Thinking in Enterprise Java
by Bruce Eckel et. Al.

Revision 1.1, 5-06-2003

____

Modifications in Revision 1.1:

- Removed superscripts to improve viewing with IE

Modifications in Revision 1.0:

- Raw assembly of initial document. Please don't expect too much at this point, we're just getting started.
- Some material is carried forward from Thinking in Java 2nd edition, some material is new.
- No feedback mechanism in place at this time

Contents

Introduction to Enterprise Programming

Historically, programming across multiple machines has been error-prone, difficult, and complex.

The programmer had to know many details about the network and sometimes even the hardware. You usually needed to understand the various “layers” of the networking protocol, and there were a lot of different functions in each different networking library concerned with connecting, packing, and unpacking blocks of information; shipping those blocks back and forth; and handshaking. It was a daunting task.

However, the basic idea of distributed computing is not so difficult, and is abstracted very nicely in the Java libraries. You want to:

- Get some information from that machine over there and move it to this machine here, or vice versa. This is accomplished with basic network programming.
- Connect to a database, which may live across a network. This is accomplished with Java DataBase Connectivity (JDBC), which is an abstraction away from the messy, platform-specific details of SQL (the structured query language used for most database transactions).
- Provide services via a Web server. This is accomplished with Java’s servlets and JavaServer Pages (JSPs).
- Execute methods on Java objects that live on remote machines transparently, as if those objects were resident on local machines. This is accomplished with Java’s Remote Method Invocation (RMI).
- Use code written in other languages, running on other architectures. This is accomplished using the Extensible Markup Language (XML), which is directly supported by Java.
- Isolate business logic from connectivity issues, especially connections with databases including transaction management and security. This is accomplished using Enterprise JavaBeans (EJBs). EJBs are not actually a

distributed architecture, but the resulting applications are usually used in a networked client-server system.

- Easily, dynamically, add and remove devices from a network representing a local system. This is accomplished with Java's Jini.

Please note that each subject is voluminous and by itself the subject of entire books, so this book is only meant to familiarize you with the topics, not make you an expert (however, you can go a long way with the information presented here).

Prerequisites

This book assumes you have read (and understood most of) Thinking in Java, 3rd Edition (Prentice-Hall, 2003, available for download at www.MindView.net).

Summary

This chapter has introduced some, but not all, of the components that Sun refers to as J2EE: the Java 2 Enterprise Edition. The goal of J2EE is to create a set of tools that allows the Java developer to build server-based applications more quickly than before, and in a platform-independent way. It's not only difficult and time-consuming to build such applications, but it's especially hard to build them so that they can be easily ported to other platforms, and also to keep the business logic separated from the underlying details of the implementation. J2EE provides a framework to assist in creating server-based applications; these applications are in demand now, and that demand appears to be increasing.

Network programming with Sockets & Channels

One of Java's great strengths is painless networking. The Java network library designers have made it quite similar to reading and writing files, except that the "file" exists on a remote machine and the remote machine can decide exactly what it wants to do about the information you're requesting or sending. As much as possible, the underlying details of networking have been abstracted away and taken care of within the JVM and local machine installation of Java. The programming model you use is that of a file; in fact, you actually wrap the network connection (a "socket") with stream objects, so you end up using the same method calls as you do with all other streams. In addition, Java's built-in multithreading is exceptionally handy when dealing with another networking issue: handling multiple connections at once.

This section introduces Java's networking support using easy-to-understand examples.

Identifying a machine

Of course, in order to tell one machine from another and to make sure that you are connected with a particular machine, there must be some way of uniquely identifying machines on a network. Early networks were satisfied to provide unique names for machines within the local network. However, Java works within the Internet, which requires a way to uniquely identify a machine from all the others in the world. This is accomplished with the IP (Internet Protocol) address which can exist in two forms":

1. The familiar DNS (Domain Name System) form. My domain name is bruceeckel.com, and if I have a computer called Opus in my domain, its domain name would be Opus.bruceeckel.com. This is exactly the kind of name that you use when you send email to people, and is often incorporated into a World Wide Web address.

2. Alternatively, you can use the dotted quad" form, which is four numbers separated by dots, such as 123.255.28.120.

In both cases, the IP address is represented internally as a 32-bit number[1] (so each of the quad numbers cannot exceed 255), and you can get a special Java object to represent this number from either of the forms above by using the static

InetAddress.getByName( ) method that's in java.net. The result is an object of type InetAddress that you can use to build a "socket," as you will see later.

As a simple example of using InetAddress.getByName( ), consider what happens if you have a dial-up Internet service provider (ISP). Each time you dial up, you are assigned a temporary IP address. But while you're connected, your IP address has the same validity as any other IP address on the Internet. If someone connects to your machine using your IP address then they can connect to a Web server or FTP server that you have running on your machine. Of course, they need to know your IP address, and since a new one is assigned each time you dial up, how can you find out what it is?

The following program uses InetAddress.getByName( ) to produce your IP address. To use it, you must know the name of your computer. On Windows 95/98, go to "Settings," "Control Panel," "Network," and then select the "Identification" tab. "Computer name" is the name to put on the command line.

```
//: c15:WhoAmI.java
// Finds out your network address when
// you're connected to the Internet.
// {RunByHand} Must be connected to the Internet
// {Args: www.google.com}
import java.net.*;

public class WhoAmI {
  public static void main(String[] args)
      throws Exception {
    if(args.length != 1) {
      System.err.println(
        "Usage: WhoAmI MachineName");
      System.exit(1);
    }
    InetAddress a =
      InetAddress.getByName(args[0]);
    System.out.println(a);
  }
} ///:~
```

In this case, the machine is called "peppy." So, once I've connected to my ISP I run the program:

```
java WhoAmI peppy
```

I get back a message like this (of course, the address is different each time):

```
peppy/199.190.87.75
```

If I tell my friend this address and I have a Web server running on my computer, he can connect to it by going to the URL http://199.190.87.75 (only as long as I continue to stay connected during that session). This can sometimes be a handy way to distribute information to someone else, or to test out a Web site configuration before posting it to a “real” server.

Servers and clients

The whole point of a network is to allow two machines to connect and talk to each other. Once the two machines have found each other they can have a nice, two-way conversation. But how do they find each other? It’s like getting lost in an amusement park: one machine has to stay in one place and listen while the other machine says, “Hey, where are you?”

The machine that “stays in one place” is called the server, and the one that seeks is called the client. This distinction is important only while the client is trying to connect to the server. Once they’ve connected, it becomes a two-way communication process and it doesn’t matter anymore that one happened to take the role of server and the other happened to take the role of the client.

So the job of the server is to listen for a connection, and that’s performed by the special server object that you create. The job of the client is to try to make a connection to a server, and this is performed by the special client object you create. Once the connection is made, you’ll see that at both server and client ends, the connection is magically turned into an I/O stream object, and from then on you can treat the connection as if you were reading from and writing to a file. Thus, after the connection is made you will just use the familiar I/O commands from Chapter 11. This is one of the nice features of Java networking.

Testing programs without a network

For many reasons, you might not have a client machine, a server machine, and a network available to test your programs. You might be performing exercises in a classroom situation, or you could be writing programs that aren’t yet stable enough to put onto the network. The creators of the Internet Protocol were aware of this issue, and they created a special address called localhost to be the “local loopback” IP address for testing without a network. The generic way to produce this address in Java is:

```
InetAddress addr = InetAddress.getByName(null);
```

If you hand getByName( ) a null, it defaults to using the localhost. The InetAddress is what you use to refer to the particular machine, and you must produce this before you can go any further. You can’t manipulate the contents of an InetAddress (but you can print them out, as you’ll see in the next example). The only way you can create an InetAddress is through one of that class’s overloaded static member methods getByName( ) (which is what you’ll usually use), getAllByName( ), or getLocalHost( ).

You can also produce the local loopback address by handing it the string localhost:

```
InetAddress.getByName("localhost");
```

(assuming “localhost” is configured in your machine’s “hosts” table), or by using its dotted quad form to name the reserved IP number for the loopback:

```
InetAddress.getByName("127.0.0.1");
```

All three forms produce the same result.

Port: a unique place within the machine

An IP address isn’t enough to identify a unique server, since many servers can exist on one machine. Each IP machine also contains ports, and when you’re setting up a client or a server you must choose a port where both client and server agree to connect; if you’re meeting someone, the IP address is the neighborhood and the port is the bar.

The port is not a physical location in a machine, but a software abstraction (mainly for bookkeeping purposes). The client program knows how to connect to the machine via its IP address, but how does it connect to a desired service (potentially one of many on that machine)? That’s where the port numbers come in as a second level of addressing. The idea is that if you ask for a particular port, you’re requesting the service that’s associated with the port number. The time of day is a simple example of a service. Typically, each service is associated with a unique port number on a given server machine. It’s up to the client to know ahead of time which port number the desired service is running on.

The system services reserve the use of ports 1 through 1024, so you shouldn’t use those or any other port that you know to be in use. The first choice for examples in this book will be port 8080 (in memory of the venerable old 8-bit Intel 8080 chip in my first computer, a CP/M machine).

Sockets

The socket is the software abstraction used to represent the “terminals” of a connection between two machines. For a given connection, there’s a socket on each machine, and you can imagine a hypothetical “cable” running between the two machines with each end of the “cable” plugged into a socket. Of course, the physical hardware and cabling between machines is completely unknown. The whole point of the abstraction is that we don’t have to know more than is necessary.

In Java, you create a socket to make the connection to the other machine, then you get an InputStream and OutputStream (or, with the appropriate converters, Reader and Writer) from the socket in order to be able to treat the connection as an I/O stream object. There are two stream-based socket classes: a ServerSocket that a server uses to “listen” for incoming connections and a Socket that a client uses in order to initiate a connection. Once a client makes a socket connection, the ServerSocket returns (via the accept( ) method) a corresponding Socket through which communications will take place on the server side. From then on, you have a true Socket to Socket connection and you treat both ends the same way because they are the same. At this point, you use the methods getInputStream( ) and getOutputStream( ) to produce the corresponding InputStream and OutputStream objects from each Socket. These must be wrapped inside buffers and formatting classes just like any other stream object described in Chapter 11.

The use of the term ServerSocket would seem to be another example of a confusing naming scheme in the Java libraries. You might think ServerSocket would be better named "ServerConnector" or something without the word "Socket" in it. You might also think that ServerSocket and Socket should both be inherited from some common base class. Indeed, the two classes do have several methods in common, but not enough to give them a common base class. Instead, ServerSocket's job is to wait until some other machine connects to it, then to return an actual Socket. This is why ServerSocket seems to be a bit misnamed, since its job isn't really to be a socket but instead to make a Socket object when someone else connects to it.

However, the ServerSocket does create a physical "server" or listening socket on the host machine. This socket listens for incoming connections and then returns an "established" socket (with the local and remote endpoints defined) via the accept( ) method. The confusing part is that both of these sockets (listening and established) are associated with the same server socket. The listening socket can accept only new connection requests and not data packets. So while ServerSocket doesn't make much sense programmatically, it does "physically."

When you create a ServerSocket, you give it only a port number. You don't have to give it an IP address because it's already on the machine it represents. When you create a Socket, however, you must give both the IP address and the port number where you're trying to connect. (However, the Socket that comes back from ServerSocket.accept( ) already contains all this information.)

A simple server and client

This example makes the simplest use of servers and clients using sockets. All the server does is wait for a connection, then uses the Socket produced by that connection to create an InputStream and OutputStream. These are converted to a Reader and a Writer, then wrapped in a BufferedReader and a PrintWriter. After that, everything it reads from the BufferedReader it echoes to the PrintWriter until it receives the line "END," at which time it closes the connection.

The client makes the connection to the server, then creates an OutputStream and performs the same wrapping as in the server. Lines of text are sent through the resulting PrintWriter. The client also creates an InputStream (again, with appropriate conversions and wrapping) to hear what the server is saying (which, in this case, is just the words echoed back).

Both the server and client use the same port number and the client uses the local loopback address to connect to the server on the same machine so you don't have to test it over a network. (For some configurations, you might need to be connected to a network for the programs to work, even if you aren't communicating over that network.)

Here is the server:

```
//: c15:JabberServer.java
// Very simple server that just
// echoes whatever the client sends.
// {RunByHand}
import java.io.*;
import java.net.*;

public class JabberServer {
  // Choose a port outside of the range 1-1024:
  public static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Started: " + s);
    try {
      // Blocks until a connection occurs:
      Socket socket = s.accept();
      try {
        System.out.println(
          "Connection accepted: "+ socket);
        BufferedReader in =
          new BufferedReader(
            new InputStreamReader(
              socket.getInputStream()));
        // Output is automatically flushed
        // by PrintWriter:
        PrintWriter out =
          new PrintWriter(
            new BufferedWriter(
              new OutputStreamWriter(
                socket.getOutputStream())),true);
        while (true) {
          String str = in.readLine();
          if (str.equals("END")) break;
          System.out.println("Echoing: " + str);
          out.println(str);
        }
      // Always close the two sockets...
      } finally {
        System.out.println("closing...");
        socket.close();
      }
    } finally {
      s.close();
    }
  }
} ///:~
```

You can see that the ServerSocket just needs a port number, not an IP address (since it's running on this machine!). When you call accept( ), the method blocks until some client tries to connect to it. That is, it's there waiting for a connection, but other processes can run (see Chapter 14). When a connection is made, accept( ) returns with a Socket object representing that connection.

The responsibility for cleaning up the sockets is crafted carefully here. If the ServerSocket constructor fails, the program just quits (notice we must assume that the constructor for ServerSocket doesn't leave any open network sockets lying around if it fails). For this case, main( ) throws IOException so a try block is not necessary. If the ServerSocket constructor is successful then all other method calls must be guarded in a try-finally block to ensure that, no matter how the block is left, the ServerSocket is properly closed.

The same logic is used for the Socket returned by accept( ). If accept( ) fails, then we must assume that the Socket doesn't exist or hold any resources, so it doesn't need to be cleaned up. If it's successful, however, the following statements must be in a try-finally block so that if they fail the Socket will still be cleaned up. Care is required here because sockets use important nonmemory resources, so you must be diligent in order to clean them up (since there is no destructor in Java to do it for you).

Both the ServerSocket and the Socket produced by accept( ) are printed to System.out. This means that their toString( ) methods are automatically called. These produce:

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

Shortly, you'll see how these fit together with what the client is doing.

The next part of the program looks just like opening files for reading and writing except that the InputStream and OutputStream are created from the Socket object. Both the InputStream and OutputStream objects are converted to Reader and Writer objects using the "converter" classes InputStreamReader and OutputStreamWriter, respectively. You could also have used the Java 1.0 InputStream and OutputStream classes directly, but with output there's a distinct advantage to using the Writer approach. This appears with PrintWriter, which has an overloaded constructor that takes a second argument, a boolean flag that indicates whether to automatically flush the output at the end of each println( ) (but not print( )) statement. Every time you write to out, its buffer must be flushed so the information goes out over the network. Flushing is important for this particular example because the client and server each wait for a line from the other party before proceeding. If flushing doesn't occur, the information will not be put onto the network until the buffer is full, which causes lots of problems in this example.

When writing network programs you need to be careful about using automatic flushing. Every time you flush the buffer a packet must be created and sent. In this case, that’s exactly what we want, since if the packet containing the line isn’t sent then the handshaking back and forth between server and client will stop. Put another way, the end of a line is the end of a message. But in many cases, messages aren’t delimited by lines so it’s much more efficient to not use auto flushing and instead let the built-in buffering decide when to build and send a packet. This way, larger packets can be sent and the process will be faster.

Note that, like virtually all streams you open, these are buffered. There’s an exercise at the end of this chapter to show you what happens if you don’t buffer the streams (things get slow).

The infinite while loop reads lines from the BufferedReader in and writes information to System.out and to the PrintWriter out. Note that in and out could be any streams, they just happen to be connected to the network.

When the client sends the line consisting of “END,” the program breaks out of the loop and closes the Socket.

Here’s the client:

```
//: c15:JabberClient.java
// Very simple client that just sends
// lines to the server and reads lines
// that the server sends.
// {RunByHand}
import java.net.*;
import java.io.*;

public class JabberClient {
  public static void main(String[] args)
      throws IOException {
    // Passing null to getByName() produces the
    // special "Local Loopback" IP address, for
    // testing on one machine w/o a network:
    InetAddress addr =
      InetAddress.getByName(null);
    // Alternatively, you can use
    // the address or name:
    // InetAddress addr =
    //    InetAddress.getByName("127.0.0.1");
    // InetAddress addr =
    //    InetAddress.getByName("localhost");
    System.out.println("addr = " + addr);
    Socket socket =
      new Socket(addr, JabberServer.PORT);
    // Guard everything in a try-finally to make
    // sure that the socket is closed:
    try {
      System.out.println("socket = " + socket);
      BufferedReader in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Output is automatically flushed
      // by PrintWriter:
      PrintWriter out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())),true);
      for(int i = 0; i < 10; i ++) {
        out.println("howdy " + i);
        String str = in.readLine();
        System.out.println(str);
      }
      out.println("END");
    } finally {
      System.out.println("closing...");
      socket.close();
    }
  }
} ///:~
```

In main( ) you can see all three ways to produce the InetAddress of the local loopback IP address: using null, localhost, or the explicit reserved address 127.0.0.1. Of course, if you want to connect to a machine across a network you substitute that machine's IP address. When the InetAddress addr is printed (via the automatic call to its toString( ) method) the result is:

```
localhost/127.0.0.1
```

By handing getByName( ) a null, it defaulted to finding the localhost, and that produced the special address 127.0.0.1.

Note that the Socket called socket is created with both the InetAddress and the port number. To understand what it means when you print one of these Socket objects, remember that an Internet connection is determined uniquely by these four pieces of data: clientHost, clientPortNumber, serverHost, and serverPortNumber. When the server comes up, it takes up its assigned port (8080) on the localhost (127.0.0.1). When the client comes up, it is allocated to the next available port on its machine, 1077 in this case, which also happens to be on the same machine (127.0.0.1) as the server. Now, in order for data to move between the client and server, each side has to know where to send it. Therefore, during the process of connecting to the "known" server, the client sends a "return address" so the server knows where to send its data. This is what you see in the example output for the server side:

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

This means that the server just accepted a connection from 127.0.0.1 on port 1077 while listening on its local port (8080). On the client side:

```
Socket[addr=localhost/127.0.0.1,PORT=8080,localport=1077]
```

which means that the client made a connection to 127.0.0.1 on port 8080 using the local port 1077.

You'll notice that every time you start up the client anew, the local port number is incremented. It starts at 1025 (one past the reserved block of ports) and keeps going up until you reboot the machine, at which point it starts at 1025 again. (On UNIX machines, once the upper limit of the socket range is reached, the numbers will wrap around to the lowest available number again.)

Once the Socket object has been created, the process of turning it into a BufferedReader and PrintWriter is the same as in the server (again, in both cases you start with a Socket). Here, the client initiates the conversation by sending the string "howdy" followed by a number. Note that the buffer must again be flushed (which happens automatically via the second argument to the PrintWriter constructor). If the buffer isn't flushed, the whole conversation will hang because the initial "howdy" will never get sent (the buffer isn't full enough to cause the send to happen automatically). Each line that is sent back from the server is written to System.out to verify that everything is working correctly. To terminate the conversation, the agreed-upon "END" is sent. If the client simply hangs up, then the server throws an exception.

You can see that the same care is taken here to ensure that the network resources represented by the Socket are properly cleaned up, using a try-finally block.

Sockets produce a "dedicated" connection that persists until it is explicitly disconnected. (The dedicated connection can still be disconnected unexplicitly if one side, or an intermediary link, of the connection crashes.) This means the two parties are locked in communication and the connection is constantly open. This seems like a logical approach to networking, but it puts an extra load on the network. Later in this chapter you'll see a different approach to networking, in which the connections are only temporary.

Serving multiple clients

The JabberServer works, but it can handle only one client at a time. In a typical server, you'll want to be able to deal with many clients at once. The answer is multithreading, and in languages that don't directly support multithreading this means all sorts of complications. In Chapter 14 you saw that multithreading in Java is about as simple as possible, considering that multithreading is a rather complex topic. Because threading in Java is reasonably straightforward, making a server that handles multiple clients is relatively easy.

The basic scheme is to make a single ServerSocket in the server and call accept( ) to wait for a new connection. When accept( ) returns, you take the resulting Socket and use it to create a new thread whose job is to serve that particular client. Then you call accept( ) again to wait for a new client.

In the following server code, you can see that it looks similar to the JabberServer.java example except that all of the operations to serve a particular client have been moved inside a separate thread class:

```
//: c15:MultiJabberServer.java
// A server that uses multithreading
// to handle any number of clients.
// {RunByHand}
import java.io.*;
import java.net.*;

class ServeOneJabber extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  public ServeOneJabber(Socket s)
      throws IOException {
    socket = s;
    in =
      new BufferedReader(
        new InputStreamReader(
          socket.getInputStream()));
    // Enable auto-flush:
    out =
      new PrintWriter(
        new BufferedWriter(
          new OutputStreamWriter(
            socket.getOutputStream())), true);
    // If any of the above calls throw an
    // exception, the caller is responsible for
    // closing the socket. Otherwise the thread
    // will close it.
    start(); // Calls run()
  }
  public void run() {
    try {
      while (true) {
        String str = in.readLine();
        if (str.equals("END")) break;
        System.out.println("Echoing: " + str);
        out.println(str);
      }
      System.out.println("closing...");
    } catch(IOException e) {
      System.err.println("IO Exception");
    } finally {
      try {
        socket.close();
      } catch(IOException e) {
        System.err.println("Socket not closed");
      }
    }
  }
}

public class MultiJabberServer {
  static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Server Started");
    try {
      while(true) {
        // Blocks until a connection occurs:
        Socket socket = s.accept();
        try {
          new ServeOneJabber(socket);
        } catch(IOException e) {
          // If it fails, close the socket,
          // otherwise the thread will close it:
          socket.close();
        }
      }
    } finally {
      s.close();
    }
  }
} ///:~
```

The ServeOneJabber thread takes the Socket object that's produced by accept( ) in main( ) every time a new client makes a connection. Then, as before, it creates a BufferedReader and auto-flushed PrintWriter object using the Socket. Finally, it calls the special Thread method start( ), which performs thread initialization and then calls run( ). This performs the same kind of action as in the previous example: reading something from the socket and then echoing it back until it reads the special "END" signal.

The responsibility for cleaning up the socket must again be carefully designed. In this case, the socket is created outside of the ServeOneJabber so the responsibility can be shared. If the ServeOneJabber constructor fails, it will just throw the exception to the caller, who will then clean up the thread. But if the constructor succeeds, then the ServeOneJabber object takes over responsibility for cleaning up the thread, in its run( ).

Notice the simplicity of the MultiJabberServer. As before, a ServerSocket is created and accept( ) is called to allow a new connection. But this time, the return value of accept( ) (a Socket) is passed to the constructor for ServeOneJabber, which creates a new thread to handle that connection. When the connection is terminated, the thread simply goes away.

If the creation of the ServerSocket fails, the exception is again thrown through main( ). But if the creation succeeds, the outer try-finally guarantees its cleanup. The inner try-catch guards only against the failure of the ServeOneJabber constructor; if the constructor succeeds, then the ServeOneJabber thread will close the associated socket.

To test that the server really does handle multiple clients, the following program creates many clients (using threads) that connect to the same server. The maximum number of threads allowed is determined by the final int MAX_THREADS.

```
//: c15:MultiJabberClient.java
// Client that tests the MultiJabberServer
// by starting up multiple clients.
// {RunByHand}
import java.net.*;
import java.io.*;

class JabberClientThread extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  private static int counter = 0;
  private int id = counter++;
  private static int threadcount = 0;
  public static int threadCount() {
    return threadcount;
  }
  public JabberClientThread(InetAddress addr) {
    System.out.println("Making client " + id);
    threadcount++;
    try {
      socket =
        new Socket(addr, MultiJabberServer.PORT);
    } catch(IOException e) {
      System.err.println("Socket failed");
      // If the creation of the socket fails,
      // nothing needs to be cleaned up.
    }
    try {
      in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Enable auto-flush:
      out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())), true);
      start();
    } catch(IOException e) {
      // The socket should be closed on any
      // failures other than the socket
      // constructor:
      try {
        socket.close();
      } catch(IOException e2) {
        System.err.println("Socket not closed");
      }
    }
    // Otherwise the socket will be closed by
    // the run() method of the thread.
  }
  public void run() {
    try {
      for(int i = 0; i < 25; i++) {
        out.println("Client " + id + ": " + i);
        String str = in.readLine();
        System.out.println(str);
      }
      out.println("END");
    } catch(IOException e) {
      System.err.println("IO Exception");
    } finally {
      // Always close it:
      try {
        socket.close();
      } catch(IOException e) {
        System.err.println("Socket not closed");
      }
      threadcount--; // Ending this thread
    }
  }
}

public class MultiJabberClient {
  static final int MAX_THREADS = 40;
  public static void main(String[] args)
      throws IOException, InterruptedException {
    InetAddress addr =
      InetAddress.getByName(null);
    while(true) {
      if(JabberClientThread.threadCount()
         < MAX_THREADS)
        new JabberClientThread(addr);
      Thread.currentThread().sleep(100);
    }
  }
} ///:~
```

The JabberClientThread constructor takes an InetAddress and uses it to open a Socket. You're probably starting to see the pattern: the Socket is always used to create some kind of Reader and/or Writer (or InputStream and/or OutputStream) object, which is the only way that the Socket can be used. (You can, of course, write a class or two to automate this process instead of doing all the typing if it becomes painful.) Again, start( ) performs thread initialization and calls run( ). Here, messages are sent to the server and information from the server is echoed to the screen. However, the thread has a limited lifetime and eventually completes. Note that the socket is cleaned up if the constructor fails after the socket is created but before the constructor completes. Otherwise the responsibility for calling close( ) for the socket is relegated to the run( ) method.

The threadcount keeps track of how many JabberClientThread objects currently exist. It is incremented as part of the constructor and decremented as run( ) exits (which means the thread is terminating). In MultiJabberClient.main( ), you can see that the number of threads is tested, and if there are too many, no more are created. Then the method sleeps. This way, some threads will eventually terminate and more can be created. You can experiment with MAX_THREADS to see where your particular system begins to have trouble with too many connections.

Datagrams

The examples you've seen so far use the Transmission Control Protocol (TCP, also known as stream-based sockets), which is designed for ultimate reliability and guarantees that the data will get there. It allows retransmission of lost data, it provides multiple paths through different routers in case one goes down, and bytes are delivered in the order they are sent. All this control and reliability comes at a cost: TCP has a high overhead.

There's a second protocol, called User Datagram Protocol (UDP), which doesn't guarantee that the packets will be delivered and doesn't guarantee that they will arrive in the order they were sent. It's called an "unreliable protocol" (TCP is a "reliable protocol"), which sounds bad, but because it's much faster it can be useful. There are some applications, such as an audio signal, in which it isn't so critical if a few packets are dropped here or there but speed is vital. Or consider a time-of-day server, where it really doesn't matter if one of the messages is lost. Also, some applications might be able to fire off a UDP message to a server and can then assume, if there is no response in a reasonable period of time, that the message was lost.

Typically, you'll do most of your direct network programming with TCP, and only occasionally will you use UDP. There's a more complete treatment of UDP, including an example, in the first edition of this book (available on the CD ROM bound into this book, or as a free download from www.BruceEckel.com).

Using URLs from within an applet

It's possible for an applet to cause the display of any URL through the Web browser the applet is running within. You can do this with the following line:

```
getAppletContext().showDocument(u);
```

in which u is the URL object. Here's a simple example that redirects you to another Web page. Although you're just redirected to an HTML page, you could also redirect to the output of a CGI program.

```
//: c15:ShowHTML.java
// <applet code=ShowHTML width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class ShowHTML extends JApplet {
  JButton send = new JButton("Go");
  JLabel l = new JLabel();
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    send.addActionListener(new Al());
    cp.add(send);
    cp.add(l);
  }
  class Al implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
      try {
        // This could be a CGI program instead of
        // an HTML page.
        URL u = new URL(getDocumentBase(),
          "FetcherFrame.html");
        // Display the output of the URL using
        // the Web browser, as an ordinary page:
        getAppletContext().showDocument(u);
      } catch(Exception e) {
        l.setText(e.toString());
      }
    }
  }
  public static void main(String[] args) {
    Console.run(new ShowHTML(), 100, 50);
  }
} ///:~
```

The beauty of the URL class is how much it shields you from. You can connect to Web servers without knowing much at all about what's going on under the covers.

Reading a file from the server

A variation on the above program reads a file located on the server. In this case, the file is specified by the client:

```
//: c15:Fetcher.java
// <applet code=Fetcher width=500 height=300>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class Fetcher extends JApplet {
  JButton fetchIt= new JButton("Fetch the Data");
  JTextField f =
    new JTextField("Fetcher.java", 20);
  JTextArea t = new JTextArea(10,40);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    fetchIt.addActionListener(new FetchL());
    cp.add(new JScrollPane(t));
    cp.add(f); cp.add(fetchIt);
  }
  public class FetchL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      try {
        URL url = new URL(getDocumentBase(),
          f.getText());
        t.setText(url + "\n");
        InputStream is = url.openStream();
        BufferedReader in = new BufferedReader(
          new InputStreamReader(is));
        String line;
        while ((line = in.readLine()) != null)
          t.append(line + "\n");
      } catch(Exception ex) {
        t.append(ex.toString());
      }
    }
  }
  public static void main(String[] args) {
    Console.run(new Fetcher(), 500, 300);
  }
} ///:~
```

The creation of the URL object is similar to the previous example—getDocumentBase( ) is the starting point as before, but this time the name of the file is read from the JTextField. Once the URL object is created, its String version is placed in the JTextArea so we can see what it looks like. Then an InputStream is procured from the URL, which in this case will simply produce a stream of the characters in the file. After converting to a Reader and buffering, each line is read and appended to the JTextArea. Note that the JTextArea has been placed inside a JScrollPane so that scrolling is handled automatically.

Selector Based Multiplexing in JDK1.4

When you are reading or writing to a socket, you need to make the transfer of data efficient. Let us consider the write operation first. When you write data at the application layer (TCP or UDP socket), you are writing data into the operating system buffer. This data eventually forms the payload of a (TCP or UDP) packet that needs to be transferred to the destination machine through the network. When you write to the socket and if there isn't enough room available in the buffer, the write call will block. If you read from a socket and there isn't enough information to read from the operating system buffer that has the data received from the network, the read call will block. If a thread blocks on a read or write operation, that thread is unable to do anything else and may be slowing the performance of your program. Prior to JDK1.4 there was no way to break this thread from the blocked state. With channels you can perform an asynchronous close operation on the channel and the thread blocked on this channel recieves the AsynchronousCloseException.

The asynchronous IO in Java achieves the same thing as achieved by the select() call on unix systems. You can give a list of descriptors (readable or writable) to the select() function and it monitors these descriptors for some change of event. For a descriptor representing a socket you are reading from, data in the operating systems buffer for this descriptor represents event. For a descriptor representing a socket you are writing to, availability of space to write in the internal operating system buffer for this socket represents an event. Hence the select() call watches multiple descriptors to check for events.

What if you just read and write to descriptors whenever you want? Select can handle multiple descriptors thus allowing you to monitor multiple sockets. Consider an example of a chat server where the server has a connection to various clients. The type of data arriving at the server is intermittent. The server is suppose to read data from the sockets and flash it on a GUI that is shown to every client - to achieve this you read data from every client and write this data to every other client. Consider 5 clients 1, 2, 3, 4, and 5. If the server was programmed to perform a read on 1, write on 2, 3, 4, and 5, next read on 2 and write on 1, 3, 4, 5 and so on, then it may so happen that while the server thread is blocked on read on one of the client sockets, there may be data available from other sockets. One solution would be to allocate a different thread for each of the clients (pre JDK1.4). But this would not be scalable. Instead you can have a selector based mechanism which watches all the client sockets. It knows which socket has data that can be read without blocking. But if a single server thread does all this work (selection and write on each client) it would not be responsive. Hence in such a situation one thread monitors the sockets for read, selects which socket can be read, and delegates other responsibility (writing to other clients) to another thread(s) or a thread pool.

This pattern is called the reactor pattern where events are decoupled from the action associated with events (Pattern Oriented Software Architecture - Doug Schmidt).

In JDK 1.4 you create a channel, register a Selector object with the channel that will watch the channel for events. Many channels register with the same Selector object. A single thread that calls the Selector.select(), watches multiple channels. The classes ServerSocket, Socket and DatagramSocket, each have a getChannel() method but it returns null except if a channel was created using the open() call (DatagramChannel.open(), SocketChannel.open(), ServerSocketChannel.open()). You then need to associate a socket with this channel.

You multiplex several channels (hence sockets) using a Selector. The static call Selector.select() blocks for an event to occur on one of the channels. There is also a non blocking version to this method that takes, the number of milliseconds to sleep or block before it returns.

ByteBuffer is used to copy data from and into a channel. ByteBuffer is a stream of octets and you have to decode this stream as characters. At the client end in MultiJabberClient.java this was done using Writer and OuputStreamWriter classes. These classes converted the characters into a stream of bytes.

The program below NonBlockingIO.java explains how you can used Selector and Channel to do the multiplexing. This program needs Server running. It causes an exception at server, but its aim is not to communicate with the server but to show how select() works.

```
//: TIEJ:X1:NonBlockingIO.java
// Socket and selector configuration for non-blocking
// Connects to JabberServer.java
// {RunByHand}
import java.net.*;
import java.nio.channels.*;
import java.util.*;
import java.io.*;
/**
 * Aim: Shows how to use selector. No reading/writing
 *      just shows the readiness of operations.
 *
 * PseudoCode:
 * -> Create a selector.
 * -> Create a channel
 * -> Bind the socket associated with this channel to a
 *    <client-port>
 * -> Configure the channel as non-blocking
 * -> Register the channel with selector.
 * -> Invoke select() so that it blocks until registered
 *    channel is ready. (as opposed to select(long timeout)
 * -> Get the set of keys whose underlying channel is ready
 *    for the operation they showed interest when they
 *    registered with the selector.
 * -> Iterate through the keys.
 * -> For every key check if the underlying channel is ready
 *    for the operation it is interested in.
 * -> If ready print message of readiness.
 *
 * Notes:
 * -> Need MultiJabberServer running on the local machine.
 *    You run it to connect to the local MultiJabberServer
 * -> It causes and exception at MultiJabberServer but
 *    this exception is expected.
 */
public class NonBlockingIO {
  public static void main(String[] args)
    throws IOException {
    if(args.length < 2) {
      System.out.println(
        "Usage: java <client port> <local server port>");
      System.exit(1);
    }
    int cPort = Integer.parseInt(args[0]);
    int sPort = Integer.parseInt(args[1]);
    SocketChannel ch = SocketChannel.open();
    Selector sel = sel = Selector.open();
    try {
      ch.socket().bind(new InetSocketAddress(cPort));
      ch.configureBlocking(false);
      // channel interested in performing read/write/connect
      ch.register(sel, SelectionKey.OP_READ
        | SelectionKey.OP_WRITE | SelectionKey.OP_CONNECT);
      // Unblocks when ready to read/write/connect
      sel.select();
      // Keys whose underlying channel is ready, the
      // operation this channel is interested in can be
      // performed without blocking.
      Iterator it = sel.selectedKeys().iterator();
      while(it.hasNext()) {
        SelectionKey key = (SelectionKey)it.next();
        it.remove();
        // Is underlying channel of key ready to connect?
     // if((key.readyOps() & SelectionKey.OP_CONNECT) != 0) {
        if(key.isConnectable()) {
          InetAddress ad = InetAddress.getLocalHost();
          System.out.println("Connect will not block");
          //You must check the return value of connect to make
          //sure that it has connected. This call being
          //non-blocking may return without connecting when
          //there is no server where you are trying to connect
          //Hence you call finishConnect() that finishes the
          //connect operation.
          if(!ch.connect(new InetSocketAddress(ad, sPort)))
            ch.finishConnect();
        }
        // Is underlying channel of key ready to read?
        // if((key.readyOps() & SelectionKey.OP_READ) != 0)
        if(key.isReadable())
          System.out.println("Read will not block");
        // Is underlying channel of key ready to write?
        // if((key.readyOps() & SelectionKey.OP_WRITE) != 0)
        if(key.isWritable())
          System.out.println("Write will not block");
      }
    } finally {
      ch.close();
      sel.close();
    }
  }
} ///:~
```

As mentioned above you need to create a channel using the open() call. SocketChannel.open() creates a channel. Since it extends from AbstractSelectableChannel (DatagramChannel and SocketChannel) it has the functionality for registering itself to a selector. The register call does this. It takes as an argument the Selector with to register the channel with and the events that are of interest to this channel. Here the SocketChannel is shown to be interested in connect, read and write - hence SelectionKey.OP_CONNECT, SelectionKey.OP_READ and SelectionKey.OP_WRITE are specified in the register call while registering the channel with the Selector.

The static call Selector.select() watches all the channels that are registered with it for the events they registered for (second argument to register). You can have a channel interested in more than one event.

The next is an example that works like the JabberClient1.java but uses Selector.

```
//: TIEJ:X1:JabberClient1.java
// Very simple client that just sends lines to the server
// and reads lines that the server sends.
// {RunByHand}
import java.net.*;
import java.util.*;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
```

```
public class JabberClient1 {
 public static void main(String[] args)
     throws IOException {
   if(args.length < 1) {
     System.out.println(
       "Usage: java JabberClient1 <client-port>");
     System.exit(1);
   }
   int clPrt = Integer.parseInt(args[0]);
   SocketChannel sc = SocketChannel.open();
   Selector sel = Selector.open();
   try {
     sc.configureBlocking(false);
     sc.socket().bind(new InetSocketAddress(clPrt));
     sc.register(sel, SelectionKey.OP_READ |
       SelectionKey.OP_WRITE  | SelectionKey.OP_CONNECT);
     int i = 0;
     // Because of the asynchronous nature you do not know
     // when reading and writing is done, hence you need to
     // keep track of this, boolean written is used to
     // alternate between read and write. Whatever is written
     // is echoed and should be read.
     // boolean done is used to check when to break out of
     // the loop
     boolean written = false, done = false;
     //JabberServer.java to which this client connects writes
     //using BufferedWriter.println(). This method performs
     //encoding according to the defualt charset
     String encoding = System.getProperty("file.encoding");
     Charset cs = Charset.forName(encoding);
     ByteBuffer buf = ByteBuffer.allocate(16);
     while(!done) {
       sel.select();
       Iterator it = sel.selectedKeys().iterator();
       while(it.hasNext()) {
         SelectionKey key = (SelectionKey)it.next();
         it.remove();
         sc = (SocketChannel)key.channel();
         if(key.isConnectable() && !sc.isConnected()) {
           InetAddress addr = InetAddress.getByName(null);
           boolean success = sc.connect(
             new InetSocketAddress(addr,
               JabberServer.PORT));
           if(!success) sc.finishConnect();
         }
         if(key.isReadable() && written) {
           if(sc.read((ByteBuffer)buf.clear()) > 0) {
             written = false;
             String response = cs.decode(
               (ByteBuffer) buf.flip()).toString();
             System.out.print(response);
             if(response.indexOf("END") != -1) done = true;
```

```
          }
        }
        if(key.isWritable() && !written) {
          if(i < 10) sc.write(ByteBuffer.wrap(
              new String("howdy "+ i + '\n').getBytes()));
          else if(i == 10) sc.write(ByteBuffer.wrap(
              new String("END\n").getBytes()));
          written = true;
          i++;
        }
      }
    }
  } finally {
    sc.close();
    sel.close();
  }
 }
} ///:~
```

The next example below shows a simple selector based mechanism for the MultiJabberServer discussed earlier. This server works the same way as the old one did but is more efficient in that it does not need a separate thread to handle each client.

```
//: TIEJ:X1:MultiJabberServer1.java
// Has the same semantics as multi-threaded
// MultiJabberServer
// {RunByHand}
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.*;
/**
 * The Server accepts connections in non-blocking fashion.
 * A connection when established, a socket is created,
 * which is registered for read/write with the selector.
 * Reading/Writing is performed on this socket when the
 * selector unblocks.
 * This program works exactly the same way as MultiJabberServer.
 */
public class MultiJabberServer1 {
  public static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    //Channel read from data will be in ByteBuffer form
    //written by PrintWriter.println(). Decoding of this
    //byte stream requires character set of default encoding.
    String encoding = System.getProperty("file.encoding");
    //Had to initialized here since we do not wish to create
    //a new instance of Charset everytime it is required
    //Charset cs = Charset.forName(
    //  System.getProperty("file.encoding"));
    Charset cs = Charset.forName(encoding);
    ByteBuffer buffer = ByteBuffer.allocate(16);
    SocketChannel ch = null;
    ServerSocketChannel ssc = ServerSocketChannel.open();
    Selector sel = Selector.open();
    try {
      ssc.configureBlocking(false);
      //Local address on which it will listen for connections
      //Note: Socket.getChannel() returns null unless a channel
      //is associated with it as shown below.
      //i.e the expression (ssc.socket().getChannel() != null) is true
      ssc.socket().bind(new InetSocketAddress(PORT));
      // Channel is interested in OP_ACCEPT events
      SelectionKey key =
        ssc.register(sel, SelectionKey.OP_ACCEPT);
      System.out.println("Server on port: " + PORT);
      while(true) {
        sel.select();
        Iterator it = sel.selectedKeys().iterator();
        while(it.hasNext()) {
          SelectionKey skey = (SelectionKey)it.next();
          it.remove();
          if(skey.isAcceptable()) {
            ch = ssc.accept();
            System.out.println(
              "Accepted connection from:" + ch.socket());
            ch.configureBlocking(false);
            ch.register(sel, SelectionKey.OP_READ);
          } else {
            // Note no check performed if the channel
            // is writable or readable - to keep it simple
            ch = (SocketChannel)skey.channel();
            ch.read(buffer);
            CharBuffer cb = cs.decode(
              (ByteBuffer)buffer.flip());
            String response = cb.toString();
            System.out.print("Echoing : " + response);
            ch.write((ByteBuffer)buffer.rewind());
            if(response.indexOf("END") != -1) ch.close();
            buffer.clear();
          }
        }
      }
    } finally {
      if(ch != null) ch.close();
      ssc.close();
      sel.close();
    }
  }
} ///:~
```

Here is a simple implementation of a Thread Pool. There are no polling (busy-wait) threads in this implementation. It is completely based on wait() and notify().

```
//: TIEJ:X1:Worker.java
// Instances of Worker are pooled in threadpool
// {Clean: WorkerErr.log, WorkerErr.log.lck}
// {RunByHand}
import java.io.*;
import java.util.logging.*;
public class Worker extends Thread {
  public static final Logger logger =
    Logger.getLogger("Worker");
  private String workerId;
  private Runnable task;
  // Needs a reference of threadpool in which it exists so
  // that it can add itself to this threadpool when done.
  private ThreadPool threadpool;
  static {
    try {
      logger.setUseParentHandlers(false);
      FileHandler ferr = new FileHandler("WorkerErr.log");
      ferr.setFormatter(new SimpleFormatter());
      logger.addHandler(ferr);
    } catch(IOException e) {
      System.out.println("Logger not initialized..");
    }
  }
  public Worker(String id, ThreadPool pool) {
    workerId = id;
    threadpool = pool;
    start();
  }
  //ThreadPool when schedules a task uses this method
  //to delegate task to a Worker thread. In addition to setting
  //the task (of type Runnable) it also triggers the waiting
  //run() method to start executing the task.
  public void setTask(Runnable t) {
    task = t;
    synchronized(this) {
      notify();
    }
  }
  public void run() {
    try {
      while(!threadpool.isStopped()) {
        synchronized(this) {
          if(task != null) {
            try {
              task.run(); // run the task
            } catch(Exception e) {
              logger.log(Level.SEVERE,
                "Exception in source Runnable task", e);
            }
            // return itself to threadpool
            threadpool.putWorker(this);
          }
          wait();
        }
      }
      System.out.println(this + " Stopped");
    } catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
  }
  public String toString() {
    return "Worker : " + workerId;
  }
} ///:~
```

This the basic algorithm:
while true:

```
            1.check the queue of tasks
            2.if empty, wait on this queue till a task gets added
   (addTask() call that adds a task will notify this queue to
   unblock)
            3.Try to get a worker thread from the thread pool.
            4.If none is available, wait on the thread pool.
                    (When there is a free thread available notify
               this threadpool to unblock)
            5.At this point there is a task in the queue and also a
               free worker thread
            6.Delegate task from the queue to a worker thread.
```

end while:

```
//: TIEJ:X1:ThreadPool.java
// Thread that polls and executes tasks in pool.
// {RunByHand}
import java.util.*;
public class ThreadPool extends Thread {
  private static final int DEFAULT_NUM_WORKERS = 5;
  private LinkedList
    workerPool = new LinkedList(),
    taskList = new LinkedList();
  private boolean stopped = false;
  public ThreadPool() {
    this(DEFAULT_NUM_WORKERS);
  }
  public ThreadPool(int numOfWorkers) {
    for(int i = 0; i < numOfWorkers; i++)
      workerPool.add(new Worker("" + i, this));
    start();
  }
  public void run() {
    try {
      while(!stopped) {
        if(taskList.isEmpty()) {
          synchronized(taskQueue) {
            // If queue is empty, wait for tasks to be added
            taskList.wait();
          }
        } else if(workerPool.isEmpty()) {
          synchronized(workerPool) {
            // If no worker threads available, wait till
            // one is available
            workerPool.wait();
          }
        }
        // Run the next task from the tasks scheduled
        getWorker().setTask(
          (Runnable)taskList.removeLast());
      }
    } catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
  }
  public void addTask(Runnable task) {
    taskList.addFirst(task);
    synchronized(taskList) {
      taskList.notify(); // If new task added, notify
    }
  }
  public void putWorker(Worker worker) {
    workerPool.addFirst(worker);
    //There may be cases when you have a pool of 5 threads
    //and the requirement exceeds this. That is when a Worker is required
    //but none is available (or free), it just blocks on threadpool.
    //This is the event that there is now a free Worker thread in
    //threadpool. Hence this thread does a notification that unblocks
    //the ThreadPool thread waiting on threadpool
    synchronized(workerPool) {
      workerPool.notify();
    }
  }
  private Worker getWorker() {
    return (Worker)workerPool.removeLast();
  }
  public boolean isStopped() {
    return stopped;
  }
  public void stopThreads() {
    stopped = true;
    Iterator it = workerPool.iterator();
    while(it.hasNext()) {
      Worker w = (Worker)it.next();
      synchronized(w) {
        w.notify();
      }
    }
  } // Junit test
  public void testThreadPool() {
    ThreadPool tp = new ThreadPool();
    for(int i = 0; i < 10; i++) {
      tp.addTask(new Runnable() {
        public void run() {
          System.out.println("A");
        }
      });
    }
    tp.stopThreads();
  }
} ///:~
```

Next is MultiJabberServer2.java that uses threadpool. This is the Reactor pattern. As stated above, the events are decoupled from their associated actions. The threadpool asynchronously decouples the actions associated with the events. In an enterprise system this decoupling is typically achieved using Java Messaging System.(JMS)

```
//: TIEJ:X1:MultiJabberServer2.java
// Same semantics as MultiJabberServer1 using thread pooling.
// {RunByHand}
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.*;

class ServeOneJabber implements Runnable {
  private SocketChannel channel;
  private Selector sel;
  public ServeOneJabber(SocketChannel ch)
    throws IOException {
    channel = ch;
    sel = Selector.open();
  }
  public void run() {
    ByteBuffer buffer = ByteBuffer.allocate(16);
    boolean read = false, done = false;
    String response = null;
    try {
      channel.register(sel, SelectionKey.OP_READ |
        SelectionKey.OP_WRITE);
      while(!done) {
        sel.select();
        Iterator it = sel.selectedKeys().iterator();
        while(it.hasNext()) {
          SelectionKey key = (SelectionKey) it.next();
          it.remove();
          if(key.isReadable() && !read) {
            if(channel.read(buffer) > 0) read = true;
            CharBuffer cb = MultiJabberServer2.CS.decode(
              (ByteBuffer)buffer.flip());
            response = cb.toString();
          }
          if(key.isWritable() && read) {
            System.out.print("Echoing : " + response);
            channel.write((ByteBuffer)buffer.rewind());
            if(response.indexOf("END") != -1) done = true;
            buffer.clear();
            read = false;
          }
        }
      }
    } catch(IOException e) {
      // will be caught by Worker.java and logged.
      // Need to throw runtime exception since we cannot
      // keep it as IOException
      throw new RuntimeException(e);
    } finally {
      try {
        channel.close();
      } catch(IOException e) {
        System.out.println("Channel not closed.");
        // Throw it so that worker thread can log it.
        throw new RuntimeException(e);
      }
    }
  }
}

public class MultiJabberServer2 {
  public static final int PORT = 8080;
  private static String encoding =
    System.getProperty("file.encoding");
  public static final Charset CS =
    Charset.forName(encoding);
  // Make thread pool with 20 Worker threads.
  private static ThreadPool pool = new ThreadPool(20);
  public static void main(String[] args)
      throws IOException {
    ServerSocketChannel ssc = ServerSocketChannel.open();
    Selector sel = Selector.open();
    try {
      ssc.configureBlocking(false);
      ssc.socket().bind(new InetSocketAddress(PORT));
      SelectionKey key =
        ssc.register(sel, SelectionKey.OP_ACCEPT);
      System.out.println("Server on port: " + PORT);
      while(true) {
        sel.select();
        Iterator it = sel.selectedKeys().iterator();
        while(it.hasNext()) {
          SelectionKey skey = (SelectionKey) it.next();
          it.remove();
          if(skey.isAcceptable()) {
            SocketChannel channel = ssc.accept();
            System.out.println("Accepted connection from:" +
              channel.socket());
            channel.configureBlocking(false);
            // Decouple event and associated action
            pool.addTask(new ServeOneJabber(channel));
          }
        }
      }
    } finally {
      ssc.close();
      sel.close();
    }
  }
} ///:~
```

This is a minor update to JabberServer.java. Initially when a client sent 'END' JabberServer did not echo it. This version, JabberServer echoes the string 'END'. This change was made to make JabberClient1.java simpler.

```
//: TIEJ:X1:JabberServer.java
// Very simple server that just
// echoes whatever the client sends.
// {RunByHand}
import java.io.*;
import java.net.*;

public class JabberServer {
  // Choose a port outside of the range 1-1024:
  public static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Started: " + s);
    try {
      // Blocks until a connection occurs:
      Socket socket = s.accept();
      try {
        System.out.println(
          "Connection accepted: "+ socket);
        BufferedReader in =
          new BufferedReader(
            new InputStreamReader(
              socket.getInputStream()));
        // Output is automatically flushed
        // by PrintWriter:
```

```
        BufferedWriter out =
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream()));
        while(true) {
          String str = in.readLine();
          System.out.println("Echoing: " + str);
          out.write(str, 0, str.length());
          out.newLine();
          out.flush();
          if(str.equals("END")) break;
        }
      // Always close the two sockets...
      } finally {
        System.out.println("closing...");
        socket.close();
      }
    } finally {
      s.close();
    }
  }
} ///:~
```

More to networking

There's actually a lot more to networking than can be covered in this introductory treatment. Java networking also provides fairly extensive support for URLs, including protocol handlers for different types of content that can be discovered at an Internet site. You can find other Java networking features fully and carefully described in Java Network Programming by Elliotte Rusty Harold (O'Reilly, 1997).

