return an individual parameter value.

```
protected void doGet(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException {
  /* Local variables to hold parameter values */
  int months = 0;
  String wildness = "";
  /* Iterate through all the initialization parameters */
  Enumeration e = getInitParameterNames();
  while (e.hasMoreElements()) {
    String parmName = (String) e.nextElement();
    if (parmName.equals("months")) {
      months = Integer.parseInt(getInitParameter(parmName));
    }
    if (parmName.equals("wildness")) {
      wildness = getInitParameter(parmName);
    }
  }
  /* Return a page showing the values discovered */
  response.setContentType("text/plain");
  PrintWriter out = response.getWriter();
  out.write("Intialization parameters were 'months' with a value of "
    + months + ' and 'wildness' with a value of '" + wildness + "'");
}
```

Because `getInitParameter(String paramName)` returns a String value, you have to write your own parsing code to coerce numeric values to their right type—as is the case with the *months* parameter.

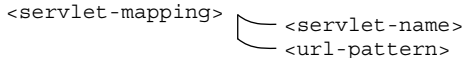
Finally, there's no servlet API to get at the optional `<description>` element of an initialization parameter. That's there for the benefit of those responsible for maintaining the deployment descriptor, and it might show up in a graphical administrative console.

exam Watch

<init-param> appears elsewhere in the deployment descriptor—not just as a nested element within <servlet>. Watch out for its very similar use as a nested element in <filter>, which makes its appearance in Chapter 3. And the trio of subelements—<description>, <param-name>, <param-value>—also crop up inside the <context-param> element.

The <servlet-mapping> element

Although we've defined a number of key aspects of the servlet, we haven't yet given users of our application any means of getting at it as a URL resource. That's what the <servlet-mapping> element is for. You might find it surprising that <servlet-mapping> is not another subelement of <servlet>; instead, it lives as a top-level element directly off the root <web-app>. Perhaps the designers of the deployment descriptor thought the <servlet> tag was already too overloaded. You can see in the following illustration expanding <servlet-mapping> and its subelements that it's considerably simpler than the related <servlet> element.



The subelement <servlet-name> should tie back to a <servlet-name> defined in a <servlet> element. The <url-pattern> subelement specifies what the user can expect to type into the URL after the context name and have her request find the associated servlet. So if the deployment descriptor has the following servlet mapping defined:

```
<servlet-mapping>
  <servlet-name>FutureServlet</servlet-name>
  <url-pattern>/myfuture</url-pattern>
</servlet-mapping>
```

And the user types something like the following URL:

```
http://localhost:8080/webappcontext/myfuture
```

Then the servlet (or JSP) defined in a corresponding <servlet> tag with a <servlet-name> of FutureServlet will execute.

exam Watch

What values are allowed for the URL pattern? You can use almost any characters you like. Two things to note: First, carriage returns and line feeds are not allowed. No real surprise there. What is more surprising is that white space is legal—a URL pattern such as <url-pattern>My Servlet</url-pattern> is legal and works. You'll probably find that your browser substitutes a hexadecimal representation of the white space character in the address line (%20).

INSIDE THE EXAM

URL Mapping Strategies

Actually, we're not quite done with servlet mapping. The servlet specification builds in a deal of flexibility into URL patterns, which you need to know for the exam. There are four kinds of mapping you can specify. There are also rules which dictate—in the case of more than one matching mapping for a URL—which mapping should take precedence. Let's look in more detail.

Exact Path Mapping The URL content following the context path exactly matches the URL pattern in the servlet mapping.

Longest Path Prefix The URL content following the context path is tested against partial paths specified in URL patterns. The longest match wins.

Extensions If the last part of the request URL is a file with an extension (e.g., the .jsp in /index.jsp), the extension is matched against any extension-type URL patterns.

Default Servlet If the above mapping methods have failed, the server may have one ace left up its sleeve: the default servlet.

The following table shows how to specify the URL patterns to indicate which of the four match methods is intended, and gives some examples of URLs that would cause a match.

Rule	URL Pattern	How to Form the URL Pattern	URLs That Would Match
Exact match	/findthis	Any string—must begin with "/."	/findthis
Path match	/findthat/here/*	String must begin with "/" and end in "/*."	/findthat/here /findthat/here/quickly /findthat/here/quickly/index.html
Extension match	*.jsp	String must begin with "*."	/index.jsp/any directory/index.jsp
Default	/	Single forward slash only: "/."	(any URL that fell through all other matching attempts)

Once a match is found according to the rules above, no further matching is attempted. And whereas these rules were merely "recommendations" in past versions of the servlet spec, web containers are now "required" to support them.

INSIDE THE EXAM (continued)

Two things to remember: URL patterns are case sensitive. A URL of /findthis would match a pattern of /findthis, but not /FindThis. Second, servers may have some implicit mappings already set up outside of web.xml—for example, something to trap a ".jsp" extension, for JSP files can't be served directly. If you specify your own extension match for ".jsp" or an alternative to the default servlet, then you will override what the server does: It's then your web application's responsibility to deal with the request.

on the job

Do you need a `<servlet-mapping>` to execute a servlet? The answer is "not necessarily." Many web servers have the capability of "serving servlets by name." There is nothing particularly magical about this, and it does—in fact—involve servlet mappings. Imagine your server had a `<url-pattern>` set up of /servlet/*, which mapped on to a servlet called ServletExecutor. This means that a request such as mycontext/servlet/SomeServletOrOther will invoke the ServletExecutor servlet. The ServletExecutor servlet determines that SomeServletOrOther is indeed a servlet within this web application, and it redirects control to the SomeServletOrOther servlet—even though there is no mapping necessarily set up for SomeServletOrOther. Convenient as such a facility is, you will want to switch it off in production environments for security reasons! This approach may be convenient for development and test environments, however; it saves some setup in the deployment descriptor. That said, you will probably be working with an IDE that sets up the deployment descriptor servlet mappings as part of the servlet creation process. In such circumstances, the usefulness of serving servlets by name dwindles somewhat. Added to this, you could argue that it's not a proper test of your servlet except in the context of mappings correctly set up in the deployment descriptor.

Other `<servlet>` Subelements

There are other elements embedded in the `<servlet>` tag that we haven't yet discussed. Some of these will return in future chapters and objectives. Others aren't explicitly mentioned in the exam objective, but your knowledge of the `<servlet>` tag wouldn't be complete without them.

We'll start with the trio of `<description>`, `<display-name>`, and `<icon>`. This is a standard grouping of elements that occurs several times in the deployment descriptor. For example, these three elements are actually the first three top-level elements under `<web-app>` (see Figure 2-2). In that case, they apply to the entire web application. As subelements of `<servlet>`, they apply to a particular servlet. As you might hope, though, they are functionally equivalent wherever they appear.

We'll end this section with elements that are entirely specific to `<servlet>`.

`<description>` Optionally, you can enter descriptive text for your servlet in this tag. There is no API in the servlet packages to retrieve this description. It's not for the consumers of your web application; it's for the benefit of administrators. So a web container might have an administrative console that chose to display this text for a deployed web application, for example.

