

# Getting started with Enterprise JavaBeans technology

Skill Level: Introductory

[Joe Sam Shirah](#)  
Author

01 Apr 2003

This tutorial introduces the basics of EJB programming and the Java 2 Enterprise Edition environment. Joe Sam Shirah takes you through stateless and stateful session beans, entity beans using both bean-managed and container-managed persistence, and message-driven beans. The tutorial also provides background material for transaction handling and the Java Message Service, and includes complete code examples for functional, Web-based applications.

## Section 1. About this tutorial

### What is this tutorial about?

The goal of this tutorial is to give you a practical introduction to the basics of Enterprise JavaBeans (EJB) programming and development. EJB components are at the heart of the Java 2 Platform, Enterprise Edition (J2EE) framework, which is the Java language's extremely successful solution to the needs of enterprise applications. EJB technology provides a standard server-side component model for J2EE application servers, or *containers*.

This tutorial covers stateless and stateful session beans, entity beans using bean-managed persistence (BMP) and container-managed persistence (CMP), and message-driven beans. Along the way, the tutorial also touches on JDBC, transaction handling with the Java Transaction API (JTA), and the Java Message Service (JMS), and uses these technologies in the sample code. Complete examples based on functional Web applications are discussed throughout the

tutorial and a link for the full download of the code is provided in [Resources](#).

After working through the tutorial, you should:

- Understand EJB technology concepts and terminology
- Understand how EJB components work cooperatively with J2EE containers
- Understand the process and requirements for creating EJB components
- Have a solid foundation for developing your own Enterprise JavaBean and J2EE applications

## Should I take this tutorial?

While the tutorial is an introduction to development with EJB technology, this is not entry-level material; you should be an intermediate or higher level Java programmer with some understanding of JDBC to get the most out of it and fully understand the code examples. You will also need to know how servlets and JavaServer Pages (JSP) technology work to understand the client code. However, you can read the material purely to gain an overview of what EJB technology is about and how J2EE and EJB components work together.

The tutorial covers EJB technology as of the J2EE 1.3.1 SDK and the EJB 2.0 specification. At the time of writing, a beta release of J2EE 1.4 and a draft of the 2.1 EJB specification had just been released. [Appendix D: What's new in the EJB 2.1 specification?](#) briefly reviews the new capabilities.

In short, if you are a reasonably experienced Java programmer interested in increasing your skill set and understanding the Java platform's approach to enterprise applications, this tutorial is for you.

If you intend to create your own J2EE and EJB applications or want to understand the flow and rationale for the example applications, I suggest going through the tutorial in the following order:

- Read the [J2EE/EJB technology overview](#) .
- Read [Appendix A: Installing and running J2EE](#).
- Read [Appendix B: Deployment](#).
- Read [Appendix C: About the example applications](#).
- Then continue with the tutorial at [Session beans overview](#).

The [Resources](#) present a list of tutorials, articles, and other references that expand

upon the material presented here.

## Code samples and installation requirements

To compile and run the example code, and to get hands-on experience in creating EJB components and J2EE applications, you will need both the J2SE and J2EE SDKs, as well as a source code editor or IDE.

The examples were tested using J2SE 1.4.1 and the J2EE 1.3.1 SDK Reference Implementation (J2EE RI) on Windows NT 4.0, Service Pack 6a. The applications have also been proven to run properly, with the appropriate deployment, under JBoss 3.0.4 on both Windows NT and SuSe Linux.

You should be aware that the example programs focus on EJB development for the relevant section, and have not been optimized for production use. Any introductory and most intermediate material suffers this limitation due to the need to zero in on the target topic without the litter of numerous real-world considerations. That's certainly true here, due to the breadth of the areas covered, and many best practices had to be left undone or unmentioned to keep the tutorial understandable and to a reasonable size. The upshot is that you will learn enough to get started with J2EE and EJB technology, but you should look to the [Resources](#) and other references before embarking on production-quality applications. In particular, if you want to be more than just a "follow the IDE wizard" EJB developer, you will find yourself frequently reviewing the EJB specification.

Even so, you should read through the example explanations whether or not you intend to review the code or run the applications. These discussions mention specific techniques, and often expand or augment the relevant tutorial section.

The classes, Enterprise Archive (EAR) files, and source code for the examples used in the tutorial are available separately in gsejbExamples.jar. The JAR contains a main directory named gsejbExamples, with subdirectories named Alice, MetricCvt, Survey1, Survey2, Survey3, and SurveyResults, each of which corresponds to an example application. You can also see the applications in action before, or instead of, deploying them yourself at conceptGO's [Community](#) page.

---

## Section 2. J2EE/EJB technology overview

### The J2EE platform

The J2EE 1.3 specification (see [Resources](#) ) defines a standard mechanism, known as the *J2EE platform*, for hosting J2EE applications. The platform uses a component-based approach to provide a multitiered, distributed, transactional model for enterprise applications. If that sounds like a mouthful, it certainly is. A J2EE application server that is compliant with the J2EE specification must provide services and APIs for:

- Enterprise JavaBeans Technology 2.0
- JDBC API 2.0
- Java Servlet Technology 2.3
- JavaServer Pages Technology 1.2
- Java Message Service 1.0
- Java Naming and Directory Interface 1.2
- Java Transaction API 1.0
- Java Mail API 1.2
- JavaBeans Activation Framework 1.0
- Java API for XML Processing 1.1
- J2EE Connector Architecture 1.0
- Java Authentication and Authorization Service 1.0

As we will see, to manage this massive amount of functionality in a multitier, multithreaded, multiuser environment, the J2EE platform imposes many restrictions on developers, especially in regard to EJB components.

## Containers and components

The application server maintains control and provides services through an interface or framework known as a *container*. There are five defined container types in the J2EE specification. Three of these are server-side containers:

- The server itself, which provides the J2EE runtime environment and the other two containers
- An EJB container to manage EJB components
- A Web container to manage servlets and JSP pages

The other two container types are client-side:

- An application container for stand-alone GUIs, console, and batch-type programs -- the familiar Java applications started with the `java` command
- An applet container, meaning a browser, usually with the Java Plug-in

There are three kinds of defined components:

- Client components, which correlate to the client containers
- Web components -- servlets and JSP pages
- EJB components

In the server-side J2EE context, a component is a unit of functionality that is assembled, along with any required resources, into a J2EE application. Note that a "J2EE application" may be a single component. The server-side components can communicate with each other inside the J2EE environment (intra-VM on a single machine) or in a networked, multimachine, distributed environment. The client-side containers communicate with the J2EE container and server-side components through a client JAR generated at deployment time. Client-side containers can be on the same machine or, more typically, somewhere on the network. Also, components can be clients of other EJB components -- that is, they can call on the services of other beans.

While any or all of the containers and components may be on the same or multiple machines -- remember, J2EE is a distributed model -- conceptually, the architecture is almost always three tier. These tiers are client, business logic, and database/Enterprise Information Systems (EIS).

## Packaging applications and components

Under J2EE, applications and components reside in Java Archive (JAR) files. These JARs are named with different extensions to denote their purpose, and the terminology is important.

- Enterprise Archive (EAR) files represent the application, and contain all other server-side component archives (as mentioned below, JARs for EJB components, WARs for Web components, etc.) that comprise the application.
- Client interface files and EJB components reside in JAR files.
- Web components reside in Web Archive (WAR) files.

Deployment descriptors are included in the JARs, along with component-related resources. Deployment descriptors are XML documents that describe configuration

and other deployment settings. Remember that the J2EE application server controls many functional aspects of the services it provides. The statements in the deployment descriptor are declarative instructions to the J2EE container; for example, transactional settings are defined in the deployment descriptor and implemented by the J2EE container. Most J2EE vendors provide a GUI tool for generating deployment descriptors and performing deployment because creating manual entries is tedious and error prone. The J2EE RI includes `deploytool`, which is the deployment vehicle we'll use in this tutorial.

The deployment descriptor for an Enterprise JavaBean component must be named `ejb-jar.xml`, and it resides in the `META-INF` directory inside the EJB JAR file. A JAR can contain multiple beans; if so, a single `ejb-jar.xml` file describes all the beans in the JAR. For more information concerning packaging and deployment, see [Appendix B: Deployment](#).

## EJB components

EJB components are server-side, modular, and reusable, comprising specific units of functionality. They are similar to the Java classes we create every day, but are subject to special restrictions and must provide specific interfaces for container and client use and access. In addition, they can only run properly in an EJB container, which manages and invokes specific life cycle behavior. You should consider using EJB components for applications that require scalability, transactional processing, or availability to multiple client types.

EJB components come in three varieties, each with its own defined role and life cycle:

- **Session beans.** These may be either *stateful* or *stateless* and are primarily used to encapsulate business logic, carry out tasks on behalf of a client, and act as controllers or managers for other beans.
- **Entity beans.** Entity beans represent persistent objects or business concepts that exist beyond a specific application's lifetime; they are typically stored in a relational database. Entity beans can be developed using *bean-managed persistence*, implemented by the developer, or *container-managed persistence*, implemented by the container.
- **Message-driven beans.** Message-driven beans listen asynchronously for Java Message Service (JMS) messages from any client or component and are used for loosely coupled, typically batch-type, processing.

This tutorial covers the EJB 2.0 specification (see [Resources](#) ), which has significant differences from the previous versions. In particular, container-managed persistence is newly implemented in the 2.0 version, and message-driven beans are a

brand-new bean type.

Much of the code in bean methods will be familiar, but, to function effectively, EJB components must be able to access J2EE resources, and know how to respond to life cycle calls and callbacks from the EJB container. Unfortunately, the specification defines life cycle methods in three different areas: some are defined in [The home interface](#); others are defined in the various bean interfaces defined in the `javax.ejb` package; still others are simply mandated by the specification. The tutorial discusses the complete life cycle methods for each bean type in the appropriate section. You will frequently see the pattern of `methodName()` in the exposed client interface, with a matching `ejbMethodName()` for life cycle methods.

According to the J2EE specification, "Applications must be able to access resources and external information in their operational environment without knowledge of how the external information is named and organized in that environment." J2EE makes use of the Java Naming and Directory Interface (JNDI) to accomplish this goal. EJB components fit under this definition of resources, and session and entity beans must provide a *home interface* to allow the container and clients to obtain a reference to a specific bean. The container makes the bean available via JNDI. Message-driven beans, which have no direct client, are an exception to the general rule for required interfaces.

It is very important to understand that *EJB components are never accessed directly by clients*. Instead, the container generates proxy objects that implement the home and component interfaces and are used by the client for communication with a bean. The proxy objects then call upon the bean to perform the requested task. This mechanism allows the container to coordinate and manage beans however and wherever it sees fit; the client remains blissfully unaware of the internal details.

## EJB programming restrictions

For the container to properly carry out its duties, provide appropriate services, and manage the components within its environment, many restrictions apply when programming with EJB technology. These rules often cause much confusion and gnashing of teeth among developers, but it is best to understand and abide by them to avoid problems in your beans and applications. The following is reproduced verbatim from Section 24.1.2, "Programming restrictions," of the EJB 2.0 Specification (see [Resources](#)):

"This section describes the programming restrictions that a Bean Provider must follow to ensure that the enterprise bean is portable and can be deployed in any compliant EJB 2.0 Container. The restrictions apply to the implementation of the business methods. Section 24.2, which describes the Container's view of these restrictions, defines the programming environment that all EJB Containers must provide.

- An enterprise Bean must not use read/write static fields. Using read-only static fields is allowed. Therefore, it is recommended that all static fields in the enterprise bean class be declared as final.
- An enterprise Bean must not use thread synchronization primitives to synchronize execution of multiple instances.
- An enterprise Bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard.
- An enterprise bean must not use the `java.io` package to attempt to access files and directories in the file system.
- An enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast.
- The enterprise bean must not attempt to query a class to obtain information about the declared members that are not otherwise accessible to the enterprise bean because of the security rules of the Java language. The enterprise bean must not attempt to use the Reflection API to access information that the security rules of the Java programming language make unavailable.
- The enterprise bean must not attempt to create a class loader; obtain the current class loader; set the context class loader; set security manager; create a new security manager; stop the JVM; or change the input, output, and error streams.
- The enterprise bean must not attempt to set the socket factory used by `ServerSocket`, `Socket`, or the stream handler factory used by URL.
- The enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread; or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups.
- The enterprise bean must not attempt to directly read or write a file descriptor.
- The enterprise bean must not attempt to obtain the security policy information for a particular code source.
- The enterprise bean must not attempt to load a native library.
- The enterprise bean must not attempt to gain access to packages and classes that the usual rules of the Java programming language make unavailable to the enterprise bean.
- The enterprise bean must not attempt to define a class in a package.
- The enterprise bean must not attempt to access or modify the security

configuration objects (Policy, Security, Provider, Signer, and Identity).

- The enterprise bean must not attempt to use the subclass and object substitution features of the Java Serialization Protocol.
- The enterprise bean must not attempt to pass this as an argument or method result. The enterprise bean must pass the result of `SessionContext.getEJBObject()`, `SessionContext.getEJBLocalObject()`, `EntityContext.getEJBObject()`, or `EntityContext.getEJBLocalObject()` instead."

## Local and remote interfaces

*Note: The following discussion of bean interfaces applies only to session and entity beans. Message-driven beans effectively run within a batch environment on the server and are represented only by the bean class -- no interfaces are used.*

Clients access a session or entity bean through the bean's interfaces. The EJB container generates the interface implementations to enforce and manage this behavior, acting as a conduit for communication between the client and the bean. In versions before the EJB 2.0 specification, all beans were defined and implemented as distributed, remote components. As a result, the two interfaces required of beans were termed the *home interface* (which in general defines life cycle methods) and the *remote interface* (which in general defines functional business methods).

Internally, J2EE uses Java Remote Method Invocation over Internet Inter-ORB Protocol technology (RMI-IIOP; see [Resources](#)) to enable remote, distributed method calls and applications. While this approach provides many benefits, it does involve a large amount of overhead and a corresponding performance hit as stubs are referenced, parameters go through the marshaling process, and objects are tossed around the network.

Considerations of performance, practicality, and typical usage in the field resulted in the introduction of *local interfaces* in the EJB 2.0 specification. As noted, prior terminology referred to the home interface and the remote interface; at this point, depending on which approach is used, it's better to use the terms *local interface* and *local home interface* or *remote interface* and *remote home interface*. Either of the local home or remote home interfaces is referred to as the home interface; either of the local or remote interfaces is referred to as the *component interface*. This tutorial refers to the interfaces in these terms and uses these conventions for names.

When using J2EE technologies, it is normal to focus on distributed, or remote, beans, but you should keep the local option in mind, when applicable. It may surprise you to learn that a bean can have local interfaces, remote interfaces, or both. However, the client must write to a specific (that is, local or remote) interface.

There are some issues to keep in mind when using local interfaces:

- The beans must run in the same VM -- they are, after all, *local*.
- Parameters running under a local interface are sent by *reference* rather than being copied, as is the case for remote objects. Unexpected side effects can result if you ignore this distinction and do not code accordingly.

Typically, you'll decide whether to use local or remote access based on:

- **The type of client.** Unless the client is always expected to be a Web component or another bean, choose remote access.
- **Whether the beans are tightly or loosely coupled.** If beans depend on each other and interact frequently, you should consider local access.
- **Scalability.** Remote access is inherently scalable and should be used if scalability is an important factor.

With the advent of local interfaces in the EJB 2.0 specification, it is recommended that *entity beans* should almost always be based on local access. When using local interfaces, most performance issues regarding very fine-grained data access go away. If the client is remote, the standard design pattern has the client use a remote interface to access a session bean, which then acts as a liaison to the entity bean. The session bean communicates with the entity bean through a local interface (from a patterns viewpoint, this technique is called a *Session Facade*, which can actually be used in either a remote or local context).

**Performance note:** The tutorial examples all have Web components (JSP pages) as clients. For the purposes of the tutorial, remote interfaces are used for communication between page and bean. Remote interfaces will work for any client. However, the servlet engine will almost always be running inside the J2EE container and in the same VM. This means that you can use local interfaces for servlet and JSP clients, with a resulting performance boost.

For other considerations regarding local and remote interfaces, consult the EJB 2.0 specification (see [Resources](#) ).

## The home interface

*Note: The following discussion of bean interfaces applies only to session and entity beans. Message-driven beans effectively run within a batch environment on the server and are represented only by the bean class -- no interfaces are used.*

A bean's home interface specifies methods that allow the client to create, remove,

and find objects of the same type. The home interface may also provide definitions for *home business methods* for entity beans. Home business methods are methods that are not specific to a particular bean instance. While the developer writes the home interface, the container creates the implementation for client interaction. In essence, the home interface provides bean management and life cycle methods.

The client uses a JNDI lookup to locate a bean's home interface. The EJB 1.1 specification introduced the environment naming context (ENC) as a means of enhancing portability and avoiding name clashes in the JNDI namespace. In practice, this means that you should preface the lookup string with `java:comp/env/`. The container is required to recognize a lookup coded in this manner as an alias or nickname rather than the direct JNDI name. A deployment descriptor entry links the alias to the actual JNDI entry. Use of the ENC also means that the developer doesn't have to worry about hardcoded JNDI names; the ENC effectively makes them softcoded. The sample code below locates the remote home interface for a bean with a JNDI name of `ejb/MyEJB` (this could be a completely different external name, depending on the `ejb-ref` entry in the deployment descriptor). This code, and the following code for local home lookup, is virtually identical in every client's bean access routine:

```
InitialContext ic = new InitialContext();
Object oRef =
    ic.lookup( "java:comp/env/ejb/MyEJBBean" );
MyEJBRemoteHome MyEJBHome =
    (MyEJBRemoteHome)PortableRemoteObject.
        narrow( oRef, MyEJBRemoteHome.class );
```

The `PortableRemoteObject.narrow()` method is required to ensure conversion to the proper remote interface type.

For a bean that provides a local home interface instead of, or in addition to, a remote home interface, the following code is typical:

```
InitialContext ic = new InitialContext();
MyEJBLocalHome MyEJBHome =
    (MyEJBHome)ic.lookup( "java:comp/env/ejb/MyEJBBean" );
```

**Figure 1. The `javax.ejb.EJBHome` and `javax.ejb.EJBLocalHome` interfaces**

«interface» EJBHome extends java.rmi.Remote	«interface» EJBLocalHome
<pre>getEJBMetaData() : javax.ejb.EJBMetaData     throws java.rmi.RemoteException  getHomeHandle() : javax.ejb.HomeHandle     throws java.rmi.RemoteException  remove( javax.ejb.Handle )     throws java.rmi.RemoteException,            javax.ejb.RemoveException  remove( java.lang.Object )     throws java.rmi.RemoteException,            javax.ejb.RemoveException</pre>	<pre>remove(java.lang.Object)     throws javax.ejb.RemoveException,            javax.ejb.EJBException</pre>

The remote home interfaces that you write extend the `javax.ejb.EJBHome` interface; your local home interfaces extend the `javax.ejb.EJBLocalHome` interface. Within the home interface, the specification normally requires that you implement at least one `create()` method.

The `create()` and `remove()` methods have very different effects for entity beans than they do for other bean types. As will be seen in the appropriate sections, for entity beans the methods create and delete persistent data (think SQL `INSERT` and `DELETE` statements for RDBMS datastores). For session beans, these methods create (or draw from a pool) and disassociate bean instances. The `create()` methods also ask the container to return the component interface for the requested Enterprise JavaBean instance.

The prototype for a remote home interface `create()` method looks like this:

```
MyEJBRemote create() throws RemoteException,
                CreateException;
```

For a local home interface `create()` method, the code looks like this:

```
MyEJBLocal create() throws CreateException;
```

You may notice later that there is no direct corresponding `create()` method in your bean. Instead, your bean defines, per the specification, an `ejbCreate()` method to match each interface `create()` method, which the container calls *before* persisting new data for entity beans and *after* bean instantiation for the other EJB component types. The `ejbCreate()` method for session and message-driven beans may be viewed as similar to the `init()` method in applets and servlets.

The pattern of pairs of `methodName()` and `ejbMethodName()` is repeated often for bean contract methods, with `methodName()` used for the client interface and `ejbMethodName()` used in the bean's implementation. Remember that the container always calls the bean's methods, so it knows that it needs to preface the method with `ejb`. See the relevant bean contract sections in the EJB 2.0 specification for complete information (available from [Resources](#)).

## The component interface

*Note: The following discussion of bean interfaces applies only to session and entity beans. Message-driven beans effectively run within a batch environment on the server and are represented only by the bean class -- no interfaces are used.*

Enterprise JavaBean functionality is obtained through the bean's component interface, which defines the business methods visible to, and callable by, the client. Again, the developer writes the component interface, and the container creates the implementation for client interaction.

The client uses a home interface's `create()` method to obtain a reference to a bean's component interface. In the entity bean section, we will see that a component interface may also be returned by `findByPrimaryKey()` and other *finder methods*. The sample code below locates the home interface for a remote bean with a JNDI name of `ejb/MyEJB`, then obtains the component interface by invoking `<HomeInterface>.create()`. This code, and the following code for local home lookup, is virtually identical in every client's EJB access routine:

```
// get the bean's Home interface
InitialContext ic = new InitialContext();
Object oRef =
    ic.lookup( "java:comp/env/ejb/MyEJBBean" );
MyEJBRemoteHome MyEJBHome =
    (MyEJBRemoteHome)PortableRemoteObject.
        narrow( oRef, MyEJBRemoteHome.class );

// get the component interface
MyEJBRemote MyEJB = MyEJBHome.create();
```

Again, note that `PortableRemoteObject.narrow()` must be used on the object returned from the JNDI lookup rather than Java language casts for remote types.

For a bean that provides a local home and component interface, the code looks like this:

```
// get the EJB's Home interface
InitialContext ic = new InitialContext();
MyEJBLocalHome MyEJBHome =
    (MyEJBHome)ic.lookup( "java:comp/env/ejb/MyEJBBean" );

// get the component interface
MyEJBLocal MyEJB = MyEJBHome.create();
```

**Figure 2. The javax.ejb.EJBObject and javax.ejb.EJBLocalObject interfaces**

«interface» EJBObject extends java.rmi.Remote	«interface» EJBLocalObject
getEJBHome() : javax.ejb.EJBHome throws java.rmi.RemoteException	getEJBLocalHome() : javax.ejb.EJBLocalHome throws javax.ejb.EJBException
getHandle() : javax.ejb.Handle throws java.rmi.RemoteException	getPrimaryKey() : java.lang.Object throws javax.ejb.EJBException
getPrimaryKey() : java.lang.Object throws java.rmi.RemoteException	isIdentical( javax.ejb.EJBLocalObject ) : boolean throws javax.ejb.EJBException
isIdentical( javax.ejb.EJBObject ) : boolean throws java.rmi.RemoteException	remove() throws javax.ejb.RemoveException, javax.ejb.EJBException
remove() throws java.rmi.RemoteException, javax.ejb.RemoveException	

When writing a remote interface, your interface extends `javax.ejb.EJBObject`. When writing a local interface, your interface extends `javax.ejb.EJBLocalObject`. Remember that the component interface is the client's view of your bean's functionality. Therefore, this is the place where you define all of the methods that should be available to the client.

