Chapter Overview

Topics in This Chapter

- The J2EE Architecture
- J2EE Components
- J2EE Application Server
- Session Bean Overview
- Entity Bean Overview
- Message-Driven Bean Overview
- Local Interfaces
This chapter presents an overview of the J2EE architecture and Enterprise JavaBeans architecture. We’ll begin with the concepts behind the J2EE architecture and how the J2EE components help you design distributed applications. Next, we’ll discuss the J2EE application server and the role of the container in each server. This helps you understand how enterprise beans fit into the big picture.

There’s a lot of terminology that goes along with enterprise beans, so this chapter is a good place to define terms. We’ll start with the concepts behind session beans (stateless and stateful), entity beans, and message-driven beans. We’ll help you understand how entity beans with bean-managed and container-managed persistence interact with a database. You’ll also learn about bean life cycles, entity relationships, interfaces (home, local home, remote, and local) and asynchronous messaging.

This chapter is meant as an overview, so some of the topic discussions will be brief. Others will be explained in more detail. There will, of course, be much more to talk about when we start designing enterprise systems in later chapters. Let’s start with the J2EE architecture, which is the foundation of enterprise beans.
2.1 The J2EE Architecture

The J2EE platform gives you a multitiered application model to develop distributed components. Although any number of tiers is possible, a three-tier architecture is typical. Figure 2–1 shows the approach.

The client machine supports web browsers, Java applets, and stand-alone applications. A client application may be as simple as a command line program running as an administrator client or a graphical user interface created from Java Swing or AWT (Abstract Window Toolkit) components. Regardless, J2EE applications encourage thin clients in the presentation tier. A thin client is a lightweight interface that does not perform database queries, implement business logic, or connect to legacy code. These types of “heavyweight” operations preferably belong to other tiers.

The J2EE server machine is the center of the architecture. This middle tier contains web components and business objects managed by the application server. The web components dynamically process user requests and construct responses to client applications. The business objects implement the logic of a business domain. Both components are managed by a J2EE application server that provides important system services for these components, such as security, transaction management, naming and directory lookups, and remote connectivity. By placing these services under control of the J2EE application server, client components focus only on presentation logic. And, business objects are easier for developers to write. Furthermore, the architecture encourages the separation of business logic from presentation logic (or model from view).

![Three-Tier J2EE Architecture](image)

*Figure 2–1 Three-Tier J2EE Architecture*
The database server machine handles the database back end. This includes mainframe transactions, databases, Enterprise Resource Planning (ERP) systems, and legacy code. Another advantage of the three-tier architecture is that older systems can take on a whole new “look” using the J2EE platform. This is the approach many businesses are taking as they migrate their legacy systems to the web in a modern distributed computing environment.

### 2.2 The J2EE Components

To develop distributed components for the J2EE architecture, you need component technologies, APIs, and administrative tools. Let’s take a look at each of these categories as they apply to the J2EE architecture and see what they offer to developers.

**Enterprise JavaBeans (EJB)**

EJB is a component technology that helps developers create business objects in the middle tier. These business objects (enterprise beans) consist of fields and methods that implement business logic. EJ Bs are the building blocks of enterprise systems. They perform specific tasks by themselves, or forward operations to other enterprise beans. EJ Bs are under control of the J2EE application server.

**Java Servlets**

This component technology presents a request-response programming model in the middle tier. Servlets let you define HTTP-specific servlet classes that accept data from clients and pass them on to business objects for processing. Servlets run under the control of the J2EE application server and often extend applications hosted by web servers.

**JavaServer Pages (JSP)**

A JSP page is a text-based document interspersed with Java code. A JSP engine translates JSP text into Java Servlet code. It is then dynamically compiled and executed. This component technology lets you create dynamic web pages in the middle tier. JSP pages contain static template data (HTML, WML, and XML) and JSP elements that determine how a page constructs dynamic content. The JSP API provides an efficient, thread-based mechanism to create dynamic page content. We provide several JSP clients as example clients to our EJB components in this book. See “Introducing Servlets and JSP” on page 64 in Chapter 3 for a more detailed JSP overview.
Chapter 2  Enterprise JavaBeans Overview

Java Naming Directory Interface (JNDI)

JNDI is a standard Java API for accessing different directory and naming services in the presentation tier or middle tier. JNDI lets you access databases and administered objects (queues, topics, etc.) without knowing their specific names or protocols. This makes JNDI an important step in the design of portable enterprise systems. J2EE application servers use JNDI so that clients can “find” needed distributed components or access environment information (such as database resource names or class names) used to customize EJB behavior.

