

Chapter 3

The J2EE Tour

3.1 Introduction

Two issues are explored in this chapter. Firstly, the relationship between the Java 2 Enterprise Edition (J2EE) and the Java 2 Standard Edition (J2SE) platforms is described. Secondly, we deconstruct the J2EE to take a tour of its constituent technologies. These technologies are revisited in depth in subsequent chapters, but here our aim is to give you an overview before plunging into detail later in the book.

3.2 The J2EE Platform

J2EE consists of a set of specifications (Shannon, 2001) and application programming interfaces (APIs) that build on top of the J2SE platform. J2SE provides APIs that are appropriate for the development of standalone applications, simple networked applications, or two-tier applications, for example client to database interaction. J2EE adds to the existing J2SE APIs by providing explicit support for developing server-side enterprise applications. These are applications that have the following characteristics:

- They are multi-tiered. Each tier has a particular responsibility, allowing both the separation of concerns and the physical separation of different aspects of the application. Physical separation provides for improved security, performance optimization and increased reliability.
- They are often accessed from clients via HTTP. This might support the customer-to-business model of interaction or indeed the business-to-business model of interaction, where HTTP or its secure form, HTTPS, is used as the transport protocol.
- They often have stringent security requirements. Enterprise applications act as a shop front to the outside world. Leaving the shop door unlocked at night or windows open will lead to unauthorized access or accidental exposure of confidential information. The consequences

of failing to take security seriously will be reported by the media, discrediting the industry, thereby reducing customer confidence. We have all read these press articles, such as the accidental exposure of customer credit card account numbers (BBC News, 2001). J2EE builds on the security model provided in J2SE, providing support for authentication (having, as it were, the right front door key), for authorization (having the right internal door keys and codes), and for encryption of data in transit.

- They must often integrate with existing legacy information systems. Many organization will want to *webify* existing internal applications. These will have their own information systems. It is essential therefore that J2EE provides mechanisms that enable a J2EE application to hook into these existing sources of business data.
- Many will require the use of both synchronous and asynchronous communication. Some client requests must be serviced immediately (synchronous handling of the request). Other requests might not demand an immediate response and can be queued for later processing (asynchronous handling of the request). J2EE supports both styles of communication.
- Many must be able to handle high volumes of incoming client requests. The J2EE specifications support concurrency and threading issues. However, it is the application server implementation of J2EE that will dictate the performance and reliability of your applications. You need to shop for J2EE application servers with care.
- They must support ACID (Atomicity, Consistency, Isolation, Durability) transaction principles, and within a distributed context. This is an important part of J2EE.

Projects developing server-side enterprise applications also have certain project requirements, in particular the requirement to get the product to market as soon as possible, without compromising the characteristics discussed above. Of course, this largely depends on the nature of the team and the development process followed. However, as mentioned in Chapter 2, J2EE provides a technological aid through its component technologies: EJBs and Web components. These technologies hide underlying APIs from the application's business logic code. The distribution properties of a component are specified externally of the component's code using XML. This also makes for rapid reconfiguration if the components need to be ported to other deployment environments.

Like J2SE, J2EE has a free Sun-provided implementation called the J2EE SDK (Software Development Kit).¹ This is a full J2EE application server. Lots of commercial implementations of J2EE exist² as well as open source implementations such as JBoss. Be careful not to confuse J2EE with its implementations. J2EE is comprised of a family of specification documents that can be downloaded from the Sun Web site. The API classes and documentation can also be downloaded from Sun. It is important to realize that these classes and interfaces are largely unimplemented. In fact, if you obtain any J2EE implementation it will come with these API classes and interfaces, but will, of course, also contain their full implementation. Although called a J2EE application server, in many cases the server consists of a federation of servers responsible for different aspects of J2EE; for example the EJB server, the HTTP server, the JNDI server and so on.