The component interface's `remove()` method disengages the current bean from the client for session beans but *deletes* data from the datastore for entity beans. The life cycle and associated methods are discussed more completely for each bean type in the relevant sections ahead.

## Section 3. Session beans

### Session beans overview

*Session beans* are reusable components that have conversations or sessions with clients. In many ways, session beans are the workhorses of the EJB component family, at various times providing direct functionality, managing or controlling interaction with resources and other beans, and acting as a facade or liaison for cooperating beans. Their lifetimes and life cycles are relatively short and dependent on the client. Session beans are non-persistent and live no longer than the client and possibly less. Under the hood, the container may actually reuse session beans, but from the client's perspective, once they are gone, they are permanently gone. There are two types of session beans, *stateless* and *stateful*, and they have distinctly different types of relationships with their clients. The container is made aware of the type of the session bean through a deployment descriptor entry.

**Figure 3. The javax.ejb.SessionBean interface**



When you write any session bean, you must implement the `javax.ejb.SessionBean` interface shown in Figure 3. All beans have an associated context; for session beans, the container calls `setSessionContext()` after instance creation, which allows the bean to obtain a reference to the session context. The `ejbActivate()` and `ejbPassivate()` methods are only invoked on stateful session beans. `ejbRemove()` is called at the end of a session bean's lifetime for any cleanup before destruction.

## Stateless session beans

Stateless session beans do not maintain conversational state with any given client for longer than the duration of a specific method call. From the client perspective, then, a stateless session bean may be seen as a group of methods that use only local variables, remembering nothing from previous client interaction. In addition, the container may choose to serve the same instance of a stateless session bean to multiple clients or use different bean instances at different times to service one client's requests: the container sees every instance of a specific session bean type as equivalent. One implication of the way stateless session beans work is that there can only be a single `ejbCreate()` method, with no arguments. A container may choose to *passivate* (put to secondary storage) a stateful session bean, but never does so with stateless session beans. Because of these characteristics, stateless session beans can offer greater scalability and performance than most other types of beans.

There are only two milestones in a stateless session bean's life cycle: Does Not Exist and Ready. A container may maintain a *method-ready pool*, which is a pool of active, ready-to-serve beans. To place a session bean in the Ready state, the container:

- Instantiates a bean using the `Class.newInstance()` method
- Invokes the bean's `setSessionContext()` method
- Invokes the bean's single `ejbCreate()` method

When the container no longer requires the bean, it calls the bean's `ejbRemove()` method. After this action, the bean is effectively destroyed and back to the Does Not Exist state.

Note that there is not necessarily any correlation in time between a *client's* `create()` and `remove()` calls and a *container's* `ejbCreate()` and `ejbRemove()` calls. Also, the bean class must have a public, no-arg constructor to accommodate `Class.newInstance()` invocations.

It is important to understand that stateless session beans can maintain state (that is, use instance variables) among *internal* method invocations, for non-client-related

data, or for data and references that apply to all clients as opposed to a specific client. If the client invokes bean method A, for example, and A then calls the bean's B method, which calls the bean's C method, the state from method A through method C can be maintained. Once method A returns, however, that state is no longer reliable for the given client.

## Example: A metric conversion program

Our first example uses a stateless session bean and a Web component to provide an application that performs various conversions between U.S. English and metric measures. It might come in handy for a salesman working for a U.S. company in Europe or similar scenarios. For a look at the display and user view of the application, see [The Metric Converter application](#). You should also read about and deploy the Alice's World page in order for everything to work properly; details are at [Appendix C: About the example applications](#).

Stateless session beans do not have much code to interact with the container. In this example, you will see that the `SessionBean` implementation is all empty methods. As to the functional part of `MetricCvtBean`, there is a `convert()` method that takes a `String` to identify the type of conversion desired, and a `double`, which is the value to convert. All of the other methods convert to or from the various units. The most important thing about the code, as far as EJB components are concerned, is that the class implements the `SessionBean` interface. Here's an abbreviated look at the `MetricCvtBean` code:

```
import javax.ejb.*;

public class MetricCvtBean implements SessionBean,
                                     MetricCvtConstants
{
    public MetricCvtBean() {} // end constructor

    // required by the specification contract
    public void ejbCreate() {} // end ejbCreate

    // required for SessionBean implementation
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext sc) {}

    public String Convert( String sType, double dToConvert )
    {
        ...
    } // end Convert

    // other methods
    ...
} // end class MetricCvtBean
```

Now that we have the main bean code, it's time to create the home and component

interfaces. The Metric Converter Web component is written to remote interfaces. Here's the abbreviated remote component interface code:

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

public interface MetricCvtRemote extends EJBObject,
                                         MetricCvtConstants
{
    public String Convert( String sType, double dToConvert )
        throws RemoteException;

    // other method declarations
    ...
} // end MetricCvtRemote
```

The important features of the remote component interface are that it extends `EJBObject` and that every method is declared to throw a `RemoteException`. In contrast, a local component interface extends `EJBLocalObject` and does not throw `RemoteExceptions`; the code looks very much like a standard Java interface.

Next, we'll create the remote home interface:

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

public interface MetricCvtRemoteHome extends EJBHome
{
    // required
    MetricCvtRemote create() throws RemoteException,
        CreateException;
} // end MetricCvtRemoteHome
```

The important things here are that the interface extends `EJBHome`, that the `create()` method returns the remote component interface, and that the method throws `RemoteException` and `CreateException`. A local home interface would extend `EJBLocalHome`, its `create()` method would return the local component interface, and the method would throw only `CreateException`.

Now it's time to compile everything. You'll notice that the J2EE documentation recommends using Ant, and you will be well rewarded for learning how to use the tool; but this tutorial can only cover so much, so we will rely on the familiar and standard `javac` command. Go to the `MetricCvt` directory, and from the command line enter `javac *.java` to compile all `.java` files in the directory. If you prefer, you can copy the `.class` files from the `MetricCvt/prod` directory.

Next, we want to take a quick look at the client code (the client is a JSP page) to access and use the bean. Following are the relevant portions:

```

private MetricCvtRemote mc = null;

public void jspInit()
{
    InitialContext ic;
    MetricCvtRemoteHome mcHome;
    Object oRef;
    String sMsg = "Couldn't create MetricCvtBean.";

    try
    {
        ic = new InitialContext();
        oRef =
            ic.lookup( "java:comp/env/ejb/MetricCvtBean" );
        mcHome =
            (MetricCvtRemoteHome)PortableRemoteObject.
                narrow( oRef, MetricCvtRemoteHome.class );
        mc = mcHome.create();
    }
    catch( RemoteException re )
    {
        System.out.println( sMsg +
            re.getMessage() );
    }
    catch( CreateException ce )
    {
        System.out.println( sMsg +
            ce.getMessage() );
    }
    catch( NamingException ne )
    {
        System.out.println( "Unable to lookup home: " +
            "MetricCvtBean. " + ne.getMessage() );
    }
} // end jspInit

public void jspDestroy()
{
    try
    {
        mc.remove();
    }
    catch( RemoteException remoteEx )
    {
        // don't care
    }
    catch( RemoveException removeEx )
    {
        // don't care
    }

    mc = null;
} // end jspDestroy

```

In `jspInit()`, the code creates an `InitialContext` and then does a JNDI lookup for `MetricCvtBean`. It then retrieves the home interface and uses the `create()` method to get a reference to the component interface. In `jspDestroy()`, the bean is removed and the reference is set to null.

A client uses the retrieved component interface to invoke exposed bean methods. The actual use of the bean in the Metric Converter is to display its output from the `Convert()` method, as shown by the following statement:

```
<%= mc.Convert( sRB, d ) %>
```

## Deploying the Metric Converter example

If you deployed the Alice's World page (see [Appendix C: About the example applications](#)), you have an idea of the steps involved in deployment. If you haven't, you should do so now, because the Metric Converter application expects to return to that page. We are going to go through the deployment process step by step for both the Enterprise JavaBean component and the Web component here. For the other examples, I recommend that you copy the .ear file from the associated prod directory, open it in `deploytool`, and deploy from that (in other words, start at the step labeled "Deployment" below), unless you want more deployment practice. Another good way to practice is to copy an application directory to one of your own, then create the application from scratch with a new name, using the original .ear file for reference.

### The `MetricCvtBean` deployment:

1. Start J2EE and `deploytool`.
2. Create new application:
  - From the menu, select File, New, Application.
  - Browse to the `MetricCvt` folder.
  - Key `MetricCvt.ear` in for the file name.
  - Click New Application.
  - Application Display name is `MetricCvt` by default.
  - Click OK.
3. Create new Enterprise JavaBean component:
  - Ensure that Application `MetricCvt` is selected.
  - From the menu, select File, New, Enterprise Bean -- Intro display appears, click Next.
  - Select Create New JAR File In Application.
  - Ensure that `MetricCvt` is selected in the drop-down menu.
  - In JAR Display Name, key `MetricCvtJAR`.
  - Click Edit.

- Select the .class files that comprise the bean. These are:
    - MetricCvtBean.class
    - MetricCvtConstants.class
    - MetricCvtRemote.class
    - MetricCvtRemoteHome.class
  - Click Add; click OK; click Next.
  - Under Bean Type, click Session and Stateless.
  - In the Enterprise Bean Class combo box, select the bean implementation: MetricCvtBean.
  - In Enterprise Bean Name, accept MetricCvtBean.
  - Select the corresponding interfaces in the combo boxes:
    - For Remote Home Interface, select: MetricCvtRemoteHome.
    - For Remote Interface, select: MetricCvtRemote.
  - Click Next. You may continue with Next to look at the additional displays until you get to Finish, or click Finish now. For this bean, there are no more entries.
  - MetricCvtJAR and MetricCvtBean now appear under MetricCvt.
  - Select MetricCvtJAR, then click the JNDI Names tab.
  - Under EJBs, key ejb/MetricCvtBean next to MetricCvtBean.
4. **Deployment.** All of the preceding has been set up for the deployment descriptors. We can now actually deploy the bean:
- Select MetricCvt.
  - Select Tools Deploy from the menu.
  - Under Object To Deploy, ensure that MetricCvt is selected.
  - Under Target Server, select localhost.
  - Select the Return Client JAR check box. While our Web component won't need it, any remote clients would use this JAR for the necessary RMI/IIOP stubs.
  - Ensure that the complete path shows for MetricCvtClient.jar in the Client JAR File Name field.
  - Select Save object before deploying.

- Click Next.
- Verify that the JNDI name is `ejb/MetricCvtBean`.
- Click Next, then Finish.

The Deployment Progress dialog will display. Wait until the progress bars are complete and the Cancel button changes to OK, then click OK. The MetricCvt Enterprise JavaBean component is deployed.

### **The Metric Converter Web component deployment:**

1. Start J2EE and `deploytool`.
2. Create new application:
  - From the menu, select File, New, Application.
  - Browse to the MetricCvt folder.
  - Key `MetricCvtApp.ear` in file name.
  - Click New Application.
  - Application Display name is `MetricCvtApp` by default.
  - Click OK.
3. Create new Web component:
  - Ensure that Application `MetricCvtApp` is selected.
  - From the menu, select File, New, Web Component -- Intro display appears, click Next.
  - Select Create New WAR File In Application.
  - Ensure that `MetricCvtApp` is selected in the drop-down menu.
  - In WAR Display Name, key `MetricCvtAppWAR`.
  - Click Edit.
  - Select the `.jsp` file and the bean interface `.class` files needed for the JSP files compilation. These are:
    - `index.jsp`
    - `MetricCvtConstants.class`
    - `MetricCvtRemote.class`

- MetricCvtRemoteHome.class
  - Click Add, Click OK, Click Next.
  - Select JSP for the type of Web component; click Next.
  - In the JSP filename combo box, select index.jsp.
  - In Web Component Name, allow the default -- index.
  - Click Next. There are no Initialization Parameters. Click Next.
  - Under Component Aliases, enter "/index" (without quotes).
  - You may continue with Next to look at the additional displays until you get to Finish, or click Finish now.
  - MetricCvtAppWAR now appears under MetricCvtApp.
  - Select MetricCvtAppWAR, then select the EJB Refs tab and click Add.
  - When it invokes the lookup method, the Web client refers to the home of an enterprise bean like this:  

```
Object objRef =  
ic.lookup( "java:comp/env/ejb/MetricCvtBean" );
```

  
So, in the Coded Name column, enter ejb/MetricCvtBean.
  - Accept Session for Type and Remote for Interface.
  - For Home Interface, enter MetricCvtRemoteHome.
  - In the Local/Remote Interface column, enter MetricCvtRemote.
  - Now select MetricCvtApp and click on the JNDI Names tab.
  - Under References, key ejb/MetricCvtBean for the JNDI name next to the ejb/MetricCvtBean Reference Name. This maps the actual JNDI name for the bean to the name used in the application code.
4. **Deployment.** We can now actually deploy the Web component:
- Select MetricCvtApp.
  - Select Tools, Deploy from the menu.
  - Under Object To Deploy, ensure that MetricCvtApp is selected.
  - Under Target Server, we are using localhost. Be sure that is selected.
  - Unselect the Return Client JAR check box. In this case, the JSP page *is* the client.
  - Select Save object before deploying.

- Click Next.
- Verify that the JNDI name under References is `ejb/MetricCvtBean`; click Next.
- Key `"/MetricCvtApp"` in the context root.
- Click Next, then Finish.

The Deployment Progress dialog will display. Wait until the progress bars are complete and the Cancel button changes to OK, then click OK. The MetricCvtApp Web component is deployed.

If you look in the MetricCvt folder, you will see that it now contains MetricCvt.ear, MetricCvtApp.ear, and MetricCvtClient.jar. To run MetricCvtApp, start your browser and enter `http://localhost:8000/Alice` as the target URL. When the Alice's World home page appears, click on the "Alice's Metric Converter" link. The JSP page must be compiled, so the first run will appear to take a while to load. You can also run the application by entering the URL

`http://localhost:8000/MetricCvtApp`, but the Done button expects to return to the home page, so you should start from there.

## Stateful session beans

Stateful session beans are assigned to a specific client and are able to retain session state across multiple requests from that client for the duration of a bean's lifetime. Because of the link to the client and the need to retain state, stateful session beans are not as scalable as the stateless variety, although the container can still use pooling. You should consider the importance to your application of maintaining state externally, as opposed to maintaining state in the client, when deciding whether to use stateless or stateful session beans.

Other than the required life cycle methods, stateful session beans will look very similar to standard Java objects with instance variables. A public, no-arg constructor is required. There are three milestones in a stateful session bean's life cycle: Does Not Exist, Ready, and Passive. To place a session bean in the Ready state, the container:

- Instantiates a bean using the `Class.newInstance()` method
- Invokes the bean's `setSessionContext()` method
- Invokes the bean's appropriate `ejbCreate()` method

There may be multiple `ejbCreate()` methods with differing input arguments, just as an ordinary Java class might have multiple constructors.

Prior to placing a bean in the Passive state, the container invokes `ejbPassivate()`. Before placing the bean back in the Ready state, the container calls `ejbActivate()`.

When the container no longer requires the bean, it calls the bean's `ejbRemove()` method. After this operation, the bean is effectively destroyed and back to the Does Not Exist state.

A container may use passivation to enhance scalability. While stateless session beans are relatively easy to pool, stateful session beans must appear to be linked to a single client and maintain state with that client. One way to accomplish a semblance of pooling is to use something similar to context switching in an operating system's application management environment. Serialization/deserialization is most commonly employed to save/restore bean state to and from secondary storage. When the container decides the time is appropriate, it will invoke `ejbPassivate()` just before serialization and `ejbActivate()` just after deserialization. The bean should be prepared to release or reacquire, respectively, resources that are not suitable for serialization (such as a database connection) when these methods are called. If there are no serialization, performance, or efficiency implications, the bean may choose to do nothing at these times. Clearly, any acquired resources should also be released on an `ejbRemove()` invocation.

## Example: The Rock Survey, take 1

At this point, you should take a moment to look over [The Rock Survey application](#), because we will revisit it several times in the tutorial. As you can see, the application requires that data be maintained across four Web pages, so a stateful session bean is the right choice to manage and maintain state and data. In this first go-around, the data is not actually persisted anywhere after the client completes the survey; we'll handle that in later iterations. In anticipation, the `persist()` method exists, but for the moment it does nothing more than return `true`.

The `RockSurveyBean` maintains and validates Rock Survey data, and returns a data object that also contains error information for the client's use. As with the stateless session bean example, `RockSurveyBean` implements the `SessionBean` interface. Let's take a look:

```
import javax.ejb.*;

public class RockSurveyBean implements SessionBean
{
    ...

    public RockSurveyBean() {} // end constructor

    // required by the specification contract
    public void ejbCreate() {} // end ejbCreate
}
```

```
// required for SessionBean implementation
public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void setSessionContext(SessionContext sc) {}

...

} // end class RockSurveyBean
```

For the Rock Survey application, only remote interfaces are defined. The extension of the remote home interface is very basic, with only one `create()` method:

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

public interface RockSurveyRemoteHome extends EJBHome
{
    // required
    RockSurveyRemote create()
        throws RemoteException,
        CreateException;
} // end RockSurveyRemoteHome
```

The `create()` and `remove()` methods, except for the name of the bean (and the component interface returned from `create()`), are identical to those for the client in [Example: A metric conversion program](#). There are two things to note in the application.

First, in [Example: A metric conversion program](#), the *stateless* session bean was created in `jspInit()` and destroyed in `jspDestroy()`, each of which occurs exactly once in the life of a servlet. That's because the application made use of the same bean (from its perspective) for the life of the application. Here, a *stateful* session bean is allocated to, and tracks state for, the duration of the entire conversation for each individual client, by definition. For that reason, on the first page the code checks if the servlet session object was newly created, and, if so, creates a new `RockSurveyBean`. The bean is destroyed when the session is invalidated at the end of the survey.

Secondly, the Rock Survey application spans several pages, so there must be a means of retaining the reference to the `RockSurveyBean` as the user moves from page to page. To resolve this issue, the bean reference is stored in and retrieved from the servlet's session object.

Next up is the remote component interface:

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

public interface RockSurveyRemote extends EJBObject
```

```
{
    public RockSurveyData getData()
        throws RemoteException;

    public boolean persist()
        throws RemoteException;

    public RockSurveyData setUserData(
        String sGenderIn, String sLastNameIn,
        String sMaritalIn, String sPostalIn )
        throws RemoteException;

    public RockSurveyData setRockData(
        String sCountIn, String sLocationIn,
        String sMeasureIn, String sPreferenceIn,
        String sWeightIn )
        throws RemoteException;
} // end RockSurveyRemote
```

Although `RockSurveyBean` has several methods, the component interface only exposes those the client needs to get, set, and persist the data. `RockSurveyData` is a helper class that effectively contains a read-only copy of the data, with a complete set of `getXXX()` methods, and any errors. Because the data is gathered over several pages, there are two `setXXX()` methods, one for general user information, and the other for the actual survey data. The `RockSurveyData` class always returns all of the currently available data.

You may wonder why I went to the trouble of creating the `RockSurveyData` class when all of the information is available from `RockSurveyBean`. The answer has to do with a topic you will hear discussed frequently in EJB developer circles: *fine-grained* versus *coarse-grained* access. The classic example of fine-grained access is a data object representing a row with, say, 50 columns and having 50 getters and 50 setters. A coarse-grained version might return the entire row or multiple rows from one method.

Because the Rock Survey uses remote interfaces, `RockSurveyBean` is a remote component, which means each method invocation is a distributed call, as discussed earlier. Fine-grained access with distributed calls is a performance killer. Since Web page data retrieval is inherently page oriented, `RockSurveyBean`'s gets and sets are oriented toward a page of data. The `RockSurveyData` class is the vehicle to carry the data in a coarse-grained manner. Once the application has a `RockSurveyData` instance, it can call `getXXX()` as usual, but against a local object, avoiding performance hits as much as possible.

## Deploying the Rock Survey example

Deployment for the Rock Survey application will look very similar to [Deploying the Metric Converter example](#). The major differences are the names, the fact that we are using a stateful session bean, and the fact that the bean and the Web component pieces are in the same EAR. Because we will deploy several versions of this

application using components with the same name, we will use Survey1 for the Application EAR, WAR, and JAR files. Here are the directions:

### The Survey1 application deployment:

1. Compile Java source files in the Survey1 directory, or copy the .class files from the prod folder.
2. Start J2EE and `deploytool`.
3. Create new application:
  - From the menu, select File, New, Application.
  - Browse to the Survey1 folder.
  - Key Survey1App.ear in for the file name.
  - Click New Application.
  - Application Display name is Survey1App by default; click OK.
4. Create new Enterprise JavaBean component:
  - Ensure that Application Survey1 is selected.
  - From the menu, select File, New, Enterprise Bean -- Intro display appears; click Next.
  - Select Create New JAR File In Application.
  - Ensure that Survey1 is selected in the drop-down menu.
  - In JAR Display Name, key Survey1JAR.
  - Click Edit.
  - Select the .class files that comprise the bean. These are:
    - RockSurveyBean.class
    - RockSurveyConstants.class
    - RockSurveyData.class
    - RockSurveyRemote.class
    - RockSurveyRemoteHome.class
    - SurveyConstants.class
  - Click Add; click OK; click Next.

- Under Bean Type, click Session and Stateful.
  - In the Enterprise Bean Class combo box, select the bean implementation: RockSurveyBean.
  - In Enterprise Bean Name, key RockSurvey1Bean.
  - Select the corresponding interfaces in the combo boxes:
    - For Remote Home Interface, select: RockSurveyRemoteHome.
    - For Remote Interface, select: RockSurveyRemote.
  - Click Next, click Finish.
  - Survey1JAR and RockSurvey1Bean now appear under Survey1App.
  - Select Survey1JAR, then click the JNDI Names tab.
  - Under EJBs, key ejb/RockSurvey1Bean next to RockSurvey1Bean.
5. Create new Web component:
- Ensure that Application Survey1App is selected.
  - From the menu, select File, New, Web Component -- Intro display appears; click Next.
  - Select Create New WAR File In Application.
  - Ensure that Survey1App is selected in the drop-down menu.
  - In WAR Display Name, key Survey1WAR.
  - Click Edit.
  - Select the files needed for the component. These are:
    - The images directory
    - index.jsp
    - RockSurvey.jsp
    - RockSurveyExit.jsp
    - SurveyWelcome.jsp
  - Click Add; click OK; click Next.
  - Select JSP for the type of Web component, then click Next.
  - In the JSP filename combo box, select index.jsp.
  - In Web Component Name, keep the default -- index.

