

Understanding Java and the J2EE Platform

Java 2 Enterprise Edition, or J2EE, is a package of specifications aligned to enable the development of multi-tier enterprise applications. The specifications outline the various components needed within a J2EE enterprise system, the technologies for accessing and providing services, and even the roles played during the development, deployment, and runtime lifecycle. The combination of these specifications introduced faster and more streamlined development processes, to the software industry, that have been mapped onto common software methodologies such as RUP, XP, and others.

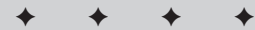
J2EE has fast become the *de facto* standard for developing and deploying enterprise systems. It represents Sun's attempt to take their Java mantra of "Write Once, Run Anywhere" to the next level and make it "Write Once, Deploy Anywhere." While using it is not as easy as dropping new code fragments into existing code, J2EE has made significant strides in easing the burden on the developers and deployers of a system.

This chapter will introduce J2EE. At the time of this writing J2EE 1.4 is in beta but it should be in public release by the time this book is published.

Reviewing a Brief History of Java

In 1995, Sun released Java, a fully object-oriented programming language. While most of the concepts within Java were not new, it did meld many features, such as memory management and garbage collection from Smalltalk and the syntax of C/C++, into a new easy-to-learn programming language.

CHAPTER



In This Chapter

Reviewing a brief history of Java

Understanding J2SE

Examining the origin of J2EE

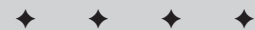
Working with the Model-View-Controller (MVC)

Understanding the J2EE APIs

Discovering what's new in J2EE 1.4

Looking toward the future of J2EE

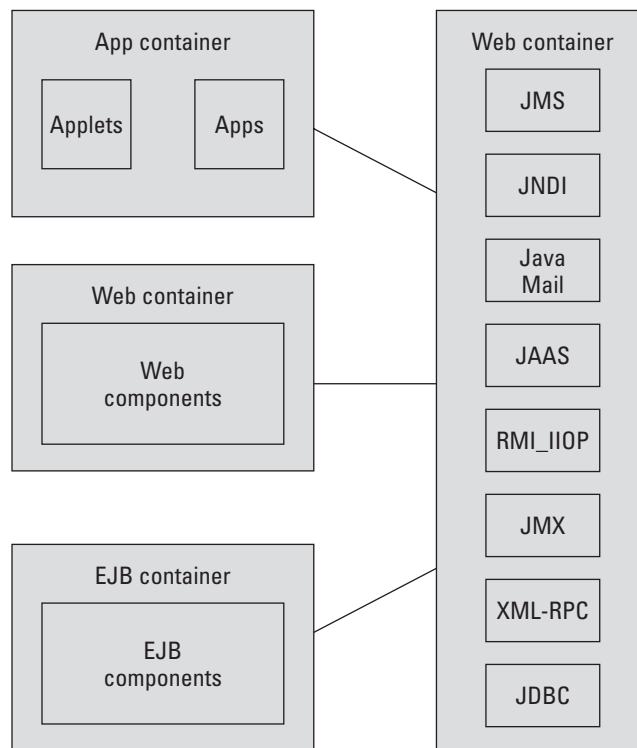
Understanding the Java Community Process



4 Part I ♦ Introduction

Java brought the concept of a virtual machine into the mainstream. Traditionally, programs written in a particular language, such as C, were compiled directly for the operating system on which the program would run. In order for companies to support multiple-target runtime environments, a new build environment became necessary for each target—for example, Windows95, HP-UX, Solaris, and so on. However, Java is not compiled completely, but instead is compiled to an intermediary stage as Java bytecodes. At runtime, the Java bytecodes are executed within a *virtual machine*, which is a piece of software that interprets the bytecodes in runtime into the native binary for the operating system.

