PART I
Creating Enterprise Applications

1

Introducing Java Platform, Enterprise Edition

IN THIS CHAPTER

- Java SE and Java EE version timeline
- Introducing Servlets, filters, listeners, and JSPs
- Understanding WAR, and EAR files, and the class loader hierarchy

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

There are no code downloads for this chapter.

NEW MAVEN DEPENDENCIES FOR THIS CHAPTER

There are no Maven dependencies for this chapter.

A TIMELINE OF JAVA PLATFORMS

The Java language and its platforms have had a long and storied history. From its invention in the mid-'90s to an evolution drought from 2007 to nearly 2012, Java has gone through many changes and encountered its share of controversy. In the earliest days, Java, known as the Java Development Kit or JDK, was a language tightly coupled to a platform composed of a small set of essential *application programming interfaces* (*API*s). Sun Microsystems unveiled the earliest alpha and beta versions in 1995, and although Java was extremely slow and primitive by today's standards, it began a revolution in software development.

In the Beginning

Java's history is summarized in Figure 1-1, a timeline of Java platforms. As of the publication of this book, the Java language and the Java SE platform have always evolved together — new versions of each always release at the same time and are tightly coupled to one another. The platform was called the JDK through version 1.1 in 1997, but by version 1.2 it was clear that the JDK and the platform were not synonymous. Starting with version 1.2 in late 1998, the Java technology stack was divided into the following key components:

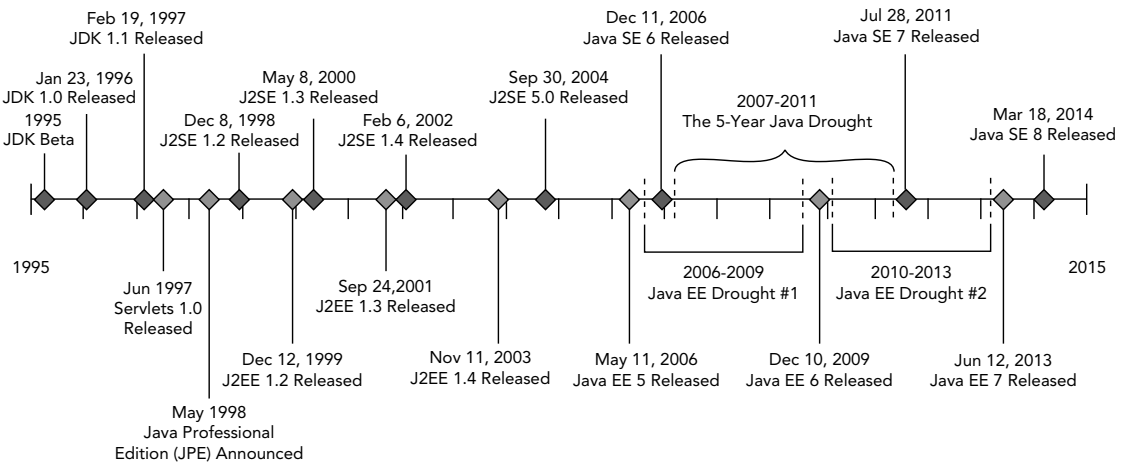

FIGURE 1-1: A timeline showing the correlation of the evolution of Java Platform, Standard Edition and Java Platform, Enterprise Edition. The events on top of the timeline represent Java SE milestones while the events on the bottom represent Java EE milestones.

- *Java* is the language and includes a strict and strongly typed syntax with which you should be very familiar by now.
- *Java 2 Platform, Standard Edition*, also known as *J2SE*, referred to the platform and included the classes in the `java.lang` and `java.io` packages, among others. It was the building block that Java applications were built upon.
- A *Java Virtual Machine*, or *JVM*, is a software virtual machine that runs compiled Java code. Because compiled Java code is merely bytecode, the JVM is responsible for compiling that bytecode to machine code before running it. (This is often called the *Just In Time Compiler* or *JIT Compiler*.) The JVM also takes care of memory management so that application code doesn't have to.
- The *Java Development Kit*, or *JDK*, was and remains the piece of software Java developers use to create Java applications. It contains a Java language compiler, a documentation generator, tools for working with native code, and (typically) the Java source code for the platform to enable debugging platform classes.
- The *Java Runtime Environment*, or *JRE*, was and remains the piece of software end users download to run compiled Java applications. It includes a JVM but does not contain any of the development tools bundled in the JDK. The JDK, however, does contain a JRE.