Exercises

7. Compile and run the JabberServer and JabberClient programs in this chapter. Now edit the files to remove all of the buffering for the input and output, then compile and run them again to observe the results.

1. Create a server that asks for a password, then opens a file and sends the file over the network connection. Create a client that connects to this server, gives the appropriate password, then captures and saves the file. Test the pair of programs on your machine using the localhost (the local loopback IP address 127.0.0.1 produced by calling InetAddress.getByName(null)).

1. Modify the server in Exercise 2 so that it uses multithreading to handle multiple clients.

2. Modify JabberClient.java so that output flushing doesn't occur and observe the effect.

3. Modify MultiJabberServer so that it uses thread pooling. Instead of throwing away a thread each time a client disconnects, the thread should put itself into an “available pool” of threads. When a new client wants to connect, the server will look in the available pool for a thread to handle the request, and if one isn’t available, make a new one. This way the number of threads necessary will naturally grow to the required quantity. The value of thread pooling is that it doesn’t require the overhead of creating and destroying a new thread for each new client.

4. Starting with ShowHTML.java, create an applet that is a password-protected gateway to a particular portion of your Web site.

Remote Method Invocation (RMI)

Traditional approaches to executing code on other machines across a network have been confusing as well as tedious and error-prone to implement. The nicest way to think about this problem is that some object happens to live on another machine, and that you can send a message to the remote object and get a result as if the object lived on your local machine. This simplification is exactly what Java Remote Method Invocation (RMI) allows you to do. This section walks you through the steps necessary to create your own RMI objects.

Remote interfaces

RMI makes heavy use of interfaces. When you want to create a remote object, you mask the underlying implementation by passing around an interface. Thus, when the client gets a reference to a remote object, what they really get is an interface reference, which happens to connect to some local stub code that talks across the network. But you don't think about this, you just send messages via your interface reference.

When you create a remote interface, you must follow these guidelines:

- The remote interface must be public (it cannot have "package access," that is, it cannot be "friendly"). Otherwise, a client will get an error when attempting to load a remote object that implements the remote interface.
- The remote interface must extend the interface java.rmi.Remote.
- Each method in the remote interface must declare java.rmi.RemoteException in its throws clause in addition to any application-specific exceptions.
- A remote object passed as an argument or return value (either directly or embedded within a local object) must be declared as the remote interface, not the implementation class.

Here's a simple remote interface that represents an accurate time service:

```
//: c15:rmi:PerfectTimeI.java
// The PerfectTime remote interface.
package c15.rmi;
import java.rmi.*;

public interface PerfectTimeI extends Remote {
  long getPerfectTime() throws RemoteException;
} ///:~
```

It looks like any other interface except that it extends Remote and all of its methods throw RemoteException. Remember that all the methods of an interface are automatically public.

Implementing the remote interface

The server must contain a class that extends UnicastRemoteObject and implements the remote interface. This class can also have additional methods, but only the methods in the remote interface are available to the client, of course, since the client will get only a reference to the interface, not the class that implements it.

You must explicitly define the constructor for the remote object even if you're only defining a default constructor that calls the base-class constructor. You must write it out since it must throw RemoteException.

Here's the implementation of the remote interface PerfectTimeI:

```
//: c15:rmi:PerfectTime.java
// The implementation of
// the PerfectTime remote object.
// {Broken}
package c15.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class PerfectTime
extends UnicastRemoteObject
implements PerfectTimeI {
  // Implementation of the interface:
  public long getPerfectTime()
      throws RemoteException {
    return System.currentTimeMillis();
  }
  // Must implement constructor
  // to throw RemoteException:
  public PerfectTime() throws RemoteException {
    // super(); // Called automatically
  }
  // Registration for RMI serving. Throw
  // exceptions out to the console.
  public static void main(String[] args)
  throws Exception {
    System.setSecurityManager(
    new RMISecurityManager());
    PerfectTime pt = new PerfectTime();
    Naming.bind(
      "//peppy:2005/PerfectTime", pt);
    System.out.println("Ready to do time");
  }
} ///:~
```

Here, main( ) handles all the details of setting up the server. When you’re serving RMI objects, at some point in your program you must:

- Create and install a security manager that supports RMI. The only one available for RMI as part of the Java distribution is RMISecurityManager.
- Create one or more instances of a remote object. Here, you can see the creation of the PerfectTime object.
- Register at least one of the remote objects with the RMI remote object registry for bootstrapping purposes. One remote object can have methods that produce references to other remote objects. This allows you to set it up so the client must go to the registry only once, to get the first remote object.

Setting up the registry

Here, you see a call to the static method Naming.bind( ). However, this call requires that the registry be running as a separate process on the computer. The name of the registry server is rmiregistry, and under 32-bit Windows you say:

```
start rmiregistry
```

to start it in the background. On Unix, the command is:

```
rmiregistry &
```

Like many network programs, the rmiregistry is located at the IP address of whatever machine started it up, but it must also be listening at a port. If you invoke the rmiregistry as above, with no argument, the registry’s port will default to 1099. If you want it to be at some other port, you add an argument on the command line to specify the port. For this example, the port is located at 2005, so the rmiregistry should be started like this under 32-bit Windows:

```
start rmiregistry 2005
```

or for Unix:

```
rmiregistry 2005 &
```

The information about the port must also be given to the bind( ) command, as well as the IP address of the machine where the registry is located. But this brings up what can be a frustrating problem if you’re expecting to test RMI programs locally the way the network programs have been tested so far in this chapter. In the JDK 1.1.1 release, there are a couple of problems:[2]

1. localhost does not work with RMI. Thus, to experiment with RMI on a single machine, you must provide the name of the machine. To find out the name of your machine under 32-bit Windows, go to the control panel and select "Network." Select the "Identification" tab, and you'll see your computer name. In my case, I called my computer "Peppy." It appears that capitalization is ignored.

2. RMI will not work unless your computer has an active TCP/IP connection, even if all your components are just talking to each other on the local machine. This means that you must connect to your Internet service provider before trying to run the program or you'll get some obscure exception messages.

With all this in mind, the bind( ) command becomes:

```
Naming.bind("//peppy:2005/PerfectTime", pt);
```

If you are using the default port 1099, you don't need to specify a port, so you could say:

```
Naming.bind("//peppy/PerfectTime", pt);
```

You should be able to perform local testing by leaving off the IP address and using only the identifier:

```
Naming.bind("PerfectTime", pt);
```

The name for the service is arbitrary; it happens to be PerfectTime here, just like the name of the class, but you could call it anything you want. The important thing is that it's a unique name in the registry that the client knows to look for to procure the remote object. If the name is already in the registry, you'll get an AlreadyBoundException. To prevent this, you can always use rebind( ) instead of bind( ), since rebind( ) either adds a new entry or replaces the one that's already there.

Even though main( ) exits, your object has been created and registered so it's kept alive by the registry, waiting for a client to come along and request it. As long as the rmiregistry is running and you don't call Naming.unbind( ) on your name, the object will be there. For this reason, when you're developing your code you need to shut down the rmiregistry and restart it when you compile a new version of your remote object.

You aren't forced to start up rmiregistry as an external process. If you know that your application is the only one that's going to use the registry, you can start it up inside your program with the line:

```
LocateRegistry.createRegistry(2005);
```

Like before, 2005 is the port number we happen to be using in this example. This is the equivalent of running rmiregistry 2005 from a command line, but it can often be more convenient when you're developing RMI code since it eliminates the extra steps of starting and stopping the registry. Once you've executed this code, you can bind( ) using Naming as before.

Creating stubs and skeletons

If you compile and run PerfectTime.java, it won't work even if you have the rmiregistry running correctly. That's because the framework for RMI isn't all there yet. You must first create the stubs and skeletons that provide the network connection operations and allow you to pretend that the remote object is just another local object on your machine.

What's going on behind the scenes is complex. Any objects that you pass into or return from a remote object must implement Serializable (if you want to pass remote references instead of the entire objects, the object arguments can implement Remote), so you can imagine that the stubs and skeletons are automatically performing serialization and deserialization as they "marshal" all of the arguments across the network and return the result. Fortunately, you don't have to know any of this, but you do have to create the stubs and skeletons. This is a simple process: you invoke the rmic tool on your compiled code, and it creates the necessary files. So the only requirement is that another step be added to your compilation process.

The rmic tool is particular about packages and classpaths, however. PerfectTime.java is in the package c15.rmi, and even if you invoke rmic in the same directory in which PerfectTime.class is located, rmic won't find the file, since it searches the classpath. So you must specify the location off the class path, like so:

```
rmic c15.rmi.PerfectTime
```

You don't have to be in the directory containing PerfectTime.class when you execute this command, but the results will be placed in the current directory.

When rmic runs successfully, you'll have two new classes in the directory:

```
PerfectTime_Stub.class
PerfectTime_Skel.class
```

corresponding to the stub and skeleton. Now you're ready to get the server and client to talk to each other.

Using the remote object

The whole point of RMI is to make the use of remote objects simple. The only extra thing that you must do in your client program is to look up and fetch the remote interface from the server. From then on, it's just regular Java programming: sending messages to objects. Here's the program that uses PerfectTime:

```
//: c15:rmi:DisplayPerfectTime.java
// Uses remote object PerfectTime.
// {Broken}
```

```
package c15.rmi;
import java.rmi.*;
import java.rmi.registry.*;

public class DisplayPerfectTime {
  public static void main(String[] args)
  throws Exception {
    System.setSecurityManager(
      new RMISecurityManager());
    PerfectTimeI t =
      (PerfectTimeI)Naming.lookup(
        "//peppy:2005/PerfectTime");
    for(int i = 0; i < 10; i++)
      System.out.println("Perfect time = " +
        t.getPerfectTime());
  }
} ///:~
```

The ID string is the same as the one used to register the object with Naming, and the first part represents the URL and port number. Since you’re using a URL, you can also specify a machine on the Internet.

What comes back from Naming.lookup( ) must be cast to the remote interface, not to the class. If you use the class instead, you’ll get an exception.

You can see in the method call

```
t.getPerfectTime()
```

that once you have a reference to the remote object, programming with it is indistinguishable from programming with a local object (with one difference: remote methods throw RemoteException).

Summary

Exercises

Connecting to Databases

It has been estimated that half of all software development involves client/server operations. A great promise of Java has been the ability to build platform-independent client/server database applications. This has come to fruition with Java DataBase Connectivity (JDBC).

One of the major problems with databases has been the feature wars between the database companies. There is a “standard” database language, Structured Query Language (SQL-92), but you must usually know which database vendor you’re working with despite the standard. JDBC is designed to be platform-independent, so you don’t need to worry about the database you’re using while you’re programming. However, it’s still possible to make vendor-specific calls from JDBC so you aren’t restricted from doing what you must.

One place where programmers may need to use SQL type names is in the SQL TABLE CREATE statement when they are creating a new database table and defining the SQL type for each column. Unfortunately there are significant variations between SQL types supported by different database products. Different databases that support SQL types with the same semantics and structure may give those types different names. Most major databases support an SQL data type for large binary values: in Oracle this type is called a LONG RAW, Sybase calls it IMAGE, Informix calls it BYTE, and DB2 calls it LONG VARCHAR FOR BIT DATA. Therefore, if database portability is a goal you should try to use only generic SQL type identifiers.

Portability is an issue when writing for a book where readers may be testing the examples with all kinds of unknown data stores. I have tried to write these examples to be as portable as possible. You should also notice that the database-specific code has been isolated in order to centralize any changes that you may need to perform to get the examples operational in your environment.

JDBC, like many of the APIs in Java, is designed for simplicity. The method calls you make correspond to the logical operations you’d think of doing when gathering data from a database: connect to the database, create a statement and execute the query, and look at the result set.

To allow this platform independence, JDBC provides a driver manager that dynamically maintains all the driver objects that your database queries will need. So if you have three different kinds of vendor databases to connect to, you’ll need three different driver objects. The driver objects register themselves with the driver manager at the time of loading, and you can force the loading using Class.forName( ).

To open a database, you must create a "database URL" that specifies:

1. That you're using JDBC with "jdbc."
2. The "subprotocol": the name of the driver or the name of a database connectivity mechanism. Since the design of JDBC was inspired by ODBC, the first subprotocol available is the "jdbc-odbc bridge," specified by "odbc."
3. The database identifier. This varies with the database driver used, but it generally provides a logical name that is mapped by the database administration software to a physical directory where the database tables are located. For your database identifier to have any meaning, you must register the name using your database administration software. (The process of registration varies from platform to platform.)

All this information is combined into one string, the "database URL." For example, to connect through the ODBC subprotocol to a database identified as "people," the database URL could be:

```
String dbUrl = "jdbc:odbc:people";
```

If you're connecting across a network, the database URL will contain the connection information identifying the remote machine and can become a bit intimidating. Here is an example of a CloudScape database being called from a remote client utilizing RMI:

```
jdbc:rmi://192.168.170.27:1099/jdbc:cloudscape:db
```

This database URL is really two jdbc calls in one. The first part "jdbc:rmi://192.168.170.27:1099/" uses RMI to make the connection to the remote database engine listening on port 1099 at IP Address 192.168.170.27. The second part of the URL, "jdbc:cloudscape:db" conveys the more typical settings using the subprotocol and database name but this will only happen after the first section has made the connection via RMI to the remote machine.

When you're ready to connect to the database, call the static method DriverManager.getConnection( ) and pass it the database URL, the user name, and a password to get into the database. You get back a Connection object that you can then use to query and manipulate the database.

The following example opens a database of contact information and looks for a person's last name as given on the command line. It selects only the names of people that have email addresses, then prints out all the ones that match the given last name:

```
//: c15:jdbc:Lookup.java
```

```
// Looks up email addresses in a
// local database using JDBC.
// {Broken}
import java.sql.*;

public class Lookup {
  public static void main(String[] args)
  throws SQLException, ClassNotFoundException {
    String dbUrl = "jdbc:odbc:people";
    String user = "";
    String password = "";
    // Load the driver (registers itself)
    Class.forName(
      "sun.jdbc.odbc.JdbcOdbcDriver");
    Connection c = DriverManager.getConnection(
      dbUrl, user, password);
    Statement s = c.createStatement();
    // SQL code:
    ResultSet r =
      s.executeQuery(
        "SELECT FIRST, LAST, EMAIL " +
        "FROM people.csv people " +
        "WHERE " +
        "(LAST='" + args[0] + "') " +
        " AND (EMAIL Is Not Null) " +
        "ORDER BY FIRST");
    while(r.next()) {
      // Capitalization doesn't matter:
      System.out.println(
        r.getString("Last") + ", "
        + r.getString("fIRST")
        + ": " + r.getString("EMAIL") );
    }
    s.close(); // Also closes ResultSet
  }
} ///:~
```

You can see the creation of the database URL as previously described. In this example, there is no password protection on the database so the user name and password are empty strings.

Once the connection is made with DriverManager.getConnection( ), you can use the resulting Connection object to create a Statement object using the createStatement( ) method. With the resulting Statement, you can call executeQuery( ), passing in a string containing an SQL-92 standard SQL statement. (You’ll see shortly how you can generate this statement automatically, so you don’t have to know much about SQL.)

The executeQuery( ) method returns a ResultSet object, which is an iterator: the next( ) method moves the iterator to the next record in the statement, or returns false if the end of the result set has been reached. You’ll always get a ResultSet object back from executeQuery( ) even if a query results in an empty set (that is, an exception is not thrown). Note that you must call next( ) once before trying to read any record data. If the result set is empty, this first call to next( ) will return false. For each record in the result set, you can select the fields using (among other approaches) the field name as a string. Also note that the capitalization of the field name is ignored—it doesn't matter with an SQL database. You determine the type you'll get back by calling getInt( ), getString( ), getFloat( ), etc. At this point, you've got your database data in Java native format and can do whatever you want with it using ordinary Java code.

Getting the example to work

With JDBC, understanding the code is relatively simple. The confusing part is making it work on your particular system. The reason this is confusing is that it requires you to figure out how to get your JDBC driver to load properly, and how to set up a database using your database administration software.

Of course, this process can vary radically from machine to machine, but the process I used to make it work under 32-bit Windows might give you clues to help you attack your own situation.

Step 1: Find the JDBC Driver

The program above contains the statement:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

This implies a directory structure, which is deceiving. With this particular installation of JDK 1.1, there was no file called JdbcOdbcDriver.class, so if you looked at this example and went searching for it you'd be frustrated. Other published examples use a pseudo name, such as "myDriver.ClassName," which is less than helpful. In fact, the load statement above for the jdbc-odbc driver (the only one that actually comes with the JDK) appears in only a few places in the online documentation (in particular, a page labeled "JDBC-ODBC Bridge Driver"). If the load statement above doesn't work, then the name might have been changed as part of a Java version change, so you should hunt through the documentation again.

If the load statement is wrong, you'll get an exception at this point. To test whether your driver load statement is working correctly, comment out the code after the statement and up to the catch clause; if the program throws no exceptions it means that the driver is loading properly.

Step 2: Configure the database

Again, this is specific to 32-bit Windows; you might need to do some research to figure it out for your own platform.

First, open the control panel. You might find two icons that say “ODBC.” You must use the one that says “32bit ODBC,” since the other one is for backward compatibility with 16-bit ODBC software and will produce no results for JDBC. When you open the “32bit ODBC” icon, you’ll see a tabbed dialog with a number of tabs, including “User DSN,” “System DSN,” “File DSN,” etc., in which “DSN” means “Data Source Name.” It turns out that for the JDBC-ODBC bridge, the only place where it’s important to set up your database is “System DSN,” but you’ll also want to test your configuration and create queries, and for that you’ll also need to set up your database in “File DSN.” This will allow the Microsoft Query tool (that comes with Microsoft Office) to find the database. Note that other query tools are also available from other vendors.

The most interesting database is one that you’re already using. Standard ODBC supports a number of different file formats including such venerable workhorses as DBase. However, it also includes the simple “comma-separated ASCII” format, which virtually every data tool has the ability to write. In my case, I just took my “people” database that I’ve been maintaining for years using various contact-management tools and exported it as a comma-separated ASCII file (these typically have an extension of .csv). In the “System DSN” section I chose “Add,” chose the text driver to handle my comma-separated ASCII file, and then un-checked “use current directory” to allow me to specify the directory where I exported the data file.

You’ll notice when you do this that you don’t actually specify a file, only a directory. That’s because a database is typically represented as a collection of files under a single directory (although it could be represented in other forms as well). Each file usually contains a single table, and the SQL statements can produce results that are culled from multiple tables in the database (this is called a join). A database that contains only a single table (like my “people” database) is usually called a flat-file database. Most problems that go beyond the simple storage and retrieval of data generally require multiple tables that must be related by joins to produce the desired results, and these are called relational databases.

Step 3: Test the configuration

To test the configuration you’ll need a way to discover whether the database is visible from a program that queries it. Of course, you can simply run the JDBC program example above, up to and including the statement:

```
Connection c = DriverManager.getConnection(
  dbUrl, user, password);
```

If an exception is thrown, your configuration was incorrect.

However, it’s useful to get a query-generation tool involved at this point. I used Microsoft Query that came with Microsoft Office, but you might prefer something else. The query tool must know where the database is, and Microsoft Query required that I go to the ODBC Administrator’s “File DSN” tab and add a new entry there, again specifying the text driver and the directory where my database lives. You can name the entry anything you want, but it’s helpful to use the same name you used in “System DSN.”

Once you’ve done this, you will see that your database is available when you create a new query using your query tool.

Step 4: Generate your SQL query

The query that I created using Microsoft Query not only showed me that my database was there and in good order, but it also automatically created the SQL code that I needed to insert into my Java program. I wanted a query that would search for records that had the last name that was typed on the command line when starting the Java program. So as a starting point, I searched for a specific last name, “Eckel.” I also wanted to display only those names that had email addresses associated with them. The steps I took to create this query were:

1. Start a new query and use the Query Wizard. Select the “people” database. (This is the equivalent of opening the database connection using the appropriate database URL.)
2. Select the “people” table within the database. From within the table, choose the columns FIRST, LAST, and EMAIL.
3. Under “Filter Data,” choose LAST and select “equals” with an argument of “Eckel.” Click the “And” radio button.
4. Choose EMAIL and select “Is not Null.”
5. Under “Sort By,” choose FIRST.

The result of this query will show you whether you’re getting what you want.

Now you can press the SQL button and without any research on your part, up will pop the correct SQL code, ready for you to cut and paste. For this query, it looked like this:

```
SELECT people.FIRST, people.LAST, people.EMAIL
FROM people.csv people
WHERE (people.LAST='Eckel') AND
(people.EMAIL Is Not Null)
ORDER BY people.FIRST
```

Especially with more complicated queries it’s easy to get things wrong, but by using a query tool you can interactively test your queries and automatically generate the correct code. It’s hard to argue the case for doing this by hand.

Step 5: Modify and paste in your query

You’ll notice that the code above looks different from what’s used in the program. That’s because the query tool uses full qualification for all of the names, even when there's only one table involved. (When more than one table is involved, the qualification prevents collisions between columns from different tables that have the same names.) Since this query involves only one table, you can optionally remove the "people" qualifier from most of the names, like this:

```
SELECT FIRST, LAST, EMAIL
FROM people.csv people
WHERE (LAST='Eckel') AND
(EMAIL Is Not Null)
ORDER BY FIRST
```

In addition, you don't want this program to be hard coded to look for only one name. Instead, it should hunt for the name given as the command-line argument. Making these changes and turning the SQL statement into a dynamically-created String produces:

```
"SELECT FIRST, LAST, EMAIL " +
"FROM people.csv people " +
"WHERE " +
"(LAST='" + args[0] + "') " +
" AND (EMAIL Is Not Null) " +
"ORDER BY FIRST");
```

SQL has another way to insert names into a query called stored procedures, which is used for speed. But for much of your database experimentation and for your first cut, building your own query strings in Java is fine.

You can see from this example that by using the tools currently available—in particular the query-building tool—database programming with SQL and JDBC can be quite straightforward.

A GUI version of the lookup program

It's more useful to leave the lookup program running all the time and simply switch to it and type in a name whenever you want to look someone up. The following program creates the lookup program as an application/applet, and it also adds name completion so the data will show up without forcing you to type the entire last name:

```
//: c15:jdbc:VLookup.java
// GUI version of Lookup.java.
// <applet code=VLookup
// width=500 height=200></applet>
// {Broken}
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import java.sql.*;
import com.bruceeckel.swing.*;
```

```
public class VLookup extends JApplet {
  String dbUrl = "jdbc:odbc:people";
  String user = "";
  String password = "";
  Statement s;
  JTextField searchFor = new JTextField(20);
  JLabel completion =
    new JLabel("                        ");
  JTextArea results = new JTextArea(40, 20);
  public void init() {
    searchFor.getDocument().addDocumentListener(
      new SearchL());
    JPanel p = new JPanel();
    p.add(new Label("Last name to search for:"));
    p.add(searchFor);
    p.add(completion);
    Container cp = getContentPane();
    cp.add(p, BorderLayout.NORTH);
    cp.add(results, BorderLayout.CENTER);
    try {
      // Load the driver (registers itself)
      Class.forName(
        "sun.jdbc.odbc.JdbcOdbcDriver");
      Connection c = DriverManager.getConnection(
        dbUrl, user, password);
      s = c.createStatement();
    } catch(Exception e) {
      results.setText(e.toString());
    }
  }
  class SearchL implements DocumentListener {
    public void changedUpdate(DocumentEvent e){}
    public void insertUpdate(DocumentEvent e){
      textValueChanged();
    }
    public void removeUpdate(DocumentEvent e){
      textValueChanged();
    }
  }
  public void textValueChanged() {
    ResultSet r;
    if(searchFor.getText().length() == 0) {
      completion.setText("");
      results.setText("");
      return;
    }
    try {
      // Name completion:
      r = s.executeQuery(
        "SELECT LAST FROM people.csv people " +
        "WHERE (LAST Like '" +
        searchFor.getText()  +
        "%') ORDER BY LAST");
      if(r.next())
        completion.setText(
          r.getString("last"));
      r = s.executeQuery(
        "SELECT FIRST, LAST, EMAIL " +
        "FROM people.csv people " +
        "WHERE (LAST='" +
        completion.getText() +
        "') AND (EMAIL Is Not Null) " +
        "ORDER BY FIRST");
    } catch(Exception e) {
      results.setText(
        searchFor.getText() + "\n");
      results.append(e.toString());
      return;
    }
    results.setText("");
    try {
      while(r.next()) {
        results.append(
          r.getString("Last") + ", "
          + r.getString("fIRST") +
          ": " + r.getString("EMAIL") + "\n");
      }
    } catch(Exception e) {
      results.setText(e.toString());
    }
  }
  public static void main(String[] args) {
    Console.run(new VLookup(), 500, 200);
  }
} ///:~
```

Much of the database logic is the same, but you can see that a DocumentListener is added to listen to the JTextField (see the javax.swing.JTextField entry in the Java HTML documentation from java.sun.com for details), so that whenever you type a new character it first tries to do a name completion by looking up the last name in the database and using the first one that shows up. (It places it in the completion JLabel, and uses that as the lookup text.) This way, as soon as you've typed enough characters for the program to uniquely find the name you're looking for, you can stop.

Why the JDBC API seems so complex

When you browse the online documentation for JDBC it can seem daunting. In particular, in the DatabaseMetaData interface—which is just huge, contrary to most of the interfaces you see in Java—there are methods such as dataDefinitionCausesTransactionCommit( ), getMaxColumnNameLength( ), getMaxStatementLength( ), storesMixedCaseQuotedIdentifiers( ), supportsANSI92IntermediateSQL( ), supportsLimitedOuterJoins( ), and so on. What's this all about?

As mentioned earlier, databases have seemed from their inception to be in a constant state of turmoil, primarily because the demand for database applications, and thus database tools, is so great. Only recently has there been any convergence on the common language of SQL (and there are plenty of other database languages in common use). But even with an SQL "standard" there are so many variations on that theme that JDBC must provide the large DatabaseMetaData interface so that your code can discover the capabilities of the particular "standard" SQL database that it's currently connected to. In short, you can write simple, transportable SQL, but if you want to optimize speed your coding will multiply tremendously as you investigate the capabilities of a particular vendor's database.

This, of course, is not Java's fault. The discrepancies between database products are just something that JDBC tries to help compensate for. But bear in mind that your life will be easier if you can either write generic queries and not worry quite as much about performance, or, if you must tune for performance, know the platform you're writing for so you don't need to write all that investigation code.

A more sophisticated example

A more interesting example[3] involves a multitable database that resides on a server. Here, the database is meant to provide a repository for community activities and to allow people to sign up for these events, so it is called the Community Interests Database (CID). This example will only provide an overview of the database and its implementation, and is not intended to be an in-depth tutorial on database development. There are numerous books, seminars, and software packages that will help you in the design and development of a database.

In addition, this example presumes the prior installation of an SQL database on a server (although it could also be run on a local machine), and the interrogation and discovery of an appropriate JDBC driver for that database. Several free SQL databases are available, and some are even automatically installed with various flavors of Linux. You are responsible for making the choice of database and locating the JDBC driver; the example here is based on an SQL database system called "Cloudscape," which can be freely downloaded for development (not deployment) from http://www.Cloudscape.com. You'll need to follow the instructions in the download in order to properly install and set up Cloudscape.

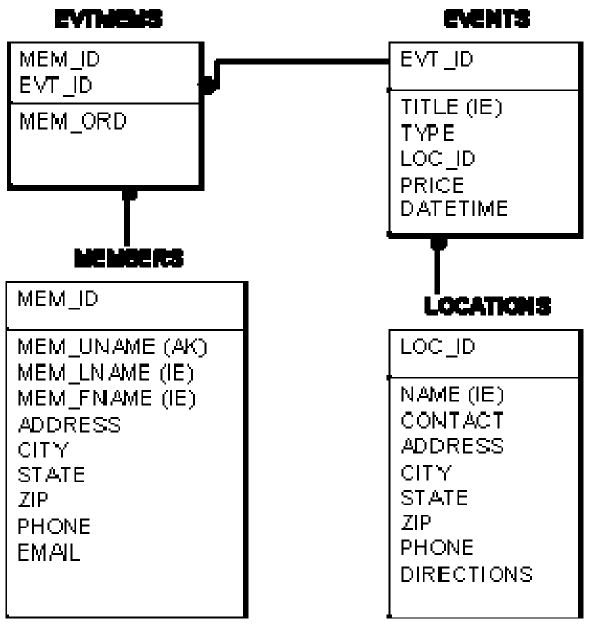
To keep changes in the connection information simple, the database driver, database URL, user name, and password are placed in a separate class:

```
//: c15:jdbc:CIDConnect.java
// Database connection information for
// the community interests database (CID).
```

```
public class CIDConnect {
  // All the information specific to CloudScape:
  public static String dbDriver =
    "COM.cloudscape.core.JDBCDriver";
  public static String dbURL =
    "jdbc:cloudscape:d:/docs/_work/JSapienDB";
  public static String user = "";
  public static String password = "";
} ///:~
```

In this example, there is no password protection on the database so the user name and password are empty strings. With Cloudscape, the dbURL contains the directory path where the database is located, but other JDBC drivers will use other ways to encode this information. This example assumes that the database "JSapienDB" has already been created, but in order to get the example to work you'll need to use the cview tool that comes with Cloudscape in order to create the new database, and then you must change the above dbURL to reflect the path of the database you created.

The database consists of a set of tables that have a structure as shown here:



"Members" contains community member information, "Events" and "Locations" contain information about the activities and where they take place, and "Evtmems" connects events and members that would like to attend that event. You can see that a data member in one table produces a key in another table.

The following class contains the SQL strings that will create these database tables (refer to an SQL guide for an explanation of the SQL code):

```
//: c15:jdbc:CIDSQL.java
// SQL strings to create the tables for the CID.
```

```
public class CIDSQL {
  public static String[] sql = {
    // Create the MEMBERS table:
    "drop table MEMBERS",
    "create table MEMBERS " +
    "(MEM_ID INTEGER primary key, " +
    "MEM_UNAME VARCHAR(12) not null unique, "+
    "MEM_LNAME VARCHAR(40), " +
    "MEM_FNAME VARCHAR(20), " +
    "ADDRESS VARCHAR(40), " +
    "CITY VARCHAR(20), " +
    "STATE CHAR(4), " +
    "ZIP CHAR(5), " +
    "PHONE CHAR(12), " +
    "EMAIL VARCHAR(30))",
    "create unique index " +
    "LNAME_IDX on MEMBERS(MEM_LNAME)",
    // Create the EVENTS table
    "drop table EVENTS",
    "create table EVENTS " +
    "(EVT_ID INTEGER primary key, " +
    "EVT_TITLE VARCHAR(30) not null, " +
    "EVT_TYPE VARCHAR(20), " +
    "LOC_ID INTEGER, " +
    "PRICE DECIMAL, " +
    "DATETIME TIMESTAMP)",
    "create unique index " +
    "TITLE_IDX on EVENTS(EVT_TITLE)",
    // Create the EVTMEMS table
    "drop table EVTMEMS",
    "create table EVTMEMS " +
    "(MEM_ID INTEGER not null, " +
    "EVT_ID INTEGER not null, " +
    "MEM_ORD INTEGER)",
    "create unique index " +
    "EVTMEM_IDX on EVTMEMS(MEM_ID, EVT_ID)",
    // Create the LOCATIONS table
    "drop table LOCATIONS",
    "create table LOCATIONS " +
    "(LOC_ID INTEGER primary key, " +
    "LOC_NAME VARCHAR(30) not null, " +
    "CONTACT VARCHAR(50), " +
    "ADDRESS VARCHAR(40), " +
    "CITY VARCHAR(20), " +
    "STATE VARCHAR(4), " +
    "ZIP VARCHAR(5), " +
    "PHONE CHAR(12), " +
    "DIRECTIONS VARCHAR(4096))",
    "create unique index " +
    "NAME_IDX on LOCATIONS(LOC_NAME)",
  };
} ///:~
```

The following program uses the CIDConnect and CIDSQL information to load the JDBC driver, make a connection to the database, and then create the table structure diagrammed above. To connect with the database, you call the static method DriverManager.getConnection( ), passing it the database URL, the user name, and a password to get into the database. You get back a Connection object that you can use to query and manipulate the database. Once the connection is made you can simply push the SQL to the database, in this case by marching through the CIDSQL array. However, the first time this program is run, the “drop table” command will fail, causing an exception, which is caught, reported, and then ignored. The reason for the “drop table” command is to allow easy experimentation: you can modify the SQL that defines the tables and then rerun the program, causing the old tables to be replaced by the new.

In this example, it makes sense to let the exceptions be thrown out to the console:

```
//: c15:jdbc:CIDCreateTables.java
// Creates database tables for the
// community interests database.
// {Broken}
import java.sql.*;

public class CIDCreateTables {
  public static void main(String[] args)
  throws SQLException, ClassNotFoundException,
  IllegalAccessException {
    // Load the driver (registers itself)
    Class.forName(CIDConnect.dbDriver);
    Connection c = DriverManager.getConnection(
      CIDConnect.dbURL, CIDConnect.user,
      CIDConnect.password);
    Statement s = c.createStatement();
    for(int i = 0; i < CIDSQL.sql.length; i++) {
      System.out.println(CIDSQL.sql[i]);
      try {
        s.executeUpdate(CIDSQL.sql[i]);
      } catch(SQLException sqlEx) {
        System.err.println(
          "Probably a 'drop table' failed");
      }
    }
    s.close();
    c.close();
  }
} ///:~
```

Note that all changes in the database can be controlled by changing Strings in the CIDSQL table, without modifying CIDCreateTables.

executeUpdate( ) will usually return the number of rows that were affected by the SQL statement. executeUpdate( ) is more commonly used to execute INSERT, UPDATE, or DELETE statements that modify one or more rows. For statements such as CREATE TABLE, DROP TABLE, and CREATE INDEX, executeUpdate( ) always returns zero.

To test the database, it is loaded with some sample data. This requires a series of INSERTs followed by a SELECT to produce result set. To make additions and changes to the test data easy, the test data is set up as a two-dimensional array of Objects, and the executeInsert( ) method can then use the information in one row of the table to create the appropriate SQL command.

```
//: c15:jdbc:LoadDB.java
// Loads and tests the database.
// {Broken}
import java.sql.*;

class TestSet {
  Object[][] data = {
    { "MEMBERS", new Integer(1),
      "dbartlett", "Bartlett", "David",
      "123 Mockingbird Lane",
      "Gettysburg", "PA", "19312",
      "123.456.7890",  "bart@you.net" },
    { "MEMBERS", new Integer(2),
      "beckel", "Eckel", "Bruce",
      "123 Over Rainbow Lane",
      "Crested Butte", "CO", "81224",
      "123.456.7890", "beckel@you.net" },
    { "MEMBERS", new Integer(3),
      "rcastaneda", "Castaneda", "Robert",
      "123 Downunder Lane",
      "Sydney", "NSW", "12345",
      "123.456.7890", "rcastaneda@you.net" },
    { "LOCATIONS", new Integer(1),
      "Center for Arts",
      "Betty Wright", "123 Elk Ave.",
      "Crested Butte", "CO", "81224",
      "123.456.7890",
      "Go this way then that." },
    { "LOCATIONS", new Integer(2),
      "Witts End Conference Center",
      "John Wittig", "123 Music Drive",
      "Zoneville", "PA", "19123",
      "123.456.7890",
      "Go that way then this." },
    { "EVENTS", new Integer(1),
      "Project Management Myths",
      "Software Development",
      new Integer(1), new Float(2.50),
      "2000-07-17 19:30:00" },
    { "EVENTS", new Integer(2),
      "Life of the Crested Dog",
      "Archeology",
      new Integer(2), new Float(0.00),
      "2000-07-19 19:00:00" },
    // Match some people with events
    {  "EVTMEMS",
      new Integer(1),  // Dave is going to
      new Integer(1),  // the Software event.
      new Integer(0) },
    { "EVTMEMS",
      new Integer(2),  // Bruce is going to
      new Integer(2),  // the Archeology event.
      new Integer(0) },
    { "EVTMEMS",
      new Integer(3),  // Robert is going to
      new Integer(1),  // the Software event.
      new Integer(1) },
    { "EVTMEMS",
      new Integer(3), // ... and
      new Integer(2), // the Archeology event.
      new Integer(1) },
  };
  // Use the default data set:
  public TestSet() {}
  // Use a different data set:
  public TestSet(Object[][] dat) { data = dat; }
}

public class LoadDB {
  Statement statement;
  Connection connection;
  TestSet tset;
  public LoadDB(TestSet t) throws SQLException {
    tset = t;
    try {
      // Load the driver (registers itself)
      Class.forName(CIDConnect.dbDriver);
    } catch(java.lang.ClassNotFoundException e) {
      e.printStackTrace(System.err);
    }
    connection = DriverManager.getConnection(
      CIDConnect.dbURL, CIDConnect.user,
      CIDConnect.password);
    statement = connection.createStatement();
  }
  public void dispose() throws SQLException {
    statement.close();
    connection.close();
  }
  public void executeInsert(Object[] data) {
    String sql = "insert into "
      + data[0] + " values(";
    for(int i = 1; i < data.length; i++) {
      if(data[i] instanceof String)
        sql += "'" + data[i] + "'";
      else
        sql += data[i];
      if(i < data.length - 1)
        sql += ", ";
    }
    sql += ')';
    System.out.println(sql);
    try {
      statement.executeUpdate(sql);
    } catch(SQLException sqlEx) {
      System.err.println("Insert failed.");
      while (sqlEx != null) {
        System.err.println(sqlEx.toString());
        sqlEx = sqlEx.getNextException();
      }
    }
  }
  public void load() {
    for(int i = 0; i< tset.data.length; i++)
      executeInsert(tset.data[i]);
  }
  // Throw exceptions out to console:
  public static void main(String[] args)
  throws SQLException {
    LoadDB db = new LoadDB(new TestSet());
    db.load();
    try {
      // Get a ResultSet from the loaded database:
      ResultSet rs = db.statement.executeQuery(
        "select " +
        "e.EVT_TITLE, m.MEM_LNAME, m.MEM_FNAME "+
        "from EVENTS e, MEMBERS m, EVTMEMS em " +
        "where em.EVT_ID = 2 " +
        "and e.EVT_ID = em.EVT_ID " +
        "and m.MEM_ID = em.MEM_ID");
      while (rs.next())
        System.out.println(
          rs.getString(1) + "  " +
          rs.getString(2) + ", " +
          rs.getString(3));
    } finally {
      db.dispose();
    }
  }
} ///:~
```

The TestSet class contains a default set of data that is produced if you use the default constructor; however, you can also create a TestSet object using an alternate data set with the second constructor. The set of data is held in a two-dimensional array of Object because it can be any type, including String or numerical types. The executeInsert( ) method uses RTTI to distinguish between String data (which must be quoted) and non-String data as it builds the SQL command from the data. After printing this command to the console, executeUpdate( ) is used to send it to the database.

The constructor for LoadDB makes the connection, and load( ) steps through the data and calls executeInsert( ) for each record. dispose( ) closes the statement and the connection; to guarantee that this is called, it is placed inside a finally clause.

Once the database is loaded, an executeQuery( ) statement produces a sample result set. Since the query combines several tables, it is an example of a join.