- Click Next. There are no Initialization Parameters. Click Next.
- Under Component Aliases, click Add, enter `"/index"` (without quotes).
- Click Next; click Finish.
- Survey1WAR now appears under Survey1App.
- Select Survey1WAR, then select the EJB Refs tab and click Add.
- When it invokes the `lookup()` method, the Web client refers to the home of an enterprise bean like this:  

```
Object objRef =  
ic.lookup("java:comp/env/ejb/RockSurveyBean");
```

So, in the Coded Name column, enter `ejb/RockSurveyBean`.
- Accept Session for Type and Remote for Interface.
- For Home Interface, enter `RockSurveyRemoteHome`.
- In the Local/Remote Interface column, enter `RockSurveyRemote`.
- At the bottom, select `ejb-jar-ic.jar#RockSurvey1Bean` from the Enterprise Bean Name drop-down menu. Then click the JNDI Name button and select `ejb/RockSurvey1Bean` from the drop-down menu. This maps the actual JNDI name for the bean to the name used in the application code.

6. **Deployment.** We can now actually deploy the application:

- Select Survey1App.
- Select Tools, Deploy from the menu.
- Under Object To Deploy, ensure that Survey1App is selected.
- Under Target Server, we are using localhost. Be sure that is selected.
- Select the Return Client JAR check box.
- Select Save object before deploying.
- Click Next.
- Verify that the JNDI name under both Application and References is `ejb/RockSurvey1Bean`, then click Next.
- Key `"/Survey1App"` in the context root.
- Click Next, then Finish.

The Deployment Progress dialog will display. Wait until the progress bars are complete and the Cancel button changes to OK, then click OK. The Survey1App

application is deployed.

If you look in the Survey1 folder, you will see that it now contains Survey1App.ear and Survey1AppClient.jar. To run the Survey1App, start your browser and enter `http://localhost:8000/Alice` as the target URL. When the Alice's World home page appears, click on the "Alice's Surveys - Take 1" link. The JSP page must be compiled, so the first run will appear to take a while to load. You can also run the application by using the URL `http://localhost:8000/Survey1App`, but the Done button expects to return to the home page, so you should start from there.

---

## Section 4. Entity beans

### Entity beans overview

Session beans are all about functionality, application logic, and application state. Entity beans, on the other hand, represent persistent business entities or data. In a typical project requirements document, session beans would model the tasks or verbs, while entity beans would model the nouns: people, places, and things. While an entity bean usually represents a row from a database table, the datastore can be any persistence mechanism that the container or developer can manage, including incoming data from other applications (legacy connector applications, for example). Note that for read-only applications, entity beans may be overkill and raise performance issues. For these applications, you may want to consider direct access from the client or, more likely, a session bean using JDBC or other access methods as possible alternatives.

The specification mandates that there normally must be a `create() / ejbCreate()` method pair, an `ejbPostCreate()` method with matching arguments for each `ejbCreate()` method, and always a `findByPrimaryKey() / ejbFindByPrimaryKey()` method pair.

As mentioned earlier, `create() / ejbCreate()` and `remove() / ejbRemove()` have completely different functions with entity beans: `create()` creates persistent data, usually meaning the insertion of a row into a database table, and `remove()` deletes the persistent data. In this instance, the `create()` method will have at least a primary key argument and normally will have several data initialization arguments as well.

The `findByPrimaryKey()` method, like the `create()` method, returns a component interface and is the means by which a client obtains an entity bean to represent existing data. There may be multiple *finder methods* based on different

values, each of which must return either a component interface or a Collection of component interfaces. Finder methods are named, again in matching sets, `findByXXX()` and `ejbFindByXXX()` with appropriate arguments. The `ejbFindByPrimaryKey()` method, like the `ejbCreate()` method, returns the primary key; other `ejbFindXXX()` methods may return a single primary key or a Collection of primary keys. The container handles the magic of converting from primary key to component interface instance in the `create()` method (but see the differences between BMP and CMP in Table 1 below).

A *primary key* is a unique identifier for a bean instance and is a key to the data, usually correlating to the primary key for a row in a table. A primary key must be an `Object`; primitives are not allowed. If no existing class can serve as a primary key or if the primary key consists of more than one variable (or column, if you prefer), then a custom class must be written to represent the key, with instance variables acting as composite key components. To work with both container-managed and bean-managed persistence, a primary key class must meet these guidelines:

- The class must be public, and must have a public constructor with no parameters.
- All fields in the primary key class must be declared as public.
- For CMP, variable names in the class must be a subset of the names of container-managed fields.
- The class must provide specific `hashCode()` and `equals(Object obj)` methods.

In addition to `create`, `finder`, and `remove` methods, an entity bean may expose *home methods* in its home interface. Home methods provide logic for operations that are not specific to a unique bean instance. This usually means an operation that requires working with more than one instance to obtain the desired result. Some examples include counting U.S. customers with a zip code of 80110; imposing a late charge on all accounts 30 days overdue; or calculating sales totals for a specific region. Home methods may have an arbitrary name in the home interface. In the bean implementation, the corresponding method must match arguments and have a name composed of the prefix `ejbHome` plus the name given in the home interface with the initial letter capitalized. For instance, a home interface home method defined as `int countCustomers( String country, String postalCode )` would, in the bean implementation, be written as `int ejbHomeCountCustomers( String country, String postalCode )`.

Entity beans may be pooled in a manner similar to stateful session beans, but in this case passivation/activation also causes the data to be persisted to and read from the datastore, respectively; all other resources must be released and reacquired, as appropriate.

An entity bean provides shared access to its data, and it is possible that the same data may be modified by multiple clients. In this sort of scenario, it becomes critical that beans can work within the proven methodology of transactions (see [What are transactions?](#)). The container provides for both bean-managed and container-managed demarcated transactions; the type of transaction management and transactional attributes are declared in the deployment descriptor. *Entity beans must use container-managed transactions.*

As previously noted, there are two methods of persistence available: bean-managed persistence (BMP), which is implemented by the developer, and container-managed persistence (CMP), which is implemented by the container. Both of these persistence methods are discussed in the following sections and illustrated in the example application. In each case, the container controls the timing of inserts, updates, refreshes/synchronization, and deletes.

While the specifics of BMP and CMP will be covered shortly, Table 1 (taken from Chapter 6 of the "J2EE Tutorial" -- see [Resources](#)) is a useful quick reference for responsibility and coding differences between the two persistence types.

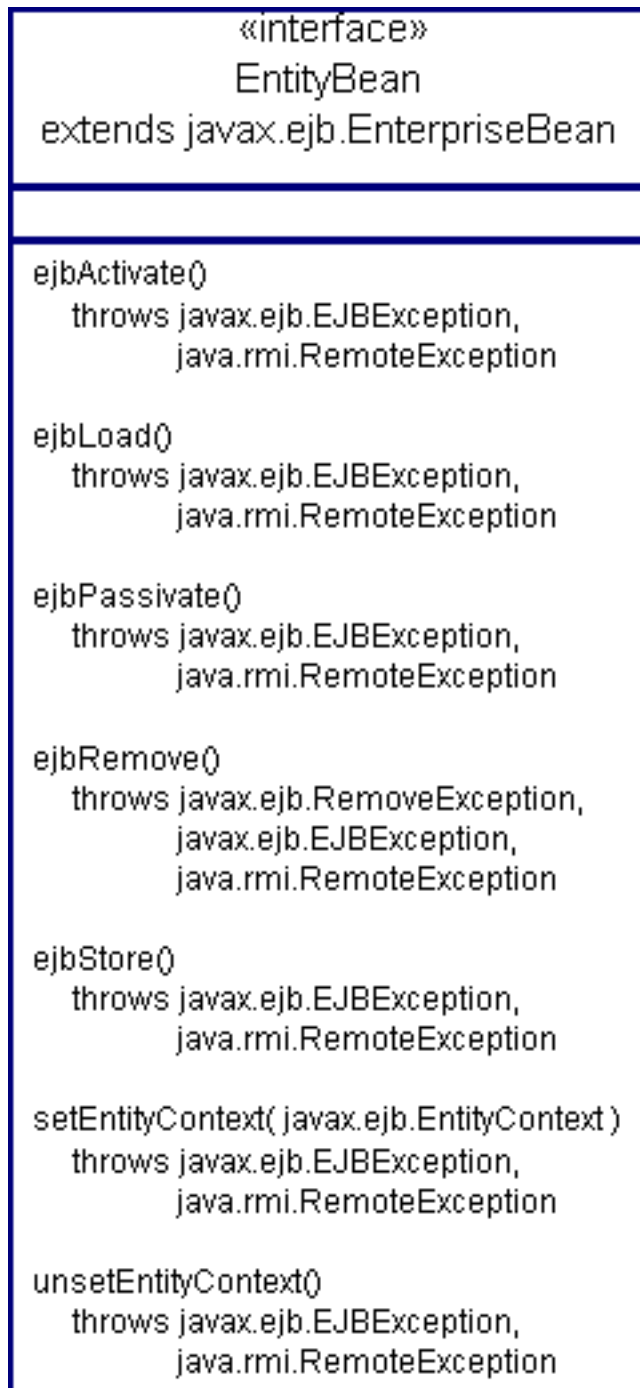
**Table 1. BMP/CMP differences**

Difference	Container-managed	Bean-managed
Class definition	Abstract	Not abstract
Database access calls	Generated by tools	Coded by developers
Persistent state	Represented by virtual persistent fields	Coded as instance variables
Access methods for persistent and relationship fields	Required	None
<code>findByPrimaryKey</code> method	Handled by container	Coded by developers
Customized finder methods	Handled by container, but the developer must define the EJB QL queries	Coded by developers
Select methods	Handled by container	None
Return value of <code>ejbCreate</code>	Should be null	Must be the primary key

## The entity bean life cycle

The milestones in an entity bean's life cycle are Does Not Exist, Pooled, and Ready. In the Pooled state, the instance is not associated with any particular entity. In the Ready state, the instance has been drawn from the pool, associated with a specific entity, and assigned a unique object identity.

**Figure 4. The `javax.ejb.EntityBean` interface**



Every entity bean implements the `javax.ejb.EntityBean` interface. It should be evident from Figure 4 that there are several more life cycle methods here than there are in the `SessionBean` interface, and it is usually the case that much more attention must be paid to them, particularly for BMP.

To start an entity bean's life cycle, the container invokes `newInstance()`, then

sends an `EntityContext` to the bean using the `setEntityContext()` callback method. At that point, the bean instance is moved into a bean pool. While in the pool, an instance may be used to execute the bean's `ejbFindByPrimaryKey()`, other finder methods, or `ejbHomeXXX()` methods. The instance does not move to the Ready state on finder method execution; for `ejbHome` methods, the bean is returned to the pool after execution.

An entity instance moves to the Ready state either when the `ejbCreate()` and `ejbPostCreate()` method pair is called, or when the `ejbActivate()` method is called. While in the Ready state, an instance has a specific object identity.

During a bean's life cycle, the container may call `ejbLoad()` and `ejbStore()` at any time and in any order it deems necessary to read from and write to the persistent datastore.

The container may also call `ejbPassivate()` at any time to move the instance back into the pool. Generally, the `ejbStore()` method will be invoked just prior to this call. As with any passivation method, resources should be released and reacquired upon activation.

An instance will be moved back to the pool if a transaction rollback occurs during `ejbCreate()` or `ejbPostCreate()` operations. `ejbRemove()` invocations will result in transition to the pool as well.

An instance is removed from the pool and goes to the Does Not Exist state immediately after the container calls `unsetEntityContext()`, so this is the time to perform any final clean-up operations.

## What are transactions?

Transaction technology was developed primarily in the database area, and a transaction generally means one or more statements that comprise a logical unit of work (known as an LUW). In this context, the term *transaction* is used to mean an all-or-nothing series of operations; that is, everything should complete successfully or nothing should, in which case all state should revert to the state that existed prior to the transaction start. The classic example of a transaction involves withdrawing money from one bank account and depositing it in another. If only the withdrawal completes, money is lost. Debits and credits in a double-entry accounting system provide another example: both the debit and credit must complete; otherwise, it should appear that neither operation was attempted. Once a transaction begins, it continues until a `commit` (make the changes persistent) or a `rollback` (revert to the previous state) is issued.

You should understand that transaction handlers only control the results on persistent data. If your application's variables change during the course of a

transaction that is eventually rolled back, you will have to write your own code to revert those variables to the previous state.

Transactions also allow concurrent, multiuser operations to appear as if they occurred serially. *Isolation levels* control the degree of concurrency and how transactions may affect each other. JDBC, the Java API for working with relational databases, defines the following isolation levels:

- TRANSACTION\_NONE
- TRANSACTION\_READ\_COMMITTED
- TRANSACTION\_READ\_UNCOMMITTED
- TRANSACTION\_REPEATABLE\_READ
- TRANSACTION\_SERIALIZABLE

A client normally may set the isolation level or use the default provided by the JDBC driver. It is important, however, to be aware that entity beans have some transactional restrictions:

- As mentioned in [Entity beans overview](#), while with session and message-driven beans the developer may start and end transactions, entity beans must use container-managed transactions.
- The EJB 2.0 specification does not define any method for container-managed transactions to set the isolation level. As a result, standard container-managed transactions use the driver default.

Other transactional systems may use different isolation levels or contexts. If the default level has to be changed to meet the application's requirements, the developer must write to resource manager-specific APIs to accomplish the task (JDBC is a resource manager).

EJB containers are required to support the Java Transaction API (JTA) and the Connector APIs, providing flat transaction support for all types of beans (see [Controlling transactions](#)). The container may, but is not required to, use a Java Transaction Service (JTS) implementation for propagating transactions across servers. JTS is a Java binding of the CORBA Object Transaction Service (OTS) 1.1 specification, which uses the IIOP protocol. Regardless of the specific transaction technology used, the actual transaction operations are mostly transparent to the EJB developer.

For more information about transactions, JDBC, isolation levels, JTS, and JTA, see Chapter 17, "Support for Transactions," of the Enterprise JavaBean 2.0 specification, along with the other material in [Resources](#).

## Controlling transactions

### Bean-managed transactions

Session and message-driven beans can declare that transactions are bean-managed and use the Java Transaction API directly for transaction demarcation and control. The interface for JTA is `javax.transaction.UserTransaction`, which is accessed through an `EJBContext` object. `EJBContext` is the parent of `EntityContext`, `MessageDrivenContext`, and `SessionContext`, which are obtained from the appropriate `setXXXContext` method. Once the `UserTransaction` object is obtained, the bean can invoke the `begin()`, `commit()`, and `rollback()` methods for transaction control. When using JTA, the developer must not use any resource manager methods for transactions (like JDBC commits, or rollbacks), but instead must use only `UserTransaction` methods.

Typical code to access and use the `UserTransaction` follows:

```
javax.transaction.UserTransaction ut =
    ejbContext.getUserTransaction();
// start the transaction
ut.begin();
// Do some work.
if( aGoodResult )
{
    // commit the transaction
    ut.commit();
}
else
{
    // roll back the transaction
    ut.rollback();
}
```

To enable bean-managed transactions (BMT), the developer must ensure that the deployment descriptor contains a `<transaction-type>Bean</transaction-type>` entry.

### Container-managed transactions

Any bean can use container-managed transaction handling (CMT), which is completely declarative (there is no indication in the code that transactions are being used), by means of a `<transaction-type>Container</transaction-type>` entry in the deployment descriptor. The container's responsibility for CMT is declared in the deployment descriptor with a `<trans-attribute></trans-attribute>` pair enclosing one of the following *transaction attributes*:

- `NotSupported`

- Required
- Supports
- RequiresNew
- Mandatory
- Never

The `Required` attribute is a safe choice available to all EJB components: if a transaction is in progress when an operation is performed, the operation joins it; if not, a new transaction is started. The other attributes provide transactional variations, but are restricted to specific bean types.

A *system* exception will automatically cause a transaction rollback. This is not the case for *application-level* exceptions. If an application exception occurs that affects data integrity or otherwise causes the transaction to be invalid, under CMT the exception handler should invoke the `EJBContext.setRollbackOnly()` method. As noted in the `javax.ejb.EJBContext` API documentation, `EJBContext.setRollbackOnly()` will "mark the current transaction for rollback.... A transaction marked for rollback can never commit. Only enterprise beans with container-managed transactions are allowed to use this method."

For more information about `javax.transaction.UserTransaction` and transaction attributes, see Chapter 17, "Support for Transactions," in the EJB 2.0 specification, and the other JTA/JTS material in [Resources](#).

## Entity beans with bean-managed persistence (BMP)

The primary difference between BMP and CMP is that, under BMP, developers must write their own datastore access and persistence routines and respond appropriately to life cycle related calls and callbacks. The bean must provide a public, no-arg constructor for instance creation (invoked by the container).

The bean implements the following methods (for information on the sequence of events, see [The entity bean life cycle](#)):

- **`public void setEntityContext(EntityContext ec)`**  
The container normally invokes this method exactly once, just after instantiation, to pass the `EntityContext`. If the bean will use any `EntityContext` methods, this is the time to obtain the `EntityContext` and save the reference in an instance variable. In addition, it is a good time to get any resources that will be used for the bean's lifetime. Be aware that the bean has no object identity at this point and is about to be placed into the pool.

- **public void unsetEntityContext()**  
This method is also normally invoked exactly once, just before the bean is terminated and sent into the Does Not Exist state. There is no object identity available during this method. This is the proper time to release *any* resources obtained and to perform any final clean-up.
- **public PrimaryKeyClass ejbCreate(...)**  
This method is invoked sometime after the client calls a home interface `create()` method with the same arguments. Notice that the `ejbCreate()` method returns the primary key, while the (container-implemented) `create()` method returns a component instance.  
The `ejbCreate()` method normally validates any arguments, then *the developer-written code inserts a row into a database (or performs other datastore create operations), and initializes the corresponding instance variables*. In particular, the instance variable or variables representing the primary key must be set at this time.  
It is possible to write a bean without an `ejbCreate()` method, when the class deals only with existing data. In this case, there must be no `create()` methods exposed on the home interface.  
`javax.ejb.CreateException` and `javax.ejb.DuplicateKeyException` are standard, API-provided application exceptions that the method may throw.
- **public void ejbPostCreate(...)**  
There must be a matching `ejbPostCreate()` method for each `ejbCreate()` method. The container calls this method after the `ejbCreate()` invocation so that the bean can perform any necessary post-creation operations. The entity object identity is now available. The method may throw the `javax.ejb.CreateException` standard, API-provided application exception.
- **public <primary key type or collection> ejbFind<METHOD>(...)**  
These methods are called when the client calls a `findXXX()` method with matching arguments. An instance is selected from the pool, then returned to the pool upon method completion; therefore, any resources obtained during the method should be released. *The developer-written code performs datastore search operations in these methods*. The finder methods return a single primary key or a `Collection` of primary keys, depending on whether the argument is expected to return information about a unique entity or multiple entities. For a finder method that returns a `Collection`, an empty `Collection` should be returned if no data matching the input arguments is found.  
`javax.ejb.ObjectNotFoundException` may be thrown for a finder that should return a unique value. `javax.ejb.FinderException` may

be thrown by any finder method that encounters an unexpected error.

- **public void ejbLoad()**  
The container invokes this method when it determines that instance variables representing persistent data must be synchronized with the datastore. *This method may be called at any time.* The `EntityContext` methods are available at this time. *The developer-written code accesses data in a row from a database (or performs other datastore retrieval operations) in this method.*
- **public void ejbStore()**  
The container invokes this method when it determines that the datastore must be synchronized with the bean instance variables that represent persistent data. *This method may be called at any time.* *The developer-written code updates data in a database row (or performs other datastore update operations) in this method.* The developer can rely on instance data to be current during this method call.
- **public void ejbRemove()**  
This method is invoked some time after the client calls a `remove()` method. The `EntityContext` methods are available at this time. *The developer-written code deletes a row from a database (or performs other datastore removal operations) in this method.* On completion, the instance leaves the Ready state and is moved to the pool. Because of the transition, the bean state should be the same as it would be after an `ejbPassivate()` call. A simple way to ensure this result is to invoke `ejbPassivate()` after the deletion is performed (the container does not invoke `ejbPassivate()` after `ejbRemove()`). The method may throw `javax.ejb.RemoveException` if an unexpected error occurs.
- **public void ejbActivate()**  
The container invokes this method when assigning an object identity and moving the instance from the Pooled state to the Ready state. This is a good time to acquire (or reacquire) any resources needed for the specific identity instance. Persistent data *should not* be accessed from this method; the container will call `ejbLoad()` to obtain persisted data. `ejbActivate()` will be called for a previously passivated bean. The container will also transition a bean instance from the pool using this method to service an existing entity, which may not have gone through the `create()` method during the current program run. Therefore, to determine the identity of the data it represents, the bean must call `getPrimaryKey()` on its `EntityContext` and set the instance variables representing the primary key. Other `EntityContext` methods may be called as well. The information associated with the `EntityContext` will be valid until `ejbPassivate()` or `ejbRemove()` is called.

- **public void ejbPassivate()**  
The container calls this method when it decides to transition the instance back to the pool. This could happen at any time. Any identity-specific resources should be released in this method. The `EntityContext` methods are available at this time. Persistent data *should not* be written in this method; the container will call `ejbStore()` to persist data. Because a bean instance can not be certain of the data it represents in the `ejbActivate()` method until it invokes `getPrimaryKey()`, and because the instance variables representing the rest of the entity data will be reset on the ensuing `ejbLoad()` call, it makes sense to set those variables to `null` here, so they will be eligible for garbage collection. Note that, although the `ejbRemove()` method transitions an instance to the pool on completion, `ejbPassivate()` is *not* called.
- **public type ejbHome<METHOD>(...)**  
The important thing to understand about home business methods is that there is no specific object identity during method operations. An instance is selected from the pool and, on method completion, returned to the pool. As with `ejbRemove()`, then, the method should clean up after itself before being returned to the pool. There is no real difference between home methods written for BMP and those written for CMP.

## The Rock Survey database

Now it's time to take a look at the database and tables that will store the Rock Survey data. For the tutorial, we'll use the Cloudscape database engine, which is included with the J2EE RI. To use a database within the J2EE environment, the following steps are necessary:

1. A JDBC driver for the DBMS must be added to J2EE resources.
2. The database to be used in an application must be created or available. The tutorial database will be named `gsejbDB`.
3. A JDBC `DataSource` entry must be added to J2EE resources. The `DataSource` entry for `gsejbDB` will be `jdbc/gsejbDB`.
4. The coded JNDI access and the real JNDI name must be reconciled. This step was actually done when the Alice application was deployed in [Appendix C: About the example applications](#).
5. Finally, the database engine (Cloudscape for the J2EE RI) must be started (see [Appendix A: Installing and running J2EE](#)).