All five of these components have historically been specifications, not implementations. Any company may create its own implementation of this Java technology stack, and many companies have. Though Sun offered a standard implementation of Java, J2SE, the JVM, the JDK, and the JRE, IBM, Oracle, and Apple also created competing implementations that offered different features.

The IBM implementation was born out of need — Sun didn't offer binaries capable of running on IBM operating systems, so IBM created its own. The situation was similar for the Apple Mac OS operating system, so Apple rolled its own implementation as well. Although the implementations offered by these companies were all free as in beer, they were not free as in freedom, so they were not considered open source software. As such, the open source community quickly formed the OpenJDK project, which provided an open source implementation of the Java stack.

Still more companies created less popular implementations, some of which compiled your application to machine code for a target architecture to improve performance by avoiding JIT compilation. For the vast majority of users and developers, the Sun Java implementation was both sufficient and preferred. After Oracle's purchase of Sun, the Sun and Oracle implementations became one and the same.

Not shown in Figure 1-1 is the development of other languages capable of using the J2SE and running on the JVM. Over the years, dozens of languages appeared that can compile to Java bytecode (or machine code, in some cases) and run on the JVM. The most high-profile of these are Clojure (a Lisp dialect), Groovy, JRuby (a Java-based Ruby implementation), Jython (a Java-based Python implementation), Rhino, and Scala.

The Birth of Enterprise Java

This brief history lesson might seem unnecessary — as an existing Java developer, you have likely heard most of this before. However, it's important to include the context of the history of the Java Platform, Standard Edition, because it is tightly woven into the birth and evolution of the Java Platform, Enterprise Edition. Sun was already aware of the need for more advanced tools for application development, particularly in the arena of the growing Internet and the popularity of web applications. In 1998, shortly before the release of J2SE 1.2, Sun announced it was working on a product called the Java Professional Edition, or JPE. Work had already begun on a technology known as *Servlets*, which are miniature applications capable of responding to HTTP requests. In 1997, Java Servlets 1.0 released alongside the Java Web Server with little fanfare because it lacked many features that the Java community wanted.

After several internal iterations of Servlets and the JPE, Sun released *Java 2 Platform, Enterprise Edition* (or *J2EE*) version 1.2 on December 12, 1999. The version number corresponded with the current Java and J2SE version at the time, and the specification included:

- Servlets 2.2
- JDBC Extension API 2.0
- Java Naming and Directory Interface (JNDI) 1.0
- JavaServer Pages (JSP) 1.2
- Enterprise JavaBeans (EJB) 1.1

- Java Message Service (JMS) 1.0
- Java Transaction API (JTA) 1.0
- JavaMail API 1.1
- JavaBeans Activation Framework (JAF) 1.0.

Like J2SE, J2EE was a mere specification. Sun provided a *reference implementation* of the specification's components, but companies were free to create their own as well. Many implementations evolved, and you learn about some of them in the next chapter. These implementations included and still include open source and commercial solutions. The J2EE quickly became a successful complement to the J2SE, and over the years some components were deemed so indispensable that they have migrated from J2EE to J2SE.

Java SE and Java EE Evolving Together

J2EE 1.3 released in September 2001, a little more than a year after Java and J2SE 1.3 and before Java/J2SE 1.4. Most of its components received minor upgrades, and new features were added into the fold. The following joined the J2EE specification, and the array of implementations expanded and upgraded:

- Java API for XML Processing (JAXP) 1.1
- JavaServer Pages Standard Tag Library (JSTL) 1.0
- J2EE Connector Architecture 1.0
- Java Authentication and Authorization Service (JAAS) 1.0

At this point the technology was maturing considerably, but it still had plenty of room for improvement.

J2EE 1.4 represented a major leap in the evolution of the Java Platform, Enterprise Edition. Released in November 2003 (approximately a year before Java/J2SE 5.0 and 2 years after Java/J2SE 1.4), it included Servlet 2.4 and JSP 2.0. It was in this version that the JDBC Extension API, JNDI, and JAAS specifications were removed because they had been deemed essential to Java and moved to Java/J2SE 1.4. This version also represented the point at which J2EE components were broken up into several higher-level categories:

- **Web Services Technologies:** Included JAXP 1.2 and the new Web Services for J2EE 1.1, Java API for XML-based RPC (JAX-RPC) 1.1, and Java API for XML Registries (JAXR) 1.0
- **Web Application Technologies:** Included the Servlet, JSP, and JSTL 1.1 components, as well as the new Java Server Faces (JSF) 1.1
- **Enterprise Application Technologies:** Included EJB 2.1, Connector Architecture 1.5, JMS 1.1, JTA, JavaMail 1.3, and JAF
- **Management and Security Technologies:** Included Java Authorization Service Provider Contract for Containers (JACC) 1.0, Java Management Extensions (JMX) 1.2, Enterprise Edition Management API 1.0, and Enterprise Edition Deployment API 1.1

