Sun Java System Mobile Enterprise Platform 1.0 Developer's Guide for Enterprise Connectors

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–3754–10
July 2008

081007@21288

Contents

Preface

This guide explains how to develop Enterprise Connectors for Sun Java System Mobile Enterprise Platform 1.0.

MEP is a comprehensive mobility solution that enables offline data access, data synchronization, and secure access to EIS/EAI applications such as Siebel and SAP.

MEP is based entirely upon open standards, including the following:

- Java Platform, Mobile Edition (Java ME)
- Java Platform, Enterprise Edition (Java EE)
- The dominant industry standard OMA DS, formerly known as SyncML. The specifications for Open Mobile Alliance Data Synchronization V1.1.2 and V1.2.1 are available at `http://www.openmobilealliance.org/Technical/release_program/ds_v112.aspx` and `http://www.openmobilealliance.org/Technical/release_program/ds_v12.aspx`.

Who Should Use This Book

This guide is intended for developers who have experience creating Java applications.

How This Book Is Organized

This book contains the following chapters:

- Chapter 1, “Introduction to Enterprise Connectors”
- Chapter 2, “Creating an Enterprise Connector”
- Chapter 3, “Classes and Methods in the Enterprise Connector Business Object API Package”

Mobile Enterprise Platform Documentation Set

The Mobile Enterprise Platform documentation set is available at `http://docs.sun.com/coll/1780.1`. To learn about Mobile Enterprise Platform, refer to the books listed in the following table.

TABLE P–1 Books in the Mobile Enterprise Platform Documentation Set

Book Title	Description
Sun Java System Mobile Enterprise Platform 1.0 Release Notes	Late-breaking information about the software and the documentation. Includes a comprehensive summary of the supported hardware, operating systems, application server, Java^{TM} Development Kit (JDK^{TM}), databases, and EIS/EAI systems.
Sun Java System Mobile Enterprise Platform 1.0 Architectural Overview	Introduction to the architecture of Mobile Enterprise Platform.
Sun Java System Mobile Enterprise Platform 1.0 Installation Guide	Installing the software and its components, and running a simple application to verify that installation succeeded.
Sun Java System Mobile Enterprise Platform 1.0 Deployment Guide	Deployment of applications and application components to Mobile Enterprise Platform.
Sun Java System Mobile Enterprise Platform 1.0 Developer's Guide for Client Applications	Creating and implementing Java Platform, Mobile Edition (Java ME platform) applications for Mobile Enterprise Platform that run on mobile devices.
Sun Java System Mobile Enterprise Platform 1.0 Developer's Guide for Enterprise Connectors	Creating and implementing Enterprise Connectors for Mobile Enterprise Platform intended to run on Sun Java System Application Server.
Sun Java System Mobile Enterprise Platform 1.0 Administration Guide	System administration for Mobile Enterprise Platform, focusing on the use of the MEP Administration Console.

Application Server Documentation Set

When you install MEP, it is deployed to Sun Java System Application Server 9.1 Update 2.

The Application Server documentation set describes deployment planning and system installation. The Uniform Resource Locator (URL) for Application Server documentation is `http://docs.sun.com/coll/1343.5`. For an introduction to Application Server, refer to the books in the order in which they are listed in the following table.

TABLE P–2 Books in the Application Server Documentation Set

Book Title	Description
Documentation Center	Application Server documentation topics organized by task and subject.

TABLE P–2 Books in the Application Server Documentation Set *(Continued)*

Book Title	Description
Release Notes	Late-breaking information about the software and the documentation. Includes a comprehensive, table-based summary of the supported hardware, operating system, Java Development Kit (JDK), and database drivers.
Quick Start Guide	How to get started with the Application Server product.
Installation Guide	Installing the software and its components.
Deployment Planning Guide	Evaluating your system needs and enterprise to ensure that you deploy the Application Server in a manner that best suits your site. General issues and concerns that you must be aware of when deploying the server are also discussed.
Application Deployment Guide	Deployment of applications and application components to the Application Server. Includes information about deployment descriptors.
Developer's Guide	Creating and implementing Java Platform, Enterprise Edition (Java EE platform) applications intended to run on the Application Server that follow the open Java standards model for Java EE components and APIs. Includes information about developer tools, security, debugging, and creating lifecycle modules.
Java EE 5 Tutorial	Using Java EE 5 platform technologies and APIs to develop Java EE applications.
Java WSIT Tutorial	Developing web applications using the Web Service Interoperability Technologies (WSIT). Describes how, when, and why to use the WSIT technologies and the features and options that each technology supports.
Administration Guide	System administration for the Application Server, including configuration, monitoring, security, resource management, and web services management.
High Availability Administration Guide	Post-installation configuration and administration instructions for the high-availability database.
Administration Reference	Editing the Application Server configuration file, `domain.xml`.
Upgrade and Migration Guide	Upgrading from an older version of Application Server or migrating Java EE applications from competitive application servers. This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications.
Performance Tuning Guide	Tuning the Application Server to improve performance.
Troubleshooting Guide	Solving Application Server problems.
Error Message Reference	Solving Application Server error messages.
Reference Manual	Utility commands available with the Application Server; written in man page style. Includes the `asadmin` command line interface.

Typographic Conventions

The following table describes the typographic changes that are used in this book.

TABLE P–3 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	A placeholder to be replaced with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized (note that some emphasized items appear bold online)	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file.

Symbol Conventions

The following table explains symbols that might be used in this book.

TABLE P–4 Symbol Conventions

Symbol	Description	Example	Meaning
`[ ]`	Contains optional arguments and command options.	`ls [-l]`	The `-l` option is not required.
`{ | }`	Contains a set of choices for a required command option.	`-d {y|n}`	The `-d` option requires that you use either the `y` argument or the `n` argument.
`${ }`	Indicates a variable reference.	`${com.sun.javaRoot}`	References the value of the `com.sun.javaRoot` variable.
-	Joins simultaneous multiple keystrokes.	Control-A	Press the Control key while you press the A key.
+	Joins consecutive multiple keystrokes.	Ctrl+A+N	Press the Control key, release it, and then press the subsequent keys.

TABLE P–4 Symbol Conventions *(Continued)*

Symbol	Description	Example	Meaning
→	Indicates menu item selection in a graphical user interface.	File → New → Templates	From the File menu, choose New. From the New submenu, choose Templates.

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Searching Sun Product Documentation

Besides searching Sun product documentation from the docs.sun.com^{SM} web site, you can use a search engine by typing the following syntax in the search field:

search-term `site:docs.sun.com`

For example, to search for "broker," type the following:

```
broker site:docs.sun.com
```

To include other Sun web sites in your search (for example, java.sun.com, www.sun.com, and developers.sun.com), use `sun.com` in place of `docs.sun.com` in the search field.

Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. To share your comments, go to `http://docs.sun.com` and click Send Comments. In the online form, provide the full document title and part number. The part number is a 7-digit or 9-digit number that can be found on the book's title page or in the document's URL. For example, the part number of this book is 820-3754.

CHAPTER 1

Introduction to Enterprise Connectors

Sun Java System Mobile Enterprise Platform (MEP) includes a library, the Enterprise Connector Business Object (ECBO) API, that enables development of objects that provide an interface between data on mobile client devices and data stored in a database or an EIS/EAI system. This library, in conjunction with the Mobile Client Business Object (MCBO) API , provides a complete solution that allows you to synchronize arbitrary enterprise data between a client device and a database or EIS/EAI system, using a Gateway Engine deployed on the Sun Java System Application Server that acts as an intermediary.

Although the MCBO and ECBO APIs are based on Open Mobile Alliance Data Synchronization (OMA DS), you do not need to know specifics of OMA DS in order to use the APIs. The ECBO API also depends upon Java Content Repository (JCR) technology as defined by JSR-170 (`http://jcp.org/aboutJava/communityprocess/final/jsr170/index.html`), but it hides the complexity of this technology from the developer.

About the Enterprise Connector Business Object (ECBO) API

The Enterprise Connector Business Object (ECBO) API provides classes and methods that allow you to create an Enterprise Connector, which is a Java artifact that consists of extensions of five classes compiled and packaged into a JAR file along with a resource file.

The methods you implement in your connector are called by the MEP libraries deployed on the Gateway Engine. In this respect, the ECBO API is more like a Service Provider Interface (SPI) than an Application Programmer Interface (API).

Each connector defines a particular business object whose data is to be synchronized. It also defines commands and operations that allow the connector to perform Create, Retrieve, Update, and Delete (CRUD) operations on the business objects.