You can include as many descriptions as you want (i.e., separate occurrences of the description element). What's wrong with just one, you ask? The reason is to accommodate multiple languages. You can qualify each description element with the *xml:lang* attribute, giving a valid two-character country code. If you omit *xml:lang*, then a default of *xml:lang*="*en*" is presumed. Here's an extract from a longer web.xml file, which shows both an English and a French description:

```
<description>A servlet for predicting the future</description>
<description xml:lang="fr">une servlette pour prédire l'avenir</description>
```

`<display-name>` The function of `<display-name>` is very similar to the `<description>` element. It's also meant for use in web container administrative user interfaces in order to provide a short descriptive name—perhaps less cryptic than the servlet name, but less expansive than the description text. However, you can provide any string you want, of course. The same rules apply about language: You can have as many display names as different languages. Here's an example:

```
<display-name>Future Predictor</display-name>
<display-name xml:lang="fr">Pour Prédire L'avenir</display-name>
```

`<icon>` This subelement is the last of the descriptive trio. As with the others, it is entirely optional, and you can have many occurrences. Within the icon element you can embed a `<small-icon>` and a `<large-icon>` element (one, both, or neither). The element describes a path (from the context root) to an image file (JPG and GIF are the permitted formats) that might be used by your web container administrative GUI to display next to your servlet. Example:

```
<icon>
  <small-icon>/images/futurep.gif</small-icon>
</icon>
```

That concludes the "descriptive trio" of elements. There are three remaining subelements of `<servlet>`: `<load-on-startup>`, `<run-as>`, and `<security-role-ref>`. Actually, we're going to postpone the last two until we discuss security in Chapter 5, leaving just `<load-on-startup>`.

`<load-on-startup>` A web container's usual practice is to load a servlet at the point where it is first accessed. In fact, the web container is free to load the servlet at any point it regards as suitable. However, by defining the `<load-on-startup>` element, the servlet is loaded at the point when the web container starts. Furthermore, you can control the order in which servlets load by the integers you specify as the values of the `<load-on-startup>` tags:

- A servlet with a lower number will be loaded before a servlet with a higher number.
- If the numbers are the same, you're in the lap of the web container designers—there are no guarantees on which servlet starts first.
- If the number is negative, the web container can do whatever it pleases regarding loading the servlet: It's as if the `<load-on-startup>` element wasn't there at all.

If a `<jsp-file>` is specified rather than a `<servlet-class>`, a `<load-on-startup>` setting ensures that the JSP is pre-compiled (turned into a servlet), then loaded as any other servlet would be.

Before you individually register hundreds of JSPs from your web-app in web.xml just to force pre-compilation, check out the facilities of your application server. These days, almost every application server has an option for pre-compiling JSPs at the point of deploying the application into the server.

Other Deployment Descriptor Elements

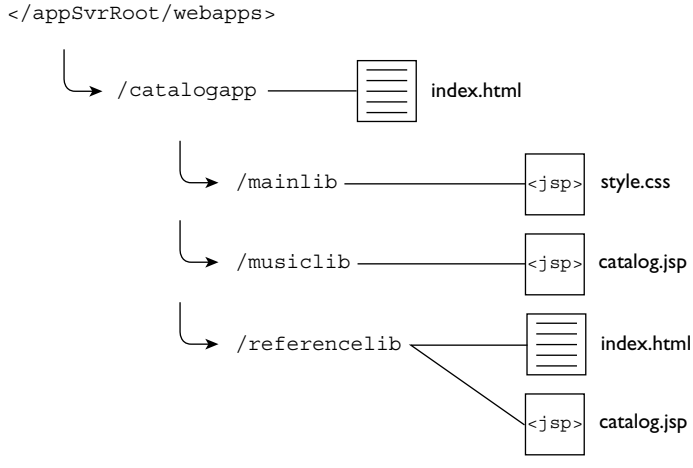
Now we'll turn our attention to some of the deployment descriptor elements that affect the web application as a whole, as opposed to individual servlets.

Welcome Files: <welcome-file-list>

There have probably been many occasions in your life where you have typed in a web site address into your browser—such as www.osborne.com—and, owing to some magic in the target web server, you are taken to the resource for a specific URL, say http://www.osborne.com/index.html. If a J2EE web application is the object of your request, the chances are that this behavior comes about through the specification of a welcome file list in the deployment descriptor. Here's an example:

```
<welcome-file-list>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>mainlibrary/catalog.jsp</welcome-file>
</welcome-file-list>
```

To see how this works, we need a set of files and a directory structure for the corresponding web application. We're imagining a library application with a host of mylibrary.com and a context of catalogapp. Here we see the visible (HTTP-accessible) directories and files from context root downward.



Now we need to consider what happens when you request a URL that falls to the web container to deal with but that doesn't immediately match a specific resource. A trailing slash is appended to the URL if not already present. Then each entry in the

http://mylibrary.com/catalogapp	■ "/" appended to URL. ■ "index.html"—first welcome-file—appended to URL. ■ catalogapp directory checked for index.html—found! ■ index.html returned to requester.
http://mylibrary.com/catalogapp/mainlib/	■ "index.html"—first welcome file—appended to URL. ■ mainlib directory checked for index.html—not found. ■ "index.jsp"—second welcome file—appended to URL. ■ mainlib directory checked for index.jsp—found and returned!
http://mylibrary.com/catalogapp/musiclib/	■ "index.html"—first welcome file—appended to URL but not found in musiclib directory. ■ "index.jsp"—second welcome file—appended to URL but still no match in musiclib directory. ■ "catalog.jsp"—third welcome file—appended to URL. Found and returned!
http://mylibrary.com/catalogapp/referencelib/	■ "index.html"—first welcome file—appended to URL. Found and returned! ■ *Even though catalog.jsp—the third welcome file—is present in the referencelib directory, it won't be returned by the welcome file mechanism, which will always find index.html first.*

welcome file list is appended—in turn—to this URL and tested for a match against a specific resource. If the resource is present in the directory, it's returned to the requester. Here are some examples:

The rules in the servlet specification state that welcome-file entries must describe a partial URL with *no leading and no trailing slashes*. This makes sense when you consider that the entries are appended to a URL that has a trailing slash (either because the user typed it into the browser or the web server put it in implicitly) and should match up to a specific resource (not a directory).

exam Watch

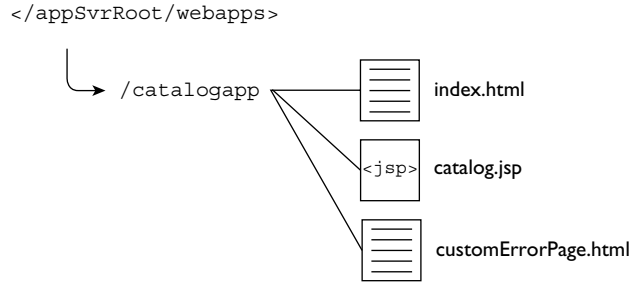
It's very easy to be caught out on the exact wording and nesting of the tags. Look out for questions that get the element order inside out (<welcome-file-list> nested under <welcome-file>) or that introduce subtle and incorrect variant spellings (<welcome-files>). Remember also that the direct parent of <welcome-file-list> is <web-app>, the root element.

Error Files: <error-page>

In Chapter 1 we examined the world of HTTP requests, including the error codes that crop up when requests go bad. The <error-page> element in the deployment descriptor gives you customized control over the web page displayed to the user in the event of requests going wrong. Let's suppose you have the following set up in your deployment descriptor:

```
<web-abb>
  <error-page>
    <error-code>404</error-code>
    <location>/customErrorPage.html</location>
  </error-page>
</web-app>
```

Your context root contains your customErrorPage.html file and a couple of other files.