The Era of the Name Changes

Enter the era of the name changes, which are often a source of confusion for Java developers. They are highlighted here so that you fully understand the naming conventions used in this book and how they relate to the previous naming conventions you may already be familiar with. Java and J2SE 5.0 were released in September 2004, and included generics, annotations, and enums, three of the most radical language syntax changes in Java history. This version number was a departure from previous patterns, made more confusing by the fact that the J2SE APIs and the `java` command-line tool reported the version number as being 1.5. Sun had made the decision to drop the 1 from the publicized version number and go by the minor version, instead. It quickly recognized that the "dot-oh" on the end of the version number was a source of confusion and quickly began referring to it as simply version 5.

About the same time, the decision was made to retire the name Java 2 Platform, Standard Edition in favor of Java Platform, Standard Edition and to abbreviate this new name Java SE. The changes were made formal with Java SE 6, released in December 2006, and to this day the name and version convention has remain unchanged. Java SE 6 is internally 1.6, Java SE 7 is internally 1.7, and Java SE 8 is internally 1.8.

The same name and number change decisions were applied to J2EE, but because J2EE 1.5 was set to release between J2SE 5.0 and Java SE 6, the changes were applied a version early. Java Platform, Enterprise Edition 5, or Java EE 5, was released in May 2006, approximately 18 months after J2SE 5.0 and 7 months before Java SE 6. Internally Java EE 5 is 1.5, Java EE 6 is 1.6, and Java EE 7 is 1.7. Whenever you see the terms J2SE or Java SE, they are interchangeable, and the preferred and accepted name today is Java EE. Likewise, J2EE and Java EE are interchangeable, but Java EE is preferred today. The rest of this book refers to them exclusively as Java SE and Java EE.

Java EE 5 grew and included numerous changes and improvements again, and today it is still one of the most widely deployed Java EE versions. It included the following changes and additions:

- JAXP and JMX moved to J2SE 5.0 and were not included in Java EE 5.
- Java API for XML-based Web Services (JAX-WS) 2.0, Java Architecture for XML Binding (JAXB) 2.0, Web Service Metadata for the Java Platform 2.0, SOAP with Attachments API for Java (SAAJ) 1.2, and Streaming API for XML (StAX) 1.0 were added to Web Services Technology.
- Java Persistence API (JPA) 1.0 and Common Annotations API 1.0 were added to Enterprise Applications Technology.

The Java SE and EE Droughts

The release of Java SE 6 in December 2006, marked the beginning of a drought for Java SE releases that lasted approximately 5 years. This time was a period of frustration and even anger for many in the Java community. Sun continued to promise new language features and APIs for Java SE 7, but the schedule continued to slip year after year with no end in sight. Meanwhile other technologies, such as the C# language and .NET platform, caught up to and surpassed Java in language features and platform APIs, causing some to speculate whether Java had reached the end of its useful life. To make matters worse, Java EE entered its own drought period and by 2009, more than 3 years

had passed since Java EE 5 was released. All was not lost, however. Java EE 6 development picked up in early 2009, and it released in December 2009, 3 years and 7 months after Java EE 5, and 3 years almost to the day after Java SE 6.

By this time, Java Enterprise Edition became enormous:

- SAAJ, StAX, and JAF moved to Java SE 6.
- The Java API for RESTful Web Services (JAX-RS) 1.1 and Java APIs for XML Messaging (JAXM) 1.3 specifications were added to Web Services Technologies.
- The Java Unified Expression Language (JUEL or just EL) 2.0 was added to Web Application Technologies.
- Management and Security Technologies saw the addition of Java Authentication Service Provider Interface for Containers (JASPIC) 1.0.
- Enterprise Application Technologies realized the most dramatic increase in features, including Contexts and Dependency Injection for Java (CDI) 1.0, Dependency Injection for Java 1.0, Bean Validation 1.0, Managed Beans 1.0, and Interceptors 1.1, in addition to updates to all its other components.

Java EE 6 also represented a major turning point in the architecture of Java EE on two fronts:

- This version introduced annotation-based and programmatic application configuration to complement the traditional XML configuration used for more than a decade.
- This version marked the introduction of the Java EE Web Profile.