The J2EE specifications comprise:

1 <http://java.sun.com/j2ee/>

2 <http://java.sun.com/j2ee/licensees.html>

- A series of detailed J2EE technology-specific specifications, e.g. the EJB 2.1 specification and the JMS 1.1 specification.
- The J2EE 1.4 specification. This integrates the other specifications, dealing with issues that span the technologies, providing an overview of J2EE, and defining those J2EE features not dealt with elsewhere, such as the notion of the J2EE application (see Chapter 26).

Figure 3.1 provides an illustration of a generic, multi-tiered J2EE application where several clients interact with the application's Web and EJB components. The Web components interact with EJBs and also directly with databases in the Enterprise Information System (EIS) tier. Before dissecting this diagram further, it is important to realize that this diagram represents just one possible configuration of components and tiers. It is quite possible to have a J2EE application that leaves out one or more of the tiers, using a subset of the J2EE technologies. Also, for particularly complicated applications there might be any number of sub-tiers within one of the major tiers, perhaps dealing with separate subsystems.

Let us examine the diagram in more detail. There are four major logical tiers. The client tier contains the client components such as Web browsers, applets, Java applications, special J2EE clients, CORBA clients or other J2EE applications. Normally, they will be located on external machines. These clients communicate with the other tiers using a variety of mechanisms. HTTP and its secure encrypted form, HTTPS, will be used by Web browsers to send requests and receive responses. Other kinds of client may also use HTTP to communicate with the J2EE application server. A major advantage of HTTP is that firewalls allow communication on the default HTTP port, but for security reasons, prevent communication via other ports. The clients may also communicate with the other tiers using RMI, where the client makes a method call on a server object as if the server object was co-located with the client in the same JVM. This is particularly useful if the client is a Java application or applet. The disadvantage is that TCP/IP ports required by RMI may be barred by a firewall, although some servers (for example Weblogic) overcome this problem

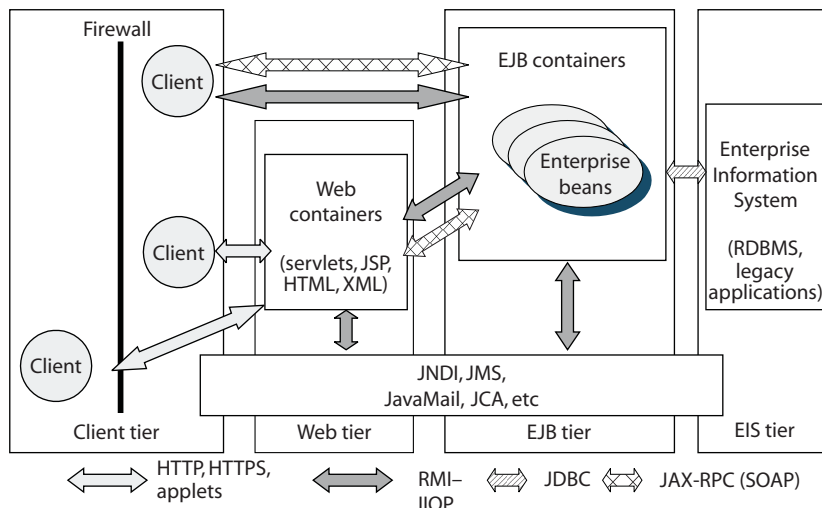


Figure 3.1 A multi-tier J2EE application.

by implementing RMI over HTTP. The clients could also use JMS to send messages asynchronously to the server-side application. Finally, and new to J2EE 1.4, clients may also communicate with certain kinds of EJB via the JAX-RPC API. This will send XML SOAP messages to the EJB. These messages ride on the back of HTTP(S) and so can bypass many firewall restrictions associated with other protocols.

The Web tier contains server-side components that are responsible for receiving, processing and responding to HTTP/S requests. Servlets and JSPs are the two J2EE technologies used in this tier. Servlets and JSPs may respond with HTML, XML, GIFs, applets etc. that are received by the original client. Normally, these Web components need to interact with business data, either directly from a database or via EJBs. Communication with EJBs is achieved using RMI, JMS or JAX-RPC. Communication with databases, and other external information systems, is achieved using JDBC or the Java Connector Architecture (JCA). JavaMail may also be used to communicate back with the client, e.g. to send purchase details to a customer.

