Getting Started with Berkeley DB Java Edition Direct Persistence Layer

Makers of Berkeley DB

Legal Notice

To obtain a copy of this document's original source code, please write to <support@sleepycat.com>.

Published 5/25/2006

Table of Contents

Preface

This document describes how to use the Direct Persistence Layer (DPL) to store and retrieve Java object data. The DPL is introduced in this document in a tutorial-style fashion. This document describes the DPL concepts and the scenarios under which you would use it.

This book is aimed at the software engineer responsible for writing an application in which the ability to persist object data is a desired feature.


The DPL is currently in beta.

Conventions Used in this Book

The following typographical conventions are used within in this manual:

Class names are represented in `monospaced font`, as are `method names`. For example: "The `Environment()` constructor returns an `Environment` class object."

Variable or non-literal text is presented in *italics*. For example: "Go to your *JE_HOME* directory."

Program examples are displayed in a `monospaced font` on a shaded background. For example:

In some situations, programming examples are updated from one chapter to the next. When this occurs, the new code is presented in **`monospaced bold`** font. For example:


Finally, notes of special interest are represented using a note block such as this.

For More Information

Beyond this manual, you may also find the following sources of information useful when building a transactional JE application:

- Berkeley DB, Java Edition Getting Started with Transaction Processing [http://www.sleepycat.com/jedocs/TransactionGettingStarted/BerkeleyDB-JE-Txn.pdf]
- Berkeley DB Java Edition Getting Started Guide [http://www.sleepycat.com/jedocs/GettingStartedGuide/BerkeleyDB-JE-GSG.pdf]
- Berkeley DB Java Edition Javadoc [http://www.sleepycat.com/jedocs/java/index.html]

Chapter 1. Introduction

This document provides an introduction to the concepts and APIs used to store Java objects in Berkeley DB, Java Edition using the Direct Persistence Layer (DPL) which is currently in beta. The DPL is a layer on top of the Berkeley DB, Java Edition library, and as such it offers the same high-quality data guarantees as does JE.

By using the DPL, you can cause any Java type to be persistent without implementing special interfaces. The only real requirement is that each persistent class have a default constructor.

The DPL offers the following features:

- Type safe, convenient way of accessing persistent objects.
- No hand-coding of bindings is required.
- No external schema is required to define primary and secondary keys. Java annotations are used to define all metadata.
- Interoperability with external components is supported using the Java collections framework. Any index can be accessed using a standard `java.util` collection.
- Class evolution is explicitly supported. This means you can add fields or widen types automatically and transparently.

  You can also perform many incompatible class changes, such as renaming fields or refactoring a single class. This is done using a built-in DPL mechanism called *mutations*. Mutations are automatically applied as data is accessed so as to avoid downtime to convert large databases during a software upgrade.
- Persistent class fields can be private, package-private, protected or public. The DPL can access persistence fields either by bytecode enhancement or by reflection.
- The performance of the underlying JE engine is safe-guarded. All DPL operations are mapped directly to the underlying APIs, object bindings are lightweight, and all engine tuning parameters are available.
- Java 1.5 generic types and annotations are supported.
- Transactions can be used with the DPL.

Note that Sleepycat recommends you use the DPL if all you want to do is make classes with a relatively static schema to be persistent. However, the DPL requires Java 1.5, so if you want to use Java 1.4 then you must use the Berkeley DB, Java Edition APIs.

Further, if you are porting an application between Berkeley DB and Berkeley DB, Java Edition, then you should use the Berkeley DB, Java Edition APIs instead of the DPL APIs. Additionally, if your application uses a highly dynamic schema, then you might want to

use the Berkeley DB, Java Edition API instead of the DPL, although the use of annotations can weaken this recommendation to a degree.

Java 1.5 Features

The DPL makes use of two features that are specific to Java 1.5.

The Java 1.5 features used by the DPL are:

- Generic Types

  These are used to provide type safety for index and cursor objects. If you want to use the DPL, and you are content to avoid this feature, do not declare your index and cursor objects using generic type parameters.

- Annotations

  Annotations allow you to provide metadata about your classes. In particular, you use annotations to identify whether a class is an entity or persistent class. You also use annotations to declare whether data members are primary or secondary keys.

  You do not have to use annotations. As an alternative, you can provide an alternate source of metadata by implementing an `EntityModel` class. Naming conventions, static members or an XML configuration file can be used as a source of metadata if you go this route.

Library Dependencies

To use the DPL:

- You must use Java 1.5.

- Use the Berkeley DB, Java Edition version 3.0 jar file, just as if you were using the Berkeley DB, Java Edition APIs themselves (see the *Berkeley DB, Java Edition Getting Started Guide* for more information). This jar file includes all the APIs that you need in order to use the DPL.

Chapter 2. Database Environments

The DPL requires you to use a database environment. The environment is used by the DPL to manage the databases that it maintains for you, and to manage transactions.

You also use the database environment for administrative and configuration activities related to your database log files and the in-memory cache. See the *Berkeley DB Java Edition Getting Started Guide* for more information.

Opening Database Environments

You open a database environment by instantiating an `Environment` object. You must provide to the constructor the name of the on-disk directory where the environment is to reside. This directory location must exist or the open will fail.

By default, the environment is not created for you if it does not exist. Set the creation property to `true` if you want the environment to be created. For example:

```
package persist.gettingStarted;

import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;

import java.io.File;

...

// Open the environment. Allow it to be created if it does not already exist.
Environment myDbEnvironment = null;

try {
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setAllowCreate(true);
    myDbEnvironment = new Environment(new File("/export/dbEnv"), envConfig);
} catch (DatabaseException dbe) {
    // Exception handling goes here
}
```

Your application can open and use as many environments as you have disk and memory to manage, although most applications will use just one environment. Also, you can instantiate multiple `Environment` objects for the same physical environment.

Opening an environment usually causes some background threads to be started. JE uses these threads for log file cleaning and some administrative tasks. However, these threads will only be opened once per process, so if you open the same environment more than once from within the same process, there is no performance impact on your application. Also, if you open the environment as read-only, then the background threads (with the exception of the evictor thread) are not started.

Note that opening your environment causes normal recovery to be run. This causes the underlying databases used by the DPL to be brought into a consistent state relative to the changed data found in your log files. See the *Berkeley DB Java Edition Getting Started Guide* for more information on background threads and database log files.