To account for the fact that Java EE had become so large (and maintaining and updating certified implementations was becoming increasingly difficult), the Web Profile certification program offered the opportunity to certify Java EE implementations that included only a subset of the entire Java EE platform. This subset included the features deemed to be most critical to a large number of applications and excluded specifications that are used only by a small minority of applications. As of Java EE 6:

- None of the Web Services or Management and Security components are part of the Java EE Web Profile.
- The Web Profile includes everything from Web Application Technologies and everything from Enterprise Application Technologies except Java EE Connector Architecture, JMS, and JavaMail.

It was during the 5-year Java drought that Oracle Corporation bought Sun Microsystems in January 2010. Coupled with the Java SE drought, this brought a whole new set of concerns for the Java community. Oracle was never known for its agility or willingness to cooperate with open source projects, and many people feared Oracle had bought Sun to shut Java down. However, this turned out not to be the case.

Early on, Oracle began reorganizing the Java team, creating communication pipelines with the open source community, and releasing roadmaps for future Java SE and Java EE versions that were more realistic than anything Sun had promised. Work began anew on Java SE 7, which released on

(Oracle's) schedule in June 2011, almost 5 years after Java SE 6. A second Java EE drought ended with the release of Java EE 7 in June 2013, 3 years and 7 months after Java EE 6. Oracle now says it is on track to begin releasing new versions of both platforms every 2 years, on alternate years. It remains to be seen whether that will come to pass.

Understanding the Most Recent Platform Features

Java SE 7 and 8 and Java EE 7 have brought major changes to the language and supporting APIs and resulted in a rejuvenation of Java technologies. You use these new features throughout this book, so this section provides an overview of them.

Java SE 7

Originally, Java SE 7 had a very ambitious feature list, but after acquiring Sun, Oracle quickly admitted that achieving the goals for Java SE 7 would take many, many years. Every feature was the most important feature to some group of users, so the decision was made to defer some of them to future versions. The alternative was to delay the release of Java SE 7 until 2015 or later — an option that was not acceptable.

Java SE 7 included support for dynamic languages as well as compressed 64-bit pointers (for improved performance on 64-bit JVMs). It also added several language features that made developing Java applications more productive. Perhaps one of the most useful changes was *diamonds*, a shortcut for generic instantiation. Prior to Java 7, both the variable declaration and the variable assignment for generic types had to include the generic type arguments. For example, here is a declaration and assignment for a very complex `java.util.Map` variable:

```
Map<String, Map<String, Map<Integer, List<MyBean>>>> map =
        new Hashtable<String, Map<String, Map<Integer, List<MyBean>>>>();
```

Of course, this declaration contains a lot of redundant information. Assigning anything other than a `Map<String, Map<String, Map<Integer, List<MyBean>>>>` to this variable would be illegal, so why should you have to specify all those type arguments again? Using Java 7 diamonds, this declaration and assignment becomes much simpler. The compiler infers the type arguments for the instantiated `java.util.Hashtable`.

```
Map<String, Map<String, Map<Integer, List<MyBean>>>> map = new Hashtable<>();
```

Another common complaint about Java prior to Java 7 is the management of closable resources as it relates to `try-catch-finally` blocks. In particular, consider this nasty bit of JDBC code:

```
Connection connection = null;
PreparedStatement statement = null;
ResultSet resultSet = null;
try
{
    connection = dataSource.getConnection();
    statement = connection.prepareStatement(...);
    // set up statement
    resultSet = statement.executeQuery();
    // do something with result set
}
```

```
catch(SQLException e)
{
    // do something with exception
}
finally
{
    if(resultSet != null) {
        try {
            resultSet.close();
        } catch(SQLException ignore) { }
    }

    if(statement != null) {
        try {
            statement.close();
        } catch(SQLException ignore) { }
    }

    if(connection != null && !connection.isClosed()) {
        try {
            connection.close();
        } catch(SQLException ignore) { }
    }
}
```

Java 7's *try-with-resources* has drastically simplified this task. Any class implementing `java.lang.AutoCloseable` is eligible for use in a try-with-resources construct. The JDBC `Connection`, `PreparedStatement`, and `ResultSet` interfaces extend this interface. When you use try-with-resources as shown in the following example, the resources you declare within the `try` keyword's parentheses are automatically closed in an implicit `finally` block. Any exceptions thrown during this cleanup are added to an existing exception's suppressed exceptions or, if there is no existing exception, are thrown after the resources have all been closed.

```
try(Connection connection = dataSource.getConnection();
    PreparedStatement statement = connection.prepareStatement(...))
{
    // set up statement
    try(ResultSet resultSet = statement.executeQuery())
    {
        // do something with result set
    }
}
catch(SQLException e)
{
    // do something with exception
}
```