A user means to request catalog.jsp but erroneously enters the following in his browser:

```
http://mylibrary.com/catalogapp/qatalog.jsp
```

The request correctly routes to the web application because the context is correct, but it doesn't find the specific resource qatalog.jsp. The result is an HTTP 404 error code, page not found. Instead of returning the standard web server page for this error, the web container looks for a matching error code in the deployment descriptor's list of error codes. A match is found, so the custom error page customErrorPage.html is set up instead.

There are some important things to note about the deployment descriptor construction:

- `<error-page>` has `<web-app>` as its immediate parent.
- `<error-code>` and `<location>` both have `<error-page>` as their parent.
- The resource specified in `<location>` *must* start with a "/." The path described is from the context root.

exam Watch

It's very easy to get the rules for `<location>` in `<error-page>` confused with `<welcome-file>` in `<welcome-file-list>`. Resources specified in `<location>` must begin with a "/," whereas those in `<welcome-file>` must not begin with a "/." Neither should end with a "/," of course, because they specify particular resources—files and servlets—not directories.

The web container is perfectly capable of generating HTTP status codes when the conditions are right (or should I say wrong!). We can—in servlet code—generate our own status or error codes, using the `sendError()` and `setStatus()` methods on HttpServletResponse. So servlet code like this (to generate a "404" error):

```
response.sendError(HttpServletResponse.SC_NOT_FOUND);
```

would cause the error page mechanism to kick in just as effectively as the user typing error we saw a few moments ago. And so would

```
response.setStatus(HttpServletResponse.SC_NOT_FOUND);
```

However, it is a *bad thing* to use `setStatus()` for anything other than normal conditions.

on the job

You are not limited to specifying static html pages as the location for an error page. You can specify a servlet to perform any dynamic processing you like. Just make sure that you specify a location value that will chime in with the servlet mapping for your dynamic error servlet.

That's not the end of the story for `<error-page>`. HTTP status codes are not the only error page mechanism at your disposal. You can also map plain old Java exceptions to particular error pages. In that case, you substitute the `<exception-type>` tag for `<error-code>`:

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

If you have a servlet that happens to throw a ServletException at runtime, the web container will return the custom error page as specified. Take care to specify the full qualified name of the exception (e.g., *javax.servlet*.ServletException), though, or the mechanism will not be triggered, and you'll get some standard application server error page instead.

Mime-mapping: `<mime-file>`

MIME (Multipurpose Internet Mail Extensions) is an Internet standard for describing media types. By "media," understand "file"—which could be anything from plain text to images to movies. Don't be fooled by the "Mail" part of the MIME acronym: The standard has been embraced by web servers, application servers, and web browsers everywhere. See http://www.iana.org/assignments/media-types/ for the official list of MIME types.

We already saw in Chapter 1 how you could set the MIME type of a servlet response programmatically. That's fine and appropriate when the object of your request is a servlet. However, your web application may serve up other types of resources. Some are likely to be understood. If I set up plain text files with a .txt extension in my web application, my web container serves them up to my browser without difficulty. What if I wanted to serve up my own brand of XML files, though, with an .xmldavid extension? My application server returns them happily enough, but with no mime-type, the browser has to do the best it can to figure out the response. However, if I set up a `<mime-mapping>` entry in the deployment descriptor as follows:

```
<web-app>
  <mime-mapping>
    <extension>xmldavid</extension>
    <mime-type>text/xml</mime-type>
  </mime-mapping>
</web-app>
```

The client has an opportunity to process the file as an XML file, despite the unusual extension. I say "opportunity," because browsers may ignore the MIME-type set in the response and apply their own rules on processing the content.

EXERCISE 2-2

Defining Deployment Descriptor Elements

In this exercise we are going to deploy a small web application containing a single servlet. The servlet isn't very useful because the supplied deployment descriptor file is minimal (`<web-app></web-app>`). You will complete the deployment descriptor in the course of the exercise and run the servlet.

Install and Deploy

1. Start the Tomcat server.
2. In the book CD, find file sourcecode/ch02/ex0202.war.
3. Copy this file to <Tomcat Installation Directory>/webapps.
4. Observe the messages on the Tomcat console—make sure that it finds ex0202.war and installs it without error messages.
5. Delete ex0202.war.

Adjust the Deployment Descriptor

6. Find the deployment descriptor file, web.xml (in <Tomcat Installation Directory>/webapps/ex0202/WEB-INF).
7. Using your favorite text editor, amend the file to have a `<welcome-file-list>` pointing to index1.jsp, and an `<error-page>` pointing to error.jsp for an HTTP status code 404 error.
8. Restart Tomcat.
9. Check that index1.jsp displays for URL http://localhost:8080/ex0202/. (Hint: If this—or any other page—fails to display, first make sure you *clear your*

browser cache by hitting the refresh button. This applies throughout this and future exercises!)

10. Check that error.jsp displays for URL http://localhost:8080/ex0202/notfound.html. (You may have a problem seeing this page—some browsers have a habit of substituting their own 404 "page not found" error page instead of deferring to the server's page.)
11. Stop Tomcat.
12. Re-edit web.xml. Add a servlet definition for servlet class webcert.ch02.ex0202.ShowInitParms. Give the servlet any number of initialization parameters you like. Don't forget to add a servlet mapping for the servlet.
13. Restart Tomcat, and call the servlet using your mapping details.
14. If you are really stuck (and only if!), check out the web.solution.xml file in the /WEB-INF directory of the web application.

CERTIFICATION OBJECTIVE

WAR Files (Exam Objective 2.4)

Explain the purpose of a WAR file and describe the contents of a WAR file, how one may be constructed.

Web applications get chock-full of directories and files in even the most unambitious of projects. If you are happy to deploy and manage all those directories and files individually, you have more of a taste for configuration management than I do. Fortunately, the J2EE providers thought of that and provided a standard for packaging all web application components into a single zip-format file whose format is known as the web archive—or WAR, for short.

Packaging Your Web Application

Provided you have abided by the file and directory naming rules outlined in the first section of this chapter, packaging your web application is no big deal. You simply take the contents beneath the context path and zip up the whole structure into

one file. The context path itself is not part of the WAR. WARs are designed to be unzipped into a context path of the deployer's choosing.

The structure of a WAR file is exactly the same as a Java archive (JAR), which is in turn the same as a ZIP file. So on a Windows system, a tool such as WinZip is very useful for interrogating the contents of a WAR file. A zip-type tool is all that's required to package and unpackage a WAR file, and, of course, the J2SDK comes with one supplied in the shape of the "jar" tool.

A WAR Is Not a JAR

Although a WAR file can be produced in the same way as a JAR file, and has the same underlying file format, it is different. The most obvious difference is the file extension naming convention: .jar for Java ARchive, and .war for Web (Application) ARchive.

JARs are packaged in a particular way to make it easy for a running JVM to find and load Java class files. WARs are packaged for a different purpose: to make it as easy as possible for a web container to deploy an application.

Several web containers have automatic deployment mechanisms. The server recommended for this book—Tomcat—has a "webapps" directory. Place a WAR file in this directory, and Tomcat (by default) will un-jar the contents into the file system under the webapps directory. It provides a context root directory with the same name as the WAR file (but without the .war extension)—then makes the application available for use. The interesting contrast here is that WAR files are not *necessarily* "un-jarred" for use; some web containers run web applications directly from the WAR file itself. For example, the Sun application server that comes with the J2EE 1.4 download has an "autodeploy" directory. Placing the WAR file there causes the server to load the constituent parts of the application into memory and available for use—but doesn't unzip them.

Either way, I hope you are beginning to see the point of having the WAR file standard. When it comes to deployment, life is very easy. Get the packaging wrong, however, and your web container will disown the WAR file in no uncertain manner.