There is more JDBC information available in the electronic documents that come as part of the Java distribution from Sun. In addition, you can find more in the book JDBC Database Access with Java (Hamilton, Cattel, and Fisher, Addison-Wesley, 1997). Other JDBC books appear regularly.

Summary

Exercises

5.Modify CIDCreateTables.java so that it reads the SQL strings from a text file instead of CIDSQL.

6.Configure your system so that you can successfully execute CIDCreateTables.java and LoadDB.java.

7. (More challenging) Take the VLookup.java program and modify it so that when you click on the resulting name it automatically takes that name and copies it to the clipboard (so you can simply paste it into your email). You'll need to look back at Chapter 13 to remember how to use the clipboard in JFC.

Servlets

Client access from the Internet or corporate intranets is a sure way to allow many users to access data and resources easily[4].

This type of access is based on clients using the World Wide Web standards of Hypertext Markup Language (HTML) and Hypertext Transfer Protocol (HTTP). The Servlet API set abstracts a common solution framework for responding to HTTP requests.

Traditionally, the way to handle a problem such as allowing an Internet client to update a database is to create an HTML page with text fields and a “submit” button. The user types the appropriate information into the text fields and presses the “submit” button. The data is submitted along with a URL that tells the server what to do with the data by specifying the location of a Common Gateway Interface (CGI) program that the server runs, providing the program with the data as it is invoked. The CGI program is typically written in Perl, Python, C, C++, or any language that can read from standard input and write to standard output. That’s all that is provided by the Web server: the CGI program is invoked, and standard streams (or, optionally for input, an environment variable) are used for input and output. The CGI program is responsible for everything else. First it looks at the data and decides whether the format is correct. If not, the CGI program must produce HTML to describe the problem; this page is handed to the Web server (via standard output from the CGI program), which sends it back to the user. The user must usually back up a page and try again. If the data is correct, the CGI program processes the data in an appropriate way, perhaps adding it to a database. It must then produce an appropriate HTML page for the Web server to return to the user.

It would be ideal to go to a completely Java-based solution to this problem—an applet on the client side to validate and send the data, and a servlet on the server side to receive and process the data. Unfortunately, although applets are a proven technology with plenty of support, they have been problematic to use on the Web because you cannot rely on a particular version of Java being available on a client’s Web browser; in fact, you can’t rely on a Web browser supporting Java at all! In an intranet, you can require that certain support be available, which allows a lot more flexibility in what you can do, but on the Web the safest approach is to handle all the processing on the server side and deliver plain HTML to the client. That way, no client will be denied the use of your site because they do not have the proper software installed.

Because servlets provide an excellent solution for server-side programming support, they are one of the most popular reasons for moving to Java. Not only do they provide a framework that replaces CGI programming (and eliminates a number of thorny CGI problems), but all your code has the platform portability gained from using Java, and you have access to all the Java APIs (except, of course, the ones that produce GUIs, like Swing).

The basic servlet

The architecture of the servlet API is that of a classic service provider with a service( ) method through which all client requests will be sent by the servlet container software, and life cycle methods init( ) and destroy( ), which are called only when the servlet is loaded and unloaded (this happens rarely).

```
public interface Servlet {
  public void init(ServletConfig config)
    throws ServletException;
  public ServletConfig getServletConfig();
  public void service(ServletRequest req,
    ServletResponse res)
    throws ServletException, IOException;
  public String getServletInfo();
  public void destroy();
}
```

getServletConfig( )'s sole purpose is to return a ServletConfig object that contains initialization and startup parameters for this servlet. getServletInfo( ) returns a string containing information about the servlet, such as author, version, and copyright.

The GenericServlet class is a shell implementation of this interface and is typically not used. The HttpServlet class is an extension of GenericServlet and is designed specifically to handle the HTTP protocol— HttpServlet is the one that you'll use most of the time.

The most convenient attribute of the servlet API is the auxiliary objects that come along with the HttpServlet class to support it. If you look at the service( ) method in the Servlet interface, you'll see it has two parameters: ServletRequest and ServletResponse. With the HttpServlet class these two object are extended for HTTP: HttpServletRequest and HttpServletResponse. Here's a simple example that shows the use of HttpServletResponse:

```
//: c15:servlets:ServletsRule.java
// {Depends: j2ee.jar}
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ServletsRule extends HttpServlet {
  int i = 0; // Servlet "persistence"
  public void service(HttpServletRequest req,
  HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.print("<HEAD><TITLE>");
    out.print("A server-side strategy");
    out.print("</TITLE></HEAD><BODY>");
    out.print("<h1>Servlets Rule! " + i++);
    out.print("</h1></BODY>");
    out.close();
  }
} ///:~
```

ServletsRule is about as simple as a servlet can get. The servlet is initialized only once by calling its init( ) method, on loading the servlet after the servlet container is first booted up. When a client makes a request to a URL that happens to represent a servlet, the servlet container intercepts this request and makes a call to the service( ) method, after setting up the HttpServletRequest and HttpServletResponse objects.

The main responsibility of the service( ) method is to interact with the HTTP request that the client has sent, and to build an HTTP response based on the attributes contained within the request. ServletsRule only manipulates the response object without looking at what the client may have sent.

After setting the content type of the response (which must always be done before the Writer or OutputStream is procured), the getWriter( ) method of the response object produces a PrintWriter object, which is used for writing character-based response data (alternatively, getOutputStream( ) produces an OutputStream, used for binary response, which is only utilized in more specialized solutions).

The rest of the program simply sends HTML back to the client (it’s assumed you understand HTML, so that part is not explained) as a sequence of Strings. However, notice the inclusion of the “hit counter” represented by the variable i. This is automatically converted to a String in the print( ) statement.

When you run the program, you’ll notice that the value of i is retained between requests to the servlet. This is an essential property of servlets: since only one servlet of a particular class is loaded into the container, and it is never unloaded (unless the servlet container is terminated, which is something that only normally happens if you reboot the server computer), any fields of that servlet class effectively become persistent objects! This means that you can effortlessly maintain values between servlet requests, whereas with CGI you had to write values to disk in order to preserve them, which required a fair amount of fooling around to get it right, and resulted in a non-cross-platform solution.

Of course, sometimes the Web server, and thus the servlet container, must be rebooted as part of maintenance or during a power failure. To avoid losing any persistent information, the servlet’s init( ) and destroy( ) methods are automatically called whenever the servlet is loaded or unloaded, giving you the opportunity to save data during shutdown, and restore it after rebooting. The servlet container calls the destroy( ) method as it is terminating itself, so you always get an opportunity to save valuable data as long as the server machine is configured in an intelligent way.

There’s one other issue when using HttpServlet. This class provides doGet( ) and doPost( ) methods that differentiate between a CGI “GET” submission from the client, and a CGI “POST.” GET and POST vary only in the details of the way that they submit the data, which is something that I personally would prefer to ignore. However, most published information that I’ve seen seems to favor the creation of separate doGet( ) and doPost( ) methods instead of a single generic service( ) method, which handles both cases. This favoritism seems quite common, but I’ve never seen it explained in a fashion that leads me to believe that it’s anything more than inertia from CGI programmers who are used to paying attention to whether a GET or POST is being used. So in the spirit of “doing the simplest thing that could possibly work,”[5] I will just use the service( ) method in these examples, and let it care about GETs vs. POSTs. However, keep in mind that I might have missed something and so there may in fact be a good reason to use doGet( ) and doPost( ) instead.

Whenever a form is submitted to a servlet, the HttpServletRequest comes preloaded with all the form data, stored as key-value pairs. If you know the names of the fields, you can just use them directly with the getParameter( ) method to look up the values. You can also get an Enumeration (the old form of the Iterator) to the field names, as is shown in the following example. This example also demonstrates how a single servlet can be used to produce the page that contains the form, and to respond to the page (a better solution will be seen later, with JSPs). If the Enumeration is empty, there are no fields; this means no form was submitted. In this case, the form is produced, and the submit button will re-call the same servlet. If fields do exist, however, they are displayed.

```
//: c15:servlets:EchoForm.java
// Dumps the name-value pairs of any HTML form
// {Depends: j2ee.jar}
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class EchoForm extends HttpServlet {
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    Enumeration flds = req.getParameterNames();
    if(!flds.hasMoreElements()) {
      // No form submitted -- create one:
      out.print("<html>");
      out.print("<form method=\"POST\"" +
        " action=\"EchoForm\">");
      for(int i = 0; i < 10; i++)
        out.print("<b>Field" + i + "</b> " +
          "<input type=\"text\""+
          " size=\"20\" name=\"Field" + i +
          "\" value=\"Value" + i + "\"><br>");
      out.print("<INPUT TYPE=submit name=submit"+
        " Value=\"Submit\"></form></html>");
    } else {
      out.print("<h1>Your form contained:</h1>");
      while(flds.hasMoreElements()) {
        String field= (String)flds.nextElement();
        String value= req.getParameter(field);
        out.print(field + " = " + value+ "<br>");
      }
    }
    out.close();
  }
} ///:~
```

One drawback you'll notice here is that Java does not seem to be designed with string processing in mind—the formatting of the return page is painful because of line breaks, escaping quote marks, and the "+" signs necessary to build String objects. With a larger HTML page it becomes unreasonable to code it directly into Java. One solution is to keep the page as a separate text file, then open it and hand it to the Web server. If you have to perform any kind of substitution to the contents of the page, it's not much better since Java has treated string processing so poorly. In these cases you're probably better off using a more appropriate solution (Python would be my choice; there's a version that embeds itself in Java called JPython) to generate the response page.

Servlets and multithreading

The servlet container has a pool of threads that it will dispatch to handle client requests. It is quite likely that two clients arriving at the same time could be processing through your service( ) at the same time. Therefore the service( ) method must written in a thread-safe manner. Any access to common resources (files, databases) will need to be guarded by using the synchronized keyword.

The following simple example puts a synchronized clause around the thread's sleep( ) method. This will block all other threads until the allotted time (five seconds) is all used up. When testing this you should start several browser instances and hit this servlet as quickly as possible in each one—you'll see that each one has to wait until its turn comes up.

```
//: c15:servlets:ThreadServlet.java
// {Depends: j2ee.jar}
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ThreadServlet extends HttpServlet {
  int i;
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    synchronized(this) {
      try {
        Thread.currentThread().sleep(5000);
      } catch(InterruptedException e) {
        System.err.println("Interrupted");
      }
    }
    out.print("<h1>Finished " + i++ + "</h1>");
    out.close();
  }
} ///:~
```

It is also possible to synchronize the entire servlet by putting the synchronized keyword in front of the service( ) method. In fact, the only reason to use the synchronized clause instead is if the critical section is in an execution path that might not get executed. In that case, you might as well avoid the overhead of synchronizing every time by using a synchronized clause. Otherwise, all the threads will have to wait anyway so you might as well synchronize the whole method.

Handling sessions with servlets

HTTP is a “sessionless” protocol, so you cannot tell from one server hit to another if you’ve got the same person repeatedly querying your site, or if it is a completely different person. A great deal of effort has gone into mechanisms that will allow Web developers to track sessions. Companies could not do e-commerce without keeping track of a client and the items they have put into their shopping cart, for example.

There are several methods of session tracking, but the most common method is with persistent “cookies,” which are an integral part of the Internet standards. The HTTP Working Group of the Internet Engineering Task Force has written cookies into the official standard in RFC 2109 (ds.internic.net/rfc/rfc2109.txt or check www.cookiecentral.com).

A cookie is nothing more than a small piece of information sent by a Web server to a browser. The browser stores the cookie on the local disk, and whenever another call is made to the URL that the cookie is associated with, the cookie is quietly sent along with the call, thus providing the desired information back to that server (generally, providing some way that the server can be told that it’s you calling). Clients can, however, turn off the browser’s ability to accept cookies. If your site must track a client who has turned off cookies, then another method of session tracking (URL rewriting or hidden form fields) must be incorporated by hand, since the session tracking capabilities built into the servlet API are designed around cookies.

The Cookie class

The servlet API (version 2.0 and up) provides the Cookie class. This class incorporates all the HTTP header details and allows the setting of various cookie attributes. Using the cookie is simply a matter of adding it to the response object. The constructor takes a cookie name as the first argument and a value as the second. Cookies are added to the response object before you send any content.

```
Cookie oreo = new Cookie("TIJava", "2002");
res.addCookie(oreo);
```

Cookies are recovered by calling the getCookies( ) method of the HttpServletRequest object, which returns an array of cookie objects.

```
Cookie[] cookies = req.getCookies();
```

You can then call getValue( ) for each cookie, to produce a String containing the cookie contents. In the above example, getValue("TIJava") will produce a String containing “2002.”

The Session class

A session is one or more page requests by a client to a Web site during a defined period of time. If you buy groceries online, for example, you want a session to be confined to the period from when you first add an item to “my shopping cart” to the point where you check out. Each item you add to the shopping cart will result in a new HTTP connection, which has no knowledge of previous connections or items in the shopping cart. To compensate for this lack of information, the mechanics supplied by the cookie specification allow your servlet to perform session tracking.

A servlet Session object lives on the server side of the communication channel; its goal is to capture useful data about this client as the client moves through and interacts with your Web site. This data may be pertinent for the present session, such as items in the shopping cart, or it may be data such as authentication information that was entered when the client first entered your Web site, and which should not have to be reentered during a particular set of transactions.

The Session class of the servlet API uses the Cookie class to do its work. However, all the Session object needs is some kind of unique identifier stored on the client and passed to the server. Web sites may also use the other types of session tracking but these mechanisms will be more difficult to implement as they are not encapsulated into the servlet API (that is, you must write them by hand to deal with the situation when the client has disabled cookies).

Here’s an example that implements session tracking with the servlet API:

```
//: c15:servlets:SessionPeek.java
// Using the HttpSession class.
// {Depends: j2ee.jar}
```

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionPeek extends HttpServlet {
  public void service(HttpServletRequest req,
  HttpServletResponse res)
  throws ServletException, IOException {
    // Retrieve Session Object before any
    // output is sent to the client.
    HttpSession session = req.getSession();
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<HEAD><TITLE> SessionPeek ");
    out.println(" </TITLE></HEAD><BODY>");
    out.println("<h1> SessionPeek </h1>");
    // A simple hit counter for this session.
    Integer ival = (Integer)
      session.getAttribute("sesspeek.cntr");
    if(ival==null)
      ival = new Integer(1);
    else
      ival = new Integer(ival.intValue() + 1);
    session.setAttribute("sesspeek.cntr", ival);
    out.println("You have hit this page <b>"
      + ival + "</b> times.<p>");
    out.println("<h2>");
    out.println("Saved Session Data </h2>");
    // Loop through all data in the session:
    Enumeration sesNames =
      session.getAttributeNames();
    while(sesNames.hasMoreElements()) {
      String name =
        sesNames.nextElement().toString();
      Object value = session.getAttribute(name);
      out.println(name + " = " + value + "<br>");
    }
    out.println("<h3> Session Statistics </h3>");
    out.println("Session ID: "
      + session.getId() + "<br>");
    out.println("New Session: " + session.isNew()
      + "<br>");
    out.println("Creation Time: "
      + session.getCreationTime());
    out.println("<I>(" +
      new Date(session.getCreationTime())
      + ")</I><br>");
    out.println("Last Accessed Time: " +
      session.getLastAccessedTime());
    out.println("<I>(" +
      new Date(session.getLastAccessedTime())
      + ")</I><br>");
    out.println("Session Inactive Interval: "
      + session.getMaxInactiveInterval());
    out.println("Session ID in Request: "
      + req.getRequestedSessionId() + "<br>");
    out.println("Is session id from Cookie: "
      + req.isRequestedSessionIdFromCookie()
      + "<br>");
    out.println("Is session id from URL: "
      + req.isRequestedSessionIdFromURL()
      + "<br>");
    out.println("Is session id valid: "
      + req.isRequestedSessionIdValid()
      + "<br>");
    out.println("</BODY>");
    out.close();
  }
  public String getServletInfo() {
    return "A session tracking servlet";
  }
} ///:~
```

Inside the service( ) method, getSession( ) is called for the request object, which returns the Session object associated with this request. The Session object does not travel across the network, but instead it lives on the server and is associated with a client and its requests.

getSession( ) comes in two versions: no parameter, as used here, and getSession(boolean). getSession(true) is equivalent to getSession( ). The only reason for the boolean is to state whether you want the session object created if it is not found. getSession(true) is the most likely call, hence getSession( ).

The Session object, if it is not new, will give us details about the client from previous visits. If the Session object is new then the program will start to gather information about this client's activities on this visit. Capturing this client information is done through the setAttribute( ) and getAttribute( ) methods of the session object.

```
java.lang.Object getAttribute(java.lang.String)
void setAttribute(java.lang.String name,
                  java.lang.Object value)
```

The Session object uses a simple name-value pairing for loading information. The name is a String, and the value can be any object derived from java.lang.Object. SessionPeek keeps track of how many times the client has been back during this session. This is done with an Integer object named sesspeek.cntr. If the name is not found an Integer is created with value of one, otherwise an Integer is created with the incremented value of the previously held Integer. The new Integer is placed into the Session object. If you use same key in a setAttribute( ) call, then the new object overwrites the old one. The incremented counter is used to display the number of times that the client has visited during this session.

getAttributeNames( ) is related to getAttribute( ) and setAttribute( ); it returns an enumeration of the names of the objects that are bound to the Session object. A while loop in SessionPeek shows this method in action.

You may wonder how long a Session object hangs around. The answer depends on the servlet container you are using; they usually default to 30 minutes (1800 seconds), which is what you should see from the ServletPeek call to getMaxInactiveInterval( ). Tests seem to produce mixed results between servlet containers. Sometimes the Session object can hang around overnight, but I have never seen a case where the Session object disappears in less than the time specified by the inactive interval. You can try this by setting the inactive interval with setMaxInactiveInterval( ) to 5 seconds and see if your Session object hangs around or if it is cleaned up at the appropriate time. This may be an attribute you will want to investigate while choosing a servlet container.

Running the servlet examples

If you are not already working with an application server that handles Sun's servlet and JSP technologies for you, you may download the Tomcat implementation of Java servlets and JSPs, which is a free, open-source implementation of servlets, and is the official reference implementation sanctioned by Sun. It can be found at jakarta.apache.org.

Follow the instructions for installing the Tomcat implementation, then edit the server.xml file to point to the location in your directory tree where your servlets will be placed. Once you start up the Tomcat program you can test your servlet programs.

Summary

This has only been a brief introduction to servlets; there are entire books on the subject. However, this introduction should give you enough ideas to get you started. In addition, many of the ideas in the next section are backward compatible with servlets.

Exercises

8. Modify ServletsRule.java by overriding the destroy( ) method to save the value of i to a file, and and the init( ) method to restore the value. Demonstrate that it works by rebooting the servlet container. If you do not have an existing servlet container, you will need to download, install, and run Tomcat from jakarta.apache.org in order to run servlets.

9. Create a servlet that adds a cookie to the response object, thereby storing it on the client's site. Add code to the servlet that retrieves

and displays the cookie. If you do not have an existing servlet container, you will need to download, install, and run Tomcat from jakarta.apache.org in order to run servlets.

10. Create a servlet that uses a Session object to store session information of your choosing. In the same servlet, retrieve and display that session information. If you do not have an existing servlet container, you will need to download, install, and run Tomcat from jakarta.apache.org in order to run servlets.

11. Create a servlet that changes the inactive interval of a session to 5 seconds by calling getMaxInactiveInterval( ). Test to see that the session does indeed expire after 5 seconds. If you do not have an existing servlet container, you will need to download, install, and run Tomcat from jakarta.apache.org in order to run servlets.

JavaServer Pages

JavaServer Pages (JSP) is a standard Java extension that is defined on top of the servlet Extensions. The goal of JSPs is the simplified creation and management of dynamic Web pages.

The previously mentioned, freely available Tomcat reference implementation from jakarta.apache.org automatically supports JSPs.

JSPs allow you to combine the HTML of a Web page with pieces of Java code in the same document. The Java code is surrounded by special tags that tell the JSP container that it should use the code to generate a servlet, or part of one. The benefit of JSPs is that you can maintain a single document that represents both the page and the Java code that enables it. The downside is that the maintainer of the JSP page must be skilled in both HTML and Java (however, GUI builder environments for JSPs should be forthcoming).

The first time a JSP is loaded by the JSP container (which is typically associated with, or even part of, a Web server), the servlet code necessary to fulfill the JSP tags is automatically generated, compiled, and loaded into the servlet container. The static portions of the HTML page are produced by sending static String objects to write( ). The dynamic portions are included directly into the servlet.

From then on, as long as the JSP source for the page is not modified, it behaves as if it were a static HTML page with associated servlets (all the HTML code is actually generated by the servlet, however). If you modify the source code for the JSP, it is automatically recompiled and reloaded the next time that page is requested. Of course, because of all this dynamism you'll see a slow response for the first-time access to a JSP. However, since a JSP is usually used much more than it is changed, you will normally not be affected by this delay.

The structure of a JSP page is a cross between a servlet and an HTML page. The JSP tags begin and end with angle brackets, just like HTML tags, but the tags also include percent signs, so all JSP tags are denoted by

```
<% JSP code here %>
```

The leading percent sign may be followed by other characters that determine the precise type of JSP code in the tag.

Here's an extremely simple JSP example that uses a standard Java library call to get the current time in milliseconds, which is then divided by 1000 to produce the time in seconds. Since a JSP expression (the <%=) is used, the result of the calculation is coerced into a String and placed on the generated Web page:

```
//:! c15:jsp:ShowSeconds.jsp
<html><body>
<H1>The time in seconds is:
<%= System.currentTimeMillis()/1000 %></H1>
</body></html>
///:~
```

In the JSP examples in this book, the exclamation point in the first "comment tag line" means that the first and last lines will not be included in the actual code file that is extracted and placed in the book's source-code tree.

When the client creates a request for the JSP page, the Web server must have been configured to relay the request to the JSP container, which then invokes the page. As mentioned above, the first time the page is invoked, the components specified by the page are generated and compiled by the JSP container as one or more servlets. In the above example, the servlet will contain code to configure the HttpServletResponse object, produce a PrintWriter object (which is always named out), and then turn the time calculation into a String which is sent to out. As you can see, all this is accomplished with a very succinct statement, but the average HTML programmer/Web designer will not have the skills to write such code.

Implicit objects

Servlets include classes that provide convenient utilities, such as HttpServletRequest, HttpServletResponse, Session, etc. Objects of these classes are built into the JSP specification and automatically available for use in your JSP without writing any extra lines of code. The implicit objects in a JSP are detailed in the table below.

Implicit variable	**Of Type (javax.servlet)**	**Description**	**Scope**
request	**protocol dependent subtype of HttpServletRequest**	**The request that triggers the service invocation.**	**request**
response	**protocol dependent subtype of HttpServletResponse**	**The response to the request.**	**page**
pageContext	**jsp.PageContext**	**The page context encapsulates implementation-dependent features and provides convenience methods and namespace access for this JSP.**	**page**
session	**Protocol dependent subtype of http.HttpSession**	**The session object created for the requesting client. See servlet Session object.**	**session**

application	**ServletContext**	**The servlet context obtained from the servlet configuration object (e.g., getServletConfig(), getContext( ).**	**app**
out	**jsp.JspWriter**	**The object that writes into the output stream.**	**page**
config	**ServletConfig**	**The ServletConfig for this JSP.**	**page**
page	**java.lang.Object**	**The instance of this page's implementation class processing the current request.**	**page**

The scope of each object can vary significantly. For example, the session object has a scope which exceeds that of a page, as it many span several client requests and pages. The application object can provide services to a group of JSP pages that together represent a Web application.

JSP directives

Directives are messages to the JSP container and are denoted by the "@":

```
<%@ directive {attr="value"}* %>
```

Directives do not send anything to the out stream, but they are important in setting up your JSP page's attributes and dependencies with the JSP container. For example, the line:

```
<%@ page language="java" %>
```

says that the scripting language being used within the JSP page is Java. In fact, the JSP specification only describes the semantics of scripts for the language attribute equal to "Java." The intent of this directive is to build flexibility into the JSP technology. In the future, if you were to choose another language, say Python (a good scripting choice), then that language would have to support the Java Run-time Environment by exposing the Java technology object model to the scripting environment, especially the implicit variables defined above, JavaBeans properties, and public methods.

The most important directive is the page directive. It defines a number of page dependent attributes and communicates these attributes to the JSP container. These attributes include: language, extends, import, session, buffer, autoFlush, isThreadSafe, info and errorPage. For example:

```
<%@ page session="true" import="java.util.*" %>
```

This line first indicates that the page requires participation in an HTTP session. Since we have not set the language directive the JSP container defaults to using Java and the implicit script language variable named session is of type javax.servlet.http.HttpSession. If the directive had been false then the implicit variable session would be unavailable. If the session variable is not specified, then it defaults to "true."

The import attribute describes the types that are available to the scripting environment. This attribute is used just as it would be in the Java programming language, i.e., a comma-separated list of ordinary import expressions. This list is imported by the translated JSP page implementation and is available to the scripting environment. Again, this is currently only defined when the value of the language directive is "java."

JSP scripting elements

Once the directives have been used to set up the scripting environment you can utilize the scripting language elements. JSP 1.1 has three scripting language elements—declarations, scriptlets, and expressions. A declaration will declare elements, a scriptlet is a statement fragment, and an expression is a complete language expression. In JSP each scripting element begins with a "<%". The syntax for each is:

```
<%! declaration %>
<%  scriptlet   %>
<%= expression  %>
```

White space is optional after "<%!", "<%", "<%=", and before "%>."

All these tags are based upon XML; you could even say that a JSP page can be mapped to a XML document. The XML equivalent syntax for the scripting elements above would be:

```
<jsp:declaration> declaration </jsp:declaration>
<jsp:scriptlet>   scriptlet   </jsp:scriptlet>
<jsp:expression>  expression  </jsp:expression>
```

In addition, there are two types of comments:

```
<%-- jsp comment --%>
<!-- html comment -->
```

The first form allows you to add comments to JSP source pages that will not appear in any form in the HTML that is sent to the client. Of course, the second form of comment is not specific to JSPs—it's just an ordinary HTML comment. What's interesting is that you can insert JSP code inside an HTML comment and the comment will be produced in the resulting page, including the result from the JSP code.

Declarations are used to declare variables and methods in the scripting language (currently Java only) used in a JSP page. The declaration must be a complete Java statement and cannot produce any output in the out stream. In the Hello.jsp example below, the declarations for the variables loadTime, loadDate and hitCount are all complete Java statements that declare and initialize new variables.

```
//:! c15:jsp:Hello.jsp
<%-- This JSP comment will not appear in the
generated html --%>
<%-- This is a JSP directive: --%>
<%@ page import="java.util.*" %>
<%-- These are declarations: --%>
<%!
   long loadTime= System.currentTimeMillis();
   Date loadDate = new Date();
   int hitCount = 0;
%>
<html><body>
<%-- The next several lines are the result of a
JSP expression inserted in the generated html;
the '=' indicates a JSP expression --%>
<H1>This page was loaded at <%= loadDate %> </H1>
<H1>Hello, world! It's <%= new Date() %></H1>
<H2>Here's an object: <%= new Object() %></H2>
<H2>This page has been up
<%= (System.currentTimeMillis()-loadTime)/1000 %>
seconds</H2>
<H3>Page has been accessed <%= ++hitCount %>
times since <%= loadDate %></H3>
<%-- A "scriptlet" that writes to the server
console and to the client page.
Note that the ';' is required: --%>
<%
  System.out.println("Goodbye");
  out.println("Cheerio");
%>
</body></html>
///:~
```

When you run this program you'll see that the variables loadTime, loadDate and hitCount hold their values between hits to the page, so they are clearly fields and not local variables.

At the end of the example is a scriptlet that writes "Goodbye" to the Web server console and "Cheerio" to the implicit JspWriter object out. Scriptlets can contain any code fragments that are valid Java statements. Scriptlets are executed at request-processing time. When all the scriptlet fragments in a given JSP are combined in the order they appear in the JSP page, they should yield a valid statement as defined by the Java programming language. Whether or not they produce any output into the out stream depends upon the code in the scriptlet. You should be aware that scriptlets can produce side effects by modifying the objects that are visible to them.

JSP expressions can found intermingled with the HTML in the middle section of Hello.jsp. Expressions must be complete Java statements, which are evaluated, coerced to a String, and sent to out. If the result of the expression cannot be coerced to a String then a ClassCastException is thrown.

Extracting fields and values

The following example is similar to one shown earlier in the servlet section. The first time you hit the page it detects that you have no fields and returns a page containing a form, using the same code as in the servlet example, but in JSP format. When you submit the form with the filled-in fields to the same JSP URL, it detects the fields and displays them. This is a nice technique because it allows you to have both the page containing the form for the user to fill out and the response code for that page in a single file, thus making it easier to create and maintain.

```
//:! c15:jsp:DisplayFormData.jsp
<%-- Fetching the data from an HTML form. --%>
<%-- This JSP also generates the form. --%>
<%@ page import="java.util.*" %>
<html><body>
<H1>DisplayFormData</H1><H3>
<%
  Enumeration flds = request.getParameterNames();
  if(!flds.hasMoreElements()) { // No fields %>
    <form method="POST"
    action="DisplayFormData.jsp">
<%  for(int i = 0; i < 10; i++) {  %>
      Field<%=i%>: <input type="text" size="20"
      name="Field<%=i%>" value="Value<%=i%>"><br>
<%  } %>
    <INPUT TYPE=submit name=submit
    value="Submit"></form>
<%} else {
    while(flds.hasMoreElements()) {
      String field = (String)flds.nextElement();
      String value = request.getParameter(field);
%>
      <li><%= field %> = <%= value %></li>
<%  }
  } %>
</H3></body></html>
///:~
```

The most interesting feature of this example is that it demonstrates how scriptlet code can be intermixed with HTML code, even to the point of generating HTML within a Java for loop. This is especially convenient for building any kind of form where repetitive HTML code would otherwise be required.

JSP page attributes and scope

By poking around in the HTML documentation for servlets and JSPs, you will find features that report information about the servlet or JSP that is currently running. The following example displays a few of these pieces of data.

```
//:! c15:jsp:PageContext.jsp
<%--Viewing the attributes in the pageContext--%>
<%-- Note that you can include any amount of code
inside the scriptlet tags --%>
<%@ page import="java.util.*" %>
<html><body>
Servlet Name: <%= config.getServletName() %><br>
Servlet container supports servlet version:
<% out.print(application.getMajorVersion() + "."
+ application.getMinorVersion()); %><br>
<%
  session.setAttribute("My dog", "Ralph");
  for(int scope = 1; scope <= 4; scope++) {  %>
    <H3>Scope: <%= scope %> </H3>
<%  Enumeration e =
      pageContext.getAttributeNamesInScope(scope);
    while(e.hasMoreElements()) {
      out.println("\t<li>" +
        e.nextElement() + "</li>");
    }
  }
%>
</body></html>
///:~
```

This example also shows the use of both embedded HTML and writing to out in order to output to the resulting HTML page.

The first piece of information produced is the name of the servlet, which will probably just be “JSP” but it depends on your implementation. You can also discover the current version of the servlet container by using the application object. Finally, after setting a session attribute, the “attribute names” in a particular scope are displayed. You don’t use the scopes very much in most JSP programming; they were just shown here to add interest to the example. There are four attribute scopes, as follows: The page scope (scope 1), the request scope (scope 2), the session scope (scope 3—here, the only element available in session scope is “My dog,” added right before the for loop), and the application scope (scope 4), based upon the ServletContext object. There is one ServletContext per “Web application” per Java Virtual Machine. (A “Web application” is a collection of servlets and content installed under a specific subset of the server’s URL namespace such as /catalog. This is generally set up using a configuration file.) At the application scope you will see objects that represent paths for the working directory and temporary directory.

Manipulating sessions in JSP

Sessions were introduced in the prior section on servlets, and are also available within JSPs. The following example exercises the session object and allows you to manipulate the amount of time before the session becomes invalid.

```
//:! c15:jsp:SessionObject.jsp
<%--Getting and setting session object values--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H3><li>This session was created at
<%= session.getCreationTime() %></li></H1>
<H3><li>Old MaxInactiveInterval =
  <%= session.getMaxInactiveInterval() %></li>
<% session.setMaxInactiveInterval(5); %>
<li>New MaxInactiveInterval=
  <%= session.getMaxInactiveInterval() %></li>
</H3>
<H2>If the session object "My dog" is
still around, this value will be non-null:<H2>
<H3><li>Session value for "My dog" =
<%= session.getAttribute("My dog") %></li></H3>
<%-- Now add the session object "My dog" --%>
<% session.setAttribute("My dog",
                  new String("Ralph")); %>
<H1>My dog's name is
<%= session.getAttribute("My dog") %></H1>
<%-- See if "My dog" wanders to another form --%>
<FORM TYPE=POST ACTION=SessionObject2.jsp>
<INPUT TYPE=submit name=submit
Value="Invalidate"></FORM>
<FORM TYPE=POST ACTION=SessionObject3.jsp>
<INPUT TYPE=submit name=submit
Value="Keep Around"></FORM>
</body></html>
///:~
```

The session object is provided by default so it is available without any extra coding. The calls to getID( ), getCreationTime( ) and getMaxInactiveInterval( ) are used to display information about this session object.

When you first bring up this session you will see a MaxInactiveInterval of, for example, 1800 seconds (30 minutes). This will depend on the way your JSP/servlet container is configured. The MaxInactiveInterval is shortened to 5 seconds to make things interesting. If you refresh the page before the 5 second interval expires, then you'll see:

```
Session value for "My dog" = Ralph
```

But if you wait longer than that, “Ralph” will become null.

To see how the session information can be carried through to other pages, and also to see the effect of invalidating a session object versus just letting it expire, two other JSPs are created. The first one (reached by pressing the “invalidate” button in SessionObject.jsp) reads the session information and then explicitly invalidates that session:

```
//:! c15:jsp:SessionObject2.jsp
<%--The session object carries through--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<% session.invalidate(); %>
</body></html>
///:~
```

To experiment with this, refresh SessionObject.jsp, then immediately click the “invalidate” button to bring you to SessionObject2.jsp. At this point you will still see “Ralph,” and right away (before the 5-second interval has expired), refresh SessionObject2.jsp to see that the session has been forcefully invalidated and “Ralph” has disappeared.

If you go back to SessionObject.jsp, refresh the page so you have a new 5-second interval, then press the “Keep Around” button, it will take you to the following page, SessionObject3.jsp, which does NOT invalidate the session:

```
//:! c15:jsp:SessionObject3.jsp
<%--The session object carries through--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<FORM TYPE=POST ACTION=SessionObject.jsp>
<INPUT TYPE=submit name=submit Value="Return">
</FORM>
</body></html>
///:~
```

Because this page doesn’t invalidate the session, “Ralph” will hang around as long as you keep refreshing the page before the 5 second time interval expires. This is not unlike a “Tomagotchi” pet—as long as you play with “Ralph” he will stick around, otherwise he expires.

Creating and modifying cookies

Cookies were introduced in the prior section on servlets. Once again, the brevity of JSPs makes playing with cookies much simpler here than when using servlets. The following example shows this by fetching the cookies that come with the request, reading and modifying their maximum ages (expiration dates) and attaching a new cookie to the outgoing response:

```
//:! c15:jsp:Cookies.jsp
<%--This program has different behaviors under
 different browsers! --%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<%
Cookie[] cookies = request.getCookies();
for(int i = 0; i < cookies.length; i++) { %>
  Cookie name: <%= cookies[i].getName() %> <br>
  value: <%= cookies[i].getValue() %><br>
  Old max age in seconds:
  <%= cookies[i].getMaxAge() %><br>
  <% cookies[i].setMaxAge(5); %>
  New max age in seconds:
  <%= cookies[i].getMaxAge() %><br>
<% } %>
<%! int count = 0; int dcount = 0; %>
<% response.addCookie(new Cookie(
    "Bob" + count++, "Dog" + dcount++)); %>
</body></html>
///:~
```

Since each browser stores cookies in its own way, you may see different behaviors with different browsers (not reassuring, but it might be some kind of bug that could be fixed by the time you read this). Also, you may experience different results if you shut down the browser and restart it, rather than just visiting a different page and then returning to Cookies.jsp. Note that using session objects seems to be more robust than directly using cookies.

After displaying the session identifier, each cookie in the array of cookies that comes in with the request object is displayed, along with its maximum age. The maximum age is changed and displayed again to verify the new value, then a new cookie is added to the response. However, your browser may seem to ignore the maximum age; it's worth playing with this program and modifying the maximum age value to see the behavior under different browsers.

JSP summary

This section has only been a brief coverage of JSPs, and yet even with what was covered here (along with the Java you've learned in the rest of the book, and your own knowledge of HTML) you can begin to write sophisticated web pages via JSPs. The JSP syntax isn't meant to be particularly deep or complicated, so if you understand what was presented in this section you're ready to be productive with JSPs. You can find further information in most current books on servlets, or at java.sun.com.

It’s especially nice to have JSPs available, even if your goal is only to produce servlets. You’ll discover that if you have a question about the behavior of a servlet feature, it’s much easier and faster to write a JSP test program to answer that question than it is to write a servlet. Part of the benefit comes from having to write less code and being able to mix the display HTML in with the Java code, but the leverage becomes especially obvious when you see that the JSP Container handles all the recompilation and reloading of the JSP for you whenever the source is changed.

As terrific as JSPs are, however, it’s worth keeping in mind that JSP creation requires a higher level of skill than just programming in Java or just creating Web pages. In addition, debugging a broken JSP page is not as easy as debugging a Java program, as (currently) the error messages are more obscure. This should change as development systems improve, but we may also see other technologies built on top of Java and the Web that are better adapted to the skills of the web site designer.

Exercises

12. Create a JSP page that prints a line of text using the <H1> tag. Set the color of this text randomly, using Java code embedded in the JSP page. If you do not have an existing JSP container, you will need to download, install, and run Tomcat from jakarta.apache.org in order to run JSPs.

13. Modify the maximum age value in Cookies.jsp and observe the behavior under two different browsers. Also note the difference between just re-visiting the page, and shutting down and restarting the browser. If you do not have an existing JSP container, you will need to download, install, and run Tomcat from jakarta.apache.org in order to run JSPs.

14. Create a JSP with a field that allows the user to enter the session expiration time and and a second field that holds data that is stored in the session. The submit button refreshes the page and fetches the current expiration time and session data and puts them in as default values of the aforementioned fields. If you do not have an existing JSP container, you will need to download, install, and run Tomcat from jakarta.apache.org in order to run JSPs.

Custom Tags

Custom Tags provide you with the full power of the Java language with almost seamless integration into your presentation layer[6].

 A custom tag is a homemade JSP tag. In order to fully understand the implications of this, you first need to be clear about what a tag is.

The chances are that you will have made good use of tags long before reading this chapter. Very few developers these days work in environments which are completely divorced from the Internet, and that means you have probably had at least some basic HTML or XML exposure.

 In a mark-up language, like HTML, or XML, the structure is embedded within the data. This allows the parser, e.g. a browser, to interpret instructions for displaying or storing the information.

The main mechanism for this is the tag.

At its most simple, a tag, like for instance, <b> just tells the reader (in this case, a web browser) to apply some formatting to the data. With this particular tag, it would be to display the data (which is plain text) in bold format. Every tag should have a start and end point i.e. in this case </b> which allows the parser of the mark-up language to identify a portion of content, by determining which of the data lies between the start and end tags. This allows some structure or format to be applied to that data. In this case any text between the start and end tags would be displayed in bold font.

The other important mechanism for applying this structure to the data, is the attribute, making the tag slightly more complicated, like the “table” tag in HTML, for example. A table tag can have a border attribute e.g. <table border=”3”>, this attribute value, border, applies to the data which appears after the <table> tag and before the </table> tag. This data found in between the two tags is referred to as the body of the tag. All data in the body of this tag would be displayed in a table, and the border around each element in the table would be of width “3”.

In the case of the table tag in HTML, it gets more complicated still, because the body of the tag can contain other tags as well as data, allowing nesting. Thus you can define a table row within a table, and table data (or columns) within a row. This allows for very good structuring of the data.

e.g.

```
<table border=”3”>
  <tr>
    <td>black</td>
    <td>white</td>
  </tr>
</table>
```

This defines a table, with a border of size three, one row and two columns, containing data black and white. The words black and white are the data items which make up the body of the <td> or table data tag. The <tr> or table row tag with its embedded <td> tags and data make up the body of the table tag. You can clearly see that the relationship between these tags is important. The table row tag, <tr>, doesn't have much meaning outside of a table tag, and similarly, the table data tag, <td>, has no meaning outside of the table row tag. We refer to the table tag as being the parent of the table row tag, and the table row tag as being a parent of the table data tag.

XML introduces the notion of namespaces which allow you a little more flexibility again when naming tags. (see the XML chapter "" for more detail). If you want to create your own HTML-type mark up language, you could define a namespace called mytags and define your tag within it, thus having a tag <mytag:table> and </mytag:table>. Put simply, this would mean that there would be no confusion between the original HTML table tag and your own, newly defined one. Your table tag is in its own namespace. This means that you can use consistent naming conventions, e.g. call a "table" a "table", rather than "myspecialtable", without having to worry about it clashing with the HTML standard definition.

Custom tags have all of the above properties. They have bodies that can either contain data, or might be empty and they have properties that can be applied to the body of the tag. They also use namespaces to allow consistent naming conventions. You will use some or all of these features, depending on what is appropriate for your application.

What do custom tags give us?

Power: As with JavaBeans, the real programming problems like querying databases or making complex calculations etc. can all be done in Java code with the full power of the Java libraries behind you. JSP custom tags also allow you to make objects available to a web page author via fragments of Java code, or "scriptlets", so with careful design, you can make a lot of functionality available via a simple and neat interface.

Consistency: Unlike JavaBeans, the presentation code is neatly separated from the business logic code, the need for scriptlets is reduced and more development is done using HTML type tags. It is therefore easier for a web page author to understand, and much easier to read and debug.

Encapsulation: The business logic, is now defined in Java classes that can solve complex programming problems like network communication or database querying. This code can be unit tested in isolation, and finally presented as simple tags, ideal for a web author to use when getting on with building the web application.

Reuse: Tags are stored in libraries which are perfect entities for re-use in web development. Tag libraries typically contain a collection of tags of similar or connected functionality. The library can easily be included by a web page author needing that functionality. A tag is even easier to use than your average Java method!

At this point, if you remember the earlier discussions about using Java beans from JSP in the JSP chapter, you might be asking yourself why this is necessary. Javabeans in JSP also help us to separate the presentation layer from the business logic. Beans can be stored in libraries too, they are reusable, have a consistent interface and can even communicate with scriptlets, so why do we need tags?

The answer is quite subtle. Beans are indispensable to the application developer and are very helpful if used in the right way. In fact, as we shall see, beans and custom tags complement each other very nicely. The difference, though, is that beans are components, and custom tags are essentially actions, or handlers. This needs to be understood well and carefully considered in your design if it is not to lead to the incorrect use of beans.

Beans, like any other components, have properties, methods and internal state. What makes using beans with JSP so easy is that the interface between JSP pages and Java beans is designed specifically for getting and setting the bean properties. As long as you have named your bean methods according to the Javabeans convention, you can access these properties very simply via JSP.

Let’s revisit Javabeans. Consider a person bean that has two properties, name and age. In order to access these via JSP tags, you would need to use the

<jsp:usebean ...> action, and the

<jsp:getProperty name=”beanName” property=”requiredProperty”> action for each of these two values. Should you want to put these in the row of an HTML table, you would have to specify

```
..
<table>
<tr>
<td><jsp:getProperty name=”personbean” property=”name”></td>
<td><jsp:getProperty name=”personbean” property=”age”></td>
</tr>
</table>
..
```

This is good-looking, consistent HTML type code. It is very manageable at the moment, but will get more ugly and “bloated” as you use beans with more properties. You might have five beans, with five properties each, and instantly you would have twenty five lines to type. In a real application you are very unlikely to have as few as two properties, as we do in the example above. What is needed here is clearly some programming so we don’t have to do so much typing and create so much source code.

One of the great strengths of using JavaBeans with JSP is also a limitiation. Because of the JSP getProperty and setProperty tags, it is very easy to get and set the bean properties. Remember, these are short cuts for using the set and get methods. But, if you are trying to call other methods on the bean, you have to do it directly and you are confined to using scriptlet code or expression code, using the <% %> tags. This starts to make the code look a little ugly and inconsistent. Consider the fragment.

```
<ol>
<% while (list.hasNextElement()) {
  list.nextElement();
%>
<li><jsp:getProperty name=”list” property=”currentElement”></li>
<% } %>
</ol>
```

In the example above, notice that to iterate through a simple loop, and display items from a list (using an imagined list bean which iterates through its data) the first two lines of the Java scriptlet code have to be put in scriptlet brackets. The jsp:getProperty tag which allows the use of HTML type brackets gives consistent looking presentation code, but then the close brace of the while loop has to use the scriptlet brackets, <%}%> again. Two tags are now enclosing a single close bracket and as you can see, it is not very nice to look at. If you were trying to debug it, it might be very difficult to work out where one loop ends and the next starts. JSP debuggin tools are improving but still leave a lot to be desired. Add to this the fact that it is almost impossible to arrive at a consistent convention for indenting your brackets and you can see it is potentially messy.

Some scriptlet code is inevitable, but too much is undesirable. As the complexity of the code increases, so the combination of HTML tags, JSP directives, actions and scriptlet code, each with its own bracket notation, further increases the complexity of the code, making it difficult to read and maintain.

How can this be prevented? It is very tempting to write a bean that returns values that are already in tags. This would at first glance appear to remove a great deal of the HTML code and keep the documents shorter and simpler. Implemented like this, a bean could just return a string value from the get method that already has formatting tags in it and our data would display itself nicely on the screen.

As attractive as this sounds, it turns out to be a very bad idea as you will see.

Consider a bean which has a String method which returns a table row of data including the tags. It might look something like this..

