Oracle Berkeley DB

Getting Started with Berkeley DB for Java

Release 4.8

Legal Notice

Published 4/12/2010

Table of Contents

Preface

Welcome to Berkeley DB (DB). This document introduces DB, version 4.8. It is intended to provide a rapid introduction to the DB API set and related concepts. The goal of this document is to provide you with an efficient mechanism with which you can evaluate DB against your project's technical requirements. As such, this document is intended for Java developers and senior software architects who are looking for an in-process data management solution. No prior experience with Berkeley DB is expected or required.

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 `Database()` constructor returns a `Database` class object."

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

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

```
import com.sleepycat.db.DatabaseConfig;

...

// Allow the database to be created.
DatabaseConfig myDbConfig = new DatabaseConfig();
myDbConfig.setAllowCreate(true);
```

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:

```
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;

...

// Allow the database to be created.
DatabaseConfig myDbConfig = new DatabaseConfig();
myDbConfig.setAllowCreate(true);
Database myDb = new Database("mydb.db", null, myDbConfig);
```

Finally, notes of 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 DB application:

- Getting Started with Transaction Processing for Java [http://www.oracle.com/technology/documentation/berkeley-db/db/gsg_txn/JAVA/BerkeleyDB-Core-JAVA-Txn.pdf]
- Berkeley DB Getting Started with Replicated Applications for Java [http://www.oracle.com/technology/documentation/berkeley-db/db/gsg_db_rep/JAVA/Replication_JAVA_GSG.pdf]
- Berkeley DB Programmer's Reference Guide [http://www.oracle.com/technology/documentation/berkeley-db/db/programmer_reference/BDB_Prog_Reference.pdf]
- Berkeley DB Javadoc [http://www.oracle.com/technology/documentation/berkeley-db/db/java/index.html]
- Berkeley DB Collections Tutorial [http://www.oracle.com/technology/documentation/berkeley-db/db/collections/tutorial/BerkeleyDB-Java-Collections.pdf]

Chapter 1. Introduction to Berkeley DB

Welcome to Berkeley DB (DB). DB is a general-purpose embedded database engine that is capable of providing a wealth of data management services. It is designed from the ground up for high-throughput applications requiring in-process, bullet-proof management of mission-critical data. DB can gracefully scale from managing a few bytes to terabytes of data. For the most part, DB is limited only by your system's available physical resources.

You use DB through a series of programming APIs which give you the ability to read and write your data, manage your database(s), and perform other more advanced activities such as managing transactions. The Java APIs that you use to interact with DB come in two basic flavors. The first is a high-level API that allows you to make Java classes persistent. The second is a lower-level API which provides additional flexibility when interacting with DB databases.

For long-time users of DB, the lower-level API is the traditional API that you are probably accustomed to using.

Because DB is an embedded database engine, it is extremely fast. You compile and link it into your application in the same way as you would any third-party library. This means that DB runs in the same process space as does your application, allowing you to avoid the high cost of interprocess communications incurred by stand-alone database servers.

To further improve performance, DB offers an in-memory cache designed to provide rapid access to your most frequently used data. Once configured, cache usage is transparent. It requires very little attention on the part of the application developer.

Beyond raw speed, DB is also extremely configurable. It provides several different ways of organizing your data in its databases. Known as *access methods*, each such data organization mechanism provides different characteristics that are appropriate for different data management profiles. (Note that this manual focuses almost entirely on the BTree access method as this is the access method used by the vast majority of DB applications).

To further improve its configurability, DB offers many different subsystems, each of which can be used to extend DB's capabilities. For example, many applications require write-protection of their data so as to ensure that data is never left in an inconsistent state for any reason (such as software bugs or hardware failures). For those applications, a transaction subsystem can be enabled and used to transactional-protect database writes.

The list of operating systems on which DB is available is too long to detail here. Suffice to say that it is available on all major commercial operating systems, as well as on many embedded platforms.

Finally, DB is available in a wealth of programming languages. DB is officially supported in C, C++, and Java, but the library is also available in many other languages, especially scripting languages such as Perl and Python.

Before going any further, it is important to mention that DB is not a relational database (although you could use it to build a relational database). Out of the box, DB does not provide higher-level features such as triggers, or a high-level query language such as SQL. Instead, DB provides just those minimal APIs required to store and retrieve your data as efficiently as possible.

About This Manual

This manual introduces DB. As such, this book does not examine intermediate or advanced features such as threaded library usage or transactional usage. Instead, this manual provides a step-by-step introduction to DB's basic concepts and library usage.

Specifically, this manual introduces the high-level Java API (the DPL), as well as the "base" Java API that the DPL relies upon. Regardless of the API set that you choose to use, there are a series of concepts and APIs that are common across the product. This manual starts by providing a high-level examination of DB. It then describes the APIs you use regardless of the API set that you choose to use. It then provides information on using the Direct Persistence Layer (DPL) API, followed by information on using the more extensive "base" API.

Examples are given throughout this book that are designed to illustrate API usage. At the end of each chapter or section in this book, a complete example is given that is designed to reinforce the concepts covered in that chapter or section. In addition to being presented in this book, these final programs are also available in the DB software distribution. You can find them in

```
DB_INSTALL/examples_java/db/GettingStarted
```

where *DB_INSTALL* is the location where you placed your DB distribution.

This book uses the Java programming languages for its examples. Note that versions of this book exist for the C and C++ languages as well.

Berkeley DB Concepts

Before continuing, it is useful to describe some of the concepts you will encounter when building a DB application.

The concepts that you will encounter depend upon the actual API that you are using. Some of these concepts are common to both APIs, and so we present those first. Others are only interesting if you use the DPL, while others apply only to the base API. We present each of these in turn.

Environments

Environments are required for applications built using the DPL. They are optional, but very commonly used, for applications built using the base API. Therefore, it is worthwhile to begin with them.

An *environment* is essentially an encapsulation of one or more databases. You open an environment and then you open databases in that environment. When you do so, the databases are created/located in a location relative to the environment's home directory.

Environments offer a great many features that a stand-alone DB database cannot offer:

- Multi-database files.

It is possible in DB to contain multiple databases in a single physical file on disk. This is desirable for those application that open more than a few handful of databases. However, in order to have more than one database contained in a single physical file, your application *must* use an environment.

- Multi-thread and multi-process support

  When you use an environment, resources such as the in-memory cache and locks can be shared by all of the databases opened in the environment. The environment allows you to enable subsystems that are designed to allow multiple threads and/or processes to access DB databases. For example, you use an environment to enable the concurrent data store (CDS), the locking subsystem, and/or the shared memory buffer pool.

- Transactional processing

  DB offers a transactional subsystem that allows for full ACID-protection of your database writes. You use environments to enable the transactional subsystem, and then subsequently to obtain transaction IDs.

- High availability (replication) support

  DB offers a replication subsystem that enables single-master database replication with multiple read-only copies of the replicated data. You use environments to enable and then manage this subsystem.

- Logging subsystem

  DB offers write-ahead logging for applications that want to obtain a high-degree of recoverability in the face of an application or system crash. Once enabled, the logging subsystem allows the application to perform two kinds of recovery ("normal" and "catastrophic") through the use of the information contained in the log files.

For more information on these topics, see the *Berkeley DB Getting Started with Transaction Processing* guide and the *Berkeley DB Getting Started with Replicated Applications* guide.

Key-Data Pairs

DB stores and retrieves data using *key-data pairs*. The *data* portion of this is the data that you have decided to store in DB for future retrieval. The *key* is the information that you want to use to look up your stored data once it has been placed inside a DB database.

For example, if you were building a database that contained employee information, then the *data* portion is all of the information that you want to store about the employees: name, address, phone numbers, physical location, their manager, and so forth.

The *key*, however, is the way that you look up any given employee. You can have more than one key if you wish, but every record in your database must have a primary key. If you are using the DPL, then this key must be unique; that is, it must not be used multiple times in the database. However, if you are using the base API, then this requirement is relaxed. See Duplicate Data (page 5) for more information.

For example, in the case of an employee database, you would probably use something like the employee identification number as the primary key as this uniquely identifies a given employee.

You can optionally also have secondary keys that represent indexes into your database. These keys do not have to be unique to a given record; in fact, they often are not. For example, you might set up the employee's manager's name as a secondary key so that it is easy to locate all the employee's that work for a given manager.

Storing Data

How you manage your stored information differs significantly, depending on which API you are using. Both APIs ultimately are doing the same thing, but the DPL hides a lot of the details from you.

Storing Data in the DPL

The DPL is used to store Java objects in an underlying series of databases. These databases are accessed using an `EntityStore` class object.

To use the DPL, you must decorate the classes you want to store with Java annotations that identify them as either an *entity class* or a *persistent class*.

Entity classes are classes that have a primary key, and optionally one or more secondary keys. 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.

The primary key for an object is obtained from one of the class' data members. You identify which data member to use as the primary key using the `@PrimaryKey` 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.

Also, 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`.

Storing Data using the Base API

When you are not using the DPL, both record keys and record data must be byte arrays and are passed to and returned from DB using `DatabaseEntry` instances. `DatabaseEntry` only supports storage of Java byte arrays. Complex objects must be marshaled using either Java serialization, or more efficiently with the bind APIs provided with DB

Database records and `byte` array conversion are described in Database Records (page 68).

You store records in a `Database` by calling one of the put methods on a `Database` handle. DB automatically determines the record's proper placement in the database's internal B-Tree using whatever key and data comparison functions that are available to it.

You can also retrieve, or get, records using the `Database` handle. Gets are performed by providing the key (and sometimes also the data) of the record that you want to retrieve.

You can also use cursors for database puts and gets. Cursors are essentially a mechanism by which you can iterate over the records in the database. Like databases and database environments, cursors must be opened and closed. Cursors are managed using the `Cursor` class.

Databases are described in Databases (page 58). Cursors are described in Using Cursors (page 96).

Duplicate Data

If you are using the base API, then at creation time databases can be configured to allow duplicate data. Remember that DB database records consist of a key/data pair. *Duplicate data*, then, occurs when two or more records have identical keys, but different data. By default, a `Database` does not allow duplicate data.

If your `Database` contains duplicate data, then a simple database get based only on a key returns just the first record that uses that key. To access all duplicate records for that key, you must use a cursor.