Closing Database Environments

You close your environment by calling the `Environment.close()` method. This method performs a checkpoint, so it is not necessary to perform a sync or a checkpoint explicitly before calling it. For information on checkpoints, see the *JE Transaction Processing Guide*. For information on syncs, see the *Berkeley DB Java Edition Getting Started Guide*.

```
import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;

...

try {
    if (myDbEnvironment != null) {
        myDbEnvironment.close();
    }
} catch (DatabaseException dbe) {
    // Exception handling goes here
}
```

You should close your environment(s) only after all other store activities have completed and you have closed any stores currently opened in the environment.

It is possible for the environment to close before JE's cleaner thread has finished its work. This happens if you perform a large number of deletes immediately before shutting down your environment. The result is that your log files may be quit a lot larger than you expect them to be because the cleaner thread has not had a chance to finish its work.

If want to make sure that the cleaner has finished running before the environment is closed, call `Environment.cleanLog()` before calling `Environment.close()`:

```
import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;

...

try {
    if (myDbEnvironment != null) {
        myDbEnvironment.cleanLog(); // Clean the log before closing
        myDbEnvironment.close();
    }
} catch (DatabaseException dbe) {
    // Exception handling goes here
}
```

Closing the last environment handle in your application causes all internal data structures to be released and the background threads to be stopped. If there are any opened stores, then JE will complain before closing them as well.

Environment Properties

You set properties for the `Environment` using the `EnvironmentConfig` class. You can also set properties for a specific `Environment` instance using `EnvironmentMutableConfig`.

The EnvironmentConfig Class

The `EnvironmentConfig` class makes a large number of fields and methods available to you. Describing all of these tuning parameters is beyond the scope of this manual. However, there are a few properties that you are likely to want to set. They are described here.

Note that for each of the properties that you can commonly set, there is a corresponding getter method. Also, you can always retrieve the `EnvironmentConfig` object used by your environment using the `Environment.getConfig()` method.

You set environment configuration parameters using the following methods on the `EnvironmentConfig` class:

- `EnvironmentConfig.setAllowCreate()`

  If `true`, the database environment is created when it is opened. If `false`, environment open fails if the environment does not exist. This property has no meaning if the database environment already exists. Default is `false`.

- `EnvironmentConfig.setReadOnly()`

  If `true`, then all databases opened in this environment must be opened as read-only. If you are writing a multi-process application, then all but one of your processes must set this value to `true`. Default is `false`.

  You can also set this property using the `je.env.isReadOnly` parameter in your *env_home*`/je.properties` file.

- `EnvironmentConfig.setTransactional()`

  If `true`, configures the database environment to support transactions. Default is `false`.

  You can also set this property using the `je.env.isTransactional` parameter in your *env_home*`/je.properties` file.

For example:

```
package persist.gettingStarted;

import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
```

```
import java.io.File;

...

Environment myDatabaseEnvironment = null;
try {
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setAllowCreate(true);
    envConfig.setTransactional(true);
    myDatabaseEnvironment =
        new Environment(new File("/export/dbEnv"), envConfig);
} catch (DatabaseException dbe) {
   System.err.println(dbe.toString());
   System.exit(1);
}
```

EnvironmentMutableConfig

`EnvironmentMutableConfig` manages properties that can be reset after the `Environment` object has been constructed. In addition, `EnvironmentConfig` extends `EnvironmentMutableConfig`, so you can set these mutable properties at `Environment` construction time if necessary.

The `EnvironmentMutableConfig` class allows you to set the following properties:

- `setCachePercent()`

  Determines the percentage of JVM memory available to the JE cache. See the *Berkeley DB, Java Edition Getting Started Guide* for more information.

- `setCacheSize()`

  Determines the total amount of memory available to the database cache. See the *Berkeley DB, Java Edition Getting Started Guide* for more information.

- `setTxnNoSync()`

  Determines whether change records created due to a transaction commit are written to the backing log files on disk. A value of `true` causes the data to not be flushed to disk. See the *JE Transaction Processing Guide*.

- `setTxnWriteNoSync()`

  Determines whether logs are flushed on transaction commit (the logs are still written, however). By setting this value to `true`, you potentially gain better performance than if you flush the logs on commit, but you do so by losing some of your transaction durability guarantees.

There is also a corresponding getter method (`getTxnNoSync()`). Moreover, you can always retrieve your environment's `EnvironmentMutableConfig` object by using the `Environment.getMutableConfig()` method.

For example:

```
package persist.gettingStarted;

import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentMutableConfig;

import java.io.File;


...

try {
    Environment myEnv = new Environment(new File("/export/dbEnv"), null);
    EnvironmentMutableConfig envMutableConfig =
        new EnvironmentMutableConfig();
    envMutableConfig.setTxnNoSync(true);
    myEnv.setMutableConfig(envMutableConfig);
} catch (DatabaseException dbe) {
    // Exception handling goes here
}
```

Environment Statistics

JE offers a wealth of information that you can examine regarding your environment's operations. The majority of this information involves numbers relevant only to the JE developer and as such a description of those statistics is beyond the scope of this manual.

However, one statistic that is very important (especially for long-running applications) is `EnvironmentStats.getNCacheMiss()`. This statistic returns the total number of requests for database objects that were not serviceable from the cache. This number is important to the application administrator who is attempting to determine the proper size for the in-memory cache. See the *Berkeley DB, Java Edition Getting Started Guide* for details.

To obtain this statistic from your environment, call `Environment.getStats()` to return an `EnvironmentStats` object. You can then call the `EnvironmentStats.getNCacheMiss()` method. For example:

```
import com.sleepycat.je.Environment;

...

long cacheMisses = myEnv.getStats(null).getNCacheMiss();

...
```

Note that `Environment.getStats()` can only obtain statistics from with your application's process. In order for the application administrator to obtain this statistic, you must either use JMX to retrieve the statistic (see the *Berkeley DB, Java Edition Getting Started Guide* for more information) or you must print it for examination (for example, log the value once a minute).

Remember that what is really important for cache sizing is the change in this value over time, and not the actual value itself. So you might consider offering a delta from one examination of this statistic to the next (a delta of 0 is desired while large deltas are an indication that the cache is too small).

Chapter 3. Getting Going with the Direct Persistence Layer

In this chapter we will build a couple of basic applications that use the DPL to store and retrieve objects. To do this, we will create two applications that use fictional inventory and vendor information. The first application, `ExampleDatabasePut`, is used to create inventory and vendor objects that are stored for later retrieval. The second application, `ExampleInventoryRead`, is used to retrieve and display this data.

The examples that we use here are identical to the examples provided in the *Berkeley DB, Java Edition Getting Started Guide*. The only difference is that the DPL is used instead of the JE API. We did this to make it easier to compare the two APIs.

Before we begin building our main applications, which are used to perform data reads and writes, we have to build several other classes that provide important infrastructure for our application. These classes encapsulate the data we want to store, provide data access to the data store, and open and close our data store.