The virtual machine is responsible for allocating and releasing memory, ensuring security, and optimizing the execution of the Java bytecodes, among other functions. This has indeed created a new market simply for virtual machines for various operating systems. As long as a virtual machine is available for a particular operating system, the Java bytecodes should be able to be executed on it, assuming that all the Java APIs are implemented. Figure 1-1 shows the stages that Java code must go through before being executed on a target machine.

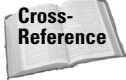


* Not all APIs shown

Figure 1-1: Java Virtual Machine compilation

Understanding J2SE

Around 1998, Sun updated the Java specification and introduced Java 1.2 along with the accompanying libraries, making Java not only a language, but also a platform — Java 2 Standard Edition (J2SE). Prior to the release of J2SE, Java had gone through the number of revisions and new libraries were not necessarily introduced in a concerted manner, making it difficult for developers to understand. Prior to the J2SE, the Java Development Kit (JDK) was the primary package that was installed, and developers would choose which additional libraries they would want such as Java Database Connectivity (JDBC) or Swing. This led to inconsistent environments making it difficult to port code since the deploying party would not be guaranteed of the libraries on the deployment platform.



JDBC is the topic of Chapter 18.

With J2SE, Sun attempted to fix the problem by bundling the various libraries into a single unit. J2SE provided libraries for GUI support, networking, database access, and more. J2SE is also the foundation for the J2EE.

Examining the Origin of (J2EE)

J2SE was sufficient for developing stand-alone applications, but what was missing was a standard way to develop and deploy enterprise applications — one similar to the standard method for using the Common Object Request Broker Architecture (CORBA). While J2SE already included enterprise-level APIs such as Remote Method Invocations (RMI), too much was still left undefined — such as persistence, transaction management, security, and so on. This resulted in a plethora of architectures being developed.

J2EE, introduced in 1998, defines a multi-tier architecture for enterprise information systems (EIS). By defining the way in which multi-tier applications should be developed, J2EE reduces the costs, in both time and money, of developing large-scale enterprise systems. Figure 1-2 illustrates the J2EE architecture, highlighting the new additions within the 1.4 release.

The J2EE platform specifies the logical application components within a system and defines the roles played in the development process.

6 Part I ♦ Introduction

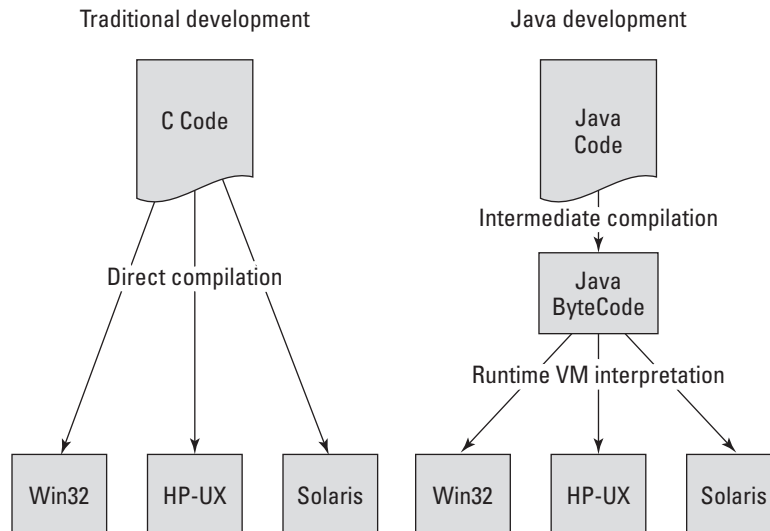


Figure 1-2: J2EE Architecture (source: Javasoft)

Application components

Four application components are defined within the J2EE platform. They are as follows:

- ♦ Application clients (Standalone Java clients)
- ♦ Applets (Java code which executes within a browser)
- ♦ Web components (JSPs, Servlets)
- ♦ Server components (EJBs, J2EE API implementations)

A product does not need to support all types of components; the norm is to provide an implementation to support a particular component type. However, all components are similar in that they run within a *container*. The container is responsible for providing the runtime environment, the mechanism for identifying and understanding the file formats used for deployment, and the standard services for application components to use.