If you are using the DPL, then you can duplicate date using secondary keys, but not by using the primary key. For more information, see Retrieving Multiple Objects (page 34).

Replacing and Deleting Entries

If you are using the DPL, then replacing a stored entity object simply consists of retrieving it, updating it, then storing it again. To delete the object, use the `delete()` method that is available on either its primary or secondary keys. If you use the `delete()` method available on the secondary key, then all objects referenced by that key are also deleted. See Deleting Entity Objects (page 39) for more information.

If you are using the base API, then how you replace database records depends on whether duplicate data is allowed in the database.

If duplicate data is not allowed in the database, then simply calling `Database.put()` with the appropriate key will cause any existing record to be updated with the new data. Similarly, you can delete a record by providing the appropriate key to the `Database.delete()` method.

If duplicate data is allowed in the database, then you must position a cursor to the record that you want to update, and then perform the put operation using the cursor.

To delete records using the base API, you can use either `Database.delete()` or `Cursor.delete()`. If duplicate data is not allowed in your database, then these two method behave identically. However, if duplicates are allowed in the database, then `Database.delete()` deletes every record that uses the provided key, while `Cursor.delete()` deletes just the record at which the cursor is currently positioned.

Secondary Keys

Secondary keys provide an alternative way to locate information stored in DB, beyond that which is provided by the primary key. Frequently secondary keys refer to more than one record in the database. In this way, you can find all the cars that are green (if you are maintaining an automotive database) or all the people with brown eyes (if you are maintaining a database about people). In other words, secondary keys represent a index into your data.

How you create and maintain secondary keys differs significantly, depending on whether you are using the DPL or the base API.

Using Secondaries with the DPL

Under the DPL, you declare a particular field to be a secondary key by using the `@SecondaryKey` annotation. When you do this, you must declare what kind of an index you are creating. For example, you can declare a secondary key to be part of a `ONE_TO_ONE` index, in which case the key is unique to the object. Or you could declare the key to be `MANY_TO_ONE`, in which case the key can be used for multiple objects in the data store.

Once you have identified secondary keys for a class, you can access those keys by using the `EntityStore.getSecondaryIndex()` method.

For more information, see Declaring Secondary Indexes (page 24).

Using Secondaries with the Base API.

When you are using the base API, you create and maintain secondary keys using a special type of a database, called a *secondary database*. When you are using secondary databases, the database that holds the data you are indexing is called the *primary database*.

You create a secondary database by opening it and associating it with an existing primary database. You must also provide a class that generates the secondary's keys (that is, the index) from primary records. Whenever a record in the primary database is added or changed, DB uses this class to determine what the secondary key should be.

When a primary record is created, modified, or deleted, DB automatically updates the secondary database(s) for you as is appropriate for the operation performed on the primary.

You manage secondary databases using the `SecondaryDatabase` class. You identify how to create keys for your secondary databases by supplying an instance of a class that implements the `SecondaryKeyCreator` interface.

Secondary databases are described in Secondary Databases (page 113).

Which API Should You Use?

Of the two APIs that DB makes available to you, we recommend that 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 cannot use the DPL.

Further, if you are porting an application between the C or C++ versions of DB and the Java version of this API, then you should not use the DPL as the base API is a much closer match to the other languages available for use with DB.

Additionally, if your application uses a highly dynamic schema, then the DPL is probably a poor choice for your application, although the use of Java annotations can make the DPL work a little better for you in this situation.

Access Methods

While this manual will focus primarily on the BTree access method, it is still useful to briefly describe all of the access methods that DB makes available.

If you are using the DPL, be aware that it only supports the BTree access method. For that reason, you can skip this section.

Note that an access method can be selected only when the database is created. Once selected, actual API usage is generally identical across all access methods. That is, while some exceptions exist, mechanically you interact with the library in the same way regardless of which access method you have selected.

The access method that you should choose is gated first by what you want to use as a key, and then secondly by the performance that you see for a given access method.

The following are the available access methods:

Access Method	Description
BTree	Data is stored in a sorted, balanced tree structure. Both the key and the data for BTree records can be arbitrarily complex. That is, they can contain single values such as an integer or a string, or complex types such as a structure. Also, although not the default behavior, it is possible for two records to use keys that compare as equals. When this occurs, the records are considered to be duplicates of one another.
Hash	Data is stored in an extended linear hash table. Like BTree, the key and the data used for Hash records can be of arbitrarily complex data. Also, like BTree, duplicate records are optionally supported.
Queue	Data is stored in a queue as fixed-length records. Each record uses a logical record number as its key. This access method is designed for fast inserts at the tail of the queue, and it has a special operation that deletes and returns a record from the head of the queue. This access method is unusual in that it provides record level locking. This can provide beneficial performance improvements in applications requiring concurrent access to the queue.
Recno	Data is stored in either fixed or variable-length records. Like Queue, Recno records use logical record numbers as keys.

Selecting Access Methods

To select an access method, you should first consider what you want to use as a key for you database records. If you want to use arbitrary data (even strings), then you should use either BTree or Hash. If you want to use logical record numbers (essentially integers) then you should use Queue or Recno.

Once you have made this decision, you must choose between either BTree or Hash, or Queue or Recno. This decision is described next.

Choosing between BTree and Hash

For small working datasets that fit entirely in memory, there is no difference between BTree and Hash. Both will perform just as well as the other. In this situation, you might just as well use BTree, if for no other reason than the majority of DB applications use BTree.

Note that the main concern here is your working dataset, not your entire dataset. Many applications maintain large amounts of information but only need to access some small portion of that data with any frequency. So what you want to consider is the data that you will routinely use, not the sum total of all the data managed by your application.

However, as your working dataset grows to the point where you cannot fit it all into memory, then you need to take more care when choosing your access method. Specifically, choose:

- BTree if your keys have some locality of reference. That is, if they sort well and you can expect that a query for a given key will likely be followed by a query for one of its neighbors.

- Hash if your dataset is extremely large. For any given access method, DB must maintain a certain amount of internal information. However, the amount of information that DB must maintain for BTree is much greater than for Hash. The result is that as your dataset grows, this internal information can dominate the cache to the point where there is relatively little space left for application data. As a result, BTree can be forced to perform disk I/O much more frequently than would Hash given the same amount of data.

  Moreover, if your dataset becomes so large that DB will almost certainly have to perform disk I/O to satisfy a random request, then Hash will definitely out perform BTree because it has fewer internal records to search through than does BTree.

Choosing between Queue and Recno

Queue or Recno are used when the application wants to use logical record numbers for the primary database key. Logical record numbers are essentially integers that uniquely identify the database record. They can be either mutable or fixed, where a mutable record number is one that might change as database records are stored or deleted. Fixed logical record numbers never change regardless of what database operations are performed.

When deciding between Queue and Recno, choose:

- Queue if your application requires high degrees of concurrency. Queue provides record-level locking (as opposed to the page-level locking that the other access methods use), and this can result in significantly faster throughput for highly concurrent applications.

  Note, however, that Queue provides support only for fixed length records. So if the size of the data that you want to store varies widely from record to record, you should probably choose an access method other than Queue.

- Recno if you want mutable record numbers. Queue is only capable of providing fixed record numbers. Also, Recno provides support for databases whose permanent storage is a flat text file. This is useful for applications looking for fast, temporary storage while the data is being read or modified.

Database Limits and Portability

Berkeley DB provides support for managing everything from very small databases that fit entirely in memory, to extremely large databases holding millions of records and terabytes of data. DB databases can store up to 256 terabytes of data. Individual record keys or record data can store up to 4 gigabytes of data.

DB's databases store data in a binary format that is portable across platforms, even of differing endian-ness. Be aware, however, that portability aside, some performance issues can crop up in the event that you are using little endian architecture. See Setting Comparison Functions (page 141) for more information.

Also, DB's databases and data structures are designed for concurrent access — they are thread-safe, and they share well across multiple processes. That said, in order to allow multiple processes to share databases and the cache, DB makes use of mechanisms that do not work well on network-shared drives (NFS or Windows networks shares, for example). For this reason, you cannot place your DB databases and environments on network-mounted drives.

Exception Handling

Before continuing, it is useful to spend a few moments on exception handling in DB with the java.

Most DB methods throw `DatabaseException` in the event of a serious error. So your DB code must either catch this exception or declare it to be throwable. Be aware that `DatabaseException` extends `java.lang.Exception`. For example:

```
import com.sleepycat.db.DatabaseException;

    ...
try
{
    // DB and other code goes here
}
catch(DatabaseException e)
{
```

```
  // DB error handling goes here
}
```

You can obtain the DB error number for a `DatabaseException` by using `DatabaseException.getErrno()`. You can also obtain any error message associated with that error using `DatabaseException.getMessage()`.

Error Returns

In addition to exceptions, the DB interfaces always return a value of 0 on success. If the operation does not succeed for any reason, the return value will be non-zero.

If a system error occurred (for example, DB ran out of disk space, or permission to access a file was denied, or an illegal argument was specified to one of the interfaces), DB returns an `errno` value. All of the possible values of `errno` are greater than 0.

If the operation did not fail due to a system error, but was not successful either, DB returns a special error value. For example, if you tried to retrieve data from the database and the record for which you are searching does not exist, DB would return `DB_NOTFOUND`, a special error value that means the requested key does not appear in the database. All of the possible special error values are less than 0.

Getting and Using DB

You can obtain DB by visiting the Berkeley DB download page: http://www.oracle.com/technology/software/products/berkeley-db/db/index.html.

To install DB, untar or unzip the distribution to the directory of your choice. You will then need to build the product binaries. For information on building DB, see *DB_INSTALL*`/docs/index.html`, where *DB_INSTALL* is the directory where you unpacked DB. On that page, you will find links to platform-specific build instructions.

That page also contains links to more documentation for DB. In particular, you will find links for the *Berkeley DB Programmer's Reference Guide* as well as the API reference documentation.

Chapter 2. Database Environments

Environments are optional, but very commonly used, for Berkeley DB applications built using the base API. If you are using the DPL, then environments are required.

Database environments encapsulate one or more databases. This encapsulation provides your threads with efficient access to your databases by allowing a single in-memory cache to be used for each of the databases contained in the environment. This encapsulation also allows you to group operations performed against multiple databases inside a single transactions (see the *Berkeley DB Java Edition Getting Started with Transaction Processing* guide for more information).