The five classes you must implement are called `BusinessObject`, `BusinessObjectProvider`, `InsertCommand`, `UpdateCommand`, and `DeleteCommand`. The `BusinessObjectProvider` class provides a method to perform the retrieve operation. Chapter 2, "Creating an Enterprise Connector," describes in detail how to implement these classes.

The resource file is an XML file that is used by the underlying JCR implementation. It defines a repository name, a workspace name, and a few node types. All Enterprise Connectors use the same set of node types.

The data to be synchronized takes the form of arbitrary Java objects. The way in which these Java objects are serialized is open-ended and is part of the contract between an Enterprise Connector and its corresponding Java ME client. It is recommended that you use a format that supports data versioning.

Packaging and Deploying Enterprise Connectors

To package an Enterprise Connector, compile the five Java classes and then place them in a JAR file along with the resource file. To see how the sample Enterprise Connectors provided with MEP are packaged, use the `jar` command to view the contents of any of the following files in the `lib` directory of the Application Server domain for MEP:

```
ds-jcr-musicdb.jar
ds-jcr-sap-eway.jar
ds-jcr-siebel-eway.jar
```

The same Enterprise Connector works equally well in a one-tier or two-tier MEP installation. (See "MEP Architecture" in *Sun Java System Mobile Enterprise Platform 1.0 Architectural Overview* for details on these two types of installation.) To deploy the Enterprise Connector, place it in the `lib` directory of the Application Server domain that contains your MEP installation.

Figure 1–1 shows the location of the Enterprise Connector in a single-tier deployment. The Gateway Engine communicates with the Enterprise Connector using the Java Content Repository (JCR) API. The Enterprise Connector in turn communicates with an EIS/EAI system using a Sun JCA Adapter (JCA stands for Java Connector Architecture).

There are other uses for an Enterprise Connector in addition to communicating with an EIS/EAI system using a Sun JCA Adapter. For example, you can use it in the following ways:

- To store data in a database using the JDBC API
- To store data on the file system
- To access an EIS/EAI system using JAX-WS

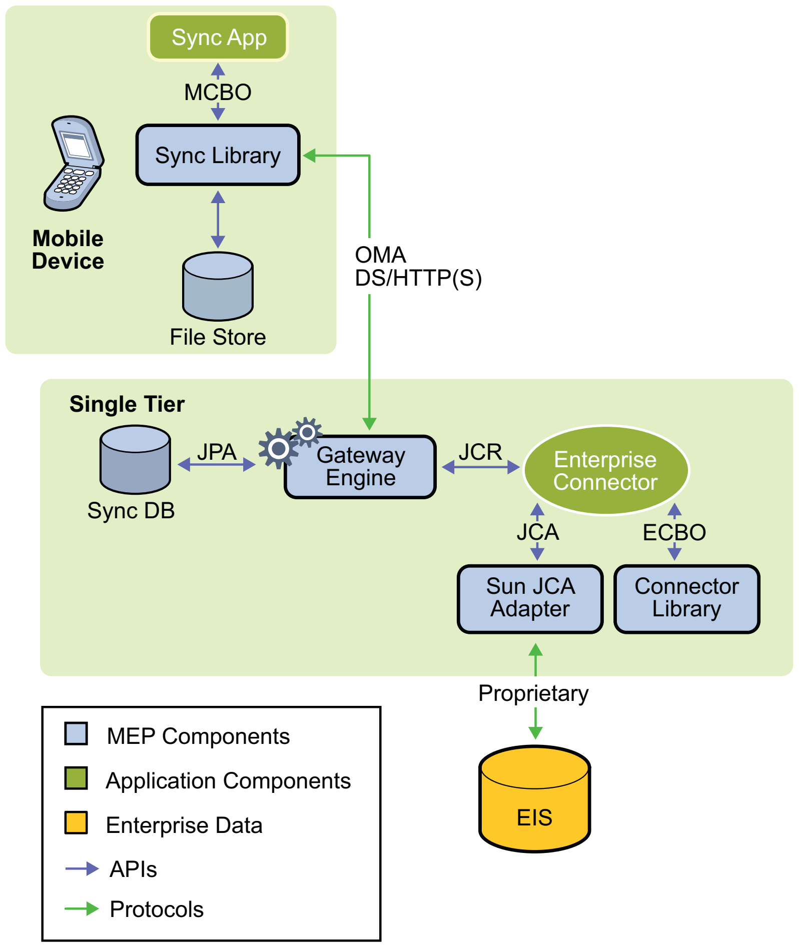

FIGURE 1–1 An Enterprise Connector in a Single-tier MEP Deployment

In a two-tier MEP deployment, the Enterprise Connector is deployed on the second tier, as shown in Figure 1–2. In this situation, communication between the Gateway Engine and the Enterprise Connector goes through a web service. The Gateway Engine communicates with the web service client using the Java Content Repository (JCR) API. The web service client communicates with the web service endpoint using SOAP/HTTPS. Finally, the web service endpoint communicates with the Enterprise Connector using the JCR API.

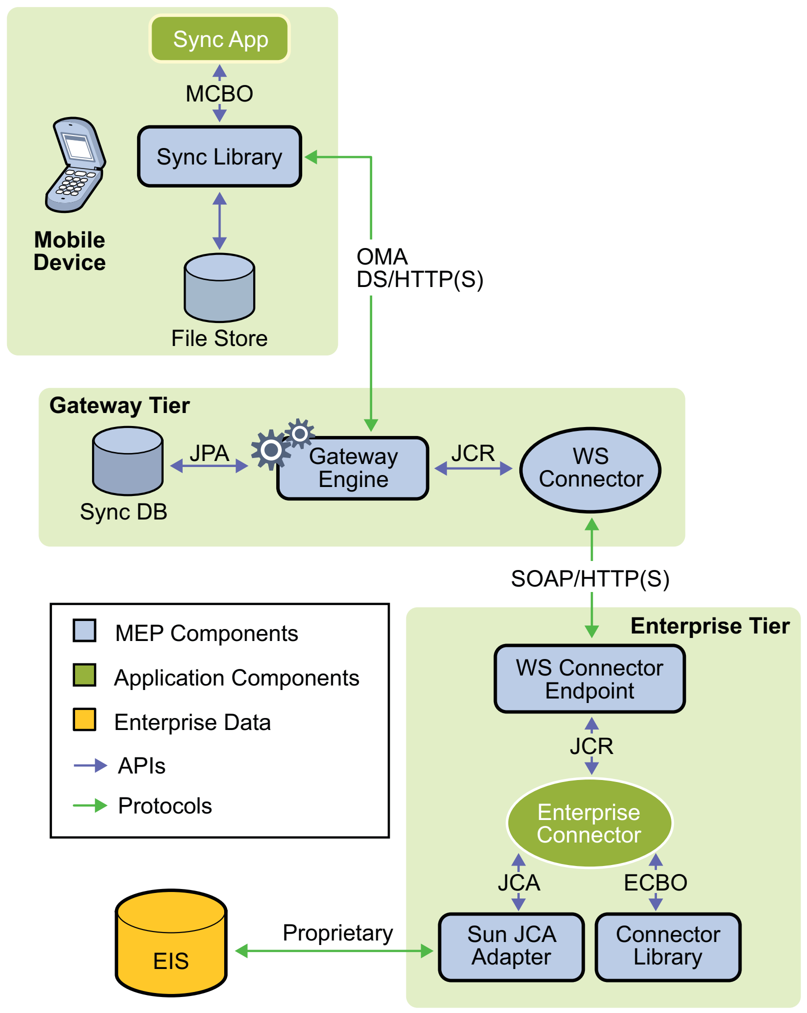

FIGURE 1–2 An Enterprise Connector in a Two-tier MEP Deployment

To configure your deployed Enterprise Connector, use the MEP Admin Console. For details, see "Using the Connectors Tab" in *Sun Java System Mobile Enterprise Platform 1.0 Administration Guide*.

◆ ◆ ◆ CHAPTER 2

Creating an Enterprise Connector

A Mobile Enterprise Platform (MEP) Enterprise Connector typically extends five classes in the Enterprise Connector Business Object (ECBO) API, along with an XML file that defines parameters used by the underlying Java Content Repository (JCR) implementation.

For details on the ECBO API classes and methods, see Chapter 3, "Classes and Methods in the Enterprise Connector Business Object API Package." The API documentation is also included in the MEP client bundle. In the directory where you unzipped the client bundle (see the *Sun Java System Mobile Enterprise Platform 1.0 Installation Guide* for details), it is in the subdirectory `sjsmep-client-1_0_02-fcs/doc/ecbo/api`.