Another improvement made to try-catch-finally is the addition of *multi-catch*. As of Java 7 you can now catch multiple exceptions within a single `catch` block, separating the exception types with a single pipe. For example:

```
try
{
    // do something
```

```
}
catch(MyException | YourException e)
{
    // handle these exceptions the same way
}
```

One caveat to keep in mind is that you can't multi-catch two or more exceptions such that one inherits from another. For example, the following is prohibited because `FileNotFoundException` extends `IOException`:

```
try {
    // do something
} catch(IOException | FileNotFoundException e) {
    // handle these exceptions the same way
}
```

Of course, this can easily be considered a matter of common sense. In this case, you would simply catch `IOException`, which would catch both types of exceptions.

A few other miscellaneous language features in Java 7 include *binary literals* for bytes and integers (you can write the literal `1928` as `0b11110001000`) and *underscores in numeric literals* (you can write the same literals as `1_928` and `0b111_1000_1000`, if desired). In addition, you can finally use `String`s as `switch` arguments.

Java EE 7

Java EE 7, released on June 12, 2013, contains a number of changes and new features. You'll cover many of these new features throughout this book, so they are not detailed here. In summary, the changes to Java EE 7 are as follows:

- JAXB was added to Java SE 7 and is no longer included in Java EE.
- Batch Applications for the Java Platform 1.0 and Concurrency Utilities for Java EE 1.0 were added to Enterprise Application Technologies.
- Web Application Technologies picked up Java API for WebSockets 1.0 (which you learn about in Chapter 10) and Java API for JSON Processing 1.0.
- The Java Unified Expression Language has been significantly expanded to include lambda expressions and an analog of the Java SE 8 Collections Stream API. (You learn more about this in Chapter 6.)
- The Web Profile was expanded slightly to include specifications more likely to be required in common web applications: JAX-RS, Java API for WebSockets, and Java API for JSON Processing.

Java SE 8

The new features in Java SE 8 can come in very handy as you work the examples in this book. Perhaps most visible is the addition of *lambda expressions* (unofficially known as *closures*). Lambda expressions are anonymous functions that are defined, and possibly called, without being assigned

a type name or bound to an identifier. Lambda expressions are particularly useful for anonymously implementing those one-method interfaces that are so common in Java applications. For example, a `Thread` that was previously instantiated with an anonymous `Runnable` like this:

```
public String doSomethingInThread(String someArgument)
{
    ...
    Thread thread = new Thread(new Runnable() {
        @Override
        public void run()
        {
            // do something
        }
    });
    ...
}
```

can now be simplified with a lambda expression:

```
public String doSomethingInThread(String someArgument)
{
    ...
    Thread thread = new Thread(() -> {
        // do something
    });
    ...
}
```

Lambda expressions can have arguments, return types, and generics. And where desired, you can use a *method reference* instead of a lambda expression to pass a reference to an interface-matching method. The following code is also equivalent to the previous two instantiations of `Thread`. You can also assign method references and lambda expressions to variables.

```
public String doSomethingInThread(String someArgument)
{
    ...
    Thread thread = new Thread(this::doSomething);
    ...
}

public void doSomething()
{
    // do something
}
```

One of the biggest complaints among Java users since its earliest days is the lack of a decent date and time API. `java.util.Date` has always been rife with problems, and the addition of `java.util.Calendar` just made many problems worse. Java SE 8 finally addresses that with JSR 310, a new date and time API. This API is based largely on Joda Time, but with improvements to the underlying architecture to fix problems in it that the Joda Time inventor pointed out. This API is a revolutionary addition to the Java SE platform APIs and finally brings a powerful and well-designed date and time API to Java.

A Continuing Evolution

As you can tell, the Java SE and EE platforms were born together and have evolved hand-in-hand for nearly two decades. It's probable that they will continue to evolve together for many years or decades to come. You should be fairly familiar with Java SE, but it's possible you know absolutely nothing about using Java EE. It's also possible you're familiar with older Java EE versions but want to learn more about the new features in Java EE.

Part I of this book teaches you about the most important features in Java EE, including:

- Application servers and web containers (Chapter 2)
- Servlets (Chapter 3)
- JSPs (Chapters 4, 6, 7, and 8)
- HTTP sessions (Chapter 5)
- Filters (Chapter 9)
- WebSockets (Chapter 10).