Java Database Connectivity (JDBC)

JDBC is an API that lets you invoke SQL commands from Java methods in the middle tier. You can use the JDBC API to access a database from either an enterprise bean or from a servlet or JSP page. The JDBC API has an application-level interface for database access, and a service provider interface to attach JDBC drivers to the J2EE platform. In support of JDBC, J2EE application servers manage a pool of database connections. This provides business objects efficient access to database servers. We provide an overview of database fundamentals and JDBC (see “Database Fundamentals” on page 87 and “Introducing JDBC” on page 88 in Chapter 4).

Java Message Service (JMS)

The JMS API lets you perform asynchronous messaging in the presentation and middle tiers. Asynchronous messaging means that clients do not have to wait for a business method to complete. A JMS server stores messages with topic objects for publish-subscribe broadcasts and queue objects for point-to-point communications. JMS lets you implement push/pull technologies in enterprise designs. See “Introducing JMS” on page 372 in Chapter 8 for a JMS overview.

Java Transaction API (JTA)

Under control of the J2EE application server, the JTA provides a standard interface for demarcating transactions in the middle tier. When multiple clients access the same database, it’s important to update the data correctly for each read and write operation. The JTA is useful for marking where dependent database operations occur so that they may be committed (written to the database) if successful or rolled back (undone) when there are errors. The J2EE architecture provides defaults for transaction auto commits and rollbacks.
2.2 The J2EE Components

**Java API for XML (JAXP)**

JAXP lets you read and write text-based XML programs in the presentation and middle tiers. XML is a portable language known to a large number of tools and applications on the web. In the J2EE platform, XML is used extensively with deployment descriptors for J2EE components. As more system services are provided by the J2EE application server (and not coded by developers), developers customize component behavior declaratively using XML.

**JavaMail**

The JavaMail API lets you send and receive e-mail in the presentation and middle tiers. JavaMail has a service provider interface and an application-level interface for application components to send mail. JavaMail is a valuable part of the J2EE platform because it allows J2EE components to send and receive e-mails with different protocols.

**J2EE Connector API**

The Connector API defines a standard for connecting the J2EE platform to systems in the database tier. This includes mainframe transaction processing, database systems, ERP, and legacy applications that are not written in Java. J2EE vendors and systems integrators use the Connector API to create resource adapters, which allow J2EE components to access and interact with a resource manager of another system.

**Java Authentication and Authorization Service (JAAS)**

JAAS extends the Java 2 platform security architecture to support user-based authorization. This provides a way for a J2EE application to authenticate a specific user or a group of users that wish to run an application.

**J2EE Reference Implementation**

The Reference Implementation is a noncommercial product from Sun Microsystems that provides a J2EE application server, web server, relational database (Cloudscape), the J2EE APIs, and a set of development and deployment tools. This implementation is made freely available by Sun for demonstrations, prototyping, and educational use. All of the examples in *Enterprise JavaBeans Component Architecture* have been developed and tested with the Reference Implementation.
Remote Method Invocation (RMI)

RMI, introduced in the Java Development Kit (JDK) 1.1, lets Java programs find and use Java classes residing on remote machines across networks. RMI makes the use of distributed objects within Java programs transparent and easy to use. RMI provides support for the network calls used by Enterprise JavaBeans in the J2EE architecture. RMI hides the nitty-gritty details of network sockets and TCP/IP. It also marshals and unmarshals method arguments and handles exceptions across the network.