This chapter uses the MusicDB sample application provided with MEP to demonstrate how to use the ECBO API. The Enterprise Connector in this application acts as the intermediary between a client on a mobile device and a database. For this simple demo application, the database is not a full-fledged EIS/EAI system but an ordinary database that is accessed using the Java Database Connectivity (JDBC) API.

The source code for the MusicDB Enterprise Connector is included in the MEP client bundle. In the directory where you unzipped the client bundle, it is in the subdirectory `sjsmep-client-1_0_02-fcs/samples/ecbo/`.

This chapter covers the following topics:

- "Packages in the Enterprise Connector Business Object API" on page 16
- "Extending the `BusinessObject` Class" on page 17
- "Extending the `BusinessObjectProvider` Class" on page 22
- "Extending the `TransactionManager` Class" on page 27
- "Extending the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` Classes" on page 28
- "Creating the Resource File for an Enterprise Connector" on page 29
- "Accessing a Sun JCA Adapter for an EIS/EAI System" on page 30

Packages in the Enterprise Connector Business Object API

The Enterprise Connector Business Object (ECBO) API contains the following classes:

- `com.sun.mep.connector.api.BusinessObject` is the base class for all business objects. Business objects are the entities synchronized with client applications. The fields of the Enterprise Connector business object must match those of the business object for the corresponding client application.
- `com.sun.mep.connector.api.BusinessObjectProvider` is a provider class for instances of `BusinessObject`.
- `com.sun.mep.connector.api.Command` is the base class for all business object commands.
- `com.sun.mep.connector.api.InsertCommand` is the base class for the command that inserts a business object into the database or EIS/EAI system.
- `com.sun.mep.connector.api.UpdateCommand` is the base class for the command that updates a business object in the database or EIS/EAI system.
- `com.sun.mep.connector.api.DeleteCommand` is the base class for the command that deletes a business object from the database or EIS/EAI system.
- `com.sun.mep.connector.api.SessionContext` is a helper class used by `BusinessObjectProvider`. It stores contextual information about the session in which a `BusinessObjectProvider` is instantiated.
- `com.sun.mep.connector.api.TransactionManager` is a helper class used by `BusinessObjectProvider`. It supports methods for starting, stopping, and aborting database transactions.

The MusicDB example implements its own versions of all of these classes except for `SessionContext`. It uses the default implementation of `SessionContext`.

See Chapter 3, "Classes and Methods in the Enterprise Connector Business Object API Package," for summaries of the classes and methods in the ECBO API packages.

To synchronize data with an EIS/EAI system such as Siebel or SAP, your `BusinessObjectProvider` implementation and the three command implementations will need to call methods that access the Sun JCA Adapter for that system. These methods are provided by the Sun Java Composite Application Platform Suite (Java CAPS). See "Accessing a Sun JCA Adapter for an EIS/EAI System" on page 30 for details.

Extending the BusinessObject Class

The BusinessObject class holds the data you need to synchronize. In addition to the required properties name and extension, you specify properties and define getter and setter methods for this data.

For details on this class, see "The BusinessObject Class" on page 43.

The name property is the most important BusinessObject property. This property defines the identity of the business object. The name you specify must be unique within your database or EIS/EAI system.

For the MusicDB example, the class that extends BusinessObject is MusicAlbum. The source file MusicAlbum.java begins by importing Java SE packages, along with com.synchronica logging packages and the required ECBO API classes:

```
package com.sun.mep.connector.jdbc.album;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.sql.Connection;
import java.util.Calendar;

import com.synchronica.logging.Loggers;
import com.synchronica.logging.Logger;
import com.sun.mep.connector.api.BusinessObject;
import com.sun.mep.connector.api.InsertCommand;
import com.sun.mep.connector.api.UpdateCommand;
import com.sun.mep.connector.api.DeleteCommand;
```

The class code itself begins by declaring a string value and setting up a logger:

```
public class MusicAlbum extends BusinessObject<MusicAlbumProvider> {

    private static final String DEFAULT_VALUE = "$$default$$";

    static Logger logger = Loggers.getLogger(MusicAlbum.class);
```

This example then declares its data properties (there are only three in addition to the name property):

```
    /**
     * Album’s artist.
     */
    String artist;

    /**
     * Date when the album was published.
     */
    Calendar datePublished;

    /**
     * Album’s rating from 1 to 5.
     */
    int rating;
```

The file then declares a StringBuilder and defines the one-argument constructor, which takes the MusicAlbumProvider as its argument:

```
    /**
     * String builder used to return SQL commands.
     */
    StringBuilder stringBuilder = new StringBuilder();

    public MusicAlbum(MusicAlbumProvider provider) {
        super(provider);
    }
```

Now the class implements its getter and setter methods for the artist, rating, and datePublished properties:

```
    public String getArtist() {
        return artist;
    }

    public void setArtist(String artist) {
        this.artist = artist;
    }

    public int getRating() {
        return rating;
    }

    public void setRating(int rating) {
        this.rating = rating;
    }

    /**
     * Returns the date published as a string in the format
     * ’YYYYMMDD’.
     */
    public String getDatePublished() {
        stringBuilder.setLength(0);
        stringBuilder.append(datePublished.get(Calendar.YEAR));
        int month = datePublished.get(Calendar.MONTH) + 1;
        if (month < 10) {
            stringBuilder.append('0');
        }
        stringBuilder.append(month);
        int day = datePublished.get(Calendar.DAY_OF_MONTH);
        if (day < 10) {
            stringBuilder.append('0');
        }
        stringBuilder.append(day);
        return stringBuilder.toString();
    }

    /**
     * Set the date published in the format 'YYYYMMDD'.
     */
    public void setDatePublished(String date) {
        datePublished = Calendar.getInstance();
        datePublished.set(Calendar.YEAR,
                Integer.parseInt(date.substring(0, 4)));
        datePublished.set(Calendar.MONTH,
                Integer.parseInt(date.substring(4, 6)) - 1);
        datePublished.set(Calendar.DAY_OF_MONTH,
                Integer.parseInt(date.substring(6, 8)));
    }
```

The class implements the `getExtension` method by specifying `.alb` as the file extension for `MusicAlbum` objects. This extension must match the extension used by the client.

```
    @Override
    public String getExtension() {
        return ".alb";
    }
```

The class does not implement its own versions of the `getName` and `setName` methods; instead, it uses the versions defined in the `BusinessObject` class.

The class uses Java Serialization to implement the `serialize` and `deserialize` methods as follows. Note the calls to the `BusinessObject` versions of `getName` and `setName` in addition to the getter and setter methods defined by `MusicAlbum`. The format used in the `serialize` and `deserialize` methods is part of the contract between the client and the Enterprise Connector.

```
    public byte[] serialize() throws IOException {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        DataOutputStream dOut = new DataOutputStream(out);
        dOut.writeUTF(getName());
        dOut.writeUTF(getArtist() != null ? getArtist() : DEFAULT_VALUE);
        dOut.writeUTF(getDatePublished() != null ? getDatePublished() : DEFAULT_VALUE);
        dOut.writeUTF(Integer.toString(getRating()));
        dOut.flush();
        return out.toByteArray();
    }

    public void deserialize(byte[] array) throws IOException {
        ByteArrayInputStream in = new ByteArrayInputStream(array);
        DataInputStream dIn = new DataInputStream(in);

        setName(dIn.readUTF());
        artist = dIn.readUTF();
        if (artist.equals(DEFAULT_VALUE)) {
            artist = null;
        }
        String date = dIn.readUTF();
        if (date.equals(DEFAULT_VALUE)) {
            datePublished = null;
        }
        else {
            setDatePublished(date);
        }
        rating = Integer.parseInt(dIn.readUTF());
    }
```

The class implements the `getInsertCommand`, `getUpdateCommand`, and `getDeleteCommand` methods using the constructors specific to this business object:

```
    /**
     * {@inheritDoc}
     */
    @Override
    public MusicAlbumInsertCommand getInsertCommand() {
        return new MusicAlbumInsertCommand(this, getSQLConnection(),
                getInsertString());
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public MusicAlbumUpdateCommand getUpdateCommand() {
        return new MusicAlbumUpdateCommand(this, getSQLConnection(),
                getUpdateString());
    }
```

```
/**
 * {@inheritDoc}
 */
@Override
public MusicAlbumDeleteCommand getDeleteCommand() {
    return new MusicAlbumDeleteCommand(this, getSQLConnection(),
            getDeleteString());
}
```