Most commonly you use database environments to create and open databases (you close individual databases using the individual database handles). You can also use environments to delete and rename databases. For transactional applications, you use the environment to start transactions. For non-transactional applications, you use the environment to sync your in-memory cache to disk.

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:

```
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
}
```

```
package db.gettingStarted;

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

import java.io.File;
import java.io.FileNotFoundException;

...

// 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
} catch (FileNotFoundException fnfe) {
// 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.

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 *Berkeley DB Java Edition Getting Started with Transaction Processing* guide. For information on syncs, see the *Getting Started with Transaction Processing for Java* guide.

```
import com.sleepycat.db.DatabaseException;

import com.sleepycat.db.Environment;

...

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

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

Closing the last environment handle in your application causes all internal data structures to be released. If there are any opened databases or stores, then DB will complain before closing them as well. At this time, any open cursors are also closed, and any on-going transactions are aborted.

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`.

- `EnvironmentConfig.setTransactional()`

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

For example:

```
package db.gettingStarted;

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

import java.io.File;
```

```
import java.io.FileNotFoundException;

...

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);
} catch (FileNotFoundException fnfe) {
    System.err.println(fnfe.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 DB cache. See Selecting the Cache Size (page 138) for more information.

- `setCacheSize()`

  Determines the total amount of memory available to the database cache. See Selecting the Cache Size (page 138) 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 *Getting Started with Transaction Processing for Java* guide for more information.

- `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. See the *Getting Started with Transaction Processing for Java* guide for more information.

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 db.gettingStarted;

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

import java.io.File;
import java.io.FileNotFoundException;


...

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
} catch (FileNotFoundException fnfe) {
    // Exception handling goes here
}
```

Part I. Programming with the Direct Persistence Layer

This section discusses how to build an application using the DPL. The DPL is ideally suited for those applications that want a mechanism for storing and managing Java class objects in a DB database. Note that the DPL is best suited for applications that work with classes with a relatively static schema.

Also, the DPL requires Java 1.5.

If you want to use Java 1.4 for your DB application, or if you are porting an application from the Berkeley DB API, then you probably want to use the base API instead of the DPL. For information on using the base API, see Programming with the Base API (page 57).

Chapter 3. Direct Persistence Layer First Steps

This chapter guides you through the first few steps required to use the DPL with your application. These steps include:

1. Opening your environment as was described in Opening Database Environments (page 11).
2. Opening your entity store.
3. Identifying the classes that you want to store in DB as either a `persistent` class or an `entity`.

Once you have done these things, you can write your classes to the DB databases, read them back from the databases, delete them from the databases, and so forth. These activities are described in the chapters that follow in this part of this manual.

Entity Stores

Entity stores are the basic unit of storage that you use with the DPL. That is, it is a unit of encapsulation for the classes that you want to store in DB. Under the hood it actually interacts with DB databases, but the DPL provides a layer of abstraction from the underlying DB APIs. The store, therefore, provides a simplified mechanism by which you read and write your stored classes. By using a store, you have access to your classes that is more simplified than if you were interacting with databases directly, but this simplified access comes at the cost of reduced flexibility.

Entity stores have configurations in the same way that environments have configurations. You can use a `StoreConfig` object to identify store properties. Among these are methods that allow you to declare whether:

- the store can be created if it does not exist at the time it is opened. Use the `StoreConfig.setAllowCreate()` method to set this.
- the store is read-only. Use the `StoreConfig.setReadOnly()` method to set this.
- the store supports transactions. Use the `StoreConfig.setTransactional()` method to set this.

  Writing DB transactional applications is described in the *Berkeley DB Java Edition Getting Started with Transaction Processing* guide.

`EntityStore` objects also provide methods for retrieving information about the store, such as:

- the store's name. Use the `EntityStore.getStoreName()` method to retrieve this.
- a handle to the environment in which the store is opened. Use the `EntityStore.getEnvironment` method to retrieve this handle.

You can also use the `EntityStore` to retrieve all the primary and secondary indexes related to a given type of entity object contained in the store. See Working with Indices (page 22) for more information.

Opening and Closing Environments and Stores

As described in Database Environments (page 11), an *environment* is a unit of encapsulation for DB databases. It also provides a handle by which activities common across the databases can be managed.

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

For example, the following code fragment configures both the environment and the entity store such that they can be created if they do not exist. Both the environment and the entity store are then opened.

```
package persist.gettingStarted;

import java.io.File;
import java.io.FileNotFoundException;

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

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

...

private Environment myEnv;
private EntityStore store;

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

    myEnvConfig.setAllowCreate(!readOnly);
    storeConfig.setAllowCreate(!readOnly);

    try {
        // Open the environment and entity store
        myEnv = new Environment(envHome, myEnvConfig);
        store = new EntityStore(myEnv, "EntityStore", storeConfig);
    } catch (FileNotFoundException fnfe) {
        System.err.println(fnfe.toString());
        System.exit(-1);
    }
} catch(DatabaseException dbe) {
    System.err.println("Error opening environment and store: " +
                       dbe.toString());
```

```
    System.exit(-1);
}
```

As always, before you exit your program you should close both your store and your environment. Be sure to close your store before you close your environment.

```
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 environment.
        myEnv.close();
    } catch(DatabaseException dbe) {
        System.err.println("Error closing MyDbEnv: " +
                           dbe.toString());
        System.exit(-1);
    }
}
```

Persistent Objects

When using the DPL, you store data in the underlying DB databases by making objects *persistent*. You do this using Java annotations that both identify the type of persistent object you are declaring, as well as the primary and secondary indices.

The following are the annotations you will use with your DPL persistent classes:

Annotation	Description
@Entity	Declares an entity class; that is, a class with a primary index and optionally one or more indices.
@Persistent	Declares a persistent class; that is, a class used by an entity class. They do not have indices but instead are are stored or retrieved when an entity class makes direct use of them.
@PrimaryKey	Declares a specific data member in an entity class to be the primary key for that object. This annotation must be used one and only one time for every entity class.

Annotation	Description
@SecondaryKey	Declares a specific data member in an entity class to be a secondary key for that object. This annotation is optional, and can be used multiple times for an entity class.

For example, the following is declared to be an entity class:

```
package persist.gettingStarted;

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

@Entity
public class ExampleEntity {

    // The primary key must be unique in the database.
    @PrimaryKey
    private String aPrimaryKey;

    @SecondaryKey(relate=MANY_TO_ONE)
    private String aSecondaryKey;

    ...

    // The remainder of the class' implementation is purposefully
    // omitted in the interest of brevity.

    ...
}
```

We discuss primary and secondary keys in more detail in Working with Indices (page 22).

Saving a Retrieving Data

All data stored using the DPL has one primary index and zero or more secondary indices associated with it. (Sometimes these are referred to as the primary and secondary *keys*.) So to store data under the DPL, you must:

1. Declare a class to be an entity class.
2. Identify the features on the class which represent indexed material.
3. Retrieve the store's primary index for a given class using the `EntityStore.getPrimaryIndex()` method.
4. Put class objects to the store using the `PrimaryIndex.put()` method.

In order to retrieve an object from the store, you use the index that is most convenient for your purpose. This may be the primary index, or it may be some other secondary index that you declared on your entity class.

You obtain a primary index in the same was as when you put the object to the store: using `EntityStore.getPrimaryIndex()`. You can get a secondary index for the store using the `EntityStore.getSecondaryIndex()` method. Note that `getSecondaryIndex()` requires you to provide a `PrimaryIndex` class instance when you call it, so a class's primary index is always required when retrieving objects from an entity store.

Usually all of the activity surrounding saving and retrieving data is organized within a class or classes specialized to that purpose. We describe the construction of these data accessor classes in SimpleDA.class (page 28). But before you perform any entity store activity, you need to understand indexes. We therefore describe them in the next chapter.

Chapter 4. Working with Indices

All entity classes stored in DB using the DPL must have a primary index, or key, identified for them. All such classes may also have one or more secondary keys declared for them. This chapter describes primary and secondary indexes in detail, and shows how to access the indexes created for a given entity class.

One way to organize access to your primary and secondary indexes is to create a *data accessor* class. We show an implementation of a data accessor class in SimpleDA.class (page 28).

Accessing Indexes

In order to retrieve any object from an entity store, you must access at least the primary index for that object. Different entity classes stored in an entity store can have different primary indexes, but all entity classes must have a primary index declared for it. The primary index is just the default index used for the class. (That is, it is the data's primary *key* for the underlying database.)

Entity classes can optionally have secondary indexes declared for them. In order to access these secondary indexes, you must first access the primary index.

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 an `Inventory` class (we provide an implementation of this class in Inventory.java (page 43)). 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");
```

Creating Indexes

To create an index using the DPL, you use Java annotations to declare which feature on the class is used for the primary index, and which features (if any) are to be used as secondary indexes.

All entity classes stored in the DPL must have a primary index declared for it.

Entity classes can have zero or more secondary indexes declared for them. There is no limit on the number of secondary indexes that you can declare.

Declaring a Primary Indexes

You declare a primary key for an entity class by using the `@PrimaryKey` annotation. This annotation must appear immediately before the data member which represents the class's primary key. For example:

```
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;

    ...
```

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's 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. You must also provide a name for the sequence. For example: For example:

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

Declaring Secondary Indexes

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();
```

Chapter 5. Saving and Retrieving Objects

To store an object in an `EntityStore` you must annotate the class appropriately and then store it using `PrimaryIndex.put()`.

To retrieve and object from an `EntityStore` you use the `get()` method from either the `PrimaryIndex` or `SecondaryIndex`, whichever is most appropriate for your application.

In both cases, it simplifies things greatly if you create a data accessor class to organize your indexes.

In the next few sections we:

1. Create an entity class that is ready to be stored in an entity store. This class will have both a primary index (required) declared for it, as well as a secondary index (which is optional).

   See the next section for this implementation.

2. Create a data accessor class which is used to organize our data.

   See SimpleDA.class (page 28) for this implementation.

3. Create a simple class that is used to put objects to our entity store.

   See Placing Objects in an Entity Store (page 29) for this implementation.

4. Create another class that retrieves objects from our entity store.

   See Retrieving Objects from an Entity Store (page 32) for this implementation.