Persistent Objects

To begin, we build the classes that we actually want to store. In our case, we build two such classes; a class that contains product inventory information and a class that contains vendor contact information.

These classes look pretty much the same as any class might that encapsulates data. They have private data members that hold the information and they have getter and setter methods for data access and retrieval.

However, to use these with the DPL, the classes must be decorated with Java annotations that identify the classes as either an entity class or a persistent class. Java annotations are also used to identify primary and secondary indices.

Entity versus Persistent

The DPL is used to store Java objects in an underlying series of databases (accessed via an *EntityStore*). To do this, you must identify classes to be stored as either entity classes or persistent classes.

Entity classes are classes that have a primary index, and optionally one or more secondary indices. That is, these are the classes that you will save and retrieve directly using the DPL. You identify an entity class using the `@Entity` java annotation.

Persistent classes are classes used by entity classes. They do not have primary or secondary indices used for object retrieval. Rather, they are stored or retrieved when an entity class makes direct use of them. You identify an persistent class using the `@Persistent` java annotation.

Note that all non-transient instance fields of a persistent class, as well as its superclasses and subclasses, are persistent. static and transient fields are not persistent. The persistent fields of a class may be private, package-private (default access), protected or public.

Note that simple Java types, such as `java.lang.String` and `java.util.Date`, are automatically handled as a persistent class when you use them in an entity class; you do not have to do anything special to cause these simple Java objects to be stored in the `EntityStore`.

Vendor.class

The simplest class that our example wants to store contains vendor contact information. This class contains no secondary indices so all we have to do is identify it as an entity class and identify the field in the class used for the primary key.

Primary and secondary indices are described in detail in the *Berkeley DB, Java Edition Getting Started Guide*, but essentially a primary index is the *main* information you use to organize and retrieve a given object. Primary index keys are always unique to the object in order to make it easier to locate them in the data store.

Conversely, secondary indices represent other information that you might use to locate an object. We discuss these more in the next section.

In the following example, we identify the `vendor` data member as containing the primary key. This data member is meant to contain a vendor's name. Because of the way we will use our `EntityStore`, the value provided for this data member must be unique within the store or runtime errors will result.

When used with the DPL, our `Vendor` class appears as follows. Notice that the `@Entity` annotation appears immediately before the class declaration, and the `@PrimaryKey` annotation appears immediately before the `vendor` data member declaration.

```
package persist.gettingStarted;

import com.sleepycat.persist.model.Entity;
import com.sleepycat.persist.model.PrimaryKey;

@Entity
public class Vendor {

    private String address;
    private String bizPhoneNumber;
    private String city;
    private String repName;
    private String repPhoneNumber;
    private String state;

    // Primary key is the vendor's name
    // This assumes that the vendor's name is
    // unique in the database.
```

```
    @PrimaryKey
    private String vendor;

    private String zipcode;

    public void setRepName(String data) {
        repName = data;
    }

    public void setAddress(String data) {
        address = data;
    }

    public void setCity(String data) {
        city = data;
    }

    public void setState(String data) {
        state = data;
    }

    public void setZipcode(String data) {
        zipcode = data;
    }

    public void setBusinessPhoneNumber(String data) {
        bizPhoneNumber = data;
    }

    public void setRepPhoneNumber(String data) {
        repPhoneNumber = data;
    }

    public void setVendorName(String data) {
        vendor = data;
    }

    public String getRepName() {
        return repName;
    }

    public String getAddress() {
        return address;
    }

    public String getCity() {
        return city;
    }
```

```
    public String getState() {
        return state;
    }

    public String getZipcode() {
        return zipcode;
    }

    public String getBusinessPhoneNumber() {
        return bizPhoneNumber;
    }

    public String getRepPhoneNumber() {
        return repPhoneNumber;
    }
}
```

For this class, the `vendor` value is set for an individual `Vendor` class object by the `setVendorName()` method. If our example code fails to set this value before storing the object, the data member used to store the primary key is set to a null value. This would result in a runtime error.

You can avoid the need to explicitly set a value for a class' primary index by specifying a sequence to be used for the primary key. This results in an unique integer value being used as the primary key for each stored object.

You declare a sequence is to be used by specifying the `sequence` keyword to the `@PrimaryKey` annotation. For example:

```
@PrimaryKey(sequence="")
long myPrimaryKey;
```

If you provide the `sequence` keyword with a name, then the sequence is obtained from that named sequence. For example:

```
@PrimaryKey(sequence="Sequence_Namespace")
long myPrimaryKey;
```

Inventory.class

Our example's `Inventory` class is much like our `Vendor` class in that it is simply used to encapsulate data. However, in this case we want to be able to access objects two different ways: by product SKU and by product name.

In our data set, the product SKU is required to be unique, so we use that as the primary key. The product name, however, is not a unique value so we set this up as a secondary key.

The class appears as follows in our example:

```
package persist.gettingStarted;

import com.sleepycat.persist.model.Entity;
import com.sleepycat.persist.model.PrimaryKey;
import static com.sleepycat.persist.model.Relationship.*;
import com.sleepycat.persist.model.SecondaryKey;

@Entity
public class Inventory {

    // Primary key is sku
    @PrimaryKey
    private String sku;

    // Secondary key is the itemName
    @SecondaryKey(relate=MANY_TO_ONE)
    private String itemName;

    private String category;
    private String vendor;
    private int vendorInventory;
    private float vendorPrice;

    public void setSku(String data) {
        sku = data;
    }

    public void setItemName(String data) {
        itemName = data;
    }

    public void setCategory(String data) {
        category = data;
    }

    public void setVendorInventory(int data) {
        vendorInventory = data;
    }

    public void setVendor(String data) {
        vendor = data;
    }

    public void setVendorPrice(float data) {
        vendorPrice = data;
    }

    public String getSku() {
        return sku;
```

```
    }

    public String getItemName() {
        return itemName;
    }

    public String getCategory() {
        return category;
    }

    public int getVendorInventory() {
        return vendorInventory;
    }

    public String getVendor() {
        return vendor;
    }

    public float getVendorPrice() {
        return vendorPrice;
    }
}
```

Secondary Indices

To declare a secondary index, we use the `@SecondaryKey` annotation. Note that when we do this, we must declare what sort of an index it is; that is, what is its relationship to other data in the data store.

The *kind* of indices that we can declare are:

- `ONE_TO_ONE`

  This relationship indicates that the secondary key is unique to the object. If an object is stored with a secondary key that already exists in the data store, a run time error is raised.

  For example, a person object might be stored with a primary key of a social security number (in the US), with a secondary key of the person's employee number. Both values are expected to be unique in the data store.