One of the constructor arguments for each command is the value returned by a helper method (`getInsertString`, `getUpdateString`, `getDeleteString`) that generates an SQL statement string. These methods are implemented as follows:

```
/**
 * Returns an SQL insert statement to add this instance
 * to the database.
 */
public String getInsertString() {
    stringBuilder.setLength(0);
    stringBuilder.append("INSERT INTO album VALUES ('")
         .append(getName()).append("','")
         .append(artist).append("', DATE '")
         .append(datePublished.get(Calendar.YEAR)).append("-")
         .append(datePublished.get(Calendar.MONTH) + 1).append("-")
         .append(datePublished.get(Calendar.DAY_OF_MONTH)).append("',")
         .append(Integer.toString(rating)).append(")")
         .append(getBusinessObjectProvider().getUsername()).append("')");
    return stringBuilder.toString();
}

/**
 * Returns an SQL update statement to modify this instance
 * in the database.
 */
public String getUpdateString() {
    stringBuilder.setLength(0);
    stringBuilder.append("UPDATE album SET artist='")
         .append(artist).append("', date_published=DATE '")
         .append(datePublished.get(Calendar.YEAR)).append("-")
         .append(datePublished.get(Calendar.MONTH) + 1).append("-")
         .append(datePublished.get(Calendar.DAY_OF_MONTH)).append("', rating=")
         .append(Integer.toString(rating))
         .append(" WHERE name = '").append(getName())
         .append("' AND username = '" + getBusinessObjectProvider().getUsername() +
             "'");
    return stringBuilder.toString();
}
```

```
    /**
     * Returns an SQL delete statement to remove this instance
     * from the database.
     */
    public String getDeleteString() {
        stringBuilder.setLength(0);
        stringBuilder.append("DELETE FROM album WHERE name = '")
                     .append(getName())
                     .append("' AND username = '"
                              + getBusinessObjectProvider().getUsername() + "'");
        return stringBuilder.toString();
    }
```

You may notice that the SQL statements show an additional column, `username`, in the database's `album` table in addition to columns for the `MusicAlbum` class's properties (`name`, `artist`, `datePublished`, and `rating`). The `username` column and the `name` column provide a composite primary key for the `album` table. The `username` column identifies the owner of an album and allows the `MusicAlbumProvider.getBusinessObjects` method to return only the albums for a particular user, so that multiple users can share the `album` table.

An additional method, `getSelectString`, is provided for testing purposes.

Another constructor argument for the commands is the value returned by another helper method, `getSQLConnection`, which returns a JDBC `Connection` object created by the `MusicAlbumProvider` class.

```
    /**
     * Returns a connection object that can be used to
     * execute SQL commands.
     */
    public Connection getSQLConnection() {
        return getBusinessObjectProvider().getSQLConnection();
    }
```

Extending the `BusinessObjectProvider` Class

The `BusinessObjectProvider` class serves several purposes:

- It allows you to retrieve all the business objects from a database by calling the `getBusinessObjects()` method.
- It allows you to create new business objects by calling the `newBusinessObject()` method.
- It provides access to a transaction manager and a session context.

For details on this class, see "The `BusinessObjectProvider` Class" on page 45.

For the MusicDB example, the class that extends `BusinessObjectProvider` is `MusicAlbumProvider`. Like the file for the `MusicAlbum` class, the `MusicAlbumProvider.java` source file begins by importing Java SE packages, along with com.synchronica logging packages and the required ECBO API classes. It then begins by setting up a logger and declaring some string constants, a JDBC connection object, its implementation of the TransactionManager class, and a user name object:

```
public class MusicAlbumProvider extends BusinessObjectProvider<MusicAlbum> {

    static Logger logger = Loggers.getLogger(MusicAlbumProvider.class);

    public static final String REPOSITORY_NAME = "MusicDbRepository";
    public static final String MUSICDB_JNDI_DATASOURCE = "jdbc/musicdb";
    public static final String DB_USER_NAME = "musicdbuser";
    public static final String DB_USER_PASS = "musicdbpass";
    Connection sqlConnection = null;

    MusicAlbumTransactionManager transactionManager;

    String username;
```

The REPOSITORY_NAME value is identical to the repository name specified in the resource file for the Enterprise Connector. In the MusicDB sample, the resource file is named MusicDbRepository.xml and defines a repository named MusicDbRepository.

The code implements two forms of the business object constructor: the no-argument constructor specified by the API and a one-argument form that takes a user name as argument for testing purposes.

Next, the code implements the two lifecycle methods for the BusinessObjectProvider class, initialize and terminate, which coincide with the start and end of a synchronization session.

The initialize method allocates resources required for a synchronization session or for database authentication. In this case, the code does the following:

- Looks up a JDBC datasource.
- Retrieves the username value from the SessionContext for use in updating the album table in the database.
- Obtains a JDBC connection using the credentials established for the MusicDB database when it was created.
- Instantiates a transaction manager.

```
    /**
     * Creates a connection to the {@link #MUSICDB_JNDI_DATASOURCE}
     * database.
     */
    @Override
    public void initialize() {
        logger.debug("Initializing provider " + this);

        try {
            Context jndiContext = new InitialContext();
            DataSource ds = null;

            // If unable to get JNDI datasource, use local one for testing
            try {
                ds = (DataSource) jndiContext.lookup(MUSICDB_JNDI_DATASOURCE);
            }
            catch (NoInitialContextException e) {
                ds = new MusicDbDataSource();       // testing only!
            }

            // Get database credentials from provider's context
            SessionContext sessionContext = getSessionContext();
            username = sessionContext.getUsername();

            // Get connection using default credentials
            sqlConnection = ds.getConnection(DB_USER_NAME, DB_USER_PASS);

            // Init transaction manager
            transactionManager = new MusicAlbumTransactionManager();
        }
        catch (Exception ex) {
            throw new RuntimeException(ex);
        }
    }
```

The implementation of the `terminate` method releases any resources allocated by the `initialize` method. In this case, it closes the JDBC connection.

```
    /**
     * Closes a connection to the {@link #MUSICDB_JNDI_DATASOURCE}
     * database.
     */
    @Override
    public void terminate() {
        logger.debug("Terminating provider " + this);

        try {
            if (sqlConnection != null) {
                sqlConnection.close();
            }
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

The implementation of the getRepositoryName method specifies the string value declared at the beginning of the class. The repository in question is the JCR repository that is used for communication between the Gateway Engine and the Enterprise Connector and that is specified by the resource file.

```
/**
 * {@inheritDoc}
 */
@Override
public String getRepositoryName() {
    return REPOSITORY_NAME;
}
```

The implementation of the getBusinessObjects method uses a JDBC query to retrieve all the albums for the user username from the database, instantiates a MusicAlbum object for each retrieved album, and adds it to an ArrayList of albums.

```
/**
 * {@inheritDoc}
 */
@Override
public List<MusicAlbum> getBusinessObjects() {
    logger.debug("Getting objects from provider " + this);

    Statement stmt = null;
    List<MusicAlbum> albums = null;

    try {
        stmt = sqlConnection.createStatement();

        // Read all music albums and store them in array
        albums = new ArrayList<MusicAlbum>();
        ResultSet rs = stmt.executeQuery(
                "SELECT * FROM album WHERE username = '" + username + "'");
        while (rs.next()) {
            MusicAlbum album = new MusicAlbum(this);
            album.setName(rs.getString(1));
            album.setArtist(rs.getString(2));
            album.setDatePublished(rs.getString(3).replace("-", ""));
            album.setRating(rs.getInt(4));
            albums.add(album);
        }
        rs.close();
    } catch (Exception ex) {
        throw new RuntimeException(ex);
    } finally {
        if (stmt != null) {
            try { stmt.close(); } catch (Exception e) { /* ignore !*/ }
            }
        }
        return albums;
    }
```

The implementation of the newBusinessObject method is much simpler: it calls the one-argument constructor for MusicAlbum.

```
    /**
     * {@inheritDoc}
     */
    @Override
    public MusicAlbum newBusinessObject() {
        return new MusicAlbum(this);
    }
```

The provider class also implements the helper method getSQLConnection, which returns the JDBC connection that was instantiated by the initialize method. This method is called by the MusicAlbum class.

```
    /**
     * Returns a connection object that can be used to
     * execute SQL commands.
     */
    public Connection getSQLConnection() {
        return sqlConnection;
    }
```

The provider class also implements a getUsername method that is called by the MusicAlbum class's utility methods:

```
    /**
     * Returns the user name logged into the session that
     * created this provider.
     */
    public String getUsername() {
        return username;
    }
```

The getTransactionManager method retrieves the MusicAlbumTransactionManager that is declared at the beginning of the file, instantiated in the initialize method, and implemented within the MusicAlbumProvider.java file.