A Simple Entity Class

For clarity's sake, this entity class is a simple a class as we can write. It contains only two data members, both of which are set and retrieved by simple setter and getter methods. Beyond that, by design this class does not do anything or particular interest.

Its implementation is as follows:

```
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 SimpleEntityClass {

    // Primary key is pKey
    @PrimaryKey
```

```
    private String pKey;

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

    public void setPKey(String data) {
        pKey = data;
    }

    public void setSKey(String data) {
        sKey = data;
    }

    public String getPKey() {
        return pKey;
    }

    public String getSKey() {
        return sKey;
    }
}
```

SimpleDA.class

As mentioned above, we organize our primary and secondary indexes using a specialize data accessor class. The main reason for this class to exist is to provide convenient access to all the indexes in use for our entity class (see the previous section, A Simple Entity Class (page 27), for that implementation).

For a description on retrieving primary and secondary indexes under the DPL, see Working with Indices (page 22)

```
package persist.gettingStarted;

import java.io.File;

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

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

        // Primary key for SimpleEntityClass classes
        pIdx = store.getPrimaryIndex(
```

```
            String.class, SimpleEntityClass.class);

        // Secondary key for SimpleEntityClass classes
        // Last field in the getSecondaryIndex() method must be
        // the name of a class member; in this case, an
        // SimpleEntityClass.class data member.
        sIdx = store.getSecondaryIndex(
            pIdx, String.class, "sKey");
    }

    // Index Accessors
    PrimaryIndex<String,SimpleEntityClass> pIdx;
    SecondaryIndex<String,String,SimpleEntityClass> sIdx;
}
```

Placing Objects in an Entity Store

In order to place an object in a DPL entity store, you must:

1. Open the environment and store.

2. Instantiate the object.

3. Put the object to the store using the `put()` method for the object's primary index.

The following example uses the `SimpleDA` class that we show in SimpleDA.class (page 28) to put a `SimpleEntityClass` object (see A Simple Entity Class (page 27)) to the entity store.

To begin, we import the Java classes that our example needs. We also instantiate the private data members that we require.

```
package persist.gettingStarted;

import java.io.File;
import java.io.FileNotFoundException;

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

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

public class SimpleStorePut {

    private static File envHome = new File("./JEDB");

    private Environment envmnt;
    private EntityStore store;
    private SimpleDA sda;
```

Next we create a method that simply opens our database environment and entity store for us.

```
    // The setup() method opens the environment and store
     // for us.
     public void setup()
         throws DatabaseException {

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

         envConfig.setAllowCreate(true);
         storeConfig.setAllowCreate(true);

         try {
             // Open the environment and entity store
             envmnt = new Environment(envHome, envConfig);
             store = new EntityStore(envmnt, "EntityStore", storeConfig);
         } catch (FileNotFoundException fnfe) {
             System.err.println("setup(): " + fnfe.toString());
             System.exit(-1);
         }
     }
```

We also need a method to close our environment and store.

```
     // Close our environment and store.
     public void shutdown()
         throws DatabaseException {

         store.close();
         envmnt.close();
     }
```

Now we need to create a method to actually write objects to our store. This method creates a `SimpleDA` object (see SimpleDA.class (page 28) that we will use to access our indexes. Then we instantiate a serious of `SimpleEntityClass` (see A Simple Entity Class (page 27)) instances that we will place in our store. Finally, we use our primary index (obtained from the `SimpleDA` class instance) to actually place these objects in our store.

In Retrieving Objects from an Entity Store (page 32) we show a class that is used to retrieve these objects.

```
     // Populate the entity store
     private void run()
         throws DatabaseException {

         setup();

         // Open the data accessor. This is used to store
         // persistent objects.
         sda = new SimpleDA(store);
```

```
        // Instantiate and store some entity classes
        SimpleEntityClass sec1 = new SimpleEntityClass();
        SimpleEntityClass sec2 = new SimpleEntityClass();
        SimpleEntityClass sec3 = new SimpleEntityClass();
        SimpleEntityClass sec4 = new SimpleEntityClass();
        SimpleEntityClass sec5 = new SimpleEntityClass();

        sec1.setPKey("keyone");
        sec1.setSKey("skeyone");

        sec2.setPKey("keytwo");
        sec2.setSKey("skeyone");

        sec3.setPKey("keythree");
        sec3.setSKey("skeytwo");

        sec4.setPKey("keyfour");
        sec4.setSKey("skeythree");

        sec5.setPKey("keyfive");
        sec5.setSKey("skeyfour");

        sda.pIdx.put(sec1);
        sda.pIdx.put(sec2);
        sda.pIdx.put(sec3);
        sda.pIdx.put(sec4);
        sda.pIdx.put(sec5);

        shutdown();
    }
```

Finally, to complete our class, we need a `main()` method, which simply calls our `run()` method.

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

```
}
```

Retrieving Objects from an Entity Store

You retrieve objects placed in an entity store by using either the object's primary index, or the appropriate secondary index if it exists. The following application illustrates this by retrieving some of the objects that we placed in an entity store in the previous section.

To begin, we import the Java classes that our example needs. We also instantiate the private data members that we require.

```
package persist.gettingStarted;

import java.io.File;
import java.io.FileNotFoundException;

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

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

public class SimpleStoreGet {

    private static File envHome = new File("./JEDB");

    private Environment envmnt;
    private EntityStore store;
    private SimpleDA sda;
```

Next we create a method that simply opens our database environment and entity store for us.

```
    // The setup() method opens the environment and store
     // for us.
     public void setup()
         throws DatabaseException {

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

         envConfig.setAllowCreate(true);
         storeConfig.setAllowCreate(true);

         try {
             // Open the environment and entity store
             envmnt = new Environment(envHome, envConfig);
             store = new EntityStore(envmnt, "EntityStore", storeConfig);
         } catch (FileNotFoundException fnfe) {
```

```
            System.err.println("setup(): " + fnfe.toString());
            System.exit(-1);
        }
    }
```

We also need a method to close our environment and store.

```
    // Close our environment and store.
    public void shutdown()
        throws DatabaseException {

        store.close();
        envmnt.close();
    }
```

Now we retrieve a few objects. To do this, we instantiate a `SimpleDA` (see SimpleDA.class (page 28)) class that we use to access our primary and secondary indexes. Then we retrieve objects based on a primary or secondary index value. And finally, we display the retrieved objects.

```
    // Retrieve some SimpleEntityClass objects from the store.
    private void run()
        throws DatabaseException {

        setup();

        // Open the data accessor. This is used to store
        // persistent objects.
        sda = new SimpleDA(store);

        // Instantiate and store some entity classes
        SimpleEntityClass sec1 = sda.pIdx.get("keyone");
        SimpleEntityClass sec2 = sda.pIdx.get("keytwo");

        SimpleEntityClass sec4 = sda.sIdx.get("skeythree");

        System.out.println("sec1: " + sec1.getPKey());
        System.out.println("sec2: " + sec2.getPKey());
        System.out.println("sec4: " + sec4.getPKey());


        shutdown();
    }
```

Finally, to complete our class, we need a `main()` method, which simply calls our `run()` method.

```
    // main
    public static void main(String args[]) {
        SimpleStoreGet ssg = new SimpleStoreGet();
        try {
            ssg.run();
```

```
        } catch (DatabaseException dbe) {
            System.err.println("SimpleStoreGet: " + dbe.toString());
            dbe.printStackTrace();
        } catch (Exception e) {
            System.out.println("Exception: " + e.toString());
            e.printStackTrace();
        }
        System.out.println("All done.");
    }

}
```

Retrieving Multiple Objects

It is possible to iterate over every object referenced by a specific index. You may want to do this if, for example, you want to examine or modify every object accessible by a specific primary index.

In addition, some indexes result in the retrieval of multiple objects. For example, `MANY_TO_ONE` secondary indexes can result in more than one object for any given key (also known as *duplicate keys*). When this is the case, you must iterate over the resulting set of objects in order to examine each object in turn.

There are two ways to iterate over a collection of objects as returned by an index. One is to use a standard Java `Iterator`, which you obtain using an `EntityCursor`, which in turn you can obtain from a `PrimaryIndex`:

```
PrimaryIndex<String,SimpleEntityClass> pi =
    store.getPrimaryIndex(String.class, SimpleEntityClass.class);
EntityCursor<SimpleEntityClass> pi_cursor = pi.entities();
try {
    Iterator<SimpleEntityClass> i = pi_cursor.iterator();
    while (i.hasNext()) {
        // Do something here
    }
} finally {
    // Always close the cursor
    pi_cursor.close();
}
```

Alternatively, you can use a Java "foreach" statement to iterate over object set:

```
PrimaryIndex<String,SimpleEntityClass> pi =
    store.getPrimaryIndex(String.class, SimpleEntityClass.class);
EntityCursor<SimpleEntityClass> pi_cursor = pi.entities();
try {
    for (SimpleEntityClass seci : pi_cursor) {
        // do something with each object "seci"
    }
// Always make sure the cursor is closed when we are done with it.
```

```
} finally {
    sec_cursor.close();
}
```

Cursor Initialization

When a cursor is first opened, it is not positioned to any value; that is, it is not *initialized*. Most of the `EntityCursor` methods that move a cursor will initialize it to either the first or last object, depending on whether the operation is moving the cursor forward (all `next...` methods) or backwards (all `prev...`) methods.

You can also force a cursor, whether it is initialized or not, to return the first object by calling `EntityCursor.first()`. Similarly, you can force a return of the last object using `EntityCursor.last()`.

Operations that do not move the cursor (such as `EntityCursor.current()` or `EntityCursor.delete()` will throw an `IllegalStateException` when used on an uninitialized cursor.

Working with Duplicate Keys

If you have duplicate secondary keys, you can return an `EntityIndex` class object for them using `SecondaryIndex.subIndex()` Then, use that object's `entities()` method to obtain an `EntityCursor` instance.