Java Interface Definition Language (IDL)

The EJB specification includes a mapping for EJBs to communicate with each other using Common Object Request Broker Architecture (CORBA). Developers use an Interface Definition Language (IDL) to specify interfaces in a Java IDL file. This file defines the methods and fields that must be platform and language independent. The IDL file is compiled with a special compiler that generates the classes necessary to communicate over the network. CORBA and IDL enable Java programs and EJBs to call legacy code written in other languages. This makes it possible, for example, to call member functions in a C++ program running on a Windows machine from an EJB executing on a UNIX system across a network.

2.3 The J2EE Application Server

Now that we’ve introduced the J2EE components, let’s take a closer look at the J2EE platform. Figure 2–2 shows a more detailed view of the J2EE architecture.

Containers and Services

Both the presentation and middle tiers use containers. An application client container runs on the client machine and manages the execution of client programs. In the J2EE server machine, a web container manages JSP and servlet components. The EJB container, which also runs on the J2EE server machine, manages the execution and life cycle of all enterprise beans. The browser is an applet container and runs on the client machine.

Why are containers important in the J2EE application server? Rather than have client programs handle the complex details of transaction and state management, multithreading, and resource pooling, the web and EJB containers provide these services in the middle tier. This arrangement makes thin clients
possible and centralizes the important aspects of distributed computing in a dedicated server or cluster of servers.

All J2EE components, regardless of whether they are web components, enterprise beans, or application components, must be assembled into a J2EE application and deployed in their containers before executing. This assembly process involves specifying container settings for each component that is to be deployed.

Part of the packaging of EJB components involves the creation of a deployment descriptor. Deployment descriptors are XML-based files that describe how the container configures a bean to run on a server. In the deployment descriptor, you may specify settings for security, transactions, JNDI lookups, environment resources, and much more. The deployment descriptor is important because it affects the behavior of an EJB. By changing an EJB’s deployment descriptor, you can make an EJB behave differently. We show you how to customize EJBs through the deployment descriptor (see “Naming Environment Entry” on page 124 in Chapter 4).
Chapter 2  Enterprise JavaBeans Overview

The containers also manage other types of services, such as EJB and servlet life cycles, database connection resource pooling, data persistence, transaction management, and security.

With this introduction to the J2EE architecture and its components, let’s take a closer look at EJB components in the middle tier. We’ll discuss the different types of enterprise beans and how the EJB container manages them in enterprise designs.

2.4 Enterprise JavaBeans

Enterprise JavaBeans are server-side components that encapsulate an application’s business logic. An enterprise bean, for example, might calculate interest payments for a loan application or access a relational database for banking applications. A client that needs information calls business methods in the EJB, which may result in a remote invocation across a network.

EJBs have several benefits. First of all, they help bean developers write large applications with distributed components more easily. Recall that the EJB container provides system-level services to each bean deployed in a J2EE application server. This means bean developers do not have to handle complex issues such as transaction management, resource pooling, security, and multi-threaded programming. Instead, they focus only on business logic in their bean methods.

EJBs also benefit clients in the presentation tier. Clients become thinner, since they implement only presentation logic and use EJBs for their business methods. EJBs are also portable components, which means you can build different enterprise applications with the same EJBs and run them on any J2EE application server.

There are three types of enterprise beans: session beans, entity beans, and message-driven beans. Let’s give you a brief overview of each enterprise bean and discuss how they might be used in an enterprise application. We’ll also discuss bean life cycles and how the EJB container manages each bean.

**Session Beans**

Session beans represent an interactive session with one or more clients. Session beans may maintain state, but only during the time a client interacts with the bean. This means session beans do not store their data in a database after a client terminates. Session beans, therefore, are not persistent.

Session beans come in two flavors: stateless and stateful. Let’s discuss the characteristics of each session bean and suggest when they might be appropriate in an enterprise application.
Stateless Session Beans

The values of any object’s instance variables (fields) define the state of an object. Usually, method calls change an object’s state. A stateless session bean, however, does not keep track of client-specific data. In fact, no instance variable in a stateless session bean stores client-specific data. This unique property allows the EJB container to create a pool of instances, all from the same stateless session bean. Why is this important?