The EJB tier contains server-side components that are responsible for encapsulating the application's business logic. For those familiar with the Model-View-Controller design pattern, the EJB tier implements the model, whereas the Web tier implements the view and controller. We will examine J2EE's version of MVC in Chapter 34. However, it is important that business logic remains separated from the code responsible for presenting the graphical user interface, thereby making it easier to change one without affecting the other, and to physically separate these subsystems for security reasons. For example, the Web tier may be exposed to the Internet; however, the EJB tier containing the business logic and associated data could be located on a separate machine that can only be accessed from within the organization's intranet. This is useful if you want to prevent direct access to the EJB tier from clients; that is, all client communication must be directed to the Web tier, enforcing a single point of entry.

The EJB business logic components may need to communicate with each other in the same J2EE server, but may also communicate with EJBs supported by other servers. Communication is achieved using RMI and JMS, although communication between EJBs in the same JVM can be achieved via simple method calls. EJBs are an in-memory representation of business data, some of which is persistent. An object to relational mapping between the EJBs and databases in the EIS tier is supported using JDBC and the JCA.

The Web and EJB components and special J2EE application clients are maintained in containers. Containers are responsible for maintaining the life cycle and operation of their components. In the case of Web and EJB components, they provide a buffer between the client and the component. Containers will also interact with various J2EE services on behalf of their components, for example, using the JAAS APIs to authenticate client requests before those requests are received by a specific component. Of course, components may also access these J2EE APIs directly themselves. We discuss the role of containers in more detail in Chapter 12.

3.3 J2EE Technology Tour

Contrived “hello world” examples are provided for some of the technologies to give you a flavour of the technology, without having to learn too much detail. A more realistic example will be developed during subsequent chapters.

The technologies have been divided into three groups:

- **Communication services.** These are the APIs and their implementations that enable either a component in one container to interact with a component in another, or even same, container, or to interact with components external to the J2EE platform, such as Web browsers and CORBA applications.
- **Horizontal services.** These are general services that may be required by several tiers in a multi-tier application. For example, authentication will be required on first access by a client to your enterprise application. This authentication could either be performed in the Web tier if access is required through a Web component, or performed in the EJB tier, if direct access is requested on an EJB component.
- **Component technologies.** As explained in Section 2.5, these are the “power tool” technologies that make developing enterprise applications more straightforward than they would be if you only had access to communication and horizontal services. For most J2EE applications these component technologies will insulate your business code from many of the underlying J2EE APIs.

However, be aware of the ambiguity of this grouping as, for example, JDBC (the database API) may fall under either horizontal services or communication services, since database interaction often occurs across a network, perhaps using RMI as the communication mechanism.

The full set of J2EE 1.4 APIs are as follows. The J2EE APIs that form part of the J2SE 1.4 platform are: Java IDL API, JDBC API (including its `javax.sql` extension), RMI-IIOP API, JNDI API, JAXP 1.1 API and JAAS API.

Those that are contained within Java Optional packages are: EJB 2.1, Servlet 2.4, JSP 2.0, JMS 1.1, JTA 1.0, JavaMail 1.3, JAF 1.0, JAXP 1.2, Connector 1.5, Web Services 1.1, JAX-RPC 1.0, SAAJ 1.1, JAXR 1.0, J2EE Management 1.0, JMX 1.2, J2EE Deployment 1.1 and JACC 1.0.

As you will notice, we focus on a core set of these APIs to keep the book to a manageable size.

3.3.1 Communication Services

Remote Method Invocation (RMI)

RMI enables communication between distributed objects. However, the programmer writes code at a level that largely ignores the presence of a network; for example, if object A needs to call a method on a remote object B then it simply makes the call `B.method(args)`. But how can A access B if it is remote? What actually happens is that the JVM containing A receives a local representative of remote B, often called a proxy or stub, which we now call `B'` to avoid confusion. The code for this is generated automatically and “knows” how to communicate using sockets with remote B. When A calls `B'.method(args)` the stub packages up the arguments and method name as a stream of bytes that are directed across a socket to remote B. Actually, there is some helper code at the remote end that reads from the socket, processes the bytes and makes the method call on B on behalf of A.