For example:

```
PrimaryIndex<String,SimpleEntityClass> pi =
    store.getPrimaryIndex(String.class, SimpleEntityClass.class);

SecondaryIndex<String,String,SimpleEntityClass> si =
    store.getSecondaryIndex(pi, String.class, "sKey");

EntityCursor<SimpleEntityClass> sec_cursor =
    si.subIndex("skeyone").entities();

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

Note that if you are working with duplicate keys, you can control how cursor iteration works by using the following `EntityCursor` methods:

- `nextDup()`

Moves the cursor to the next object with the same key as the cursor is currently referencing. That is, this method returns the next duplicate object. If no such object exists, this method returns `null`.

- `prevDup()`

  Moves the cursor to the previous object with the same key as the cursor is currently referencing. That is, this method returns the previous duplicate object in the cursor's set of objects. If no such object exists, this method returns `null`.

- `nextNoDup()`

  Moves the cursor to the next object in the cursor's set that has a key which is different than the key that the cursor is currently referencing. That is, this method skips all duplicate objects and returns the next non-duplicate object in the cursor's set of objects. If no such object exists, this method returns `null`.

- `prevNoDup()`

  Moves the cursor to the previous object in the cursor's set that has a key which is different than the key that the cursor is currently referencing. That is, this method skips all duplicate objects and returns the previous non-duplicate object in the cursor's set of objects. If no such object exists, this method returns `null`.

For example:

```
PrimaryIndex<String,SimpleEntityClass> pi =
    store.getPrimaryIndex(String.class, SimpleEntityClass.class);

SecondaryIndex<String,String,SimpleEntityClass> si =
    store.getSecondaryIndex(pi, String.class, "sKey");

EntityCursor<SimpleEntityClass> sec_cursor =
    si.subIndex("skeyone").entities();

try {
    SimpleEntityClass sec;
    Iterator<SimpleEntityClass> i = sec_cursor.iterator();
    while (sec = i.nextNoDup() != null) {
        // Do something here
    }
// Always make sure the cursor is closed when we are done with it.
} finally {
    sec_cursor.close(); }
```

Key Ranges

You can restrict the scope of a cursor's movement by specifying a *range* when you create the cursor. The cursor can then never be positioned outside of the specified range.

When specifying a range, you indicate whether a range bound is *inclusive* or *exclusive* by providing a boolean value for each range. `true` indicates that the provided bound is inclusive, while `false` indicates that it is exclusive.

You provide this information when you call `PrimaryIndex.entities()` or `SecondaryIndex.entities()`. For example, suppose you had a class indexed by numerical information. Suppose further that you wanted to examine only those objects with indexed values of 100 - 199. Then (assuming the numerical information is the primary index), you can bound your cursor as follows:

```
EntityCursor<SomeEntityClass> cursor =
    primaryIndex.entities(100, true, 200, false);

try {
    for (SomeEntityClass sec : cursor {
        // Do something here to objects ranged from 100 to 199
    }
// Always make sure the cursor is closed when we are done with it.
} finally {
    cursor.close(); }
```

Join Cursors

If you have two or more secondary indexes set for an entity object, then you can retrieve sets of objects based on the intersection of multiple secondary index values. You do this using an `EntityJoin` class.

For example, suppose you had an entity class that represented automobiles. In that case, you might be storing information about automobiles such as color, number of doors, fuel mileage, automobile type, number of passengers, make, model, and year, to name just a few.

If you created a secondary index based this information, then you could use an `EntityJoin` to return all those objects representing cars with, say, two doors, that were built in 2002, and which are green in color.

To create a join cursor, you:

1. Open the primary index for the entity class on which you want to perform the join.
2. Open the secondary indexes that you want to use for the join.
3. Instantiate an `EntityJoin` object (you use the primary index to do this).
4. Use two or more calls to `EntityJoin.addCondition()` to identify the secondary indexes and their values that you want to use for the equality match.
5. Call `EntityJoin.entities()` to obtain a cursor that you can use to iterate over the join results.

For example, suppose we had an entity class that included the following features:

```
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 Automobiles {

    // Primary key is the vehicle identification number
    @PrimaryKey
    private String vin;

    // Secondary key is the vehicle's make
    @SecondaryKey(relate=MANY_TO_ONE)
    private String make;

    // Secondary key is the vehicle's color
    @SecondaryKey(relate=MANY_TO_ONE)
    private String color;

    ...

    public String getVIN() {
        return vin;
    }

    public String getMake() {
        return make;
    }

    public String getColor() {
        return color;
    }

    ...
```

Then we could perform an entity join that searches for all the red automobiles made by Toyota as follows:

```
PrimaryIndex<String,Automobiles> vin_pidx;
SecondaryIndex<String,String,Automobiles> make_sidx;
SecondaryIndex<String,String,Automobiles> color_sidx;

EntityJoin<String,Automobiles> join = new EntityJoin(vin_pidx);
join.addCondition(make_sidx,"Toyota");
join.addCondition(color_sidx,"Red");
```

```
// Now iterate over the results of the join operation
ForwardCursor<Automobiles> join_cursor = join.entities();
try {
    for (Automobiles autoi : join_cursor) {
        // do something with each object "autoi"
    }
// Always make sure the cursor is closed when we are done with it.
} finally {
    join_cursor.close();
}
```

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 `SimpleDA` class that we created earlier in this document (see SimpleDA.class (page 28)), you can delete the `SimpleEntityClass` object with a primary key of `keyone` as follows:

```
sda.pIdx.delete("keyone");
```

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 `SimpleEntityClass` with a secondary key of `skeyone`:

```
sda.sIdx.delete("skeyone");
```

You can delete any single object by positioning a cursor to that object and then calling the cursor's `delete()` method.

```
PrimaryIndex<String,SimpleEntityClass> pi =
    store.getPrimaryIndex(String.class, SimpleEntityClass.class);

SecondaryIndex<String,String,SimpleEntityClass> si =
    store.getSecondaryIndex(pi, String.class, "sKey");

EntityCursor<SimpleEntityClass> sec_cursor =
    si.subIndex("skeyone").entities();

try {
    SimpleEntityClass sec;
    Iterator<SimpleEntityClass> i = sec_cursor.iterator();
    while (sec = i.nextDup() != null) {
        if (sec.getSKey() == "some value") {
            i.delete();
        }
    }
// Always make sure the cursor is closed when we are done with it.
```

```
} finally {
    sec_cursor.close(); }
```

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 25) for more information.

Replacing Entity Objects

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

```
SimpleEntityClass sec = sda.pIdx.get("keyone");
sec.setSKey("skeyoneupdated");
sda.pIdx.put(sec);
```

Note that because we updated a field on the object that is a secondary key, this object will now be accessible by the secondary key of `skeyoneupdated` instead of the previous value, which was `skeyone`

Be aware that if you modify the object's primary key, the behavior is somewhat different. In this case, you cause a new instance of the object to be created in the store, instead of replacing an existing instance:

```
// Results in two objects in the store.  One with a
// primary index of "keyfive" and the other with primary index of
//'keyfivenew'.
SimpleEntityClass sec = sda.pIdx.get("keyfive");
sec.setPKey("keyfivenew");
sda.pIdx.put(sec);
```

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 `SimpleEntityClass` object in the entity store, and it changes them all so that they have a secondary index of `updatedskey`:

```
EntityCursor<SimpleEntityClass> sec_pcursor = sda.pIdx.entities();
for (SimpleEntityClass sec : sec_pcursor) {
    sec.setSKey("updatedskey");
    sec_pcursor.update(item);
}
sec_pcursor.close();
```

Chapter 6. A DPL Example

In order to illustrate DPL usage, we provide a complete working example in this chapter. This example reads and writes inventory and vendor information for a mythical business. The application consists of the following classes:

- Several classes used to encapsulate our application's data. See Vendor.java (page 41) and Inventory.java (page 43).
- A convenience class used to open and close our environment and entity store. See MyDbEnv (page 45).
- A class that loads data into the store. See ExampleDatabasePut.java (page 48).
- Finally, a class that reads data from the store. See ExampleInventoryRead.java (page 52).

Vendor.java

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.

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.

Inventory.java

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;
    }
}
```

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 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 java.io.FileNotFoundException;

import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.Environment;
import com.sleepycat.db.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);

        try {
            // Open the environment and entity store
            myEnv = new Environment(envHome, myEnvConfig);
            store = new EntityStore(myEnv, "EntityStore", storeConfig);
        } catch (FileNotFoundException fnfe) {
            System.err.println("setup(): " + fnfe.toString());
            System.exit(-1);
        }

    }

    // 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 environment.
                myEnv.close();
            } catch(DatabaseException dbe) {
                System.err.println("Error closing MyDbEnv: " +
                                   dbe.toString());
               System.exit(-1);
```

```
            }
        }
    }
}
```

DataAccessor.java

Now that we have implemented our data classes, we can write a class that will provide convenient access to our primary and secondary indexes. Note that like our data classes, this class is shared by both our example programs.

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.java (page 41) and Inventory.java (page 43) for those implementations.

```
package persist.gettingStarted;

import java.io.File;

import com.sleepycat.db.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;
}
```

ExampleDatabasePut.java

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`.

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 DB 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.db.DatabaseException;
```

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.java (page 47) and MyDbEnv (page 45).

```
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 its `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 45) 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.
            da.vendorByName.put(theVendor);
        }
    }
```

Now we can implement our `loadInventoryDb()` method. This does exactly the same thing as the `loadVendorsDb()` method.

```
    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.

        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.
            da.inventoryBySku.put(theInventory);
        }
    }
```

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() {}
}
```

ExampleInventoryRead.java

`ExampleInventoryRead` 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 the previous section (ExampleDatabasePut.java (page 48)).

```
package persist.gettingStarted;

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

import com.sleepycat.db.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();
                }
            }
        }
    }
}
```

Part II. Programming with the Base API

This section discusses application that are built using the DB base API. Note that most DB applications can probably be written using the DPL (see Programming with the Direct Persistence Layer (page 16) for more information). However, if you want to use Java 1.4 for your DB application, or if you are porting an application from the Berkeley DB API, then the base API is right for you.

Chapter 7. Databases

In Berkeley DB, a database is a collection of *records*. Records, in turn, consist of key/data pairings.

Conceptually, you can think of a `Database` as containing a two-column table where column 1 contains a key and column 2 contains data. Both the key and the data are managed using `DatabaseEntry` class instances (see Database Records (page 68) for details on this class). So, fundamentally, using a DB `Database` involves putting, getting, and deleting database records, which in turns involves efficiently managing information encapsulated by `DatabaseEntry` objects. The next several chapters of this book are dedicated to those activities.

Also, note that in the previous section of this book, Programming with the Direct Persistence Layer (page 16), we described the DPL The DPL handles all database management for you, including creating all primary and secondary databases as is required by your application. That said, if you are using the DPL you can access the underlying database for a given index if necessary. See the Javadoc for the DPL for more information.

Opening Databases

You open a database by instantiating a `Database` object.

Note that by default, DB does not create databases if they do not already exist. To override this behavior, set the creation property to true.

The following code fragment illustrates a database open:

```
package db.GettingStarted;

import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;

import java.io.FileNotFoundException;
...