When a client invokes a method of a stateless session bean, the EJB container fetches an instance from the pool. Any instance will do, since the bean does not store any client-specific information. As soon as the method finishes executing, the instance is available for another client’s request. This arrangement makes stateless session beans highly scalable for a large number of clients (a small number of instances can service many clients). It also means better performance. The EJB container does not have to move stateless session beans from memory to secondary storage to free up resources—it simply regains memory and other resources by destroying the instances.

All enterprise beans have different states that they go through during their lifetimes. The life cycle of a bean is managed by the EJB container. Figure 2–3 shows the life cycle for a stateless session bean.

There are only two states in a stateless session bean: a Does Not Exist state and a Ready state. After the container creates an instance of a stateless session bean with `Class.newInstance()`, the container invokes the `setSessionContext()` and `ejbCreate()` methods in the bean. This makes the bean transition

![Figure 2–3 Stateless Session Bean Life Cycle](image_url)
to the Ready state. The bean developer uses the `setSessionContext()` method to access the bean’s context from the container. The container invokes the `ejbCreate()` method to initialize the bean and access resources. Once the bean is in the Ready state, a client may call its business methods. The container calls the bean’s `ejbRemove()` method when it no longer requires a bean instance. This makes the bean return to the Does Not Exist state.

When should you use stateless session beans? Stateless session beans are appropriate when a task is not tied to a specific client. You could, for instance, use a stateless session bean to send an e-mail confirmation or calculate interest payments for loan applications. You could also use a stateless session bean to read data from a database. Such a bean would be useful for generating reports or viewing a collection of items.

**Stateful Session Beans**

In a stateful session bean, the instance variables store client-specific data. Each stateful session bean, therefore, stores the **conversational state** of one client that interacts with the bean. This conversational state is maintained by the bean while clients call its business methods. The conversational state is not saved when the client terminates the session.

The EJB container manages stateful session beans differently from stateless beans. Note that with stateful session beans, it’s not possible for the container to create a pool of instances and share them among multiple clients. Since a stateful session bean stores client-specific data, the container creates a separate bean instance for each client. So that conversational state is not lost, the container saves and restores stateful session beans when moving them between memory and secondary storage. All this means that stateful session beans have more overhead associated with them and are not as scalable as stateless session beans.

The life cycle of a stateful session bean is also more involved, as Figure 2–4 shows. Stateful session beans have three states: a Does Not Exist state, a Ready state, and a Passive state. When a client calls `create()`, the EJB container instantiates the bean with `Class.newInstance()` and calls its `setSessionContext()` and `ejbCreate()` methods. This makes the bean transition to the Ready state where it can accept calls to its business methods. When the client terminates a session, the container calls the `ejbRemove()` method in the bean. The bean returns to the Does Not Exist state where it is marked for garbage collection.

A stateful session bean transitions to the Passive state when the container *passivates* the bean; i.e., the container moves the bean from memory to secondary storage. The container calls the bean’s `ejbPassivate()` method just before passivating a bean. If the bean is in the Passive state when a client calls one of its business methods, the container *activates* the bean. The container restores
the bean in memory before calling the bean’s `ejbActivate()` method. This makes the bean return to the Ready state.

When should you use stateful session beans? In general, any situation where a bean must remember client information between method invocations is a candidate for stateful session beans. A good example on the web is a virtual shopping cart in an online store. When clients log on to the system, a stateful session bean can maintain the items in the shopping cart. Each client has its own instance of a stateful session bean, which maintains a separate shopping cart for each client.

Note that with stateful session beans, client-specific information is stored in memory, not to a database. Therefore, you should use stateful session beans in situations where losing session data is not a problem when a client terminates a session. In our hypothetical online store, for instance, we discard the virtual shopping cart if a client decides not to buy the items. Saving the shopping cart contents comes under the application and use of entity beans, our next topic.

**Entity Beans**

Entity beans are in-memory business objects that correspond to data in persistent storage. An entity bean typically corresponds to a row in a relational database. The bean’s instance variables represent data in the columns of the database table. The container must synchronize a bean’s instance variables with the database. Entity beans differ from session beans in that instance variables are stored persistently. Entity beans also have primary keys for identifica-
tion and may have relationships with other entity beans. Another key concept is that clients may share entity beans.