Why is RMI useful? It allows the programmer to concentrate on writing business logic rather than worrying about distribution details. In particular, the programmer does not need to maintain socket code in either the client or the server, since this code is generated automatically. Other reasons are discussed in Chapter 4.

Originally, RMI only permitted communication between Java objects. However, RMI now allows communication to objects written in other languages, which is achieved using RMI-IIOP (Internet Inter-ORB Protocol). IIOP is a CORBA (Common Object Request Broker Architecture) transport protocol that usually sits on top of TCP/IP. One of the key aspects of IIOP is that it provides a programming language-neutral way of transmitting data. Data in one programming language is mapped to its equivalent neutral form in IIOP that is then mapped to an equivalent programming language representation at the receiving end. These mappings are defined as part of the CORBA standard.

Listing 3.1 provides a simple example of the use of RMI. This implements a server object that publishes the `sayHello` method to the world. The code that registers the server has not been included, but could be placed either in `HelloServer` or in a separate class. More on this in Chapter 4.

Listing 3.1 A snippet of RMI code.

```
public interface HelloInterface extends java.rmi.Remote {
    public String sayHello() throws java.rmi.RemoteException;
}

public class HelloServer extends UnicastRemoteObject implements
    HelloInterface {
    public HelloServer() throws RemoteException {
    }
    public String sayHello() {
        return "Hello world" ;
    }
}
```

Java Messaging Service (JMS)

Figure 3.2 illustrates the main concepts behind JMS. Essentially, JMS supports asynchronous communication between producers and consumers. In other words, a producer sends a message to a queue or topic, but rather than waiting for a response it continues to undertake other tasks. When ready, consumers will consume messages from topics or queues.

There are two models of communication. The simplest involves the use of FIFO queues and normally supports one-to-one or many-to-one interactions between producers and consumers. Message objects are created by the producers and sent to a named queue. Consumers who wish to read messages from the head of that queue will need to obtain a reference to the queue and then can wait for messages to be placed on the queue. If a message is placed on the queue it will be read (and removed from the queue) by the listening consumer. The other model of communication is through the use of topics. Topics are analogous to newsgroups, since consumers, which we call subscribers, subscribe to one or more topics. Producers, which we now call publishers, publish messages to topics. A separate copy of a message is sent to each subscriber. Consequently, topics support many-to-many communication. We discuss JMS in some detail in Chapter 7 and its relationship to EJBs in Chapter 16.

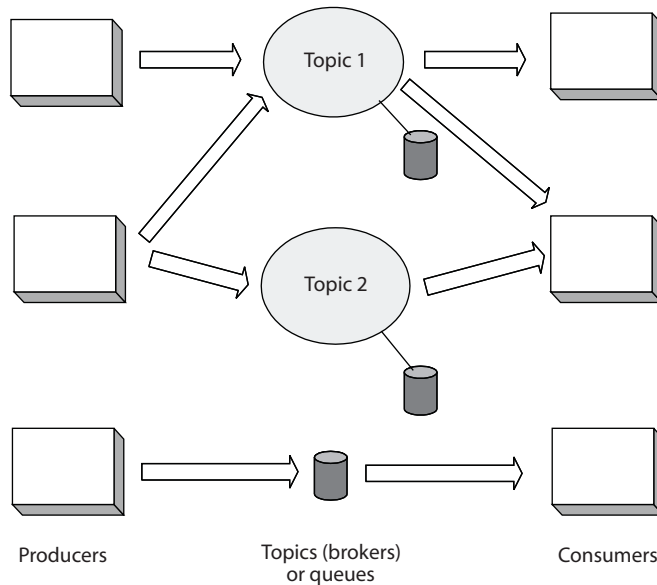


Figure 3.2 Overview of producers and consumers using JMS topics and queues.