- `MANY_TO_ONE`

  Indicates that the secondary key may be used for multiple objects in the data store. That is, the key appears more than once, but for each stored object it can be used only once.

  Consider a data store that relates managers to employees. A given manager will have multiple employees, but each employee is assumed to have just one manager. In this

case, the manager's employee number might be a secondary key, so that you can quickly locate all the objects related to that manager's employees.

- `ONE_TO_MANY`

  Indicates that the secondary key might be used more than once for a given object. Index keys themselves are assumed to be unique, but multiple instances of the index can be used per object.

  For example, employees might have multiple unique email addresses. In this case, any given object can be access by one or more email addresses. Each such address is unique in the data store, but each such address will relate to a single employee object.

- `MANY_TO_MANY`

  There can be multiple keys for any given object, and for any given key there can be many related objects.

  For example, suppose your organization has a shared resource, such as printers. You might want to track which printers a given employee can use (there might be more than one). You might also want to track which employees can use a specific printer. This represents a many-to-many relationship.

Note that for `ONE_TO_ONE` and `MANY_TO_ONE` relationships, you need a simple data member (not an array or collection) to hold the key. For `ONE_TO_MANY` and `MANY_TO_MANY` relationships, you need an array or collection to hold the keys:

```
@SecondaryKey(relate=ONE_TO_ONE)
private String primaryEmailAddress = new String();

@SecondaryKey(relate=ONE_TO_MANY)
private Set<String> emailAddresses = new HashSet<String>();
```

Foreign Key Constraints

Sometimes a secondary index is related in some way to another entity class that is also contained in the data store. That is, the secondary key might be the primary key for another entity class. If this is the case, you can declare the foreign key constraint to make data integrity easier to accomplish.

For example, you might have one class that is used to represent employees. You might have another that is used to represent corporate divisions. When you add or modify an employee record, you might want to ensure that the division to which the employee belongs is known to the data store. You do this by specifying a foreign key constraint.

When a foreign key constraint is declared:

- When a new secondary key for the object is stored, it is checked to make sure it exists as a primary key for the related entity object. If it does not, a runtime error occurs.

- When a related entity is deleted (that is, a corporate division is removed from the data store), some action is automatically taken for the entities that refer to this object (that is, the employee objects). Exactly what that action is, is definable by you. See below.

When a related entity is deleted from the data store, one of the following actions are taken:

- `ABORT`

  The delete operation is not allowed. A runtime error is raised as a result of the operation. This is the default behavior.

- `CASCADE`

  All entities related to this one are deleted as well. For example, if you deleted a `Division` object, then all `Employee` objects that belonged to the division are also deleted.

- `NULLIFY`

  All entities related to the deleted entity are updated so that the pertinent data member is nullified. That is, if you deleted a division, then all employee objects related to that division would have their division key automatically set to null.

You declare a foreign key constraint by using the `relatedEntity` keyword. You declare the foreign key constraint deletion policy using the `onRelatedEntityDelete` keyword. For example, the following declares a foreign key constraint to `Division` class objects, and it causes related objects to be deleted if the `Division` class is deleted:

```
@SecondaryKey(relate=ONE_TO_ONE, relatedEntity=Division.class,
    onRelatedEntityDelete=CASCADE)
private String division = new String();
```

Entity Stores

All entity and persistent objects are saved to and retrieved from an *entity store*. Entity stores can contain multiple class types. In fact, you typically only need one entity store for a given application.

To use an entity store, you must first open an environment and then provide that environment handle to the `EntityStore` constructor. When you shutdown your application, first close your entity store and then close your environment.

Entity stores have configurations in the same way that environments have configurations. You can use a `StoreConfig` object to identify store properties, such as whether is it allowed to create the store if it does not currently exist or if the store is read-only. You also use the `StoreConfig` to declare whether the store is transactional.

`EntityStore` objects also provide methods for retrieving information about the store, such as the store's name or a handle to the underlying environment. You can also use the

`EntityStore` to retrieve all the primary and secondary indices related to a given type of entity object contained in the store. See Accessing Indices (page 19) for more information.

MyDbEnv

The applications that we are building for our example both must open and close environments and entity stores. One of our applications is writing to the entity store, so this application needs to open the store as read-write. It also wants the store to be transactional, and it wants to be able to create the store if it does not exist.

Our second application only reads from the store. In this case, the store should be opened as read-only.

We perform these activities by creating a single class that is responsible for opening and closing our store and environment. This class is shared by both our applications. To use it, callers need to only provide the path to the environment home directory, and to indicate whether the object is meant to be read-only. The class implementation is as follows:

```
package persist.gettingStarted;

import java.io.File;

import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;

import com.sleepycat.persist.EntityStore;
import com.sleepycat.persist.StoreConfig;

public class MyDbEnv {

    private Environment myEnv;
    private EntityStore store;

    // Our constructor does nothing
    public MyDbEnv() {}

    // The setup() method opens the environment and store
    // for us.
    public void setup(File envHome, boolean readOnly)
        throws DatabaseException {

        EnvironmentConfig myEnvConfig = new EnvironmentConfig();
        StoreConfig storeConfig = new StoreConfig();

        myEnvConfig.setReadOnly(readOnly);
        storeConfig.setReadOnly(readOnly);

        // If the environment is opened for write, then we want to be
```

```
        // able to create the environment and entity store if
        // they do not exist.
        myEnvConfig.setAllowCreate(!readOnly);
        storeConfig.setAllowCreate(!readOnly);

        // Allow transactions if we are writing to the store.
        myEnvConfig.setTransactional(!readOnly);
        storeConfig.setTransactional(!readOnly);

        // Open the environment and entity store
        myEnv = new Environment(envHome, myEnvConfig);
        store = new EntityStore(myEnv, "EntityStore", storeConfig);

    }

    // Return a handle to the entity store
    public EntityStore getEntityStore() {
        return store;
    }

    // Return a handle to the environment
    public Environment getEnv() {
        return myEnv;
    }

    // Close the store and environment.
    public void close() {
        if (store != null) {
            try {
                store.close();
            } catch(DatabaseException dbe) {
                System.err.println("Error closing store: " +
                                   dbe.toString());
               System.exit(-1);
            }
        }

        if (myEnv != null) {
            try {
                // Finally, close the store and environment.
                myEnv.close();
            } catch(DatabaseException dbe) {
                System.err.println("Error closing MyDbEnv: " +
                                   dbe.toString());
               System.exit(-1);
            }
        }
    }
}
```

Accessing Indices

Before we write the main parts of our example programs, there is one last common piece of code that we need. Both our examples are required to work with primary and secondary indices contained within our `EntityStore`. We could just refer to these directly every time we need to access a primary or secondary index, but instead we choose to encapsulate this information in a class so as to share that code between the applications.