The EJB container locates an entity bean by its primary key. Primary keys are unique identifiers. Database software prevents you from inserting new data if the primary key is not unique. If multiple clients attempt to access the same data in an entity bean, the container handles the transaction for you. Through an entity bean's deployment descriptor, the developer specifies the transaction's attributes associated with entity bean methods. The container performs the necessary rollbacks if any step in the transaction fails. This is one of the most vital services that the container provides for entity bean developers. We provide an overview of transactions and entity beans. See “Transaction Overview” on page 246 in Chapter 6.

An entity bean life cycle has three states: Does Not Exist, Pooled, and Ready. Figure 2–5 shows the life cycle diagram.

To transition from the Does Not Exist state to the Pooled state, the container creates a bean instance with Class.newInstance() and calls the setEntityContext() method in the bean. This allows bean developers to access a bean's

![Figure 2–5 Entity Bean Life Cycle](image-url)
context from the argument passed to `setEntityContext()`. In the *Pooled* State, all entity bean instances are identical.

Note that there are two paths for an entity bean to transition from the *Pooled* state to the *Ready* state. The client can invoke the entity bean’s `create()` method and consequently insert *new data* into the underlying database. The client can alternatively invoke one of the bean’s “finder” methods. This performs a select query on the underlying database, synchronizing the bean’s persistent fields from the *data already in* the database.

If a client wants to insert data into the database, the client calls the `create()` method with arguments representing the data values. This makes the container call `ejbCreate()` to initialize the bean before calling `ejbPostCreate()`. In the *Ready* state, clients may invoke business methods in the entity bean.

If, on the other hand, a client reads data from the database, the client calls the `findByPrimaryKey()` method or another finder method. This makes the container deliver an entity bean instance directly to the client if its state is *Ready*. If the requested bean is in the *Pooled* state, the container activates the bean and calls the bean’s `ejbActivate()` method. This changes the bean’s state to *Ready*.

There are also two paths from the *Ready* state to the *Pooled* state. If a client wants to remove data from the database, the client calls `remove()`. This makes the container call the bean’s `ejbRemove()` method. If the container needs to reclaim resources used by an entity bean, it can passivate the bean. To passivate an entity bean, the container calls `ejbPassivate()`. Both calls change the bean’s state from *Ready* to *Pooled*.

At the end of the life cycle, the container removes the bean instance from the pool and calls the bean’s `unsetEntityContext()` method. This changes the bean’s state from *Pooled* to *Does Not Exist*.

When should you use entity beans? An entity bean is appropriate for any situation where data must be maintained (created, updated, selected, deleted) in persistent storage. Entity beans should represent business data rather than perform a task-related function.

Entity beans have two types of persistence: Bean-Managed Persistence (BMP) and Container-Managed Persistence (CMP). Let’s take a look at each persistence type as it relates to a database.

**Bean-Managed Persistence (BMP)**

Entity beans with bean-managed persistence contain code that accesses a database. The beans’ code contains SQL calls to read and write to the database. BMP gives developers more control over how an entity bean interacts with a database.

An entity bean with BMP can implement SQL code targeted for a specific database platform or it can use a Data Access Object (DAO) to hide the details of a particular database. A DAO encapsulates database operations into helper
classes for a specific database. DAOs make entity beans with BMP more portable, although the bean developer still has to manipulate database access with Java methods and classes. We present the DAO pattern for BMP entity beans in Chapter 6 (see “DAO Pattern Implementation” on page 215).

**Container-Managed Persistence (CMP)**

Entity beans with container-managed persistence do not contain code for database access. The container generates the necessary database calls for you. This approach makes CMP entity beans more portable than BMP entity beans with DAOs. With the deployment descriptor set to specific attributes for CMP behavior, entity bean developers are spared from having to write SQL code for database access.