Just before we move on to methods for making WAR files, we need to consider one last required directory for our web application: META-INF.

The META-INF Directory

In the beginning, WARs were intended to be completely self-contained. Everything a web application relied on would be packaged in the WAR. This setup was convenient, but it overlooked the fact that many web applications deployed on the same

server might make use of many common libraries of code. Including these as JAR files in every WAR seemed wasteful; if nothing else, it led to huge inflation of WAR file size. Configuration management was potentially an issue, for updating a common library JAR file meant duplicating it to every web application that used it.

The solution was to allow web containers to provide a common repository for code. How and where the web container defines this common repository is specific to the web container; it may have several. It's quite a complex business that demands associated rules to do with class loading. For a good explanation on how one web container does this—Tomcat—take a look at

```
http://jakarta.apache.org/tomcat/tomcat-5.0-doc/class-loader-howto.html
```

For exam purposes, though, you don't need to know anything of the detail. All you need is some grasp of the general principle so that the purpose of META-INF is clearer to you.

Although the rules for storing and loading common code are container-specific, the mechanism by which a WAR references such common code is a J2EE standard. Each WAR can now reference the JARs it needs in this common repository by using an existing mechanism: a manifest file. This file is called MANIFEST.MF, and it must be located in the META-INF directory in the WAR file.

The essential content of MANIFEST.MF is a list of JARs under a heading of "classpath." Suppose you wanted to take advantage of a logging library such as log4j in your web server's common repository. You might see an entry in the file such as this:

```
Classpath: log4j.jar
```

Note that only the JAR file name is present; the server knows which specific directories to search.

There is plenty more you can specify about referenced code in MANIFEST.MF, such as version information or the vendor that sold (and digitally signed) a code library. You can find chapter and verse on the standard at

```
http://java.sun.com/j2se/1.4.2/docs/guide/extensions/
```

You have to introduce META-INF only at the WAR-making stage, and then only if you have a need for it. Chances are you will want it available to you in most

applications; indeed, most development tools will put it there for you whether you request it or not.

exam Watch

When META-INF is present in your web application, the same access rules apply as for WEB-INF. Server-side code is welcome to access resource files in the META-INF directory. However, the web container should reject any attempt at client-side access with the regular "page not found" HTTP error code of 404.

Cross-referencing common code isn't the only purpose for the META-INF directory. If you are signing the WAR file for security reasons, or JARs contained in WEB-INF/lib, the META-INF directory is also the right place to store digital certificates.

Making WARs

Now that we know what a WAR is for and how it's made up, let's look a bit more closely at how we make a WAR. The most obvious method is using the **jar** tool in the J2SDK. For the exam you will be expected to know the basic parameters to provide to **jar**, both for packaging and unpackaging WAR files.

The **jar** tool is not the only game in town, however. We'll touch on some other WAR-making devices that you're more likely to encounter in "real" development than naked use of the **jar** command.

Packaging Your Web Application

To run the jar command, you will want a command line. To make things easier, change directory to the context directory of the web application that you want to package. You'll need to have your <J2SDK>/bin directory in your PATH. Enter the following command (for Windows—UNIX is very similar):

```
jar cvf0 mywarfile.war *.*
```

This creates a WAR file called mywarfile.war in the context directory, containing all the files in the context directory and any subdirectories (/WEB-INF, /WEB-INF/lib, etc.). The following table explains the parameters:

c	Creates the WAR file.
v	Verbose: outputs messages on the command line telling you about every file added.
f	A WAR file name will be specified (*can* be omitted, but you're unlikely to want to go there).
0	Don't compress the file (you would usually omit this).
mywarfile .war	The name of the WAR file to be created. It must follow straight after the "cvf0" group (separated by a space).
.	The files to include in the WAR file: in this case, everything. Must follow straight after the WAR file name (separated by a space).

Unpackaging Your Web Application

Unpackaging looks very similar, and it is not usually something you would do manually anyway, for web servers invariably have their own deployment mechanisms, as previously explained. If you do want to expand a WAR file manually so that you know how to for the exam, here are instructions:

- Create a context directory in the relevant part of your web server's hierarchy.
- Put the WAR file in the context directory.
- Execute the following command (Windows and UNIX are very similar):

```
jar xvf mywebapp.war
```

- Command explanation follows:

x	Extracts the contents of the WAR file.
v	Verbose: outputs messages on the command line telling you about every file extracted.
f	A WAR file name will be specified.
mywarfile .war	The name of the WAR file to be created. It must follow straight after the "xvf" group (separated by a space).

exam Watch

Another useful command parameter for the **jar** *command is* **t**, *to display the contents of a WAR file. The full command might look like this:* `jar tf mywarfile.war`.

on the Job

Development Tools (IDEs) often have built-in functionality to export WAR files directly from your web development environment, and most will do more than this. Furthermore, if your IDE is lacking, your web server probably won't be. Most web servers come with some sort of assembly and packaging tool. You specify the files and directories you want to include using the tool's graphical user interface; out pops a WAR file at the other end. These tools tend to more than just package WARs: They have facilities to build the deployment descriptor from graphical dialogs and to cope with other aspects of J2EE: Enterprise Java Beans (EJBs), EAR files, Resource Files . . . the full story belongs in a separate book!

exam Watch

In real life, you will use all the packaging support tools you can get hold of. But don't neglect to practice more primitive assembly methods so you really get the structure in your mind. Just as you had to "be the compiler" in the programmer exam, you have to "be the assembler" in the web component exam!

EXERCISE 2-3

Making and Deploying WARs

In this exercise we are going to jar up a web application, then deploy it.

Make the WAR File

1. Stop the Tomcat server.
2. Using your command line facility, navigate to <Tomcat Installation Directory>/webapps/ex0201.

3. Use the **jar** command with appropriate parameters to create ex0203.war, which zips up all the files from Exercise 2-1.

Adjust the Deployment Descriptor

4. Make a directory under <Tomcat Installation Directory>/webapps called ex0203.
5. Move the WAR file you made in step 3 to this directory.
6. Use the jar command to extract the contents of the file.
7. Check that the extraction worked: There should be an error.jsp and index1 .jsp directly in the ex0203 directory, as well as a WEB-INF and META-INF directory.
8. Start the Tomcat server.
9. This application is a clone of Exercise 2-1, deployed to a new context root—ex0203. Check the deployment by running the ShowContext servlet:

```
http://localhost:8080/ex0203/ShowContext
```

CERTIFICATION SUMMARY

In this chapter you have learned a lot about the fundamental structure of web applications. You are now equipped not just to write servlet code but also to structure the code and all other resources that make up a web application—by putting them in the right directory structure and by providing declarative information about them in the deployment descriptor.

You started by looking at directory structure. You learned that web applications have a "context path" at their root. By making up a stub URL from the server details and context path, you saw how you could append a path to find a particular resource in your web application. You learned that resources meant for public HTTP access should be placed directly in the directory matching the context path, or in subdirectories with names of your own choosing off the context root.

You further learned that every web application has a directory called WEB-INF, which must exist directly in the context root. You saw that any resources kept in

WEB-INF are not for direct public HTTP access. You learned that, as a minimum, WEB-INF must contain a file called web.xml — the deployment descriptor for the application. WEB-INF also has two other officially sanctioned directories: WEB-INF/classes (for separate Java class files, typically servlets and other supporting code) and WEB-INF/lib (for JAR files of Java code). You learned that a web container should look for the classes it needs first in WEB-INF/classes, then in WEB-INF/lib.