Steps 1 and 3 are accomplished with the J2EE Administration Tool, `j2eeadmin`, which is discussed in the J2EE documentation under "J2EE SDK Tools" (see [Resources](#)). There is no standard for database creation, which is DBMS specific. For Cloudscape, step 1 is included in the J2EE RI as installed; Database creation is an option ( `create=true` ) in the JDBC URL for the connection, so it will be handled in step 3. For step 3, start J2EE, then use `j2eeadmin` from the command line as follows:

```
j2eeadmin -addJdbcDatasource jdbc/gsejbDB jdbc:cloudscape:rmi:gsejbDB;create=true
```

To ensure that the `DataSource` was added, use this command to see the current `DataSource` S:

```
j2eeadmin -listJdbcDatasource
```

The Rock Survey tables are minimal, with no analytical decomposition or normalization, to avoid distraction from the main focus of the tutorial, which is EJB technology. You should note that the "design" is not very flexible and should not be used as a model for production tables.

**Table 2. The table for storing Survey data -- "SurveyBeanTable"**

PK	Name	Type	Description
Yes	Type	VARCHAR	Survey Type = "ROCK"
	SF	INTEGER	Single female count
	MF	INTEGER	Married female count
	SM	INTEGER	Single male count
	MM	INTEGER	Married male count
	USF	INTEGER	US female count
	NUSF	INTEGER	Non-US female count
	USM	INTEGER	US male count
	NUSM	INTEGER	Non-US male count
	Igneous	INTEGER	Igneous count
	Metamorphic	INTEGER	Metamorphic count
	Meteorite	INTEGER	Meteorite count
	Sedimentary	INTEGER	Sedimentary count
	InCountry	INTEGER	My country count
	ExCountry	INTEGER	Another country count
	Island	INTEGER	Island count

OS	INTEGER	Outer space count
Total	INTEGER	Total rocks
Pounds	FLOAT	Total rock weight

The "SurveyBeanTable" table holds all survey results except for last name entries, and will contain exactly one row. The initial set of columns may appear confusing at first, but they just contain the breakdowns by gender, marital status, and residence. The other columns should be self-explanatory. For the Rock Survey example, the "SurveyBeanTable" table will be managed by an entity bean using CMP.

**Table 3. The table for storing survey names -- SurveyNames**

PK	Name	Type	Description
Yes	Type	VARCHAR	Survey type = "ROCK"
Yes	LastName	VARCHAR	Last name given for survey
	Total	INTEGER	Count

Last name data is kept in a separate table that contains a row with a count for each distinct last name. As we'll see in [The Survey Results application](#), the three most frequent last names are displayed with the other results, so we need a way to track them. For the Rock Survey example, the SurveyNames table will be managed by an entity bean using BMP.

## Creating and resetting the Rock Survey tables

The program to create and load the tables, `CreateRSTables`, was actually included when the Alice application (see [Appendix C: About the example applications](#)) was deployed, but could not run against the database until we added the `DataSource` in the previous section. *After starting Cloudscape*, you should be able to run it now by clicking the Go button next to "Reset the Database."

`CreateRSTables` is a helper class rather than a bean and uses JDBC for its task. The program first drops the "SurveyBeanTable" and SurveyNames tables from the database to remove any existing data, then creates them. Next, it loads predefined rows to give us something to work with. This is a straight JDBC application; there's nothing interesting about it from the J2EE/EJB technology standpoint, other than the way it retrieves a connection to the database. As shown in the code below, `CreateRSTables` retrieves a `DataSource` using a JNDI lookup for `java:comp/env/jdbc/gsejdb` -- remember adding that entry in [The Rock Survey database](#)? It then obtains a connection from the `DataSource`:

```
Connection con      = null;
DataSource ds       = null;
InitialContext ic   = null;
Statement stmt      = null;
String sJNDIdbName = "java:comp/env/jdbc/gsejdbDB";
try
{
    ic = new InitialContext();
    ds = (DataSource)ic.lookup( sJNDIdbName );
    con = ds.getConnection();
    stmt = con.createStatement();
}
...

```

## Example: The Rock Survey, take 2

Now that we've actually created a database, and have some knowledge of entity beans and transactions, it's time to add data handling capability to the Rock Survey application. The first go-around (see [Example: The Rock Survey, take 1](#)) handled the user view and tracked the survey through the pages of the Web component, but never actually stored anything at the end of the survey. This portion of take 2 will add a BMP entity bean ( `SurveyNamesBean` ) to deal with data for the `SurveyNames` table. It will also expand the `persist()` method in `RockSurveyBean` to include a transaction manager, and control adding and updating names and name counts.

The primary key for the `SurveyNames` table (see Table 3 in [The Rock Survey database](#)) consists of two `Strings` (SQL type `VARCHAR`). You may recall from [Entity beans overview](#) that a primary key consisting of more than one element requires a custom primary key class. The `SurveyNamesKey` class represents the primary key for `SurveyNames`. The class consists of the two composite key `Strings`, a no-arg and a two-argument constructor, `getXXX` methods for the data, and `equals()` and `hashCode()` methods. The class is used by the `SurveyNamesBean` `create()` and `findByPrimaryKey()` methods.

The `SurveyNamesBean` class performs access and maintenance operations against the `SurveyNames` table. `SurveyNamesBean` is an entity bean with BMP, and implements all of the life cycle methods discussed in [The entity bean life cycle](#) and [Entity beans with bean-managed persistence \(BMP\)](#). In addition, it has getter methods for the `Type`, `LastName`, and `Total` columns. There is also an `incrementTotal()` method, which adds 1 to the current count. The class uses the same technique of connecting to the database that the `CreateRSTables` program uses (see [Creating and resetting the Rock Survey tables](#)), and, as a BMP bean, uses standard SQL for access and updates. As recommended for EJB 2.0 components, the class exposes its methods with local home and component interfaces, as shown below:

```
// the local home interface for SurveyNamesBean
import javax.ejb.*;

public interface SurveyNamesLocalHome extends EJBLocalHome
{
    public SurveyNamesLocal create( SurveyNamesKey snkIn )
        throws CreateException;

    public SurveyNamesLocal findByPrimaryKey(
        SurveyNamesKey snkIn )
        throws FinderException;

    // no home business methods
} // end interface SurveyNamesLocalHome
```

```
// the local component interface for SurveyNamesBean
import javax.ejb.*;

public interface SurveyNamesLocal extends EJBLocalObject
{
    public String getLastName();

    public String getType();

    public int getTotal();

    public void incrementTotal();
} //end interface SurveyNamesLocal
```

SurveyNamesBean is managed by a new version of RockSurveyBean named RockSurvey2Bean. The primary additions are instance variables for the SessionContext and a javax.transaction.UserTransaction, along with variables to access SurveyNamesBean. The persist() method now actually does something, being expanded to handle the creation and updating of SurveyNames data. As you might imagine, there are many details in the code, but the following snippets show the essence of the new operations.

persist() uses the now familiar JNDI lookup with the local interface version to get the SurveyNamesBean home interface:

```
...
ic = new InitialContext();
snlHome =
    (SurveyNamesLocalHome)ic.lookup(
        "java:comp/env/ejb/SurveyNamesBean" );
...
```

The code then obtains a UserTransaction ( ut is a UserTransaction and sc is a SessionContext ) and begins a new transaction:

```
ut = sc.getUserTransaction();
...
```

```
ut.begin(); // begin transaction
```

The specification mandates that entity beans use container-managed transactions, but by using the `Required` transaction attribute, the entity bean adds and updates will join in the transaction initiated by the session bean.

Because the same last names will often be entered into the survey, especially over time, the application takes the approach of first attempting to find existing data for the given last name, using `findByPrimaryKey()`. If successful, the existing count is incremented; if the finder fails, then new data is created with an initial count of one. The `snk` variable below is a reference to a `SurveyNamesKey` instance; `snl` is a `SurveyNamesLocal` reference:

```
...
try
{
    snl = snlHome.findByPrimaryKey( snk );
    bWasSuccessful = true;

    snl.incrementTotal();
    System.out.println("snl find successful." );
}
catch( ObjectNotFoundException onfe )
{ // create if not found
    System.out.println("trying create." );
    try
    {
        snl = snlHome.create( snk );
    }
}
...

```

If the find or create was successful, the transaction is committed; if not, the transaction is rolled back:

```
if( bWasSuccessful )
{
    bWasSuccessful = doCommit();
}
else { doRollback(); }

return bWasSuccessful;
```

From the Web component's view, nothing has changed. As we'll see next, deployment descriptor entries map `RockSurvey2Bean` to the `java:comp/env/ejb/RockSurveyBean` lookup. Even the same remote home and component interfaces are used, without change, again being mapped using deployment descriptor entries. Therefore, the JSP pages, constants classes, and the `RockSurveyRemote` home and component classes from [Example: The Rock Survey, take 1](#) are reused in take 2.

From the end user perspective, the only difference is a pause while the data is persisted after the user clicks Done.

## Initial deployment settings for the Rock Survey, take 2

It's time to create the initial descriptor information for Survey2App. While we are accumulating quite a few files, remember that the JSP pages, the constants classes, and the `RockSurveyRemote` home and component classes are copies from [Example: The Rock Survey, take 1](#); `RockSurvey2Bean` is an expansion of `RockSurveyBean`. In this portion, we will create the EAR, add the `SurveyNamesBean` entity bean, add the revised `RockSurvey2Bean` session bean, and add the Web component's WAR file.

1. Compile Java source files in the Survey2 directory, or copy the .class files from the prod folder.
2. Start J2EE and `deploytool`.
3. Ensure that the Cloudscape database is started.
4. Create new application:
  - From the menu, select File, New, Application.
  - Browse to the Survey2 folder.
  - Key Survey2App.ear in for the file name.
  - Click New Application.
  - Application Display name is Survey2App by default; click OK.
5. Create new Enterprise JavaBean component -- SurveyNamesBean:
  - Ensure that Application Survey2App is selected.
  - From the menu, select File, New, Enterprise Bean -- Intro display appears; click Next.
  - Select Create New JAR File In Application.
  - Ensure that Survey2App is selected in the drop-down menu.
  - In JAR Display Name, key Survey2JAR.
  - Click Edit.
  - Select the .class files that comprise the bean. These are:
    - SurveyNamesBean.class
    - SurveyNamesKey.class

- SurveyNamesLocal.class
  - SurveyNamesLocalHome.class
  - Click Add; click OK; click Next.
  - Under Bean Type, click Entity.
  - In the Enterprise Bean Class combo box, select the bean implementation: SurveyNamesBean.
  - In Enterprise Bean Name, accept SurveyNamesBean.
  - Select the corresponding interfaces in the combo boxes:
    - For Local Home Interface, select: SurveyNamesLocalHome.
    - For Local Interface, select: SurveyNamesLocal.
  - Click Next.
  - Ensure that Bean-Managed Persistence is selected under Persistence Management.
  - Key SurveyNamesKey in the Primary Key Class textfield. Click Finish.
  - Survey2JAR and SurveyNamesBean now appear under Survey2App.
  - Click the Resource Refs tab; click Add.
  - Under Coded Name, enter "jdbc/gsejbDB". Leave Type as "javax.sql.DataSource", Authentication as "Container", and Sharable checked.
  - Select Survey2JAR, then click the JNDI Names tab.
  - Under EJBs, key ejb/SurveyNamesBean next to SurveyNamesBean.
  - Under References, enter "jdbc/gsejbDB".
6. Create new Enterprise JavaBean component -- RockSurvey2Bean:
- Ensure that Application Survey2App is selected.
  - From the menu, select File, New, Enterprise Bean -- Intro display appears; click Next.
  - Select Add to Existing JAR File.
  - Select Survey2JAR in the drop-down menu.
  - Click Edit.
  - Select the .class files that comprise the bean. These are:

- RockSurvey2Bean.class
  - RockSurveyConstants.class
  - RockSurveyData.class
  - RockSurveyRemote.class
  - RockSurveyRemoteHome.class
  - SurveyConstants.class
- Click Add; click OK; click Next.
  - Under Bean Type, click Session and Stateful.
  - In the Enterprise Bean Class combo box, select the bean implementation: RockSurvey2Bean.
  - In Enterprise Bean Name, accept RockSurvey2Bean.
  - Select the corresponding interfaces in the combo boxes:
    - For Remote Home Interface, select: RockSurveyRemoteHome.
    - For Remote Interface, select: RockSurveyRemote.
  - Click Next; click Finish.
  - RockSurvey2Bean now appears under Survey2JAR. Select the EJB Refs tab;, click Add.
  - When it invokes the lookup method, the bean refers to the home of an enterprise bean like this:

```
Object objRef =  
ic.lookup( "java:comp/env/ejb/SurveyNamesBean" );
```

So, in the Coded Name column, enter `ejb/SurveyNamesBean`.
  - Select Entity for Type and Local for Interfaces.
  - For Home Interface, enter `SurveyNamesLocalHome`.
  - In the Local/Remote Interface column, enter `SurveyNamesLocal`.
  - At the bottom, select "ejb-jar-ic.jar#SurveyNamesBean" from the Enterprise Bean Name drop-down menu.
  - Select Survey2JAR, then click the JNDI Names tab.
  - Under EJBs, key `ejb/RockSurvey2Bean` next to RockSurvey2Bean.
7. Create new Web component:
- Ensure that Application Survey2App is selected.

- From the menu, select File, New, Web Component -- Intro display appears; click Next.
- Select Create New WAR File In Application.
- Ensure that Survey2App is selected in the drop-down menu.
- In WAR Display Name, key Survey2WAR.
- Click Edit.
- Select the files needed for the Web component. These are:
  - The images directory
  - index.jsp
  - RockSurvey.jsp
  - RockSurveyExit.jsp
  - SurveyWelcome.jsp
- Click Add; click OK; click Next.
- Select JSP for the type of Web component, then click Next.
- In the JSP filename combo box, select index.jsp.
- In Web Component Name, allow the default -- index.
- Click Next. There are no Initialization Parameters. Click Next.
- Under Component Aliases, click Add, enter "/index" (without quotes).
- Click Next; click Finish.
- Survey2WAR now appears under Survey2App.
- Select Survey2WAR, then select the EJB Refs tab and click Add.
- When it invokes the lookup method, the Web client refers to the home of an enterprise bean like this:

```
Object objRef =  
ic.lookup( "java:comp/env/ejb/RockSurveyBean" );
```

So, in the Coded Name column, enter `ejb/RockSurveyBean`.
- Accept Session for Type and Remote for Interface.
- For Home Interface, enter `RockSurveyRemoteHome`.
- In the Local/Remote Interface column, enter `RockSurveyRemote`.
- At the bottom, select "ejb-jar-ic.jar#RockSurvey2Bean" from the Enterprise Bean Name drop-down menu. Then click the JNDI Name

button and select `ejb/RockSurvey2Bean` from the drop-down menu.

- Now select `Survey2App` and click on the JNDI Names tab.
- Verify that all JNDI Name columns have the proper entries.

The code we have so far only handles writing data to the `SurveyNames` table. The bean to handle the survey data table will be added after the CMP discussion, so we will wait until [Final Deployment of the Rock Survey example, take 2](#) for the actual, final deployment of `Survey2App`.

## Entity beans with container-managed persistence (CMP)

Put another way, the primary difference between BMP and CMP mentioned in [Entity beans with bean-managed persistence \(BMP\)](#) is that with CMP, the container handles the datastore access, insert, update, and deletion routines. Another big difference in version 2.0 of the EJB specification is that an entity bean class that uses CMP must be declared `abstract`.

Let's talk about that for a moment. Abstract classes are normally used when there is a partial implementation and the class wants to mandate the other methods used without specifying their actual operation. In this case, the *specification* mandates the methods, namely getters and setters. CMP entity beans do not declare instance variables to represent persistent data. Instead, the container-managed data element names are declared in the deployment descriptor. In the bean class, `public abstract` `getXXX` and `setXXX` methods are declared for the data elements; the container subclasses the bean class and generates the actual implementation. The familiar JavaBeans convention is used: if the data element is named `value`, the getter/setter pair would be `type getValue()` and `void setValue( type aValue )`. So, CMP not only generates the datastore routines, but also manages the individual data elements. This means that the developer can access data only through the getters and setters. The specification calls the data elements *virtual fields*. The deployment descriptor declaration of the virtual fields is called an *abstract persistence schema*, and the entries look like this:

```
...
<cmp-version>2.x</cmp-version>
<abstract-schema-name>ValueBean</abstract-schema-name>
<cmp-field>
<field-name>value</field-name>
</cmp-field>
<cmp-field>
<field-name>cost</field-name>
</cmp-field>
...
```

Before moving on to CMP method handling, a couple more differences between BMP and CMP should be mentioned. One is that the described data elements must

be mapped to actual database columns (or whatever the data elements are in the datastore used). This mapping description goes into a vendor-specific container descriptor, usually generated by vendor tools. In the words of the EJB 2.0 specification, "The EJB deployment descriptor ... does not provide a mechanism for specifying how the abstract persistence schema of an entity bean ... is to be mapped to an underlying database. This is the responsibility of the Deployer."

There's another important difference between BMP and CMP. The EJB 2.0 specification introduced the EJB Query Language (QL), a portable subset of SQL92. QL syntax is used to define queries for finder and select methods under CMP. The queries are defined in the deployment descriptor. The QL syntax is normally translated into the access language of the underlying datastore. In version 2.0 CMP, QL is required for all finders except `ejbFindByPrimaryKey()`. For more information, see [Resources](#).

## Handling life cycle methods under CMP

As with BMP, CMP entity beans must respond appropriately in life cycle methods; it's just that the timing and response is different, due to the container's management of persistence. We saw in [Entity beans with container-managed persistence \(CMP\)](#) that a CMP bean class must be declared `abstract`. As with BMP, the class must provide a public, no-arg constructor. The sequence of events (see [The entity bean life cycle](#)) is also the same.

Let's take a look at how the methods (first described in [Entity beans with bean-managed persistence \(BMP\)](#)) work when using CMP:

- **`public void setEntityContext(EntityContext ec)`**  
Same as BMP.
- **`public void unsetEntityContext()`**  
Same as BMP.
- **`public PrimaryKeyClass ejbCreate(...)`**  
The method is now called *before* the actual creation mechanism executes. The code should only validate input arguments as necessary, and initialize the virtual fields with the input arguments through the setter methods. The return *type* is still the primary key type, but the return *value* should be `null`.  
It is possible to write a bean without an `ejbCreate()` method when the class deals only with existing data. In this case, there must be no `create()` methods exposed on the home interface.
- **`public void ejbPostCreate(...)`**  
Same as BMP.

- **public <primary key type or collection> ejbFind<METHOD>(…)**  
No finder methods are coded by the developer; all finder methods declared in the home interface are implemented by the container. All finder methods, except `ejbFindByPrimaryKey()`, are specified in the query deployment descriptor; the container uses the EJB Query Language string defined by the developer as the basis for the finder methods.
- **public void ejbLoad()**  
The container invokes this method immediately *after* reading from the datastore and setting the virtual fields. The only tasks the developer needs to do here would involve some sort of data transformation from the raw data kept in the datastore, or to recalculate/reset instance variables that depend on the persistent data state.
- **public void ejbStore()**  
The container invokes this method *before* writing to the datastore. As with `ejbLoad()`, the only developer tasks to be performed here would have to do with data translation or preparation prior to persistence -- for example, compressing text and invoking the appropriate setter method.
- **public void ejbRemove()**  
The container invokes this method *before* removing an entity from the datastore. The method can be used to perform any operations required before the data is deleted.
- **public void ejbActivate()**  
Same as BMP.
- **public void ejbPassivate()**  
Same as BMP.
- **public abstract type ejbSelect<METHOD>(…)**  
When using CMP entity beans, the developer can define internal query methods called *select* methods. The actual implementation is done at deployment time using vendor-provided tools and EJB Query Language syntax.

## Example: The Rock Survey, take 2 (continued)

The "SurveyBeanTable", created in [The Rock Survey database](#), contains exactly one row consisting of counter and total columns for the survey data. The last step for take 2 of the Rock Survey application will add data management for the table. As you might expect from the way we went about things in [Example: The Rock Survey, take 2](#) and the preceding discussion, we will accomplish the task by adding another entity bean, this time using CMP, and expanding the `persist()` method in

RockSurvey2Bean.

You may ask yourself why any normal human would name a table "SurveyBeanTable" (unexpectedly, the double quotes are part of the name). The answer is that *deploytool* generated and named it -- see the discussion about generating the default SQL in [Final Deployment of the Rock Survey example, take 2](#). The developer has no real control over the datastore creation when using CMP; table naming and Object/Relational mapping is not addressed by the specification, meaning that it is vendor proprietary. This state of affairs can become a major issue when dealing with existing databases and for data portability among vendor implementations. And, yes, I cheated in [The Rock Survey database](#) by using the container-generated SQL to ensure that the data was set up to match the container's expectations.

The `SurveyBean` class manages "SurveyBeanTable" data, and, other than providing getters/setter for 19 column variables, is surprisingly small. That's because all of the persistence code (read JDBC/SQL here) is generated by the container. As we will see shortly, the reduction in code comes at the price of more complicated and complex deployment descriptors, along with some vendor-dependent entries.

The local home interface should look familiar:

```
// the local home interface for SurveyBean
import javax.ejb.*;

public interface SurveyLocalHome extends EJBLocalHome
{
    public SurveyLocal create( String Type )
        throws CreateException;

    public SurveyLocal findByPrimaryKey (String Type)
        throws FinderException;
} // end interface SurveyLocalHome
```

The local component interface is completely composed of getter and setters. These are all declared abstract in the `SurveyBean` code.

```
// the local component interface for SurveyBean
import javax.ejb.*;

public interface SurveyLocal extends EJBLocalObject
{
    public String getType();
    public void setType( String TypeIn );

    public int getSF();
    public void setSF( int SFIn );
    ...
    public double getPounds();
    public void setPounds( double PoundsIn );
}
```

```
} // end interface SurveyLocal
```

In this case, `RockSurvey2Bean`, the managing session bean, does all of the editing and validation. At times, you may prefer to have these tasks centralized, probably in the entity bean itself. So how do you do that when the setter is `abstract`? The answer is to have another method to perform any tasks for a data element, and, if the element passes editing, have that method call the setter. A naming convention for the editing methods becomes very helpful for maintainable code in that situation. We do have one edit in the bean: in the `ejbCreate()` method, which in CMP is invoked *before* the actual insertion, the primary key `String` argument is checked to be sure that it is "ROCK". Again, there will only be one row in this table and the key will always be "ROCK". If the test fails, a `CreateException` is thrown.