Database myDatabase = null;

...

try {
    // Open the database. Create it if it does not already exist.
    DatabaseConfig dbConfig = new DatabaseConfig();
    dbConfig.setAllowCreate(true);
    myDatabase = new Database ("sampleDatabase.db",
                               null,
                               dbConfig);
} catch (DatabaseException dbe) {
    // Exception handling goes here
```

```
} catch (FileNotFoundException fnfe) {
    // Exception handling goes here
}
```

Closing Databases

Once you are done using the database, you must close it. You use the method to do this.

Closing a database causes it to become unusable until it is opened again. Note that you should make sure that any open cursors are closed before closing your database. Active cursors during a database close can cause unexpected results, especially if any of those cursors are writing to the database. You should always make sure that all your database accesses have completed before closing your database.

Cursors are described in Using Cursors (page 96) later in this manual.

Be aware that when you close the last open handle for a database, then by default its cache is flushed to disk. This means that any information that has been modified in the cache is guaranteed to be written to disk when the last handle is closed. You can manually perform this operation using the `Database.sync()` method, but for normal shutdown operations it is not necessary. For more information about syncing your cache, see Data Persistence (page 73).

The following code fragment illustrates a database close:

```
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.Database;

...

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

Database Properties

You can set database properties using the `DatabaseConfig` class. For each of the properties that you can set, there is a corresponding getter method. Also, you can always retrieve the `DatabaseConfig` object used by your database using the `Database.getConfig()` method.

There are a large number of properties that you can set using this class (see the javadoc for a complete listing). From the perspective of this manual, some of the more interesting properties are:

- `DatabaseConfig.setAllowCreate()`

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

- `DatabaseConfig.setBtreeComparator()`

  Sets the class that is used to compare the keys found on two database records. This class is used to determine the sort order for two records in the database. By default, byte for byte comparison is used. For more information, see Setting Comparison Functions (page 141).

- `DatabaseConfig.setDuplicateComparator()`

  Sets the class that is used to compare two duplicate records in the database. For more information, see Setting Comparison Functions (page 141).

- `DatabaseConfig.setSortedDuplicates()`

  If `true`, duplicate records are allowed in the database. If this value is `false`, then putting a duplicate record into the database results in an error return from the put call. Note that this property can be set only at database creation time. Default is `false`.

  Note that your database must not support duplicates if it is to be associated with one or more secondary indices. Secondaries are described in Secondary Databases (page 113).

- `DatabaseConfig.setExclusiveCreate()`

  If `true`, the database open fails if the database currently exists. That is, the open must result in the creation of a new database. Default is `false`.

- `DatabaseConfig.setReadOnly()`

  If true, the database is opened for read activities only. Default is `false`.

- `DatabaseConfig.setTruncate()`

  If true, the database is truncated; that is, it is emptied of all content.

- `DatabaseConfig.setType()`

Identifies the type of database that you want to create. This manual will exclusively use `DatabaseType.BTREE`.

In addition to these, there are also methods that allow you to control the IO stream used for error reporting purposes. These are described later in this manual.

For example:

```
package db.GettingStarted;

import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.DatabaseType;

import java.io.FileNotFoundException;

...
Database myDatabase = null;
try {
    DatabaseConfig dbConfig = new DatabaseConfig();
    dbConfig.setAllowCreate(true);
    dbConfig.setSortedDuplicates(true);
    dbConfig.setType(DatabaseType.BTREE);
    myDatabase = new Database("sampleDatabase.db",
                              null,
                              dbConfig);
} catch (DatabaseException dbe) {
    // Exception handling goes here.
} catch (FileNotFoundException fnfe) {
    // Exception handling goes here
}
```

Administrative Methods

Both the `Environment` and `Database` classes provide methods that are useful for manipulating databases. These methods are:

- `Database.getDatabaseName()`

  Returns the database's name.

  ```
  String dbName = myDatabase.getDatabaseName();
  ```

- `Database.rename()`

  Renames the specified database. If no value is given for the *`database`* parameter, then the entire file referenced by this method is renamed.

Never rename a database that has handles opened for it. Never rename a file that contains databases with opened handles.

```
import java.io.FileNotFoundException;
...
myDatabase.close();
try {
    myDatabase.rename("mydb.db",     // Database file to rename
                      null,          // Database to rename. Not used so
                                     // the entire file is renamed.
                      "newdb.db",    // New name to use.
                      null);         // DatabaseConfig object.
                                     // None provided.
} catch (FileNotFoundException fnfe) {
    // Exception handling goes here
}
```

- `Environment.truncateDatabase()`

  Deletes every record in the database and optionally returns the number of records that were deleted. Note that it is much less expensive to truncate a database without counting the number of records deleted than it is to truncate and count.

```
int numDiscarded =
    myEnv.truncate(null,                          // txn handle
                   myDatabase.getDatabaseName(),  // database name
                   true);                         // If true, then the
                                                  // number of records
                                                  // deleted are counted.
System.out.println("Discarded " + numDiscarded +
                   " records from database " +
                   myDatabase.getDatabaseName());
```

Error Reporting Functions

To simplify error reporting and handling, the `DatabaseConfig` class offers several useful methods.

- `DatabaseConfig.setErrorStream()`

  Sets the Java `OutputStream` to be used for displaying error messages issued by the DB library.

- `DatabaseConfig.setMessageHandler()`

  Defines the message handler that is called when an error message is issued by DB. The error prefix and message are passed to this callback. It is up to the application to display this information correctly.

  Note that the message handler must be an implementation of the `com.sleepycat.db.MessageHandler` interface.

- `DatabaseConfig.setErrorPrefix()`

  Sets the prefix used for any error messages issued by the DB library.

For example, to send all your error messages to a particular message handler, first implement the handler:

```
package db.GettingStarted;

import com.sleepycat.db.Environment;
import com.sleepycat.db.MessageHandler;

public class MyMessageHandler implements MessageHandler  {

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

    public void message(Environment dbenv, String message)
    {
        // Put your special message handling code here
    }

}
```

And then set up your database to use the message handler by identifying it on the database's `DatabaseConfig` object:

```
package db.GettingStarted;

import com.sleepycat.db.DatabaseConfig;

...

DatabaseConfig myDbConfig = new DatabaseConfig();
MyMessageHandler mmh = new MyMessageHandler();
myDbConfig.setMessageHandler(mmh);
```

Managing Databases in Environments

In Database Environments (page 11), we introduced environments. While environments are not used in the example built in this book, they are so commonly used for a wide class of DB applications that it is necessary to show their basic usage, if only from a completeness perspective.

To use an environment, you must first open it. At open time, you must identify the directory in which it resides. This directory must exist prior to the open attempt. You can also identify open properties, such as whether the environment can be created if it does not already exist.

You will also need to initialize the in-memory cache when you open your environment.

For example, to create an environment handle and open an environment:

```
package db.GettingStarted;

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

import java.io.File;
import java.io.FileNotFoundException;

...

Environment myEnv = null;
File envHome = new File("/export1/testEnv");
try {
    EnvironmentConfig envConf = new EnvironmentConfig();
    envConf.setAllowCreate(true);         // If the environment does not
                                          // exist, create it.
    envConf.setInitializeCache(true);     // Initialize the in-memory
                                          // cache.

    myEnv = new Environment(envHome, envConf);
} catch (DatabaseException de) {
    // Exception handling goes here
} catch (FileNotFoundException fnfe) {
    // Exception handling goes here
}
```

Once an environment is opened, you can open databases in it. Note that by default databases are stored in the environment's home directory, or relative to that directory if you provide any sort of a path in the database's file name:

```
package db.GettingStarted;

import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseType;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.Environment;
import com.sleepycat.db.EnvironmentConfig;

import java.io.File;
import java.io.FileNotFoundException;

...

Environment myEnv = null;
Database myDb = null;
File envHome = new File("/export1/testEnv");
String dbFileName = new String("mydb.db", "UTF-8");
```

```
try {
    EnvironmentConfig envConf = new EnvironmentConfig();
    envConf.setAllowCreate(true);
    DatabaseConfig dbConfig = new DatabaseConfig();
    dbConfig.setAllowCreate(true);
    dbConfig.setType(DatabaseType.BTREE);

    myEnv = new Environment(envHome, envConf);
    myDb = myEnv.openDatabase(null, dbFileName, null, dbConfig);
} catch (DatabaseException de) {
    // Exception handling goes here
} catch (FileNotFoundException fnfe) {
    // Exception handling goes here
}
```

When you are done with an environment, you must close it. Before you close an environment, make sure you close any opened databases.

```
finally {
    try {
        if (myDb != null) {
            myDb.close();
        }

        if (myEnv != null) {
            myEnv.close();
        }
    } catch (DatabaseException de) {
        // Exception handling goes here
    }
}
```

Database Example

Throughout this book we will build a couple of applications that load and retrieve inventory data from DB databases. While we are not yet ready to begin reading from or writing to our databases, we can at least create the class that we will use to manage our databases.

Note that subsequent examples in this book will build on this code to perform the more interesting work of writing to and reading from the databases.

Note that you can find the complete implementation of these functions in:

```
DB_INSTALL/examples_java/db/GettingStarted
```

where *DB_INSTALL* is the location where you placed your DB distribution.

Example 7.1. MyDbs Class

To manage our database open and close activities, we encapsulate them in the `MyDbs` class. There are several good reasons to do this, the most important being that we can ensure our databases are closed by putting that activity in the `MyDbs` class destructor.