```
    /**
     * Returns a transaction manager that uses JDBC to start,
     * stop and abort transactions.
     */
    @Override
    public MusicAlbumTransactionManager getTransactionManager() {
        return transactionManager;
    }
```

Extending the `TransactionManager` Class

The `MusicAlbumProvider.java` file includes the implementation of the `TransactionManager` class. The `TransactionManager` class can be implemented in a separate file, but the relationships between the two classes mean that it is simpler to keep them together. It is also possible to use the default implementation of the `TransactionManager` class instead of implementing it yourself.

For details on this class, see "The `TransactionManager` Class" on page 48.

The MusicDB implementation of this class is called `MusicAlbumTransactionManager`. This class manages the database transactions for `MusicAlbum` objects using the JDBC API. It turns the database's auto-commit feature off if it has one and starts, stops, and aborts database transactions. The class definition begins with the constructor, which takes no arguments.

```
public class MusicAlbumTransactionManager extends
        TransactionManager<MusicAlbumProvider> {

    public MusicAlbumTransactionManager() {
        super(MusicAlbumProvider.this);

        assert (sqlConnection != null);
        try {
            sqlConnection.setAutoCommit(false);
        }
        catch (SQLException e) {
            // Ignore if not supported by DB
        }
    }
```

The `abortTransaction` method calls the JDBC `Connection.rollback` method:

```
    @Override
    public void abortTransaction() {
        logger.debug("Aborting transaction on SQL connection "
                + sqlConnection);
        try {
            sqlConnection.rollback();
        }
        catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
```

The beginTransaction and endTransaction methods are closely linked. The endTransaction method commits the current transaction, an action that automatically starts the next transaction. The beginTransaction method simply calls endTransaction.

```
@Override
public void beginTransaction() {
    endTransaction();   // starts a new one
}

@Override
public void endTransaction() {
    logger.debug("Starting/Committing transaction on SQL connection "
            + sqlConnection);
    try {
        sqlConnection.commit();
    }
    catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

Extending the InsertCommand, UpdateCommand, and DeleteCommand Classes

The InsertCommand, UpdateCommand, and DeleteCommand classes all extend the Command class.

For details on these classes, see "The InsertCommand Class" on page 47, "The UpdateCommand Class" on page 48, "The DeleteCommand Class" on page 46, and "The Command Class" on page 46.

For the MusicDB example, the implementations of the three classes are almost identical. The source files for the implementations are MusicAlbumInsertCommand.java, MusicAlbumUpdateCommand.java, and MusicAlbumDeleteCommand.java.

Each source file begins by importing Java SE packages, along with com.synchronica logging packages and the required ECBO API classes. It then begins by setting up a logger and declaring two objects that are used by the constructor method and the execute method. For example, MusicAlbumInsertCommand.java begins as follows:

```
public class MusicAlbumInsertCommand extends InsertCommand<MusicAlbum> {

    static Logger logger = Loggers.getLogger(MusicAlbumInsertCommand.class);

    private String sqlStatement;

    private Connection sqlConnection;
```

The code then extends the class constructor. While the constructor for the base class takes one argument, the constructor for each of the implementation classes takes three arguments: the `MusicAlbum`, a string that represents a SQL statement, and a JDBC connection. For example, the constructor for `MusicAlbumUpdateCommand` looks like this:

```
public MusicAlbumUpdateCommand(MusicAlbum album,
        Connection sqlConnection, String sqlStatement)
{
    super(album);
    this.sqlConnection = sqlConnection;
    this.sqlStatement = sqlStatement;
    logger.debug("Creating instance " + this + ": " + sqlStatement);
}
```

Finally, the code for each command implements the `execute` method in exactly the same way. First, it uses the instantiated connection to create a JDBC `Statement` object using the instantiated string. Then it executes the statement.

```
public void execute() {
    int result = 0;
    Statement stmt = null;

    try {
        logger.debug("Executing instance " + this + ": " + sqlStatement);
        stmt = sqlConnection.createStatement();
        result = stmt.executeUpdate(sqlStatement);
    }
    catch (Exception ex) {
        throw new RuntimeException(ex);
    }
    finally {
        if (stmt != null) {
            try { stmt.close(); } catch (Exception e) { /* ignore !*/ }
        }
    }
}
```

Creating the Resource File for an Enterprise Connector

The resource file that you need to package with an Enterprise Connector (as described in "About the Enterprise Connector Business Object (ECBO) API" on page 11) is an XML file for Jeceira, the implementation of JCR used by MEP. The name of the file typically refers to the database or EIS/EAI system. For example, the resource file for MusicDB is named `MusicDbRepository.xml`. The file begins and ends as follows:

```
<jeceira xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
    xmlns:jcr="http://www.jcp.org/jcr/1.0"
    xmlns:sync="http://www.synchronica.com/jcr/types"
    xmlns:aprzv="http://www.aparzev.com/jrc/aprzv"
    xmlns:udc="http://www.synchronica.com/udc/types/1.0">
    <repositories>
        <repository name="MusicDbRepository">

            <workspaces>
                <workspace name="MusicDbWorkspace" />
            </workspaces>
            ....
        </repository>
    </repositories>
</jeceira>
```

All resource files are identical except for two values:

- The name attribute of the `repository` element, in this case `MusicDbRepository`
- The name attribute of the `workspace` element, in this case `MusicDbWorkspace`

To create your own resource file, you can copy the resource file from the sample Enterprise Connector source directory, rename it, and modify these two values. In the unzipped client bundle, you can find the file in `sjsmep-client-1_0_02-fcs/samples/ecbo/src/MusicDbRepository.xml`.

You use these values when you configure the Enterprise Connector in the MEP Admin Console. See "Using the Connectors Tab" in *Sun Java System Mobile Enterprise Platform 1.0 Administration Guide* for details.

Accessing a Sun JCA Adapter for an EIS/EAI System

If you are designing your Enterprise Connector to access an EIS/EAI system instead of a database, the connector must access the Sun JCA Adapter for that system instead of making JDBC calls.

You must first create an Object Type Definition (OTD) that maps your business object properties to data on the EIS/EAI system. To create the OTD, you need to use the NetBeans IDE with plugins that are provided with MEP.

After you create the OTD, use the NetBeans code completion feature to call methods on the classes generated by the OTD wizard.

This section covers the following topics:

- "Creating an Object Type Definition (OTD)" on page 31
- "Writing Code to Access a Sun JCA Adapter" on page 32

Creating an Object Type Definition (OTD)

For information on working with Sun JCA Adapters, see the Designing section of the Java CAPS documentation (`http://developers.sun.com/docs/javacaps/designing/`). Specific sections you will need to look at include the following:

- Technical Overview for Sun JCA Adapters (`http://developers.sun.com/docs/javacaps/designing/jcapssunjcaad.cnfg_tech-ovrvw_t.html`)
- Object Type Definition Wizards (`http://developers.sun.com/docs/javacaps/designing/jcapssunjcaad.ggrbu.html`)

  The two middle sections on inbound and outbound JCA Resource Adapter client code are not relevant to Enterprise Connectors.

To obtain the NetBeans plugins needed to create an OTD, follow the instructions in Installation of Netbeans Modules (`http://developers.sun.com/docs/javacaps/designing/jcapssunjcaad.inst_jca-adapter_t.html`). The `NetBeansModules` referred to in these instructions are part of your MEP installation. In the location where you unzipped the installation bundle `sjsmep-1_0-fcs-`*operating-system*`.zip`, you will find them in the directory `sjsmep-1_0-fcs/NetBeansModules`. (The Java CAPS documentation states that they are in the directory `AdapterPack/NetBeansModules/CommonLib`, but for MEP they are in the directory `NetBeansModules/commonlib`.)

To develop an OTD for your application, follow the instructions appropriate to your EIS/EAI system in Developing OTDs for Application Adapters (`http://developers.sun.com/docs/javacaps/designing/dotdappadptr.dotdappadptr_intro.html`).

The instructions for adapters supported by MEP are in the following sections:

- Creating SAP BAPI OTDs (`http://developers.sun.com/docs/javacaps/designing/dotdappadptr.dsgn_sap-bapi-otd_t.html`)
- Creating Siebel EAI OTDs (`http://developers.sun.com/docs/javacaps/designing/dotdappadptr.dsgn_siebel-eai-otd_t.html`)
- Using the Oracle Applications Wizard and JCA Adapter Tooling with an EJB Project (`http://developers.sun.com/docs/javacaps/designing/jcapssunjcaad.cnfg_oracleapps-wiz_t.html`)

An Enterprise Connector is a Sun JCA Adapter client application that is not an Enterprise JavaBeans (EJB) component. You may find that you need to create the OTD and develop the Enterprise Connector inside an EJB project. However, you should then remove the Enterprise Connector and OTD from the EJB JAR file and place them in an ordinary JAR file before you place the JAR file in the `domains/mep/lib` directory for the Application Server. The OTD is generated in a separate JAR file, so it is easy to copy it to another project.