I should mention that it is more typical to send the key and also arguments for all of the data to be inserted in the row to the `ejbCreate()` method. Here, because inserts should happen very infrequently, probably once in this table's lifetime, and because 19 arguments could get unwieldy to handle correctly, we create the row with just the key, then call the setters using a `RockSurvey2Bean` method, `setupSurveyData()`, already written to handle data updates. The coding time and maintenance efficiencies gained by using this method -- in this case -- more than offset any performance losses.

In `RockSurvey2Bean`, instance variables for the `SurveyBean` local interfaces have been added, and the `ejbPassivate()` method now sets the interface variables for both entity beans and the `UserTransaction` to null. The `persist()` method tests for null values and resets the variables if appropriate.

The new code related to `SurveyBean` should look familiar. The code first obtains the local home:

```
if( slHome == null )
{
    sMsg = "SurveyBean";
    ic = new InitialContext();
    slHome =
        (SurveyLocalHome)ic.lookup(
            "java:comp/env/ejb/SurveyBean" );
}
```

Since there will always be only one row in the "SurveyBeanTable" table, the "try the finder first, perform create upon failure" technique is even more appropriate than before. Here's the code:

```
try
{
    sl = slHome.findByPrimaryKey( "ROCK" );
    bWasSuccessful = true;
}
```

```
        setupSurveyData();
        System.out.println("sl find successful." );
    }
    catch( ObjectNotFoundException onfe )
    { // create if not found
        System.out.println("trying create." );
        try
        {
            sl = slHome.create( "ROCK" );
            setupSurveyData();
            ...
        }
        ...
    }
}
```

This code is performed in-line with the `SurveyNamesBean` management code and within the same transaction.

The other addition to `RockSurvey2Bean` is the `setupSurveyData()` method, which breaks down the survey input data to the appropriate categories for updating the datastore. Notice that `setupSurveyData()` is invoked for both creates and updates.

Again, externally, to both the Web component and the end user, there is no difference between take 1 and take 2 other than whatever time the datastore updates take.

## Final Deployment of the Rock Survey example, take 2

Now that both entity beans are coded and the `RockSurvey2Bean`'s persistence method is complete, it's time to add the final information to the deployment descriptor and actually deploy take 2 of the Rock Survey:

1. If you haven't already, compile Java source files in the `Survey2` directory, or copy the `.class` files from the `prod` folder.
2. Start J2EE and `deploytool`.
3. Ensure that the Cloudscape database is started.
4. Create new Enterprise JavaBean component -- `SurveyBean`:
  - Ensure that Application `Survey2App` is selected.
  - From the menu, select `File, New, Enterprise Bean -- Intro display` appears; click `Next`.
  - Select `Add to Existing JAR File`.
  - Select `Survey2JAR` in the drop-down menu.

- Click Edit.
- Select the .class files that comprise the bean. These are:
  - SurveyBean.class
  - SurveyLocal.class
  - SurveyLocalHome.class
- Click Add; click OK; click Next.
- Under Bean Type, click Entity.
- In the Enterprise Bean Class combo box, select the bean implementation: SurveyBean.
- In Enterprise Bean Name, accept SurveyBean.
- Select the corresponding interfaces in the combo boxes:
  - For Local Home Interface, select: SurveyLocalHome.
  - For Local Interface, select: SurveyLocal.
- Click Next.
- Ensure that Container Managed Persistence 2.0 is selected under Persistence Management.
- In Abstract Scheme Name, key Survey.
- Select *all* of the check boxes in the "Fields to be persisted" scrolled list.
- Key java.lang.String in the Primary Key Class textfield.
- Select "type" in the Primary Key Field Name drop-down menu. Click Next; click Finish.
- SurveyBean now appears under Survey2JAR under Survey2App.
- Click the Entity tab, then click Deployment Settings.
- Unselect the check boxes under Database Table.
- Click Database Settings.
- Under Database JNDI Name, enter "jdbc/gsejbDB".
- Click OK.
- Click Generate Default SQL.
  - If you want to see the SQL, click Container Methods, then click

on each method. *Do not* change anything in the SQL Query box. `deploytool` does not track changes properly, and you'll get to do everything over again. You can also see the table name and columns generated by the container. Note that there is no vendor-independent means of describing the name.

- Click OK.
  - Click the Security tab.
  - Under Method Permissions, click Local Home. Change Availability for the `remove()` method to No Users. Click Local. Change Availability for the `remove()` method to No Users. No deletes allowed.
  - Select Survey2JAR, then click the JNDI Names tab.
  - Under EJBs, key `ejb/SurveyBean` next to SurveyBean.
5. Modify Enterprise JavaBean component -- RockSurvey2Bean:
- Select RockSurvey2Bean.
  - Select the EJB Refs tab; click Add.
  - When it invokes the lookup method, the bean refers to the home of an enterprise bean like this:  

```
Object objRef =  
ic.lookup("java:comp/env/ejb/SurveyBean");
```

So, in the Coded Name column, enter `ejb/SurveyBean`.
  - Select Entity for Type and Local for Interfaces.
  - For Home Interface, enter `SurveyLocalHome`.
  - In the Local/Remote Interface column, enter `SurveyLocal`.
  - At the bottom, select "ejb-jar-ic.jar#SurveyBean" from the Enterprise Bean Name drop-down menu.
6. Deploy the Survey2App:
- Select Survey2App.
  - Select Tools, Deploy from the menu.
  - Under Object To Deploy, ensure that Survey2App is selected.
  - Under Target Server, we are using localhost. Be sure that is selected.
  - Select the Return Client JAR check box.
  - Select Save object before deploying.

- Click Next.
- Verify that the JNDI names are correct, then click Next.
- Key `"/Survey2App"` in the context root.
- Click Next, then Finish.

The Deployment Progress dialog will display. Wait until the progress bars are complete and the Cancel button changes to OK, then click OK. The Survey2App application is deployed.

If you look in the Survey2 folder, you will see that it now contains Survey2App.ear and Survey2AppClient.jar. To run the Survey2App, start both J2EE and Cloudscape, then start up your browser and enter `http://localhost:8000/Alice` as the target URL. When the Alice's World home page appears, click on the "Alice's Surveys - Take 2" link.

---

## Section 5. Message-driven beans

### Message-driven beans overview

Message-driven processing is a very effective, and often underused, technique for programs that do not need to operate directly in response to user requests (that is, for *batch* rather than interactive type processing). For example, how can a program know that it should shut down gracefully, or automatically react to some other external event? Messaging is a good solution in these cases. It is also useful for processing that is not time critical, including some interactive applications; why should a user wait for a process to complete when it doesn't matter if the task is done exactly at that moment? Many data entry tasks in which a set of data (debits/credits, invoices, or receipts, for instance) is entered, then reviewed, and finally posted, are appropriate examples. Messaging can also be useful for interprocess communication and systems integration using text or XML data, even when programs are written in different languages. From the messaging client's perspective, it just sends a message and immediately goes on to other tasks, relying on the message's receiver to perform operations against the message data.

Message-driven beans (MDBs) are a new type of bean introduced with the EJB 2.0 specification. MDBs are stateless in terms of client-conversational state. They are used for *asynchronous processing* of Java Message Service (JMS) messages, and go into action upon the receipt of a client message, although they are not linked to

any specific client. In fact, since the messages come from a *queue* or a *topic*, a single bean may act on behalf of many clients. In contrast to the other bean types, MDBs have no home or component interfaces, and are invoked by the container in response to the arrival of JMS messages. From the client view, an MDB is a message consumer that performs one or more tasks on the server, based on its interpretation of the message.

Notice that MDBs are all about asynchronicity; session and entity beans could, and still can, process synchronous messages. Prior to the advent of message-driven beans, however, there was no real way that EJB components could process asynchronous messages; generally, an external application was necessary.

**Figure 5. The `javax.ejb.MessageDrivenBean` interface**



All message-driven beans must implement the `javax.ejb.MessageDrivenBean` interface, as shown in Figure 5. An MDB's life cycle is very similar to that of a stateless session bean (see [Stateless session beans](#)) in that it only has two states: Does Not Exist and Ready. The container is responsible for creating instances of MDBs before message delivery begins. To place an MDB in the Ready state, the container:

- Instantiates a bean using the `Class.newInstance()` method
- Invokes the bean's `setMessageDrivenContext()` method
- Invokes the bean's `ejbCreate()` method

At this point, the bean enters the method-ready pool and can be called upon to process messages. When the container no longer requires the bean, it calls the bean's `ejbRemove()` method. After this action, the bean is effectively destroyed and back to the Does Not Exist state.

**Figure 6. The `javax.jms.MessageListener` interface**



Message-driven beans must also implement the `javax.jms.MessageListener` interface, as shown in Figure 6. The interface consists of just one callback method, `onMessage()`, which has a single `javax.jms.Message` input argument. The callback feature means that the bean doesn't have to spend CPU cycles and time polling for messages, leading to more scalable and potentially higher performance applications. If you wondered why other bean types couldn't handle asynchronous messages, the reason is that the EJB 2.0 specification states that they are not "permitted" to implement `MessageListener`. This ban is not simply a specification mandate; several technical reasons exist for the restriction.

`onMessage()` is really where all of the action occurs, but before discussing that method and message-driven beans further, an understanding of JMS concepts, lingo, and basic operations is required. The next few sections briefly cover the necessary JMS background.

## The Java Message Service (JMS)

Enterprise messaging systems (EMS) comprise a product category often used in larger systems to provide assured delivery of messages between participating applications. These products are also known as *message oriented middleware*, or MOM. As with many software products, the API for virtually every EMS/MOM is vendor proprietary. The JMS specification was written to unify these APIs for the Java developer, and includes both an API and a Service Provider Interface (SPI). Any vendor can use the SPI to write what amounts to a pluggable driver to implement the JMS specification. To the developer, this aspect of JMS means product independence and transparency, and only one standard API to learn.

JMS supports two messaging methodologies (or *domains*): *point-to-point* (PTP) and *publish and subscribe* (pub/sub). PTP deals with *queues*, *senders*, and *receivers*. There may be many message senders to a queue, but, normally, only one type of receiver (since MDBs can be pooled, there may be several of a given type consuming messages from the queue). Once the message has been received, it is removed from the queue. Pub/sub deals with *topics*, *publishers*, and *subscribers*. There may be any number of publishers for a topic, and any number of subscribers to the topic (similar in concept to Java listeners). Subscribers are notified of a newly

published message for a topic, and every subscriber gets a copy. JMS uses the notion of message *producers* for clients that generate and send or publish messages, and message *consumers* for clients that receive messages via queues or topics. Queues and topics are known as *destinations*.

The basic steps in JMS programming are:

- **Obtain a JMS ConnectionFactory.** Use a JNDI lookup to obtain a `QueueConnectionFactory` (PTP) or a `TopicConnectionFactory` (pub/sub).
- **Obtain a JMS destination.** Use a JNDI lookup to obtain a `Queue` (PTP) or a `Topic` (pub/sub).
- **Create a JMS Connection.** This will either be a `QueueConnection` (PTP) or a `TopicConnection` (pub/sub). Use the specific `ConnectionFactory.createXXXConnection()` method.
- **Create a JMS session.** This will either be a `QueueSession` (PTP) or a `TopicSession` (pub/sub). Use the specific `Connection.createXXXSession()` method.
- **Create a producer or consumer.** This can be a `QueueSender` or a `QueueReceiver` created using the `QueueSession` (PTP) or a `TopicPublisher` or a `TopicSubscriber` created using the `TopicSession` (pub/sub).
- **Send and/or receive a Message.** Use the producer or consumer in conjunction with the destination. For consumers, call `Connection.start()` to initiate the flow of incoming messages.

All of the classes referenced above are in the `javax.jms` package. Although we will only use PTP in the example, these steps apply to programs for both message domains. For more information about JMS and JMS programming, see [Resources](#).

## JMS destinations and messages

Both JMS `Queue`s and `Topic`s are referred to generally as *destinations*. From a J2EE perspective, they also fall under a group, along with connection factories, known as *administered objects*. This means that they must be created and maintained by an administrator, external to a bean or application. It also means using a vendor-proprietary method to perform these tasks. The J2EE RI provides an administration tool, `j2eeadmin`, which you can use to create queues and topics.

We've seen `j2eeadmin` before, when adding a `DataSource` entry (see [The Rock Survey database](#) ). For destinations, the

`-addJmsDestination`

input argument is sent to `j2eeadmin`. The deployment panel for the

message-driven bean example program, [Deploying the Rock Survey example, take 3](#), shows how to use `j2eeadmin` to create a `Queue`.

Ultimately, messages are the reason for using JMS. The API defines a `Message` interface, along with subinterfaces for specific message types. A `Message` is composed of three parts:

- **Header:** Header values contain information useful for identifying and routing messages.
- **Properties:** Properties are a built-in mechanism for adding application-specific header information. Programs can use this information for message selection and filtering.
- **Body:** JMS defines five body types (and corresponding subinterfaces) for a `Message`:
  - `Bytes`
  - `Map`
  - `Object`
  - `Stream`
  - `Text`

You can read about the details of the types and subinterfaces in the J2EE API documentation. The important thing to consider here is message *data* portability; clearly a `Message` containing a `Java Object` will not be understood by a C language, COBOL, RPG, or other language program. While writing portable code often means doing more work, Java developers, more than any others, should value portability in data as well as code, right? Therefore, you should give careful consideration to message types and their implications as you design applications.

## Sending and receiving JMS messages

The following code snippet illustrates how to set up the JMS framework for sending and receiving messages. Notice how the code follows the steps outlined in [The Java Message Service \(JMS\)](#). Most applications, naturally, will only create a `QueueReceiver` or a `QueueSender` (or `TopicPublisher` or a `TopicSubscriber` for pub/sub). JMS has a transaction framework of its own, and you can specify transaction handling when creating a `Session`. Here, and in the example application for this section, the code specifies `false`, meaning no transactions are used. The `Session.AUTO_ACKNOWLEDGE` argument specifies automatic acknowledgement of the message when the `onMessage()` method completes. If the message is not acknowledged, it will be resent. You should be

aware that the transactional mode can have a major impact on quality of service (QoS) and guaranteed delivery of messages. For more information about JMS transactions, see [Resources](#).

If a `QueueConnection` calls `start()` to begin receiving incoming messages, `QueueConnection.stop()` should be called before exiting the application. `QueueConnection.close()` should *always* be called prior to exit to conserve resources. Pub/sub programs look very similar to the following code, the primary difference being that `Topic` objects are used rather than `Queue` objects.

```
import javax.jms.*;
import javax.naming.*;

public class XXX implements MessageListener
{
    InitialContext          ic;
    Queue                   q;
    QueueConnection         qc;
    QueueConnectionFactory qcf;
    QueueReceiver           qReceiver;
    QueueSender             qSender;
    QueueSession            qSession;
    TextMessage             tm;

    public void setupJMS( String sQueueConnectionFactoryName,
                        String sQueueName )
    {
        try
        {
            // lookup the JMS driver and queue
            ic = new InitialContext();
            qcf = (QueueConnectionFactory)initContext.lookup(
                "java:comp/env/jms/" +
                sQueueConnectionFactoryName );
            q = (Queue)initContext.lookup(
                "java:comp/env/jms/" + sQueueName );

            // create the necessary JMS objects
            qc = qcf.createQueueConnection();
            qSession = qc.createQueueSession(
                false, Session.AUTO_ACKNOWLEDGE );
            // create the sender
            qSender = session.createSender( q );

            // create the receiver
            qReceiver = qSession.createReceiver( q );
            qReceiver.setMessageListener( this );
            qc.start();
        }
        ...
    } // end setupJMS

    public void sendMsg( String sMsgText )
    {
        try
        {
            tm = qSession.createTextMessage( sMsgText );
            qSender.send( tm );
        }
        ...
    } // end sendMsg
}
```

```
public void onMessage( Message msg )
{
    try
    {
        String sMessage = ((TextMessage) msg).getText();
        ...
    }
    ...
} // end onMessage

...

// if the QueueConnection issued start() for incoming
// messages, invoke stop() prior to exit.

// close the QueueConnection before exiting!!! This
// operation will also close the Session and Sender.

} // end class XXX
```

## Using message-driven beans

After going through the JMS information, you'll probably find it surprisingly simple to code message-driven beans for receiving messages. That's because the container handles all of the steps mentioned in [The Java Message Service \(JMS\)](#) except actually receiving the messages. In general, the same MDB can be used to receive PTP or pub/sub messages by setting the appropriate `ConnectionFactory`, destination, and destination type on deployment. As is often the case with EJB technology, some of this coding ease translates into more complexity in the deployment descriptor.

An MDB must have a public, no-arg constructor, and is not allowed to throw application exceptions. In addition, an MDB must implement the following methods (for the life cycle sequence, see [Message-driven beans overview](#)):

- **public void setMessageDrivenContext(MessageDrivenContext mdc)**  
The container normally calls this method exactly once, after instantiation, to pass in the associated `MessageDrivenContext`.
- **public void ejbCreate()**  
Called after the `setMessageDrivenContext()` method. This is a good time to access or obtain any resources that will be used for the life of the bean.
- **public void onMessage(Message msg)**  
Called by the container when a message has arrived on the bean's associated `Queue` or `Topic`. Your code should check for the expected message types, using the `instanceof` operator: there's nothing to stop a client from sending *any* of the available message types. Other than those that deal with the `Message` object, no JMS methods need be

invoked by the bean; it just cracks or parses the received message and performs the relevant operations.

- `public void ejbRemove()`  
Called when the container intends to terminate the bean. All resources should be released at this time.

Once the message has been received, the MDB can perform all the operations itself, or act as a manager, sending the information to and controlling other beans. Queue architecture is fairly limited, because the specification envisions a queue for each bean type. It also warns against deploying multiple beans against the same queue, primarily citing message order concerns; however, there is no guaranteed order for message receipt in any event. The deployment descriptor does allow for JMS message selectors to direct specific messages to a specific bean type.

## Example: The Rock Survey, take 3

After we completed take 2 of the Rock Survey (see [Example: The Rock Survey, take 2](#) and [Example: The Rock Survey, take 2 \(continued\)](#)), the application really does everything necessary to perform its task. But, for tutorial purposes, we're going to add a messaging layer. This can be a valid design under heavy loads and various other situations and design requirements. In addition, the example may give you some ideas about different approaches to processing in terms of timeliness, the client's view of the speed of an application, and loosely coupled operations.

Take 3 adds a message-driven bean ( `RockSurveyMDBean` ) and a modification of the existing session bean ( `RockSurvey2Bean` ). In the process, the application flow changes from a direct update of the datastore by the client application. Now the session bean sends a message, and the message-driven bean asynchronously manages the database update. Again, from the Web component's and end user's perspective there is no change except that the Done button operation returns more quickly.

All of the actual persistence code in the `persist()` method of `RockSurvey2Bean`, which in this version becomes `RockSurvey3Bean`, is removed and placed in `RockSurveyMDBean`. `RockSurvey3Bean.persist()` now gathers the data and sends a message containing an initialized `RockSurveyData` object (see [Example: The Rock Survey, take 1](#) ). Most of the relevant code is very similar to the code snippet in [Sending and receiving JMS messages](#).

First, we declare the JMS-related instance variables:

```
// JMS variables for PTP
ObjectMessage      om;
Queue              q;
QueueConnection    qc;
```

```

QueueConnectionFactory  qcf;
QueueReceiver           qReceiver;
QueueSender             qSender;
QueueSession           qSession;
TextMessage             tm;

```

In `persist()`, if the `QueueConnection` hasn't yet been created, the necessary messaging objects are obtained:

```

if( qc == null)
{
    // lookup the administered objects
    ic = new InitialContext();
    qcf = (QueueConnectionFactory)ic.lookup(
        "java:comp/env/jms/QueueConnectionFactory" );
    q = (Queue)ic.lookup(
        "java:comp/env/jms/RockSurveyQueue" );

    // create the necessary JMS objects
    qc = qcf.createQueueConnection();
    qSession = qc.createQueueSession(
        false, Session.AUTO_ACKNOWLEDGE );
    // create the sender
    qSender = qSession.createSender( q );
} // end if qc is null

```

Notice that the `QueueSession` is created as nontransacted with automatic acknowledgement.

At this point, the program is ready to create message objects and send messages. Beginning and ending `TextMessages` are sent, with the send of the `ObjectMessage`, which contains the survey data, sandwiched in the middle. The `ObjectMessage` is loaded with `rsd`, a `RockSurveyData` object.

```

...
tm = qSession.createTextMessage(
    "sending RockSurveyData to RockSurveyQueue." );
qSender.send( tm );
...
om = qSession.createObjectMessage( rsd );
qSender.send( om );
...
tm.setText( "RockSurveyData sent." );
qSender.send( tm );
...

```

That's it for the message sending portion. To conserve resources, in `ejbPassivate()`, the `QueueConnection` is closed (this also closes the other related JMS objects) and the relevant instance variables are set to `null`. On termination, `ejbRemove()` invokes `ejbPassivate()` for the same purpose.

```

public void ejbPassivate()
{
    if( qc != null)

```

```

    {
        try { qc.close(); }
        catch (JMSEException e) { /* can't do anything */ }
    }
    om = null;
    q = null;
    qc = null;
    qcf = null;
    qReceiver = null;
    qSender = null;
    qSession = null;
    tm = null;
} // end ejbPassivate

```

## Example, take 3: The message-driven bean

In the message-driven bean ( `RockSurveyMDBean` ), the persistence-related instance variables and the `persist()`, `doCommit()`, and `doRollback()` methods are transferred wholesale from `RockSurvey2Bean` (see [Example: The Rock Survey, take 2](#) and [Example: The Rock Survey, take 2 \(continued\)](#) ). Since MDBs don't have a `SessionContext`, the `MessageDrivenContext` is used instead to get the `UserTransaction` object:

```
ut = mdc.getUserTransaction();
```

And now, what you've all been waiting for: the MDB's `onMessage()` method:

```

public void onMessage( Message msg )
{
    try
    {
        if( msg instanceof TextMessage )
        {
            TextMessage tm = (TextMessage)( msg );
            System.out.println( "Message received: " +
                               tm.getText() );
            return;
        }

        if( msg instanceof ObjectMessage )
        {
            System.out.println( "RockSurveyData " +
                               "ObjectMessage received. " );
            ObjectMessage om = (ObjectMessage)( msg );
            rsd = (RockSurveyData)om.getObject();

            if( persist() )
            {
                System.out.println( "Table data persisted, " );
            }
            else
            {
                System.out.println( "Problem persisting data." );
            }
            return;
        }
    }
    else

```

```
        {
            System.out.println( "Unknown Message type: " +
                               msg.getClass().getName() );
        }
    }
    catch( JMSEException jmse )
    {
        System.out.println( "JMSEException: " +
                            jmse.getMessage() );
        jmse.printStackTrace();
    }
} // end onMessage
```

onMessage is prepared to handle TextMessage and ObjectMessage type messages. If it receives a TextMessage, the text is extracted and printed. For ObjectMessages, the contained object is retrieved into a RockSurveyData. Then the persist() method is invoked to update the database and the result is printed. Any other message types are noted and then ignored.