To begin, we import some needed classes:

```
// File: MyDbs.java
package db.GettingStarted;

import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.DatabaseType;

import java.io.FileNotFoundException;
```

And then we write our class declaration and provided some necessary private data members:

```
public class MyDbs {

    // The databases that our application uses
    private Database vendorDb = null;
    private Database inventoryDb = null;

    private String vendordb = "VendorDB.db";
    private String inventorydb = "InventoryDB.db";

    // Our constructor does nothing
    public MyDbs() {}
```

Next we need a `setup()` method. This is where we configure and open our databases.

```
    // The setup() method opens all our databases
    // for us.
    public void setup(String databasesHome)
        throws DatabaseException {

        DatabaseConfig myDbConfig = new DatabaseConfig();

        myDbConfig.setErrorStream(System.err);
        myDbConfig.setErrorPrefix("MyDbs");
        myDbConfig.setType(DatabaseType.BTREE);
        myDbConfig.setAllowCreate(true);

        // Now open, or create and open, our databases
        // Open the vendors and inventory databases
        try {
            vendordb = databasesHome + "/" + vendordb;
            vendorDb = new Database(vendordb,
```

```
                                    null,
                                    myDbConfig);

            inventorydb = databasesHome + "/" + inventorydb;
            inventoryDb = new Database(inventorydb,
                                       null,
                                       myDbConfig);
        } catch(FileNotFoundException fnfe) {
            System.err.println("MyDbs: " + fnfe.toString());
            System.exit(-1);
        }
    }
```

Finally, we provide some getter methods, and our `close()` method.

```
   // getter methods
    public Database getVendorDB() {
        return vendorDb;
    }

    public Database getInventoryDB() {
        return inventoryDb;
    }

    // Close the databases
    public void close() {
        try {
            if (vendorDb != null) {
                vendorDb.close();
            }

            if (inventoryDb != null) {
                inventoryDb.close();
            }
        } catch(DatabaseException dbe) {
            System.err.println("Error closing MyDbs: " +
                               dbe.toString());
            System.exit(-1);
        }
    }
}
```

Chapter 8. Database Records

DB records contain two parts — a key and some data. Both the key and its corresponding data are encapsulated in `DatabaseEntry` class objects. Therefore, to access a DB record, you need two such objects, one for the key and one for the data.

`DatabaseEntry` can hold any kind of data from simple Java primitive types to complex Java objects so long as that data can be represented as a Java `byte` array. Note that due to performance considerations, you should not use Java serialization to convert a Java object to a `byte` array. Instead, use the Bind APIs to perform this conversion (see Using the BIND APIs (page 73) for more information).

This chapter describes how you can convert both Java primitives and Java class objects into and out of `byte` arrays. It also introduces storing and retrieving key/value pairs from a database. In addition, this chapter describes how you can use comparators to influence how DB sorts its database records.

Using Database Records

Each database record is comprised of two `DatabaseEntry` objects — one for the key and another for the data. The key and data information are passed to- and returned from DB using `DatabaseEntry` objects as `byte` arrays. Using `DatabaseEntry`s allows DB to change the underlying byte array as well as return multiple values (that is, key and data). Therefore, using `DatabaseEntry` instances is mostly an exercise in efficiently moving your keys and your data in and out of `byte` arrays.

For example, to store a database record where both the key and the data are Java `String` objects, you instantiate a pair of `DatabaseEntry` objects:

```
package db.GettingStarted;

import com.sleepycat.db.DatabaseEntry;

...

String aKey = "key";
String aData = "data";

try {
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry(aData.getBytes("UTF-8"));
} catch (Exception e) {
    // Exception handling goes here
}

    // Storing the record is described later in this chapter
```

Notice that we specify `UTF-8` when we retrieve the `byte` array from our `String` object. Without parameters, `String.getBytes()` uses the Java system's default encoding. You

should never use a system's default encoding when storing data in a database because the encoding can change.

When the record is retrieved from the database, the method that you use to perform this operation populates two `DatabaseEntry` instances for you, one for the key and another for the data. Assuming Java `String` objects, you retrieve your data from the `DatabaseEntry` as follows:

```
package db.GettingStarted;

import com.sleepycat.db.DatabaseEntry;

...

// theKey and theData are DatabaseEntry objects. Database
// retrieval is described later in this chapter. For now,
// we assume some database get method has populated these
// objects for us.

// Use DatabaseEntry.getData() to retrieve the encapsulated Java
// byte array.

byte[] myKey = theKey.getData();
byte[] myData = theData.getData();

String key = new String(myKey, "UTF-8");
String data = new String(myData, "UTF-8");
```

There are a large number of mechanisms that you can use to move data in and out of `byte` arrays. To help you with this activity, DB provides the bind APIs. These APIs allow you to efficiently store both primitive data types and complex objects in `byte` arrays.

The next section describes basic database put and get operations. A basic understanding of database access is useful when describing database storage of more complex data such as is supported by the bind APIs. Basic bind API usage is then described in Using the BIND APIs (page 73).

Reading and Writing Database Records

When reading and writing database records, be aware that there are some slight differences in behavior depending on whether your database supports duplicate records. Two or more database records are considered to be duplicates of one another if they share the same key. The collection of records sharing the same key are called a *duplicates set*. In DB, a given key is stored only once for a single duplicates set.

By default, DB databases do not support duplicate records. Where duplicate records are supported, cursors (see below) are typically used to access all of the records in the duplicates set.

DB provides two basic mechanisms for the storage and retrieval of database key/data pairs:

- The `Database.put()` and `Database.get()` methods provide the easiest access for all non-duplicate records in the database. These methods are described in this section.
- Cursors provide several methods for putting and getting database records. Cursors and their database access methods are described in Using Cursors (page 96).

Writing Records to the Database

Records are stored in the database using whatever organization is required by the access method that you have selected. In some cases (such as BTree), records are stored in a sort order that you may want to define (see Setting Comparison Functions (page 141) for more information).

In any case, the mechanics of putting and getting database records do not change once you have selected your access method, configured your sorting routines (if any), and opened your database. From your code's perspective, a simple database put and get is largely the same no matter what access method you are using.

You can use the following methods to put database records:

- `Database.put()`

  Puts a database record into the database. If your database does not support duplicate records, and if the provided key already exists in the database, then the currently existing record is replaced with the new data.

- `Database.putNoOverwrite()`

  Disallows overwriting (replacing) an existing record in the database. If the provided key already exists in the database, then this method returns `OperationStatus.KEYEXIST` even if the database supports duplicates.

- `Database.putNoDupData()`

  Puts a database record into the database. If the provided key and data already exists in the database (that is, if you are attempting to put a record that compares equally to an existing record), then this returns `OperationStatus.KEYEXIST`.

When you put database records, you provide both the key and the data as `DatabaseEntry` objects. This means you must convert your key and data into a Java `byte` array. For example:

```
package db.GettingStarted;

import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.Database;

...

// Database opens omitted for clarity.
// Databases must NOT be opened read-only.

String aKey = "myFirstKey";
```

```
String aData = "myFirstData";

try {
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry(aData.getBytes("UTF-8"));
    myDatabase.put(null, theKey, theData);
} catch (Exception e) {
    // Exception handling goes here
}
```

Getting Records from the Database

The `Database` class provides several methods that you can use to retrieve database records. Note that if your database supports duplicate records, then these methods will only ever return the first record in a duplicate set. For this reason, if your database supports duplicates, you should use a cursor to retrieve records from it. Cursors are described in Using Cursors (page 96).

You can use either of the following methods to retrieve records from the database:

- `Database.get()`

  Retrieves the record whose key matches the key provided to the method. If no records exists that uses the provided key, then `OperationStatus.NOTFOUND` is returned.

- `Database.getSearchBoth()`

  Retrieve the record whose key matches both the key and the data provided to the method. If no record exists that uses the provided key and data, then `OperationStatus.NOTFOUND` is returned.

Both the key and data for a database record are returned as byte arrays in `DatabaseEntry` objects. These objects are passed as parameter values to the `Database.get()` method.

In order to retrieve your data once `Database.get()` has completed, you must retrieve the `byte` array stored in the `DatabaseEntry` and then convert that `byte` array back to the appropriate datatype. For example:

```
package db.GettingStarted;

import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.Database;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;

...

Database myDatabase = null;
// Database opens omitted for clarity.
// Database may be opened read-only.
```

```
String aKey = "myFirstKey";

try {
    // Create a pair of DatabaseEntry objects. theKey
    // is used to perform the search. theData is used
    // to store the data returned by the get() operation.
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();

    // Perform the get.
    if (myDatabase.get(null, theKey, theData, LockMode.DEFAULT) ==
        OperationStatus.SUCCESS) {

        // Recreate the data String.
        byte[] retData = theData.getData();
        String foundData = new String(retData, "UTF-8");
        System.out.println("For key: '" + aKey + "' found data: '" +
                           foundData + "'.");
    } else {
        System.out.println("No record found for key '" + aKey + "'.");
    }
} catch (Exception e) {
    // Exception handling goes here
}
```

Deleting Records

You can use the `Database.delete()` method to delete a record from the database. If your database supports duplicate records, then all records associated with the provided key are deleted. To delete just one record from a list of duplicates, use a cursor. Cursors are described in Using Cursors (page 96).

You can also delete every record in the database by using `Environment.truncateDatabase()`.

For example:

```
package db.GettingStarted;

import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.Database;

...

Database myDatabase = null;
// Database opens omitted for clarity.
// Database can NOT be opened read-only.

try {
    String aKey = "myFirstKey";
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));
```

```
    // Perform the deletion. All records that use this key are
    // deleted.
    myDatabase.delete(null, theKey);
} catch (Exception e) {
    // Exception handling goes here
}
```

Data Persistence

When you perform a database modification, your modification is made in the in-memory cache. This means that your data modifications are not necessarily flushed to disk, and so your data may not appear in the database after an application restart.

Note that as a normal part of closing a database, its cache is written to disk. However, in the event of an application or system failure, there is no guarantee that your databases will close cleanly. In this event, it is possible for you to lose data. Under extremely rare circumstances, it is also possible for you to experience database corruption.

Therefore, if you care if your data is durable across system failures, and to guard against the rare possibility of database corruption, you should use transactions to protect your database modifications. Every time you commit a transaction, DB ensures that the data will not be lost due to application or system failure. Transaction usage is described in the *Berkeley DB Getting Started with Transaction Processing* guide.

If you do not want to use transactions, then the assumption is that your data is of a nature that it need not exist the next time your application starts. You may want this if, for example, you are using DB to cache data relevant only to the current application runtime.