Accessing Primary Indices

You retrieve a primary index using the `EntityStore.getPrimaryIndex()` method. To do this, you indicate the index key type (that is, whether it is a String, Integer, and so forth) and the class of the entities stored in the index.

For example, the following retrieves the primary index for the `Inventory` class. These index keys are of type `String`.

```
PrimaryIndex<String,Inventory> inventoryBySku =
    store.getPrimaryIndex(String.class, Inventory.class);
```

Accessing Secondary Indices

You retrieve a secondary index using the `EntityStore.getSecondaryIndex()` method. Because secondary indices actually refer to a primary index somewhere in your data store, to access a secondary index you:

1. Provide the primary index as returned by `EntityStore.getPrimaryIndex()`.
2. Identify the key data type used by the secondary index (`String`, `Long`, and so forth).
3. Identify the name of the secondary key field. When you declare the `SecondaryIndex` object, you identify the entity class to which the secondary index must refer.

For example, the following first retrieves the primary index, and then uses that to retrieve a secondary index. The secondary key is held by the `itemName` field of the `Inventory` class.

```
PrimaryIndex<String,Inventory> inventoryBySku =
store.getPrimaryIndex(String.class, Inventory.class);

SecondaryIndex<String,String,Inventory> inventoryByName =
    store.getSecondaryIndex(inventoryBySku, String.class, "itemName");
```

DataAccessor.class

Now that you understand how to retrieve primary and secondary indices, we can implement our `DataAccessor` class. Again, this class is shared by both our example programs and it is used to access the primary and secondary indices that our programs use.

If you compare this class against our `Vendor` and `Inventory` class implementations, you will see that the primary and secondary indices declared there are referenced by this class.

See Vendor.class (page 10) and Inventory.class (page 12) for those implementations.

```
package persist.gettingStarted;

import java.io.File;

import com.sleepycat.je.DatabaseException;
import com.sleepycat.persist.EntityStore;
import com.sleepycat.persist.PrimaryIndex;
import com.sleepycat.persist.SecondaryIndex;

public class DataAccessor {
    // Open the indices
    public DataAccessor(EntityStore store)
        throws DatabaseException {

        // Primary key for Inventory classes
        inventoryBySku = store.getPrimaryIndex(
            String.class, Inventory.class);

        // Secondary key for Inventory classes
        // Last field in the getSecondaryIndex() method must be
        // the name of a class member; in this case, an Inventory.class
        // data member.
        inventoryByName = store.getSecondaryIndex(
            inventoryBySku, String.class, "itemName");

        // Primary key for Vendor class
        vendorByName = store.getPrimaryIndex(
            String.class, Vendor.class);
    }

    // Inventory Accessors
    PrimaryIndex<String,Inventory> inventoryBySku;
    SecondaryIndex<String,String,Inventory> inventoryByName;

    // Vendor Accessors
    PrimaryIndex<String,Vendor> vendorByName;
}
```

Chapter 4. Writing to Entity Stores

Once you have implemented your classes for storage using the DPL, then writing your instances to the entity store is straight-forward. Simply open the `EntityStore` and then put the instance to it using the appropriate primary index (which you retrieve from the entity store).

There are a few caveats that you should consider before we proceed.

First, if you have not provided a value to the primary key field for the object you are storing, then you will see runtime errors when you try to write the object to the store. The primary key is not allowed to be null.

As described in Vendor.class (page 10), you can avoid the need to explicitly set a primary key by declaring a sequence for the primary key. In our example, however, we explicitly set a primary key value.

Secondly, you need to decide if you want to use transactions to write to the entity store. Transactions are a mechanism that allow you to group multiple write operations in a single atomic unit; either all the operations succeed or none of them succeed. Transactions also provide isolation guarantees for multi-threaded applications.

Transactions represent a large topic that are described in the *JE Getting Started with Transaction Processing* guide. For our purposes here, it is important to know the following:

- Once you open an `EntityStore` as transactional, all write operations are performed using a transaction whether or not you explicitly use transactions. If you do not explicitly use a transaction, then each individual write operation is performed under a transaction using *auto-commit*.
- If you explicitly use a transaction, then you must either commit or abort the transaction when you are done performing write operations for that transaction instance.

All of these topics, and more, are described in detail in the Transaction Processing guide noted above.

Because it is likely that your application will use transactions, we show their usage here in a minimal way.

ExampleDatabasePut.class

Our example reads inventory and vendor information from flat text files, encapsulates this data in objects of the appropriate type, and then writes each object to an `EntityStore`.

`Vendor` objects are written one at a time without use of an explicit transaction handle. Because the store is opened for transactional writes, this means that each such write operation is performed under a transaction using auto-commit.

`Inventory` class objects are all written using a single transaction. This means that if any failures are encountered during the data write, the entire operation is aborted and the store will have no inventory information in it.

There only reason why we use transactions in this way is for illustrative purposes. You could just as easily use auto-commit for all the operations, or wrap both the vendor and the inventory writes in a single giant transaction, or do something in between.

To begin, we import the Java classes that our example needs. Most of the imports are related to reading the raw data from flat text files and breaking them apart for usage with our data classes. We also import classes from the JE package, but we do not actually import any classes from the DPL. The reason why is because we have placed almost all of our DPL work off into other classes, so there is no need for direct usage of those APIs here.

```
package persist.gettingStarted;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Transaction;
```

Now we can begin the class itself. Here we set default paths for the on-disk resources that we require (the environment home, and the location of the text files containing our sample data). We also declare `DataAccessor` and `MyDbEnv` members. We describe these classes and show their implementation in DataAccessor.class (page 19) and MyDbEnv (page 17).

```
public class ExampleDatabasePut {

    private static File myDbEnvPath = new File("/tmp/JEDB");
    private static File inventoryFile = new File("./inventory.txt");
    private static File vendorsFile = new File("./vendors.txt");

    private DataAccessor da;

    // Encapsulates the environment and data store.
    private static MyDbEnv myDbEnv = new MyDbEnv();
```

Next, we provide our `usage()` method. The command line options provided there are necessary only if the default values to the on-disk resources are not sufficient.

```
    private static void usage() {
        System.out.println("ExampleDatabasePut [-h <env directory>]");
```

```
        System.out.println("      [-i <inventory file>] [-v <vendors file>]");
        System.exit(-1);
    }
```

Our `main()` method is also reasonably self-explanatory. We simply instantiate an `ExampleDatabasePut` object there and then call it's `run()` method. We also provide a top-level `try` block there for any exceptions that might be thrown during runtime.