Writing Code to Access a Sun JCA Adapter

To access a Sun JCA Adapter, your code needs to use Java CAPS APIs, which are documented at `http://developers.sun.com/docs/javacaps/reference/javadocs/index.jsp`.

This section describes how to extend the ECBO classes to access a Sun JCA Adapter.

- "Extending the `BusinessObjectProvider` Class to Access a Sun JCA Adapter" on page 32
- "Extending the `BusinessObject` Class to Access a Sun JCA Adapter" on page 37
- "Extending the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` Classes to Access a Sun JCA Adapter" on page 40

The default implementation of the `TransactionManager` class may be sufficient for your application.

Extending the `BusinessObjectProvider` Class to Access a Sun JCA Adapter

To allow your Enterprise Connector to work with a Sun JCA Adapter, your `BusinessObjectProvider` implementation needs to create a connection to the Adapter in its `initialize` method, close that connection in its `terminate` method, and retrieve objects through the Adapter in its `getBusinessObject` method.

For a SAP BAPI application, for example, you import the following packages:

```
import com.stc.connector.appconn.common.ApplicationConnectionFactory;
import com.stc.connector.appconn.common.ApplicationConnection;
import com.stc.connector.sapbapiadapter.appconn.SAPApplicationConnection;
import com.stc.util.OtdObjectFactory;
```

When you create a provider for a `Customer` business object, you declare objects like the following. The `customer.Customer` class is generated by the OTD wizard.

```
public class CustomerProvider extends BusinessObjectProvider<Customer> {
    ...
    public static final String SAP_JNDI_DATASOURCE = "jcaps/sap";
    public static final String REPOSITORY_NAME = "SAPRepository";

    private ApplicationConnectionFactory mJCAsapadapter = null;
    private ApplicationConnection mJCAsapadapterConnection = null;
    private customer.Customer mJCAsapcustomerCommObj = null;
```

The provider's `initialize` method then allocates these resources. It obtains an `ApplicationConnectionFactory` object by means of a JNDI lookup, then uses the factory to create the `ApplicationConnection` object. These method calls are the same no matter which EIS/EAI system you are using:

```
    @Override
    public void initialize() {
        logger.debug("Initializing provider " + this);

        try {
            InitialContext ic = new InitialContext();
            // First get ApplicationConnectionFactory through JNDI lookup
            mJCAsapadapter =
                (ApplicationConnectionFactory) ic.lookup(SAP_JNDI_DATASOURCE);

            /* Then create ApplicationConnection. One AppConn can be dynamically
             * allocated to a physical connection defined in connection pool;
             * this results in connection reuse according to JCA and Appserver
             * contract
             */
            mJCAsapadapterConnection = mJCAsapadapter.getConnection();
```

The `initialize` method then uses the `OtdObjectFactory` to create an instance of a SAP customer communication object. Methods called on this object are specific to the SAP OTD. The code casts the generic `ApplicationConnection` object `mJCAsapadapterConnection` to another application connection specific to SAP:

```
            /* Create Customer communication object
             */
            mJCAsapcustomerCommObj =
                (customer.Customer) OtdObjectFactory.createInstance(null,
                    "customer.Customer");

            /* Set ApplicationConnection on Customer communication object
             */
            mJCAsapcustomerCommObj.setAppConn(
                (SAPApplicationConnection) mJCAsapadapterConnection);
```

The `initialize` method next uses the ECBO API `SessionContext` object to retrieve the user name and password. It then uses these values to create user credentials specific to SAP, and finally connects to the Sun JCA Adapter for SAP.

```
            // Get backend credentials from provider’s context
            SessionContext sessionContext = getSessionContext();
            String param = sessionContext.getUsername();
            if (param != null) {
                mJCAsapcustomerCommObj.getSAPConnectionParams().setUserid(param);
            }
            param = sessionContext.getPassword();
            if (param != null) {
                mJCAsapcustomerCommObj.getSAPConnectionParams().setPassword(param);
            }
            mJCAsapcustomerCommObj.connectWithNewParams();
            ic.close();
        }
        catch (Exception ex) {
            logger.debug("Initializing provider exception" + ex.getMessage());
            throw new RuntimeException(ex);
        }
    }
```

The `terminate` method closes the connection created by the `initialize` method:

```
    @Override
    public void terminate() {
        logger.debug("Terminating provider " + this);

        try {
            if (mJCAsapadapterConnection != null) {
                mJCAsapadapterConnection.close();
                logger.info("terminate provider close connection"
                        + mJCAsapadapterConnection.toString());
            }
        }
        catch (Exception e) {
            logger.debug("terminating provider exception" + e.getMessage());
            throw new RuntimeException(e);
        }
    }
```

The provider code also implements a utility method, `getSAPCustomerClient`, which retrieves the customer communication object:

```
    /**
     * @return SAPCustomerClient object that can be used to
     * operate on a Customer BAPI.
     */
    public customer.Customer getSAPCustomerClient() {
        return mJCAsapcustomerCommObj;
    }
```

The `getBusinessObjects` method uses the `getSAPCustomerClient` method to retrieve the SAP customer data and store it in the Enterprise Connector's `Customer` object. It again calls methods on the communication object generated by the OTD wizard.

```
    @Override
    //Retrieve all IDocs and map here between Customer and VendorAccount Object
    public List<Customer> getBusinessObjects() {
        logger.debug("Getting objects from provider " + this);
```

```
HashMap<String, Customer> customerMap = new HashMap<String, Customer>();

try {
    // Getting customer list
    getSAPCustomerClient().getGetList().getIDRANGE(0).setOPTION("CP");
    getSAPCustomerClient().getGetList().getIDRANGE(0).setLOW("*");
    logger.info("Executing Customer with the following values Option " +
        "[" + getSAPCustomerClient().getGetList().getIDRANGE(0).getLOW()
        + "] Option ["
        + getSAPCustomerClient().getGetList().getIDRANGE(0).getOPTION() + "]");
    getSAPCustomerClient().getGetList().execute();

    // Process returned data and populate customer list
    customer.Customer.GetList.ExportParams.RETURN ret =
        getSAPCustomerClient().getGetList().getExportParams().getRETURN();
    logger.info("Retrieved ["
        + getSAPCustomerClient().getGetList().countADDRESSDATA()
        + "] customers");

    customer.Customer.GetList.ADDRESSDATA[] addressList =
        getSAPCustomerClient().getGetList().getADDRESSDATA();
    for (int i = 0; i < addressList.length; i++) {
        customer.Customer.GetList.ADDRESSDATA addr = addressList[i];

        // Ignore companies whose names start with "DELETED" -- hack
        if (!addr.getNAME().startsWith("DELETED")) {
            // Ignore customers whose names are repeated
            if (customerMap.containsKey(addr.getNAME())) {
                continue;
            }

            // Create a new Customer instance
            Customer comp = new Customer(this);

            // Set unique name for business object
            comp.setName(addr.getNAME());

            // Set customer number and name
            comp.setCustomerNumber(addr.getCUSTOMER());
            comp.setCustomerName(addr.getNAME());

            // Get sales area
            getSAPCustomerClient().getGetSalesAreas().getImportParams()
                .setCUSTOMERNO(comp.getCustomerNumber());
            getSAPCustomerClient().getGetSalesAreas().execute();
            String retNo = getSAPCustomerClient().getGetSalesAreas()
                .getExportParams().getRETURN().getMESSAGE();
            String retMsg = getSAPCustomerClient().getGetSalesAreas()
    .getExportParams().getRETURN().getCODE();
logger.info("Return Number [" + retNo + "] retMsg ["
    + retMsg + "].");
if (retNo.length() > 0) {
    throw new RuntimeException(retMsg);
}

// Set sales related fields
comp.setSalesOrg(getSAPCustomerClient().getGetSalesAreas()
    .getSALESAREAS(0).getSALESORG());
comp.setDistChannel(getSAPCustomerClient().getGetSalesAreas()
    .getSALESAREAS(0).getDISTRCHN());
comp.setDivision(getSAPCustomerClient().getGetSalesAreas()
    .getSALESAREAS(0).getDIVISION());

// Get detail on customer
getSAPCustomerClient().getGetDetail1().getImportParams()
    .setCUSTOMERNO(comp.getCustomerNumber());
getSAPCustomerClient().getGetDetail1().getImportParams()
    .setPI_SALESORG(comp.getSalesOrg());
getSAPCustomerClient().getGetDetail1().getImportParams()
    .setPI_DISTR_CHAN(comp.getDistChannel());
getSAPCustomerClient().getGetDetail1().getImportParams()
    .setPI_DIVISION(comp.getDivision());
getSAPCustomerClient().getGetDetail1().execute();
retNo =
    getSAPCustomerClient().getGetDetail1().getExportParams()
        .getRETURN().getMESSAGE();
retMsg =
    getSAPCustomerClient().getGetDetail1().getExportParams()
        .getRETURN().getNUMBER();
logger.info("Return Number [" + retNo + "] retMsg ["
    + retMsg + "].");
if (retNo.length() > 0) {
    throw new RuntimeException(retMsg);
}

// Populate customer object data
customer.Customer.GetDetail1.ExportParams.PE_COMPANYDATA
    currAddr = getSAPCustomerClient().getGetDetail1()
        .getExportParams().getPE_COMPANYDATA();
comp.setCity(currAddr.getCITY());
comp.setPostalCode(currAddr.getPOSTL_COD1());
comp.setStreet(currAddr.getSTREET());
comp.setCountryKey(currAddr.getCOUNTRY());
comp.setLanguageKey(currAddr.getLANGU_ISO());
comp.setRegion(currAddr.getREGION());
comp.setTelephone(currAddr.getTEL1_NUMBR());
```

```
                comp.setFaxNumber(currAddr.getFAX_NUMBER());
                comp.setCurrencyKey(currAddr.getCURRENCY());

                customerMap.put(comp.getCustomerName(), comp);
            }
        }
        return new ArrayList<Customer>(customerMap.values());
    }
    catch (Exception ex) {
        throw new RuntimeException(ex);
    }
}
```