**Entity Relationships**

Entity beans, regardless of whether they use BMP or CMP for database access, can have relationships with other entity beans according to an abstract persistence schema. An entity bean’s abstract schema defines a bean’s persistent fields and its relationships with other entity beans. The persistent fields of an entity bean are stored in a database.

A bean’s relationship to another bean is stored as a relationship field. Relationship fields must also be stored in the database. With BMP, the developer decides how relationship fields are represented in the underlying database (using foreign keys allows data in one table to relate to data in another table). With CMP, the container constructs the appropriate cross-reference tables based on the abstract schema description the bean developer provides. Chapter 7 explores entity relationships with CMP.

**Message-Driven Beans**

Message-driven beans allow J2EE applications to receive messages asynchronously. This means a client’s thread does not block while waiting for an EJB’s business method to complete. Instead of calling a business method directly in a bean, clients send messages to a server that stores them and returns control to the client right away. The EJB container has a pool of message bean instances that it uses to process messages. When the message is received, the message bean can access a database or call an EJB business method. This arrangement allows the invocation of lengthy business methods without making the client wait for the method to complete its job.

**Using JMS**

Message beans use the Java Message Service (JMS) to handle messaging. Figure 2–6 shows the approach. Clients use a JMS server to store messages in a
2.4 Enterprise JavaBeans

queue or topic destination. In JMS, a topic is used for a one-to-many broadcast
and a queue for a one-to-one communication. When a message arrives, the con-
tainer calls the `onMessage()` method in the message bean to process the mes-
sage. Using the JMS server as an intermediary decouples the client from the
message bean. This is a key point with message beans.

The container uses a bean instance from a pool of message beans. The con-
tainer also handles all the details of registering a message bean as a listener for
queue or topic messages.

Another key point with message beans is that they are stateless. This makes
message beans highly scalable, like stateless session beans. A message bean
retains no conversational state and can handle messages from multiple clients.
Message beans can connect to databases and call methods in other EJBs, too.
This makes message beans a valuable component in enterprise designs that
require asynchronous processing from clients.

Message beans also have a simple life cycle, as shown in Figure 2–7. There
are only two states: Does Not Exist and Ready. To change from the Does Not Exist
state to the Ready state, the container instantiates the message bean with
`Class.newInstance()` and calls its `setMessageDrivenContext()` and `ejb-
Create()` methods.

In the Ready state, a message bean may receive messages from the JMS
server. When a message arrives, the container calls the `onMessage()` method in
the message bean and passes the message to the method as an argument. Note
that message processing does not make a message bean change state.
Like stateless session beans, the container never passivates a message bean because message beans do not contain client-specific data. The life cycle of a message bean ends when the container calls the `ejbRemove()` method. This makes the message bean’s state change back to the **Does Not Exist** state.

When should you use message beans? In general, message beans are useful for receiving messages asynchronously (no waiting). You should consider using a message bean to decouple a client who cannot tolerate waiting for a lengthy business method to complete. A message bean that sends an e-mail confirmation to a large group of recipients is a good example.

The J2EE application server uses JMS to implement message-driven beans. A message bean is relatively easy to implement, since the container does most of the setup work that JMS requires.

**Clients and Interfaces**

A well-designed interface is important in enterprise programming because it represents the client’s view of an enterprise bean. Clients invoke business methods in a session bean or an entity bean only through a bean’s interface. This approach allows the EJB container to intercept client calls made through the EJB interface. The container can then perform any required system processing (such as transaction management) before forwarding the call to the method inside the EJB implementation class.
Two types of interfaces are possible with session and entity beans. Let’s find out what they are and how you might use them. (Note that message-driven beans do not have client interfaces since access is only through the JMS server.)

**Home and Remote Interfaces**

Clients may access session and entity beans remotely (from a machine running a different JVM) or locally (within the same JVM). For remote access, session and entity beans have a *remote interface* and a *home interface*. These interfaces represent the client’s view of an enterprise bean. The remote interface defines a bean’s business methods and the home interface defines life cycle methods. The home interface also defines finder and home methods for entity beans. Figure 2–8 shows the home and remote interfaces for a Customer EJB entity bean that has remote access.