The four application components are discussed in the following sections.

Application clients

Clients are generally stand-alone applications written in Java. They run within a virtual machine and can use the J2EE standard services to access components located within another tier. The J2EE standard services are usually provided on the client via an installation of J2SE, or along with the distribution of the application itself.

Applets

Applets are similar to application clients, but execute within a Web browser. Initially applets garnered extensive attention, as they were seen as a means of making Web pages more dynamic. Most Web browsers have an embedded Java Virtual Machine (JVM); however, the Java plugin can be used to force the browser to use a particular version of JVM.

Web components

Although the term can be misleading, Web components do not execute on the client side. Web components are server-side components, generally used to provide the presentation layer to be returned to a client. Two types of Web components exist: Java Server Pages (JSPs) and Java servlets. Very basically, JSPs are similar to regular HTML pages but contain embedded Java code while Java servlets are Java classes that use Java's I/O application programming interfaces (APIs) to output HTML to the client. Both JSPs and servlets can be used to output other format types.

Server components

Server components come in the form of Enterprise JavaBeans (EJBs). EJBs execute within a container that manages the runtime behavior of the EJB. EJBs are usually where the business logic for an enterprise system resides.

Roles

The *roles* specified within the J2EE are those played during the development and deployment cycles of an enterprise application. While the roles are distinct, in reality multiple roles tend to be filled by the same organization. The following roles are discussed in this section:

- ♦ J2EE product provider
- ♦ Application component provider
- ♦ Application assembler
- ♦ Deployer
- ♦ System administrator
- ♦ Tool provider
- ♦ System component provider

The J2EE product provider

A J2EE product provider is a company that provides a product that implements a part of the J2EE specification. For example, one company may provide a product that implements the J2EE container for EJBs, and another may provide a product that provides an implementation for a JMS server.

8 Part I ♦ Introduction

The application component provider

An application component provider is a developer who creates a component that is intended to reside within one of the J2EE containers. The application component provider develops application components adhering to the J2EE API specifications with the intention that the component will be deployed within a J2EE Server. This enables a developer to select a different J2EE product provider without modifying the component. Application component providers develop a range of components, including EJBs, HTML pages, and other Web components.

The application assembler

An application assembler generally uses various application components to create a single application for distribution. Generally, in a large project, one team will be responsible for developing the Web components, another for the business-logic components, and perhaps another for the data-object components. The application assembler would package the various components and then distribute them as an enterprise archive (.ear) file.

The deployer

The deployment of an enterprise application nearly always requires a different configuration for each rollout. J2EE has taken this into consideration by specifying the role of deployer. The deployer is responsible for configuring the applications developed by the application assembler for execution within a platform provided by the J2EE product provider.

The system administrator

A *system administrator* generally uses tools provided by a tool provider to monitor the runtime environment and to ensure that services are performing optimally. Various tools are available on the market, ranging from those which allow for monitoring the system as a whole, to runtime inspection on individual services to help determine where bottlenecks may reside.

The tool provider

The J2EE specification also provides tools to make development easier and to monitor the runtime environment. Tools vary from integrated development environments to runtime-performance products.

The system-component provider

Many system components are available for the J2EE architecture. The J2EE architecture provides ways to introduce these new components for accessing services such as existing messaging systems, transaction services, and others, such as billing systems that may be industry-specific. Using the connector architecture is one way to introduce these new components.

In addition to specifying the lifecycle roles, the J2EE also recommends the usage of the model-view-controller (MVC) design pattern to ease the burden on developing long-lived applications.