```
public class PersonDataBean  {
  private int age;
  private String name;
  String getCurrentRow() {
    return  “<tr><td>” + name + “</td><td>”+ age +
    “</td></tr>”;
  }
}
```

The getCurrentRow() method now returns a string which displays a table row, and the HTML which displayed the person in a table, two examples above, becomes simply:

```
..
<table>
jsp:getProperty name=”personbean” property=”currentRow”>
</table>
..
```

This might makes for shorter HTML coding but it is a poor design approach for a number of reasons. For one thing, it creates a very tight cohesion between the data and its presentation. That tends to cloud one’s ability to think of the bean as an object, and clutter one’s design.

From a programming perspective, it is even worse. A programmer using this bean would get the results back in a table whether or not it was appropriate. The layout and the field order wouldn’t be changeable programmatically and although a stylesheet could be used, to change the style of the output, it would not be possible to do anything more specific, (like make one column appear in bold text, for example).

This means that the web page author using the bean is now limited to presenting data in a format defined by the bean developer. The bean developer might be an excellent database developer, but might not be the person responsible for presenting the final layout. In fact, it is very likely that the layout requirements are not even known at the time of the bean’s development. The bean should have been written with reuse in mind, and no one designing a web page would appreciate having to work with a pre-determined layout.

So, although this appears to solve the problem of increased code complexity, it in fact detracts enormously from the code’s re-usability.

You will always come across these layout issues with dynamic web content. When developing web pages, you constantly need layouts or presentations that are dependent on certain program conditions (like variable length tables, different colors for values within certain ranges etc.) This is the dynamic behavior one would expect from a web page, but this behavior makes the code hard to read by requiring scriptlet programming with funny brackets embedded in the code.

Scriptlet code obviously can’t be removed from JSP pages completely, and nor would you want that. In fact, writing scriptlets, with JavaBeans, (or any kind of scripting language with any kind of component) is an excellent way to build an application, as long as it is not abused. So far, we have highlighted two problems, the need to separate presentation code from business logic, and the need to prevent the kind of complexity that creeps into our presentation code when we mix HTML and scriptlet code. We can also see, that as useful as JavaBeans are, they are not always the solution to these problems and may lead to bad design. That is where the custom tag comes into play.

A JSP tag can be thought of as an action, or a handler for a number of events. These events are actually triggered by the JSP container reading the JSP source and parsing each tag. The different methods that correspond to these events, are amongst others, doStartTag() and doEndTag(), the point at which they are called should be obvious from their names.

There are of course more than just 2 methods you will need to create custom tags, but when writing our own tags we have various classes and interfaces we can use which provide us with more or less functionality depending on our needs.

Let us have a look at a simple tag, a "Hello World" tag will do nicely.

What would be ideal is a tag that would display Hello World. The tag should look something like this:

```
<tijtags:helloworldtag/>
```

Note that its namespace is tijtags and its name is helloworldtag. Note also that there is a forward slash at the end of the tag name. This means that the tag is a start and end tag all in one. This makes use of the short hand notation defined in XML. It could also be written like this:

```
<tijtags:helloworldtag>
</tijtags:helloworldtag>
```

In either case, the tag clearly has no body, and as you might have guessed, needs the very minimum of programming to make it work. You will need to tell the JSP container how to find your Java classes, though, so in order to do this bare minimum the following tasks need to be performed:

1. Write a class that implements the Tag interface, found in the package javax.servlet.jsp.tagext.Tag
2. implement the methods "setPageContext(), doStartTag() and doEndTag()"
3. create a TLD file, or tag library descriptor for the tag.

Let's take care of the first 2 tasks:

```
package cx2.tags;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.JspException;
import java.io.IOException;

public class HelloWorldTag implements Tag {
  PageContext pageContext;
  public void setPageContext(PageContext pageContext) {
    this.pageContext = pageContext;
  }
  public void setParent(Tag parent) {}

  public Tag getParent() { return null; }
  public int doStartTag() throws JspException {
    try {
      pageContext.getOut().println("Hello World!");
    } catch(IOException ioe) {
      ioe.printStackTrace();
    }
    return Tag.SKIP_BODY;
  }
  public int doEndTag() throws JspException {
    return Tag.EVAL_PAGE;
  }
  public void release() {}
}
```

You will notice that there is no constructor defined, which means that this class has a default, parameterless constructor. This is one of the prerequisites of writing a tag handler class.

You will notice that a property called pageContext has been defined, which is of type PageContext, found in the javax.servlet.jsp package. This is initialized in the method setPageContext(). The container calls this automatically and passes through a reference to the PageContext when it first reads the tag. This reference has to be stored because it is needed later on. The JSP Tag Extension specification guarantees that the setPageContext() method will be always be called before the doStartTag() method.

Next, the method doStartTag() is implemented. This method is where all of the work for this particular tag is done.

Using the pageContext object, which has already been initialized, the JSP output stream is requested. This is the output stream that echoes back to the browser which is reading the JSP page, and its println() method is called, in exactly the same way it would be if you were using System.out.

Notice that the doStartTag() method returns the value Tag.SKIP_BODY. This tells the container that it should ignore anything between the start and end tags. Since this tag does not even have a body this is all that is needed for now.

The doEndTag() method is implemented simply by returning the value Tag.EVAL_PAGE. This tells the container to continue processing the JSP page as it was until it came across this tag. Again, with a tag this simple, that is all that is needed.

There are some other methods of the Tag interface that must be implemented, but only the three mentioned above are important for now.

All that is left to do now for the HelloWorld tag, is to describe to the container what your new tag is called and which class will be implementing it. To do this you need to write a tag library descriptor, an XML file which the container uses to gain information about the tag or tags you have defined. This file is saved with a .tld extension and is placed on your server with your JSP files.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
        PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
   "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>htl</shortname>
  <uri>http://www.eckelobjects.com/tagtut/tags1.0</uri>
  <info> Library of tags on increasing complexity to demonstrate the use of tag libraries.</info>
  <tag>
    <name>helloworldtag</name>
    <tagclass>cx2.tags.HelloWorldTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info> echo's hello world to the browser.
    </info>
  </tag>
</taglib>
```

Don’t be intimidated by all of this. The tags in this file are less arcane than they seem. They will be presented in greater detail later on, for now, all you need to worry about is the tag tag. This tag, unsurprisingly, tells the container things it needs to know about the new tag that you have just created. There are four things that are important with this particular tag; its name, which class contains the tag implementation, what kind of its body content it should have and lastly, a little description of what the tag does. The description should be short and clear, the kind of thing you might expect to see on the toolbar of a GUI tool giving you the option to use this tag in your application.

To use the new tag in a JSP page you would specify something like this:

```
<%@ taglib uri="/tijtags.tld" prefix="tijtags"%>
<b>
<tijtags:helloworldtag/>
</b>
```

That is all you need to do to send “Hello World” back to the browser!

The first line is the JSP taglib directive. The uri attribute tells the container that your tag library descriptor can be found in a file called “tijtags.tld” in the current directory. The prefix attribute tells the container what namespace you are going to use to refer to your tags.

The <b> tag is just for decoration and to emphasize the point that you can use HTML freely with JSP tags. This combination becomes very powerful when you are writing iterative tags.

Using Tags and JavaBeans

The “hello world” tag was not a very powerful tag. It certainly didn’t save anyone any effort. In this case, it would have course been easier just to type the words “Hello World” into the text yourself. Let’s look at an example that does a little bit more in the way of programming, and at the same time illustrates how tags can make objects available to the JSP page.

This time, the intention is to create a greeting tag that is slightly more useful than “hello world”, and therefore a little more general. Imagine a greeting tag, that when used would create a greeting object and make it available on the JSP page. If this object, a Javabean, had a greeting property that produced a random greeting, such as “Hi there!” or “Hello”, you might use it something like this..

```
<tijtags:greetingtag id="randomgreeting"/>
<jsp:getProperty name="randomgreeting" property="greeting"/>
```

This is a little bit more involved than the last tag. Let’s look closely at the differences.

Firstly, although this is another bodiless tag, it has an attribute, called id. The tag reference above passes the string “randomgreeting” through to the tag.

Next there is a JSP action, getProperty which appears to be requesting the value of the property “greeting” from a Javabean called “randomgreeting”. It is no coincidence that the name of this bean is the same as the value of the “id” attribute, you might have guessed by now that it is the tag that has enabled the getProperty action to be used on a JavaBean by this name. Usually you would expect to see the JSP action <jsp:useBean... declared at the top of the JSP file before you could use the getProperty action to access a bean, but in this case it is the tag that has effectively defined the bean and made it accessible from the page.

The tasks needed for this tag to work would be the following:

1. Create a Javabean with a property greeting, and a get method that returns a random greeting string, like “Hi there!” or “Hello”.

2. Write a class that implements the Tag interface.

3. Give the class a setId() method to allow the container to set the id attribute.

4. Create an instance of the greeting JavaBean and store it in the page context.
5. create a TLD or tag library descriptor file for the tag.

There are a few more things to do this time, but this is still pretty straightforward. You already know how to write a JavaBean, but here is a minimal one that will be sufficient.

```
package cx2.tags;

public class GreetingBean  {
  String greetings[] = {"Hello!","Howdy!","Hi There!","How do you do?"};
   public String getGreeting() {
      return greetings[(int)(Math.random()*greetings.length)];
   }
}
```

The idea is that the tag class will make an instance of this JavaBean available so that the JSP page author can use the getProperty action on his page.

Here is the Tag class itself.

```
package cx2.tags;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.JspException;

public class GreetingTag implements Tag {
   private String id;
   private PageContext pageContext;
   public GreetingTag() {
   }
    public void setId(String id) {
       this.id = id;
   }
   public void setPageContext(PageContext pageContext) {
      this.pageContext = pageContext;
   }
   public void setParent(Tag parent) {
   }
   public Tag getParent() {
      return null;
   }
   public int doStartTag() throws JspException {
       pageContext.setAttribute(id, new GreetingBean(),
                              PageContext.PAGE_SCOPE);
        return Tag.SKIP_BODY;
   }
   public int doEndTag() throws JspException {
       return Tag.EVAL_PAGE;
   }
   public void release() {
  }
}
```

You will notice that there is a setId() method. This is not on the Tag interface, but the container will call this method anyway. It will be passed the value of the attribute id as it is specified in the JSP file. This highlights another interesting relationship between JavaBeans and tags. A tag which has attributes must have JavaBean style properties which obey the correct naming convention and must match the attribute names. These properties will be set automatically by the container, using the JavaBeans mechanism built into Java. Two additional rules not normally associated with JavaBeans, but which are important here, are that you may only have one setter method per attribute, and you should not call a setter method yourself, i.e. from anywhere else in the tag code.

As before, most of the work for this tag is done in the doStartTag() method. It is worth paying slightly closer attention this time, to the fact that the doStartTag() and doEndTag() methods may each throw a JspException. This can be ignored for the time being, but if an exception occurs in your tag implementation, a JspException will be thrown which might end up being displayed on the client browser window, which is rarely desirable. A technique for handling this will be explained later.

This time the doStartTag() method uses the setAttribute() method on the pageContext object to create an instance of the greeting bean, and store it in the pageContext, specifying its name and its scope. The name comes from the id attribute, and the scope is the familiar constant PAGE_SCOPE, a final property of the class PageContext. If that sounds complicated, think of it in two parts. The tag class has already been passed the name via the setId() method, and stored it in the variable id. Now it creates a GreetingBean object and adds it to the pageContext, giving it the name stored in the id variable. This is the programmatic equivalent of the JSP action: <jsp:useBean...>. The JavaBean is now available for use with the <jsp:getProperty..> action on the JSP page.

All that is left to do now, is to create the TLD file. Since one exists already, all you need to do is add another tag tag before the final </taglib>

```
..
  <tag>
    <name>greetingtag</name>
    <tagclass>cx2.tags.GreetingTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info> Adds a HelloWorld Bean to the page context. </info>
    <attribute>
      <name>id</name>
      <required>true</required>
    </attribute>
  </tag>
..
```

Notice that in this tag tag there is a subtag, called attribute. The name subtag of attribute tells the container what that attribute is called, and the required subtag tells the container whether that attribute is mandatory or not. Mandatory subtags must be specified, i.e in this case,

```
<tijtags:greetingtag id="randomgreeting"/>
```

otherwise the container will throw an exception when an attempt is made to load the JSP page.

In order for your tag to properly declare and initialize an attribute, the name of that attribute, the name in the tag library descriptor file and the name of the JavaBean property (and setter methods) must all be the same.

To use the random greeting tag, and the bean object it creates, you would need to specify the tag:

```
<tijtags:greetingtag id="randomgreeting"/>
```

which would create a JavaBean in the page context for you to use.

Then you would need to get the property from the bean:

```
<jsp:getProperty name="randomgreeting" property="greeting"/>
```

which would echo the greeting back to the browser. This latter line can then be used as often as you like on the JSP page after the declaration above it. Each time it appears, it will produce a different greeting. Refreshing the page in your browser will also produce new random greetings, because every time the container processes the getProperty action, it will make a call to the method getGreeting() on the GreetingBean class, which will return a new, random greeting. (Note that that the use of a get method on a JavaBean that returns a random value might offend some purists. It is used here as a good demonstration of working with beans.)

Tags that manipulate their body content

Up until now, the two tags that we have looked at ignore their body content, or to be more accurate, they are defined as not having any body content at all. This kind of tag is very simple, and fairly limited in its usefulness, although, as you can imagine, making a bean available to the page author can be pretty useful. Tags which manipulate their body content are generally much more powerful, if slightly harder to program.

Let’s look at a tag that actually interacts with its body content.

This time, suppose you wanted to generate a tag that gives you a nice looking drop capital for your text. For example..

It was a dark and stormy night...

This can be done in HTML in a somewhat primitive fashion, by using heading text and tables, which serves our purposes, but rather than write out a table every time you want a drop capital, why not create a tag to do it for you? Ideally it should be possible to do something like the following:

```
<tijtags:dropcapitaltag color = "black">
```

it was a dark and stormy night, the rain fell in torrents -- except at occasional intervals, when it was checked....

```
</tijtags:dropcapitaltag>
```

The color attribute would be a nice thing to have so that a giant capital could be displayed in the specified color.

To create this tag,. the important tasks you would need to perform would be the following:

Write a class that implements the Tag interface.

Give the class a setColor() method to allow the container to set the color attribute.

Write the code that reads in the body content and inserts the relevant HTML tags so that the first letter is displayed as a drop capital.

Create a tag library descriptor file for the tag.

Lets have a look at the code for this tag:

```
package cx2.tags;
import javax.servlet.jsp.tagext.BodyTagSupport;
import javax.servlet.jsp.tagext.TryCatchFinally;
import javax.servlet.jsp.JspException;
import java.util.StringTokenizer;
public class DropCapitalTag extends BodyTagSupport implements TryCatchFinally {
  String color = "black";

  public void setColor(String color) {
    this.color = color;
  }

  public int doAfterBody() throws JspException {
    int period;
    StringBuffer result = new StringBuffer();
    String body = bodyContent.getString().trim();
    result.append("<table><tr><td><h1><font color=\""+color+"\">");
    result.append(Character.toUpperCase(body.charAt(0)));
    result.append("</font></h1></td><td>");
    period = body.indexOf('.');
    result.append (body.substring(1, period+1));
    result.append ("</td></tr></table>");
    result.append (body.substring(period +1, body.length()));
    try {
      if (result.length() > 0) {
        bodyContent.clearBody();
        bodyContent.println(result.toString());
      }
      bodyContent.writeOut(bodyContent.getEnclosingWriter());
    } catch(Exception e) {
      e.printStackTrace();
    }
    return SKIP_BODY;
  }
  public void doCatch(Throwable t) {
    System.out.println(
      "An error occurred with the message" +
      t.getMessage());
  }
  public void doFinally() {}
  public void release() {
    color = "black";
  }
}
```

The first thing you will notice is that unlike the previous tags, this one does not implement the Tag interface. This is because when you need to manipulate your body content, it is more useful to extend the BodyTagSupport class, which takes care of storing the pageContext object and providing default implementations of the doStartTag() and doEndTag() methods. This class implements the Tag interface so you don't have to.

The doAfterBody() method of this tag is where everything happens. You will see that this class references a handle called bodyContent. This is a reference to the BodyContent object, which is found in the javax.servlet.jsp.tagext package. This is provided by the container via a setBodyContent() method and is something else we get for free when we extend the BodyTagSupport class. This container will always provide this object before it calls the doInitBody() method, so you are always guaranteed that you have a valid BodyContent object in your doInitBody(), doAfterBody(), doStartTag and doEndTag() methods.

The BodyContent object is an encapsulation of the body content of this particular tag, it contains all of the text from the JSP page that is in between the start and end tag. It is a quite a lot like an output stream, but with some extra functionality. This object gives you access to the JSP text. This is what the drop capital tag does, and is what you will need to do if you are going to manipulate the body content. You can get the entire body as a string, via a call to the getString() or as a stream, using the getReader() method if that is more appropriate for your needs.

In this example a simple algorithm is used to trim off any white space that comes from the JSP source, capitalize the first letter, and add a few HTML tags, to format it into a drop capital effect.

After the HTML tags have been added (using a StringBuffer object to append strings) there is one new, long string which needs to replace the old body text. To do this, the method clearBody() on the BodyContent is called first. If you don't do this, the body content will appear twice on the browser. Next we use the println() method to add our new string to the BodyContent, and finally, the writeOut() method is called. This method tells the BodyContent object to direct its output back to the browser, but you will notice that the method takes a JspWriter as a parameter, and in this case the writer is obtained by a call to getEnclosingWriter(). You have to do this, because the body content object only includes the body of your particular tag and you need access to the output stream that directs output back to the client's browser.

If you look closely, you will see that the release() method does nothing. This method will be called by the container to release its state, so it is best used to tidy up any resources that you might have used in the tag. Be aware that there might be multiple calls to doStartTag() and doEndTag() in between each call to release, so this is not the method to use if you want to perform an operation each time the tag processing has completed. If you want to do that, you are better off using the doFinally(), described in the next paragraph.

You might have noticed that this class implements the TryCatchFinally interface, which is in the javax.servlet.jsp.tagext package. When the container sees that your tag implements this interface, it knows to put a try/catch block around your tag handler, and passes the processing on to you if an exception occurs. This means that you have the ultimate control over the exception handling code, and if you wish, you can prevent weird exceptions being displayed on the browser window.

The methods on the TryCatchFinally interface are doCatch() and doFinally() and they correspond to the catch and finally keywords in Java. The doCatch() method is called if there is an exception with the exception passed as a parameter, and the doFinally() is always called, regardless of what happens in the other tag handler methods. In this example, if there is an exception, the message is just displayed on the console on the server, not the client browser. This is a slightly lazy design approach used to demonstrate the interface. The algorithm used here is not very robust so an exception could easily occur. In the doFinally() method above, the color is set to black so that that is always the default value.

Use this interface with caution. Its chief purpose is to give you more control when handling resources, like open streams etc. Remember that exceptions are all about recovery and it might be more appropriate to handle exceptions in your doAfterBody(), or doInitBody() methods and produce some meaningful output to display on the client's browser, for example an error page which asks the user to re-enter some details. You might just want to forward the browser to a generic error page if that is more appropriate.

Now that you have seen how the class is implemented, lets look at the tag entry in the tag library descriptor file. It looks like this:

```
..
<tag>
   <name>dropcapitaltag</name>
   <tagclass>cx2.tags.DropCapitalTag</tagclass>
   <bodycontent>JSP</bodycontent>
   <info> Transforms text to have a drop capital.
   </info>
   <attribute>
     <name>color</name>
     <required>false</required>
   </attribute>
 </tag>
..
```

Notice that the color attribute is not required. As you can see in the source code, the tag will default to black if no color is specified, so it is not enforced here.

That is all you need to do to manipulate the body of a tag!

Tags that iterate

One of the most powerful features of custom tags in JSP is that they can iterate like a for loop or while loop in Java and produce dynamic output this way.

Lets look at a more complicated tag that actually iterates over its body content, producing variable output depending on its attributes.

Imagine you wanted to generate a list of prime numbers (numbers divisible only by themselves and 1) and display them in an HTML unordered list. What you would really like is a tag that would generate all of the prime numbers within a specified range, and allow you to access each value and put it in your preferred list format. Since you don't know how many numbers there will be within the given range, and there could be quite a few, you don't want to have to type out the HTML list item tags, <li> and </li>, for each one. Assuming that your tag is called primetag, you would want your JSP file to look something like this:

```
<ul>
<tijtags:primetag start = "1" end= "25">
<li><%=value%></li>
</tijtags:primetag>
</ul>
```

Notice that there is a start and end <ul> HTML tag round the entire block. **(An HTML lesson is beyond the scope of this chapter, but briefly, this defines an unordered list and within the body of this tag, any data found between a <li> and </li> pair, will be displayed as a bulleted list item.)

Next you will notice the now familiar tijtags namespace reference, specifying the new custom tag, primetag, with two attributes, start and end.

In this tag, unlike the two previous tags, not only is an end tag specified separately, </tijtags..., but there is also some body content between the start and end tags. That is what makes this tag so interesting. The <li> tags are just HTML as explained above, but in between them, enclosed in JSP scriptlet brackets, is a reference to the variable value.

There are two very powerful properties of this custom tag. Firstly it has made the variable value available to us for use in a scriptlet, or simple JSP expression, like <%= value%>. Secondly, everything within the body of our custom tag is repeatedly processed and added to the output stream to be presented back to the browser. This means that not only will the tag above keep looping until it has found every prime number within the specified range, but it will display the <li> and </li> tags as well as the value of the variable for each number! In other words, the JSP code above will be all you need to do to display the prime number values in a neat list. And you don't even need to know how many numbers there are going to be.

This has suddenly become very useful. It is a tag that helps reduce the need to embed scriptlet code in our neat tags, but gives us full programming functionality, like looping.

In order to implement such a tag you need to perform the following tasks:

Create (or locate) a class with the appropriate logic for generating prime numbers.

Write a tag class which implements the relevant tag interface.

Write setter methods for the start and end properties.

Program a looping mechanism to process the body content and for each iteration, make a variable available to the JSP page with the current prime number value.

Create a TLD or tag library descriptor file for the tag.

You can find the PrimeUtilities class on the CD (??)or in appendix (??). This example makes use of the sievePrimes() method, which finds prime numbers. There are many resources on the Internet for finding such algorithms if you are interested in this type of thing.

Let's look at the actual tag class.

```
package cx2.tags;
import java.util.ArrayList;
import java.util.Iterator;
import javax.servlet.jsp.tagext.BodyTagSupport;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;
import com.bruceeckel.PrimeUtilities;

public class PrimeNumTag extends BodyTagSupport {
  private int start = 0;
  private int end = 0;
  ArrayList primeList;
  Iterator primeIterator;
  public void setStart(int start) {
     this.start = start;
  }
  public void setEnd(int end) {
     this.end = end;
  }
  public int doStartTag() throws JspException {

     try {
        primeList = PrimeUtilities.sievePrimes(start, end);
        primeIterator = primeList.iterator();
     } catch (Exception e) {
        e.printStackTrace();
        return SKIP_BODY;
     }
     return EVAL_BODY_BUFFERED;
  }
  public void doInitBody() throws JspException {
     try {
        if (primeIterator.hasNext()) {
           pageContext.setAttribute("value",
           primeIterator.next());
        }
     } catch (Exception e) {
        e.printStackTrace();
     }
  }
  public int doAfterBody() {
     try {
        if (primeIterator.hasNext()) {
           pageContext.setAttribute("value",
                          primeIterator.next());
           return EVAL_BODY_BUFFERED;
        } else {
           bodyContent.writeOut(bodyContent.getEnclosingWriter());
           return SKIP_BODY;
        }
     } catch (Exception e) {
        e.printStackTrace();
        return SKIP_BODY;
     }
  }
   public void release() {
     primeList = null;
     start = 0;
  }
```

The first thing you will notice is that this tag class doesn’t implement the Tag interface directly any more. This is because the tag has a little more functionality now and it is more useful to extend the BodyTagSupport class. This class implements the Tag interface and provides some extra functionality. The various Tag classes and interfaces will be discussed in greater detail later.

This class again makes use of the the PageContext object, supplied by the container using the setPageContext() method. Since this tag class inherits from BodyTagSupport, it is no longer necessary to store the pageContext object yourself. That will be taken care of by the parent class. This is a good example of reuse using inheritance.

The doStartTag() method has now become the first method you need worry about. In the implementation above, the PrimeUtilities class is used to obtain and store an Iterator object which contains the prime numbers. Note that unless an exception occurs, this method will return the value EVAL_BODY_BUFFERED, unlike the two previous examples which returned SKIP_BODY in all cases. As you might have guessed, this is the key to implementing a tag which processes its body content.

Next you will notice that the method doInitBody(), has been implemented. This method is called by the JSP container before the body of your tag has been processed, and is the method in which you will perform any initialization code required before processing the body content. In the previous example, this wasn't necessary, as there wasn't really anything to do. In the example above, however, this method queries the prime number iterator to make sure there are prime numbers available in the list, and if there are, the method setAttribute()is used to add the first available prime number value to the page context. As you can see, the value is associated with the name value.

You came across a similar form of the setAttribute() earlier, but this time you will notice it has only two parameters that is because this class makes use of the default scope which is PAGE_SCOPE. The job of making this object available to the JSP page is actually in part the responsibility of a special class called a TagExtraInfo class which will be discussed a bit later on. The looping construct is now initialized and provided no exceptions have occurred, the tag will read in the body text at this point.

The method doAfterBody()is not dissimilar to the doInitBody() method. It checks to see whether there are any more prime numbers in the iterator, and if there are, it adds the available prime number value to the page context. If there are no more prime numbers available, it writes the body content of the tag to the output stream that echoes it back to the browser.

If you look closely you will see that this is not quite straightforward. The object referenced by bodyContent is in fact an instance of the BodyContent object, found in the javax.servlet.jsp.tagext package. This is an encapsulation of the body content of this particular tag, and is a bit like an output stream, but with some extra functionality. If the PrimeNumTag object had not extended the BodyTagSupport class, then a setBodyContent() method would have had to be implemented and the bodyContent object stored "manually" in the same way that we did with the pageContext object in the previous examples.

Because the body content object only includes the body of your particular tag, you will need to make a call to the getEnclosingWriter() method in order to get to the right output stream.

What actually causes the loop to iterate is the return value. If the doAfterBody() method returns the constant EVAL_BODY_BUFFERED, then the container will process the body of the tag and call that method again. If the method returns SKIP_BODY, then the loop will terminate, and provided you have sent your body content to the enclosing writer stream, your values will be displayed on the browser screen.

All that is necessary now, is to create the tag entry in the TLD file.

```
<tag>
  <name>primetag</name>
  <tagclass>cx2.tags.PrimeNumTag</tagclass>
      <bodycontent>JSP</bodycontent>
  <info> Generates Prime Number sequences </info>
  <attribute>
    <name>start</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>end</name>
    <required>true</required>
  </attribute>
  <variable>
    <name-given>value</name-given>
    <variable-class>Integer</variable-class>
    <declare>true</declare>
    <scope>AT_BEGIN</scope>
  </variable>
</tag>
```

If you look at the attribute tag, you will see that the attribute start is not required, but the attribute end is. This was just a case of design choice for the class. It makes sense to default to 0 if no start point is specified, but since there are (apparently) an infinite number of prime numbers, it is a good idea to enforce an end point. You will remember that if the JSP page author does not conform to this constraint when writing the JSP page, and supply at least an end value, the container will throw an exception back to the web browser.

After the attribute tag, there is a new tag that you are unfamiliar with, which is the <variable> tag, with its sub-tags <name-given>, <variable-class> and <declare>. These, as you might expect, tell the JSP container about the variable which is going to be added to the page context by your tag. Since this variable is an object which is not a JavaBean, there is no automatic mechanism for telling the container what type it is, so you need to do it with this tag. The meaning of these tags is listed at the end of this chapter, but let's look at some of them now.

The <name-given> tag tells the container what name to give the variable that it uses. You could use another tag, <name-from-attribute> in its place, which would allow you to specify the name of an attribute whose translation time value will contain the name of your variable. This way you could make the name dependent on some other translation time value.

The <variable-class> tag tells the container what type to make the variable, and the class name specified must be in the classpath at translation time, which is when your JSP file is compiled for the first time on the server. Mostly you will use the standard types found in java.lang anyway.

The <declare> tag tells the container whether or not this variable is declared. This appears to be intended for future JSP development. For now you will always want it to be declared which is the default value.

The <scope> value tells the container at what point your scripting variable should be available to the JSP author. Possible values are NESTED, which makes the variable available only between the start and end tag in the JSP page, AT_BEGIN which makes it available between the start tag and the end of the page, or AT_END which makes it available after the end tag until the end of the page.

At time of writing, there seems to be a problem with the current implementation of Tomcat (4.0.4) reading the <attribute> tag. For this reason you might have to use the more advanced and powerful technique of the TagExtraInfo class. This will entail replacing the entire <variable> tag, with a tag called <teiclass> or tag extra info class tag. The details of this are explained later in this chapter.

Tags within Tags

Tags can also be nested within other tags which gives us a powerful structuring facility and also the ability to create conditional output, by having child tags which are only evaluated if certain conditions apply within their parent tag. The JSTL, Java Standard Tag Libraries, make full use of this, but let's have a look at the highlights of how we might create such a tag ourselves.

Now that you are getting more comfortable with the tag extension API in JSP we only need look the highlights of the next tag. Imagine that you are using the prime number generation tag and you want to check each prime number in turn to see if it is a "Fermat" prime number, and display that to the browser. A Fermat number is a number which can be expressed as "2 to the power 2 to the power n plus 1", where n is an integer. If your eyes glaze over at mathematical descriptions, don't worry. Suffice it to say that some prime numbers are Fermat prime numbers, and some aren't. In fact, the numbers 3, 5, 17, 257, 65537 are the only Fermat prime numbers known.

Since the Fermat prime checker tag is going to have the job of checking each number generated by the prime number tag, it obviously doesn't make any sense for a fermatprimetag to exist outside of a primenumtag. That means it needs to be defined as a child tag which will prevent users of the tag library containing this tag incorrectly and means that you can make certain assumptions. These will be illustrated in the code. Here is the tag class:

```
public class FermatPrimeTag extends BodyTagSupport {
  public int doStartTag() throws JspException {
    PrimeNumTag parentTag =(PrimeNumTag)findAncestorWithClass(this,
                    PrimeNumTag.class);
    if (parentTag == null) {
      throw new JspException("Tag should be nested in
      \"primenumtag\" Tag");
    }
    return EVAL_BODY_TAG;
  }

  public int doAfterBody() throws JspException {

    Integer prime =(Integer)pageContext.getAttribute("value");
    String s = bodyContent.getString();
    try
    {
      if (PrimeUtilities.isFermatPrime(prime.intValue())) {
        bodyContent.clearBody();
        bodyContent.println(s + " (is a Fermat Prime Number) ");
      }
      bodyContent.writeOut(bodyContent.getEnclosingWriter());
    } catch(IOException ioe) {
      throw new JspException(ioe.getMessage());
    }
    return SKIP_BODY;
  }
}
```

Notice the findAncestorWithClass() method. This traverses the hierarchy of tags, finding the parents recursively, until one is found which matches the class you are looking for. The way this method works is by calling the getParent() method recursively. This is another bit of tag management that we inherit from the BodyTagSupport class. If you don't inherit from this class, you will need to implement the setParent() and getParent() methods yourself. The former is called by the container and will pass you a reference to the parent tag. You would have to store this in a handle for retrieval by the getParent() method.

There might be cases where it is necessary to do all of this by yourself, but they will probably be pretty few and far between. By now you should be convinced of the wisdom of creating your own tags via inheritance from BodyTagSupport.

If the findAncestorWithClass() method does not find a parent tag of the correct type, it will throw an exception that will appear on the client's browser window. This will tell the user of the JSP library that they have used the tag incorrectly.

The other important thing to notice with this tag is that it is calling the getAttribute() method on the pageContext object to get the value of an attribute called value. This is the attribute that is added to the page context by the parent tag, primenumtag. We have access to this because our child tag, fermatprimetag and our parent tag primenumtag are on the same page, and therefore share the same pageContext object. Remember though, that you still need to clear the body content and get the enclosing writer. What you are actually doing here is replacing the existing prime number with a string displaying the prime number and a message saying ".. is a Fermat Prime Number."

This tag is much simpler than its parent because it doesn't need to define any of the looping mechanism. This tag will be treated as a brand new tag each time the parent tag iterates, so the doAfterBody() and other methods will be called repetitively.

The Tag Classes

Knowing whether it is better to use an interface or a class to implement your tag handler can be tricky. It might help to look at a diagram of the classes and interfaces you have at your disposal.

The following table describes some of the useful interfaces in the tag handler API:

Interface	Description	When to use
Tag	The basic Tag Handler from which all other classes and interfaces must inherit	Use this if you want specific behavior for every one of its methods, or your tag handler needs to extend another class. Otherwise, use it indirectly via one of the classes that implements it, like TagSupport..
Iteration Tag	Defines the looping mechanism for a tag by adding the doAfterBody() method to the Tag interface.	Use this if you would otherwise have used the Tag interface but require your tag to be iterative. Otherwise use it via one of the classes that implements it like BodyTagSupport.

BodyTag	Defines the mechanism for handling the body content of a tag, by adding the setBodyContent() and doInitBody() methods to the Tag interface.	Use this if you would otherwise have used the Tag interface but require your tag to handle its body content in some way. Otherwise use it via the BodyTagSupport class.
TryCatchFinally	Allows the implementor to create exception handling code that will catch any exception thrown in the handler.	Use this is you require specific behaviour (especially using finally) or if it is imperative that no exceptions be displayed to the client web browser.

This table describes some of the useful classes in the tag handler API:

Class	Description	When to use
TagSupport	Simple class which implements the Tag and IterationTag interfaces and provides useful default behavior for creating basic or iterative tags.	Use this if you want to implement a minimal tag or an iterative tag without doing too much work and you don't need your handler class to implement any other interfaces.
BodyTagSupport	Simple class which extends the TagSupport class and defines behavior for handling the body content by implementing the BodyTag interface.	Use this if you want to implement a tag or an iterative tag that manipulates its body content, and you don't need your tag handler to implement any other interfaces.

Once you have chosen the appropriate class with which to implement your tag handler, you will need to work with the companion classes in the API.

This table describes the two companion classes most often needed by tag handlers.

Class	Description	When to use
BodyContent	Class which encapsulates the	You will need to use this class

	body content of a tag and allows you to manipulate it.	whenever you are manipulating the body content of your tag.
PageContext	Class which allows storage and access of attributes in the JSP page. The instance of this class you have access to in your tag handler is the same one that can be accessed via the JSP page.	Use this class to make attributes and beans available to the JSP page.

Some Advanced Tag Features

Using TagExtraInfo classes

Sometimes, the variables that you want to make available from your tag for the JSP page author are a little bit too complicated to define in a static TLD file. The names of the variables might be dependent on some set of conditions, or you might want to validate the JSP page. For instance, you might want to make sure that the JSP author hasn't specified incorrect attributes or nested a child tag in the wrong parent tag. For this kind of functionality, the <variable> tag in the tag library descriptor file is no longer adequate.

In the primenumbertag above, instead of having a <variable> tag, we could have had the following:

```
<teiclass>cx2.tags.PrimeNumTagExtraInfo</teiclass>
```

This tag tells the container that you have an entirely separate class to describe your attributes for you.

Actually this isn't as bad as it sounds, the class you need is very simple and you generally don't need to override more than one method. In the case of the PrimeNumTag class, you would have named the class called PrimeNumTagExtraInfo, to follow the convention, and implement it like this:

```
package cx2.tags;
import javax.servlet.jsp.tagext.TagExtraInfo;
import javax.servlet.jsp.tagext.TagData;
import javax.servlet.jsp.tagext.VariableInfo;

public class PrimeNumTagExtraInfo extends TagExtraInfo {
   public VariableInfo[] getVariableInfo(TagData data) {
      VariableInfo[] vi = new VariableInfo[1];
      vi[0] = new VariableInfo("value",
                          "Integer",
                          true,
                          VariableInfo.AT_BEGIN);
      return vi;
```

```
  }
}
```

Notice that the class extends the class TagExtraInfo, found in the javax.servlet.jsp.tagext package. The getVariableInfo() method creates an array of VariableInfo objects describing the name, type, whether or not it should be declared, and its scope. These correspond with the similarly named elements of the <variable> tag. The array of VariableInfo objects is only of size 1 because there is only one variable.

The TagData object is used as a parameter to the getVariableInfo() method, but this has been designed with future development in mind. Some of the functionality it is intended to support is not yet available in JSP.

Tag Library Descriptors Revisited

If you have another look at the tag library descriptor we used for the tags above, you will find that you are familiar with the first two lines after reading the chapter on XML in this book, The <taglib> tag is self explanatory, defining the start and end of the tag library. Typically you would have one of these per TLD descriptor. It can have several sub elements:

tlibversion - defines a version number which can help the programmer apply some validation at translation time by making it possible to read the version no.

jspversion - defines a version number which helps the container determine which version of JSP is being used, so that if necessary it can turn off more advanced features or perform any special operations that need to be performed for backward compatibility.

shortname - defines an easy-to-use name which a GUI based builder tool might use as the preferred prefix to any generated tags or directives..

uri - defines a unique identifier for the tag library. Typically it is useful to define it in the form of an HTTP URL like the one in the example in this chapter, but it doesn’t have to resolve to a web page, as long as it is unique.

displayname – defines a human readable name that would be useful to display on the toolbar of of a GUI based authoring tool.

small-icon - defines an optional image file, suitable for a builder tool to display on a tool bar.

large-icon - defines an optional image file, suitable for a builder tool to display on a menu panel.

description - defines a description of the tag library which a GUI based build tool might use to describe this library on a tool bar or menu.

validator - defines an optional class that can be used to validate the tag by checking through its parameters and even other tags on the same JSP page, to make sure that the JSP author has used it correctly.

listener - defines an optional class that implements a listener class for the servlet created from the JSP page. The class must implement one of the servlet listener interfaces, e.g. javax.servlet.http.HttpSessionListener.

A <taglib> tag must have at least one <tag> subelement defined. This makes sense, you wouldn't want to have a tag library with no tags. It wouldn't be very useful.

The tag element itself has several subelements, some of which you have already come across earlier in this chapter:

name - defines a unique name for this tag, or action.

tagclass - defines the fully qualified name of the class which implements the tag handler interface javax.servlet.jsp.tagext.Tag.

teiclass – defines an optional extra info class, which contains information for validation of the variables for this tag. Must implement the interface javax.servlet.jsp.tagext.TagExtraInfo.

bodycontent – The type of content to be found in the body of the tag, valid values are empty, JSP and tagdependant. If tagdependant is specified, then content will be passed uninterpreted to the tag handler. This would be useful for embedding other languages, e.g. SQL queries.

displayname – A short, human-readable name that can be displayed by a GUI based authoring tool.

small-icon - defines an image file, suitable for a builder tool to display on a tool bar.

large-icon - defines an image file, suitable for a builder tool to display on a menu panel.

description - defines more, optional tag specific information about the tag and how it may be used by other tags. It does this by providing a class which can apply constraints. The class must be a sub-class of the one that implements the tag.

example - defines an optional example of how to use the tag, suitable for a builder tool to display on a help menu.

The last two subelements of the <tag> tag, have subelements themselves.

variable – defines information for a scripting variable defined on this tag. Can be substituted by using the teiclass tag, and providing a tag extra info class.

The subelements of variable are:

name-given – defines the variable name as a constant

name-from-attribute – defines the name of an attribute which will give the name of the variable at translation time.

variable-class – defines the type of the variable. The default is java.lang.String.

declare – whether the variable is declared or not. True is the default.

scope – the scope of the variable. NESTED, AT_BEGIN, or AT_END.

attribute - defines information for an attribute used by this custom tag.

The subelements of attribute are:

name – defines the attribute's name.

required – defines whether or not the attribute is optional.

rtexprevalue – defines whether or not the attribute may be passed as an expression which can be calculated at run time.

type – defines the type of the attributes. String is default and the most useful.

description – an optional description of the attribute.

Deploying Tag Libraries

Deploying tag libraries takes a fair bit of administration but like any administrative task, the more you understand requirements of each application, the easier it becomes. In the case of tags, there is more than one way to deploy a tag library and you will ultimately have to make the decision as to which is the most appropriate for your application. This would apply whether you were deploying your own tag libraries or ones you were using from a third party vendor.

You have already seen that a tag library descriptor file needs to be created for each tag library, and obviously you want to put that somewhere where the server can find it. You also need to put the Java classes somewhere that the container can find them.

One easy way to do this is as follows:

Put your type library descriptor file, or .tld file on the server in the WEB-INF directory, under your web application directory. e.g. ../webroot/tijtags/WEB-INF, using the path character most suitable for your operating system.

Specify the location of the .tld file in your JSP page using the taglib directive, e.g. <%@ taglib uri="/WEB-INF/tijtags.tld" prefix="tijtags" %>. The uri attribute here indicates the relative path of the tag library descriptor file.

Put your JSP files where you normally would, say in the jsp directory.

Place your classes in the classes directory of your WEB-INF directory, e.g. ../webroot/tijtags/WEB-INF/classes/ making sure that you use subdirectories for the package names, as you would for any other Java application.

That is all you need to do. When you restart your server, and access the JSP page, e.g., by typing http://localhost/tijtags/index.jsp, assuming you are running your server locally, the page will be able to access the tag library descriptor and classes.

This is all right for testing, but it is much more typical to keep your classes in a jar file and put the jar file in the /WEB-INF/lib directory. If your tag library descriptor file is in the jar file with the handler classes, then the container will be able to locate it, and will resolve the tags at translation time and runtime.

If you are deploying your tag libraries in a release scenario, or if you are using a third party tag library, you will want to have a more consistent directory structure and you might want to keep all of your tag libraries in one place. This would allow you to share tag libraries across different web applications.

To do this, you can use a second approach, which is to make use of the <taglib> element of the web.xml file associated with your web application by adding the following lines:

```
<taglib>
<taglib-uri>http://www.eckelobjects.com/tagtut/tags1.0</taglib-uri>
<taglib-location>tijtags.tld</taglib-location>
</taglib>
```

This creates a mapping between the URI and the location of the tag library descriptor file. Once this mapping exists, your JSP page can reference the tag library by its URI, rather than via its Tag Library Descriptor file:

```
<%@ taglib uri="http://www.eckelobjects.com/tagtut/tags1.0" prefix="tijtags" %>
```

This allows a web administrator to move tag libraries around on the server, should that be necessary, without the web applications being affected. As long as the server has created that mapping, any web application will be able to make a reference to the tag library via the URI. This makes more sense when deploying third party tag libraries which need to be in a shared directory where they can be accessed by all web applications.

Remember that the URI doesn’t actually need to be in the form of an HTTP URL as above, but this is quite a good convention which guarantees its uniqueness.

You will notice in the example above that the mapping specified in the web.xml file is between a .tld file and a URI. If your tag library classes are in a jar file, then this means having a tag library descriptor which is outside the jar file. This could be a nuisance, so the tag container specification defines an automatic mapping directly to a jar file which contains a tag library descriptor.

To make use of this, you need to ensure that your tag library descriptor is in a file called taglib.tld and that it appears in the tag library jar file under the directory META-INF. If that is the case, then you can define the mapping in your web.xml as follows:

```
<taglib>
<taglib-uri>
http://www.eckelobjects.com/tagtut/tags1.0
</taglib-uri>
<taglib-location>tijtags.jar</taglib-location>
</taglib>
```

Notice that the jar file is specified as the location of the tag library. This is probably the best way to deploy tag libraries, it allows you to share libraries between applications, refer to the tag library by a unique URI, and keep the tag library descriptor file neatly packaged with the classes.

Actually the specification of how a JSP container should handle mappings to tag libraries is pretty flexible. The container will try and build an implicit mapping if it can find your tag library descriptor and there is no explicit mapping in your web.xml file. It is not recommended that you leave too much to the container, though. The more you do that, the more you run the risk of losing portability, when, say, moving your application to another application server, you are also more subject to experiencing different behavior on different platforms.

Using Third Party Tag Libraries

Now that you understand how to create and deploy tag libraries, you might be happy to discover that you don't have to create your own. As with any other type of library development, there will always be times when you are forced to write your own code, but there are large and very powerful libraries already available for you to use, and you should always make sure that you are not reinventing the wheel when you embark on new development.

The JCP, (Java Community Process) have, at time of writing, just published the first public draft of the JSP Standard Tag Library, JSTL, which defines a large number of powerful, multi-purpose tags. These provide standardized solutions for some of the most common problems one faces as a developer and will be discussed shortly.

The Apache Jakarta group have another large collection of very useful tags freely available to developers these are grouped under the project name "taglibs" and serve the same purpose as the JSTL. Although there is some common functionality across the two libraries, this group have a close working relationship with the JCP and consistently been producing excellent open source Java products for some years now. The Jakarta projects are always worthwhile keeping an eye on.