You went on to learn about the deployment descriptor file. You examined the XML structure and several of the more fundamental elements within the file. In the exercises you set up `<servlet>` elements, logically defining servlet names and their corresponding servlet classes. You examined the deployment descriptor elements and code required to access initialization parameters in servlets, using a combination of `<init-param>` tags and ServletConfig methods such as `getInitParameter(String paramName)` and `getInitParameterNames()`. You learned how to set up corresponding `<servlet-mapping>` elements so that users of your web applications can enter a URL to access a servlet resource. You learned about the four different sorts of URL mappings and the order in which they are processed: exact match, path prefix, extension, and default. You also learned about some of the more esoteric elements affecting servlets — optional `<description>`, `<display-name>`, and `<icon>` elements for the benefit of web server administrators and graphical web server consoles — and the concept of being able to load servlets in a predefined order determined by `<load-on-startup>` elements.

After learning about deployment descriptor elements affecting servlets, you looked at elements that affect the web application as a whole. These include the `<welcome-file-list>`, which gives a web server a possible resource to serve up when the user requests a directory rather than a specific file. You also met `<error-page>`, which allows you to associate customized pages with specific HTTP error codes (such as 404 — SC_NOT_FOUND) and/or Java exception types. Finally, you learned about `<mime-mapping>` and saw how you used this to associate an arbitrary file extension with a known file type from a predefined list of MIME file types.

In the last section of the chapter, you turned your attention to web archive (WAR) files. You found out how you could compress your application into a single, ZIP-format file with a .war extension. In the exercises, you experimented with the process of deploying WAR files on to the Tomcat server. You also learned how to handcraft your own WAR files using the **jar** command with appropriate parameters (cf mywar.war *.*), and how to use the **jar** command to reverse the process and extract the files from a WAR into a web application directory structure on the file system. You learned that WAR files may contain a META-INF directory (short for meta-information), which may have a manifest file (MANIFEST.MF) describing dependencies on common code lying outside of the web application context.

TWO-MINUTE DRILL

File and Directory Structure

- ❏ Every web application within a web container has a unique context path.
- ❏ The context path and any directories you choose to create within it contain resources that are accessible through HTTP.
- ❏ HTTP-accessible resources in your context path might include but are not restricted to static HTML files, JavaServer Pages, Java applets and support code (including JARs), JavaScript files, images, media clips, and stylesheets.
- ❏ The context path contains a special directory called WEB-INF, which must contain the deployment descriptor file, web.xml.
- ❏ A client application may not directly access resources in WEB-INF or its subdirectories through HTTP—any attempt to do so results in an HTTP 404 (page not found) error.
- ❏ Server-side web application code is permitted to access files in WEB-INF and its subdirectories, using methods such as `getResourceAsStream(String path)` on the ServletContext interface.
- ❏ The special directory /WEB-INF/classes contains Java class files—servlets and supporting code.
- ❏ The special directory /WEB-INF/lib contains JAR files with supporting libraries of code.
- ❏ You can create your own directories as needed under /WEB-INF.
- ❏ Apart from those already mentioned (web.xml, class files, JAR files), resources that you expect to find under /WEB-INF include tag libraries (.tld files—discussed in Chapter 8), configuration files (typically xml or properties files), and server-side scripts.

Deployment Descriptor Elements

- ❏ The deployment descriptor file is called web.xml, and it *must* be located in the WEB-INF directory.
- ❏ web.xml is in XML (extended markup language) format. Its root element is `<web-app>`.
- ❏ web.xml houses other elements, each with a start and end tag, which in turn may house other elements.

- At the lowest level of nesting, elements in web.xml have character data. Example: `<web-app><welcome-file-list><welcome-file>index.html </welcome-file></welcome-file></web-app>`.
- The deployment descriptor is now validated by a schema file (at servlet spec level 2.4) rather than a DTD (at servlet spec level 2.3). The validation rules are little altered, though top-level elements can now appear in any order.
- The `<servlet>` element holds declarative information about a servlet. It has only two mandatory subelements—`<servlet-name>`, a logical name for the servlet, and `<servlet-class>`, the fully qualified Java class name *without the .class extension*.
- A `<servlet>` element can include zero to many `<init-param>` elements: These aid in passing initialization parameter information from the deployment descriptor to the servlet.
- Each `<init-param>` element contains the optional `<description>` element and the mandatory `<param-name>` and `<param-value>` elements.
- The ServletConfig interface (implemented by GenericServlet and its subtypes) contains APIs to get at deployment descriptor information about the servlet: `getServletName()`, `getInitParameterNames()`, and `getInitParameter(String paramName)`.
- The same servlet class can be declared using different logical names in the deployment descriptor.
- Different logically named servlets are implemented as different instances by the web container (even if the same servlet class is referenced).
- The `<servlet>` element can define a JSP rather than a servlet. In that case, the element `<jsp-file>` is substituted for `<servlet-class>`.
- The `<servlet-mapping>` element provides a means of defining URLs to use the servlet resources otherwise inaccessible under WEB-INF.
- There are four different sorts of mapping information you can provide in the `<url-pattern>` element. The web container processes them in strict order of precedence: exact path (/exactmatch), path prefix (longest match first) (/partial/*), extension matching (*.jsp), and default servlet (/).
- Paths are case-sensitive.
- The `<welcome-file-list>` element provides a list of one or more pages to return when the user types a path that identifies a directory.

- ❏ The `<error-page>` element associates custom error pages with HTTP status codes and/or Java exception types.
- ❏ The `<mime-mapping>` element serves to associate file extensions with officially recognized file types.

WAR Files

- ❏ Web archive (WAR) files provide a convenient means of storing an entire web application in a single, compressed file.
- ❏ WAR files must have a .war file extension.
- ❏ The contents of the context directory and all its subdirectories (including WEB-INF) should be included in the WAR file, but *not the context directory itself*. A WAR file can be installed at any context path.
- ❏ A WAR file can be created with the **jar** command, using *cf* as parameters: `jar cf myapp.war *.*`.
- ❏ A WAR file can be extracted with the **jar** command, using *xf* as parameters: `jar xf myapp.war`.
- ❏ A WAR file must contain a META-INF directory, containing a file called MANIFEST.MF. This lists dependencies on common code JAR files stored outside of the web application context (but available through a web server's own mechanisms).
- ❏ The META-INF directory can also be used to store security-related resources, such as signature files and digital certificates.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there might be more than one correct answer. The number of correct choices to make is stated in the question, as in the real SCWCD exam.

File and Directory Structure

1. Which of the following directories are legal locations for the deployment descriptor file? Note that all paths are shown as from the root of the machine or drive. (Choose two.)
   A. /WEB-INF
   B. /appserverInstallDirectory/webapps/webappName/WEB-INF/xml
   C. /appserverInstallDirectory/webapps/webappName/WEB-INF
   D. /appserverInstallDirectory/webapps/webappName/WEB-INF/classes
2. What would be the best directory in which to store a supporting JAR file for a web application? Note that in the list below, all directories begin from the context root. (Choose one.)
   A. /WEB-INF
   B. /WEB-INF/classes
   C. /jars
   D. /web-inf/jars
   E. /CLASSES
   F. /WEB-INF/lib
   G. /lib
   H. None of the above.
3. What's the likely outcome of a user entering the following URL in her browser? You can assume that index.html does exist in /WEB-INF/html, where /WEB-INF/html is a directory off the context root, and that the server, port, and context details are specified correctly. (Choose one.)