The `getRepositoryName` and `newBusinessObject` methods have implementations very similar to those in the `MusicAlbumProvider` class:

```
@Override
public String getRepositoryName() {
    return REPOSITORY_NAME;
}

@Override
public Customer newBusinessObject() {
    return new Customer(this);
}
```

The other methods in the provider class use the default `BusinessObjectProvider` implementation: `getSessionContext`, `setSessionContect`, and `getTransactionManager`.

Extending the `BusinessObject` Class to Access a Sun JCA Adapter

The `BusinessObject` class for an Enterprise Connector that accesses a Sun JCA Adapter may have straightforward implementations of the `BusinessObject` methods, but it may also require some additional utility methods. A SAP BAPI `Customer` object, for example, implements a large number of getter and setter methods for its properties. Its `serialize` and `deserialize` methods can be relatively simple.

The `Customer` object implementations of the `getInsertCommand`, `getUpdateCommand`, and `getDeleteCommand` methods call the constructors for the command classes, as expected. However, here the constructors take two arguments, and the second argument is the value returned by a utility method.

```
/**
 * {@inheritDoc}
 */
@Override
public CustomerInsertCommand getInsertCommand() {
        return new CustomerInsertCommand(this, getInsertCustomer());
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public CustomerUpdateCommand getUpdateCommand() {
        return new CustomerUpdateCommand(this, getUpdateCustomer());
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public CustomerDeleteCommand getDeleteCommand() {
        return new CustomerDeleteCommand(this, getDeleteCustomer());
    }
```

The utility methods use the provider class's `getSAPCustomerClient` method to retrieve first the customer communication object, and then the `CreateFromData1` object. For example, the `getInsertCustomer` method begins as follows:

```
    /**
     * Returns a Customer CreateFromData1 object to be used for insert.
     */
    public customer.Customer.CreateFromData1 getInsertCustomer() {
        customer.Customer.CreateFromData1 cfd = getBusinessObjectProvider()
                .getSAPCustomerClient().getCreateFromData1();
```

The rest of the `getInsertCustomer` method uses the `CreateFromData1` object to assign the `Customer` properties to the SAP BAPI customer object. Finally, it returns the `CreateFromData1` object.

```
       // Set import parameters
        cfd.getImportParams().getPI_COPYREFERENCE().setREF_CUSTMR(
            CustomerProvider.REF_CUSTOMER);
        cfd.getImportParams().getPI_COPYREFERENCE().setSALESORG(getSalesOrg());
        cfd.getImportParams().getPI_COPYREFERENCE().setDISTR_CHAN(getDistChannel());
        cfd.getImportParams().getPI_COPYREFERENCE().setDIVISION(getDivision());

        // Required import parameters
        cfd.getImportParams().getPI_COMPANYDATA().setNAME(getCustomerName());
        cfd.getImportParams().getPI_COMPANYDATA().setLANGU_ISO(getLanguageKey());
        cfd.getImportParams().getPI_COMPANYDATA().setCURRENCY(getCurrencyKey());
        cfd.getImportParams().getPI_COMPANYDATA().setCOUNTRY(getCountryKey());
        cfd.getImportParams().getPI_COMPANYDATA().setPOSTL_COD1(getPostalCode());
        cfd.getImportParams().getPI_COMPANYDATA().setCITY(getCity());
        // Additional import parameters
        cfd.getImportParams().getPI_COMPANYDATA().setSTREET(getStreet());
        cfd.getImportParams().getPI_COMPANYDATA().setREGION(getRegion());
        cfd.getImportParams().getPI_COMPANYDATA().setTEL1_NUMBR(getTelephone());
        cfd.getImportParams().getPI_COMPANYDATA().setFAX_NUMBER(getFaxNumber());

        return cfd;
    }
```

The `getUpdateCustomer` method is almost identical to the `getInsertCustomer` method except that it also marks the fields as having changed:

```
        // Mark fields to be changed
        String ex = "X";
        cfd.getImportParams().getPI_COMPANYDATAX().setNAME(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setLANGU_ISO(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setCURRENCY(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setCOUNTRY(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setPOSTL_COD1(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setCITY(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setSTREET(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setREGION(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setTEL1_NUMBR(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setFAX_NUMBER(ex);
```

Similarly, the `getDeleteCustomer` method informs SAP to delete a record by changing its name to begin with the string `DELETED`:

```
        // Mark customer as deleted by prepending "DELETE" to the name
        setCustomerName("DELETED " + getCustomerName());
        logger.fine("Changing NAME field to [" + getCustomerName() + "].");
        cfd.getImportParams().getPI_COMPANYDATA().setNAME(getCustomerName());
        cfd.getImportParams().getPI_COMPANYDATAX().setNAME("X");
        cfd.getImportParams().getPI_COMPANYDATA().setLANGU_ISO(getLanguageKey());
        cfd.getImportParams().getPI_COMPANYDATA().setCURRENCY(getCurrencyKey());
        cfd.getImportParams().getPI_COMPANYDATA().setCOUNTRY(getCountryKey());
        cfd.getImportParams().getPI_COMPANYDATA().setPOSTL_COD1(getPostalCode());
        cfd.getImportParams().getPI_COMPANYDATA().setCITY(getCity());
        cfd.getImportParams().setCUSTOMERNO(getCustomerNumber());
        cfd.getImportParams().setPI_SALESORG(getSalesOrg());
        cfd.getImportParams().setPI_DISTR_CHAN(getDistChannel());
        cfd.getImportParams().setPI_DIVISION(getDivision());
```

Extending the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` Classes to Access a Sun JCA Adapter

For the three command classes, you need to use generated classes and methods from the OTD. For a SAP BAPI application, for example, you need to import the following packages for the `CustomerInsertCommand` implementation:

```
import customer.Customer.CreateFromData1;
import customer.Customer.CreateFromData1.ExportParams.RETURN;
```

You then use the first of the imported classes in the class constructor, which takes two arguments instead of the single argument of the default implementation:

```
    public CustomerInsertCommand(Customer bobject, CreateFromData1 cfd) {
        super(bobject);
        mCreateFromData = cfd;
        logger.debug("Creating instance " + this);
    }
```

The `CreateFromData1` object passed to the constructor is the returned value from the `Customer` class's `getInsertCustomer` method.

You implement the `execute` command by calling the class's own `execute` method and retrieving any return value through the second imported class:

```
    @Override
    public void execute() {
        try {
            mCreateFromData.execute();
            RETURN ret = mCreateFromData.getExportParams().getRETURN();
            String retMsg = ret.getMESSAGE();
            String message = "SAP (" + ret.getNUMBER() + "): " + retMsg;
            logger.info(message);
            if (retMsg.length() > 0) {
                throw new RuntimeException(message);
            }
        }
        catch (RuntimeException e) {
            throw e;
        }
        catch (Exception e) {
            logger.severe(e.getMessage());
            throw new RuntimeException(e.getMessage(), e);
        }
    }
```

For both the `CustomerDeleteCommand` and the `CustomerUpdateCommand` implementations, you import the following:

```
import customer.Customer.ChangeFromData1;
import customer.Customer.ChangeFromData1.ExportParams.RETURN;
```

The constructors and the `execute` methods for these classes use the `ChangeFromData1` object returned from the `Customer.getUpdateCustomer` and `Customer.getDeleteCustomer` methods. Otherwise the class implementations are identical to those of the `CustomerInsertCommand` implementation.