## Deploying the Rock Survey example, take 3

This time, instead of creating everything from scratch, we'll use a copy of the EAR from the take 2 version and modify it to meet this version's needs. Follow the directions carefully.

1. Compile Java source files in the Survey3 directory, or copy the .class files from the prod folder.
2. Start J2EE and deploytool.
3. Ensure that the Cloudscape database is started.
4. Create the destination queue:
  - From the command line, key and execute: `j2eeadmin -addJmsDestination jms/RockSurveyQueue queue`
5. Create new application:
  - There is a copy of Survey2App.ear from the Survey2 Directory in the Survey3 directory. Rename it to Survey3App.ear.
  - From the menu, select File, Open.
  - Browse to Survey3 folder.
  - Select Survey3App.ear.
  - Click Open Object.

- Application Displays as Survey2App1.
  - Under the General tab, change the Application Display Name to Survey3App.
  - Select Survey2WAR. Under the General tab, change the WAR Display Name to Survey3WAR.
  - Click Edit, then browse to the Survey3 directory.
  - Select the Survey3 files:
    - The images directory
    - index.jsp
    - RockSurvey.jsp
    - RockSurveyExit.jsp
    - SurveyWelcome.jsp
  - Click Add. Select "Yes to All" for the overwrite warning.
  - Click OK.
  - Select Survey2JAR. Under the General tab, change the JAR Display Name to Survey3JAR.
  - Click Edit, browse to the Survey3 directory.
  - Select the *all* of the .class files in Survey3.
  - Click Add. Select "Yes to All" for the overwrite warning.
  - Click OK.
  - Select RockSurvey2Bean. From the Edit menu, select Delete. Answer Yes to "Are You Sure?"
  - Click Edit. Under Contents of Survey3JAR, Select RockSurvey2Bean.class, then click Remove. Answer Yes to "Are You Sure?" Click OK.
  - Select SurveyNamesBean. Under the General tab, change the Enterprise Bean Name to SurveyNames3Bean.
  - Select SurveyBean. Under the General tab, change the Enterprise Bean Name to Survey3Bean.
6. Create new Enterprise JavaBean component -- RockSurveyMDBean:
- Ensure that Application Survey3App is selected.

- From the menu, select File, New, Enterprise Bean -- Intro display appears, click Next.
- Select Add to Existing JAR File.
- Select Survey3JAR in the drop-down menu. The RockSurveyMDBean.class file was added above. Click Next.
- Under Bean Type, click Message-Driven.
- In the Enterprise Bean Class combo box, select the bean implementation: RockSurveyMDBean.
- In Enterprise Bean Name, accept RockSurveyMDBean.
- Click Next, accept Bean Managed Transaction Management.
- Click Next.
- For Destination Type, select Queue.
- For Destination, select jms/RockSurveyQueue.
- For Connection Factory, select jms/QueueConnectionFactory.
- Click Next; click Finish.
- RockSurveyMDBean now appears under Survey3JAR. Select the EJB Refs tab; click Add.
- In the Coded Name column, enter ejb/SurveyNamesBean.
- Select Entity for Type and Local for Interfaces.
- For Home Interface, enter SurveyNamesLocalHome.
- In the Local/Remote Interface column, enter SurveyNamesLocal.
- At the bottom, select "ejb-jar-ic.jar#SurveyNames3Bean" from the Enterprise Bean Name drop-down menu.
- Click Add.
- In the Coded Name column, enter ejb/SurveyBean.
- Select Entity for Type and Local for Interfaces.
- For Home Interface, enter SurveyLocalHome.
- In the Local/Remote Interface column, enter SurveyLocal.
- At the bottom, select "ejb-jar-ic.jar#Survey3Bean" from the Enterprise Bean Name drop-down menu.
- Select Survey3JAR, then click the JNDI Names tab.

- Under EJBs, verify that `jms/RockSurveyQueue` is entered next to `ejb/RockSurveyMDBean`.
7. Create new Enterprise JavaBean component -- `RockSurvey3Bean`:
- Ensure that Application `Survey3App` is selected.
  - From the menu, select File, New, Enterprise Bean -- Intro display appears; click Next.
  - Select Add to Existing JAR File.
  - Select `Survey3JAR` in the drop-down menu. The `.class` file was added above. Click Next.
  - Under Bean Type, click Session and Stateful.
  - In the Enterprise Bean Class combo box, select the bean implementation: `RockSurvey3Bean`.
  - In Enterprise Bean Name, accept `RockSurvey3Bean`.
  - Select the corresponding interfaces in the combo boxes:
    - For Remote Home Interface, select: `RockSurveyRemoteHome`.
    - For Remote Interface, select: `RockSurveyRemote`.
  - Click Next; click Finish.
  - `RockSurvey3Bean` now appears under `Survey3JAR`.
  - Select Resource Refs tab; click Add.
  - In the Coded Name field, enter `jms/QueueConnectionFactory`.
  - For Type, select `javax.jms.QueueConnectionFactory`.
  - Accept Authentication as Container and check Sharable.
  - In JNDI Name textfield, enter `jms/QueueConnectionFactory`.
  - Select Resource Env. Refs tab; click Add.
  - In the Coded Name field, enter `jms/RockSurveyQueue`.
  - For Type, select `javax.jms.Queue`.
  - In JNDI Name textfield, enter `jms/RockSurveyQueue`.
  - Select `Survey3JAR`, then click the JNDI Names tab.
  - Under EJBs, key `ejb/RockSurvey3Bean` next to `RockSurvey3Bean`.
  - Verify all other JNDI Name entries.

8. Modify the Web component:
  - Select Survey3WAR, then select the EJB Refs tab and select the "ejb/RockSurveyBean" row.
  - At the bottom, select "ejb-jar-ic.jar#RockSurvey3Bean" from the Enterprise Bean Name drop-down menu. Then click the JNDI Name button and select ejb/RockSurvey3Bean from the drop-down menu.
  - Now select Survey3App and click on the JNDI Names tab.
  - Verify that all JNDI Name columns have the proper entries.
9. Deploy the Survey3App:
  - Select Survey3App.
  - Select Tools, Deploy from the menu.
  - Under Object To Deploy, ensure that Survey3App is selected.
  - Under Target Server, we are using localhost. Be sure that is selected.
  - Select the Return Client JAR check box.
  - Select Save object before deploying.
  - Click Next.
  - Verify that the JNDI names are correct, then click Next.
  - Key "/Survey3App" in the context root.
  - Click Next, then Finish.

The Deployment Progress dialog will display. Wait until the progress bars are complete and the Cancel button changes to OK, then click OK. The Survey3App application is deployed.

If you look in the Survey3 folder, you will see that it now contains Survey3App.ear and Survey3AppClient.jar. To run the Survey3App, start both J2EE and Cloudscape, then start up your browser and enter `http://localhost:8000/Alice` as the target URL. When the Alice's World home page appears, click on the "Alice's Surveys - Take 3" link.

---

## Section 6. Displaying the Rock Survey example results

## Example: The Survey Results application

The RockSurvey Results application (see [The Survey Results application](#) ) displays the information garnered from the data collected and stored by the Rock Survey application developed during the course of the tutorial. It also reinforces what you've learned about working with beans, and shows how to use JDBC with session beans for read-only access.

RockResultsBean is a stateless session bean that manages data access and information return to the client. Its home and component interfaces look rather innocuous and at first glance don't appear to offer much functionality:

```
// Remote Home interface for RockResultsBean
import javax.ejb.*;
import java.rmi.RemoteException;

public interface RockResultsRemoteHome extends EJBHome
{
    // required
    RockResultsRemote create()
        throws RemoteException,
        CreateException;
} // end RockResultsRemoteHome
```

```
// Remote component interface for RockResultsBean
import java.rmi.RemoteException;
import javax.ejb.*;

public interface RockResultsRemote extends EJBObject
{
    public RockResultsData getData( int maxNames )
        throws RemoteException;
} // end RockResultsRemote
```

However, the bean provides everything the client needs to interpret and display the information from the database tables "SurveyBeanTable" and SurveyNames.

The client code to obtain the bean should be familiar by now:

```
public RockResultsRemote createRRBean()
{
    InitialContext ic;
    Object oRef;
    RockResultsRemote rr = null;
    RockResultsRemoteHome rrHome = null;
    String sMsg = "Couldn't create RockResultsBean.";

    try
    {
        ic = new InitialContext();
```

```

    oRef =
        ic.lookup( "java:comp/env/ejb/RockResultsBean" );
    rrHome =
        (RockResultsRemoteHome)PortableRemoteObject.
            narrow( oRef, RockResultsRemoteHome.class );
    rr = rrHome.create();
}
catch( RemoteException re )
{
    System.out.println( sMsg +
        re.getMessage() );
}
catch( CreateException ce )
{
    System.out.println( sMsg +
        ce.getMessage() );
}
catch( NamingException ne )
{
    System.out.println( "Unable to lookup home: " +
        "RockResultsBean. " + ne.getMessage() );
}

return rr;
} // end createRRBean

```

The component interface only exposes the `getData()` method, which takes an integer argument and returns a `RockResultsData` object. The input argument is used both to size the name arrays and to set the fetch size for the prepared statement that accesses the `SurveyNames` table. `RockResultsData` is a helper class, similar to the `RockSurveyData` class used in the Rock Survey application. Once this object is obtained, no other remote calls to `RockResultsBean` are necessary. The client uses `RockResultsData` like this:

```

// load RockSurvey data from datastore, get a RockResultsData object
rrd = rr.getData( 3 ); // max three last names
...
// load data
asGM = rrd.loadGenderMarital();
asAW = rrd.loadAmountAndWeights();
asMFN = rrd.getLastNames();

asFav = rrd.loadFavorites( aiFav );
adFav = rrd.toPercentArray( aiFav );

asFrom = rrd.loadLocations( aiFrom );
adFrom = rrd.toPercentArray( aiFrom );
...

```

While `RockResultsData` has a complete set of getters, the methods referenced above all return arrays and contain everything the `RockSurvey Results` client needs. The remainder of the JSP page is mostly concerned with displaying the elements of the arrays.

Internally, `RockResultsBean.getData()` obtains a `JDBC DataSource`, then uses a `Connection` and a `PreparedStatement` to access the "SurveyBeanTable" and `SurveyNames` data. Once that is done, it creates and

returns a `RockResultsData` object to the client.

As you might imagine from the brief, but complete, description of `RockResultsBean.getData()`, `RockResultsData` is the real workhorse object. In its constructor, it loads all the fields for the getters, calculates necessary totals and a kilogram equivalent for the total rock weight, as well as the average weights. The other methods are somewhat involved, but basically set up the lines for client display.

## Deploying the Survey Results application

Here are the steps necessary to deploy the Survey Results application:

1. Compile Java source files in the `SurveyResults` directory, or copy the `.class` files from the `prod` folder.
2. Start J2EE and `deploytool`.
3. Ensure that the Cloudscape database is started.
4. Create new application:
  - From the menu, select File, New, Application.
  - Browse to the `SurveyResults` folder.
  - Key `SurveyResultsApp.ear` in for the file name.
  - Click New Application.
  - Application Display name is `SurveyResultsApp` by default; click OK.
5. Create new Enterprise JavaBean component:
  - Ensure that Application `SurveyResultsApp` is selected.
  - From the menu, select File, New, Enterprise Bean -- Intro display appears; click Next.
  - Select Create New JAR File In Application.
  - Ensure that `SurveyResultsApp` is selected in the drop-down menu.
  - In JAR Display Name, key `SurveyResultsJAR`.
  - Click Edit.
  - Select the `.class` files that comprise the bean. These are:
    - `RockResultsBean.class`

- RockResultsData.class
  - RockResultsRemote.class
  - RockResultsRemoteHome.class
  - RockSurveyConstants.class
- Click Add; click OK; click Next.
  - Under Bean Type, click Session and Stateless.
  - In the Enterprise Bean Class combo box, select the bean implementation: RockResultsBean.
  - In Enterprise Bean Name, accept RockResultsBean.
  - Select the corresponding interfaces in the combo boxes:
    - For Remote Home Interface, select: RockResultsRemoteHome.
    - For Remote Interface, select: RockResultsRemote.
  - Click Next; click Finish.
  - SurveyResultsJAR and RockResultsBean now appear under SurveyResultsApp.
  - Click the Resource Refs tab; click Add.
  - Under Coded Name, enter "jdbc/gsejbDB". Leave Type as "javax.sql.DataSource", Authentication as "Container", and Sharable checked.
  - Select SurveyResultsJAR, then click the JNDI Names tab.
  - Under EJBs, key ejb/RockResultsBean next to RockResultsBean.
  - Under References, enter "jdbc/gsejbDB" in the JNDI Name column.
6. Create new Web component:
- Ensure that Application SurveyResultsApp is selected.
  - From the menu, select File, New, Web Component -- Intro display appears; click Next.
  - Select Create New WAR File In Application.
  - Ensure that SurveyResultsApp is selected in the drop-down menu.
  - In WAR Display Name, key SurveyResultsWAR.

- Click Edit.
  - Select the files needed for the component. These are:
    - The images directory
    - index.jsp
    - RockResults.jsp
  - Click Add; click OK; click Next.
  - Select JSP for the type of Web component, then click Next.
  - In the JSP filename combo box, select index.jsp.
  - In Web Component Name, allow the default -- index.
  - Click Next. There are no Initialization Parameters. Click Next.
  - Under Component Aliases, click Add, enter "/index" (without quotes).
  - Click Next, then click Finish.
  - SurveyResultsWAR now appears under SurveyResultsApp.
  - Select SurveyResultsWAR, then select the EJB Refs tab and click Add.
  - When it invokes the lookup method, the Web client refers to the home of an enterprise bean like this:

```
Object objRef =  
ic.lookup( "java:comp/env/ejb/RockResultsBean" );
```

So, in the Coded Name column, enter ejb/RockResultsBean.
  - Accept Session for Type and Remote for Interface.
  - For Home Interface, enter RockResultsRemoteHome.
  - In the Local/Remote Interface column, enter RockResultsRemote.
  - At the bottom, select "ejb-jar-ic.jar#RockResultsBean" from the Enterprise Bean Name drop-down menu. Then select JNDI Name and select ejb/RockResultsBean from the drop-down menu.
7. We can now actually deploy the application:
- Select SurveyResultsApp.
  - Select Tools, Deploy from the menu.
  - Under Object To Deploy, ensure that SurveyResultsApp is selected.
  - Under Target Server, we are using localhost. Be sure that is selected.

- Select the Return Client JAR check box.
- Select Save object before deploying.
- Click Next.
- Verify that the JNDI name under both Application and References is `ejb/RockResultsBean` for `RockResultsBean`, and `"jdbc/gsejbDB"` for the Resource Reference. Click Next.
- Key `"/SurveyResults"` in the context root.
- Click Next, then Finish.

The Deployment Progress dialog will display. Wait until the progress bars are complete and the Cancel button changes to OK, then click OK. The `SurveyResultsApp` application is deployed.

If you look in the `SurveyResults` folder, you will see that it now contains `SurveyResultsApp.ear` and `SurveyResultsAppClient.jar`. To run the `SurveyResultsApp`, start both J2EE and Cloudscape, then start up your browser and enter `http://localhost:8000/Alice` as the target URL. When the Alice's World home page appears, click on the "Alice's Survey Results" link.

---

## Section 7. Wrapup

### Summary

The Java 2 Platform, Enterprise Edition, provides many services to support industrial-strength enterprise applications. At the core of most of these applications are EJB components of one sort or another. Throughout this tutorial, the goal has been to give you enough background to understand how to create and work with the variety of beans available as of the EJB 2.0 specification. The tutorial has discussed, and provided examples for, stateless and stateful session beans, bean-managed and container-managed persistence with entity beans, and using the Java Message Service with message-driven beans. Along the way, the tutorial also touched on JSP technology, JDBC, and transactions.

Now, at the end of the tutorial with a number of new skills in your toolkit, you should keep in mind that complete books have been written on each topic we've covered here. The tutorial attempts to provide solid coverage of the most important areas, but, inevitably, some were left out. In addition, in order to focus on the topic at hand,

many details were omitted to keep the material to a manageable size. While there's no substitute for (educated) hands-on programming, you can also find yourself beating your head against the walls that will surely arise. Making use of the provided [Resources](#), as well as other reading and discussions, will help you in the move from beginning to intermediate and, finally, to expert J2EE and EJB developer. The message is that, with persistence, you can eventually find yourself in that expert state.

---

## Section 8. Appendices

### Appendix A: Installing and running J2EE

This tutorial uses the J2EE 1.3.1 Reference Implementation, which is included in the J2EE SDK download. If you use a different J2EE application server, you should ensure that it is compliant with J2EE 1.3.1 and the EJB 2.0 specification. The reference implementation requires J2SE 1.3.1\_02 or later.

You can download the J2EE SDK from Sun's J2EE site; see [Resources](#). While there, be sure to read and follow the Installation Instructions for your platform. In addition, the tutorial uses the standard `javac` command to compile the example code, so you will need to *put `j2ee.jar` in your classpath*. `j2ee.jar` is located in `J2EE_HOME/lib`. Once the SDK is installed, you should review the Release Notes and the Configuration Guide. API documentation and J2EE SDK Tools instructions are also provided. These are all HTML pages that you can access from `J2EE_HOME/doc/index.html`.

#### Starting and stopping J2EE

To run J2EE, enter:

```
j2ee -verbose
```

at the command line. To stop `j2ee`, enter:

```
j2ee -stop
```

at the command line.

#### Starting and stopping Cloudscape

To run the Cloudscape database engine, which is the default for J2EE, enter:

```
cloudscape -start
```

at the command line. To stop Cloudscape, enter:

```
cloudscape -stop
```

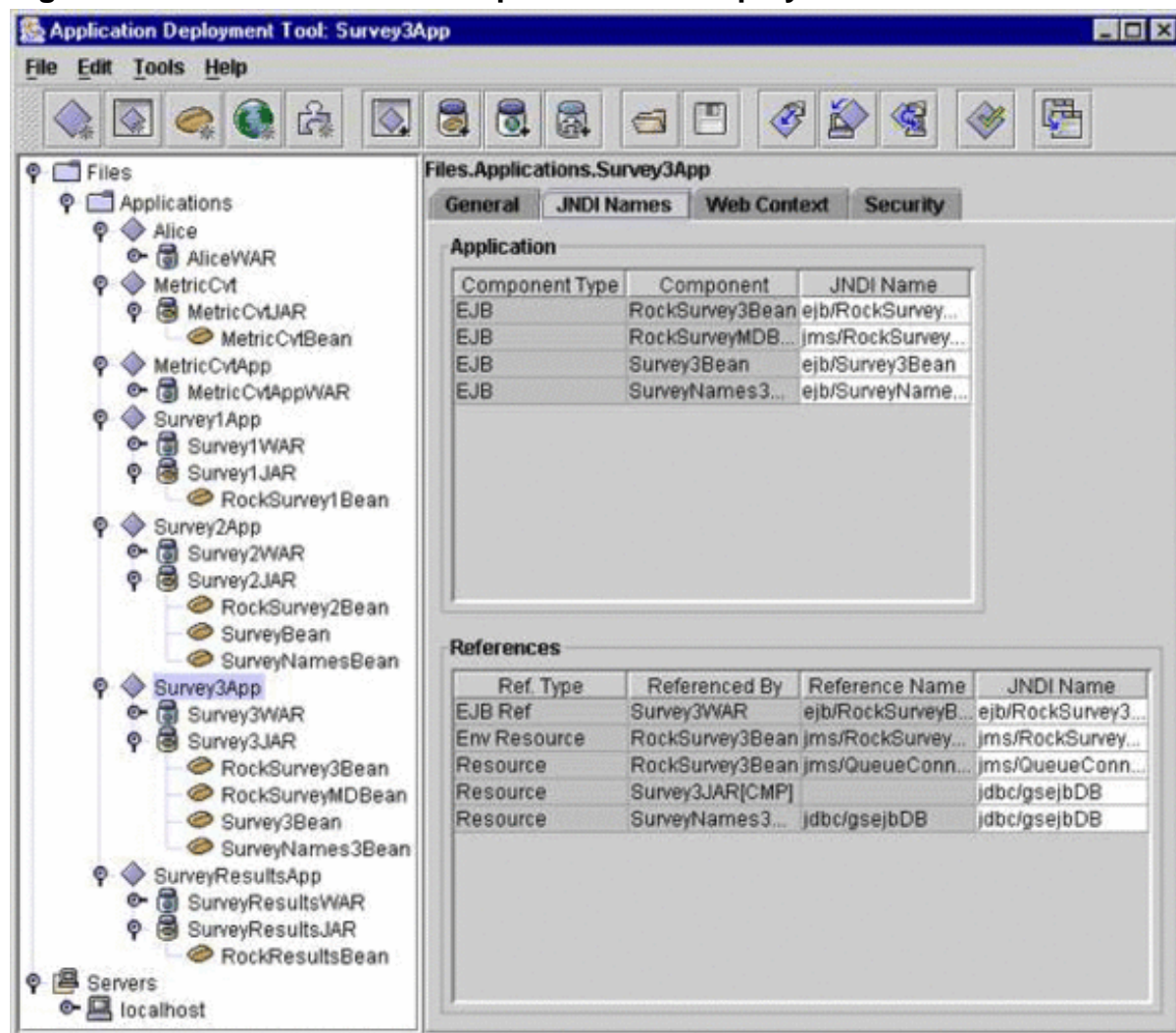
at the command line.

## Starting deploytool

To run deploytool (see [Appendix B: Deployment](#) ), enter `deploytool` at the command line. Exit as you would any other GUI application.