   ```
   http://localhost:8080/mywebapp/WEB-INF/html/index.html
   ```

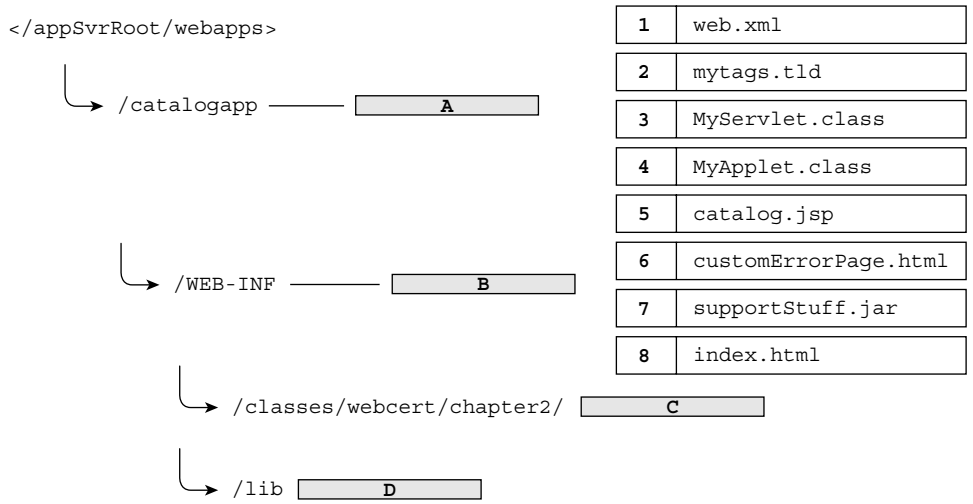
   A. Because the file is an HTML file, the web application serves it back to the browser.
   B. An HTTP response code in the 500 range is returned (server error).

C. An HTTP response code of 403 is returned to indicate that the server is not allowed to serve files from this location.

D. An HTTP response code of 404 returned to indicate that the requested resource has not been found.

E. None of the above.

4. (drag-and-drop question) In the following illustration, match the numbered files on the right to the appropriate lettered locations on the left. All files must find a home, so you will have to use some of the lettered locations for more than one file.



5. Identify which of the following are true statements about web applications. (Choose three.)

A. The only way to access resources under the /WEB-INF directory is through appropriate servlet mapping directives in the deployment descriptor.

B. Server-side code has access to all resources in the web application.

C. Clients of web applications can't directly access resources in /WEB-INF/tld.

D. A good place to keep a .tld (tag library file) is directly in the /WEB-INF directory.

Deployment Descriptor Elements

6. See the extract from web.xml below:

```
<servlet-mapping>
  <servlet-name>ServletA</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>ServletB</servlet-name>
  <url-pattern>/bservlet.html</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>ServletC</servlet-name>
  <url-pattern>*.servletC</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>ServletD</servlet-name>
  <url-pattern>/dservlet/*</url-pattern>
</servlet-mapping>
```

Given that a user enters the following into her browser, which (if any) of the mapped servlets will execute? (Choose one.)

```
http://myserver:8080/mywebapp/Bservlet.html
```

A. ServletA
B. ServletB
C. ServletC
D. ServletD
E. The answer is dependent on the web container you use.
F. None of the above: A 404 "page not found error" will result.

7. What is the parent tag for `<welcome-file-list>`? (Choose one.)

A. `<welcome-file>`
B. `<web-app>`
C. None — the tag doesn't exist.
D. `<welcome-files>`
E. `<servlet>`

8. Which of the following are true statements about the deployment descriptor for a web application? (Choose two.)

   A. At least one `<servlet>` element must be present.

   B. `<welcome-file>` is a child element of `<welcome-file-list>`.

   C. `<web-application>` is the root element.

   D. `<servlet>` elements must all be declared before `<servlet-mapping>` elements.

   E. At least one element must be present.

9. (drag-and-drop question) Complete the missing lettered elements from the deployment descriptor in the following illustration, using the numbered choices on the right. You will not have to use all the numbered choices but may have to use some more than once.

```
<servlet>
  <[ A ]>Question 09 Servlet</[ A ]>
  <display-name>Question09</display-name>
  <[ B ]>Question09</[ B ]>
  <[ C ]>com.osborne.[ D ]</[ C ]>
  <init-param>
     <[ E ]>myParm</[ E ]>
     <[ F ]>myAttribute</[ F ]>
   </init-param>
   <load-on-startup>10</load-on-startup>
</servlet>
<servlet-mapping>
  <[ G ]>Question09</[ G ]>
  <[ H ]>/Question09</[ H ]>
</servlet-mapping>
```

1	servlet-name
2	servletname
3	servlet-id
4	Question09
5	Q09.class
6	servlet
7	servlet-mapping
8	url-mapping

9	description
10	desc
11	param-attribute
12	jsp-class
13	parm-name
14	param-attribute
15	parm-attribute
16	parm-attribute
17	param-value
18	parm-value
19	url-pattern
20	servlet-class

10. What of the following represents a correct declaration of a servlet in the deployment descriptor? (Choose one.)

A.

```
<servlet>
  <servlet-class>MyServlet</servlet-class>
  <servlet-name>MyServlet</servlet-name>
</servlet>
```

B.

```
<servlet>
  <servlet-name>MyServlet</servlet-name>
  <servlet-class>MyServlet.class</servlet-class>
</servlet>
```

C.

```
<servlet>
  <description>My Servlet</description>
  <servlet-name>MyServlet</servlet-name>
  <servlet-class>MyServlet</servlet-class>
</servlet>
```

D.

```
<servlet>
  <servlet-class>MyServlet</servlet-class>
  <jsp-file>index.jsp</jsp-file>
</servlet>
```

11. Given five servlets with `<load-on-startup>` value set as follows, and declared in the following order in the deployment descriptor,

- ServletA: 1
- ServletB: 0
- ServletC: 1
- ServletD: 1
- ServletE: no value set for `<load-on-startup>`

Identify true statements from the list below. (Choose one.)

A. ServletA will load before ServletB.
B. ServletB will load before ServletC.
C. ServletC will load before ServletD.
D. ServletD will load before ServletE.
E. ServletA will load before ServletE.

12. What will be the outcome of compiling and deploying the servlet code below? (You can assume that correct import statements are provided and that the servlet lives in the default package. Line numbers are for ease of reference and are not part of the code.)

```
11 public class NameServlet extends HttpServlet {
12   protected void doGet(HttpServletRequest request,
13     HttpServletResponse response) {
14       out.write(getServletName());
15   }
16 }
```

A. Will not compile because the `doGet()` method doesn't throw the correct exceptions
B. Will not compile for some other reason
C. When run, terminates with a ServletNotFoundException at line 14
D. Outputs "NameServlet"
E. Outputs the contents of the corresponding `<servlet>` element
F. Outputs the contents of the corresponding `<servlet-name>` element

13. Assume that there is a file called secure.txt, located at /WEB-INF/securefiles, whose contents are "Password=WebCert." What statements are false about the result of compiling and running the following code?

```
11 public class CodeTestServlet extends HttpServlet {
12   protected void doGet(HttpServletRequest request,
13     HttpServletResponse response) throws IOException {
14     ServletContext sc = getServletContext();
15     InputStream is = sc.getResourceAsStream("/WEB-" +
16       "INF/securefiles/secure.txt");
17     BufferedReader br = new BufferedReader(new InputStreamReader(is));
18     System.out.println(br.readLine());
19   }
20 }
```

A. The code will not compile.
B. A RuntimeException will occur at lines 15/16.
C. An IOException will occur at line 18.
D. The string "Password=WebCert" will be returned to the requester.
E. A, B, and C above.
F. B, C, and D above.
G. A, B, C, and D above.