Notice that the `finally` statement in the top-level `try` block calls `MyDbEnv.close()`. This method closes our `EntityStore` and `Environment` objects. By placing it here in the `finally` statement, we can make sure that our store and environment are always cleanly closed.

```
    public static void main(String args[]) {
        ExampleDatabasePut edp = new ExampleDatabasePut();
        try {
            edp.run(args);
        } catch (DatabaseException dbe) {
            System.err.println("ExampleDatabasePut: " + dbe.toString());
            dbe.printStackTrace();
        } catch (Exception e) {
            System.out.println("Exception: " + e.toString());
            e.printStackTrace();
        } finally {
            myDbEnv.close();
        }
        System.out.println("All done.");
    }
```

Our `run()` method does four things. It calls `MyDbEnv.setup()`, which opens our `Environment` and `EntityStore`. It then instantiates a `DataAccessor` object, which we will use to write data to the store. It calls `loadVendorsDb()` which loads all of the vendor information. And then it calls `loadInventoryDb()` which loads all of the inventory information.

Notice that the `MyDbEnv` object is being setup as read-write. This results in the `EntityStore` being opened for transactional support. (See MyDbEnv (page 17) for implementation details.)

```
    private void run(String args[])
        throws DatabaseException {
        // Parse the arguments list
        parseArgs(args);

        myDbEnv.setup(myDbEnvPath,  // Path to the environment home
                      false);       // Environment read-only?

        // Open the data accessor. This is used to store
        // persistent objects.
        da = new DataAccessor(myDbEnv.getEntityStore());

        System.out.println("loading vendors db....");
```

```
        loadVendorsDb();

        System.out.println("loading inventory db....");
        loadInventoryDb();
    }
```

We can now implement the `loadVendorsDb()` method. This method is responsible for reading the vendor contact information from the appropriate flat-text file, populating `Vendor` class objects with the data and then writing it to the `EntityStore`. As explained above, each individual object is written with transactional support. However, because a transaction handle is not explicitly used, the write is performed using auto-commit. This happens because the `EntityStore` was opened to support transactions.

To actually write each class to the `EntityStore`, we simply call the `PrimaryIndex.put()` method for the `Vendor` entity instance. We obtain this method from our `DataAccessor` class.

```
    private void loadVendorsDb()
            throws DatabaseException {

        // loadFile opens a flat-text file that contains our data
        // and loads it into a list for us to work with. The integer
        // parameter represents the number of fields expected in the
        // file.
        List vendors = loadFile(vendorsFile, 8);

        // Now load the data into the store.
        for (int i = 0; i < vendors.size(); i++) {
            String[] sArray = (String[])vendors.get(i);
            Vendor theVendor = new Vendor();
            theVendor.setVendorName(sArray[0]);
            theVendor.setAddress(sArray[1]);
            theVendor.setCity(sArray[2]);
            theVendor.setState(sArray[3]);
            theVendor.setZipcode(sArray[4]);
            theVendor.setBusinessPhoneNumber(sArray[5]);
            theVendor.setRepName(sArray[6]);
            theVendor.setRepPhoneNumber(sArray[7]);

            // Put it in the store. Because we do not explicitly set
            // a transaction here, and because the store was opened
            // with transactional support, auto commit is used for each
            // write to the store.
            da.vendorByName.put(theVendor);
        }
    }
```

Now we can implement our `loadInventoryDb()` method. This does exactly the same thing as the `loadVendorsDb()` method except that the entire contents of the inventory data is loaded using a single transaction.

Again, the transactional activity shown here is described in detail in the *JE Getting Started with Transaction Processing* guide.

```
    private void loadInventoryDb()
        throws DatabaseException {

        // loadFile opens a flat-text file that contains our data
        // and loads it into a list for us to work with. The integer
        // parameter represents the number of fields expected in the
        // file.
        List inventoryArray = loadFile(inventoryFile, 6);

        // Now load the data into the store. The item's sku is the
        // key, and the data is an Inventory class object.

        // Start a transaction. All inventory items get loaded using a
        // single transaction.
        Transaction txn = myDbEnv.getEnv().beginTransaction(null, null);

        for (int i = 0; i < inventoryArray.size(); i++) {
            String[] sArray = (String[])inventoryArray.get(i);
            String sku = sArray[1];

            Inventory theInventory = new Inventory();
            theInventory.setItemName(sArray[0]);
            theInventory.setSku(sArray[1]);
            theInventory.setVendorPrice(
                (new Float(sArray[2])).floatValue());
            theInventory.setVendorInventory(
                (new Integer(sArray[3])).intValue());
            theInventory.setCategory(sArray[4]);
            theInventory.setVendor(sArray[5]);

            // Put it in the store. Note that this causes our secondary key
            // to be automatically updated for us.
            try {
                  da.inventoryBySku.put(txn, theInventory);
            } catch (DatabaseException dbe) {
                System.out.println("Error putting entry " +
                    sku.getBytes("UTF-8"));
                txn.abort();
                throw dbe;
            }
        }
        // Commit the transaction. The data is now safely written to the
        // store.
        txn.commit();
    }
```

The remainder of this example simple parses the command line and loads data from a flat-text file. There is nothing here that is of specific interest to the DPL, but we show this part of the example anyway in the interest of completeness.

```
    private static void parseArgs(String args[]) {
        for(int i = 0; i < args.length; ++i) {
            if (args[i].startsWith("-")) {
                switch(args[i].charAt(1)) {
                  case 'h':
                    myDbEnvPath = new File(args[++i]);
                    break;
                  case 'i':
                    inventoryFile = new File(args[++i]);
                    break;
                  case 'v':
                    vendorsFile = new File(args[++i]);
                    break;
                  default:
                    usage();
                }
            }
        }
    }

    private List loadFile(File theFile, int numFields) {
        List<String[]> records = new ArrayList<String[]>();
        try {
            String theLine = null;
            FileInputStream fis = new FileInputStream(theFile);
            BufferedReader br =
                new BufferedReader(new InputStreamReader(fis));
            while((theLine=br.readLine()) != null) {
                String[] theLineArray = theLine.split("#");
                if (theLineArray.length != numFields) {
                    System.out.println("Malformed line found in " +
                        theFile.getPath());
                    System.out.println("Line was: '" + theLine);
                    System.out.println("length found was: " +
                        theLineArray.length);
                    System.exit(-1);
                }
                records.add(theLineArray);
            }
            // Close the input stream handle
            fis.close();
        } catch (FileNotFoundException e) {
            System.err.println(theFile.getPath() + " does not exist.");
            e.printStackTrace();
            usage();
```

```
        } catch (IOException e)  {
            System.err.println("IO Exception: " + e.toString());
            e.printStackTrace();
            System.exit(-1);
        }
        return records;
    }

    protected ExampleDatabasePut() {}
}
```

This completes our example application. Be aware that this example and supporting classes can be found in your JE distribution in the following location:

JE_HOME/examples/persist/gettingStarted

where *JE_HOME* is the location where you placed your JE distribution.