UNDERSTANDING THE BASIC WEB APPLICATION STRUCTURE

A lot of components go into making a Java EE web application. First, you have your code and the third-party libraries it depends on. Then you have the deployment descriptor, which includes instructions for deploying and starting your application. You also have the `ClassLoaders` responsible for isolating your application from other web applications on the same server. Finally, you must package your application somehow, and for that you have WAR and EAR files.

Servlets, Filters, Listeners, and JSPs

Servlets are a key component of any Java EE web application. Servlets, which you learn about in Chapter 3, are Java classes responsible for accepting and responding to HTTP requests. Nearly every request to your application goes through a Servlet of some type, except those requests that are erroneous or intercepted by some other component. A filter is one such component that can intercept requests to your Servlets. You can use filters to meet a variety of needs, from data formatting, to response compression, to authentication and authorization. You explore the various uses of filters in Chapter 9.

As with many other different types of applications, web applications have a life cycle. There are both startup and shutdown processes, and many different things happen during these stages. Java EE web applications support various types of listeners, which you learn about throughout Parts I and II. These listeners can notify your code of multiple events, such as application startup, application shutdown, HTTP session creation, and session destruction.

Perhaps one of the most powerful Java EE tools at your disposal is the JavaServer Pages technology, or JSP. JSPs provide you with the means to easily create dynamic, HTML-based graphical user interfaces for your web applications without having to manually write `String`s of HTML to an `OutputStream` or `PrintWriter`. The topic of JSPs encompasses many different facets, including the

JavaServer Pages Standard Tag Library, the Java Unified Expression Language, custom tags, and internationalization and localization. You will spend significant time on these features in Chapter 4 and Chapters 6 through 9.

Of course, there are many more features in Java EE than just Servlets, filters, listeners, and JSPs. You will cover many of these in this book, but not all of them.

Directory Structure and WAR Files

Standard Java EE web applications are deployed as WAR files or "exploded" (unarchived) web application directories. You should already be familiar with *JAR*, or *Java Archive*, files. Recall that a JAR file is simply a ZIP-formatted archive with a standard directory structure recognized by JVMs. There is nothing proprietary about the JAR file format, and any ZIP archive application can create and read JAR files. A *Web Application Archive*, or *WAR*, file is the equivalent archive file for Java EE web applications.

All Java EE web application servers support WAR file application archives. Most also support exploded application directories. Whether archived or exploded, the directory structure convention, as shown in Figure 1-2, is the same. Like a JAR file, this structure contains classes and other application resources, but those classes are not stored relative to the application root as in a JAR file. Instead, the class files live in `/WEB-INF/classes`. The `WEB-INF` directory stores informational and instructional files that Java EE web application servers use to determine how to deploy and run the application. Its `classes` directory acts as the package root. All your compiled application class files and other resources live within this directory.

Unlike standard JAR files, WAR files can contain bundled JAR files, which live in `/WEB-INF/lib`. All the classes in the JAR files in this directory are also available to the application on the application's classpath. The `/WEB-INF/tags` and `/WEB-INF/tld` directories are reserved for holding JSP tag files and tag library descriptors, respectively. You'll explore the topic of tag files and tag libraries thoroughly in Chapter 8. The `i18n` directory is not actually part of the Java EE specifications, but it is a convention that most application developers follow for storing internationalization (i18n) and localization (L10n) files.

You probably also noticed the presence of two different `META-INF` directories. This can be a source of confusion for some developers, but if you remember the simple classpath rules, you can easily differentiate the two. Like JAR file `META-INF` directories, the root-level `/META-INF` directory contains the application manifest file. It can also contain resources for specific web containers or application servers. For example, Apache Tomcat (which you'll learn about in Chapter 2) looks for and uses a `context.xml` file in this directory to help customize how the application is deployed in Tomcat. None of these files

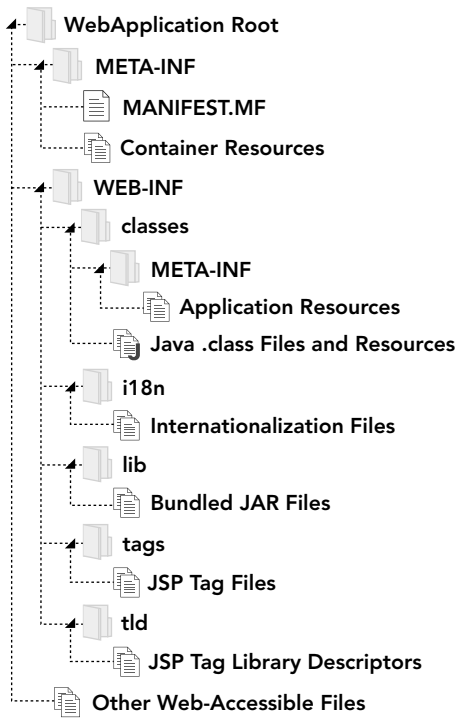

FIGURE 1-2

are part of the Java EE specification, and the supported files can vary from one application server or web container to the next.