14. Given the following deployment descriptor:

```
<web-app>
 <servlet>
   <servlet-name>InitParams</servlet-name>
   <servlet-class>com.osborne.c02.InitParamsServlet</servlet-class>
   <init-param>
     <param-name>initParm</param-name>
     <param-value>question14</param-value>
   </init-param>
  </servlet>
</web-app>
```

What is the outcome of running the following servlet? (Choose one.)

```
public class InitParamsServlet extends HttpServlet {
 protected void doGet(HttpServletRequest request,
   HttpServletResponse response) throws ServletException, IOException {
   ServletContext sc = this.getServletContext();
   PrintWriter out = response.getWriter();
   out.write("Initialization Parameter is: "
     + sc.getInitParameter("initParm"));
 }
}
```

A. A runtime error
B. "Initialization Parameter is: null" written to the console
C. "Initialization Parameter is: question14" returned to the requester
D. "Initialization Parameter is: null" returned to the requester
E. "Initialization Parameter is: question14" written to the console

15. Which of the following methods derive from the ServletConfig interface? (Choose three.)

    A. `ServletContext getServletContext()`

    B. `String getInitParameter(String name)`

    C. `MapEntry getInitParameterEntry()`

    D. `Iterator getInitParameterNames()`

    E. `String getServletName()`

16. Which of the following is a valid way to set up a mime mapping in the deployment descriptor? (Choose one.)

    A.

```
<mime-mapping-list>
  <mime-type>text/plain</mime-type>
  <extension>txt</extension>
<mime-mapping-list>
```

    B.

```
<mime-mapping-list>
  <extension>.txt</extension>
  <mime-type>text/plain</mime-type>
<mime-mapping-list>
```

    C.

```
<mime-mapping>
  <mime-type>txt</mime-type>
  <extension>text/plain</extension>
<mime-mapping>
```

    D.

```
<mime-mapping>
  <extension>txt</extension>
  <mime-type>text/plain</mime-type>
<mime-mapping>
```

17. Which of the following servlet methods can return **null?** (Choose one.)

    A. `getInitParameterNames()`

    B. `getInitParameter(String name)`

    C. `getServletName()`

    D. `getServletContext()`

WAR Files

18. Identify correct statements about the META-INF directory from the list below. (Choose three.)
    A. META-INF is a suitable location for storing digital certificates.
    B. META-INF is used as a repository for common code.
    C. The MANIFEST.MF file is found in the META-INF directory.
    D. The deployment descriptor file is found in the META-INF directory.
    E. META-INF is not directly accessible to clients.

19. Identify correct statements about WAR files from the list below. (Choose three.)
    A. A META-INF directory will be present in the WAR file.
    B. A WEB-INF directory will be present in the WAR file.
    C. A web container can't work directly from a WAR file; it must be extracted (unzipped) into the file system.
    D. A WAR file is in ZIP file format.

20. Consider the following list of files in a web application, where myApp is the context path:

```
/devDir/myapp/index.jsp
/devDir/myapp/WEB-INF/web.xml
/devDir/myapp/WEB-INF/classes/webcert/ch02/SomeServlet.class
```

    Which of the following sets of instructions will build a correctly formed web archive file? (Choose one.)
    A. None of the sets of instructions will build a valid WAR file until /webcert/ch02/SomeServlet.class is moved to the WEB-INF/lib directory.
    B. Change directory to /devDir; execute jar tvf myapp.war *.*
    C. Change directory to /devDir/myApp; execute jar cvf myapp.jar *.*
    D. Change directory to /devDir/myApp/WEB-INF; execute jar xvf myapp.war *.*
    E. Change directory to /devDir/myApp; execute jar cvf someapp.war *.*

LAB QUESTION

It's your turn now to develop a web application from scratch! Develop a servlet that displays its name back to the user. Register this same servlet class three times in the deployment descriptor. Prove to yourself (through extra code in the servlet) that by calling servlets with different names, you are genuinely getting different instances of the servlet (i.e., separate Java objects). One way of doing this is to place an instance variable in the servlet that keeps a count of how many times the servlet has been called.

SELF TEST ANSWERS

File and Directory Structure

1. ☑ **A** and **C** are the correct answers. The deployment descriptor file, web.xml, must go directly in the WEB-INF directory. **A** looks strange—it would be peculiar, not to say foolish, to have the context of a web app located in the root directory—but it is still legal.
   ☒ **B** is incorrect; though it's perfectly OK to create a directory called xml within WEB-INF to keep your own configuration files, it is not OK to have web.xml housed there. **D** is incorrect because although WEB-INF/classes is a standard J2EE-defined directory, it's meant for Java classes (such as servlet classes), not web.xml.

2. ☑ **F** is the correct answer. WEB-INF/lib is the right place for supporting JAR files, though you can include JAR files in the META-INF directory as well.
   ☒ The remaining answers are incorrect. Only **A**, **B**, and the correct answer, **F**, define directories found in the servlet specification. **D** is wrong on two counts, one of which is that the case is wrong (the directory is WEB-INF in capitals), and the other is that you're welcome to have a subdirectory called "jars," but there's no standard to say that the web container should look there. **C** and **G** come straight off the context root, which is the publicly accessible area. Finally, **H** is wrong because there is a correct answer!

3. ☑ **D**. A 404 response code should be returned: resource not found. That way, the server masks the fact that a resource even exists at the location specified (as it does in this example).
   ☒ **A** tries to fool you into thinking that certain types of file will be served from WEB-INF and its subdirectories: incorrect. **B** (a 500 range error) is reserved for genuine server problems (uncaught exceptions in servlet code). **C** (a 403 error) sounds reasonable; you might expect a "nonauthorized" type message. But that reveals that there is a resource to get at. **E** is incorrect because there is a correct answer.

4. ☑ **A** is the location for **4**, **5**, **6**, and **8**: static HTML (including custom error pages), Java Server Pages, and applet classes should live in the context directory (or—not shown in the picture—a directory that isn't WEB-INF under the context directory). **B** is the location for **1** and **2**: The deployment descriptor web.xml must be located here, and tag library descriptors (.tld files) can be located here or a subdirectory of WEB-INF. **C** is the location for **3**, a servlet class file: under /WEB-INF/classes, in its own package directory. **D** is the location for **7**: /WEB-INF/lib is for supporting JAR files.
   ☒ Other combinations are ill-advised or won't work at all.

5. ☑ **B**, **C**, and **D**. Server-side code can get at anything in the web application, even resources under the WEB-INF directory. Clients can't directly access resources under WEB-INF/tld (don't be thrown by the fact the WEB-INF/tld isn't an "official" directory; it's perfectly OK to invent a directory called tld, and because it's under WEB-INF, clients can't get at it).
☒ **A** is incorrect. The only way for a *client-side* application to access resources under WEB-INF is through a servlet mapping, true enough. But *server-side* code can get directly at those resources through, for example, `ServletContext.getResourceAsStream(String path)`.

Deployment Descriptor Elements

6. ☑ **A** is correct. ServletA—set up for the default mapping of "/"—will execute.
☒ **B** is incorrect because BServlet.html does not match the URL pattern for ServletB in terms of case sensitivity. **C** and **D** don't have mappings remotely similar to the URL requested. **E** is incorrect because mapping behavior is not permitted to be server-specific since the 2.4 servlet spec. Finally, **F** is incorrect—because a default servlet mapping is set up, you will never get a 404 error (unless you code the default servlet to return a 404 error).