Chapter 5. Reading Data From Entity Stores

Once you have written objects to an `EntityStore` you can read them back again at a later time. To do this, you use either the primary or the secondary key used by the entity that you want to retrieve.

It is possible to retrieve a single object by using its primary key. You can also retrieve multiple objects in the form of a collection, if the key that you use is capable of referring to more than one object.

Retrieving a Single Object

To retrieve a single object, you must use an index that is capable of returning just one object. A primary index is one such index, as is a `ONE_TO_ONE` or `ONE_TO_MANY` secondary index.

To retrieve the object, use the index's `get()` method, providing to it the index value that is used by the desired object. For example, the following uses the primary key for the `Inventory` class to retrieve the object with SKU `Trifdess2zJsGi`:

```
PrimaryIndex<String,Inventory> inventoryBySku =
    store.getPrimaryIndex(String.class, Inventory.class);
Inventory inventory = inventoryBySku.get("Trifdess2zJsGi");
```

Retrieving Multiple Objects

Some indices result in the retrieval of multiple objects. For example, `MANY_TO_ONE` indices can result in more than one object for any given key. When this is the case, you must iterate over the resulting set of objects in order to examine each object in turn.

To retrieve a set of objects, use the `SecondaryIndex.subIndex()` method to return an `EntityIndex` class object. Then, use that object's `entities()` method to obtain an `EntityCursor`. This cursor allows you to iterate over the object set.

For example:

```
PrimaryIndex<String,Inventory> inventoryBySku =
    store.getPrimaryIndex(String.class, Inventory.class);

SecondaryIndex<String,String,Inventory> inventoryByName =
    store.getSecondaryIndex(inventoryBySku, String.class, "itemName");

EntityCursor<Inventory> items =
    inventoryByName.subIndex("Lemon").entities();
```

Now that we have the entity cursor, we can iterate over the resulting objects. Here we use the new Java 1.5 enhanced `for` notation to perform the iteration:

```
try {
    for (Inventory item : items) {
```

```
        // do something with each Inventory object "item"
    }
// Always make sure the cursor is closed when we are done with it.
} finally {
    items.close();
}
```

Note that it is also possible to iterate over every entity object of a given type in the entity store. You do this using the entity's primary index, and then call its `entities()` method to obtain an `EntityCursor`:

```
PrimaryIndex<String,Inventory> inventoryBySku =
    store.getPrimaryIndex(String.class, Inventory.class);

EntityCursor<Inventory> items = inventoryBySku.entities();

try {
    for (Inventory item : items) {
        // do something with each Inventory object "item"
    }
// Always make sure the cursor is closed when we are done with it.
} finally {
    items.close();
}
```

ExampleInventoryRead.class

To illustrate the points that we make in the previous sections, we wrote `ExampleInventoryRead` example application. This program retrieves inventory information from our entity store and displays it. When it displays each inventory item, it also displays the related vendor contact information.

`ExampleInventoryRead` can do one of two things. If you provide no search criteria, it displays all of the inventory items in the store. If you provide an item name (using the `-s` command line switch), then just those inventory items using that name are displayed.

The beginning of our example is almost identical to our `ExampleDatabasePut` example program. We repeat that example code here for the sake of completeness. For a complete walk-through of it, see ExampleDatabasePut.class (page 21).

```
package persist.gettingStarted;

import java.io.File;
import java.io.IOException;

import com.sleepycat.je.DatabaseException;
import com.sleepycat.persist.EntityCursor;

public class ExampleInventoryRead {
```

```
    private static File myDbEnvPath =
        new File("/tmp/JEDB");

    private DataAccessor da;

    // Encapsulates the database environment.
    private static MyDbEnv myDbEnv = new MyDbEnv();

    // The item to locate if the -s switch is used
    private static String locateItem;

    private static void usage() {
        System.out.println("ExampleInventoryRead [-h <env directory>]" +
                           "[-s <item to locate>]");
        System.exit(-1);
    }

    public static void main(String args[]) {
        ExampleInventoryRead eir = new ExampleInventoryRead();
        try {
            eir.run(args);
        } catch (DatabaseException dbe) {
            System.err.println("ExampleInventoryRead: " + dbe.toString());
            dbe.printStackTrace();
        } finally {
            myDbEnv.close();
        }
        System.out.println("All done.");
    }

    private void run(String args[])
        throws DatabaseException {
        // Parse the arguments list
        parseArgs(args);

        myDbEnv.setup(myDbEnvPath, // path to the environment home
                      true);       // is this environment read-only?

        // Open the data accessor. This is used to retrieve
        // persistent objects.
        da = new DataAccessor(myDbEnv.getEntityStore());

        // If a item to locate is provided on the command line,
        // show just the inventory items using the provided name.
        // Otherwise, show everything in the inventory.
        if (locateItem != null) {
            showItem();
        } else {
            showAllInventory();
```

```
        }
    }
```

The first method that we provide is used to show inventory items related to a given inventory name. This method is called only if an inventory name is passed to `ExampleInventoryRead` via the `-s` option. Given the sample data that we provide with this example, each matching inventory name will result in the display of three inventory objects.

To display these objects we use the `Inventory` class' `inventoryByName` secondary index to retrieve an `EntityCursor`, and then we iterate over the resulting objects using the cursor.