Detailed explanations of these libraries are beyond the scope of this chapter but lets have a look at a selection of some of the popular tag libraries that are available.

Here are a few of the tag libaries available via the Jakarta “Taglibs” project, which should give you an idea of the kind of thing you can expect from a tag library.

TagLibrary Name	Function	Example Tags
Application	Allows user access to information about the JSP application	existsAttribute – checks if an attribute exists
DateTime	Allows a user to handle date and time using locales and time zones	currentTime – the current time in milliseconds.
DBTags	Allows a user to make SQL queries	query – a sql query, embedded within a statement tag.
I18N	Provides internationalization functionality	bundle – define a resource bundle
Input	Provides utilities for accepting input from users	textarea – displays a multiline textarea
IO	Provides a variety of input and output operations	request – allows you to insert the result of an HTTP request
JNDI	Provides tags for connecting to Java Naming and Directory interface	useDirContext -creates a DirContextObject
Log	Provides tags for easy logging using the log4j libraries	debug – displays a debug level message
Mailer	Provides tags for using the JavaMail libraries from JSP	setrecipient – allows specification of a recipient of the mail message
Page	Provides tags for access to the page context for the JSP page	attributes – loops through all the page attributes.
Regexp	Provides tags for Regular Expression matching	regexp – creates a regular expression script variable.
Session	Provides tags for reading or modifying client HttpSession information	isNew – boolean tag, displays if session is new
String	Provides tags for manipualting strings	upperCase – converts a string into upper-case.

Xtags	Provides tags for working with XML.	style - Performs an XSL transformation on the given XML document.

Enterprise JavaBeans

Enterprise JavaBeans (EJB) are managed, server-side components for the modular construction of enterprise applications[7].

We'll start our tour of Enterprise JavaBeans by exploring the meaning of these three words.

"Component" means that EJBs are distributed in binary format and are configurable, so that the client programmer can use them to create custom applications. It's the same concept that you see with a JavaBean graphical component: you buy a JavaBean component to create pie chart graphs, you instantiate it in your application (probably using a RAD tool such as JBuilder), you set the component's properties to your liking (for example the background color), and you use it in your reporting application. You don't have the source code for the pie chart component, nonetheless you can customize it to your needs. The way you instantiate, customize and use Enterprise JavaBeans is not the same as with plain JavaBeans, but the concept of being a component is the same. It's important to make a terminological distinction between the component and its instances: we use the term Enterprise JavaBean to refer to a type of component – for example an EJB that represents a bank account; and we use the term EJB instance to refer to an object that represents a specific bank account with a unique account number.

"Server-side" means that EJB objects are allocated in a process on some server, not on the client machine. EJBs can offer a remote view, a local view, or both. The term "view" doesn't refer to a graphical view; here we are talking about the way an EJB exposes its interface. If an EJB offers a remote view, all method invocations between the client code and the remote instance happen via a remote procedure call protocol, most likely RMI-IIOP. If an EJB offers a local view, the client code must be in the same process as the EJB object, and all calls are direct method calls.

As you have seen in Chapter 16, the ability to perform remote procedure calls implies the presence of two fundamental architectural pieces: a naming service and RPC proxies. A naming service is a network service that client applications use to locate heterogeneous resources on the network. An RPC proxy is a programmatically-generated Java class, instantiated on the client side, that exposes the same interfaces as those of a specific remote component. Since it exposes the same interfaces, the client can't tell one from the other, and the proxy can pretend to the be the remote instance. But what the proxy does, in fact, is delegate all calls to the remote component, implementing the networking code and hiding all the complexity from the client. In EJB, we have the same concepts of naming service and RPC proxies, but they are slightly different from what you've seen with Java Remote Method Invocation (RMI).

Finally, and most important, EJBs are managed. This means that when an EJB object is active in memory, its hosting process is doing with it much more than what the plain JVM does. The process hosting an EJB object is called an EJB Container, and it's a standardized runtime environment that provides management functionalities. Since the EJB specification standardizes the services provided by an EJB Container, we have vendors implementing these functionalities in commercial products, like WebSphere and WebLogic, two well-known commercial EJB Containers by IBM and BEA Systems, respectively. In an EJB Container, services like remote objects and client-protocol neutrality are implicit in the use of Java RMI; other services (listed below) are standard. Services like clustering and fail-over support are additional values offered by certain vendors.

These are the services offered by all EJB Containers:

- Object Persistence
- Declarative Security Control
- Declarative Transaction Control
- Concurrency Management
- Scalability Management

Object Persistence means that you can map the state of an EJB instance to data in some persistent storage – typically, but not only, a relational database. The Container keeps the state in the persistent storage synchronized with the state of the EJB instance, even when the same object is accessed and modified concurrently by multiple clients.

Declarative Security Control gives you the option of letting the Container check that the client invoking a specific method on a specific EJB has been authenticated and belongs to a security role that you expect; if it doesn't, an exception is generated and the method is not executed. The benefit of declarative security control is that you don't have to code any security logic in your EJB – you simply define security roles and "tell" the Container which roles are allowed to invoke which methods. Since there is no security logic hard-coded in your EJB, it's easy to specify changes to the security requirements with no recompilation.

Declarative Transaction Control is a similar mechanism: you tell the Container to do something on your behalf, but the information you provide is what should happen in terms of transaction demarcation when a specific method is called. For example, you could instruct the Container to start a new transaction when the method is called, or you could instruct it to use an existing transaction and refuse the method call if one isn't already active (transactions propagate in a chain of method calls). The Container will automatically commit a transaction when the method that started the transaction terminates gracefully, or it will roll the transaction back if it catches any exception thrown while the transaction was active. Again, the benefit of declarative transaction management is that there is no transactional logic in your EJB source code, so not only does that makes life simpler for the developer, but it also makes it easy to modify the transactional behavior of your EJB without recompilation.

If you remember that EJBs are components, you'll also see why these two features are so important. Being able to declaratively control the security and transactional logic is a fundamental requirement in a component model, because it allows people who don't have the source code, or don't want to work at that level, to adapt the component to the application that is being built with it.

Concurrency Management synchronizes concurrent method invocations coming from different remote clients and directed to the same EJB object. In practice, it guarantees that the component can be safely used in the inherently multithreaded environment of an application server (quite a valuable feature).

Scalability Management addresses the problem of the increased resource allocation that occurs when the number of simultaneous clients increases. The resources allocated by a client are not only EJB objects, but are also all supporting resources like database connections, threads, sockets, message queues and so on. For example, if the number of simultaneous clients increases from 100 to 1,000, you may end up 1,000 EJB objects in memory, and perhaps 1,000 open database connections; this may be too much for the amount of memory that you have, and it's certainly a heavy load on your database engine. An EJB Container may decide to address these problems by offering EJB instance pooling and database connection pooling. That is, the Container would keep only a limited amount of instances or connections alive in memory, and would assign them to different clients, only for the time the client actually needs one.

Enterprise JavaBean Flavors

We know that EJBs are components. However, the term "component" is one of the most overloaded in the industry, and different people (and different vendors) have different ideas of what a component is. On the Microsoft platform, including .NET, a component is something that complies with the (D)COM(+) binary specification, and the nature and type of a component is determined by the standard interfaces that it exposes. However, there is no architectural model enforced by the Microsoft platform, or at least it is intentionally very loose. In other words, there are very few enforced architectural guidelines that tell you how to develop and assemble components in order to create an enterprise application.

The J2EE platform, on the other hand, provides you with a more precise architectural framework in which you create and use components. This is especially true with EJBs, where the specification defines three different EJB types, with different responsibilities and different areas of applicability. The three EJB types are:

1. Entity Beans
2. Session Beans
3. Message-Driven Beans

Entity beans are persistent objects, in the sense that the Container transparently keeps their state synchronized with data in some persistent storage, typically a relational database. For this reason, entity beans are used to represent business entities, like a customer, a shopping cart and so on. Entity beans don't implement business logic, except for self-contained logic directly associated with the internal data. Although all entity beans are exposed to the client code in the same way, they can be implemented using two different persistence strategies. The developer has two options: 1) let the Container take care of moving the state between an EJB object and the database, or; 2) take over this mechanism and implement the code that moves the EJB's state from and to the persistent storage. In the first case, we say that we are employing Container Managed Persistence (CMP), so the entity bean is a CMP bean; in the second case we say that we are using Bean Managed Persistence (BMP), and we have a BMP bean. Again, although there are substantial practical differences between CMP and BMP, they are implementation strategies that do not affect the way an entity bean is used by clients.

Discussing all the implications of choosing CMP over BMP is beyond the scope of this chapter. However, it's important to explain that the most valuable benefit of CMP is not, as you may think, that you don't have to code the persistence logic yourself. The real benefit of CMP is component portability across different persistent storages. If you think about it, CMP works because the Container generates all the code necessary to move state between an EJB object and the persistent storage, and vice versa. This means that the Container knows how to communicate with that specific storage. An EJB Container implementation supports multiple storages, typically all major RDBMS like Oracle, DB2, MySQL, etc. Since the persistence logic is not hard-coded in a CMP entity bean, but is provided by the Container, you can use the same binary component in different operating environments and still get object persistence support from the Container. If you use BMP, the persistence logic (for example the SQL statements, if you are coding for a specific RDBMS) is built into your component and that makes it harder to use it with a different storage.

So, in general, you use CMP to promote portability, ; and you use BMP if your entity bean has to persist on some system not supported by the J2EE platform (for instance, a CICS application on an IBM mainframe) or under other special circumstances that we can't cover here.

Session beans are a different breed. They are not persistent, but are instead discrete components that implement business logic, for example the steps required to checkout and purchase some product from a virtual shopping cart. Session beans may cooperate with other session beans to access additional business logic, and they use entity beans when they need to access persistent information. Session beans come in two different variations: Stateless session beans (SLSBs) and Stateful session beans (SFSBs).

The difference between the two is that Stateful session beans keep a conversational state during the communication with the client, whereas Stateless session beans don't. The conversational state is the information that is passed from the client to the session bean in a sequence of method calls.

For example, you could have a session bean implementing the logic to reserve and purchase movie tickets over the network. Clients would connect to and use one of this Bean's instances; the instance would expose two methods, reserveSeats(...) and purchase(...). The first method receives the number of seats that the customer wants to purchase, and reserves those seats in the system; the second method receives the customer's credit card information, validates the credit and completes the purchasing process. If the Bean is a Stateful session bean, once reserveSeats(...) has been called, the session bean instance "remembers" the requested number of seats, and you don't need to pass that information again to purchase(...). If the session bean is stateless, on the other hand, the Bean has no memory of the information passed from previous method calls, so the client state needs to be passed with every call. There are several strategies to implement and optimize the state transfer, but state would still need to be passed with every method call.

The reason for these two types of session beans is quite simple: certain business processes are inherently stateless (especially those that are closely mapped to an HTTP front-end) and others are inherently stateful; the architecture is designed to reflect this.

There are technical implications in the use of one session bean type versus the other, but as in the case of CMP versus BMP, a detailed discussion is beyond the scope of this chapter. One thing that you want to remember though, is that Stateless session beans tend to work better with the Container's instance pooling mechanism than Stateful session beans. Since a Stateless session bean instance is not supposed to remember any state information passed by the client once a method completes, the same instance can easily be reassigned to a different client for another method invocation. If the session bean instance is Stateful, the Container cannot assign it to a different client, unless it first moves the instance state to some temporary storage for later retrieval (a process called, in EJB, activation and passivation). It's a common misconception that due to instance pooling, Stateless session beans make for better scalability, but this is not always the case. Your Container could have a very highly optimized activation and passivation mechanism, so that moving a Stateful session bean's state to and from some secondary storage has very little effect on performance. Second, if you use a Stateless session bean, you must pass the client's state into the session bean with every call, and if the client and the session bean are on two different distribution tiers, the act of constantly transferring state, and serializing and deserializing the method parameters as required by RMI, could become critical.

An interesting side note about instance pools is that your Container may not employ them at all. With the recent optimizations introduced in the Java compilers and JVMs, modern Containers have discovered that allocating and garbage-collecting raw memory is more efficient than managing a pool.

Finally, the third type of Enterprise JavaBean: Message-Driven Beans (MDB). MDBs work in cooperation with the Java Messaging System (JMS), which is an abstraction API on top of Message-Oriented Middleware (MOM) systems, more or less in the same way that JDBC is an abstraction API on top of SQL databases.

Once again, we don't have the space in this chapter for a full discussion on MOM systems. Briefly, MOM systems provide a publish-subscribe messaging model based on asynchronous, distributed message queues. A MOM message is a packet of information that someone publishes and someone else is interested in receiving. MOM messages are published to, and retrieved from, a queue or channel. The publisher and the subscriber can be in two separate processes, and messages sent into a MOM queue are guaranteed to be delivered, even in the case of a system crash. MOM systems are extremely valuable for implementing any form of distributed, asynchronous processing – something similar to event handling in a standalone application.

MDBs are receivers of MOM messages coming through the JMS API. An MDB is usually implemented to perform some kind of action when a message is received, and acts as an object-oriented connection point between subsystems cooperating using JMS. For example, an MDB could be implemented to send an email to the administrator (using the JavaMail API), when a message that notifies certain events is received.

One other thing about MDBs is that, unlike session beans and entity beans, they do not expose any remote or local view. In other words, client code can not access an MDB, but the MDB can use other EJBs and other services.

EJB Roles

The fact that the EJB is a component architecture implicitly means that there are different moments in the lifetime of an EJB component: development, installation, configuration and use. The EJB specification goes to the extent of defining the organization of labor and the different roles involved in the lifetime of an EJB component. Specifically, the following roles are defined:

- Enterprise Bean Provider. Implements an EJB component and packages it for distribution. Knows about the application domain, but not necessarily about the operating environment in which the component will be used.
- Deployer. In EJB, deployment is the act of installing one or more EJB components in a specific EJB Container. The Deployer is an expert on the specific operating environment, and is responsible for associating the EJB with all the resources it needs to operate (database connections, tables, other EJBs and so on).
- Application Assembler. Uses different components deployed in a specific environment to create a complete application.
- System Administrator. Is responsible for creating and maintaining users, databases and in general all the infrastructure resources needed by a specific operating environment.

It's important to remember that these are roles, not persons. On a small project, the same person may play any or all of these roles. On a large project, where components could be developed by other departments or bought from thirds parties, different persons play different roles.

The other interesting thing is that the specification not only defines the responsibilities and expected skills of each role, but it even defines which source files and tasks are assigned to each role.

In the same way, the EJB specification makes sure that every piece of non-standard configuration information is defined in separate source files. This is because different EJB Container implementations may offer different features, typically configurable by the mean of XML files. If certain features are specific to one Container, they will be specified in Container-specific files.

The Basic APIs

The EJB API is not the only one on the Java2 Enterprise Edition platform. There are many more: the Java Transaction API (JTA), the Java Messaging System API (JMS), Servlets, JavaServer Pages (JSP) and Java Database Connectivity (JDBC) to name a few. Before we can look at how to implement and use an EJB, we need to briefly cover a couple of fundamental APIs: the Java Naming and Directory Interface (JNDI) and the actual EJB API.

JNDI

JNDI is an abstraction API on top of different naming services, like the RMI Naming Service or the COS Naming Service in CORBA, and directory services like the Lightweight Directory Access Protocol (LDAP). A naming service and a directory service are not conceptually the same thing, but once again this is not the right place to discuss the differences. What we'll do is to look at what they have in common.

All naming and directory services let you populate and search some sort of repository. More specifically, they let you associate (or bind, if you wish) a name with an object, and then search for that object (or look it up) using that name. By analogy, they can be used as a white pages directory, where the physical telephone that you want to place a call to (uniquely identified by a number) is associated to a customer name. A more Java-related example, is RMI: you can register an object with a name (a String) in the RMI Naming Service, so that other applications can locate it looking that name up in the service. But this is also the exact same thing that you can do with the COS Naming Service. So that's why JNDI is such a convenient abstraction: it provides you with a uniform interface to access different naming and directory services, just like you can use JDBC to access different databases.

Like JDBC, JNDI is an abstraction API, and what you find in the JNDI package javax.naming is primarily Java interfaces. The actual implementation of those interfaces must be provided by some vendor that wants to provide JNDI support for their service. To actually use that service, you need a JNDI Service Provider – just like you need an appropriate JDBC driver to access a specific database.

JDK 1.4 comes with four standard JNDI Service Providers that provide support for RMI, DNS, COS Naming Service and LDAP. There is an additional JNDI Service Provider, that you can get from Sun's web site, that provides a JNDI view over your local file system (so you can locate files and directories using JNDI).

JNDI supports the concept of a namespace, which is a logical space in which names can be defined. The same two names defined in different namespaces will generate no ambiguity or name clashing, and namespaces can be nested. This is a concept that you see in a number of different situations — your local file system uses nested namespaces (directories), and the Internet DNS uses nested namespaces (domains and sub-domains).

In JNDI a namespace is represented by the Context interface, the most frequently used element in the API. There are various classes implementing the Context interface, depending on the actual service you are accessing via JNDI. The Context interface has methods to bind a name to an object, to remove the binding, to rename it, to create and delete sub-contexts, to look names up, to list names and so on. You should also note that since a context is a Java object, it can be registered in another context with its own name. In other words, the way we express nested sub-contexts in JNDI is by starting with the parent context, and creating a binding in it between a sub-context name and a nested Context object.

Another fundamental concept in the JNDI API is that, regardless of the actual service that you use, to be able to create bindings, perform lookups and so on you need to start from some “root” context. The InitialContext class is what you use to get a Context instance that represents the parent of all the sub-contexts you want to access.

Although certain operations in JNDI are only applicable to certain Service Providers, the concepts expressed so far are general and widely applicable. Let’s see an example. The code snippet below shows how to list all the names registered under the root of the service you are using. You’ll notice that the InitialContext constructor receives an argument of type Property. I’ll explain that bit in a second. For the moment, just take a look at the code below.

```
Context context = new InitialContext(props);
Enumeration names = context.list("");

while(names.hasMoreElements())
  System.out.println(names.nextElement());
```

You see that it’s quite simple. We create a Context object that represents the root of your naming or directory service, get an enumeration of all the elements in that context (the empty string in the list() method call means that you are not searching for a specific name), then iterate and print out all the elements in the enumeration.

The abstractions and the programming model are quite simple. The real question though is: which naming or directory service are we using? The answer is in the property bag that we pass to the InitialContext() constructor. The JNDI API defines quite a few standard property names (declared as constant Strings in the Context interface) and the value of those properties determines the nature of the service that you’ll be using. Of these properties, two are fundamental: INITIAL_CONTEXT_FACTORY and PROVIDER_URL. The first one specifies the class that will produce JNDI Context instances. If you want to use DNS, for example, you’ll need to specify a class that produces Contexts capable of interacting with a DNS server. The second property is the service location, and is hence a URL. The format of this URL depends on the specific service.

Below is the complete example that uses JNDI to browse the contents of the root context in a DNS server (you may need to use a different IP address for your DNS, depending on your network configuration).

```
//: c18:jndi:SimpleJNDI.java
import javax.naming.*;
import java.util.*;

public class SimpleJNDI {
  public static void main(String [] args)
```

```
   throws Exception
  {
   Properties props = new Properties();
   props.put(
     Context.INITIAL_CONTEXT_FACTORY,
     "com.sun.jndi.dns.DnsContextFactory"
   );

   props.put(
     Context.PROVIDER_URL,
     "dns://207.155.183.72"
   );

   Context context = new InitialContext(props);
   Enumeration names = context.list("");

   while(names.hasMoreElements())
     System.out.println(names.nextElement());
  }
}
///:~
```

If you wanted to use a different DNS, or a completely different type of service, in most cases you would just put different information in the property bag, but the rest of the code would be the same.

There is another way of providing the JNDI property values. You can use an external property file, that is a text file that associates property names with their values. This file has to be named jndi.properties, and it must be in the classpath of your JNDI application. The only difference is that you cannot use the property name constant defined in the Context interface, but you must use their "raw" value instead (which you can find in the standard JDK documentation). The content of our jndi.properties file is:

```
java.naming.factory.initial=com.sun.jndi.dns.DnsContextFactory
java.naming.provider.url=dns://207.155.183.72
```

If you take this approach, you will no longer need to pass a property bag to the InitialContext constructor. In other words, what you place in the jndi.properties file are your default JNDI settings.

The last and most important thing we have to cover about JNDI is how to look up names. As I mentioned, there is lookup() method in the Context interface. In the following example we'll assume the presence of an external jndi.properties file that defines which service you are browsing.

```
//: c18:jndi:SimpleJNDI2.java
import javax.naming.*;
```

```
public class SimpleJNDI2 {
  public static void main(String [] args)
    throws Exception
  {
    System.out.println(
      new InitialContext().lookup(args[0])
    );
  }
}
///:~
```

We create a new InitialContext and call the lookup() method on it, passing a String as the name argument. Since the lookup() method is designed to work with different services, it returns a generic Object. However, the actual run-time type returned from lookup() depends on the specific service you are using. For the sake of this example though, we ignore the type and just print the result.

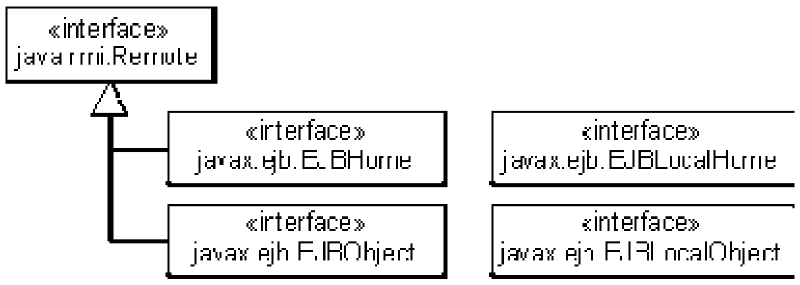
In EJB programming, we use JNDI to locate all kinds of resources, including EJBs, database connection pools, environment information and more. In other words, from within an EJB Container your only window on the world is JNDI. Client applications also use JNDI to obtain connections to EJB factories (more about this later).

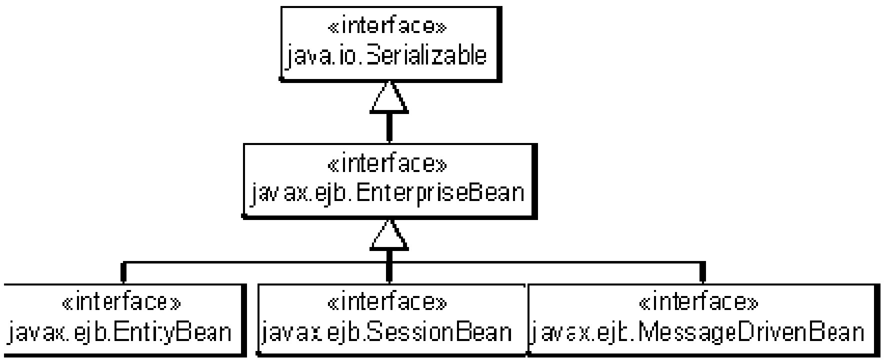
So, if we use JNDI with EJBs, which is the JNDI Service Provider? The answer is that an EJB Container runs and exposes its own JNDI service, specialized to work with resources managed by that Container. In other words, this EJB JNDI service is designed is to let clients and internal Enterprise JavaBeans locate resources by JNDI names. So remember, when you start your EJB Container, you are also implicitly starting an EJB JNDI service, which is available to you through the standard JNDI API.

EJB

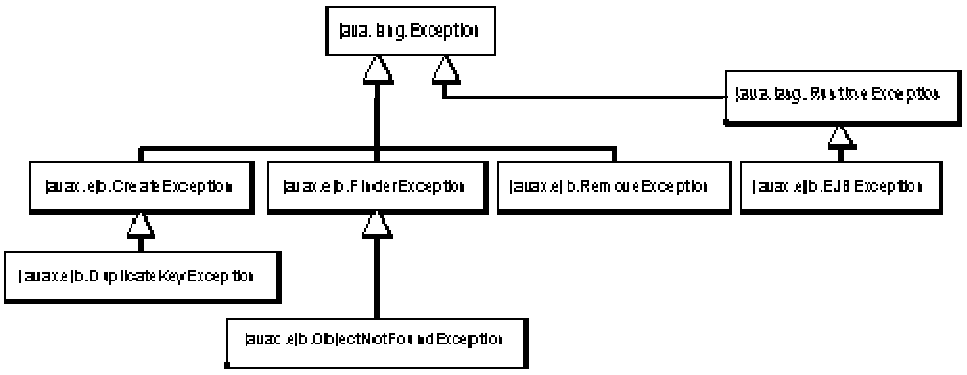
The other API we need to take a look at is the EJB API itself. It's defined in the javax.ejb package which, interestingly enough, is composed just of interfaces and a few exception classes.

The diagram below shows the main interfaces in the package. Three of them represent the different EJB types: EntityBean, SessionBean and MessageDrivenBean. Depending on which type of EJB you are coding for, your class will have to implement one of these interfaces. There are also the EJBHome and EJBObject interfaces which you use as base interfaces when you define your bean's remote view (if you want one) – for this reason EJBHome and EJBObject are derived from the RMI Remote interface. The remaining two interfaces, EJBLocalHome and EJBLocalObject, you use to provide the local view of your bean (again, assuming you want to have a local view). In the next section we'll see how to actually code using these interfaces. For the moment just keep the design in mind.





The other set of elements that we need to cover briefly is the exception classes. You can see below a diagram with the most relevant exceptions (there are a few more in the package, please refer to the JDK documentation for the details).


java.lang.Exception
java.lang.RuntimeException
javax.ejb.CreateException
javax.ejb.FinderException
javax.ejb.RemoveException
javax.ejb.EJBException
javax.ejb.DuplicateKeyException
javax.ejb.ObjectNotFoundException

As you can see, there are two major categories: exceptions derived from java.lang.Exception directly, and others derived from java.lang.RuntimeException. Or, in the Java parlance, checked and unchecked exceptions, respectively.

In EJB all checked exceptions are considered application-level exceptions. That is, they spread multiple tiers of a distributed application, and when generated by an EJB, they cross network distribution tiers and propagate to the remote client.

In fact, the use of EJB checked exceptions in the definition of the EJB interfaces that clients use, is enforced by the specification. For example, CreateException must be in the exception specification of methods that the clients use to instantiate an EJB. Similarly, FinderException has to be in the exception specification of methods that clients use to locate existing, persistent EJB objects. And RemoveException must be specified by methods that remove an EJB object.

Unchecked exceptions, instead, are used in the internal implementation of an EJB. In other words, if something your EJB is attempting fails, you throw an EJBException or one of its subclasses. The reason why EJBException is unchecked is because in most cases it represents some error that propagates from a sub-system to another inside the EJB Container. Even if you were forced to catch it there isn't much you could do about it – an EJB instance exists only inside an EJB Container, which manages the instance and takes care of what happens to it. Here's an example: you have implemented an EJB that accesses a database and at a certain point the database generates an SQLException. It would be pointless to return this exception to the remote client, because the client wouldn't know what to do with it. On the other hand, you want to abort the current operation and notify the Container that something went wrong, so that it can take some corrective action like rolling back the current transaction.

You can't simply let the SQLException propagate into the Container, because the Container has a generic way to receive system-level exceptions – so that it can consistently receive exceptions from different, even future, sub-systems. What you do is wrap the checked SQLException in an unchecked EJBException, using its constructor that takes a java.lang.Exception, and then you throw the EJBException.

All exceptions that propagate out of your EJB object are intercepted by the Container. The action taken at this point by the Container may vary depending on the circumstances, but in general the Container will automatically rollback the current transaction (if there is one) and will propagate the exception to the client. If the exception generated inside of your EJB was an EJBException, the client will receive a generic RMI RemoteException to inform it that the method failed. If the exception generated in the EJB was an application-level exception, like FinderException, that will be the exception that the Container passes on to the client.

EJB Container internals

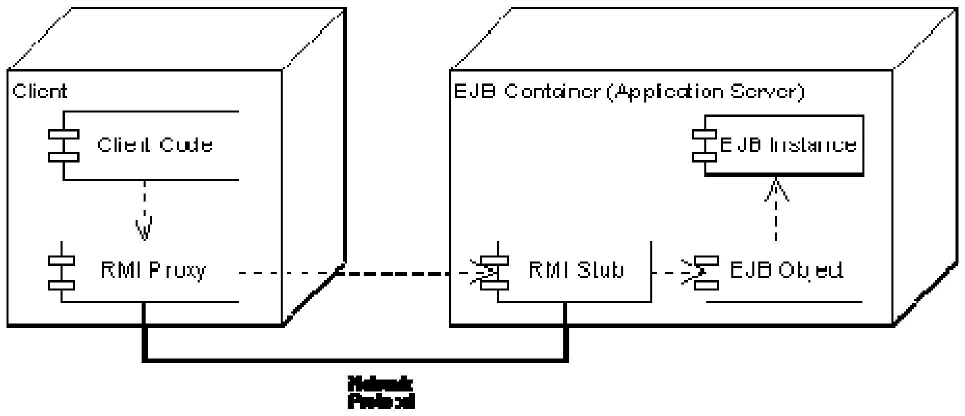
You know that EJBs are managed components, and it's now time to take a closer look at how the Container manages your EJB objects. That will help us to better understand the services that the Container provides, the code examples and the development process.

The basic idea is quite simple. Whenever the client application asks for a reference to an EJB object, the Container actually returns a reference to a Container-generated proxy that intercepts all client calls, performs some housekeeping and eventually delegates to the object implemented by the Bean Provider. We call the former EJB

Object, and the latter EJB Instance. Please make sure you don’t confuse the two: the EJB Object is generated by the Container, whereas the EJB instance is implemented by the Bean Provider.

You should also note that the presence and role of the EJB Object is a standard part of the EJB specification. Also note that its implementation strategy and the way it cooperates with the EJB instance is Container-specific.

Finally, the Container-generated proxy (the EJB Object) is not to be confused with the RMI proxy, which is also present. The diagram below should help clarify the architecture.



The diagram shows the architectural positioning of the EJB object relative to other major pieces. In the diagram, I’m showing the case of a remote client application, but you should keep in mind that there is always an intercepting EJB object, even when the method calls come from code in the same Container (another EJB, for example).

The RMI proxy and stub, on the other hand, are present only when the client accesses the remote view of the EJB component. Please note that the RMI proxy and stub are automatically generated by the Container, although different Containers do so employing different strategies.

The dashed arrows represent dependencies, and they mean that if, for example, the implementation of the EJB Instance changes, the EJB Object should also be modified, because it relies on specific methods implemented by the EJB Instance.

The fact that an EJB Instance is never accessed directly is the basic mechanism by which the Container manages your bean instances. In the EJB parlance we say that the Container injects code between the client (whatever that is) and the EJB Instance, and we call the EJB Object a Container artifact. The code injected by the Container provides the EJB instances with middleware services like instance pooling, concurrency management, security management, transaction management and object persistence.

This takes a big burden off your shoulders – imagine how much code you would need to write to implement some or all of these services yourself. But since you have an EJB Container, you can simply tell it what you want it to do on your behalf – for example, start a new transaction when a specific method is called.

In EJB we refer to the declarative control of the transactional and security logic as meta-programming. The idea is that you are, in fact, programming your component or application, but part of the behavior you don’t code yourself – you just tell the Container what to do.

At this point you have all the architectural background that you need to start looking at how we implement and use an Enterprise JavaBean. I’ll give you more information about the EJB Object and the way the Container works internally as we go through the code examples.

Software

Before we start looking at the code, let’s briefly cover the software you’ll need to install to run the examples.

All the code for the examples is provided with the book. I recommend that you use the code and the build scripts provided, since the build process involves several steps and multiple source files.

All the EJB examples in this chapter have been tested on the JBoss 3.0.3 application server (JBoss is the leading open-source EJB Container, and it can be downloaded for free from http://www.jboss.org). Although I used JBoss, and more specifically the version that comes with the embedded Tomcat Servlet container, the amount of JBoss-specific information in this chapter and in the examples has been kept to the minimum, so porting these examples to a different Container should require very little or no code modification. Please note, however, that if you use a different JBoss version you may need to adjust the Ant build scripts accordingly.

The other software tools that you’ll need are: Java SDK1.4 or above; Ant and JUnit (which you have probably already installed if you have run other examples from this book); and XDoclet, another open source utility that will help us in the development of our EJBs.

The installation guide that you’ll find in the c18 code directory will provide all the additional details.

The example application

All the examples in this chapter follow a common theme to exemplify the concepts. We will build a simple application, called JavaTheater, to sell and buy movie tickets.

The application will be incrementally built, example by example, and we'll have to address movies, shows, tickets, customers, credit cards and so on. I don't have the space to show you how to create a full-fledged application, so I'll focus on its core functionalities.

There will be classes that represent persistent entities like a movie and a show, and other classes that implement the bulk of the application logic . For example the steps required to actually purchase one or more movie tickets for a particular show.

Before we see the actual code, I'd like to mention a few things about the source file structure. All the examples are in the c18 directory of the book source code. Each subdirectory is a separate, self-contained example built on top of the previous one. At the top level of each example subdirectory there is the Ant build script. At the same level, there is a src subdirectory, which contains all the source code for that example. The src directory contains the ejb-tier and rmiclients subdirectories which contain the code for the EJB implementation and for the client application (including JUnit test), respectively. Also please note when you build the example, the Ant script will create working directories for compilation and building (refer to the comments in the Ant script for details), but nothing is created or modified in the src directory. Finally, all of our classes are defined in the javatheater package, with a few subpackages that are worth mentioning: javatheater.client contains the client applications; javatheater.test contains the JUnit test cases; javatheater.ejb contains our EJBs' interfaces and related classes, and javatheater.ejb.implementation contains the EJB implementation classes.

Our first example, example01, doesn't use any EJB functionality. It's a plain and simple Java application that we use to prototype some of the fundamental domain classes. The simplest class is Movie, and you can see its implementation below:

```
//: c18:example01:src:javatheater:Movie.java
//
package javatheater;

public class Movie {
  int id;
  String title;

  public Movie(int id, String title) {
    this.id = id;
    this.title = title;
  }

  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public String getTitle() {
    return title;
  }

  public void setTitle(String title) {
    this.title = title;
  }

  public String toString() {
    return "[" + id + "] " + title;
  }
}
///:~
```

It defines a Movie class with two properties: an id (an int) and a title (a String). The class also defines a constructor plus setters and getters to access the property values. Of course, a more realistic movie object would need to contain a lot more information like the cast, the director, the rating, synopsis and so on; but implementing those would have added unnecessary complexity to the example.

Another class is Show, which is also pretty simple: it has an id (an int), a show time (a String) and the number of available seats (an int). It also contains a reference to a Movie object, which means that there is a uni-directional, many-to-one relationship between shows and movies: a movie can be associated to many shows, but a show has only one movie. In other words, it's easy to navigate from a show to a movie, but there is no way to get the list of related shows directly out of a movie. Here is the implementation of the Show class:

```
//: c18:example01:src:javatheater:Show.java
//
package javatheater;

public class Show {
  int id;
  Movie movie;
  String showtime;
  int availableSeats;

  public Show(
    int id, Movie movie,
    String showtime, int availableSeats
  )
  {
    this.id = id;
    this.movie = movie;
    this.showtime = showtime;
    this.availableSeats = availableSeats;
  }
  public int getId() {
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public Movie getMovie() {
    return movie;
  }

  public void setMovie(Movie movie) {
    this.movie = movie;
  }

  public String getShowtime() {
    return showtime;
  }

  public void setShowtime(String showtime) {
    this.showtime = showtime;
  }

  public int getAvailableSeats() {
    return availableSeats;
  }

  public void setAvailableSeats(int availableSeats) {
    this.availableSeats = availableSeats;
  }

  public String toString() {
    return
      "[" + id + "] " + movie + ", " + showtime +
      ", " + availableSeats;
  }
}
///:~
```

Since example01 is a simple Java application, it doesn't employ the many services offered by the EJB architecture. Specifically, we don't have any persistence support. I didn't want to use JDBC to store and retrieve the object state from a database (that would have implied object/relational mapping, something too complex for our first example), so there is a class in this example, called Storage, that simulates the presence of a persistent datastore.

```
//: c18:example01:src:javatheater:Storage.java
//
package javatheater;
```

```
public class Storage {
  private static Storage ourInstance;
  private Movie [] movies;
  private Show [] shows;

  public synchronized static Storage getInstance() {
    if (ourInstance == null) {
      ourInstance = new Storage();
    }
    return ourInstance;
  }

  private Storage() {
    movies = new Movie[] {
      new Movie(1, "Return of the JNDI"),
      new Movie(2, "A Bug's Life"),
      new Movie(3, "Fatal Exception"),
      new Movie(4, "Silence of the LANs"),
      new Movie(5, "Object of my Affection")
    };

    shows = new Show[] {
      new Show(1, movies[0], "5:30pm", 100),
      new Show(2, movies[0], "7:00pm", 300),
      new Show(3, movies[1], "6:00pm", 200),
      new Show(4, movies[4], "9:00pm", 200)
    };
  }

  public synchronized Movie findMovieById(int id) {
    for (int i = 0; i < movies.length; i++) {
      Movie movie = movies[i];
      if (movie.id == id)
        return movie;
    }

    return null;
  }

  public synchronized Movie findMovieByTitle(String title) {
    for (int i = 0; i < movies.length; i++) {
      Movie movie = movies[i];
      if (movie.title == title)
        return movie;
    }

    return null;
  }

  public synchronized Movie [] findAllMovies() {
    return movies;
  }

  public synchronized Show findShowById(int id) {
    for (int i = 0; i < shows.length; i++) {
      Show show = shows[i];
      if (show.id == id)
        return show;
    }

    return null;
  }

  public synchronized Show findShowByMovie(Movie movie) {
    for (int i = 0; i < shows.length; i++) {
      Show show = shows[i];
      if (show.movie.id == movie.id)
        return show;
    }
    return null;
  }

  public synchronized Show [] findAllShows() {
    return shows;
  }
}
///:~
```

This class implements the Singleton design pattern (there would only be one data store in our system). Its private constructor populates a fictitious database with "Java-related" movies and shows. It also exposes public methods to retrieve movies and shows from the fictitious database using different criteria. There is no method to update our fake database, but you are free to add those as an exercise, if you feel so inclined.

The last class in example01 is our main class, ShowListing, which simply uses Storage to locate a bunch of shows, and prints them out.

```
//: c18:example01:src:javatheater:ShowListing.java
//
package javatheater;

public class ShowListing {
  static Storage storage = Storage.getInstance();

  public static void main(String[] args) {
    Show [] shows = storage.findAllShows();
    for (int i = 0; i < shows.length; i++) {
      Show show = shows[i];
      System.out.println(show);
    }
  }
}
///:~
```

Again, this example is intentionally simple, but it introduces a few fundamental concepts that we'll apply using the EJB architecture: object persistence, separation of entity classes from business logic classes, and object relationships. Plus, it gives you an idea of what the application we are about to build is all about.

Your first Enterprise JavaBean

The first EJB we'll implement represents a movie in your ticketing system. I chose this type of object because it contains no business logic and very little persistent information, so it will not distract us from the actual practice of implementing an EJB.

Since a movie contains information (id and title) that must be persistent, we'll implement it as an entity bean, using Container Managed Persistence. This EJB will be called, not surprisingly, Movie.

Please note that this example is fundamental, since although we are going to implement a specific type of EJB (an entity bean), most of the information that you are going to see applies both to session beans and entity beans in their different variations.

To implement the Movie bean, you will need to define two interfaces, the home interface and the component interface, and one implementation class. Depending on which kind of view you want your bean to expose, the home and component interfaces can be implemented as remote or local. We'll define the Movie interfaces as remote for the moment.

You will also need to provide an XML file, called the deployment descriptor, that you will use to tell the Container about some characteristics of your EJB, including persistence, transactional and security attributes. Finally, since the bean we are implementing is an entity bean, we may want to supply a class for the bean's primary key (or you can just use some predefined class, which is what we will do).

I'm going to cover each of the five elements listed above (home and component interfaces, implementation class, deployment descriptor and primary key) in the next sections. I'll also give you a bit more information about the Container internals and Bean deployment.

Home Interface

We our going to provide our Movie bean with a remote view. That means that when the client instantiates an EJB, the actual instance is created in the memory space of a process on the server, and the client gets a reference to that instance.

Since the EJB instance is going to be remote, the client code can't instantiate an EJB using the Java new operator, because new always allocates instances in the local address space.

You could in theory use the new operator to instantiate a bean that exposes a local view, but the EJB specification makes the remote/local instantiation differences disappear by means of a factory object. A factory exposes the methods to instantiate your beans, and its implementation, generated by the Container, provides the logic to instantiate an EJB object in the Container's address space.

But that just shifts the problem: how do we get a reference to the factory object? The answer is JNDI. As you may remember, JNDI is our universal discovery mechanism in EJB, and all EJB Containers run an internal JNDI Service that clients can access across the network.

When an EJB is made available to the Container, it is also given a JNDI name by either the Component Provider or by the Deployer. Client applications use that JNDI name to locate the factory object for that EJB (our Movie, for example), and then use the factory to instantiate Movie objects.

The factory will expose the bean's home interface, that you arbitrarily define (not completely arbitrarily though, it must conform to the EJB specification). The fact that you define the interface lets you specify which arguments to pass with method calls. For example, you can decide what information is required to instantiate a Movie. The implementation of that interface will be dynamically synthesized by the Container, which will find out about the interface methods using Reflection.

If you are defining the home interface for an entity bean, there is another thing you can do with it in addition to creating instances. You can find existing ones. That makes sense if you consider that entity beans are, by definition, persistent in some data store; so being able to get a reference to one or more beans, created by someone, some time ago, is what you would expect.

In a home interface, the methods that you use to instantiate new EJBs are called the create methods, and the methods that you use to retrieve entity bean instances are called the finder methods.

Below is the definition of MovieHome, the home interface for the Movie entity bean.

```
//: c18:example02:src:ejb-tier:javatheater:ejb:MovieHome.java
package javatheater.ejb;
```

```
import javax.ejb.*;
import java.util.Collection;
import java.rmi.RemoteException;

public interface MovieHome extends EJBHome {
  public Movie create(Integer id, String title)
    throws RemoteException, CreateException;

  public Movie findByPrimaryKey(Integer id)
    throws RemoteException, FinderException;
}
///:~
```

You should note that this interface is derived from javax.ejb.EJBHome, which is in turn derived from java.rmi.Remote. This is because we decided to provide our Movie bean with a remote view, and the implication is that all methods in your home interface will have to conform to the Java RMI specification. In particular, all of them will have to include java.rmi.RemoteException in their exception specification.

The EJB specification describes all the syntactical and semantic requirements for the create and finder methods. If you want all the details, please refer to the EJB specification that you can download from the Sun's web site. In this chapter however, due to the limited space, I will just describe the meaning of the method signatures.

The first method in the interface is a create method that clients can use to instantiate new Movie EJBs. It returns a Movie reference. Movie is the component interface of our Movie bean, and I will describe it in the next section. The first argument to the create() method is what we'll use as the primary key for our bean. In this case, we decided to use an Integer to uniquely identify Movie instances. Primary keys will be described in more detail in one of the next sections. As the second argument, we'll pass information that represents the Movie's state – in this case, just the movie title.

Finally, the method's exception specification states that create() may throw a RemoteException and a CreateException. RemoteException is required by RMI, and should be no surprise to you. CreateException is an application-level exception that is part of the EJB API, and is generated when the bean's instantiation fails inside of the Container. A specific reason why this may fail is if you try to instantiate an entity bean with the same primary key as another existing entity bean of the same type. This specific case is represented by DuplicateKeyException, a subclass of CreateException.

The second method in the interface is a finder method that clients use to retrieve one existing instance of an entity bean. It's called findByPrimaryKey(), and not surprisingly, it takes an Integer as the only argument (Integer is the primary key type) and returns a Movie reference. It also throws a FinderException, in case something goes wrong inside of the Container. If the specific problem is that no instance was found with that primary key value, the Container will generate an ObjectNotFoundException, a subclass of FinderException.

All other finder methods, which return Java collections rather than a single reference, will not generate an ObjectNotFoundException, but simply return null. Also, the findByPrimaryKey() method is mandatory in the home interface of an entity bean.

You can have as many finder methods as you like, to search persistent instances by whatever criteria you fancy. However, all finder methods must have a name starting with find..., must return a collection and must throw FinderException. The only finder method that returns a single instance is findByPrimaryKey(), since only the primary key guarantees a unique result – I will cover finder methods in more detail later in this chapter.

There can also be multiple create methods in a home interface, and even EJB business methods (something similar to Java static method, which doesn't operate on a specific instance). We'll talk about these additional methods later; for the moment, let's move on to the next interface.

Component Interface

The component interface contains the business methods that clients invoke on a specific EJB instance, like getTitle() on a Movie object. For this reason, the component interface is arbitrarily defined by the Bean Provider, but it must still conform to the EJB specification. The only requirements though, are that the interface is derived from javax.ejb.EJBObject and that all methods throw RemoteException.