7. ☑ **B**. `<welcome-file-list>` nests directly under the root element `<web-app>`.
☒ **A** is incorrect because `<welcome-file>` is the child of `<welcome-file-list>`. **D** is incorrect because `<welcome-files>` does not exist. Answer **C** tries to persuade you that `<welcome-file-list>` doesn't exist, but it does. **E** encourages you to think that `<welcome-file-list>` is a subelement of `<servlet>`; of course, it's not, as it pertains to the whole application, not just one particular servlet.

8. ☑ **B** and **E**. `<welcome-file>` is the child of `<welcome-file-list>`. And there does have to be one element in web.xml: the root element (`<web-app></web-app>`).
☒ **A** is incorrect because you don't have to have a `<servlet>` element, or indeed any element except for the root element. **C** is incorrect because `<web-app>` is the root element, not `<web-application>`. **D** is incorrect because, since servlet spec 2.4 (in J2EE 1.4), order no longer matters: Elements can come in any order. That said, Tomcat (the reference implementation) server objects to referencing the servlet in a `<servlet-mapping>` before it is declared in a `<servlet>` element—you can, however, have the elements alternating so that servlet mappings are kept close to their associated servlets.

9. ☑ **A** maps to **9** (description), **B** to **1** (servlet-name), **C** to **20** (servlet-class), **D** to **4** (Question09—the answer most likely to be a class name without the extension .class), **E** to **16** (param-name), **F** to **17** (param-value), **G** to **1** (servlet-name—again), and **H** to **19** (url-pattern).
☒ Other combinations are incorrect.

10. ☑ **C** is the correct answer. All the elements are correctly specified, in the correct order.
☒ **A**, **B**, and **D** are incorrect. **A** reverses the `<servlet-name>` and `<servlet-class>` tags (order does matter within the `<servlet>` element). **B** has correct element order but incorrectly appends ".class" to the servlet name. **D** is almost correct but for the fact that the JSP file should be expressed from the context root and so begin with a leading slash, thus: `<jsp-file>/index.jsp</jsp-file>`.

11. ☑ **B**. The web container must guarantee that ServletB, with a `<load-on-startup>` value of 0, loads before ServletA, with a `<load-on-startup>` value of 1.
☒ **A** is incorrect because servlets with a negative `<load-on-startup>` value have an indeterminate load time—probably on first user access, but not guaranteed. Servlets with no load-on-startup value are indeterminate in the same way; hence, answers **D** and **E** are incorrect. **C** is incorrect because there is no guarantee that servlets with the same `<load-on-startup>` value will load in their declared order in the deployment descriptor.

12. ☑ **B**. It won't compile for other reasons—the reason being simply that the `out` variable is not declared (it's presumably meant to be the PrintWriter obtained from HttpServletResponse).
☒ **A** is incorrect because it's OK to throw fewer exceptions on a method than are in your superclass. **C** is incorrect because the code will never run, and in any case there isn't such a thing as ServletNotFoundException. **D** could be correct if the code compiled: If the servlet isn't registered in web.xml, the class name is returned from `getServletName()`. In the same way, **F** would be correct if code compiled and the servlet was registered. **E** would never be correct; the `<servlet>` element contains a lot else besides the servlet name.

13. ☑ **G** is the correct answer, for all of **A**, **B**, **C**, and **D** are false statements. **A** is false because the code will compile. **B** is false because there's nothing wrong with the method call and the path to the file is correctly stated. **C** is false; although an IOException is always possible from IO-based methods, it mostly won't happen. **D** is false because the string read from the file is not returned to the requester, but output to the server console.
☒ **A**, **B**, **C**, **D**, **E**, and **F** are incorrect answers, following the reasoning in the correct answer.

14. ☑ **D**. Although there is a correctly set up initialization parameter for the servlet in the deployment descriptor, the code is looking for a *context* parameter. There isn't one set up, and **null** is returned.
☒ **A** is incorrect; the code runs fine. **B** and **E** are incorrect, for nothing is written to the console—the output is to the response's PrintWriter, and so it is returned to the requester. **C** would be right if the code was set up to return the *servlet's* initialization parameter.

15. ☑ **A**, **B**, and **E**. All are correct signatures for methods on the ServletConfig interface.

☒ **C** is incorrect; there's no such method as `getInitParameterEntry()`. **D** is incorrect: ServletConfig does have a method called `getInitParameterNames()`, but it returns a good old-fashioned Enumeration, not an Iterator.

16. ☑ **D** is the correct answer. This has the correct element names and sequence and content for a `<mime-mapping>`.
☒ **A** and **B** are incorrect because both have the wrong-named outer element, `<mime-mapping-list>`. In addition, **A** reverses the `<extension>` and `<mime-type>` elements, and **B**—while getting the order correct—declares the extension content with a "." (it must be specified as "txt," not ".txt"). **C** is incorrect only in that `<mime-type>` and `<extension>` are reversed.

17. ☑ **B**. If an initialization parameter name does not exist, `getInitParameter(String name)` returns **null**.
☒ **A** is incorrect because `getInitParameterNames()` returns an empty Enumeration if there are no servlet initialization parameters declared in the deployment descriptor. **C** is incorrect because `getServletName()` always returns some name or other: Even if the servlet is undeclared, it will return the class name of the servlet. **D** is incorrect because there must always be a ServletContext to return (no servlet can operate in a vacuum).

WAR Files

18. ☑ **A**, **C**, and **E**. It's the right place for digital certificates and the MANIFEST.MF file. Like WEB-INF, client access to META-INF should be rejected with an HTTP 404 error.
☒ **B** is incorrect because META-INF isn't used to store common code across web applications; the MANIFEST.MF text file within it references common code via classpath entries. **D** is incorrect because the deployment descriptor file web.xml is kept in WEB-INF, not META-INF.

19. ☑ **A**, **B**, and **D**. **A** might surprise you, and there are plenty of WAR files around without a META-INF directory that deploy OK on most web servers. However, the servlet spec section 9.6 does say that META-INF "will be present."
☒ **C** is incorrect. Although most web servers do expand WAR files into the file system (like Tomcat), it's not a requirement. The Sun application server (part of the J2EE 1.4 reference implementation) doesn't expand WAR files, but runs the application directly from the WAR file itself.

20. ☑ **E** is the correct answer. Note that the WAR file name need bear no relation to the context path.
☒ **A** is incorrect, for the servlet class is in entirely the right place. Only if it was in a JAR file should it be present in /WEB-INF/lib. **B** is incorrect because you shouldn't be zipping up the

context directory itself, only the contents of the context directory and below. **C** is incorrect because the file created has a .jar extension, and a WAR file must have a .war extension. **D** is incorrect because you can't just wrap up the WEB-INF directory; you need all the web content in the directory above, the context root.

LAB ANSWER

Deploy the WAR file from the CD called lab02.war, in the /sourcecode/chapter02 directory. This contains a sample solution. You can call the servlets using a URL such as `http://localhost:8080/lab02/ServletA` (or `../ServletB` or `../ServletC`). The source for the servlet is included in the WEB-INF/src directory. If you experience strange behavior even though your code and deployment descriptor look right (counts not incrementing, perhaps), then do make sure you *refresh your browser cache* as you make repeat calls to each servlet URL and after redeployment of the WAR file.