Working with the Model-View-Controller

The MVC paradigm provides a pattern for separating the presentation logic (view), business logic (control), and data objects (model). J2EE's architecture maps onto the MVC nicely. Typically, entity beans are used to provide the model logic, while a mix of entity beans and session beans are used to provide the control logic, and Web components are used to implement both control and presentation logic. In practice, however, the separation of the three types of logic is not as distinct, and additional patterns are often needed to support the development cycle. Figure 1-3 shows how the three different logical functional blocks work together.

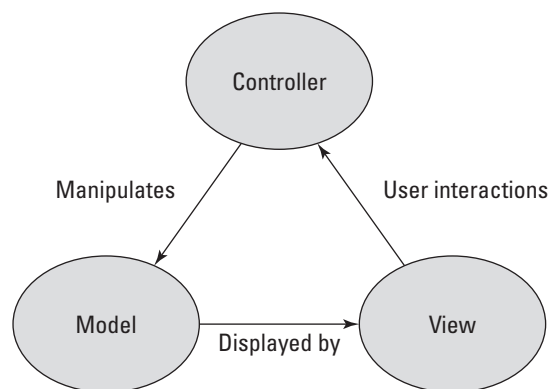


Figure 1-3: MVC pattern

Sun has provided guidelines in the form of Java BluePrints. A sample application, Java Adventure Builder, has been developed specifically for J2EE 1.4 and you can download it from <http://www.javasoft.com>.

The model

The *M* in MVC refers to the data object model. For example, in an airline ticketing service you may have the concept of a booking, which in the real world is represented by a paper ticket. The model deals with issues such as how the booking is represented within the software system, where it is persisted, and how it is accessed. For example, the booking may be held within a relational database within

10 Part I ♦ Introduction

a table named `Bookings` with the fields `PassengerName`, `DepartureCity`, `DestinationCity`, `TravelDate`, and `DepartureTime`. This data may be accessed via JDBC using Entity Beans (which we will discuss in detail later in the chapter).



Entity beans and JDBC are discussed in Chapters 16 and 18, respectively.

The view

The view is responsible for presentation issues. It handles how the client will see the application, and so HTML issues are usually dealt with here. However, other markup languages such as Wireless Markup Language (WML) and Extensible Markup Language (XML) are increasingly being used to support more varied types of clients. The Booking example may be displayed in various ways. For example, on a wireless device only the most relevant information might be displayed due to the limited screen size. In fact, the term *view* may be misleading, implying that it is meant for visual display only; the view may also be used to present the model via an audio interface if desired. The method in which the model is presented is abstracted from the underlying data.

The control

The control part of the paradigm deals with the business logic of the application. It handles how and when a client interacting with the view is able to access the model. The control layer usually interacts with authorization and authentication services, other J2EE services, and external systems to enforce the business rules to be applied to the application. In our Booking example, the control would determine whether the view can actually display the model. This may be based on whether the user is logged in, if he or she has appropriate authorization and so on. It would also hold the business logic of what to do if the user attempts to view a booking that no longer exists — for example, should an error be presented to the user? Should the user be prompted with a screen asking for additional information? These are rules that change within the business but they do not necessarily force a change on the view or model.

To support the MVC, the J2EE architecture also provides a varied set of APIs to help facilitate the separation between the model, view, and control functional blocks within an application.

Understanding J2EE APIs

The J2EE specification stipulates a number of different APIs, not all of which are mandatory for every application component type. In some cases, for example the Java Database Connectivity (JDBC) API, the API may only be mandatory for the some components, while other APIs may be optional for all components.

The J2EE specifies a set of standard services, which are listed in the next section with an accompanying chart. The standard services have been used within other APIs, such as EJB, JSP, and Java servlets.

J2EE standard services

Included in the J2EE are the following standard services. Some of these services are provided by J2SE, while others are termed “optional packages,” meaning that they are optional within a J2SE implementation, but not within a J2EE implementation.