Below is the definition of the Movie entity bean component interface.

```
//: c18:example02:src:ejb-tier:javatheater:ejb:Movie.java
package javatheater.ejb;

import javax.ejb.*;
import java.rmi.RemoteException;

/**
 * A movie, with id and title.
 *
 * Note that there is no setId() method in the
 * interface, to prevent clients from arbitrarily
 * changing a movie's primary key.
 */
public interface Movie extends EJBObject {
  public Integer getId() throws RemoteException;
  public String getTitle() throws RemoteException;
  public void setTitle(String title)
    throws RemoteException;
}
///:~
```

You see that the interface just provides setters and getters for the movie's attributes (there is rarely business logic in entity beans). However, it's missing the setter for the movie's id. The reason being that you do not want to give clients the option to modify the primary key on an existing persistent instance, since that could very easily compromise the referential integrity of your system if that instance participates in any relationship.

Primary Key

At the conceptual level, a primary key in EJB is the same as it is in a relational database. It's what you use to uniquely identify a group of information: a record in a database, or an entity bean instance in an EJB Container.

However, in EJB this concept is intentionally more abstract than in a relational database, and the reason is that the EJB architecture is designed to work with a number of different back-end systems including, but not limited to, a relational database. For example, your back-end persistence storage may be an object-oriented database, where there is no primary key as in a relational database, but object identifiers instead.

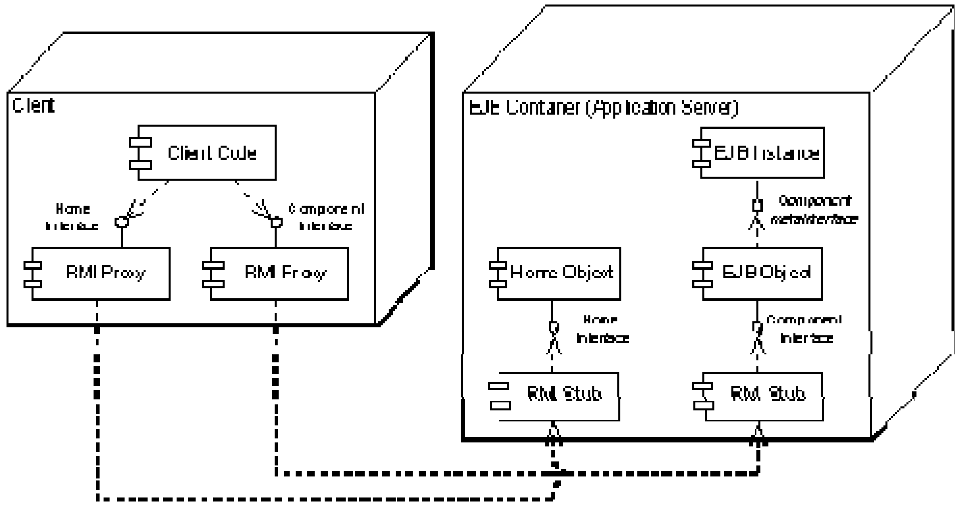
So in EJB you can use any kind of object as your primary key. In other words, you can use any class derived from Object, but not primitive types. That's why we used an Integer instead of an int as the Movie's primary key.

EJB primary keys can contain one single value, or multiple values (to have a compound primary key). They can also be classes whose definition is deferred until deployment time. For more information, including implementation requirements, please refer to one of the recommended readings at the end of this chapter. In this chapter, I will just use simple standard classes as primary keys.

Implementation Class

Now that we have defined our home and component interfaces, we can proceed to implementing the Movie bean. You may expect to define a class that implements both interfaces but, interestingly enough, you don't. Instead, you will need to define a class that implements one of the EntityBean, SessionBean or MessageDrivenBean interfaces, depending on which kind of bean you are implementing.

So, who's going to implement the home and component interface that you defined? The Container is, of course. The diagram below should help you understand how and why the Container dynamically synthesizes implementation classes for your home and component interface. In the diagram below we assume that the client application works using a TCP/IP connection, although I've omitted it from the diagram.



If you look carefully at the diagram above, you'll see that you don't actually implement any of the interfaces you define. All of them are implemented by Container-generated classes: the RMI components, the Home object and the EJB Object.

There is a very good reason for not having the Bean Provider implementing those interfaces. If you inspect the interface hierarchy, you'll discover that your interfaces contain several methods, inherited from the base interfaces, that are not supposed to be implemented manually, because they are part of the RMI or Container infrastructure.

The Container provides the implementation of the home interface (the Home object), and the implementation of the component interface (the EJB object), so that all client calls will be intercepted. Eventually, the Container will delegate those calls to your Movie instance, but the Movie implementation class will not implement MovieHome and Movie.

Which interface then is the EJB instance supposed to expose, so that the EJB Object can delegate client calls to it? For example, the Movie's getTitle() method is defined in the Movie interface, which is not implemented by EJB Movie instance. So which interface defines the semantic contract between the Movie EJB object and the Movie EJB instance?

You must remember that the EJB Object implementation is synthesized by the Container. That implementation is modeled on the definition of the component interface that you provide, which the Container discovers using reflection.

What you have to do is conform to a naming convention defined by the EJB specification, that describes which methods the EJB object expects to find on your EJB instance (a Movie, in our case). This naming convention describes the semantic rules that you must follow so that the Container can communicate with your EJB instance. In this chapter, I refer to the outcome of applying this rule to the component meta-interface. It’s not a Java interface in a strict sense, but it’s still a set of methods that, in practice, defines the contract between the Container-generated EJB object and the programmer-defined EJB instance.

There are two categories of methods in the component meta-interface: Container notification methods and business methods.

Container notification methods are methods in your bean implementation class that the Container calls under special circumstances. They are easy to spot because their names all start with the ejb<...> prefix – for example, ejbCreate(). The only exceptions to this rule are the Context-related notification methods: setEntityContext(), unsetEntityContext(), setSessionContext() and setMessageDrivenContext(). I’ll explain each of the Container notification methods later, when we look at the source code of the implementation class.

Business methods are simply the methods that you have defined in your component interface. All those methods must be in your EJB implementation class with exactly the same signature, except for RemoteException, which you do not include in the methods’ exception specification.

There is however one important architectural part of the EJB 2.x specification which affects the actual business methods that you have to implement. Starting with EJB 2.0, the architecture provides an abstract programming model for CMP entity beans. It means that the implementation of the persistent fields of an entity bean is not a Bean Provider’s responsibility, but is instead delegated to the Container. We refer to Container-generated persistent fields as virtual fields.

Let’s take for example the Movie bean. You can tell by the presence of the setters and getters in the component interface that Movie has two properties: id and title. You may expect: a) that you have to implement the getters and setters; b) that you have to declare an Integer and a String in your implementation class to hold the value of the id and title respectively, and; c) that the state of those variables will then be kept in sync by the Container with the corresponding state in the persistent storage.

That would be exactly the case if you were coding for EJB 1.x. However, EJB 2.0 introduces a simple concept, which is that the Container can do a much better job than the developer at providing the implementation of those persistent properties. The reason for this assumption is that the Container may have intimate knowledge of that specific storage which the developer doesn’t have, so it can optimize better (for example if you use the WebSphere container with a DB2 database, both from IBM).

So, in the case of methods representing persistent properties of an entity bean, you don’t provide those method implementations, because the Container will generate them for you.

In your code you declare the implementation class of your entity bean as abstract, and it will be used as the base class for a Container-generated one (yes, another one!)

that implements the persistence logic. You simply have to declare the methods representing the persistent properties as abstract methods in your implementation class.

Let’s take a look at MovieBean, the implementation class for our Movie CMP entity bean.

```
//: c18:example02:src:ejb-tier:javatheater:ejb:implementation:MovieBean.java
package javatheater.ejb.implementation;

import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;

public abstract class MovieBean implements EntityBean {
  // Container notifications methods
  public Integer ejbCreate(Integer id, String title)
    throws CreateException {

    if (id == null)
      throw new
        CreateException("Primary key cannot be null");

    if (id.intValue() == 0)
      throw new
        CreateException("Primary key cannot be zero");

    setId(id);
    setTitle(title);

    return null;
  }

  public void ejbPostCreate(Integer id, String title) {}
  public void ejbLoad() {}
  public void ejbStore() {}
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void setEntityContext(EntityContext ctx) {}
  public void unsetEntityContext() {}

  // Business methods
  public abstract void setId(Integer id);
  public abstract String getTitle();
  public abstract void setTitle(String title);
  public abstract Integer getId();
}
///:~
```

Note that the class is abstract and that it implements the EntityBean interface, but not the home or component interface.

Regarding the business methods, the last four in the class, there is nothing to add. They represent persistent properties in our Movie CMP entity bean, so you just declare them as abstract and the Container will provide the implementation.

The Container notification methods, instead, need to be explained one by one. Remember that the Bean Provider defines these methods, but never calls them. All these methods are called by the Container only, under different circumstances.

Here follows a brief description of all the notification methods for an entity bean:

- ejbCreate(): called when the instance is created and assigned to the client, usually in response to the client calling a create method on the home interface. The ejbCreate() method is responsible for validating input arguments and initializing the instance fields using the setter methods. It takes the same arguments as a corresponding create method in the home interface, but the return type is the bean's primary key, not the component interface. The returned value of the primary is null if the entity bean uses CMP, or the actual value if it uses BMP. This method is called before the new instance is stored in the persistent storage.

- ejbPostCreate(): there must be a corresponding ejbPostCreate(...) method for each ejbCreate(...) method in the class. The return type is always void. This method is called after the new instance has been stored into the persistent storage, and it's used to implement code that relies on the entity having been assigned a valid primary key in the persistent storage – typically to establish relationships with other entities.

- ejbRemove(): Called in response to the client calling the remove() method on the home or component interface. Executes before the entity is removed from the persistent storage.

- ejbLoad(): Called right after the EJB object's state has been restored from the persistent storage.

- ejbStore(): Called before the EJB object's state is copied to the persistent storage.

- ejbActivate(): Called right after the EJB object's state has been restored from some secondary storage, usually after the instance has been retrieved from a pool.

- ejbPassivate(): Called right before the EJB object's state is saved to some secondary storage, usually before the instance is returned to a pool.

- setEntityContext(): Called by the Container to pass the EJB object its Context. The Context object provides runtime information specific to that EJB object, for example the security role under which a method is being called. The EJB object is supposed to store the Context reference in a field for later use.
- unsetEntityContext(): Called to inform the EJB object that the previously passed Context is no longer valid.

Once again, I'd like to refer you to the EJB specification, or to one of the other sources listed at the end of this chapter, for detailed information about the Container notification methods and the Context interface. For the moment, the information I provided should be sufficient to understand what the MovieBean class is doing.

You see that the only method we implement is ejbCreate(), where we validate the id and title, and throw a CreateException if they don't have legal values. Once the arguments have been validated, we use the setters to initialize the values of id and title, our bean's persistent properties. All the other Container notification methods don't need to do anything special on our simple Movie bean, so we just provide empty method bodies.

That wraps it up for the Java code, but not with the source. We still need to provide our Movie bean with an additional element.

Deployment Descriptor

The deployment descriptor is an XML file that contains structural information about an EJB. This information includes, but is not limited to: the EJB type; the Java interfaces and classes that represent its home and component interfaces; its implementation class and its primary key; the EJB logical name used in that specific deployment descriptor; and the persistent fields of an entity bean.

The deployment descriptor can also include assembly information about an EJB, for example logical names used in the bean implementation that need to be mapped to physical resources.

This file, which must be named ejb-jar.xml, is created by the Bean Provider, and can be edited by the Bean Deployer and/or by the Application Assembler. In other words, the deployment descriptor contains bean configuration information that can be modified to adapt the component to a specific operating environment.

Below is the deployment descriptor for the Movie bean (I've omitted the xml and DOCTYPE elements):

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>Movie</ejb-name>
      <home>javatheater.ejb.MovieHome</home>
      <remote>javatheater.ejb.Movie</remote>
      <ejb-class>
        javatheater.ejb.implementation.MovieBean
      </ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>Movie</abstract-schema-name>
      <cmp-field><field-name>id</field-name>-</cmp-field>
      <cmp-field><field-name>title</field-name>-</cmp-field>
      <primkey-field>id</primkey-field>
    </entity>
  </enterprise-beans>
</ejb-jar>
```

The structure of the descriptor should be self-explanatory, so I won't go into all the details. However, the following XML elements are worth a few words:

- <entity>: Defines an entity bean.
- <ejb-name>: Logical name under which this bean can be referenced in other parts of the current deployment descriptor. Not to be confused with the JNDI name (discussed later).
- <home>: Fully qualified name of the Java interface used as the bean's remote home interface.
- <remote>: Fully qualified name of the Java interface used as the bean's remote component interface.
- <ejb-class>: Fully qualified name of the Java class used as the bean's implementation class.
- <persistence-type>: Container for CMP, Bean for BMP
- <prim-key-class>: Fully qualified name of the Java class used as the entity bean's primary key.
- <primkey-field>: Name of the field in the entity bean that represents its primary key
- <cmp-field>: Name of a field whose persistence is managed by the Container. An entity bean can have multiple <cmp-field> elements.
- <abstract-schema-name>: Name used to reference this entity bean in queries expressed in the EJB Query Language (EJB-QL), covered later.

A deployment descriptor can be, and usually is, richer and more complex than what you see in this example. That's why deployment descriptors today are seldom created by hand. The developer relies instead on development tools that provide assistance in the definition of the bean interfaces, classes and deployment descriptor. Most of these commercial tools provide some sort of RAD environment where “wizards” provide you with the structural information to generate the source code. As you’ll see later, in this chapter we’ll use a completely different and equally powerful tool for automatic source code generation.

Packaging

Now that you have all the source files that constitute the implementation of our Movie bean, we need to package them together for distribution.

First, you’ll need to compile the code, since what we distribute are class files, not source files. Second, you need to package all the required files in a jar file, called the ejb-jar for your component, which you create using the jar utility that comes with the J2SE SDK.

You know that jar files can maintain an internal directory and file organization. The internal file structure of an ejb-jar must conform to the following specification: all Java class files start from the top level, and are structured in subdirectories according to their package names; and the deployment descriptor must be in a subdirectory at the top level called META-INF (must be all capitals).

The diagram below shows the internal organization of javatheater.jar , the ejb-jar of the Movie bean.

```
javatheater.jar

javatheater/
    ejb/
        MovieHome.class
        Movie.class
        implementation/
           MovieBean.class

META-INF/
   ejb-jar.xml
```

This file is your component’s distribution unit: it contains the interfaces used by the client code, the implementation class, and the configuration information.

The ejb-jar is made available to the EJB Container running on the server, but is generally not available to client applications. For this reason, you usually create another jar file. It’s called the client jar, and it contains only the classfiles required by client applications to access your remote component.

In our example, the only two classfiles that a client will need are the definitions of the MovieHome and Movie interfaces. Although there is another architectural piece that remote clients will need – the RMI proxies. The next section on Deployment will cover that bit as well.

Deployment

Deployment is the act of making an EJB component available to a particular implementation of an EJB Container, which uses application-specific external and internal resources. Deployment is an inherently Container- and application-specific operation, since it implies customization of the EJB Container and of the component (for example, associating a generic component with a predefined, custom database).

Even for deploying our simple Movie bean you will need to take care of at least: a) assigning a JNDI name to your component so that clients can locate it; b) associating the Movie entity bean with a physical database, and; c) making sure to provide the remote clients with the RMI proxies for the MovieHome and Movie remote interfaces.

Conceptually, deployment is the same on all platforms, but the deployment practices can vary substantially from one Container to another. All commercial EJB Containers come with some graphical utility that you use to assemble your ejb-jar and client jar, to pass the ejb-jar to your Container, to customize your EJBs and to associate it with system resources like databases, message queues, and so on.

Please note that since we use JBoss as our EJB Container, this section on deployment will be JBoss-specific, but all the information that you've seen so far is Container-independent.

JBoss makes deployment per se extremely simple: all you have to do is to place your ejb-jar is a special sub-directory of your JBoss installation, as explained later. JBoss will detect the new file (even when the Container is running), inspect it and validate it. If all goes well, the new component will be deployed and the clients will be able to use it.

However, there is no GUI tool coming with JBoss that will assist you with the creation of the ejb-jar, deployment and the configuration of your enterprise bean. You will have to provide all the relevant information in the deployment descriptor and/or some JBoss-specific XML files.

Let's now take a look at what you have to do in JBoss to properly set up the EJB's JNDI name, the database, and the RMI proxies.

JNDI name: by default, JBoss will create a JNDI name for your EJB using the same name as the ejb-name in the deployment descriptor ("Movie", in our case). However, in a real application you'll want to specify your own JNDI for the Movie bean, perhaps to avoid name conflicts with other components. To do that, you have to provide an XML file named jboss.xml in the same directory where the deployment descriptor is, and use it to map the JNDI you want to the ejb-name of the component. Below is an example of the jboss.xml file (note how the JNDI name is different from the ejb-name).

```
<?xml version="1.0" encoding="UTF-8"?>

<jboss>
  <enterprise-beans>
    <entity>
      <ejb-name>Movie</ejb-name>
      <jndi-name>javatheater/Movie</jndi-name>
    </entity>
  </enterprise-beans>
</jboss>
```

Database: JBoss comes with a default database that's started when you start the Container. In the absence of further deployment information, all CMP entity beans will be made persistent on this database. JBoss will also create database tables named after your entity beans, and store each entity bean instance as a record in the appropriate table. The database engine that JBoss uses by default is HypersonicSQL, which comes bundled in the JBoss distribution. You can manipulate the data in the default JBoss database using the Hypersonic Database Manager (refer to the installation guide for details). However, if you use JBoss as a real production environment, you'll want to use a different database, and/or provide a custom entity/table mapping (please refer to the JBoss manual for details). Also, please note that other EJB Containers require that you create the database schema beforehand, and map your entity beans and fields to tables and columns.

RMI proxies: Most EJB Containers will generate the implementation classes for the RMI proxies when you deploy an enterprise bean, using the Container's graphical tools. The classfiles for the RMI proxies will be included in the client jar, which you make available to the client applications. JBoss doesn't require this step, since it synthesizes the RMI proxy classes at runtime, and automatically sends the classfiles to the clients across the networks. So, although there are RMI proxies using JBoss, there are no classfiles that you have to make available to the clients.

Building it all

As you can imagine, the process of pulling all the pieces together, to end up with a properly deployed component, is very error-prone and tedious. You certainly don't want to manually compile all the required files, create the ejb-jar and client jar and deploy.

As I mentioned, all commercial EJB Containers come with some sort of tools to assist you in this process. JBoss doesn't provide such a tool, but we can still automate all the build and deployment process using Ant. All the examples in the source code distribution for this chapter come with their Ant script that you can use to compile, deploy, create the jars, deploy, undeploy and so on – just run the Ant script with no arguments for a list of available targets.

Another thing that JBoss does when you give it an ejb-jar is to verify it. That is, JBoss will check the contents of the XML files and classfiles in the ejb-jar, to verify that they conform to the syntactical and semantic rules of the EJB specification. But the JBoss verifier is a tool that you can also run separately, and that is exactly what our Ant scripts do in their process of building and deploying the javatheater components.

In the end, not having to use a GUI tool to deploy components can be a good thing, because then you have more control when you programmatically build and deploy your components.

An EJB client application

The code in the c18/example03/rmiclients directory contains an example of a simple remote client Java application that uses our Movie bean to create, browse and delete movie entries in the server-side system.

As with all EJB remote applications, there are four things that you need to do for the application to run: 1) ensure that the client application can locate the packages required by the EJB API and by your Container; 2) place the client jar, containing the definition of the home and component interface, in the client application's classpath; 3) configure the JNDI properties so that the client application can connect to the JNDI service in the EJB Container, and; 4) use the JNDI API to locate EJB homes.

The EJB and JBoss client packages are in the JBoss distribution, whereas the client jar for the Movie bean can be easily created by the Ant script.

We are going to set the JNDI properties using the external jndi.properties file, which must also be in the client's classpath, as I explained in the JNDI section above.

Our JNDI service provider will be JBoss (one of its services, to be precise), so we need to provide the following JBoss-specific JNDI configuration information in the jndi.properties file:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming
```

Please note that if your client application and your JBoss installation are running on two separate machines, you will need to replace localhost in the file above with the IP address of the machine where JBoss is running.

Below is the code for our client application, MovieClient:

```
//: c18:example03:src:rmiclients:javatheater:client:MovieClient.java
package javatheater.client;

import javax.naming.*;
import javatheater.ejb.*;

public class MovieClient {
  public static void main(String[] args) throws Exception {
    javax.naming.Context initial =
      new javax.naming.InitialContext();

    Object objRef = initial.lookup("javatheater/Movie");
    MovieHome movieHome =
      (MovieHome) javax.rmi.PortableRemoteObject.narrow(
        objRef,
        MovieHome.class);

    // Generate a primary key value
    int pkValue =
     (int) System.currentTimeMillis() % Integer.MAX_VALUE;

    // Create a new Movie entity
    Movie movie =
      movieHome.create(
        new Integer(pkValue), "A Bug's Life"
      );

    // As a test, locate the newly created entity
    movie =
      movieHome.findByPrimaryKey(new Integer(pkValue));

    // Access the bean properties
    System.out.println(movie.getId());
    System.out.println(movie.getTitle());

    // Remove the entity
    movie.remove();
  }
}
///:~
```

If you remember the section about JNDI, the first couple of instructions should be easy to understand: we create an InitialContext, which according to our JNDI setting will point to the naming root in JBoss; then, we use the lookup() method to locate the home interface of the Movie, using the component's JNDI name that we have specified in the jboss.xml file.

The part using PortableRemoteObject instead, needs to be explained. What we are trying to achieve here is a type-safe remote downcast. You remember that the lookup() method returns a generic Object reference; and since we know that what we are looking up is a MovieHome interface, we need to downcast it to the appropriate runtime type. We also want to be informed, though, if we are downcasting to the wrong type. But if you think about it, the type checking has to be performed on the object on the server, not in the local process. What you have locally is the RMI proxy, not the actual object. The narrow() method implements this form of type-safe remote downcast. It goes across the network and checks that the runtime type of the object passed as the first argument is compatible with the type represented by the second argument. If the types are not compatible, you get an exception. If the remote casting is successful, you still need to downcast the local reference to the type you want, so that's the reason for the other Java downcast.

Please note that under certain circumstances, it would be acceptable to use just a Java downcast, but using the narrow() method guarantees that the remote cast will work on all platforms and environments.

Once we have a reference to the Movie home interface, we can start instantiating, locating and deleting Movie instances. Since Movie is an entity bean, we need to pass a new primary key value anytime we create a new one. To reduce the risk of using existing primary keys, which would make your application fail with a DuplicateKeyException, I use a simple computation based on System.currentTimeMillis() to generate a reasonably unique value. Please note though, that you should not use this technique in your production code, since the generated values are not guaranteed to be unique (the problem of generating unique primary key values is not trivial, and can't be covered here).

In the rest of the example, once a Movie has been created, we use its primary key to retrieve it from the persistent storage. Of course, the create() method does return a reference to the new instance, but I wanted to show the use of a finder method as well. Finally, we print its properties out and we remove it from the persistent storage.

To run this example you should use one of the runclient scripts that you'll find in the source directory. Please inspect the contents of the script first, and look at all the packages that the client application requires in order to run.

Once you get the example to run successfully, you may want to try to comment out the line that removes the entity, run the example once or twice, leave JBoss running, and use the Hypersonic Database Manager to inspect the contents of JBoss's default database. Just start the Database Manager as explained in this chapter's installation guide, then select "HSQL Database Engine Server" as the type, and use port 1476 in the URL.

In the c18/example04/rmiclients directory you'll find a different type of client: a JUnit test case for our Movie bean (it's defined in the javatheater.test package). A test case for EJBs is run by a Java application (a JUnit test runner or your own code), so everything that I've explained about remote client applications applies to running test cases as well.

There is one more piece of important information about JUnit. If you use one of the two graphical test runners shipped with JUnit, you should turn off the “reload classes on every run” option (there is a check box on the main window). If you leave this option on, JUnit will load your classes using a custom classloader. Unfortunately, this classloader is not designed to work across the network, which is exactly what’s required to load the dynamic RMI proxies that JBoss sends to the clients. If you don’t turn that that option off, your JUnit test cases will not be able to connect to your remote enterprise beans, and they’ll fail.

To run the JUnit test, I recommend that you use one of the runtest scripts in the source directory, and that you check their contents out to see how to use JUnit with remote EJBs.

Simplifying EJB Development

The first EJB example was designed to show you each and every line of code that is required to implement an EJB. In the process, you have also seen that you need at least four source files, whose content must be kept in sync. Even on a medium-size project, this proliferation of closely related source files can easily become unmanageable, or at least detrimental to the developer’s productivity.

We clearly need tools to reduce the amount of work required to pull all the pieces together. These tools exist, and they take the form of “wizards”
that are integrated in many commercial EJB development environments.

There is another option though, and that’s what we are going to use in this chapter. It’s XDoclet, an open-source tool that you can download from the SourceForge.net web site. XDoclet is actually a bundle of two different tools: WebDoclet, which I’m not going to cover here, and EJBDoclet.

Both tools take advantage of the Doclet API, the same one that JavaDoc is based on. JavaDoc reads your source code and uses the Java source, the Java comments and special tags to generate HTML documentation for your classes. EJBDoclet reads the source file with the definition of your Bean’s implementation class, and instead of generating HTML files, it generates Java source files with the definition of the bean’s interfaces, and XML files for the deployment descriptor and Container-specific configuration information. EJBDoclet can also generate other support classes, but I’m not going to cover those here.

EJBDoclet is quite a powerful tool that can drastically simplify your development process, not only because of the source code it generates on your behalf, but also because it can be invoked programmatically by Ant scripts, build daemons or other Java utilities, thus becoming part of a fully-automated build process.

However, it takes some time and patience to get EJBDoclet to work right the first time, primarily because the tool is highly configurable. I'm not going to explain how EJBDoclet works, but just show some of what it can do for you. Most of the Ant scripts for this chapter invoke EJBDoclet as part of their build process, but if you decide to use EJBDoclet in your own projects you may find it hard at first. Just be patient and invest a couple of days in experimenting with it: the benefits that you get from EJBDoclet will reward you for your efforts.

The basic mechanism in EJBDoclet is very simple: you provide your bean implementation class, enriched with special tags in your JavaDoc comments, and EJBDoclet will generate all the other source files that make up your EJB, according to your instructions.

EJBDoclet will also generate default implementations for the Container notification methods, which you would otherwise have to code by hand in your bean implementation class. However, since EJBDoclet can't and doesn't modify your source code, the additional implementation code is provided in an EJBDoclet-generated class, derived from the one you provide. The EJBDoclet-generated one, which will inherit all your code, will be the actual bean class that the Container will use.

Your implementation class has to be defined as abstract, and if you do provide the implementation of any Container notification method, the EJBDoclet-generated class will call into your implementation. Please note that, when you use EJBDoclet, your implementation classes are abstract regardless of the type of bean that you are implementing, whereas in our first example the MovieBean class was abstract because of the abstract programming model as defined in EJB 2.x for CMP entity beans.

Below is a new version of the Movie bean implementation class which uses EJBDoclet. This code is in the c18/example05/src/ejb-tier directory. Please note that there is no source code for the bean interfaces and for the deployment descriptor in this example, since those will be generated by the EJBDoclet.

```
//: c18:example05:src:ejb-tier:javatheater:ejb:implementation:MovieBean.java
package javatheater.ejb.implementation;

import javax.ejb.*;
import java.util.Collection;

/**
 * This is the <code>Movie</code> entity bean
 * implementation.
 *
 * @see javatheater.ejb.MovieHome
 *
 * @ejb:bean
 *  name="Movie"
 *  jndi-name="javatheater/Movie"
 *  type="CMP"
 *  primkey-field="id"
 *  reentrant="False"
 *
 * @ejb:pk class="java.lang.Integer"
 *
 * @ejb:home remote-class="javatheater.ejb.MovieHome"
 * @ejb:interface remote-class="javatheater.ejb.Movie"
 *
 * @ejb:finder signature=
 *   "java.util.Collection findByTitle(java.lang.String title)"
 */

public abstract class MovieBean implements EntityBean {
  /**
   * Validates the primary key and initializes the
   * bean properties.
   *
   * @ejb:create-method
   */
  public Integer ejbCreate(Integer id, String title)
    throws CreateException {
    if (id == null)
      throw
        new CreateException("Primary key cannot be null");
    if (id.intValue() == 0)
      throw
        new CreateException("Primary key cannot be zero");

    setId(id);
    setTitle(title);

    return null;
  }

  public void ejbPostCreate(Integer id, String title) {}

  /**
   * Returns the Movie id, which is also the Movie primary key.
   *
   * @ejb:pk-field
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public Integer getId();

  /**
   * This method is not part of the remote interface.
   */
  abstract public void setId(Integer id);
  /**
   * Returns the Movie title.
   *
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public String getTitle();

  /**
   * Sets the Movie title.
   *
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public void setTitle(String title);
}
///:~
```

Looking at the code above, you should note that there are three considerable advantages in using EJBDoclet:

1. The code above is the only source file that you need to implement your bean. There are no interface definition files, no deployment descriptor, no Container-specific XML files.

2. The class above contains only code that is meaningful to you – there is no “plumbing”, like empty C0ntainer notification methods.

3. You don’t define findByPrimaryKey(...). EJBDoclet always generates that method for you, since it’s required by the EJB specification, but other finders you would need to explicitly define (more about this later).

There is nothing new in the Java code. The interesting part is in the comments that enrich your code with meta-information, which EJBDoclet uses to generate code.

All the tags starting with ejb: are EJBDoclet comments. The ejb:bean tag, for example, you use to provide information about the entire bean, such as its type, EJB name, JNDI name, primary key field, and more. Other tags, like ejb:interface-method, you use to provide information that applies to a specific part of the bean. A method, in this case, that you want to be exposed in the bean’s component interface. For a complete description of what the other tags mean, please refer to the EJBDoclet manual.

The tags alone, however, do not provide all the information required to generate the source files. For example, the tags don’t specify in which directory the files should be generated, or how they should be called. This additional information is passed to EJBDoclet when it starts by the Ant task. Below is an excerpt of the Ant script in this example’s directory:

```
<ejbdoclet
```

```
  sourcepath="${ejb.source.dir};${gen.source.dir}"
  destdir="${gen.source.dir}"
  excludedtags="@version, @author"
  ejbspec="2.0">

  <fileset dir="${ejb.source.dir}">
    <include name="**/*Bean.java" />
  </fileset>

  <classpath>
    <path refid="server.classpath"/>
    <path refid="xdoclet.classpath"/>
  </classpath>

  <deploymentdescriptor destdir="${gen.source.dir}/META-INF"/>
  <homeinterface/>
  <localhomeinterface/>
  <remoteinterface/>
  <localinterface/>
  <entitycmp/>
  <session/>
  <jboss destdir="${gen.source.dir}/META-INF"/>
</ejbdoclet>
```

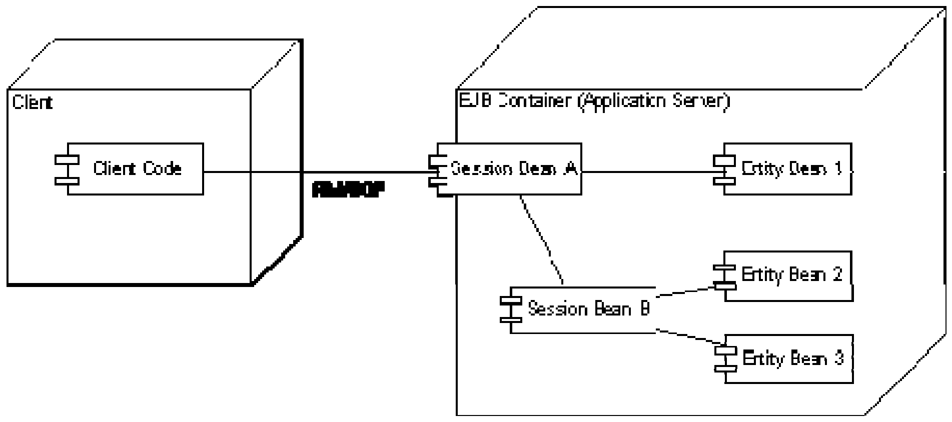
There are two interesting EJBDoclet features that show up in the Ant task: the first one is that you can decide which EJB specification version you want the generated files to comply with. In other words, the value of the ejb-spec parameter can be either “1.1” or “2.0”, and EJBDoclet will generate the XML descriptors and the Java files accordingly. The second feature is that you can specify which elements you want to be generated, and how. In the task above, the elements from <deploymentdescriptor> to <jboss> determine which source files EJBDoclet is going to generate, and where they should go (if you don’t want the default location). Specifically, we are instructing EJBDoclet to generate for us: the deployment descriptor; the home and component interface for the local and remote view of the component (if also requested by the ejb: tags in the source); the entity bean implementation classes; the session bean implementation classes, and; all the required JBoss-specific XML files (please note that EJBDoclet supports the generation of Container-specific files for a number of different EJB Containers as well).

Now that you have an idea of what EJBDoclet can do for us, we’ll use it for all the next examples in this chapter. Before proceeding, I’d like to invite you to build example05 and inspect the contents of the gen subdirectory, where all the generated source files are sent to, to gain a better understanding of the build process and EJBDoclet.

Implementing a session bean

We use Session beans to implement business logic as discrete, deployable units. For example, a session bean could implement the logic required to validate a credit card, and if you are developing an e-commerce application, you could just plug it into your application rather then implement the credit card validation functionality from scratch.

Session beans are also frequently used to implement distributed façades (although this is not the only pattern that fits the Session Bean model). A façade is an EJB that's accessible by remote applications and that hides from the client many or all of the other EJBs used in the application, as shown in the diagram below:



A façade has the advantage of simplifying the client's view of the remote application, since it hides most of the application's internal complexity. It can also make for reduced network traffic, if it provides bulk accessors to the clients –methods that can transfer a bulk of structured information in one single call across the network, rather than forcing the client to communicate with a bunch of different objects.

In the following example we'll implement a stateless session bean that we'll call ShowManager, and that will provide methods to edit the show program – adding and deleting movies, for now. ShowManager will also act as a façade to hide the primary key generation from the client (remember, every time we create a new movie we must provide a unique primary key value).

The generation of unique primary keys is a complex issue that I can't properly cover here, but that's covered exhaustively in many of the resources listed at the end of this chapter. For the next examples I want to present a more reliable solution than what you've seen in the previous client code. We'll create a new CMP entity bean, called AutoCounter, that we use to generate unique primary key value. The principle on which it works is very simple: an AutoCounter instance contains only one int property, and comes with a getNext() method that increments the property's current value and returns the result. Since AutoCounter is an entity bean, the counter value is automatically persistent in the database. We'll use a String as the bean's primary key, so that will also be the unique counter name.

Below is the implementation of the AutoCounter CMP bean, as defined in c18/example06/src/ejb-tier:

```
//: c18:example06:src:ejb-tier:javatheater:ejb:implementation:AutoCounterBean.java
package javatheater.ejb.implementation;

import javax.ejb.*;

/**
 * @ejb:bean
 *  name="AutoCounter"
 *  jndi-name="javatheater/AutoCounter"
 *  type="CMP"
 *  primkey-field="name"
 *  reentrant="False"
 *
 * @ejb:pk class="java.lang.String"
 *
 * @ejb:home remote-class="javatheater.ejb.AutoCounterHome"
 * @ejb:interface remote-class="javatheater.ejb.AutoCounter"
 * @ejb:finder signature="java.util.Collection findAll()"
 */
public abstract class AutoCounterBean
  implements EntityBean {
  /**
   * Validates the primary key and initializes the
   * bean properties.
   * @ejb:create-method
   */
  public String ejbCreate(String name)
    throws CreateException {
    if (name == null)
      throw new CreateException(
        "Counter name can not be null");
    if (name.length() == 0)
      throw new CreateException(
        "Counter name can not be an empty string");

    setName(name);
    setValue(0);

    return null;
  }

  public void ejbPostCreate(String name) {
  }
  /**
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public String getName();

  /**
   * @ejb:persistent-field
   */
  abstract public void setName(String name);

  /**
   * @ejb:persistent-field
   */
  abstract public int getValue();

  /**
   * @ejb:persistent-field
   */
  abstract public void setValue(int value);

  /**
   * Returns the next value in the sequence.
   * @ejb:interface-method
   */
  public int getNext() {
    int value = getValue();
    value += 1;
    setValue(value);
    return value;
  }
}
///:~
```

Each entity bean in the JavaTheater application will use a different AutoCounter instance, so that it will have its own set of primary key values. For the moment we just have a Movie entity bean, but later we will implement a Show entity bean. Since each counter is identified by a String name (its primary key), and different entity beans will use different counters, we want Movie and Show to provide their counter names.

I'm using this simple problem to introduce another EJB 2.0 feature: EJB home methods. A home method is implemented in a bean's home interface, and as such is not invoked on a specific instance – something similar to Java static methods, but for a distributed environment. Home methods can implement arbitrary code, not just code for creating, finding and deleting EJB instances as you would expect from a factory.

In the next example, in c18/example07/src/ejb-tier, I've added the getCounterName() home method to the Movie entity bean:

```
/**
 * This is the <code>Movie</code> entity bean
 * implementation.
 *
 * @see javatheater.ejb.MovieHome
 *
 * @ejb:bean
 *  name="Movie"
 *  jndi-name="javatheater/Movie"
 *  type="CMP"
 *  primkey-field="id"
 *  reentrant="False"
 *
 * @ejb:pk class="java.lang.Integer"
 *
 * @ejb:home remote-class="javatheater.ejb.MovieHome"
 * @ejb:interface remote-class="javatheater.ejb.Movie"
 *
* @ejb:env-entry
 *  name="CounterName"
 *  value="MoviePKCounter"
 *  type="java.lang.String"
 */

public abstract class MovieBean implements EntityBean {
  //… Omitted methods

  /**
   * Home method returning the name of the counter
   * (a String) to generate unique movie primary key values.
   * The actual name is defined as an environment entry of
   * the Movie bean.
   *
   * @ejb:home-method
   */
  public String ejbHomeGetCounterName() {
    try {
      Context initial = new InitialContext();
      return (String)
        initial.lookup("java:comp/env/CounterName");
    }
    catch(NamingException e) {
      throw new EJBException(e);
    }
  }

 // Omitted methods…
}
```

The method is called ejbHomeGetCounterName(). The ejbHome prefix is required by the EJB meta-interface naming rules. In the home interface, however, the method will be exposed as getCounterName(). The method implementation uses something that you haven't seen yet: an EJB environment entry. Conceptually, an EJB environment entry is not much different from an environment entry that you define in your operating system shell, except that it's defined for a specific EJB. An environment entry is just a name/value pair, but with two main differences: 1) the value can be any object, and; 2) the entry is defined in a special JNDI sub-context called the environment naming context (ENC), which is always mapped in all Container to java:comp/env. The value is defined in the deployment descriptor, and can be retrieved with a JNDI lookup. Using EJBDoclet, you can define an environment entry using the ejb:env-entry tag as shown above, and the actual entry will be generated in the deployment descriptor. The benefit of using environment entries is that their value can be modified by the Deployer, thus allowing for more flexible component configuration.

The implementation of the ShowManager session bean (below) should make things clear:

```
//: c18:example07:src:ejb-tier:javatheater:ejb:implementation:ShowManager.java
package javatheater.ejb.implementation;

import javatheater.ejb.*;

import javax.ejb.*;
import javax.naming.*;
import java.rmi.RemoteException;

/**
 * @ejb:bean
 *  type = "Stateless"
 *  name = "ShowManager"
 *  jndi-name = "javatheater/ShowManager"
 *  reentrant = "false"
 *
 * @ejb:home
 *  remote-class="javatheater.ejb.ShowManagerHome"
 *
 * @ejb:interface
 *  remote-class="javatheater.ejb.ShowManager"
 */
public abstract class ShowManagerBean
  implements SessionBean
{
  AutoCounterHome autoCounterHome = null;
  MovieHome movieHome = null;
  String movieCounterName = null;

  /**
 * @ejb:create-method
 */
public void ejbCreate() throws CreateException {
  try {
    Context initial = new InitialContext();

    // Get the home interfaces for the EJB we'll use later
    autoCounterHome = (AutoCounterHome)
      initial.lookup("javatheater/AutoCounter");

    movieHome = (MovieHome)
      initial.lookup("javatheater/Movie");

    movieCounterName = movieHome.getCounterName();
  }
  catch (NamingException e) {
    throw new EJBException(e);
  }
  catch (RemoteException e) {
    throw new EJBException(
      "Error accessing the Movie home interface", e);
  }
}

/**
 * Creates a new movie and returns the new movie's
 * primary key value.
 *
 * @ejb:interface-method
 */
public Integer createMovie(String movieName)
  throws TheaterException
{
  try {
    Integer pk =
      new Integer(getCounter(movieCounterName).getNext());
    movieHome.create(pk, movieName);
    return pk;
  }
  catch (CreateException e) {
    throw new TheaterException(e);
  }
  catch (RemoteException e) {
    throw new TheaterException(e);
  }
}


/**
 * Deletes a movie given its primary key
 *
 * @ejb:interface-method
```

```
  */
 public void deleteMovie(Integer moviePk)
   throws TheaterException
 {
   try {
     Movie movie = movieHome.findByPrimaryKey(moviePk);
     movie.remove();
   }
   catch (FinderException e) {
     throw new TheaterException(e);
   }
   catch (RemoveException e) {
     throw new TheaterException(e);
   }
   catch (RemoteException e) {
     throw new TheaterException(e);
   }
 }

 /**
  * Utility method to get the primary key generator
  * for the Movie bean
  */
 AutoCounter getCounter(String counterName)
   throws RemoteException, CreateException
 {
   AutoCounter counter = null;

   try {
     counter=autoCounterHome.findByPrimaryKey(counterName);
   }
   catch (FinderException e) {}

   if (counter == null)
     counter = autoCounterHome.create(counterName);

   return counter;
 }
}
///:~
```

The ShowManager stateless session bean starts with the definition of a create method with no arguments (the specification prohibits arguments for create methods of stateless session beans), in which it locates via JNDI the home interfaces of the beans that will be used by the other methods. It also calls the getCounterName() home method on the Movie home interface to retrieve the name of the counter that will be used to generate unique primary keys for movies.

The bean also exposes a createMovie() method, that client applications can use to instantiate new Movie objects in the system. You can see how this method hides to the client the complexity of generating a primary key: the only information that the client needs to provide is the movie title. The createMovie() method uses the AutoCounter to get a new unique primary key for the Movie, creates the Movie object and returns the primary key to the client.

If the Movie instantiation fails, a TheaterException is sent to the client. This is a business exception that we defined to indicate an application problem. Its implementation is straightforward, as you see below. Please note that the point here is just to show you that you can easily define and use your own exceptions in enterprise bean method specifications. In a more realistic situation, you may want to attempt some corrective action, or provide more information to the client using different exceptions.

```
//: c18:example07:src:ejb-tier:javatheater:ejb:TheaterException.java
package javatheater.ejb;

public class TheaterException extends Exception {
  public TheaterException() {
  }

  public TheaterException(String message) {
    super(message);
  }

  public TheaterException(String message, Throwable cause) {
    super(message, cause);
  }

  public TheaterException(Throwable cause) {
    super(cause);
  }
}
///:~
```

The ShowManager session bean also defines a deleteMovie() method that clients use to remove movies from the program. It's implementation is simple, but the benefit is that the bean now acts as a façade, hiding complexity and exposing to the client only methods and classes that are specific to the JavaTheater business domain.

The deployment descriptor of the ShowManager session bean is generated by XDoclet. Below you can see the portion that describes the ShowManager session bean:

```
<session >
   <description><![CDATA[No Description.]]></description>
   <ejb-name>ShowManager</ejb-name>
   <home>javatheater.ejb.ShowManagerHome</home>
   <remote>javatheater.ejb.ShowManager</remote>
   <ejb-class>
      javatheater.ejb.implementation.ShowManagerSession
   </ejb-class>
```

```
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
</session>
```

If you compare it to the definition of an entity bean, you'll see that they are quite similar, and that the definition of the session bean is actually simpler. The only two new elements are <session-type>, that in this case states that the ShowManager bean is a stateless session bean, and <transaction-type>, which we use to specify that we want the Container to take care of transaction demarcation. The alternative, specifying the "Bean" value, would mean that we programmatically manage the lifetime of a transaction in the ShowManager's implementation code (we don't cover programmatic transaction demarcation in this chapter).

All the classes, interfaces and XML descriptors for the components developed thus far are bundled in our ejb-jar (javatheater.jar) by the Ant script, and all classes that the client application needs are bundled in the client jar. I'd suggest that you perform a full build of this example using the Ant script, and inspect the contents of the gen directory and of the two generated jar files in the dist directory.

EJB Local Interfaces

So far we have been implementing entity beans and session beans that expose a remote client view. This means that all method invocations on those components will happen via RMI, via proxies and serialization of the method arguments, of the return values and of all exceptions. This process is expensive in terms of performance (an RMI call can be several orders of magnitude slower than an ordinary method call within the same JVM), but you get the benefit of complete location transparency of your component – the client code doesn't know in which process the component instance is located, and doesn't care.

It's important to remember that when the client code accesses an enterprise bean using the bean's remote view, the invocation always happens via RMI, even when the client code and the component instance are located in the same JVM (actually, containers usually optimize an in-JVM RMI call using some specialized proxy, but some proxy is still there and the call is still slower than what it could be, if the client code were able to directly access the component).

In EJB 1.1 enterprise beans can only expose a remote view. However, in the attempt to improve performance, container vendors started introducing in their products some "magic" flag that administrators could flip to disable RMI on enterprise beans. The implication is, though, that the client code had to be co-located in the same JVM as the components.