JavaMail

JavaMail supports the sending and receiving of email from within a Java program. APIs are provided that allow you to construct MIME message objects, which are then sent and received via underlying mail protocols, such as SMTP, POP3 and IMAP4. Although a form of asynchronous communication, JavaMail is slower than JMS and should be used primarily where you need to interact with human users. We discuss JavaMail in Chapter 11.

HTTP/HTTPS

Hypertext Transfer Protocol is a text-based protocol used for communication across the Internet, and in particular, to support Web browser interaction with HTTP servers listening on server host machines. The protocol is stateless, in the sense that every request an HTTP client makes on the server will have to provide all the information needed to process that request; the server does not maintain any client state. Client requests are matched with server responses. Among other things, a client request identifies the resource target on the server (e.g. a particular HTML page or a servlet), the kind of request being made (HTTP supports several) and the nature of the client itself (such as the kind of browser, whether it can handle zipped data etc.), and any data to be sent to the server. Among other things, the server response will contain information about the status of the request (e.g. was the page found), the MIME type of the response data, and the response data.

HTTPS is the secure form of HTTP. It is where HTTP communication is transmitted over the Secure Socket Layer (SSL). SSL has its own handshake protocol that ensures that clients and servers can authenticate one another using verified digital certificates, and then transmit encrypted data.

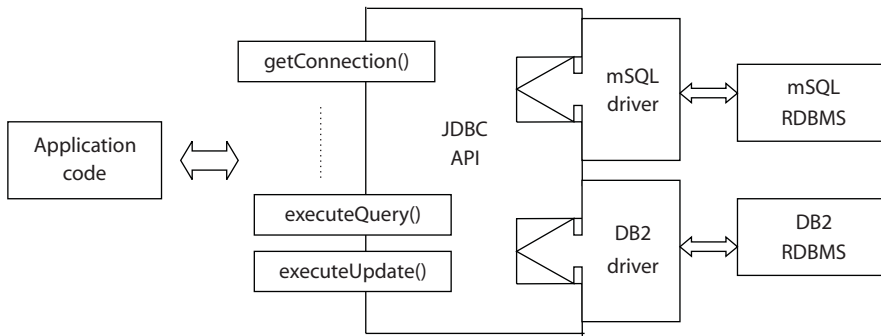


Figure 3.3 Conceptually how JDBC works.

3.3.2 Horizontal Services

JDBC

JDBC (unofficially, Java Database Connectivity) enables your programs to obtain database connections, execute SQL queries and updates via these connections, and process the results of such queries. The J2EE extensions to J2SE JDBC also provide support for connection pooling and distributed transactions. JDBC allows you to perform all these operations via database neutral APIs. Figure 3.3 illustrates this. The methods listed to the left of the diagram represent the neutral JDBC API, used to access any relational database system. The right-hand side of the diagram shows database driver modules (could be made up of several classes). There could be zero or many of these drivers plugged into JDBC. They have the responsibility of mapping a database neutral request onto the request expected by a real RDBMS. JDBC allows you to write much more portable code, since you pay less heed of the real database API, which can vary significantly.

Listing 3.2 is a very simple program that uses JDBC to query the Messages table for a row with the `id` column has the value 'world'.

Listing 3.2 A snippet of JDBC code.

```
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    String url = "jdbc:odbc:testDB";
    Connection con = DriverManager.getConnection(url);
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery("select message" +
        " from Messages where id = 'world'");
    if (rs.next())
        System.out.println(rs.getString("message"));
    rs.close();
    con.close();
} catch (Exception e) {}
```