## Appendix B: Deployment

Figure 7. The J2EE Reference Implementation deploytool



Enterprise JavaBean component configuration is defined in *deployment descriptors*, which are XML documents. Tools like `deploytool` (pictured in Figure 7) generate or update the descriptors when you set the configuration elements. For example, here's the relatively brief `ejb-jar.xml` deployment descriptor generated for `MetricCvtJAR` in [Deploying the Metric Converter example](#):

```
<ejb-jar>
  <display-name>MetricCvtJAR</display-name>
  <enterprise-beans>
    <session>
      <display-name>MetricCvtEJB</display-name>
      <ejb-name>MetricCvtEJB</ejb-name>
      <home>MetricCvtRemoteHome</home>
      <remote>MetricCvtRemote</remote>
      <local-home>MetricCvtLocalHome</local-home>
      <local>MetricCvtLocal</local>
      <ejb-class>MetricCvtBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
      <security-identity>
        <description></description>
        <use-caller-identity></use-caller-identity>
      </security-identity>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Most developers decide rather quickly that they would prefer not to do this process manually.

J2EE deployment can be a daunting task (enough so that the EJB specification defines a role for an expert deployer) for two reasons. The first is that J2EE provides an enormous amount of declarative functionality for the developer. While this is a good thing, the downside is that the functionality must be declared, and with so many options, the task quickly becomes very complicated. The second is that while portions of packaging, like `ejb` and `web` descriptor elements, are mandated by the specification, the overall deployment process is not standardized; each vendor provides its own means of deployment and requires additional associated proprietary descriptors.

`deploytool` hides this fact of J2EE life somewhat by generating both the standard and necessary proprietary descriptors. You can't really tell from the settings which are standard and which are proprietary to the Reference Implementation. For the curious, a look into the `.ear` file generated by `deploytool` will reveal that the J2EE RI creates its own `sun-j2ee-ri.xml` file that begins with a `<j2ee-ri-specific-information>` tag -- not particularly encouraging for portability.

The tutorial walks you through setting up the deployment descriptors and actual deployment with `deploytool` after the discussion for each example. If you make changes to the files referenced by the descriptors (that is, if you add, remove, or recompile them) and need to redeploy, first select "Update Files" from the Tools menu, then select "Deploy". Alternatively, you can use the "Update and Redeploy" selection.

The following lists, taken from the J2EE "XML DTD for the EJB 2.0 deployment descriptor" (see [Resources](#)), show the elements that can be described for EJB

components.

- The entity element declares an *entity bean*. The declaration consists of:
  - An optional description
  - An optional display name
  - An optional small icon file name
  - An optional large icon file name
  - A unique name assigned to the enterprise bean in the deployment descriptor
  - The names of the entity bean's remote home and remote interfaces, if any
  - The names of the entity bean's local home and local interfaces, if any
  - The entity bean's implementation class
  - The entity bean's persistence management type
  - The entity bean's primary key class name
  - An indication of the entity bean's reentrancy
  - An optional specification of the entity bean's cmp-version
  - An optional specification of the entity bean's abstract schema name
  - An optional list of container-managed fields
  - An optional specification of the primary key field
  - An optional declaration of the bean's environment entries
  - An optional declaration of the bean's EJB references
  - An optional declaration of the bean's local EJB references
  - An optional declaration of the security role references
  - An optional declaration of the security identity to be used for the execution of the bean's methods
  - An optional declaration of the bean's resource manager connection factory references
  - An optional declaration of the bean's resource environment references
  - An optional set of query declarations for finder and select methods for

an entity bean with `cmp-version 2.x`.

- The message-driven element declares a *message-driven bean*. The declaration consists of:
  - An optional description
  - An optional display name
  - An optional small icon file name
  - An optional large icon file name
  - A name assigned to the enterprise bean in the deployment descriptor
  - The message-driven bean's implementation class
  - The message-driven bean's transaction management type
  - An optional declaration of the message-driven bean's message selector
  - An optional declaration of the acknowledgment mode for the message-driven bean if bean-managed transaction demarcation is used
  - An optional declaration of the intended destination type of the message-driven bean
  - An optional declaration of the bean's environment entries
  - An optional declaration of the bean's EJB references
  - An optional declaration of the bean's local EJB references
  - An optional declaration of the security identity to be used for the execution of the bean's methods
  - An optional declaration of the bean's resource manager connection factory references
  - An optional declaration of the bean's resource environment references.
- The session element declares an (sic) *session bean*. The declaration consists of:
  - An optional description
  - An optional display name
  - An optional small icon file name
  - An optional large icon file name

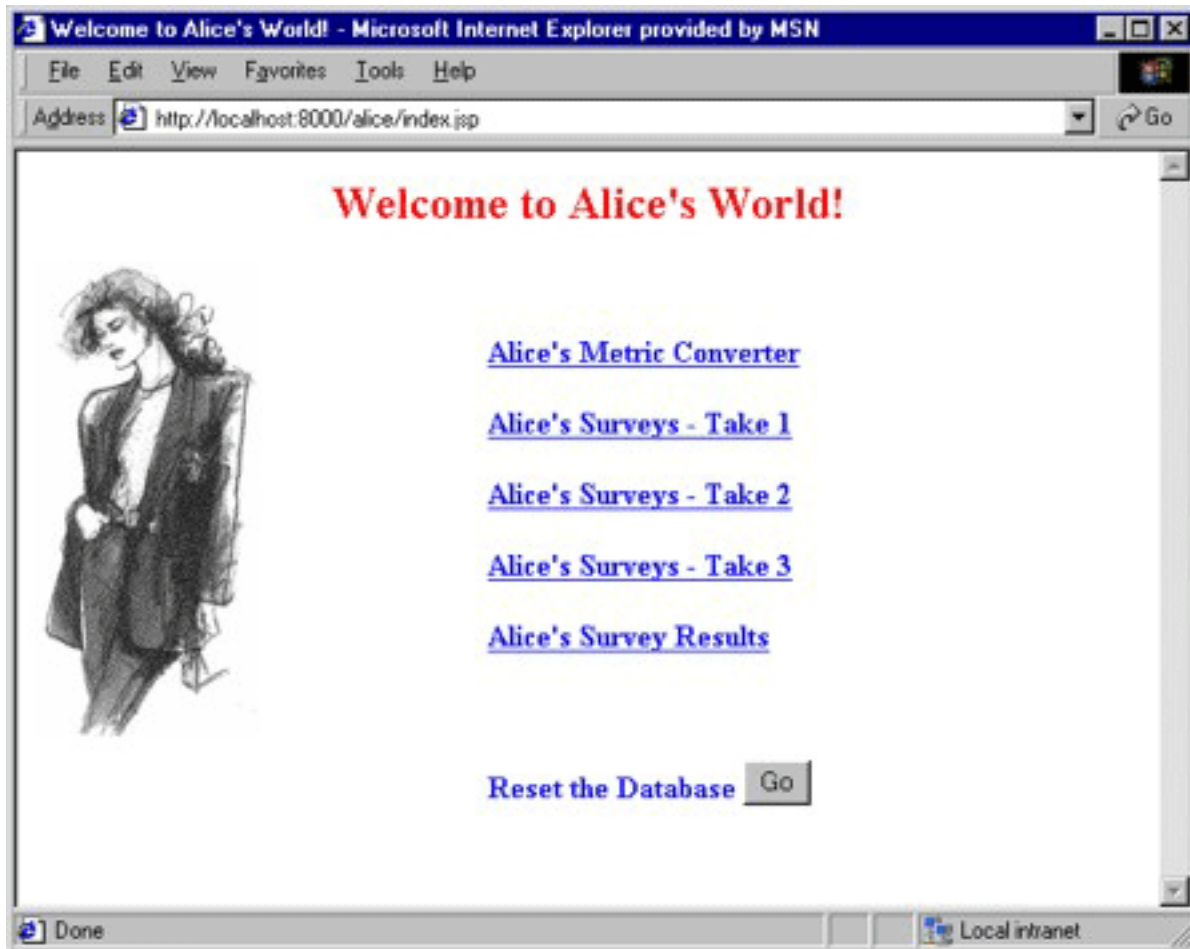
- A name assigned to the enterprise bean in the deployment description
- The names of the session bean's remote home and remote interfaces, if any
- The names of the session bean's local home and local interfaces, if any
- The session bean's implementation class
- The session bean's state management type
- The session bean's transaction management type
- An optional declaration of the bean's environment entries
- An optional declaration of the bean's EJB references
- An optional declaration of the bean's local EJB references
- An optional declaration of the security role references
- An optional declaration of the security identity to be used for the execution of the bean's methods
- An optional declaration of the bean's resource manager connection factory references
- An optional declaration of the bean's resource environment references.

You can view the generated deployment descriptors at any time by selecting the appropriate EJB JAR file, then choosing Tools->Descriptor Viewer from the `deploytool` menu.

The tutorial uses the J2EE RI `deploytool` and takes the path of deployment by example, for the reasons above and the simple fact that explaining every option could take another complete tutorial. With the number of applications and beans in the tutorial, the developer should pick up the basic patterns fairly quickly. For more detailed information about deployment, see the J2EE and EJB specifications, along with the other links in [Resources](#).

## Appendix C: About the example applications

### Figure 8. The Alice's World site



When trying to decide on example code to illustrate the material presented in the tutorial, I wanted something that was conceptually simple, but had the functionality required in real-world applications. In addition, I felt the examples should reinforce and add to the tutorial text, and possibly bring some amusement to what can be dead boring, detailed technical information. Last, I wanted *no shopping carts!* While shopping carts are great, sometimes it seems that every article or tutorial uses them for example apps. There actually are other things being done with J2EE.

The result is the Alice's World site. Alice's World presents three applications: a metric/English measure conversion program; a survey program; and a program to display the results of the survey. These applications exhibit the functionality required in nearly every real program: gather data, process it, store it, and display the resulting information. The client applications are all based on servlets and JSP pages. While a detailed explanation of the programs is "beyond the scope of the tutorial" -- that favorite technical author's excuse, but the tutorial is, after all, about EJB technology -- I felt that the applications warranted a section that described the rationale and flow of the examples. That's particularly true for the Survey application, because it is revisited three times in the tutorial -- much as happens while

developing an application in the real world. Each revision adds to or refines the application until it is complete. Each revision also introduces a different type of bean into the mix. You can see the applications in action before, or instead of, deploying them yourself at conceptGO's [Community](#) page.

The bean and client applications for the Metric Converter are deployed separately. This is what should happen in the real world: EJB components, and all of our code really, should aim for decoupling and reusability as much as possible. In the Survey and Survey Results applications, the client and associated beans are deployed together, mostly for administrative reasons, but the beans can still be accessed by other clients.

You may notice that the application tasks could have been done without using EJB components or J2EE at all, but the design seemed justified for the tutorial. If I had included a "J2EE-worthy" application, you would be reading this six months later and 600 pages longer. In that scenario, the tutorial may well have lost its focus. However, you should keep this critical mind-set when designing your own applications, to avoid the "When you have a hammer..." syndrome, and to use the proper tools and techniques for useful, functional, efficient, and effective applications.

## Deploying the Alice application

So, remember Alice? The Alice's World page is a small JSP page that primarily invokes the sample applications. If there had not been the need to have a vehicle to execute the program that creates and resets the database, it could have been just an HTML page. It does provide an opportunity to take our first look at deploying a J2EE application. If you prefer, you can take the easy route: after downloading the source code, start J2EE and `deploytool`, copy `Alice.ear` from the `Alice/prod` directory to the `Alice` directory, and open `Alice.ear` in `deploytool`. Select the "Alice" application, then "Deploy" from the Tools menu. You can also deploy any or all of the other applications, including the beans, using the same method: copy the appropriate EAR from the example directory's `prod` directory, open the EAR in `deploytool`, then select the "Deploy" menu item, and continue at the step labeled "Deployment".

To deploy the Alice application from scratch, follow these directions:

1. Compile `CreateRSTables.java` (or copy the `.class` file from the `prod` folder.)
2. Start J2EE and `deploytool`.
3. Create new application:
  - From the `deploytool` menu, select File, New, Application.

- Browse to the Alice folder.
  - Key Alice.ear in file name, click New Application.
  - Application Display name is "Alice" by default; click OK.
4. Create new Web component:
- Ensure that Application Alice is selected.
  - From the menu, select File, New, Web Component -- Intro display appears; click Next.
  - Select Create New WAR File In Application.
  - Ensure that Alice is selected in the drop-down menu.
  - In WAR Display Name, key "AliceWAR", then click Edit.
  - Select the files necessary for the application:
    - The images directory
    - CreateRSTables.class
    - index.jsp
  - Click Add; click OK; click Next.
  - Select JSP for the type of Web component.
  - Click Next.
  - From the JSP Filename combo, select "index.jsp", and leave the Web component name as "index".
  - Click Next; click Finish.
  - AliceWAR now appears under Alice.
  - Select AliceWAR, then select the Resource Refs tab, and click Add.
  - Under Coded Name, enter "jdbc/gsejbDB". Leave Type as "javax.sql.DataSource", Authentication as "Container", and Sharable checked.
  - Select the Alice application.
  - Click the JNDI Names tab. In JNDI Name under References, enter "jdbc/gsejbDB".
  - Click Web Context from the tabbed pane.
  - Enter "/Alice", click General.

5. **Deployment.** We can now actually deploy the Alice application:
  - Ensure that the Alice application is selected.
  - From the menu, select Tools, Deploy.
  - Under Object To Deploy, ensure that Alice is selected.
  - Under Target Server, we are using localhost. Be sure that is the selection.
  - Unselect the Return Client JAR check box.
  - Select Save object before deploying.
  - Click Next.
  - Verify that the JNDI name under References is "jdbc/gsejbDB"; click Next.
  - Ensure that "/Alice" is in the context root entry.
  - Click Next, then Finish.

The Deployment Progress dialog will display. Wait until the progress bars are complete and the Cancel button changes to OK, then click OK. The Alice application is deployed.

If you look in the Alice folder, you will see that it now contains Alice.ear. To run Alice, start your browser and enter `http://localhost:8000/Alice` as the target URL; you should see the Alice home page shown in Figure 8. Understand that none of the example applications have been deployed yet, so you can't do anything but look at the Alice display for now. The other applications will be deployed in the appropriate example sections of the tutorial.

## A word about naming conventions

The tutorial uses the following naming conventions for consistent tracking of all the EJB components, application pieces, and deployment names. You may use the convention for your own code or choose another; there's nothing blessed here and a real-world set of naming conventions should be considerably more rigorous and flexible. To avoid major headaches, *do* be sure that whatever you use is consistent and provides a logical way to track the component parts.

The EJB 2.0 specification "recommends, but does not require, that all references to other enterprise beans be organized in the `ejb` subcontext of the bean's environment." For that reason, the tutorial prepends `ejb/` to the bean JNDI names. The actual mechanism for JNDI loading is not addressed by the specification, so this

is definitely a vendor-proprietary area to check when deploying your own applications.

I'll use the bean name for the one used in the Metric Converter program, which is `MetricCvt`, to illustrate:

- The bean class: **MetricCvtBean**
- The remote component interface: **MetricCvtRemote**
- The remote home interface: **MetricCvtRemoteHome**
- The local component interface: **MetricCvtLocal**
- The local home interface: **MetricCvtLocalHome**
- In this case, some constants were also defined in an interface to share among the bean and the application: **MetricCvtConstants**

For the initial page of the application: **index.jsp**. I am aware of the downsides to using "index" as a page name, but it does resolve some mapping issues.

Since the tutorial at times deploys beans and applications separately, deployment names for bean "applications" are:

- The application EAR name: **MetricCvt**
- The EJB JAR name: **MetricCvtJAR**
- The Enterprise JavaBean component name: **MetricCvtBean**

Deployment names for the real applications, all Web components here, are:

- The application EAR name: **MetricCvtApp**
- The WAR JAR name: **MetricCvtWAR**

## The Alice files

These are the source, class, and JAR files used in the Alice application. They reside in the Alice directory in the code download.

**Table 4. Source, class, and JAR files used in the Alice application**

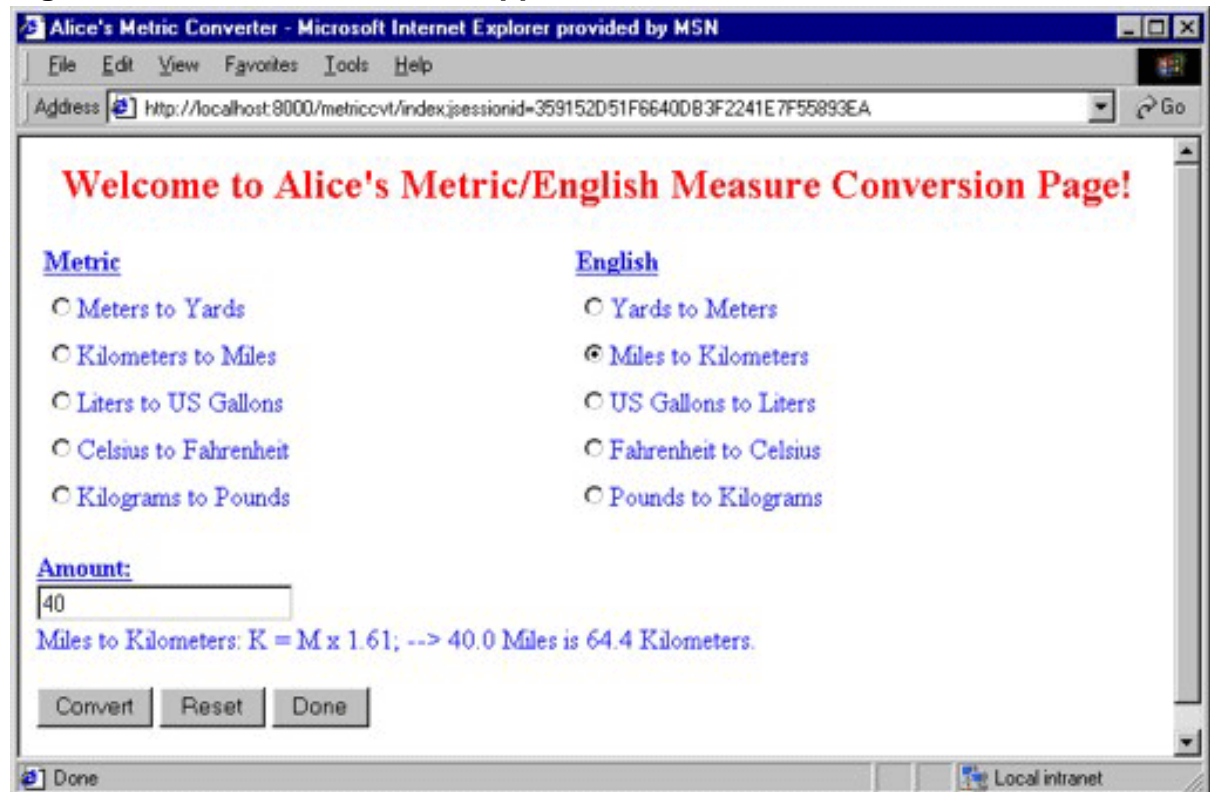
CreateRSTables.java	Straight Java JDBC application to build and refresh the database and tables for the Rock Survey application.
images	The images directory.

index.jsp	The Alice home page. Provides links to the other apps and runs CreateRSTables.
prod	The production directory. Contains source and compiled classes, Alice EAR and JARs.

## The Metric Converter application

The Metric Converter is a straightforward one-page application that converts between U.S. English and metric measures based on the type of conversion selected and the input value. It calls on a stateless session bean to perform the calculation and return a formatted answer string. When the user clicks Done, the servlet session is invalidated and control is returned to the Alice's World home page.