Eventually, the EJB 2.0 specification picked up this hint coming from the industry, and introduced the concept of an enterprise bean's local view. When an enterprise bean exposes a local view, there is no RMI involved at all, but its home and component interfaces can only be accessed from client code in the same JVM. That’s more efficient, but prevents the use of components in a highly distributed application.

A bean can support both a local and a remote view, in which case the developer has to implement the home and component interfaces in its local and remote versions. Since the local and remote interfaces are referenced by different tags in the deployment descriptor, they can be easily identified by the container and by the bean user. Only session and entity beans can have a remote view; though message-driven beans can only expose local views (in other words, they can never be accessed remotely).

Although the idea of a local and remote view sounds nice at first, there is some debate in the community about the clarity of this design. The biggest issue is the argument passing ambiguity. When an interface is remote (accessed via RMI) all arguments in a method call are passed by value: the argument values are copied into the process space of the remote instance, while the originals remain on the client side. On the other hand, when an interface is local, all arguments are passed by reference (except for primitive data types and references) and the method implementation operates on the originals. The problem with EJB local and remote views is that there’s nothing in the Java interface itself that informs the programmer of which kind of argument passing is being used. The only clue you have is when the interface is obtained using JNDI, but once you get the interface reference there is no way to tell if you are passing arguments by value or by reference. As you can imagine, in a large application, where different parts of the system are developed by different persons and interface references are passed around all the time, not having any type of compile-time or run-time information on the argument passing model is very error prone.

Since the language and the specification don’t help in any way to avoid this problem, you have to come up with your own defense strategy. One could be, of course, a naming convention on your bean’s interfaces, like MovieLocal vs. MovieRemote; another approach could be to decide that entity beans are going to expose only a local view, and session beans expose only a remote view (which fits the Façade design pattern).

It’s now time to implement our first EJB with a local view. Instead of creating a brand new EJB, we’ll modify the implementation of the Movie and AutoCounter entity beans, so that they will expose a local view only, and no longer a remote view. Here is the code for the Movie bean:

```
//: example08/src/ejb-tier/javatheater/ejb/implementation/MovieBean.java

package javatheater.ejb.implementation;

import javax.ejb.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * This is the <code>Movie</code> entity bean implementation.
 *
 * @see javatheater.ejb.MovieHome
 *
 * @ejb:bean
 *  name="Movie"
 *  local-jndi-name="javatheater/Movie"
 *  view-type="local"
 *  type="CMP"
 *  primkey-field="id"
 *  reentrant="False"
 *
 * @ejb:home
 *  local-class="javatheater.ejb.MovieHome"
 *
 * @ejb:interface
 *  local-class="javatheater.ejb.Movie"
 *
 * @ejb:pk class="java.lang.Integer"
 *
 * @ejb:env-entry
 *  name="CounterName"
 *  value="MoviePKCounter"
 *  type="java.lang.String"
 */
public abstract class MovieBean implements EntityBean {
  /**
   * @ejb:create-method
   */
  public Integer ejbCreate(Integer id, String title)
      throws CreateException
  {
    if (id == null)
        throw
          new CreateException("Primary key cannot be null");

    if (id.intValue() == 0)
        throw
          new CreateException("Primary key cannot be zero");

    setId(id);
    setTitle(title);

    return null;
  }

  /**
   * Do nothing
   */
  public void ejbPostCreate(Integer id, String title) {
  }

  /**
   * @ejb:home-method
   */
  public String ejbHomeGetCounterName() {
    try {
      Context initial = new InitialContext();
      return (String)
        initial.lookup("java:comp/env/CounterName");
    }
    catch(NamingException e) {
      throw new EJBException(e);
    }
  }

  /**
   * @ejb:pk-field
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public Integer getId();

  /**
   * This method will not be part of the remote interface.
   */
  abstract public void setId(Integer id);

  /**
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public String getTitle();

  /**
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public void setTitle(String title);
}
///:~
```

The code above shows only minor differences from the previous version with a remote view, but the implications are actually quite significant. Let’s start with the code: the first thing you should notice is the new EJBDoclet parameters in ejb:bean, ejb:home and ejb:interface tags. In ejb:bean we use the local-jndi-name parameter to specify the JNDI name of the local home interface of the Movie bean, and we use view-type=”local” to specify that this bean does, in fact, expose a local view. In the ejb:home and ejb:interface tags, we use the new local-class parameter to specify the name of the home and component interfaces that EJBDoclet will generate for us.

There is no difference in the Java code itself, but that's because EJBDoclet takes care of generating code that's appropriate for a local view. I suggest that, before proceeding with your reading, you run the Ant script to deploy the bean, and if the build is successful you take a look at the code generated by EJBDoclet in the example08/gen subdirectory. Start looking at MovieHome.java, the source file for the Movie bean home interface: you should notice how the interface is no longer derived from javax.ejb.EJBHome, which is an RMI interface, but from java.ejb.EJBLocalHome, which is not an RMI interface. So your home interface is no longer accessible by remote clients, and accordingly, its methods no longer throw a RemoteException. The same is true for the Movie local component interface, defined in Movie.java in the same directory; the only difference is that the interface is no longer derived from javax.ejb.EJBObject, but from javax.ejb.EJBLocalObject instead.

Now that remote clients are no longer able to access a Movie bean instance, changes are required in the client jar and in the client application. The Ant script has been modified and it doesn't put the home and component interfaces for Movie and AutoCounter in the client jar anymore, since they would be useless to the client application and would probably confuse the client programmer. The client application and the test suite have been modified as well, and every reference to the Movie and AutoCounter beans have been removed. You may want to take a look at the code in the rmiclients subdirectory for this example and check out the details.

However, what's most important here is not how the code has changed, but the consequences of the changes. These can be categorized in two main areas: performance and additional container control. Talking about performance, it should be clear by now that calling a method on a local interface is extremely faster than calling the same method on a remote interface, because there is no RMI involved. Plus, since a component exposing just a local view can only be accessed by other components in the same process, the remote clients will have indirectly accessed the local components, probably using some remote bean acting as a front-end. A typical example is using a remote session bean acting as a façade, which then interacts with local entity beans. The session bean implements business logic, and receives method calls representing "macro-functions" of your application; in performing its logic, the session bean may interact with several local beans. In this way, the number of method calls that actually happens across the network is greatly reduced.

The other important implication of having local interfaces on a bean, is that the container can safely assume that no other process will be able to touch that bean's instance, and can therefore do more than it could do on a remote instance. The most notable of these additional EJB features, applicable only with local views, is entity relationships, one of the major improvements in EJB 2.x. We'll talk about relationships in a later section, but first we need to talk about the EJB Query Language (EJB-QL), and before that, bulk accessors and value objects.

Bulk accessors and value objects

From a design standpoint, what we did when we introduced the ShowManager session bean was to put a simple façade that clients can use to create and delete movies from the system in front of our entity beans. It's clear that the ShowManager session bean is intended for managers of the JavaTheater system – normal users are not supposed to modify the listing information.

We are now going to introduce a second session bean which has the sole purpose of returning movie information, but provides no method for changing the movie listing. This bean, which can be safely made available to all users, we'll call ShowListing. However, returning information in a distributed system can use a lot of bandwidth if it's not done properly. What you certainly don't want, for example, is having clients that make one call to retrieve a movie's title, another call to retrieve the director, and so on. Instead, you want the client to retrieve all the information it needs about a movie in one single call across the network.

Such a method is what we call a bulk accessor, a method that accesses a bunch of information in one single shot – the information can be passed into the bean, or returned from the bean. This bulk of information need to be structured somehow, but since there is no construct for structures in Java, bulk accessors always work in concert with value objects. These are simple, plain Java instances that are serializable and that provide public fields for reading and writing values – value objects are also called, depending on the author, data objects or messenger objects.

The code below is an example of a value object designed to transfer movie information, and it's defined in its own package called javatheater.valueobject, that is going to be made available to both the client code and the component code.

```
package javatheater.valueobject;

//: example09/src/ejb-tier/javatheater/valueobject/MovieItem.java

/**
 * A MovieItem is a data obejct used to transfer
 * structured Movie information between the client
 * tier and the ejb tier.
 */
public class MovieItem implements java.io.Serializable {
  public Integer pk;
  public String title;

  public MovieItem(Integer pk, String title) {
    this.pk = pk;
    this.title = title;
  }
}
///:~
```

The code, as you can see, is simple: it’s just a public class with a couple of public fields for the movie information. The class itself is serializable and provides a convenient constructor. This class is going to be used in the implementation of the ShowListing session bean, and here is the code:

```
//: example09/src/ejb-tier/javatheater/ejb/implementation/ShowListingBean.java

package javatheater.ejb.implementation;

import javatheater.ejb.*;
import javatheater.valueobject.MovieItem;
import javax.ejb.*;
import javax.naming.*;

/**
 * @ejb:bean
 *  type = "Stateless"
 *  name = "ShowListing"
 *  jndi-name = "javatheater/ShowListing"
 *  reentrant = "false"
 *
 * @ejb:home
 *  remote-class="javatheater.ejb.ShowListingHome"
 *
 * @ejb:interface
 *  remote-class="javatheater.ejb.ShowListing"
 */
public abstract class ShowListingBean
  implements SessionBean
{
  MovieHome movieHome = null;

  /**
  * @ejb:create-method
  */
  public void ejbCreate() throws CreateException {
    try {
      movieHome = (MovieHome)
        new InitialContext().lookup("javatheater/Movie");
    }
    catch (NamingException e) {
      throw new EJBException(e);
    }
  }

  /**
   * @ejb:interface-method
   */
  public MovieItem getMovie(Integer moviePK) {
    try {
      Movie movie = movieHome.findByPrimaryKey(moviePK);
      return new MovieItem(movie.getId(), movie.getTitle());
    }
    catch (FinderException e) {
      return null;
    }
  }
}
///:~
```

You see from the EJBDoclet tags that this is just a plain stateless session bean with a remote client view. The ejbCreate() method doesn't do anything special, except for getting the home interface of the Movie entity bean, to use it later in the getMovie(...) method. getMovie(...) is implemented to: retrieve a Movie instance using the primary key value that is passed as an argument; construct a MovieItem object populated with the movie information; and return the MovieItem object.

I wanted this code to be simple, but it still demonstrates the use of bulk accessors and values objects to retrieve structured information with one single RMI call. Remember that you can use bulk accessors to pass information into the bean as well. In the next example we'll implement additional bulk accessors designed to retrieved information using different criteria.

Finally, please note that in this example we implemented a bulk accessor on a session bean, but you can do the same on an entity bean. In both cases, you may want to let your bean expose ordinary accessors as well, so that the remote client doesn't have to retrieve a large, structured object from the network in case it is interested only in one field.

Finder Methods and EJB-QL

You have certainly noticed that, so far, there is only one way for the client code to locate an existing instance of an entity bean: calling the findByPrimaryKey(...) method, which implies that the client must know the value of the primary key of a specific movie before it can retrieve it. This is definitely not acceptable: even if the client application kept on his side of the network the list of primary keys for all entities that it created (an ugly approach), how could they retrieve entities created by other clients on the same server? Or, for example, how could you get references to all movies whose title contains the word "Java"?

The answer is custom finder methods. Conceptually, a finder is not much different from performing an SQL query on a relational database: you specify the criteria for locating something (the WHERE clause in an SQL query), where it can be found (the FROM clause) and what you are interested in (the SELECT clause), and you get back a list of records, possibly empty. With EJB you don't retrieve records – in fact, you don’t even know if there is a relational database – but references to the component interfaces of the entities you were looking for.

There is another open question, though: how do you specify the criteria for locating your entities if you don’t even know which kind of data store is being used behind the scenes? Which syntax are you supposed to use to specify your search criteria? In EJB 1.1 different vendors had different ways to provide this kind of information to the container, but in EJB 2.0 the process has been standardized by means of the EJB Query Language.

EJB-QL is a simple language with a syntax similar to SQL, but is completely different in what it operates on. Whereas an SQL query is used to lookup data in relational database tables, the EJB-QL is used to lookup properties on persistent objects (entity beans), and these objects are represented by an abstract schema. The difference between performing searches on tables versus objects is something that you want to keep in mind.

Once again, I have no space here to cover the EJB-QL in detail, but I’m going to give you a simple example that should help clarify the concept. For additional information, I refer you to the documents at the end of this chapter. Let’s start taking a look at the code of the Movie entity bean, and more specifically to the new EJBDoclet tags (the Java code hasn’t changed):

```
//: example10/src/ejb-tier/javatheater/ejb/implementation/MovieBean.java

package javatheater.ejb.implementation;

import javax.ejb.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * This is the <code>Movie</code> entity bean implementation.
 *
 * @see javatheater.ejb.MovieHome
 *
 * @ejb:bean
 *  name="Movie"
 *  local-jndi-name="javatheater/Movie"
 *  view-type="local"
 *  type="CMP"
 *  primkey-field="id"
 *  reentrant="False"
 *
 * @ejb:pk class="java.lang.Integer"
 *
 * @ejb:home
 *  local-class="javatheater.ejb.MovieHome"
 *
 * @ejb:interface
 *  local-class="javatheater.ejb.Movie"
 *
 * @ejb:finder
 *  signature="java.util.Collection findByTitle(java.lang.String title)"
 *  query="SELECT OBJECT(m) FROM Movie m WHERE m.title=?1"
 *
 * @ejb:finder
 *  signature="java.util.Collection findAll()"
 *  query="SELECT OBJECT(m) FROM Movie m"
 *
 * @ejb:env-entry
 *  name="CounterName"
 *  value="MoviePKCounter"
 *  type="java.lang.String"
 */
public abstract class MovieBean implements EntityBean {
  /**
   * Validates the primary key and initializes the bean properties.
   *
   * @ejb:create-method
   */
  public Integer ejbCreate(Integer id, String title)
      throws CreateException
  {
    if (id == null)
        throw new CreateException("Primary key cannot be null");
    if (id.intValue() == 0)
        throw new CreateException("Primary key cannot be zero");

    setId(id);
    setTitle(title);

    return null;
  }

  /**
   * Do nothing
   */
  public void ejbPostCreate(Integer id, String title) {
  }

  /**
   * Home method returning the name of the counter (a String)
   * to generate unique movie primary key values. The actual
   * name is defined as an environment entry of the Movie bean.
   *
   * @ejb:home-method
   */
  public String ejbHomeGetCounterName() {
    try {
      Context initial = new InitialContext();
      return (String) initial.lookup("java:comp/env/CounterName");
    }
    catch(NamingException e) {
      throw new EJBException(e);
    }
  }

  /**
   * Returns the Movie id, which is also the Movie primary key.
   *
   * @ejb:pk-field
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public Integer getId();

  /**
   * This method will not be part of the remote interface,
   * but ejbdoclet will still generate its implementation.
   * It's called from ejbCreate().
   */
  abstract public void setId(Integer id);

  /**
   * Returns the Movie title.
   *
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public String getTitle();

  /**
   * Sets the Movie title.
   *
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public void setTitle(String title);
}
///:~
```

The two new ejb:finder tags are what we use in EJBDoclet to generate custom finder methods. A custom finder is a method in an entity bean’s home interface that clients can use to locate entities. The only finder we’ve seen so far is findByPrimaryKey(...), but the EJB specification allows for additional finders defined by the bean provider. Only findByPrimaryKey(...) returns a single bean reference; all other finders return a Java Collection – this makes sense, because only the primary key can uniquely identify one entity.

The reason to implement multiple finders is to let the clients locate entities using different criteria. In the case of the Movie bean, for example, the user may want to find movies by title, by director, by cast members, by genre and so on; of all the possible criteria, the most unlikely for the client to use is the bean's primary key, the only finder we had so far.

A finder method is designed to retrieve information from the persistent storage. Since the EJB specification allows for different types of data store, the implementation of the finders can vary greatly. If you are implementing a finder for a BMP entity bean, you are responsible for implementing all the details to access the database and the information it contains in the proper way (which can be considerably complex, depending on the nature of your entity beans and their relationships with each other). On the other hand, if you are implementing a CMP entity bean, you don't code for one specific database (remember that CMP is there to promote portability) but you still need to specify your search criteria.

The good news is that the container generates the finder method implementations for you. All you have to do is to tell the container where it should search, and for what. Which brings us to the topics of the abstract entity schema and the EJB-QL, respectively. The abstract schema is a simple concept: your entity beans are persistent object, right? That means that their state is going to be saved on some persistent storage, and the state is going to be organized in some format (tables and records, for example). However, the EJB platform is designed from the ground up to be independent from the data store. So we need, as always when things get too complex, an additional level of abstraction that will let us associate entity bean state with persistent information, and this level of abstraction is called the abstract schema. The abstract schema, in turn, defines the information that can be queried using the EJB-QL.

In practice, every CMP entity bean has an associate schema name that you can use in EJB-QL expressions, and the bean exposes information via its properties, as defined in the JavaBeans specification. Before you keep reading, you'll need to build example10, because we'll have to look at the code generated by EJBDoclet – use the Ant script as always to build the example.

First, let's look at the home interface for the Movie entity bean, which you'll find in the gen/javatheater/ejb/MovieHome.java in the example10 subdirectory. You'll immediately notice that the two ejb:finder tags, in the bean's implementation source, caused EJBDoclet to generate two new finder method definitions on the Movie home interface.

Now let's take a look at the deployment descriptor, in the gen/META-INF subdirectory. If you look in the <entity> element that defines the Movie entity bean, you'll see this line:

```
<abstract-schema-name>Movie</abstract-schema-name>
```

This element associates to the Movie entity bean a name that you can use in EJB-QL queries (more or less as if it, the bean, were an SQL table). By default, EJBDoclet uses the bean's name as the abstract schema name, but you can specify a different one, if you like. The fields that you can query on the Movie bean are the persistent properties exposed by the Movie bean (and the properties that take part in a relationship, but we'll cover that later).

Looking down in the definition of the Movie bean, in the deployment descriptor, you'll also find this code generated by EJBDoclet:

```
<query>
  <query-method>
    <method-name>findByTitle</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    <![CDATA[SELECT OBJECT(m) FROM Movie m WHERE m.title=?1]]>
  </ejb-ql>
</query>
```

This defines the query for the findByTitle(...) finder, whose signature we specified in the ejb:finder tag, along with the query arguments and the query string. The query string itself is:

```
SELECT OBJECT(m) FROM Movie m WHERE m.title=?1
```

That's not much different from an SQL query string, and it means that you want to retrieve ("SELECT") all bean instances ("OBJECT(m)") from an abstract schema of type "Movie", named "m" in this query; and that you want only those instances of "m" ("WHERE") whose "title" property is equal to "?1", which is the first argument passed into the finder method (a String in this case).

The other finder in the Movie bean, findAll(), has a very similar query but with no WHERE clause, which means that you want to retrieve all instances without any filtering.

This is almost all the container needs to know to find Movie instances according to the criteria you expressed in the query. The missing part is how to map abstract schemas to data in a specific, concrete database. At this point the specification stops, and the mapping is left to the container vendor. In other words, different containers offer different ways and different tools to map entity beans to the database – a job for the Bean Deployer. Many containers come with some kind of default mapping, and that is also the case with JBoss; that's why we don't have to worry, in these examples, about the mapping: we are using the default database and the default mapping that comes with the standard JBoss installation. If you want to do something more customized to your needs, which is certainly the case in the development of a real-world application, you will have to refer to your container's documentation.

However, being able to implement a bean and specify the search criteria in a totally database-independent way is quite an achievement. The benefit is not just that you don't have to write any database-specific code, but that you have portability, plus the fact that the container may optimize your query better than you could, because of its internal knowledge of and integration with a specific database.

Now that we have a couple more finders, we can use them to implement additional functionalities on the façade session beans, especially since our entity beans are no longer accessible directly by the clients. The following code in the ShowListing session bean implementation has two additional methods that clients can use to retrieve a movie by its title, or just all of them:

```
//: example10/src/ejb-tier/javatheater/ejb/implementation/ShowListingBean.java

package javatheater.ejb.implementation;

import javatheater.ejb.*;
import javatheater.valueobject.MovieItem;
import javax.ejb.*;
import javax.naming.*;
import java.util.Collection;
import java.util.Iterator;

/**
 * @ejb:bean
 *  type = "Stateless"
 *  name = "ShowListing"
 *  jndi-name = "javatheater/ShowListing"
 *  reentrant = "false"
 *
 * @ejb:home
 *  remote-class="javatheater.ejb.ShowListingHome"
 *
 * @ejb:interface
 *  remote-class="javatheater.ejb.ShowListing"
 */
public abstract class ShowListingBean
  implements SessionBean
{
  MovieHome movieHome = null;

  /**
  * @ejb:create-method
  */
  public void ejbCreate() throws CreateException {
    try {
      movieHome = (MovieHome)
        new InitialContext().lookup("javatheater/Movie");
    }
    catch (NamingException e) {
```

```
    throw new EJBException(e);
  }
}

/**
 * @ejb:interface-method
 */
public MovieItem getMovie(Integer moviePK) {
  try {
    Movie movie = movieHome.findByPrimaryKey(moviePK);
    return new MovieItem(movie.getId(), movie.getTitle());
  }
  catch (FinderException e) {
    return null;
  }
}

/**
 * @ejb:interface-method
 */
public MovieItem [] getMovie(String title) {
  try {
    Collection foundMovies = movieHome.findByTitle(title);
    MovieItem [] items = new MovieItem[foundMovies.size()];

    int index = 0;
    Iterator movies = foundMovies.iterator();
    while (movies.hasNext()) {
      Movie movie = (Movie) movies.next();
      items[index++] =
        new MovieItem(movie.getId(), movie.getTitle());
    }

    return items;
  }
  catch (FinderException e) {
    throw new EJBException(e);
  }
}

/**
 * @ejb:interface-method
 */
public MovieItem [] getMovies() {
  try {
    Collection foundMovies = movieHome.findAll();
    MovieItem [] items = new MovieItem[foundMovies.size()];

    int index = 0;
    Iterator movies = foundMovies.iterator();
    while (movies.hasNext()) {
      Movie movie = (Movie) movies.next();
```

```
        items[index++] =
          new MovieItem(movie.getId(), movie.getTitle());
      }

      return items;
    }
    catch (FinderException e) {
      throw new EJBException(e);
    }
  }
}
///:~
```

There would be much more to say about finders and the EJB-QL (finder permissions, for example, or how to perform more sophisticated queries) but this is an introduction to EJBs, we can't really go into more detail. However, you've been exposed to the essence of what are the custom finders and the EJB-QL.

As a last note, a topic that is closely related to finders and EJB-QL is select methods. A select method is similar to a finder, and it's defined in a similar way using a query. However, there are two major differences between finder and select methods: 1) finders always return component interface references to entity beans, whereas select methods can return any type of persistent property exposed by the bean, and; 2) select methods are never available to clients, regardless of which view the client implements. So, for example, you could have a select method that returns all movie titles (Strings), whereas a finder would return movies. Select methods, not being available to the clients, can only be used to implement part of the internal logic of your entity bean.

Entity Relationships

The last topic we are going to cover is possibly the major improvement that appeared with EJB 2.0, along with local interfaces. It's a feature that allows the bean developer to establish relationships between entity beans by just describing them in the deployment descriptor. Let me explain.

So far we have been implementing entity beans, like Movie and AutoCounter, that are completely self-contained. That is, they store internally all the information they need to be a consistent block of information. A Movie entity bean, for example, contains all information about a movie. However, it's unlikely that a real-world application will employ only such simple objects. Even in our JavaTheater application, as soon as we start introducing a richer entity like a show, things become immediately more complex.

A show in fact is composed at least with a movie, a show time and a theater. Starting from this description, it's safe to say that a show can have only one movie, but the same movie can be in many shows – a classical one-to-many relationship. From an object-oriented standpoint, implementing this relationships is trivial: a show object contains a reference to a movie object, and a movie object contains a collection of references to show objects.

The problem though, is that our objects are also persistent entities, so the references representing the relationship must be transformed into something that can be stored into the database, and later retrieved and transformed back into references to the entities they originally represented. This process is more complex than what it looks like and EJB 1.1 provides little help to the developer; but EJB 2.0 makes up for this deficiency, and introduces the concept of container-managed relationships. Thanks to this feature the developer can specify in the deployment descriptor how two entity beans (Movie and Show, in our case) are related, which fields participate in the relationship, and which is their cardinality – cardinality is also known as multiplicity. It indicates how many entities there are on each side of the relationship.

As always, we'll use a practical example to clarify the concept, but before you build and deploy the code, I recommend that you completely clean up the database that you have been using in the previous examples. The simplest way to do this is just to delete all files that store your data (and don't worry, they will be regenerated anew when you restart JBoss). Shut down JBoss and then, in your JBoss installation directory, locate the subdirectory server/default/db/hypersonic (please note that if you are using a version of JBoss other than 3.04 the directory structure could be slightly different). This directory contains all the HypersonicSQL data files, which are all named default with different extensions. Just delete them all (or move them to a different directory if you want to be conservative) and restart JBoss. Check that the files have been regenerated, and then proceed with the exercise.

What we are going to do is to implement a new CMP entity bean that represents a show. A show is going to be associated to one movie, and a movie is going to be associated to multiple shows. Below you can see the implementation of a new entity bean in our application, the Show bean, which you'll find in the example11 directory. In our design a Show is composed with a Movie, a show time (a String) and the number of available seats (an int). I have intentionally left the theater out of the picture for simplicity.

```
//: example11/src/ejb-tier/javatheater/ejb/implementation/ShowBean.java
package javatheater.ejb.implementation;

import javax.ejb.*;
import javax.naming.*;
import javatheater.ejb.*;

/**
 * This is the <code>Show</code> entity bean implementation.
 *
 * <p>A Show has an id (which is also its primary key),
 * a movie, a showtime and a number of available seats.
 *
 * @see javatheater.ejb.ShowHome
 * @see javatheater.ejb.Show
 *
 * @ejb:bean
 *  name="Show"
 *  local-jndi-name="javatheater/Show"
 *  view-type="local"
 *  type="CMP"
 *  primkey-field="id"
 *  reentrant="False"
 *
 * @ejb:pk class="java.lang.Integer"
 *
 * @ejb:home
 *   local-class="javatheater.ejb.ShowHome"
 *
 * @ejb:interface
 *   local-class="javatheater.ejb.Show"
 *
 * @ejb:finder
 *  signature="Collection findByMovie(java.lang.Integer moviePK)"
 *  query="SELECT OBJECT(s) FROM Show s WHERE s.movie.id=?1"
 *
 * @ejb:finder
 *  signature="java.util.Collection findAll()"
 *  query="SELECT OBJECT(s) FROM Show s"
 *
 * @ejb:env-entry
 *  name="CounterName"
 *  value="ShowPKCounter"
 *  type="java.lang.String"
 */
public abstract class ShowBean implements EntityBean {
 /**
  * Validates the primary key and initializes
  * the bean properties.
  *
  * @ejb:create-method
  */
  public Integer ejbCreate(
    Integer id, Movie movie,
    String showTime, int availableSeats
  )
    throws CreateException
  {
    if (id == null)
      throw new CreateException("Id can not be null");

    if (id.intValue() == 0)
      throw new CreateException("Id can not be zero");
    if (movie == null)
      throw new CreateException("A movie must be provided");

    setId(id);
    setShowTime(showTime);
    setAvailableSeats(availableSeats);

   // Note that relationship fields cannot be modified
   // or set in ejbCreate(). So we initialize the
   // movie field in ejbPostCreate() (below).
    return null;
  }

  public void ejbPostCreate(
    Integer id, Movie movie,
    String showTime, int availableSeats
  )
  {
    setMovie(movie);
  }

  /**
   * Home method returning the name of the counter
   * (a String) to generate unique Show primary key values.
   * The actual name is defined as an environment entry of
   * the Show bean.
   *
   * @ejb:home-method
   */
  public String ejbHomeGetCounterName() {
    try {
      Context initial = new InitialContext();
      return (String)
        initial.lookup("java:comp/env/CounterName");
    }
    catch(NamingException e) {
      throw new EJBException(e);
    }
  }

  /**
   * Returns the Show id, which is also the Show's
   * primary key.
   *
   * @ejb:pk-field
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public Integer getId();

  /**
   * @ejb:persistent-field
   */
  abstract public void setId(Integer id);

  /**
   * Sets the movie for this show.
   *
   * Note that this is no longer a CMP field,
   * but a Container Managed Releationship (CMR) field
   * (see generated deployment descriptor).
   *
   * @ejb:interface-method
   *
   * @ejb:relation
   *   name="Show-Movie"
   *   role-name="Show-has-a-Movie"
   */
   abstract public void setMovie(Movie movie);

  /**
   * Gets the movie for this show.
   * Note that the movie property is a CMR field.
   *
   * @see #setMovie(Movie)
   * @ejb:interface-method
   */
   abstract public Movie getMovie();

  /**
   * Returns the Show time.
   *
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public String getShowTime();

  /**
   * Sets the Show time.
   *
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public void setShowTime(String showTime);

  /**
   * Returns the Show's available seats.
   *
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public int getAvailableSeats();

  /**
```

```
   * Sets the Show's available seats.
   *
   * @ejb:persistent-field
   * @ejb:interface-method
   */
  abstract public void setAvailableSeats(
    int availableSeats
  );
} ///:~
```

There is nothing new or strange in the first part of this code. It's just a CMP entity bean with a local view and a couple of finder methods. The ejbCreate(...) method receives all the information that is necessary to instantiate a new show: a primary key, a Movie reference, the show time and the number of available seats. You'd expect to store all this information into the Show bean properties, but here's something new: the Movie reference is not set in ejbCreate(..), but in ejbPostCreate(...) instead. This is because the movie property in the Show bean represents a persistent relation, so it's dependent on the valid primary keys of both parties involved, the show and the movie; and as you know, you can't rely on the bean's primary key during ejbCreate(...). The rest of the code simply defines the CMP properties of the Show bean, except for the movie property, which is a something new to us.

The movie property in the Show bean is not a CMP field, but a Container Managed Relationship (CMR) field, and it's treated differently by the container. It requires a different description in the deployment descriptor than the CMP fields, so EJBDoclet comes with a tag specifically intended for describing relationships (ejb:relation).

Before explaining what the tag and the information in the deployment descriptor mean, we need to understand how entity relationships are modeled in EJB. First of all, an entity relationship is made up, of course, by two entity beans; second, only certain fields in each bean participate in the relation; third, each side has a cardinality (for example, a movie has many shows, but a show has just one movie). All this information must end up in the deployment descriptor, but that's not all: since there are usually several relationships that the container has to manage, they need to be named, for the container but also (especially) for the Bean Deployer.

When you use EJBDoclet, all you have to do is to specify that a field represent a relationship that the container will have to manage, and you have to do this on both parts involved. Depending on the type of properties that describe the relationship (single entity reference or collection) EJBDoclet will generate the information in the deployment descriptor with the right cardinality. However, you will need to supply the relation name and the role name. This is exactly what the following snipped of code from the example above is doing (I just removed the prose from the comments):

```
  /**
   * @ejb:interface-method
   *
   * @ejb:relation
```

```
 *   name="Show-Movie"
 *   role-name="Show-has-a-Movie"
 */
 abstract public void setMovie(Movie movie);
```

Here we are saying that you want the movie property of the Show entity bean (defined by the name of setter method) to be part of a container-managed relationship. You are also saying that this relation is called “Show-Movie” and that the name of this role in the relation is “Show-has-a-Movie”. This can be used, for example, to associate security permissions with the role. The cardinality is determined by the type of the property, which EJBDoclet can get parsing the setter method signature. In this case the type is “reference to one Movie”, so the EJBDoclet knows that on this side of the relationship the cardinality is “one”.

A relationship needs to have two parties to exist, and the other player is, of course, the Movie bean. Below you can see the new code that was added to the Movie entity bean in example11:

```
/**
 * @ejb:interface-method
 *
 * @ejb:relation
 *   name="Show-Movie"
 *   role-name="Movie-has-many-Shows"
 */
public abstract Collection getShows();

/**
 * @ejb:interface-method
 */
public abstract void setShows(Collection shows);
```

Here we are defining a new property in the Movie bean called shows. This property is a collection, since one movie can have many shows. The property participates in the same relationship (“Show-Movie”) that we have described in the Show bean, but here we are defining a different role (“Movie-has-many-Shows”).

Based on the relationship name that you have specified on both beans, and the type of properties involved in the relationship (single object vs. collection), EJBDoclet generates the following information in the deployment descriptor:

```
<relationships >
  <ejb-relation >
    <ejb-relation-name>
      Show-Movie
    </ejb-relation-name>

    <ejb-relationship-role >
      <ejb-relationship-role-name>
        Movie-has-many-Shows
      </ejb-relationship-role-name>

      <multiplicity>One</multiplicity>

      <relationship-role-source >
        <ejb-name>Movie</ejb-name>
      </relationship-role-source>

      <cmr-field >
        <cmr-field-name>
          shows
        </cmr-field-name>
        <cmr-field-type>
          java.util.Collection
        </cmr-field-type>
      </cmr-field>
    </ejb-relationship-role>

    <ejb-relationship-role >
      <ejb-relationship-role-name>
        Show-has-a-Movie
      </ejb-relationship-role-name>

      <multiplicity>Many</multiplicity>

      <relationship-role-source >
        <ejb-name>Show</ejb-name>
      </relationship-role-source>

      <cmr-field >
        <cmr-field-name>
          movie
        </cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
```

It's a brand new section in the deployment descriptor that is outside the familiar <enterprise-beans> element you've seen so far, and it serves the sole purpose of listing all the container-managed relationships in your application. In our case there is only one relationship, and its name is "Show-Movie". The two roles in the relationship are described by their own respective elements, along with cardinality (multiplicity) and the name and type of the fields involved.

Using this information, the container will take care of making the relationship persistent in your database, and it will also generate the implementation of the setters and getters for the CMR properties of your beans. How the relationship is actually mapped to the database depends on the combination of a number of factors: the container implementation; the database that you are using; the integration between the two; and the mapping that you provide (also container-specific). In our case, using JBoss with its default database (HypersonicSQL, a simple relational database) and the default mapping, the container generates one additional field in the table associated with the Show entity bean, and this field stores the primary key of the movie associated with that show. There is no additional field in the other tables, and no association table in this case, since the relationship between movie and show is simple enough to be modeled this way. When you call getMovie() on a Show instance, the generated implementation of the method performs an SQL query on the Show table, which looks for the record with the same primary key as the Show instance that you are using, and returns the primary key of the Movie; the Movie primary key is then used to create a Movie instance in memory, which is initialized with the movie information, and whose reference is finally returned to the client. If you call the getShows() method on a Movie instance, a different query is performed, which retrieves all the primary keys of Shows which contain the same Movie primary key as the instance on which you called the method.

All this complexity is of course hidden from you by the container; plus, using EJBDoclet, you don't even need to worry about the contents of the deployment descriptor. All you have to do is specify the abstract method on both sides of the relationship (the two beans involved) and use the appropriate EJBDoclet tags. At this point, we can use the new methods we have on the Show and Movie beans, and verify that the relationship works as expected. Since the two beans don't have a remote view, and cannot be used by the clients directly, I've added a few methods to our façade beans. I've also introduced a new value object, ShowItem, that we use to transfer structured show information across tiers. Here is the new ShowItem class:

```
//: example11/src/ejb-tier/javatheater/valueobject/ShowItem.java
package javatheater.valueobject;
import java.io.Serializable;

/**
 * A ShowItem is a data obejct used to transfer information
 * between the client tier and the ejb tier.
 */
public class ShowItem implements Serializable {
  public Integer pk;
  public MovieItem movie;
  public String showTime;
  public int availableSeats;

  public ShowItem(
    Integer pk, MovieItem movie,
    String showTime, int availableSeats
  )
  {
    this.pk = pk;
    this.movie = movie;
    this.showTime = showTime;
    this.availableSeats = availableSeats;
  }

  public String toString() {
    return
      "[" + pk.toString() + "] " + movie.title +
      ", " + showTime + ", " + availableSeats;
  }
} ///:~
```

Conceptually, it's not very much different from the MovieItem class. The only notable difference is that one of the public fields is a reference to a MovieItem object, instead of being some simple data type or class. What you are seeing here is that, when there are relationships between beans, the value objects can reflect the relationship too, and can be structured as a web of objects, rather than a bunch of simple data types. This is sometimes referred to as "simple aggregation".

Below is the implementation of the ShowManager stateless session bean, that managers can use to modify the contents of the catalog. It has been modified from the previous example to support the new Show entity bean.

```
//: example11/src/ejb-tier/javatheater/ejb/implementation/ShowManagerBean.java
package javatheater.ejb.implementation;

import javatheater.ejb.*;

import javax.ejb.*;
import javax.naming.*;
import java.rmi.RemoteException;

/**
 * @ejb:bean
 *  type = "Stateless"
 *  name = "ShowManager"
 *  jndi-name = "javatheater/ShowManager"
 *  reentrant = "false"
 *
 * @ejb:home
 *  remote-class="javatheater.ejb.ShowManagerHome"
 *
 * @ejb:interface
 *  remote-class="javatheater.ejb.ShowManager"
 */
public abstract class ShowManagerBean
  implements SessionBean
{
  AutoCounterHome autoCounterHome = null;
  MovieHome movieHome = null;
  ShowHome showHome = null;
  String movieCounterName = null;
  String showCounterName = null;
```

```
/**
 * @ejb:create-method
 */
public void ejbCreate() throws CreateException {
  try {
    Context initial = new InitialContext();

    // Get the home interfaces for the EJB we'll use later
    autoCounterHome = (AutoCounterHome)
      initial.lookup("javatheater/AutoCounter");

    movieHome = (MovieHome)
      initial.lookup("javatheater/Movie");

    showHome = (ShowHome)
      initial.lookup("javatheater/Show");

    movieCounterName = movieHome.getCounterName();
    showCounterName = showHome.getCounterName();
  }
  catch (NamingException e) {
    throw new EJBException(e);
  }
}

/**
 * Creates a new movie and returns the new movie's
 * primary key value.
 *
 * @ejb:interface-method
 */
public Integer createMovie(String movieName)
  throws TheaterException
{
  try {
    Integer pk =
      new Integer(getCounter(movieCounterName).getNext());
    movieHome.create(pk, movieName);
    return pk;
  }
  catch (CreateException e) {
    throw new TheaterException(e);
  }
  catch (RemoteException e) {
    throw new EJBException(e);
  }
}

/**
 * @ejb:interface-method
 */
```

```
public Integer createShow(
  Integer moviePK, String showtime, int seats
)
throws TheaterException
{
  try {
    Movie movie = movieHome.findByPrimaryKey(moviePK);
    Integer showPK =
      new Integer(getCounter(showCounterName).getNext());
    showHome.create(showPK, movie, showtime, seats);
    return showPK;
  }
  catch (FinderException e) {
    throw new TheaterException(e);
  }
  catch (CreateException e) {
    throw new TheaterException(e);
  }
  catch (RemoteException e) {
    throw new EJBException(e);
  }
}

/**
 * Deletes a movie given its primary key
 *
 * @ejb:interface-method
 */
public void deleteMovie(Integer moviePk)
  throws TheaterException
{
  try {
    Movie movie = movieHome.findByPrimaryKey(moviePk);
    if (movie.getShows().size() != 0)
      throw new TheaterException(
        "Can't delete movie: it appears in one or more shows");
    movie.remove();
  }
  catch (FinderException e) {
    throw new TheaterException(e);
  }
  catch (RemoveException e) {
    throw new TheaterException(e);
  }
}

/**
 * Deletes a show given its primary key
 *
 * @ejb:interface-method
 */
public void deleteShow(Integer showPK)
```

```
    throws TheaterException
  {
    try {
      Show show = showHome.findByPrimaryKey(showPK);
      show.remove();
    }
    catch (FinderException e) {
      throw new TheaterException(e);
    }
    catch (RemoveException e) {
      throw new TheaterException(e);
    }
  }

  /**
   * Utility method to get the primary key generator
   * for the Movie bean
   */
  AutoCounter getCounter(String counterName)
    throws RemoteException, CreateException
  {
    AutoCounter counter = null;

    try {
      counter =
        autoCounterHome.findByPrimaryKey(counterName);
    }
    catch (FinderException e) {}

    if (counter == null)
      counter = autoCounterHome.create(counterName);

    return counter;
  }
}
///:~
```

This new implementation contains two methods to create and delete shows from the system. The implementation of these methods is straightforward, and it should constitute no surprise for you. What's more interesting, because of the issues it brings up, are the modifications applied to the implementation of the pre-existing deleteMovie(...) method. As you can see from the code, this method has been modified to check that the movie you want to delete doesn't show up in any show. That makes sense: leaving a show and removing its movie wouldn't make the audience happy. This is a classical example of violation of the referential integrity of the system. In other words, being able to delete a movie entity that is part of a relationship with a show would lead to inconsistent data in the system.

What you've seen in deleteMovie(...) is one possible solution to this problem: implement the referential integrity rules of your system in the façade beans. This is not the only possible approach though, especially if referential integrity is enforced by the database, as many DB administrators like to see. This is especially the case when you are putting your new entity beans on top of an existing database, which could also make use of stored procedures to implement certain business functions. In that case, your entity and session beans can take advantage of the logic in the database, and no Java code would be required in the beans to enforce the referential integrity, except for the code that intercepts exceptions generated by the database when an operation violates the rules.

The last modification in the EJBs of example11 is the addition to the ShowListing session bean of a method to retrieve all the available shows. Here is the code:

```
//: example11/src/ejb-tier/javatheater/ejb/implementation/ShowListingBean.java
package javatheater.ejb.implementation;

import javatheater.ejb.*;
import javatheater.valueobject.*;
import javax.ejb.*;
import javax.naming.*;
import java.util.Collection;
import java.util.Iterator;

/**
 * @ejb:bean
 *  type = "Stateless"
 *  name = "ShowListing"
 *  jndi-name = "javatheater/ShowListing"
 *  reentrant = "false"
 *
 * @ejb:home
 *  remote-class="javatheater.ejb.ShowListingHome"
 *
 * @ejb:interface
 *  remote-class="javatheater.ejb.ShowListing"
 */
public abstract class ShowListingBean
  implements SessionBean
{
  MovieHome movieHome = null;
  ShowHome showHome = null;

  /**
  * @ejb:create-method
  */
  public void ejbCreate() throws CreateException {
    try {
      Context initialContext = new InitialContext();

      movieHome = (MovieHome)
        initialContext.lookup("javatheater/Movie");
      showHome = (ShowHome)
        initialContext.lookup("javatheater/Show");
    }
    catch (NamingException e) {
      throw new EJBException(e);
    }
  }

  /**
   * @ejb:interface-method
   */
  public MovieItem getMovie(Integer moviePK) {
    try {
      Movie movie = movieHome.findByPrimaryKey(moviePK);
      return new MovieItem(movie.getId(), movie.getTitle());
    }
    catch (FinderException e) {
      return null;
    }
  }

  /**
   * @ejb:interface-method
   */
  public MovieItem [] getMovie(String title) {
    try {
      Collection foundMovies = movieHome.findByTitle(title);
      MovieItem [] items =
        new MovieItem[foundMovies.size()];

      int index = 0;
      Iterator movies = foundMovies.iterator();
      while (movies.hasNext()) {
        Movie movie = (Movie) movies.next();
        items[index++] =
          new MovieItem(movie.getId(), movie.getTitle());
      }

      return items;
    }
    catch (FinderException e) {
      throw new EJBException(e);
    }
  }

  /**
   * @ejb:interface-method
   */
  public MovieItem [] getMovies() {
    try {
      Collection foundMovies = movieHome.findAll();
```

```
      MovieItem [] items =
        new MovieItem[foundMovies.size()];

      int index = 0;
      Iterator movies = foundMovies.iterator();
      while (movies.hasNext()) {
        Movie movie = (Movie) movies.next();
        items[index++] =
          new MovieItem(movie.getId(), movie.getTitle());
      }

      return items;
    }
    catch (FinderException e) {
      throw new EJBException(e);
    }
  }

  /**
   * @ejb:interface-method
   */
  public ShowItem [] getShows() {
    try {
      Collection foundShows = showHome.findAll();
      ShowItem [] items = new ShowItem[foundShows.size()];

      int index = 0;
      Iterator shows = foundShows.iterator();
      while (shows.hasNext()) {
        Show show = (Show) shows.next();
        Movie movie = show.getMovie();
        items[index++] =
          new ShowItem(
            show.getId(),
            new MovieItem(movie.getId(), movie.getTitle()),
            show.getShowTime(),
            show.getAvailableSeats()
          );
      }

      return items;
    }
    catch (FinderException e) {
      throw new EJBException(e);
    }
  }
}
///:~
```

The method uses the findAll(...) finder on the Show home interface to retrieve all shows, and for each show it creates a ShowItem object that is stored in the returned array. Also, since a ShowItem contains a MovieItem, for each show the method retrieves the associated movie using the CMR movie property (calling the getMovie() method) and creates a MovieItem using that information.

There are no more modifications to the code of the EJBs in example11. However, the client application has been completely rewritten, and it now comes with a graphical user interface that you can use to create and delete movies and shows from the system. Although the features of the client application are minimal, you may want to play with it, or maybe use it as a starting point to implement additional features, like searching for shows for a specific movie, or for shows at a certain show time. That would also require that you add more features to the façade session beans and possibly to the entity beans, but it would make for a good exercise.