CHAPTER 3

Classes and Methods in the Enterprise Connector Business Object API Package

The Enterprise Connector Business Object (MCBO) API contains one package, `com.sun.mep.connector.api`, that developers must use. This chapter summarizes the classes contained within this package:

The API documentation is included in the MEP client bundle. In the directory where you unzipped the client bundle (see the *Sun Java System Mobile Enterprise Platform 1.0 Installation Guide* for details), it is in the directory `sjsmep-client-1_0_02-fcs/doc/ecbo/api`.

The `BusinessObject` Class

Table 3–1 lists the constructor and methods belonging to the `BusinessObject` class, the base class for all business objects. Business objects are the entities synchronized with client applications. Each business object instance is identified by a name, which is also used to name the file that holds the object on the mobile client. Business objects are created by business object providers; see "The `BusinessObjectProvider` Class" on page 45 for details.

TABLE 3–1 Class com.sun.mep.connector.api.BusinessObject

Method	Description
`BusinessObject(T bobjectProvider)`	Constructor that takes the `BusinessObjectProvider` for the business object as its argument.
`public abstract void deserialize(byte[] data)`	Deserializes a business object from a byte array.
`public T getBusinessObjectProvider()`	Returns a reference to the business object provider associated with this object.
`public abstract <T extends BusinessObject> DeleteCommand<T> getDeleteCommand()`	Returns a command that when executed can delete this business object from a back end.
`public abstract java.lang.String getExtension()`	Returns the default extension for business objects of this type. Extensions are used by the files holding these objects and must be part of the contract with clients. That is, clients and connectors must use the same extension for the same type of business object. Concrete subclasses should redefine this method.
`public abstract <T extends BusinessObject> InsertCommand<T> getInsertCommand()`	Returns a command that when executed can insert this business object into a back end.
`public long getLastModified()`	Returns a timestamp indicating the last time this business object was modified. The timestamp represents the number of milliseconds since "the epoch," namely January 1, 1970, 00:00:00 GMT.
`public java.lang.String getName()`	Returns the name of this business object. Names must be unique identifiers.
`public abstract <T extends BusinessObject> UpdateCommand<T> getUpdateCommand()`	Returns a command that when executed can update this business object in a back end.
`public abstract byte[] serialize()`	Serializes a business object into a byte array.
`public void setLastModified(long millis)`	Sets a timestamp indicating the last time this business object was modified. The timestamp must represent the number of milliseconds since "the epoch," namely January 1, 1970, 00:00:00 GMT.
`public void setName(java.lang.String name)`	Sets the name of this business object. Names must be unique identifiers.

The `BusinessObjectProvider` Class

Table 3–2 lists the constructor and methods belonging to the `BusinessObjectProvider` class. This provider class for instances of `BusinessObject` serves multiple purposes:

- It can be used to retrieve all the business objects from a back end by calling the `getBusinessObjects()` method.
- It can be used to create new business objects by calling the `newBusinessObject()` method.
- It provides access to a transaction manager and a session context. See "The `TransactionManager` Class" on page 48 and "The `SessionContext` Class" on page 47 for more information.

An instance of this class is created for each synchronization session. Use the methods `initialize()` and `terminate()` to allocate and free session resources.

TABLE 3–2 Class `com.sun.mep.connector.api.BusinessObjectProvider`

Method	Description
`public BusinessObjectProvider()`	No-argument constructor.
`public abstract java.util.List<T> getBusinessObjects()`	Returns a complete list of the business objects available in the back end associated with this provider. This is in essence a query method for all the instances in the back end.
`public abstract java.lang.String getRepositoryName()`	Returns the name of the repository holding these objects.
`public SessionContext getSessionContext()`	Returns the session context associated with this provider.
`public <U extends BusinessObjectProvider> TransactionManager<U> getTransactionManager()`	Returns the transaction manager associated with this provider, or null if no transaction manager has been set. If null is returned, transactions are not supported by this provider.
`public void initialize()`	This method is called right after an instance of this class is created. You can use it to allocate resources for the duration of a synchronization session. Other uses of this method include back-end authentication. Credentials needed for authentication are available from the `SessionContext`, which you can access by calling `getSessionContext()`.
`public abstract T newBusinessObject()`	Returns a fresh instance of a business object.
`public void setSessionContext(SessionContext context)`	Sets the session context for this provider.

TABLE 3–2 Class com.sun.mep.connector.api.BusinessObjectProvider *(Continued)*

Method	Description
`public void terminate()`	This method is called when a synchronization session is about to be terminated. Use this method to free any resources allocated by this object.

The Command Class

Table 3–3 lists the constructor and methods belonging to the Commandclass. This class is the base class for all business object commands. The classes that extend this class are described in "The DeleteCommand Class" on page 46, "The InsertCommand Class" on page 47, and "The UpdateCommand Class" on page 48.

TABLE 3–3 Class com.sun.mep.connector.api.Command

Method	Description
`public Command(T bobject)`	Constructor that takes a business object argument.
`public abstract void execute()`	Executes this command against a back end. Unchecked exceptions, such as `java.lang.RuntimeException`, can be used to report errors.
`public T getBusinessObject()`	Returns the business object on which this command is executed.

The DeleteCommand Class

Table 3–4 lists the constructor and method belonging to the DeleteCommand class. This class is the base class for delete business object commands. It deletes a business object from a back end.

TABLE 3–4 Class com.sun.mep.connector.api.DeleteCommand

Method	Description
`public DeleteCommand(T bobject)`	Constructor that takes a business object argument.
`public abstract void execute()`	Executes this command against a back end. Unchecked exceptions, such as `java.lang.RuntimeException`, can be used to report errors.

The InsertCommand Class

Table 3–5 lists the constructor and method belonging to the InsertCommand class. This class is the base class for insert business object commands. It inserts a business object into a back-end.

TABLE 3–5 Class com.sun.mep.connector.api.InsertCommand

Method	Description
`public InsertCommand(T bobject)`	Constructor that takes a business object argument.
`public abstract void execute()`	Executes this command against a back end. Unchecked exceptions, such as `java.lang.RuntimeException`, can be used to report errors.

The SessionContext Class

Table 3–6 lists the constructor and methods belonging to the SessionContext class. This class stores contextual information about the session in which a BusinessObjectProvider is instantiated. This information includes credentials for logging into an EIS/EAI system or a database as well as well as properties associated with an Enterprise Connector..

TABLE 3–6 Class com.sun.mep.connector.api.SessionContext

Method	Description
`public SessionContext(java.lang.String username, java.lang.String password)`	Two-argument constructor that accepts username and password credentials.
`public java.util.Map<java.lang.String, java.lang.String> getParameters()`	Returns all properties specified in the Admin Console's definition for this Enterprise Connector.
`public java.lang.String getPassword()`	Returns the password used to log into the Enterprise Connector.
`public java.lang.String getUsername()`	Returns the user name used to log into the Enterprise Connector.
`public void setParameter(java.lang.String name, java.lang.String value)`	Stores a name and value pair in the internal map.

The `TransactionManager` Class

Table 3–7 lists the constructor and methods belonging to the `TransactionManager` class. This class provides the transaction manager for a business object provider class. It supports methods for starting, stopping, and aborting back-end transactions.

TABLE 3–7 Class `com.sun.mep.connector.api.TransactionManager`

Method	Description
`public TransactionManager(T bobjectProvider)`	Constructor that creates a new `TransactionManager` for the specified business object provider.
`public void abortTransaction()`	Rolls back the current transaction on the back-end system. By default, this operation is a no-op.
`public void beginTransaction()`	Starts a new transaction on the back-end system. All business object updates, deletes and inserts will be executed in a transaction. By default, this operation is a no-op.
`public void endTransaction()`	Ends the current transaction on the back-end system. All business object updates, deletes and inserts will be executed in a transaction. By default, this operation is a no-op.
`public T getBusinessObjectProvider()`	Returns the business object manager associated with this transaction manager.

The `UpdateCommand` Class

Table 3–8 lists the constructor and method belonging to the `UpdateCommand` class. This class is the base class for update business object commands. It updates a business object in a back end.

TABLE 3–8 Class `com.sun.mep.connector.api.UpdateCommand`

Method	Description
`public UpdateCommand(T bobject)`	Constructor that takes a business object argument.
`public abstract void execute()`	Executes this command against a back end. Unchecked exceptions, such as `java.lang.RuntimeException`, can be used to report errors.