**Figure 9. The Metric Converter application**



The stateless session bean and the application are deployed separately. An interface with `static final` constants is used to ensure that both the client and the bean use the same values to identify the conversion type. For the details, see [Example: A metric conversion program](#).

## The Metric Converter files

These are the source, class, and JAR files used in the Metric Converter application. They reside in the MetricCvt directory in the code download.

**Table 5. Source, class, and JAR files used in the Metric Converter application**

index.jsp	The application page. Allows selection of conversion type, and displays the result.
prod	The production directory. Contains source and compiled classes, MetricCvt and MetricCvtApp EAR and JARs.
MetricCvtBean.java	Stateless session bean. Handles conversions between metric and English measures.
MetricCvtConstants.java	Useful constants for the Metric Converter application.
MetricCvtRemote.java	The remote component interface for <code>MetricCvtBean</code> .
MetricCvtRemoteHome.java	The remote home interface for <code>MetricCvtBean</code> .

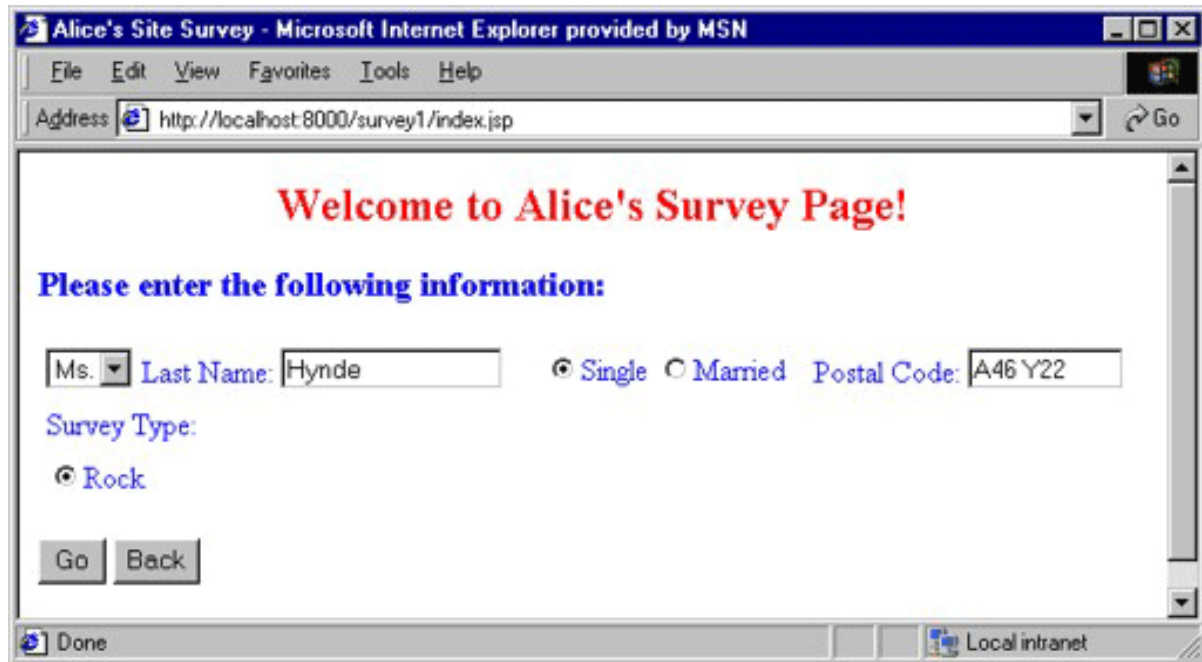
## The Rock Survey application

Alice's Rock Survey is the primary example application in the tutorial. Much as happens in practice, the application goes through several iterations before it is a finished product. Each iteration introduces a different kind of EJB component and functionality to the application.

In [Example: The Rock Survey, take 1](#), the Rock Survey uses a stateful session bean to handle the business methods for collecting the survey information. The session bean also acts as a dummy data store in this version; there is no actual persistence at this point. In [Example: The Rock Survey, take 2](#), an entity bean is added, which uses BMP for persisting the SurveyNames table data. In [Example: The Rock Survey, take 2 \(continued\)](#), another entity bean uses CMP for persistence with the "SurveyBeanTable" table. In the final iteration, [Example: The Rock Survey, take 3](#), a message-driven bean is added to allow for asynchronous updates to the database.

The application begins with a "welcome to the survey" page that collects data appropriate to all surveys and gives the user -- let's call her Chrissie -- an opportunity to return to the home page. At this point, of course, we only have one survey, the Rock Survey.

### Figure 10. Survey application -- Welcome page



The logic for purposes of the example is:

- The "Mr./Ms." salutation allows inference of gender.
- The Last Name entries are saved to determine the most frequent entries.
- Marital Status is directly entered.
- A Postal Code entry with a length of five is assumed to be a U.S. zip code. Any other entry is assumed to denote residents of other countries.

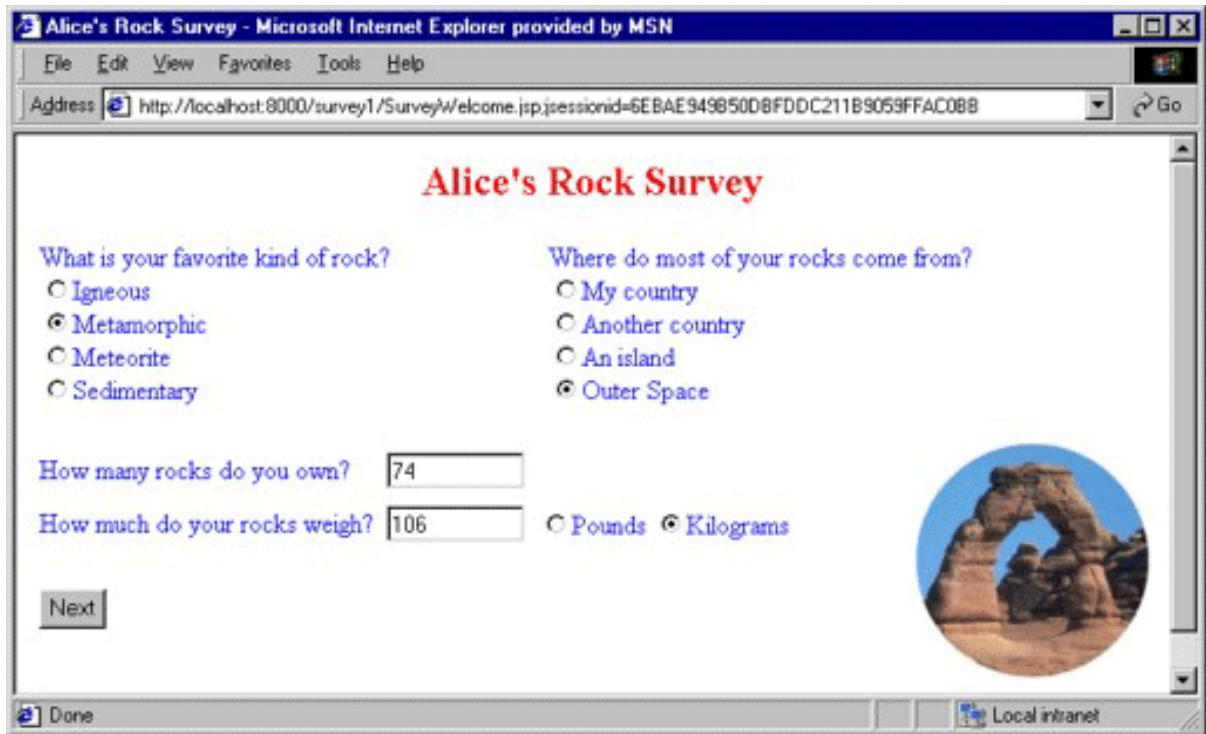
After entering the initial data, Chrissie clicks Go. The Rock Survey welcome page is displayed. This page asks her to verify the data entered and allows a return to the previous page for corrections.

**Figure 11. Rock Survey application -- Welcome page**



If Chrissie is happy with her entries, she clicks Next, which causes the actual Rock Survey page to be displayed.

**Figure 12. Rock Survey application -- Survey page**



Alice's Rock Survey - Microsoft Internet Explorer provided by MSN

File Edit View Favorites Tools Help

Address <http://localhost:8000/survey1/Survey/welcome.jsp;sessionId=6EBAE949850D8FDDC211B9059FFAC08B> Go

## Alice's Rock Survey

What is your favorite kind of rock?


- Igneous
- Metamorphic
- Meteorite
- Sedimentary

Where do most of your rocks come from?

- My country
- Another country
- An island
- Outer Space

How many rocks do you own?

How much do your rocks weigh?   Pounds  Kilograms



Done Local intranet

This page collects and validates the actual survey data. All entries here are straightforward, except that the weight of the rocks is always kept in pounds in the database. Once the entries are made, Chrissie clicks Next, and the Rock Survey "thank you" page is displayed.

**Figure 13. Rock Survey application -- Thank you page**



The thank you page asks her to verify the input data and allows a return to the previous page for corrections. If the entries are satisfactory, she clicks Done, which causes the data to be stored, invalidates the servlet session, and returns to the Alice's World home page.

## The Rock Survey files, take 1

These are the source, class, image, and JAR files used in the Rock Survey, take 1. They reside in the Survey1 directory in the code download.

**Table 6. Source, class, and JAR files used in the Rock Survey, take 1**

images	The images directory.
index.jsp	The initial Survey page. Gathers gender, name, marital, and postal data.
prod	The production directory. Contains source and

	compiled classes, and Survey1App EAR and JARs.
RockSurvey.jsp	The Rock Survey page. Gathers rock data.
RockSurveyBean.java	Stateful session bean. Tracks and manages survey data. This version has a dummy persistence method.
RockSurveyConstants.java	Useful constants for the Rock Survey.
RockSurveyData.java	Survey data class passed to the Web component pages from <code>RockSurveyBean</code> .
RockSurveyExit.jsp	The survey exit page. Invokes <code>RockSurveyBean</code> 's persistence method when Done is clicked.
RockSurveyRemote.java	The remote component interface for <code>RockSurveyBean</code> .
RockSurveyRemoteHome.java	The remote home interface for <code>RockSurveyBean</code> .
SurveyConstants.java	Useful constants for survey personal data.
SurveyWelcome.jsp	The Survey Welcome page. Allows the user to verify personal data.

## The Rock Survey files, take 2

These are the source, class, image, and JAR files used in the Rock Survey, take 2. They reside in the Survey2 directory in the code download.

**Table 7. Source, class, and JAR files used in the Rock Survey, take 2**

images	The images directory from take 1, no change.
index.jsp	The initial Survey page from take 1, no change.
prod	The production directory. Contains source and compiled classes, Survey2App EAR and JARs.
RockSurvey.jsp	The Rock Survey page from take 1, no change.
RockSurvey2Bean.java	Revised and expanded <code>RockSurveyBean</code> from take 1. Includes persistence management with the entity beans.
RockSurveyConstants.class	From take 1, no change.
RockSurveyData.class	From take 1, no change.
RockSurveyExit.jsp	The Exit page from take 1, no change.
RockSurveyRemote.class	From take 1, no change. Reused as the remote interface for <code>RockSurvey2Bean</code> .
RockSurveyRemoteHome.class	From take 1, no change. Reused as the home

	interface for <code>RockSurvey2Bean</code> .
<code>SurveyBean.java</code>	Entity bean to handle "SurveyBeanTable".
<code>SurveyConstants.class</code>	From take 1, no change.
<code>SurveyLocal.java</code>	Local component interface for <code>SurveyBean</code> .
<code>SurveyLocalHome.java</code>	Local home interface for <code>SurveyBean</code> .
<code>SurveyNamesBean.java</code>	Entity bean to handle the SurveyNames table.
<code>SurveyNamesKey.java</code>	Primary key class for the SurveyNames table.
<code>SurveyNamesLocal.java</code>	Local component interface for <code>SurveyNamesBean</code> .
<code>SurveyNamesLocalHome.java</code>	Local home interface for <code>SurveyNamesBean</code> .
<code>SurveyWelcome.jsp</code>	The Survey Welcome page from Take 1, no change.

## The Rock Survey files, take 3

These are the source, class, image, and JAR files used in the Rock Survey, take 3. They reside in the Survey3 directory in the code download.

**Table 8. Source, class, and JAR files used in the Rock Survey, take 3**

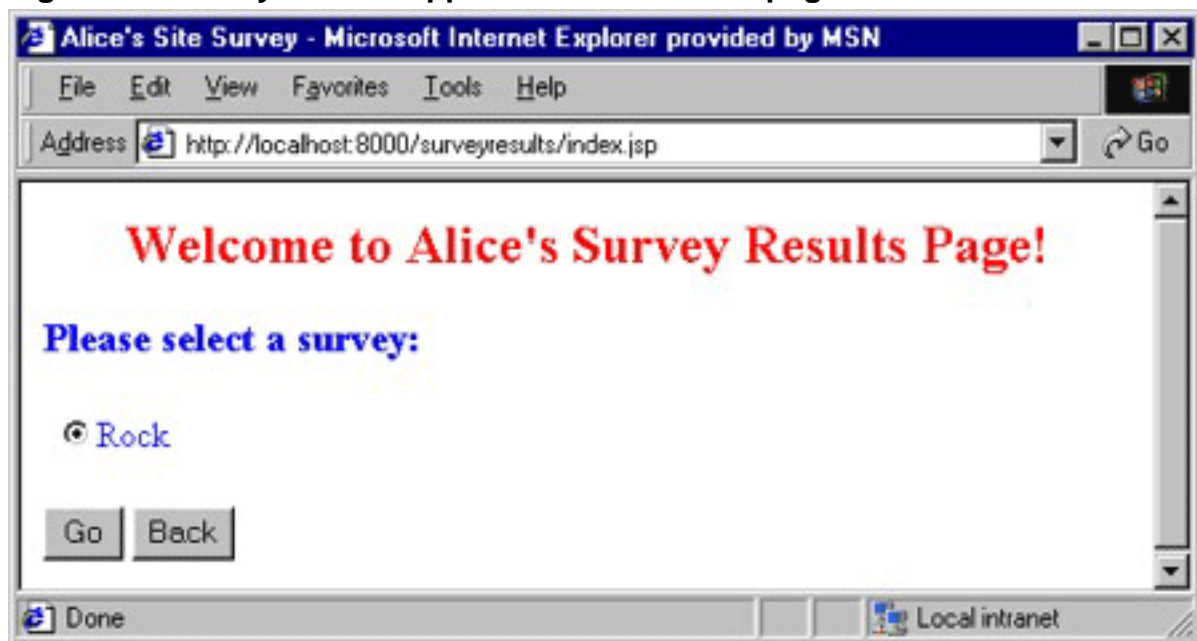
<code>images</code>	The images directory from take 1, no change.
<code>index.jsp</code>	The initial Survey page from take 1, no change.
<code>prod</code>	The production directory. Contains source and compiled classes, Survey3App EAR and JARs.
<code>RockSurvey.jsp</code>	The Rock Survey page from take 1, no change.
<code>RockSurvey3Bean.java</code>	<code>RockSurvey2Bean</code> from take 2, with persistence removed and modified to send messages to <code>RockSurveyQueue</code> .
<code>RockSurveyMDBean.java</code>	The message-driven bean. Receives messages from <code>RockSurveyQueue</code> , manages database updates with persistence operations initially in <code>RockSurvey2Bean</code> from take 2.
<code>RockSurveyConstants.class</code>	From take 1, no change.
<code>RockSurveyData.class</code>	From take 1, no change.
<code>RockSurveyExit.jsp</code>	The Exit page from take 1, no change.
<code>RockSurveyRemote.class</code>	From take 1, no change. Reused as the remote interface for <code>RockSurvey3Bean</code> .
<code>RockSurveyRemoteHome.class</code>	From take 1, no change. Reused as the home interface for <code>RockSurvey3Bean</code> .
<code>SurveyBean.class</code>	From take 2, no change. Entity bean to handle

	"SurveyBeanTable".
SurveyConstants.class	From take 1, no change.
SurveyLocal.class	From take 2, no change. Local component interface for SurveyBean.
SurveyLocalHome.class	From take 2, no change. Local home interface for SurveyBean.
SurveyNamesBean.class	From take 2, no change. Entity Bean to handle the SurveyNames table.
SurveyNamesKey.class	From take 2, no change. Primary key class for the SurveyNames table.
SurveyNamesLocal.class	From take 2, no change. Local component interface for SurveyNamesBean.
SurveyNamesLocalHome.class	From take 2, no change. Local home interface for SurveyNamesBean.
SurveyWelcome.jsp	The Survey Welcome page from Take 1, no change.

## The Survey Results application

The Survey Results application displays the information and statistics gathered from Alice's surveys.

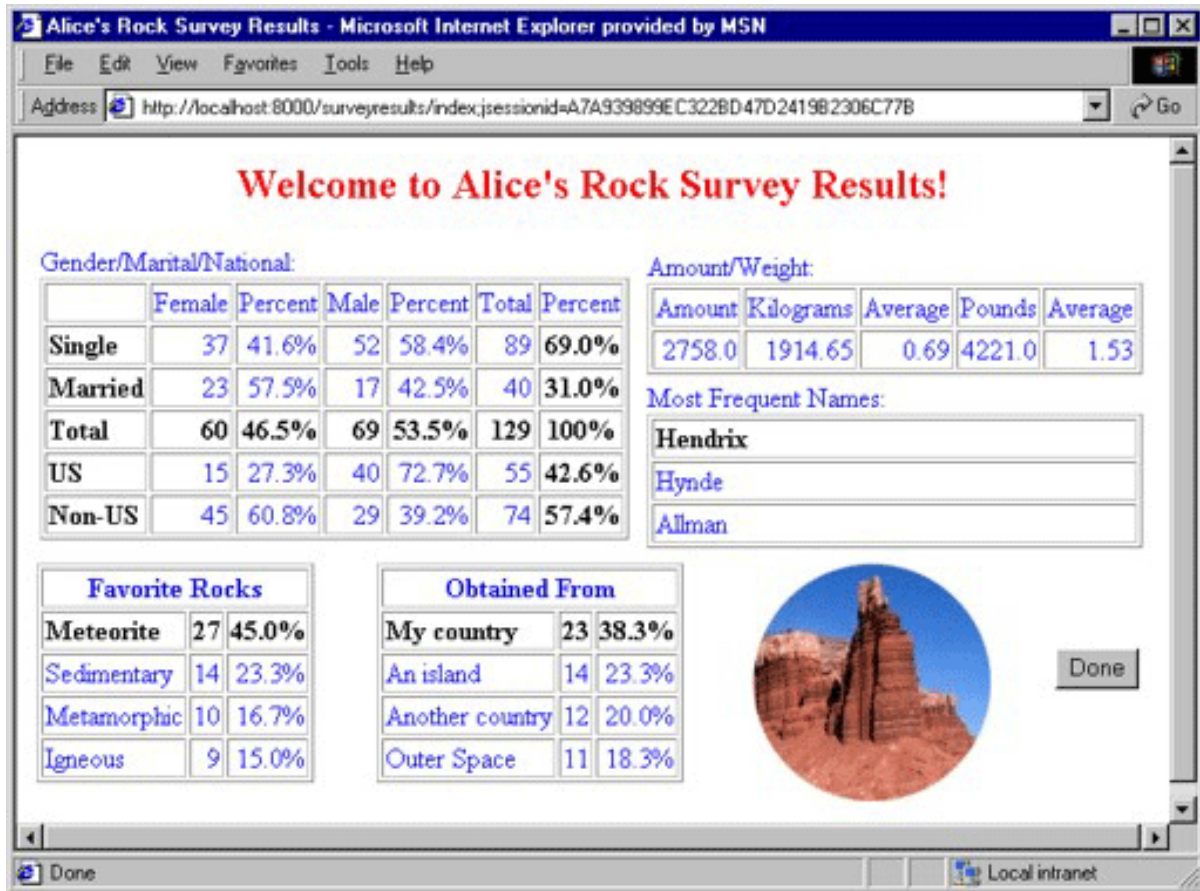
**Figure 14. Survey Results application -- Welcome page**



At this point there is only one survey, the Rock Survey, so Chrissie just clicks Go,

and the Rock Survey Results page is displayed.

**Figure 15. Rock Survey Results application -- Display page**



The basis for the displayed information is explained in [The Rock Survey application](#). When Chrissie is finished viewing the results, she clicks Done, which invalidates the servlet session and returns her to the Alice's World home page.

## The Survey Results files

These are the source, class, image, and JAR files used in the SurveyResults application. They reside in the SurveyResults directory in the code download.

**Table 9. Source, class, and JAR files used in the SurveyResults application**

images	The images directory.
index.jsp	The initial SurveyResults page. Welcomes the user to the application and offers a choice among all available surveys.
prod	The production directory. Contains source and

	compiled classes, SurveyResultsApp EAR and JARs.
RockResults.jsp	The Rock Survey results page. Displays all statistics and information.
RockResultsBean.java	Stateful session bean. Accesses the datastore and provides the <code>RockResultsData</code> object to the client.
RockResultsData.java	Survey statistics and information class passed to the Web component pages from <code>RockResultsBean</code> .
RockResultsRemote.java	The remote component interface for <code>RockResultsBean</code> .
RockResultsRemoteHome.java	The remote home interface for <code>RockResultsBean</code> .
RockSurveyConstants.class	Useful constants for the Rock Survey. Unchanged from <a href="#">Example: The Rock Survey, take 1</a> .

## Appendix D: What's new in the EJB 2.1 specification?

The changes and additional services in J2EE 1.4, and therefore the EJB 2.1 specification, primarily focus on Web services. The major new goals and capabilities for EJB 2.1 are:

- **Support for Web services.** A new interface for Web services endpoints has been added, allowing stateless session bean implementations. All beans can access external Web services.
- **A container-managed timer service.** The service "provides a coarse-grained, transactional time-based event notifications" (sic).
- **Generalized architecture for message-driven beans.** MDBs can access various messaging types in addition to JMS messages.
- **Enhanced EJB Query Language.** EJB QL now has support for various aggregate and scalar functions, and results may be ordered.

For further information, see [Resources](#).

# Resources

## Learn

- Read the [J2EE 1.3.1 Installation Instructions](#) to ensure that your J2EE Reference Implementation is properly set up.
- Download the [Enterprise JavaBeans Technology Specifications](#).
- Get Sun's J2EE Reference Implementation tutorial, [The J2EE Tutorial](#), for additional Enterprise JavaBean technology and J2EE examples.
- *Designing Enterprise Applications with the J2EE Platform* (Addison-Wesley, 2002) by the J2EE Enterprise Team (also available [online](#) ) provides a high-level view with proven principles for architecting and developing J2EE applications.
- As J2EE technology has matured, a number of best practices have been documented as design patterns. The [Core J2EE Pattern Catalog](#) presents a subset of these patterns to help guide the design process.
- To learn about the elements that make up EJB deployment descriptors, the bible is the [XML DTD for the EJB 2.0 deployment descriptor](#).
- For a good explanation of the rationale for the JNDI environment naming context (ENC), see [Best Practice: Use java:comp to Locate EJBs and Increase Application Portability](#).
- Looking for considerations and tips for real-world applications? Take a look at [EJB performance tips](#).
- For guidelines on using EJB technology, see Brett McLaughlin's *developerWorks* column, [EJB best practices](#) .
- For information on using JDBC with complete examples, see Joe Sam Shirah's [JDBC 2.0 Fundamentals Short Course](#).
- For introductory information about transactions, the Java Transaction Service (JTS), and the Java Transaction API (JTA), see Brian Goetz's "Understanding JTS" series:
  - " [An introduction to transactions](#) " ( *developerWorks*, March 2002)
  - " [The magic behind the scenes](#) " ( *developerWorks*, April 2002)
  - " [Balancing safety and performance](#) " ( *developerWorks*, May 2002)
- For a good start with Java Message Service programming, see Willy Farrell's tutorial, " [Introducing the Java Message Service](#) " ( *developerWorks*, August 2001).

- Sun also has a JMS tutorial, "[Java Message Service API Tutorial and Reference: Messaging for the J2EE Platform](#)", written by members of the J2EE Team, including the author of the JMS specification. An [online version](#) is available.
- Nicholas Whitehead's article "[Implementing vendor-independent JMS solutions](#)" (*developerWorks*, February 2002) discusses some real-world JMS issues and their resolution.
- For an introduction to remote, distributed objects using RMI, CORBA, and ORBs, see "[RMI-IIOP in the enterprise](#)" (*developerWorks*, March 2002) by IBM staffer Damian Hagge.
- Read about [Java RMI over IIOP](#) to learn more about programming distributed applications.
- Java architect Srikanth Shenoy discusses EJB error resolution issues in "[Best practices in EJB exception handling](#)" (*developerWorks*, May 2002).
- If you're interested in a quick cover of EJB Query Language, see [Learning EJB QL](#) by Jeelani Shaik.
- [Enterprise JavaBeans](#) (O'Reilly & Associates, 2001) by Richard Monson-Haefel, now in its third edition, is one of the best known introductory EJB books.
- [Mastering Enterprise JavaBeans](#) (Wiley, 2001) by Ed Roman et al. is another good resource for EJB programming.
- [Building Scalable and High-Performance Java Web Applications Using J2EE Technology](#) (Addison-Wesley, 2001) by Greg Barish broadly discusses many J2EE areas. While on an initial reading, the book's suggestions appear to be mostly common sense, the cumulative result is impressive.
- J2EE 1.4 and the EJB 2.1 specification begin to focus on Web services. To learn more about Web services, [The Java Web Services Tutorial](#) (Addison-Wesley, 2002) by Eric Armstrong, Stephanie Bodoff et al. is a good place to start. This tutorial is also available [online](#).
- To get answers to your questions on any area of the Java language, visit [the Java filter forum](#) on *developerWorks*, moderated by Joe Sam Shirah, and the other forums available from the main [developerWorks Java technology zone](#) page.
- Find hundreds more Java technology resources on the [developerWorks Java technology zone](#).

### Get products and technologies

- Download [gsejbExamples.jar](#) for the complete source code and classes used in this tutorial.

- Download the [Java 2 Platform, Standard Edition \(J2SE\)](#).
- Download the [Java 2 Platform, Enterprise Edition \(J2EE\) SDK and Specification](#). The 1.3.1 FCS Release Software and Documentation is used in the tutorial; later versions should work as well.

## About the author

Joe Sam Shirah

Joe Sam is a contributing developerWorks author.