Remote clients can be web components, J2EE application clients, or other enterprise beans. Remote clients may execute on one machine, and the enterprise bean it uses may run on a different machine. You must create both a remote interface and a home interface for a client to have remote access to the bean.

![Figure 2–8 Interfaces for Remote Access](image-url)

*Figure 2–8* Interfaces for Remote Access
Local Home and Local Interfaces

Clients may also interact with session or entity beans locally. This means a local client executes on the same machine as the enterprise bean it uses. Local clients can be web components or other enterprise beans, but not J2EE client applications. A common use of local interfaces is among related entity beans (entity beans with relationship fields to other entity beans). Also, you can construct a business process session bean as a front end (a session facade) to one or more entity beans. The session bean would typically use local access to the entity beans. (See “Session Facade Pattern” on page 250 as well as “Session Facade Pattern” on page 329 for the description, motivation, and implementation of this important design pattern.)

To have local access, you must create a local interface with business methods and a local home interface with life cycle and finder methods. A local interface is also the only way to have entity beans communicate with other entity beans in container-managed relationships. In an abstract schema, any entity bean that is the target of a container-managed relationship field must have a local interface. Figure 2–9 shows the local home and local interfaces for a Customer EJB entity bean with local access.
The primary reason for using local interfaces is increased performance. With local access to session or entity beans, method calls execute faster than remote calls, since both client and bean execute under control of the same EJB container.

2.5 Key Point Summary

This chapter introduced the J2EE architecture, J2EE components, and presents an overview of enterprise beans. You learned the basic concepts behind enterprise beans and how the container manages different beans in the J2EE architecture. You also learned how an enterprise bean might be used in a distributed architecture.

Here are the key points from this chapter.

- The J2EE platform gives you a multitiered application model to develop distributed components.
- A three-tier architecture places web components and business components in the middle tier. The presentation tier contains client applications and the database tier contains database programs.
- The J2EE components consist of technologies like enterprise beans, servlets, and JSP. They also contain APIs that are valuable to EJBs, such as JNDI, JDBC, and JMS.
- Remote Method Invocation (RMI) provides support for network calls used by enterprise beans in the J2EE architecture.
- The J2EE application server has containers that provide services to enterprise beans and web components.
- A deployment descriptor is an XML file that describes how the container will configure an EJB to run on a J2EE application server.
- The containers in the J2EE architecture manage many services, such as EJB and servlet life cycles, database connection pooling, security, transaction management, and data persistence.
- Enterprise beans are server-side components that encapsulate business logic and business data.
- The EJB container provides system-level services to each bean deployed in a J2EE application server. These services include transaction management, resource pooling, security, and multithreading control.
- There are three types of enterprise beans: session beans, entity beans, and message-driven beans. Session beans may be stateless.
or stateful, and entity beans may use bean-managed persistence or container-managed persistence.

- Stateless session beans do not store client-specific data in their instance variables. Stateless session beans are scalable and can be shared by many clients.
- Stateful session beans do store client-specific data in their instance variables. Stateless session beans store the conversational state between a client and a bean. The EJB container may passivate or activate stateful session beans by moving the bean between memory and secondary storage.
- Stateless session beans are appropriate for general tasks that do not apply to a specific client. Stateful session beans apply to situations where a bean must remember nonpersistent client information between method invocations.
- Entity beans are in-memory business objects that correspond to data in persistent storage.
- Entity beans with bean-managed persistence contain SQL calls to read and write to a database.
- Entity beans with container-managed persistence do not contain code for database access. The container generates the necessary database calls for you.
- An entity bean’s abstract schema defines a bean’s persistent fields and its relationships with other entity beans.
- A bean’s relationship to another bean is stored as a relationship field. The container uses relationship fields to identify related beans.
- Message beans allow J2EE applications to receive messages asynchronously. Message beans use the Java Message Service (JMS) to handle messaging.
- Message beans are highly scalable, similar to stateless session beans.
- The home or local home interface of an enterprise bean defines life cycle methods. The local or remote interface contains the business methods.
- Local interfaces allow enterprise beans to call methods locally, which performs better than remote calls across a network.