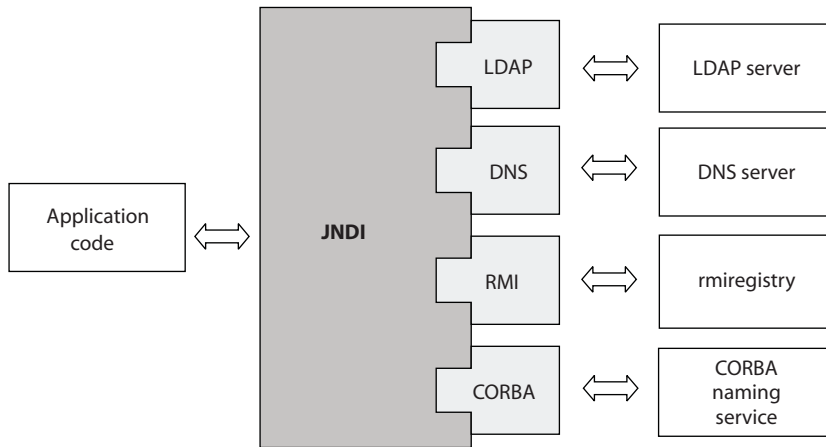


Figure 3.4 Overview of the JNDI architecture.

Java Naming and Directory Interface (JNDI)

Directory and naming services are common services across IT. They are used to map symbolic names or a set of search attribute values onto a resource. A file system is a naming service. A file resource is found by mapping a symbolic pathname onto a system file identifier (e.g. an inode in Unix). The Domain Name System maps symbolic host names onto their Internet addresses. Internally, J2EE application servers have a naming service implementation that is used by clients, Web components and EJBs to find J2EE resources using a symbolic naming scheme. All of these naming and directory services provide their own APIs and styles of interaction, and may be written in a variety of languages. JNDI provides a neutral API to these services, and in this respect is analogous to JDBC.

Figure 3.4 illustrates the main concepts. The API has methods such as `lookup`, `search`, `bind` and `rebind`. The application code uses `lookup` or `search` to locate some resource. However, before this can occur the application plugs in an appropriate mapping class, or classes (called the service provider). The application code will also need to tell the service provider where to look for the real server software. JNDI is discussed in more detail in Chapter 6 and subsequently in many other chapters, since it is the main mechanism for finding resources in J2EE.

Java Connector Architecture (JCA)

The JCA was new to J2EE 1.3. JCA provides mechanisms for integrating J2EE servers with heterogeneous Enterprise Information Systems (EIS) resources, such as Enterprise Resource Planning (ERP) applications, Customer Relationship Management (CRM) applications, RDBMSs etc. To comply with JCA, EIS vendors develop EIS specific resource adapters that are plugged into the JCA framework, which the J2EE server must support. Client applications access the EISs using the JCA's Common Client Interface API. This is analogous to using JDBC, except that this API does not focus on support for SQL. We discuss JCA in detail in Chapter 31.

Java Transaction API/Service (JTA/JTS)

JTS is a comprehensive transaction service that supports distributed transactions, and consequently two-phase commit protocol. For example, it supports a standard API that is called by a database's transaction manager when the database is ready to commit as part of a distributed transaction. JTA is a much simpler subset of JTS that is made available as a resource to J2EE applications. However, in most circumstances it is recommended that JTA is accessed by a container on behalf of its EJB components, rather than directly. We will discuss transactions in the context of EJBs in Chapter 27.

XML processing APIs

J2EE 1.4 includes the JAXP APIs. These APIs and their implementations support:

- Parsing of XML.
- Creation of in-memory representations of XML documents. This follows the Document Object Model defined by W3C (W3C, 2000).
- Parse-time triggering of callbacks. As the parser reaches a specific kind of tag or tag body, it can trigger the execution of a callback method that will take some application-specific action.
- Transformation of XML to some other representation, perhaps another XML document or an HTML document or even a PDF document. Transformation rules are encapsulated in an XSL file (W3C, 2002).

XML is becoming an important way of representing data in a standard format that can be validated against a Document Type Definition (DTD) or schema. This data can then be transmitted between various systems that convert the neutral format to system-specific formats, e.g. to a relational form. Just as Java provides code portability, XML provides data portability. XML is also used as the configuration language in J2EE. We provide a comprehensive overview of XML and related technologies in Chapter 10.