Unlike JAR files, the root-level `/META-INF` directory is *not* on the application classpath. You cannot use the `ClassLoader` to obtain resources in this directory. `/WEB-INF/classes/META-INF`, however, is on the classpath. You can place any application resources you desire in this directory, and they become accessible through the `ClassLoader`. Some Java EE components specify files that belong in this directory. For example, the Java Persistence API (which you'll learn about in Part III of this book) specifies two files — one named `persistence.xml` and another `orm.xml` — that live in `/WEB-INF/classes/META-INF`.

Most files contained within a WAR file or exploded web application directory are resources directly accessible through a URL. For example, the file `/bar.html` relative to the root of an application deployed to `http://example.org/foo` is accessible from `http://example.org/foo/bar.html`. In the absence of any filter or security rules to the contrary, this holds true for *all* resources in your application except those resources under the `/WEB-INF` and `/META-INF` directories. The files in these directories are protected resources that are not accessible via URL.

The Deployment Descriptor

The deployment descriptor is the metadata that describes the web application and provides instructions to the Java EE web application server for deploying and running the web application. Traditionally, all this metadata came from the deployment descriptor file, `/WEB-INF/web.xml`. This file contains definitions for Servlets, listeners, and filters, and configuration options for HTTP sessions, JSPs, and the application in general. Servlet 3.0 in Java EE 6 added the ability to configure web applications using annotations and a Java configuration API. It also added the notion of web fragments — JAR files within your application can contain Servlets, filters, and listeners configured in `/META-INF/web-fragment.xml` deployment descriptors within the necessary JAR files. Web fragments can also use annotations and the Java configuration API.

This change to the deployment of web applications in Java EE 6 added significant complexity to the task of organizing this process. To ease this complexity, you can configure the order of your web fragments so that they are scanned and activated in a specific sequence. This happens one of two ways:

- Each web fragment's `web-fragment.xml` file can contain an `<ordering>` element that uses nested `<before>` and `<after>` tags to control whether the web fragment activates before or after other web fragments. These tags contain nested `<name>` elements to specify the name of another fragment relative to which the current fragment should be ordered. `<before>` and `<after>` can alternatively contain nested `<others>` elements to indicate that the fragment should activate before or after any other fragments not specifically named.
- If you didn't create a particular web fragment and don't have control over its contents, you can still control the order of your web fragments within your application's deployment descriptor. The `<absolute-ordering>` element in `/WEB-INF/web.xml`, together with its nested `<name>` and `<others>` elements, configures an absolute order for bundled web fragments that overrides any order instructions that come with the web fragments.

By default, Servlet 3.0 and newer environments scan web applications and web fragments for Java EE web application annotations for configuring Servlets, listeners, filters, and more. You can disable this scanning and disable annotation configuration by adding the attribute `metadata-complete="true"` to the root `<web-app>` or `<web-fragment>` elements as needed. You can also disable all web fragments in your application by adding `<absolute-ordering />` (without any nested elements) to your deployment descriptor.

You learn more about the web application deployment descriptor and annotation configuration throughout Part I of the book. In Part II, you explore the container initializer and programmatic configuration with the Java API, and see how it can make bootstrapping Spring Framework easier and testable.

Class Loader Architecture

When working with Java EE web applications, it's essential to understand the `ClassLoader` architecture because it differs from the architecture to which you are accustomed in standard Java SE applications. In a typical application, the `java.*` classes that come with the Java SE platform are loaded in a special root `ClassLoader` that cannot be overridden. This is a security measure that prevents malicious code from, for example, replacing the `String` class or redefining `Boolean.TRUE` and `Boolean.FALSE`.

After this `ClassLoader` comes the extension `ClassLoader`, which loads classes from the extensions JARs in the JRE installation directory. Finally, the application `ClassLoader` loads all other classes in the application. This forms a hierarchy of `ClassLoader`s, with the root serving as the earliest ancestor for all `ClassLoader`s. When a lower-level `ClassLoader` is asked to load a class, it always delegates to its parent `ClassLoader` first. This continues up until the root `ClassLoader` is checked. With the exception of the root `ClassLoader`, a `ClassLoader` loads a class from its collection of JARs and directories *only* if its parent `ClassLoader` first fails to find the class.