Summary

What you had here was a bird's-eye view of Enterprise JavaBeans, and much had to be left out of the picture. We didn't cover some major topics like security, transactions and message-driven beans. We also didn't cover stateful session beans and bean-managed persistence. However, you now have the conceptual tools to understand what EJBs are all about; and you also have some technical tools and skills to keep exploring on your own.

Enterprise JavaBeans are a major, rapidly-evolving, server-side Java technology that gives its best when used in conjunction with other J2EE technologies – Servlets and JavaServer Pages (JSP) to interface your EJB business model with a web tier, and Web Services to expose your business objects via HTTP. When it comes to building a system employing these technologies though, guidance is necessary and what you need is J2EE design patterns, a fascinating topic that I strongly recommend you investigate.

If you want to proceed on the road to mastering Enterprise JavaBeans, the documents listed below will be good companions.

Resources

Here is some recommended reading , for more comprehensive information on Enterprise JavaBeans and related subjects. Where documents are available online, a URL is given.

- Enterprise Java Beans ™ Specification
  (Sun Microsystems, Inc.)
  http://java.sun.com/j2ee

- The J2EE ™ Tutorial
  (Sun Microsystems, Inc.)
  http://java.sun.com/j2ee

- Java ™ Blueprints
  (Sun Microsystems, Inc.)
  http://java.sun.com/blueprints

  - Richard Monson-Haefel, Enterprise Java Beans 3rd ed.
    (O'Reilly and Associates, Inc., 2001)

  - Deepak Alur, John Crupi, Dan Malks, Core J2EE Patterns
    (Sun Microsystems Press, 2001)

  - Floyd Marinescu, EJB ™ Design Patterns
    (John Wiley & Sons, Inc., 2002)
    http://theserverside.com

XML

Java gives you portable code and Exensible Markup Language (XML) gives you portable data[8].

XML is a standard adopted by the World Wide Web Consortium (W3C) to complement HTML for data exchange on the Web. In this chapter, we will describe XML and the tools and APIs to access XML from Java.

Most readers will have seen HTML, the language for describing Web pages. Unlike many we will not criticize HTML; it has been wildly successful by any measure. HTML was not the first tag language but it is certainly the most commonly used, most successful, and most widely understood.

If HTML is so successful, why do we need XML, what's so great about it and why will it be more successful? There are several reasons.

Firstly, XML makes data exchange easy. In XML, the data and the descriptions of how it should be structured (the markup) are stored as text that you can easily read and edit. Secondly, the XML markup can be customized and standardized. This brings extraordinary power to XML.   HTML is a markup language designed to describe web pages, but its major weakness is that it blends data and presentation When a group of users with common interestsagree on the tags for a particular markup language built with XML , what they are doing is creating a customized markup language. Hundreds of such languages are being standardized including:

- Bank Internet Payment System (BIPS)
    - Financial Information eXchange protocol (FIX)
    - Telecommunications Interchange Markup (TIM)
    - Mathematics Markup Language (MathML)

In fact, you can add extensions to markup languages created by someone else. Extensible Hypertext Markup Language (XHTML) is an XML version of HTML that allows elements to be added to pages a browser displays as normal HTML.

XML is also great because XML documents are self-describing. Here is a simple XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<phonenumber country="us">
  <areacode>215</areacode>
  <number>6424748</number>
```

```
</phonenumber>
```

Each XML element has a tag and you can quickly figure out what this data represents, even years later. This means XML documents are self-documenting.

Thirdly, XML has syntax that allows the author of the XML document to give structure to the data. The nesting of data elements into other elements is important because data is rarely simple. XML is simple in design but can represent complex data. With HTML you could represent a long purchase order but sharing that purchase order with other programs or organizations would be difficult - they don't know your structure. HTML was designed for presentation not definition of structure. In XML you can build in the semantic rules that specify the structure of the purchase order document. Another XML document would be created to describe how best to display that purchase order. XML separates structure and presentation.

What is XML?

XML gives you the ability to describe “semistructured” data. Having just said that XML allows you to structure data this may sound a bit confusing. Semistructured data is defined as “schema-less” or self-describing”. This means that there is no separate description of the type or structure of data.

As you have seen throughout this book, when we store or program a piece of data, we first describe the structure of that data and then create instances of that type. With semistructured data we directly describe the data using a simple syntax. XML is specifically designed to describe data or content, not presentation. This is its fundemental difference from HTML.

Let’s use an example. If I say I have data that describes a restaurant’s menu, it will most likely bring to mind a favorite restaurant and their menu. But look at the problem a bit harder and you realize that all menus do the same thing – describe a restaurant’s fare – but each menu does it a bit differently. Many Chinese restaurants use numbers and separate their entrees by contents – seafood, meat, poultry, pork or vegetarian. Other restaurants structure their menus by the meal – breakfast, lunch or dinner.

```
<?xml version='1.0'?>
<?xml:stylesheet type="text/xsl" href="tij_menu.xsl" ?>
<restaurant
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="tij_menu.xsd">
  <name>TIJ's Restaurant</name>
  <address>
    <addr1>108 Java Sapien Avenue</addr1>
    <city>Wayne</city>
    <state>PA</state>
    <country>USA</country>
  </address>
  <phone>610-687-1234</phone>
  <menu>
    <menugroup desc="Starters">
      <menuitem>
        <name>Nachos</name>
        <price>$7.25</price>
        <description>cheddar and monterey jack cheese, salsa, sour cream and guacamole.  add jalapenos
or black olives $.50 each</description>
      </menuitem>
      <menuitem>
        <name>Portabello Mushroom Cap</name>
        <price>$8.75</price>
        <description>Stuffed with crabmeat, andouille sausage and smoked tomato sauce</description>
      </menuitem>
      <menuitem>
        <name>Chicken Wings</name>
        <price>$4.95</price>
        <description>A dozen chicken Wings with blue cheese sauce.  Two dozen for
$8.75</description>
      </menuitem>
      <menuitem>
        <name>Quesadilla</name>
        <price>$6.25</price>
        <description>Chicken, black beans, guacamole, salsa, fresca and sour cream</description>
      </menuitem>
    </menugroup>
    <menugroup desc="Salads, Big Salads, and Soups">
      <menuitem>
        <name>House Salad</name>
        <price>$4.75/$8.00</price>
        <description>Small or large house salad with balsamic herbed vinaigrette</description>
      </menuitem>
      <menuitem>
        <name>Mesclun Salad</name>
        <price>$7.25/$13.50</price>
        <description>with marinated mushrooms, roasted peppers, artichokes, fresh mossarella and
balsamic vinaigrette</description>
      </menuitem>
      <menuitem>
        <name>Grilled Chicken Salad</name>
        <price>$6.75/$11.25</price>
        <description>soy sesame marinade, mixed greens, red and green peoppers, carrots, red onion,
crispy fried wontons and honey ginger dressing</description>
      </menuitem>
      <menuitem>
        <name>Chicken Soup of the Day</name>
        <price>$2.30/$3.95</price>
      </menuitem>
    </menugroup>
    <menugroup desc="Pizza">
      <menuitem>
      <name>Duck Confit Pizza</name>
      <price>$9.00</price>
      <description>with caramelized onion, fresh thyme and goat cheese</description>
    </menuitem>
    <menuitem>
      <name>The Salty Pizza</name>
      <price>$8.50</price>
      <description>with proscuitto, black olives cotija cheese, tomatoes and anchovies</description>
    </menuitem>
    <menuitem>
      <name>Traditional Pizza</name>
      <price>$7.50</price>
      <description>red sauce and mozzarella cheese</description>
    </menuitem>
    <menuitem>
      <name>Toppings</name>
      <price>$.50</price>
      <description>Pepperoni, Sausage, Chicken, Grilled Portabello, Roasted Peppers, Sun Dried
Tomatoes, Roasted Garlic, Broccoli Rabe, Spinach, Shrimp, Proscuitto, Black Olives, Goat Cheese,
Bacon, Roasted Veggies, Green Peppers, Onions, Mushrooms</description>
    </menuitem>
  </menugroup>
  <menugroup desc="Entrees">
    <menuitem>
      <name>Grilled Rib Eye Steak</name>
      <price>$18.00</price>
      <description>with buttermilk biscuits, vegetable of the day and button mushroom
gravy</description>
    </menuitem>
    <menuitem>
      <name>Pork Tenderloin</name>
      <price>$16.00</price>
      <description>dry rubbed with swiss chard, maple whipped sweet potatoes and apple bourbon
sauce</description>
    </menuitem>
    <menuitem>
      <name>Grilled Chicken</name>
      <price>$16.00</price>
      <description>with mushroom risotto pesto sauce and tomato salsa</description>
    </menuitem>
    <menuitem>
      <name>Center Cut Sirlion</name>
      <price>$16.00</price>
      <description>with cheddar cheese mashed potatoes, horseradish sour cream and vegetable of the
day</description>
    </menuitem>
  </menugroup>
  <menugroup desc="Kids">
    <menuitem>
      <name>Hot Dog</name>
      <price>$3.50</price>
      <description>on roll; with American Chees $3.75</description>
    </menuitem>
    <menuitem>
      <name>Grilled Cheese Sandwich</name>
      <price>$3.25</price>
      <description>The classic!</description>
    </menuitem>
    <menuitem>
      <name>Chicken Fingers</name>
      <price>$5.00</price>
      <description>Finger licking good</description>
    </menuitem>
  </menugroup>
  <menugroup desc="Beverages">
    <menuitem>
      <name>Milkshake</name>
      <price>$3.50</price>
      <description>vanilla, chocolate or strawberry; double thick, $5.00</description>
    </menuitem>
    <menuitem>
      <name>Orange Cream Soda Float</name>
      <price>$5.00</price>
    </menuitem>
    <menuitem>
      <name>Chocolate Milk</name>
      <price>$2.50</price>
    </menuitem>
  </menugroup>
 </menu>
</restaurant>
```

I don't know if this structure will define all menus for all restaurants but it does describe the menu for a favorite restaurant nearby and it should be able to accommodate most menus.

XML allows users to define tags to indicate structure. In this menu <restaurant> contains <menu>, so our restaurant could have more than one menu - one for lunch and one for dinner. <menu> contains <menugroups> and <menugroups> contain <menuitems>, which represent the food served. There may be a need for more data surrounding the menu items - say calories of the serving.

Unlike HTML, an XML document does not provide any instructions on how it is to be displayed. This type of information would be included separately in a stylesheet. Stylesheets in a specification language, XSL (XML Stylesheet Language) are used to translate XML data to HTML for presenation.

At this point I should also explain that XML is much more than simple rules for constructing data. XML comprises a whole host of specifications that complement it. An area of specification we will look at DTDs and XML Schema. These specifications allow you to describe the structure an XML document should take. Basically, it removes the “semi” from semistructured data.

XML Elements

The basic component in XML is the element. This is a piece of text bounded by matching tags such as <menu> and </menu> . Inside these tags an element may contain “raw” text, other elements or a combination of the two.

In the menu example, <menuitem> is called a start-tag and </menuitem> is called an end-tag. Start- and end-tags are also called markups because they markup or explain the data. One of the rules of XML is that these tags must be balanced. This means that they should be closed in the inverse order in which they are opened, like parentheses. Tags in XML are defined by users; there are no predefined tags as in HTML. The text between a start-tag and the corresponding end-tag, including the embedded tags is called an element and the structures between the tags are referred to as the content. A subelement is the relation between an element and its component elements. Therefore, <price> ... </price> is a subelement of <menuitem> ... </menuitem> in our example.

One element surrounds all the others - <restaurant> ... </restaurant>. This is the root element.

We use repeated elements with the same tag to represent collections. In the menu example, <menu> contains one or more <menugroup> elements and a <menugroup> could contain one or more <menuitem> elements.

An element can also be empty and an empty element may be abbreviated. This is done by putting a ‘/’ at the end of the start tag as in - <menuitem/>.

XML Attributes

XML allows us to associate attributes with elements. An attribute is a name-value pair that acts similar to a “property” in data models. In our menu example we have an attribute named “desc” within the menugroup element.

There are differences between attributes and tags. A given attribute may only occur once within a tag, while subelements with the same tag may be repeated. The value associated with an attribute must be a string while an element may also have subelements as well as values. Therefore, attributes should never be used when a piece of data could be represented as a collection.

Attributes bring a certain level of confusion as to how to represent information as an element or an attribute. For example, we could represent our menugroup as:

```
<menugroup desc="Beverages">
```

or

```
<menugroup>
  <desc>Beverages</desc>
</menugroup>
```

Either format represents the data but as your data becomes more complex you will need to be more selective.

Character Sets

The actual characters in the XML document are stored as numerical codes. The most common set is the American Standard Code for Information Interchange (ASCII). ASCII codes extend from 0 to 255 to fit within a single byte.

XML is the text-based format for data specification that will power the next generation of the World Wide Web. We have a problem here and that is "Worldwide". Many scripts are not handled in ASCII, these include Japanese, Arabic, Hebrew, Bengali and many others languages.

For this reason the default character set specified for XML by the W3C is Unicode, not ASCII. But in practice, Unicode support, like many parts of the XML technology, is not fully supported on most platforms. Windows 95/98 does not offer full support for Unicode, although Windows NT, Windows 2000 and XP come closer. Most often this means that XML documents are written in simply ASCII, or in UTF-8, which is a compressed version of Unicode that uses 8-bits to represent characters.

In our example, the first line states:

```
<?xml version="1.0" encoding="UTF-8"?>
```

this is specifying the use of UTF-8 character encoding.

The default for XML processors today is to assume that your document is in UTF-8, so if you omit the encoding specification, UTF-8 is assumed. For more information on character sets take a look at the list posted by the Internet Assigned Number Authority (IANA) at www.isi.edu/in-notes/iana/assignments/character-sets.

XML Technologies

XML starts out so simply and quickly becomes difficult. XML is difficult for one main reason – there is so much to it. The term XML covers a vast array of technologies that fall under the XML unbrella term. Some of the terminology that is pushed around with XML includes: namespaces, transformations or XSLT, XPath, XLinks, XPointers, SAX and DOM. There seem to be more and more terms every year, as well as more and more books covering these new topics and technologies. I cannot cover them all but I will try to give you the basics.

JAXP – Processing XML

The Java API for XML Processing (JAXP) provides a standard API for parsng and transforming XML documents. This API is designed to be independent from any particular XML processor implementation. In the summer of 2002 Sun released the Java XML Pack. This included a reference version of Xerces 2, as its default XML parser and Xalan as its default XSLT engine, both are from Apache.

We will use these same tools because they are widely used, most up-to-specification and they are free from the apache web site – www.apache.org. So everyone can download them.

XML Namespaces

XML is all about data. Structuring data, defining data and sharing data. XML gives you considerable freedom in defining your own tags but what happens when you define a tag that is already defined by a grammar you want to use.

For example, two popular XML programs are XHTML and MathML. What happens if you want to display equations inside a XHTML document? Some tags in MathML overlap with XHTML.

The answer is namespaces. Creating namespaces allows you to separate one set of tags from another thus avoiding conflicts. Namespaces work by letting you preppend a name followed by a colon to tag and attribute names changing those names so that they don't conflict.

The namespace example that is seen most often is the namespace definition of XML Schema within an XML document.

```
<restaurant
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="tij_menu.xsd">
```

The XML Schema instance namespace is defined and associated with a URL. This namespace can now be abbreviated with xsi and it is used for specifying information in the XML document about a schema. That is exactly what we are doing here with xsi:noNamespaceSchemaLocation. This attribute is prefaced with xsi.

The XML schema defines a set of new names and that describe the elements, types, attributes, whose definitions are written in the schema. This document then must be built according the schema rules.

Well-Formed and Valid XML

There are few constrainsts on XML documents, tags have to match and nest properly, and attributes have to be unique. A document is said to be well formed when it meets these simple rules. It does little more than ensure that XML data will parse into a labeled tree.

A more strict set of rules can be applied if the XML must abide by an underlying grammar. An XML Application is a specification of the syntax and semantics of a data structure. MathML and Scalable Vector Graphics are both XML Applications. A document type definition DTD or XML Schema provide the structure rules by which an XML document must abide to be “valid”. DTDs are somewhat unsatisfactory and the XML Schema is now becoming the standard for defining an XML grammar.

Here’s the schema for the tij_menu.xml example – tij_menu.xsd:

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <xsd:annotation>
    <xsd:documentation>
      TIJ Restaurant Menu Schema.
    </xsd:documentation>
  </xsd:annotation>

   <xsd:element name="restaurant"
        type="restaurantType"/>

   <xsd:complexType name="restaurantType">
     <xsd:element name="name" type="xsd:string"/>
     <xsd:element name="address" type="addressType"/>
     <xsd:element name="phone" type="xsd:string"/>
     <xsd:element name="menu" type="menuType"/>
   </xsd:complexType>

   <xsd:complexType name="menuType">
     <xsd:attribute name="desc" type="xsd:string"/>
     <xsd:element name="menugroup"
          type="menugroupType" minOccurs="1"
          maxOccurs=”unbounded”/>
  </xsd:complexType>

   <xsd:complexType name="menugroupType">
     <xsd:element name="menuitem" minOccurs="1"
               maxOccurs=”unbounded”>
         <xsd:complexType>
         <xsd:element name="name" type="xsd:string"
             minOccurs="1"/>
         <xsd:element name="price" type="xsd:string"
             minOccurs="0"/>
         <xsd:element name="description"
             type="xsd:string" minOccurs="0"/>
      </xsd:complexType>
    </xsd:element>
   </xsd:complexType>
</xsd:schema>
```

XML Schema provides a rich grammatical structure for XML documents that overcomes the limitations of DTD. The goal is to ensure that an XML document is built according to guidelines laid out by a schema. This allows the sharing of similar data, whether purchase orders or menus. If we wanted to start a menu service that allowed users to browse menus of restaurants in their area, we may want all submitting menus to come in a specified format. Or ask for all menu submissions to come with their own schema description so we could transform it.

In tij_menu.xml, <restaurant> is the root element and it contains various subelements. In schema terms, elements that enclose subelements or have attributes are complex types. Elements that enclose only simple data such as numbers, strings or dates – but have no subelements – are simple types. Attributes are always simple types because attributes values cannot contain any structure. If we look at the document as a tree, simple types have no subnodes, while complex types can.

The difference between simple and complex types is an important one because you declare simple and complex types differently. You declare complex types yourself, and the XML Schema specification comes with many simple types already declared.

```
<xsd:element name="price" type="xsd:string">
```

This is an example of a simple type declaration. The types are usually string, integer, decimal, time, etc. Suppose this isn't the right structure for price, we can do better. We decide to add the attribute currency to price. Not allowed, an element of simple type cannot have an attribute. So we have to change price to a complex type.

```
<xsd:complexType name="price">
  <xsd:base="decimal"/>
  <xsd:attribute name="currency" type="string"/>
</xsd:complexType>
```

Elements that embed other elements must be complex types. From the schema above you can quickly see that <restaurant> contains <menu> - <menu> contains <menugroup> and <menugroup> contains <menuitem>. Constraint can be place on element values by using minOccurs and maxOccurs. By default these values are 1 for an element. The tij_menu.xml needs to have many <menuitem> within a <menugroup> so we set the maxOccurs constraint to unbounded.

Creating well-formed and valid XML documents is two step process. Learning to write XML as we have seen is easy, there are very few rules. Next you must learn the XML application(s) you will be using. This is typically a standard or specification that has been defined by some outside organization you would like your data to adhere to. Data wants to be read and creating valid xml documents puts your data into a format that other programs can understand.

Validating Parsers: SAX and DOM

Having the right tools to enforce quality control can make life much easier. This is one of the great strengths of XML and since the goal of XML is to be a universal language that operates in the same manner all the time, standards for data integrity must be high.

The key to this level of integrity is the validating parser. A validating parser checks your XML document for well-formedness, these are usually missing end tags or misspellings. As its name implies this parser goes beyond checking for well-formed documents to check for mistakes that are more difficult to find, missing elements or improper order of elements based upon a document model put forth in a DTD or XML Schema.

Another great aspect about open source validating parsers is that the best ones are free. The parser I will use for this chapter is the Apache Xerces. There are both Java and C++ versions and it supports document models produced in DTDs as well XML Schema. (http://xml.apache.org)

XML has been progressing so quickly it is difficult to keep up. This is true with the tools we will be using. Xerces 2.1.0 is a significant re-architecture from Xerces 1.4. It had to happen because the XML environment has evolved so quickly. Xerces 2 works with SAX 2.0, DOM Level 2, adds some DOM Level 3, ... - it’s endless and the Apache team does an admirable job keeping up with all the revisions and new technologies.

So what exactly does a parser do? It tears apart an XML document and allows you to deal with it programmatically. There are two methods to programmatically deal with an XML document, you can deal with the events that happen as the document is parsed or you can break the XML document into a common representation of the content and model that all tools can manipulate.

Simple API for XML (SAX) is the interface for parsing a document and then inserting application code at important events within the a document’s parsing. SAX is event based XML document manipulation; it allows a document to be handled sequentially, without having to first read the entire document into memory.

The Document Object Model or DOM is a standard representation of the content and model of XML documents across all programming languages and tools. Bindings exist for JavaScript, Java, C++, CORBA, Python, Perl and other languages, allowing DOM to be a cross-platform and cross-language specification.

In its most fundamental form, DOM is a tree model. While SAX gives you an event-by-event perception of the parsing lifecycle, DOM gives you a complete in-memory representation of the document. Both have strengths and weaknesses, as we shall see.

SAX

That’s the basics. Let’s look at the details, we’ll start with a simple SAX program that validates our tij_menu.xml:

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
```

```
public class validate2 {
  static public void main(String[] argv) {

    if (argv.length == 0) {
     System.out.println(
        "Usage: java validate2 file:<filename> or \n"
       + "or java validate2 uri");
   }

    XMLReader parser = null;
    // try to activate validation
    try {
      parser = XMLReaderFactory.createXMLReader(
               "org.apache.xerces.parsers.SAXParser");
      // these features must be set to
      // validate schemas
      parser.setFeature(
          "http://xml.org/sax/features/validation",
          true);
      parser.setFeature(
   "http://apache.org/xml/features/validation/schema",
   true);
    } catch (SAXException e) {
      System.err.println(
         "Cannot activate validation.");
    }

    // parse the document
    try {
      System.out.println(argv[0]);
      parser.parse(argv[0]);
    } catch (SAXParseException e) {
         System.out.println(e.getMessage()
           + " at line " + e.getLineNumber()
           + ", column " + e.getColumnNumber());
    } catch (SAXException e) {
        System.err.println(
          "XML exception reading document.");
    } catch (IOException e) {
        System.err.println(
          "I/O exception reading XML document:\n"
          + e.getMessage());
    }
  }
}
```

The command line to run this example looks like this:

```
java validate2 file:tij_menu.xml
```

The import statements bring in the Java classes that we will need. These include the org.xml.sax classes, which supports the W3C Simple API for XML. SAX provides a common interface for the many different XML parsers. This standard interface is event based and provides a simpler, lower-level access to the XML document.

The first step is to parse the XML document. To parse the XML Document you need an XMLReader object, which we just call parser. We get this through the XMLReaderFactory and supplying the Xerces main SAX parser class, org.apache.xerces.parsers.SAXParser. This class implements the org.xml.sax.XMLReader interface. Each XML parser should have at least one class that implements this interface.

In order to obtain full schema validation we need to set two features on using the setFeature() method of the parser. Validation and SchemaValidation must be set to on. Without SchemaValidation this program will be looking for DTDs.

Lastly, we need to call the parse method of our XMLReader parser. That’s it.

There is great deal more you can do as you parse a document with SAX. This functionality is setup through handlers. A handler is nothing more than a set of callbacks that SAX defines to let programmers insert application code at important events within a document’s parsing. These events take place as the document is parsed, not after it is finished. In SAX 2.0 there are four core handler interfaces defined in SAX 2.0: org.xml.sax.ContentHandler, org.xml.sax.ErrorHandler, org.xml.sax.DTDHandler, and org.xml.sax.EntityResolver. The implementation classes for these interfaces are registered with setContentHandler(), setErrorHandler(), setDTDHandler, and setEntityResolver(). The XMLReader will invoke the callback method on the appropriate handlers during parsing. The ContentHandler interface has callbacks for events such as the start and end of the document, start and end of an element, and a processing instruction.

DOM

The world according to DOM is completely different from SAX. The focus of DOM is the output from the parsing process. Using SAX our program will react at specific points in the parsing process, with DOM the data is unusable until the entire document is parsed. The output of a DOM parse is intended to be used with the DOM interface defined in org.w3c.dom.Document object. This document object is supplied to you in tree format, and all of this is built upon the DOM org.w3c.dom.Node interface. Deriving from this interface, DOM provides several XML-specific interfaces - Element, Document, Attr, and Text. Your DOM-aware application can now move and manipulate various aspects of the tree.

Before the details, you should know a little about the evolution of DOM. The first version of DOM was not an official specification, just the object model that Navigator 3 and Explorer 3 implemented in their browsers. Of course, these object models weren’t totally compatible, so I guess that’s why this is termed DOM Level 0.

DOM Level 1 as an attempt to get some level of compatibility across browers. The goal was to get it out quickly before the two sides went off in opposite directions. Given these constraints, DOM Level 1 is a surprisingly good specification and it covers most of what programmers need to process XML.

DOM Level 2 cleaned up DOM 1 interfaces. Namespace support was added to the Element and Attr interfaces and a number of support interfaces were added for events, traversal, ranges, views, and style sheets. Right now, 2002, all major XML parsers support DOM Level 2 and that should be your coding standard.

DOM Level 3 is coming. Xerces 2 is starting to support parts of DOM Level 3. DOM 3 will provide for parser-independent methods to create a new Document object, either by parsing a file or by building on from scratch in memory. DOM 3 will add support for DTDs and schemas. So from the looks of it DOM 3 adds new functionality and your DOM 2 code should continue to work.

Here is an example NodeWalker.java, it uses the DOM:

```
import org.apache.xerces.parsers.DOMParser;
import org.w3c.dom.*;

public class NodeWalker {
  static String outStr[] = new String[250];
  static int lineNum = 0;

  static public void main(String[] argv) {
    try {
      // Xerces Parser
      DOMParser parser = new DOMParser();
      parser.parse(argv[0]);
      Document doc = parser.getDocument();

      walker(doc.getDocumentElement());
    } catch (Exception e) {
      e.printStackTrace(System.err);
    }

    // output
    System.out.println("Num of line:" + lineNum);
    for(int i=0; i < lineNum; i++){
      System.out.println(outStr[i]);
    }
  }

  public static void walker(Node node) {

    if (node == null) {
      return;
    }
    int type = node.getNodeType();
    switch(type) {
      case Node.ELEMENT_NODE: {
        // Name of this node
        outStr[lineNum] = node.getNodeName();
        lineNum++;

        // attributes of this node
        int length = (node.getAttributes() != null) ?
              node.getAttributes().getLength() : 0;
        for (int i = 0; i < length; i++) {
          Attr attrib =
               (Attr)node.getAttributes().item(i);
          outStr[lineNum] = attrib.getNodeName();
          lineNum++;
        }

        // children of this node
        NodeList children = node.getChildNodes();
        if (children != null) {
          length = children.getLength();
          for (int i = 0; i < length; i++) {
            walker(children.item(i));
          }
        }
        break;
      }

      case Node.TEXT_NODE: {
        String str = node.getNodeValue().trim();
        if (str.indexOf("\n") < 0 &&
            str.length() > 0) {
          outStr[lineNum] = str;
          lineNum++;
        }
        break;
      }
    }
  }
}
```

Notice that the parse() method returns void. This change allows an application to use a DOM parser class and a SAX parser class interchangeably; however, it requires an additional method to obtain the Document object result from the XML parsing. In Apache Xerces, this method is named getDocument().

The parse method translates the stream of characters from the XML file into stream of tokens. These tokens are built into a temporaty structure in memory that a program can move through and manipulate. DOM is an API for XML documents. It offers an object-oriented view of the XML document , as opposed to an abstract data type view. This means that each document component defines an interface specifying its behaviour; the data can only be accessed via this interface. DOM defines a Node interface, and subclasses Element, Attribute, and CharacterData interfaces inheriting from Node. NodeWalker only switches on Node.ELEMENT_NODE and Node.TEXT_NODE, we could add other nodes including CDATA_SECTION, and PROCESSING_INSTRUCTION_NODE. The Node interface defines methods for accessing a node's components. For example, parentNode() returns the parent node; childNodes() returns the set of children. In addition there are methods for modifying the node and for creating new nodes. Therefore, an application can completely restructure the document via the DOM.

The Document interface is based on the Node interface, which supports the W3C Node object. Nodes represent a single node in the document tree and everything, text, comments and tags, is treated as a node. The node interface has all the methods that you can use to work with nodes and move through the DOM. Let's look at the following code from the example:

```
for (Node node =
    doc.getDocumentElement().getFirstChild();
    node != null;
    node = node.getNextSibling()) {
```

This for loop gets started by using the document interface method getDocumentElement() to return the root node of the document. The getFirstChild() method of the node interface will return the node of the first child element of the root. As long as there is no null value returned the for loop will keep rolling along. The node is incremented using the getNextSibling() method which returns the node immediately following the calling node.

Plus and Minus of SAX and DOM

Right away you should realize that DOM may require a sizeabe memory commitment depending upon how large your XML documents are and how many documents will be open at any time. SAX handles the document sequentially and does not need to read the entire document into memory. However, the advantage of DOM and having the entire tree loaded into memory is that you now have the entire document available for random access. This is the ying and yang of DOM and SAX, every strength is a weakness – every weakness a strength.

SAX is generally viewed as being able to parse an XML document of any size. You can also build your oun data structure if a tree will not suit your needs. Another advantage of SAX is its usefulness when you only want a small subset of data from a document.

DOM strength lies in the structure. More complex searches are possible with DOM. One of the most important aspects of DOM is the ability to serialize data to for reconstruction in another program.

XML Serialization

You should see that XML gives you lots of tools to manipulate and work with your data and the goal of XML is portable data, you will find that the much of the XML manipulation will become boilerplate code. As you progress you will want to get rid of the boilerplate and directly serialize to and from XML.

Serialization means outputting the XML. This could be a file, an OutputStream, or a Writer. As you have read in earlier chapters, there are more output forms available in Java, but these three cover the most widely used.

One area in which DOM has been quite weak is serialization, the outputting of an in-memory DOM Document object into a text file. In fact, it's even possible to use DOM to create Document objects that cannot be serialized as well-formed XML files. Serialization has been left to the vendor to implement in classes such as Xerces's XMLSerializer. However, DOM Level 3 adds several classes for writing XML documents into files, onto the network, or anything else you can hook an OutputStream to.

Xerces Serialization

The Apache XML Project's Xerces-J includes the org.apache.xml.serialize package for writing DOM Document objects onto output streams. Although this class is bundled with Xerces, it works with any DOM Level 2 implementation. It does not depend on the details of the Xerces implementation classes, only the standard DOM interfaces.

The mechanics for serializing documents with org.apache.xml.serialize are as follows:

1. Configure an OutputFormat object with the serialization options you want.
2. Connect an OutputStream to the location where you want to store the data.
3. Use the OutputStream and OutputFormat to construct the new XMLSerializer object
4. Pass the Document object you want to serialize to the XMLSerializer's serialize() method.

Here is an example that will take the tij_menu.xml and serialize it to system.out:

```
import org.apache.xerces.parsers.DOMParser;
import org.apache.xml.serialize.XMLSerializer;
import org.apache.xml.serialize.OutputFormat;
import org.w3c.dom.Document;

public class menuSerializer {
  static public void main(String[] argv) {
    try {
      // Xerces Parser
      DOMParser parser = new DOMParser();
      parser.parse("file:tij_menu.xml");
      Document doc = parser.getDocument();

      // serialize document
      OutputFormat format = new OutputFormat(doc);
      XMLSerializer output = new XMLSerializer(
                         System.out, format);
      output.serialize(doc);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

XMLSerializer has several constructors you can pick from depending upon whether you are writing to an OutputStream or a Writer and whether or not you want to provide an OutputFormat when you create the serializer:

```
public XMLSerializer();
public XMLSerializer(OutputFormat format);
public XMLSerializer(Writer out, OutputFormat format);
public XMLSerializer(OutputStream out,
                OutputFormat format);
```

You should probably define the format and the stream or writer when you construct the XMLSerializer. You can also change them through these methods:

```
public void setOutputFormat(OutputFormat format);
public void setOutputByteStream(OutputStream out);
public void setOutputCharStream(Writer out);
```

DOM Level 3 Serialization

DOM Level 3 will finally add a standard Load and Save package so that it will be possible to write completely implementation independent DOM programs. This package, org.w3c.dom.ls is identified by the feature strings LS-Load and LS-Save. The loading is done through DOMBuilder interface and the saving is based on DOMWriter interface. DOMWriter is more powerful than XMLSerializer because it is not limited to outputting documents, document fragments and elements. DOMWriter can output to any kind of node at all and filters can be installed to control output.

The specification for this is still in a state of flux. The latest work from the W3C can be found in Document Object Model (DOM) Level 3 Abstract Schemas and Load and Save Specification and in Xerces-J 2.0.2.

XPath

To do...

XML Transformations

In the beginning of this chapter we said that Java provides portable code and XML provides portable data. So far we have gone over XML, what it is, how to define it, how to structure it, how to manipulate it and how to output it. But if we want XML to be portable data we have to be able to transform our XML so it can be read by another application, displayed or printed. Extensible Stylesheet Language (XSL) provides facilities to access and manipulate the data in XML documents.

XSL is itself an XML dialect and provides two distinct and useful mechanisms for handling and manipulating XML documents. Many of the same constructs are shared between the two mechanisms, but each plays a distinct role. One is concerned with formatting data, the other is concerned with data transformation. When XSL is used as a formatting language, the stylesheets consist of formatting objects that prepare an XML document for presentation, usually in a brower.

When XSL is used for transformation, XSL takes the form of Extensible Stylesheet Language Transformation (XSLT). An XSLT stylesheet is composed of template rules that match specific portions of an XML document and allow the transformation of the XML document content. Not only can XSLT transform an XML document from one dialect to another (often HTML), but it provides many other capabilities for extracting data from an XML document and manipulating that data.

Java API for XML Processing (JAXP) Summer 2002 Pack comes with another Apache tool named Xalan. Xalan is the Apache.org tool for performing transformation. Two tools are needed to perform a transformation, a parser and an XSLT processor. We will use these Apache.org tools.

An XSLT document, referred to as a stylesheet, consists of a series of template rules. Each template rule matches against elements, attributes, or both within the target XML document. The basic construct for a template rule is:

```
<xsl:template match=”pattern”>
  … rule body …
</xsl:template>
```

The template rule has a start tag (<xsl:template>) and an end tag (</xsl:template>). Normally, each template start tag has a match attribute that specifies the portion of the input XML document that the template rule is intended to match against.

A template rule body can consist of the following:

1. More detailed selection or match conditions and other logic
2. A specific type of action or actions to be performed
3. Text that becomes part of the results along with the selected target XML documents content

From this snippet you see the mechanics of an XSLT transformation. First, the XML must be parsed and the XML document takes the form of a tree. The nodes of that tree become important in creating matches. Second, this tree is passed to the XSLT processor. The XSLT processor compares the nodes in the tree to the instructions in the style sheet. When the XSLT processor finds a match, it outputs the fragment. Lastly, the output is serialized to another format such as HTML, or a new XML file.

Let's build an example.

XML to HTML: Displaying a menu

Going back to the menu example, suppose we joined forces with a travel bureau and we wanted to display all the restaurants in their database as well as all ours. They can output XML for us but it does not use the same tags and structure so it is virtually unreadable to our server. This means we have to transform it.

Let's start as simply as possible. menu0.xsl is the most basic xsl style sheet:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

</xsl:stylesheet>
```

We will run it against our tij_menu.xml file with the Xalan XSLT process using the following command line:

```
java org.apache.xalan.xslt.Process
    -in tij_menu.xml
    -xsl file:menu0.xsl
    -out menu0.html
```

I can't print the output here because it is too large but I can describe it. menu0.html is simply every piece of text that was not a tag or attribute. Every text node was printed out. We need to get control of this process. To get control of the transformation process we need to understand the structure of our input document, move through the elements and attributes transforming them to our desired output format.

The Root Node

The root node is the XPath node that contains the entire XML document; it is the best place to start. In the tij_menu.xml example, the root node will contain the <restaurant> element.   In an XPath expression, the root node is specified with a single slash - /.

The root node is different because it has no parent. It always has at least one child. The string value of the root node is the concatenation of all the text nodes of the root node's decendants. Let's begin our transformation of tij_menu.xml to menu.html by adding a template to act on the root node. Here is menu1.xsl:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <HTML>
    <HEAD><TITLE>Menu</TITLE></HEAD>
    <BODY>

    </BODY>
    </HTML>
  </xsl:template>

</xsl:stylesheet>
```

Now our output is tamer but we have nothing from our tij_menu.xml file. All the text from the child text nodes has been lost. Here is menu1.html:

```
<HTML>
<HEAD>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<TITLE>Menu</TITLE>
</HEAD>
<BODY></BODY>
</HTML>
```

Our simple xsl file has drilled in only as far as the root element. We have printed out what is obviously the shell of an HTML file. The task now is dig into the structure of our tij_menu.xml data. Since we are dealing with the tree structure of an XML document and we have just processed the root node – this means we must apply these template rules to the children of a node we have matched. This is done with the <xsl:apply-templates> element.

At this point we need to add two pieces to our XSL file. First, we need more rules to process the different elements in the menu. Second, we need to have the processor carry on once it is inside a rule.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
```

```
              xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <HTML>
    <HEAD><TITLE>Menu</TITLE></HEAD>
    <BODY>
    <xsl:apply-templates/>
    </BODY>
    </HTML>
  </xsl:template>

  <xsl:template match="restaurant">
    <center><h2>
    <xsl:value-of select="name"/></h2><h3>
    <xsl:value-of select="address/addr1"/><br/>
    <xsl:value-of select="address/city"/>,
    <xsl:value-of select="address/state"/>
    Phone:  <xsl:value-of select="phone"/><br/></h3>
    </center>
  </xsl:template>

</xsl:stylesheet>
```

Inside our root rule I have added the <xsl:apply-templates/> tag. This will transform all the children of the root element. Since the root element contains only one element, <restaurant>, a rule will be added to handle <restaurant>.

<xsl:template match="restaurant"> is the rule that will fire when the XSLT processor passes over the <restaurant> tag. Once in the tag we need to transform the data. This data will make up the header for our restaurant menu – restaurant name, addresss, and phone number.

The output from the transformation looks like this:

```
<HTML>
<HEAD>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<TITLE>Menu</TITLE>
</HEAD>
<BODY>
<center>
<h2>TIJ's Restaurant</h2>
<h3>108 Java Sapien Avenue<br>Wayne,
    PA<br>
    Phone:  610-687-1234<br>
</h3>
</center>
</BODY>
</HTML>
```

Now let’s build the menu. Each restaurant has at least one <menu> element, <menu> elements have <menugroup>s and <menugroup> elements have <menuitem>s. So we have to add the rest of the rules to handle those elements.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">


 <xsl:template match="/">
  <HTML>
  <HEAD><TITLE>Menu</TITLE></HEAD>
  <BODY>
  <xsl:apply-templates/>
  </BODY>
  </HTML>
 </xsl:template>

 <xsl:template match="restaurant">
  <center><h2>
  <xsl:value-of select="name"/></h2><h3>
  <xsl:value-of select="address/addr1"/><br/>
  <xsl:value-of select="address/city"/>,
  <xsl:value-of select="address/state"/>
  <xsl:value-of select="address/zip"/></h3>
  </center>
  <xsl:apply-templates select="menu"/>
 </xsl:template>

 <xsl:template match="menu">
  <h2>Menu</h2>
  <dl>
  <xsl:apply-templates select="menugroup"/>
  </dl>
 </xsl:template>

 <xsl:template match="menugroup">
  <dt><h3>
  <spacer type="horizontal" size="25"/>
  <xsl:value-of select="@desc"/></h3></dt>
  <xsl:apply-templates select="menuitem"/>
 </xsl:template>

 <xsl:template match="menuitem">
  <dd><b>
  <spacer type="horizontal" size="25"/>
  <xsl:value-of select="name"/>:</b><br/>
  <ul>
  <xsl:apply-templates select="description"/>
  Price:<xsl:value-of select="price"/><br/>
  </ul>
```

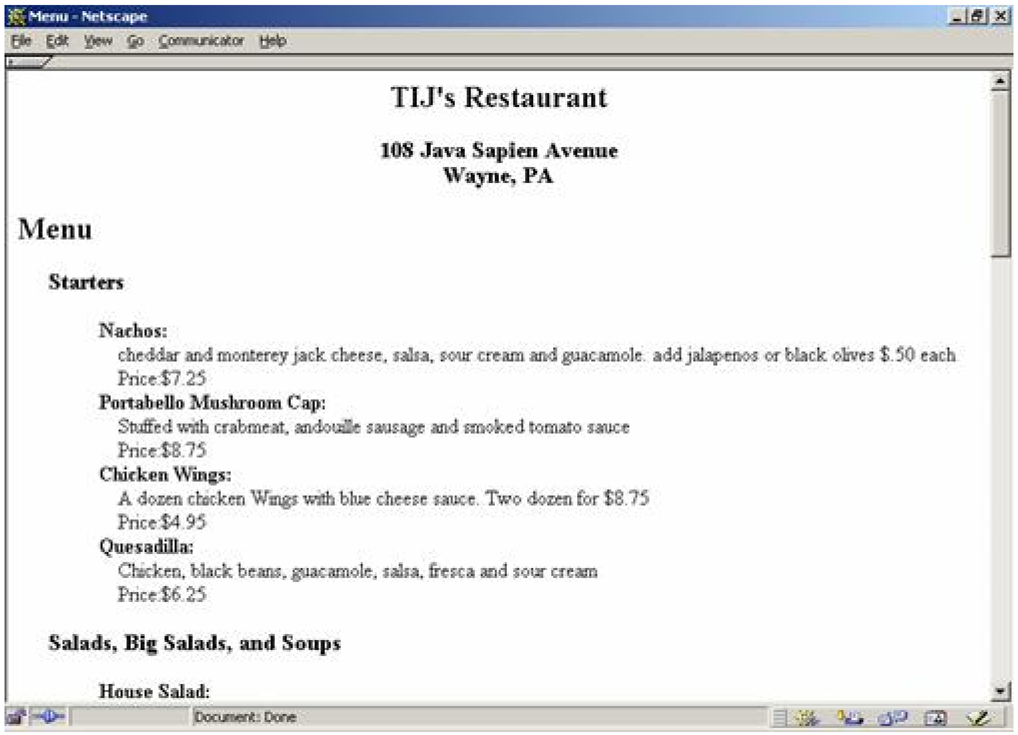
```
    </dd>
  </xsl:template>

  <xsl:template match="description">
    <xsl:value-of select="text()"/><br/>
  </xsl:template>

</xsl:stylesheet>
```

The <menu> element has not text nodes only <menugroup> elements. So the goal with the menu template is to set up the menugroups so they will be framed correctly and call the menugroup template.

The menugroup has an attribute describing what type of menugroup it is. This is in the desc attribute of the <menugroup> element. An '@' symbol must be placed in front of the attribute name in order to select the value using the value-of tag - <xsl:value-of select="@desc"/> . Other than that the rest of the rule is setting up the for the element list within menugroup – <menuitem>.

The rule for <menuitem> bolds the name of the food then creates an unorder list for the rest of the elements in <menuitem>. Since some menu items have descriptions and some do not – a rule was created specifically for description. If a menu item has a description it will be called and the text node of the description will be pulled out by the text() method placed in the select attribute's field.

Here is what the final the HTML form looked like.



TIJ's Restaurant

108 Java Sapien Avenue
Wayne, PA

Menu

Starters

Nachos:
cheddar and monterey jack cheese, salsa, sour cream and guacamole. add jalapenos or black olives $.50 each
Price:$7.25
Portabello Mushroom Cap:
Stuffed with crabmeat, andouille sausage and smoked tomato sauce
Price:$8.75
Chicken Wings:
A dozen chicken Wings with blue cheese sauce. Two dozen for $8.75
Price:$4.95
Quesadilla:
Chicken, black beans, guacamole, salsa, fresca and sour cream
Price:$6.25

Salads, Big Salads, and Soups

House Salad:

It’s nothing fancy but you should have a good idea of the capabilities of XSLT.

XLink and XPointer

To do

Summary

This chapter has introduced some but not all of the components that Sun references in its JAX-Pack Summer 2002. The goal of XML is to provide portability of data, thus enabling applications that were developed completely separate from one another to share the same information. XML is an umbrella term that covers many technologies built from XML's simple roots. These technologies have continued to evolve and standardize quickly. In your local bookstore there will be many shelves of XML books and most will be outdated before long. If you want to find more information or stay current on developments in the XML community you should point your browser to the W3C - www.w3.org and the XML work being done in the Apache community - www.apache.org.

[1] This means a maximum of just over four billion numbers, which is rapidly running out. The new standard for IP addresses will use a 128-bit number, which should produce enough unique IP addresses for the foreseeable future.
[2] Many brain cells died in agony to discover this information.
[3] Created by Dave Bartlett.
[4] Dave Bartlett was instrumental in the development of this material, and also the JSP section.
[5] A primary tenet of Extreme Programming (XP). See *www.xprogramming.com*.
[6] This chapter was written by Jeremy Meyer.

[7] This chapter was created by Andrea Provaglio.
[8] This chapter was created by Dave Bartlett.