3.3.3 Component Technologies

Servlets

Servlets are server-side programs that execute in a servlet engine. This engine often forms part of an HTTP server, but may run standalone, being passed HTTP requests from a distinct HTTP server. A servlet is just a Java class that normally extends a `javax.servlet.http.HttpServlet` framework class. Servlet classes, JSPs and other resources can be packaged together as a Web component called a Web application. This is configured and then deployed within the J2EE server. Once deployed, an incoming client HTTP request to a servlet will cause an instance of the servlet to be created, unless one already exists. This then handles the request. Listing 3.3 provides an example of a simple servlet class. When a request is received, this servlet will send the "Hello World" HTML as part of an HTTP response back to the caller.

Listing 3.3 A snippet of servlet code.

```
public class Hello extends HttpServlet
{
    ...
    public void doPost(HttpServletRequest req, HttpServletResponse res)
    {
        res.setContentType("text/html");
        out = res.getWriter();
        out.println("<html><head><title>Hi world</title></head>");
        out.println("<body>Hello World!</body>");
        out.println("</html>");
    }
}
```

Java Server Pages (JSPs)

JSPs are servlets in disguise. As you can see from Listing 3.4, the format is HTML with additional JSP-specific tags. Although not shown here, Java code, called scriptlets, can be embedded in a JSP file. However, unlike HTML, the JSP source is translated automatically to servlet Java code prior to handling incoming HTTP requests. It is the generated servlet class, not the raw JSP, that handles the request.

Sun recommends the use of JSP where significant content is being developed. The aim should be to separate presentation and content from application logic, the JSP containing the presentation and content material. A JSP should be comprised largely of JSP and HTML tags, with little scriptlet code. Application-specific code can be encapsulated within developer-defined tags, JavaBeans or other Java classes called from the JSP. Avoid using servlets that contain large amounts of embedded HTML; use JSPs instead.

Listing 3.4 A snippet of JSP code.

```
<html>
<body>
<%@ page import="hello.BasicHello" %>
<jsp:useBean id="hello" scope="page" class="hello.BasicHello" />
<h1>Basic Hello World</h1>
This is a simple test
<p> <hr> <p>
Hello <jsp:getProperty name="hello" property="name"/>
<p> <hr>
</body></html>
```

Enterprise Java Beans (EJBs)

EJB components encapsulate business logic. Components have at least four parts:

- An implementation class that contains the business logic code.

- Two interfaces that advertise the EJB's methods to the outside world.
- A deployment descriptor: an XML file that is used to configure the EJB component before deployment within a J2EE server. For example, the security properties or transaction properties of EJB methods may be defined.

EJBs are container-managed components. That is, the container manages their life cycle and interacts with various J2EE services on behalf of the EJBs, based on configuration properties specified declaratively in their deployment descriptors. As mentioned in Chapter 2, the advantage of doing this is that 80% of the EJB code can be concerned with business logic and only 20% with issues of distribution. In non-EJB based applications this ratio may well be reversed, making maintenance and portability difficult.

3.4 References

- Shannon, B. (2001). *Java 2 Platform Enterprise Edition Specification 1.3*. Sun Microsystems.
- Shannon, B. (2002). *Java 2 Platform Enterprise Edition Specification 1.4*. Sun Microsystems, Proposed Final Draft.
- BBC News (2001). Red faces after credit card web blunder, 22 June 2001. <http://news.bbc.co.uk/1/hi/business/1401648.stm>.
- Java 2 Software Development Kit, Enterprise Edition. <http://java.sun.com/j2ee/>.
- W3C (2000). *Document Object Model (DOM) Level 3 Core Specification, Version 1.0*. W3C Recommendation, November 2000, <http://www.w3.org/TR/DOM-Level-2-Core>.
- W3C (2002). *XSL Transformations (XSLT) Version 2*. W3C Working Draft, August 2002, <http://www.w3.org/TR/xslt20>.



<http://www.springer.com/978-1-85233-704-9>

Guide to J2EE: Enterprise Java

Hunt, J.; Loftus, C.

2003, XXV, 672 p., Hardcover

ISBN: 978-1-85233-704-9