Notice that this method calls `displayInventoryRecord()` to display each individual object. We show this method a little later in the example.

```
    // Shows all the inventory items that exist for a given
    // inventory name.
    private void showItem() throws DatabaseException {

        // Use the inventory name secondary key to retrieve
        // these objects.
        EntityCursor<Inventory> items =
            da.inventoryByName.subIndex(locateItem).entities();
        try {
            for (Inventory item : items) {
                displayInventoryRecord(item);
            }
        } finally {
            items.close();
        }
    }
```

Next we implement `showAllInventory()`, which shows all of the `Inventory` objects in the store. To do this, we obtain an `EntityCursor` from the `Inventory` class' primary index and, again, we iterate using that cursor.

```
    // Displays all the inventory items in the store
    private void showAllInventory()
        throws DatabaseException {

        // Get a cursor that will walk every
        // inventory object in the store.
        EntityCursor<Inventory> items =
            da.inventoryBySku.entities();

        try {
            for (Inventory item : items) {
                displayInventoryRecord(item);
            }
        } finally {
```

```
            items.close();
        }
    }
```

Now we implement `displayInventoryRecord()`. This uses the getter methods on the `Inventory` class to obtain the information that we want to display. The only thing interesting about this method is that we obtain `Vendor` objects within. The vendor objects are retrieved `Vendor` objects using their primary index. We get the key for the retrieval from the `Inventory` object that we are displaying at the time.

```
    private void displayInventoryRecord(Inventory theInventory)
            throws DatabaseException {

            System.out.println(theInventory.getSku() + ":");
            System.out.println("\t " + theInventory.getItemName());
            System.out.println("\t " + theInventory.getCategory());
            System.out.println("\t " + theInventory.getVendor());
            System.out.println("\t\tNumber in stock: " +
                theInventory.getVendorInventory());
            System.out.println("\t\tPrice per unit:  " +
                theInventory.getVendorPrice());
            System.out.println("\t\tContact: ");

            Vendor theVendor =
                    da.vendorByName.get(theInventory.getVendor());
            assert theVendor != null;

            System.out.println("\t\t " + theVendor.getAddress());
            System.out.println("\t\t " + theVendor.getCity() + ", " +
                theVendor.getState() + " " + theVendor.getZipcode());
            System.out.println("\t\t Business Phone: " +
                theVendor.getBusinessPhoneNumber());
            System.out.println("\t\t Sales Rep: " +
                                theVendor.getRepName());
            System.out.println("\t\t            " +
                theVendor.getRepPhoneNumber());
    }
```

The last remaining parts of the example are used to parse the command line. This is not very interesting for our purposes here, but we show it anyway for the sake of completeness.

```
    protected ExampleInventoryRead() {}

    private static void parseArgs(String args[]) {
        for(int i = 0; i < args.length; ++i) {
            if (args[i].startsWith("-")) {
                switch(args[i].charAt(1)) {
                    case 'h':
                        myDbEnvPath = new File(args[++i]);
                    break;
```

```
                    case 's':
                        locateItem = args[++i];
                    break;
                    default:
                        usage();
                }
            }
        }
    }
}
```

This completes our example application. Be aware that this example and supporting classes can be found in your JE distribution in the following location:

JE_HOME/examples/persist/gettingStarted

where *JE_HOME* is the location where you placed your JE distribution.

Chapter 6. Other Operations

While we are done with our example application, there are several common activities that we have not yet explained: deleting entity objects and replacing them. This are fairly simple tasks, so we cover them briefly here.

Deleting Entity Objects

The simplest way to remove an object from your entity store is to delete it by its primary index. For example, using the `DataAccessor` class that we created earlier in this document (see DataAccessor.class (page 19)), you can delete the `Inventory` object with SKU `AlmofruiPPCLz8` as follows:

```
Transaction txn = myDbEnv.getEnv().beginTransaction(null, null);
try {
    da.inventoryBySku.delete(txn, "AlmofruiPPCLz8");
    txn.commit();
} catch (Exception e) {
    txn.abort();
    System.out.println("Aborted txn: " + e.toString());
    e.printStackTrace();
}
```

You can also delete objects by their secondary keys. When you do this, all objects related to the secondary key are deleted, unless the key is a foreign object.

For example, the following deletes all `Inventory` objects that use the product name `Almonds`:

```
Transaction txn = myDbEnv.getEnv().beginTransaction(null, null);
try {
    da.inventoryByName.delete(txn, "Almonds");
    txn.commit();
} catch (Exception e) {
    txn.abort();
    System.out.println("Aborted txn: " + e.toString());
    e.printStackTrace();
}
```

Finally, if you are indexing by foreign key, then the results of deleting the key is determined by the foreign key constraint that you have set for the index. See Foreign Key Constraints (page 15) for more information.

Replacing Entity Objects

To modify a stored entity object, retrieve it, update it, then put it back to the entity store:

```
Transaction txn = myDbEnv.getEnv().beginTransaction(null, null);
try {
```

```
    Inventory iv = da.inventoryBySku.get(txn, "AlmofruiPPCLz8",
                   LockMode.DEFAULT);
    iv.setVendorPrice(1.45);
    da.inventoryBySku.put(txn, iv);
    txn.commit();
} catch (Exception e) {
    txn.abort();
    System.out.println("Aborted txn: " + e.toString());
    e.printStackTrace();
}
```

Note that if you modify the object's primary key, then the object is stored as a new object in the entity store rather than replacing the existing object:

```
// Results in two objects in the store. One with SKU
// 'AlmofruiPPCLz8' and the other with SKU  'my new sku'.
Transaction txn = myDbEnv.getEnv().beginTransaction(null, null);
try {
    Inventory iv = da.inventoryBySku.get(txn, "AlmofruiPPCLz8",
                   LockMode.DEFAULT);
    iv.setSku("my new sku");
    da.inventoryBySku.put(txn, iv);
    txn.commit();
} catch (Exception e) {
    txn.abort();
        System.out.println("Aborted txn: " + e.toString());
            e.printStackTrace();
}
```

Similarly, if you modify a secondary key for the object, the object will subsequently be accessible by that new key, not by the old one.

```
// Object 'AlmofruiPPCLz8' can now be looked up using "Almond Nuts"
// instead of the original value, "Almonds".
Transaction txn = myDbEnv.getEnv().beginTransaction(null, null);
try {
    Inventory iv = da.inventoryBySku.get(txn, "AlmofruiPPCLz8",
                    LockMode.DEFAULT);
    iv.setItemName("Almond Nuts");
    da.inventoryBySku.put(txn, iv);
    txn.commit();
} catch (Exception e) {
    txn.abort();
    System.out.println("Aborted txn: " + e.toString());
    e.printStackTrace();
}
```

Finally, if you are iterating over a collection of objects using an `EntityCursor`, you can update each object in turn using `EntityCursor.update()`. Note, however, that you must

be iterating using a `PrimaryIndex`; this operation is not allowed if you are using a `SecondaryIndex`.

For example, the following iterates over every `Inventory` object in the entity store, and it changes them all so that they have a vendor price of `1.45`.

```
Transaction txn = myDbEnv.getEnv().beginTransaction(null, null);
EntityCursor<Inventory> items =
            da.inventoryBySku.entities(txn, null);
try {
    for (Inventory item : items) {
        item.setVendorPrice(1.45);
        items.update(item);
    }
    items.close();
    txn.commit();
} catch (Exception e) {
    items.close();
    txn.abort();
    System.out.println("Aborted txn: " + e.toString());
    e.printStackTrace();
}
```