- ♦ **HyperText Transfer Protocol/HyperText Transfer Protocol Secure sockets (HTTP/HTTPS)** — Both of these protocols must be supported by J2EE servers.
- ♦ **Java Transaction API (JTA) 1.0** — JTA provides an interface for demarcating transactions. It enables the developer to attach transaction-processing systems.
- ♦ **Remote Method Invocation to Internet Inter-ORB Protocol (RMI-IIOP)** — EJB components use this service for communication. The underlying IIOP protocol can be used to access compliant CORBA objects residing in external systems.
- ♦ **Java Database Connectivity (JDBC) 3.0** — JDBC provides a Java interface for executing SQL statements without understanding the specifics of the underlying data store. JDBC 3.0 merged with the previously optional JDBC Extension package.
- ♦ **Java Message Service (JMS) 1.1** — JMS is an asynchronous messaging service that enables the user to send and receive messages via point-to-point or publish-subscribe models.
- ♦ **JavaMail 1.3** — JavaMail enables the delivery and retrieval of e-mail via message transports and message stores, respectively.
- ♦ **Java Naming and Directory Interface (JNDI) 1.2** — JNDI is used to access directories such as Lightweight Directory Access Protocol (LDAP). Typically, components use the API to obtain references to other components.
- ♦ **JavaBeans Activation Framework (JAF) 1.0** — JavaMail uses JAF to handle various different Multipurpose Internet Mail Extensions (MIME) types that may be included within an e-mail message. It converts MIME byte streams into Java objects that can then be handled by assigned JavaBeans.
- ♦ **Java API for XML Parsing (JAXP) 1.2** — JAXP includes both Simple API for XML (SAX) and Document Object Model (DOM) APIs for manipulating XML documents. The JAXP API also enables Extensible Stylesheet Language Transformation (XSLT) engines to be plugged in.
- ♦ **J2EE Connector Architecture 1.5** — The connector architecture specifies a mechanism by which to attach new resource adaptors to a J2EE server. Resource adaptors can be used to provide access to services that are not specified through other APIs.

12 Part I ♦ Introduction

- ♦ **Security Services** — These are provided via Java Authentication and Authorization Service (JAAS) 1.0, which allows J2EE servers to control access to services.
- ♦ **Web Services** — Support for Web services is provided via Simple Object Access Protocol (SOAP) for attachments; API for Java (SAAJ) 1.1 for handling of SOAP messages; Java API for XML Registries (JAXR) 1.0 for access to Universal Description, Discovery, and Integration (UDDI); and Java API for XML-based RPC (JAX-RPC) 1.0 to specify how clients can use Web services.
- ♦ **Management** — The Java 2 Platform, Enterprise Edition Management API 1.0, and Java Management Extensions (JMX) 1.2 are used to provide management support for querying a server during runtime.
- ♦ **Deployment** — The Java 2 Platform, Enterprise Edition Deployment API 1.1 allows tools to plug into a J2EE server for deployment purposes.
- ♦ **Java Authorization Service Provider Contract for Containers (JACC) 1.0** — JACC is the interface between application servers and authorization policy providers.

Table 1-1 gives a list of the various J2EE Standard Services APIs and indicates which APIs are required for each component type.

Table 1-1
J2EE Standard Services APIs

<i>Standard Service</i>	<i>Version</i>	<i>App Client</i>	<i>Web</i>	<i>EJB</i>
HTTP/HTTPS	1.0, SSL 3.0, TLS 1.0	Required	Required	Required
JTA	1.0	Not Required	Required	Required
RMI-IIOP		Required	Required	Required
JDBC	3.0	Required	Required	Required
JMS	1.1	Required	Required	Required
JavaMail	1.3	Required	Required	Required
JNDI	1.2	Required	Required	Required
JAF	1.0	Required	Required	Required
JAXP	1.2	Required	Required	Required
Connecture Architecture	1.5	Not Required	Required	Required
JAAS	1.0	Required	Required	Required

<i>Standard Service</i>	<i>Version</i>	<i>App Client</i>	<i>Web</i>	<i>EJB</i>
SAAJ	1.2	Required	Required	Required
JAXR	1.0	Required	Required	Required
JAX-RPC	1.1	Required	Required	Required
JMX	1.2	Required	Required	Required
JACC	1.0	Not Required	Required	Required

Application component APIs

The standard services described in the previous section are used to provide additional J2EE application-component specifications as Web and server components. The following is a list of the application component APIs specified in J2EE.