This method of class loading is called the *parent-first class loader delegation model*, and although it works great for many types of applications, it is not ideal for most Java EE web applications. A server that runs Java EE web applications is typically extraordinarily complex and a number of vendors could provide its implementation. The server could use some of the same third-party libraries that your application uses, but they may be of conflicting versions. In addition, different web applications could also provide conflicting versions of the same third-party libraries, leading to even more problems. To solve these problems, you need a *parent-last class loader delegation model.*

In Java EE web application servers, each web application is assigned its own isolated `ClassLoader` that inherits from the common server `ClassLoader`. By isolating the applications from each other, they cannot access each other's classes. This not only eliminates the risk of conflicting classes, but it also serves as a security measure preventing web applications from interfering with or harming other web applications. In addition, a web application `ClassLoader` (typically) asks its parent to load a class only if it can't load the class itself first. In this way, the class loading is delegated to the parent last instead of the parent first, and web application classes and libraries are preferred over those that the server supplies. To maintain the protected status of bundled Java SE classes, web application `ClassLoader`s still check the root `ClassLoader` before attempting to load any classes. Although this delegation model is more preferable for web applications in nearly

all cases, there are still rare circumstances in which it is not appropriate. For this reason, Java EE-compliant servers provide the capability of changing the delegation model from parent-last back to parent-first.

Enterprise Archives

You've learned about WAR files, but there's another type of Java EE archive that you should know about: *EAR* files. An *Enterprise Archive* is a collection of JAR files, WAR files, and configuration files compressed into a single, deployable archive (in ZIP format, just like JARs and WARs).

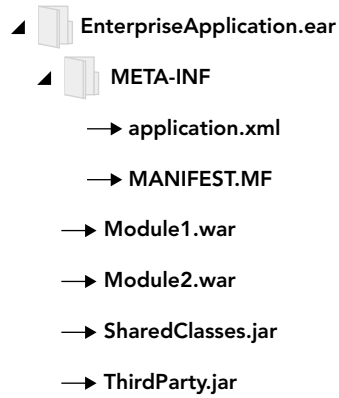

FIGURE 1-3

Figure 1-3 shows a sample EAR file. As with a WAR file, the root `/META-INF` directory contains the archive manifest and is not available to the application classpath. The `/META-INF/application.xml` file is a special deployment descriptor that describes how to deploy the various components included within the EAR file. At the root level of an EAR file are all the web application modules included within it — one WAR file for each module. There is nothing special about these WAR files; they can have all the same contents and features as a normal, standalone WAR file. The EAR file can also contain JAR libraries, which can serve many purposes. The JAR files can contain Enterprise JavaBeans declared in the `/META-INF/application.xml` deployment descriptor, or they can be simple third-party libraries that two or more WAR modules share within the enterprise archive.

As you might have figured, enterprise archives also come with their own `ClassLoader` architecture. Typically, an additional `ClassLoader` is inserted into the hierarchy between the server `ClassLoader` and the web application `ClassLoader`s assigned to each module. This `ClassLoader` isolates the enterprise application from other enterprise applications but enables multiple modules in a single EAR to share common libraries contained within the EAR. This new `ClassLoader` can use either the parent-last (default) or parent-first delegation models. The web application `ClassLoader`s can then either delegate parent-first (enabling EAR library classes to take precedence) or parent-last (enabling WAR classes to take precedence).

Although it is useful to understand enterprise archives, they are a feature of the full Java EE specification, and most web container-only servers (such as Apache Tomcat) do not support them. As such, they are not discussed further in this book.

> **WARNING** *The* `ClassLoader` *examples described in this section are just that — examples. Though the Java EE specifications do describe parent-first and parent-last class loading, different implementations achieve these models in different ways, and each server could have certain nuances that might cause problems depending on your needs. You should always read the documentation of the server you choose so that you can determine whether the* `ClassLoader` *architecture of that particular server is appropriate for you.*

SUMMARY

In this chapter you explored the histories of the Java Platform, Standard Edition and Java Platform, Enterprise Edition and learned how the two platforms evolved together over the last 19 years. You were briefly introduced to some of the topics covered in this book — Servlets, filters, listeners, JSPs, and more — and saw how Java EE applications are structured, both internally and on the filesystem. You then learned about web application archives and enterprise archives and how they serve as vessels for transporting and deploying Java EE applications.

The rest of the book explores these topics in much greater detail, answering the many questions that you likely have after reading the last several pages. In Chapter 2 you take a closer look at application servers and web containers, what they are, and how to choose one for your purposes. You also learn how to install and use Tomcat for the examples in this book.