- ♦ **Enterprise JavaBeans (EJB) 2.1** — EJBs are similar to CORBA components and typically encapsulate business-logic code or data-model code. They execute within a container, which manages their interactions with other components, including resources and security. Three different types of EJBs exist:
 - Entity beans
 - Message-driven beans
 - Session beans, which come in two flavors — either stateless or stateful.
- ♦ **Java Servlet 2.4** — Servlets are classes that reside on the server and are typically used to respond to incoming requests via HTTP. They are often used to return the presentation layer to a client.
- ♦ **JavaServer Pages (JSP) 2.0** — JSP pages are very similar to HTML pages, except that they have embedded Java code. The pages are parsed and executed on the server prior to being returned to the requesting client. JSPs can make use of additional APIs, such as JSP tag extensions, to allow for more complex logic.


Note

Not all of the preceding APIs will be discussed in this book, as many of them are fairly straightforward.

Discovering What's New in J2EE 1.4

Version 1.4 introduces significant improvements in J2EE's support for Web services and XML. Until now J2EE lagged behind the recently introduced Microsoft .NET, which provided extensive support for XML from its initial release in 2000. However,

14 Part I ♦ Introduction

J2EE 1.4 has dramatically changed that with the introduction of XML-RPC, JAXR, SAAJ, and modifications within the Enterprise JavaBeans (EJB) specification, as well as with the manner in which new libraries are deployed. XML and support for Web services are now an integral part of J2EE, providing another level of abstraction for the decoupling of systems.

In addition, J2EE 1.4 has improved tools support via the J2EE Management and J2EE Deployment APIs, and many of the other individual APIs have been enhanced as well. The following chapters will discuss the various APIs and their capabilities in greater detail.

Looking toward the Future of J2EE

Java has progressed incredibly since its inception, as has J2EE. While the needs of today and those of the near future are being met by the current release of J2EE, it is not complete, nor will it ever be. Like all enterprise systems, J2EE is constantly evolving.

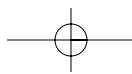
Some of the innovations planned for the future are an XML data-binding API, enhanced security APIs, support for JDBC RowSets, and more. For a full list of potential future enhancements, review “Future Directions” in the specification document. Alternatively, you can follow the Java Community Process, which is discussed next.

Understanding the Java Community Process (JCP)

The JCP is an initiative similar to a standardization body, put in place by Sun to allow for an unbiased approach to the development of Java. While it is not an official standards body, it is open to the public. All the Java APIs, along with the various distributions (J2EE, J2SE, and J2ME), are covered with the JCP.

Generally, the process works as follows:

1. A member (or group of members) within the JCP submits a Java Specification Request (JSR) which requests either a new specification or modifications to an existing one.
2. Following the acceptance of the JSR by the JCP, an expert group is formed and specification development begins.
3. Final acceptance of the specification is made via a vote by an executive committee.



The JCP Web site lists over 500 members working on 90 outstanding JSRs as of the start of 2003. If you would like to be a part of the ongoing development of Java, sign up and start contributing to one of the existing JSRs at <http://www.jcp.org>.

Summary

This chapter has given a brief introduction to Java and to the J2EE platform. It is by no means exhaustive but is more intended to give a basic grasp of the concepts. You learned about Java and the Java Virtual Machine. You took a look at the evolution of the Java platform from J2SE to J2EE and examined the various component types within the J2EE architecture. Using this information, you will be able to take advantage of the following chapters, which will discuss the various APIs and their usage in greater detail.