If, however, you are not using transactions for some reason and you still want some guarantee that your database modifications are persistent, then you should periodically run environment syncs. Syncs cause any dirty entries in the in-memory cache and the operating system's file cache to be written to disk. As such, they are quite expensive and you should use them sparingly.

Remember that by default a sync is performed any time a non-transactional database is closed cleanly. (You can override this behavior by specifying `true` on the call to `Database.close()`.) That said, you can manually run a sync by calling `Database.sync()`.

If your application or system crashes and you are not using transactions, then you should either discard and recreate your databases, or verify them. You can verify a database using Database.verify(). If your databases do not verify cleanly, use the **db_dump** command to salvage as much of the database as is possible. Use either the `-R` or `-r` command line options to control how aggressive **db_dump** should be when salvaging your databases.

Using the BIND APIs

Except for Java String and boolean types, efficiently moving data in and out of Java byte arrays for storage in a database can be a nontrivial operation. To help you with this problem, DB provides the Bind APIs. While these APIs are described in detail in the *Berkeley DB Collections Tutorial*, this section provides a brief introduction to using the Bind APIs with:

- Single field numerical and string objects

  Use this if you want to store a single numerical or string object, such as `Long`, `Double`, or `String`.

- Complex objects that implement Java serialization.

  Use this if you are storing objects that implement `Serializable` and if you do not need to sort them.

- Non-serialized complex objects.

  If you are storing objects that do not implement serialization, you can create your own custom tuple bindings. Note that you should use custom tuple bindings even if your objects are serializable if you want to sort on that data.

Numerical and String Objects

You can use the Bind APIs to store primitive data in a `DatabaseEntry` object. That is, you can store a single field containing one of the following types:

- `String`
- `Character`
- `Boolean`
- `Byte`
- `Short`
- `Integer`
- `Long`
- `Float`
- `Double`

To store primitive data using the Bind APIs:

1. Create an `EntryBinding` object.

   When you do this, you use `TupleBinding.getPrimitiveBinding()` to return an appropriate binding for the conversion.

2. Use the `EntryBinding` object to place the numerical object on the `DatabaseEntry`.

Once the data is stored in the DatabaseEntry, you can put it to the database in whatever manner you wish. For example:

```
package db.GettingStarted;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseEntry;

...

Database myDatabase = null;
// Database open omitted for clarity.

// Need a key for the put.
try {
    String aKey = "myLong";
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));

    // Now build the DatabaseEntry using a TupleBinding
    Long myLong = new Long(123456789l);
    DatabaseEntry theData = new DatabaseEntry();
    EntryBinding myBinding = TupleBinding.getPrimitiveBinding(Long.class);
    myBinding.objectToEntry(myLong, theData);

    // Now store it
    myDatabase.put(null, theKey, theData);
} catch (Exception e) {
    // Exception handling goes here
}
```

Retrieval from the `DatabaseEntry` object is performed in much the same way:

```
package db.GettingStarted;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;

...

Database myDatabase = null;
// Database open omitted for clarity

try {
    // Need a key for the get
    String aKey = "myLong";
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));
```

```
    // Need a DatabaseEntry to hold the associated data.
    DatabaseEntry theData = new DatabaseEntry();

    // Bindings need only be created once for a given scope
    EntryBinding myBinding = TupleBinding.getPrimitiveBinding(Long.class);

    // Get it
    OperationStatus retVal = myDatabase.get(null, theKey, theData,
                                            LockMode.DEFAULT);
    String retKey = null;
    if (retVal == OperationStatus.SUCCESS) {
        // Recreate the data.
        // Use the binding to convert the byte array contained in theData
        // to a Long type.
        Long theLong = (Long) myBinding.entryToObject(theData);
        retKey = new String(theKey.getData(), "UTF-8");
        System.out.println("For key: '" + retKey + "' found Long: '" +
                           theLong + "'.");
    } else {
        System.out.println("No record found for key '" + retKey + "'.");
    }
} catch (Exception e) {
    // Exception handling goes here
}
```

Serializable Complex Objects

Frequently your application requires you to store and manage objects for your record data and/or keys. You may need to do this if you are caching objects created by another process. You may also want to do this if you want to store multiple data values on a record. When used with just primitive data, or with objects containing a single data member, DB database records effectively represent a single row in a two-column table. By storing a complex object in the record, you can turn each record into a single row in an *n*-column table, where *n* is the number of data members contained by the stored object(s).

In order to store objects in a DB database, you must convert them to and from a `byte` array. The first instinct for many Java programmers is to do this using Java serialization. While this is functionally a correct solution, the result is poor space-performance because this causes the class information to be stored on every such database record. This information can be quite large and it is redundant — the class information does not vary for serialized objects of the same type.

In other words, directly using serialization to place your objects into byte arrays means that you will be storing a great deal of unnecessary information in your database, which ultimately leads to larger databases and more expensive disk I/O.

The easiest way for you to solve this problem is to use the Bind APIs to perform the serialization for you. Doing so causes the extra object information to be saved off to a unique `Database`

dedicated for that purpose. This means that you do not have to duplicate that information on each record in the `Database` that your application is using to store its information.

Note that when you use the Bind APIs to perform serialization, you still receive all the benefits of serialization. You can still use arbitrarily complex object graphs, and you still receive built-in class evolution through the serialVersionUID (SUID) scheme. All of the Java serialization rules apply without modification. For example, you can implement Externalizable instead of Serializable.

Usage Caveats

Before using the Bind APIs to perform serialization, you may want to consider writing your own custom tuple bindings. Specifically, avoid serialization if:

- If you need to sort based on the objects your are storing. The sort order is meaningless for the byte arrays that you obtain through serialization. Consequently, you should not use serialization for keys if you care about their sort order. You should also not use serialization for record data if your `Database` supports duplicate records and you care about sort order.
- You want to minimize the size of your byte arrays. Even when using the Bind APIs to perform the serialization the resulting `byte` array may be larger than necessary. You can achieve more compact results by building your own custom tuple binding.
- You want to optimize for speed. In general, custom tuple bindings are faster than serialization at moving data in and out of `byte` arrays.

For information on building your own custom tuple binding, see Custom Tuple Bindings (page 81).

Serializing Objects

To store a serializable complex object using the Bind APIs:

1. Implement java.io.Serializable in the class whose instances that you want to store.
2. Open (create) your databases. You need two. The first is the database that you use to store your data. The second is used to store the class information.
3. Instantiate a class catalog. You do this with `com.sleepycat.bind.serial.StoredClassCatalog`, and at that time you must provide a handle to an open database that is used to store the class information.
4. Create an entry binding that uses `com.sleepycat.bind.serial.SerialBinding`.
5. Instantiate an instance of the object that you want to store, and place it in a `DatabaseEntry` using the entry binding that you created in the previous step.

For example, suppose you want to store a long, double, and a String as a record's data. Then you might create a class that looks something like this:

```
package db.GettingStarted;

import java.io.Serializable;
```

```
public class MyData implements Serializable {
    private long longData;
    private double doubleData;
    private String description;

    MyData() {
        longData = 0;
        doubleData = 0.0;
        description = null;
    }

    public void setLong(long data) {
        longData = data;
    }

    public void setDouble(double data) {
        doubleData = data;
    }

    public void setDescription(String data) {
        description = data;
    }

    public long getLong() {
        return longData;
    }

    public double getDouble() {
        return doubleData;
    }

    public String getDescription() {
        return description;
    }
}
```

You can then store instances of this class as follows:

```
package db.GettingStarted;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.StoredClassCatalog;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseType;
...
```

```
// The key data.
String aKey = "myData";

// The data data
MyData data2Store = new MyData();
data2Store.setLong(123456789l);
data2Store.setDouble(1234.9876543);
data2Store.setDescription("A test instance of this class");

try {
    // Open the database that you will use to store your data
    DatabaseConfig myDbConfig = new DatabaseConfig();
    myDbConfig.setAllowCreate(true);
    myDbConfig.setSortedDuplicates(true);
    myDbConfig.setType(DatabaseType.BTREE);
    Database myDatabase = new Database("myDb", null, myDbConfig);

    // Open the database that you use to store your class information.
    // The db used to store class information does not require duplicates
    // support.
    myDbConfig.setSortedDuplicates(false);
    Database myClassDb = new Database("classDb", null, myDbConfig);

    // Instantiate the class catalog
    StoredClassCatalog classCatalog = new StoredClassCatalog(myClassDb);

    // Create the binding
    EntryBinding dataBinding = new SerialBinding(classCatalog,
                                                 MyData.class);

    // Create the DatabaseEntry for the key
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));

    // Create the DatabaseEntry for the data. Use the EntryBinding object
    // that was just created to populate the DatabaseEntry
    DatabaseEntry theData = new DatabaseEntry();
    dataBinding.objectToEntry(data2Store, theData);

    // Put it as normal
    myDatabase.put(null, theKey, theData);

    // Database and environment close omitted for brevity
} catch (Exception e) {
    // Exception handling goes here
}
```

Deserializing Objects

Once an object is stored in the database, you can retrieve the `MyData` objects from the retrieved `DatabaseEntry` using the Bind APIs in much the same way as is described above. For example:

```
package db.GettingStarted;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.StoredClassCatalog;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseType;
import com.sleepycat.db.LockMode;

...

// The key data.
String aKey = "myData";

try {
    // Open the database that stores your data
    DatabaseConfig myDbConfig = new DatabaseConfig();
    myDbConfig.setAllowCreate(false);
    myDbConfig.setType(DatabaseType.BTREE);
    Database myDatabase = new Database("myDb", null, myDbConfig);

    // Open the database that stores your class information.
    Database myClassDb = new Database("classDb", null, myDbConfig);

    // Instantiate the class catalog
    StoredClassCatalog classCatalog = new StoredClassCatalog(myClassDb);

    // Create the binding
    EntryBinding dataBinding = new SerialBinding(classCatalog,
                                                 MyData.class);

    // Create DatabaseEntry objects for the key and data
    DatabaseEntry theKey = new DatabaseEntry(aKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();

    // Do the get as normal
    myDatabase.get(null, theKey, theData, LockMode.DEFAULT);

    // Recreate the MyData object from the retrieved DatabaseEntry using
    // the EntryBinding created above
    MyData retrievedData = (MyData) dataBinding.entryToObject(theData);
```

```
    // Database and environment close omitted for brevity
} catch (Exception e) {
    // Exception handling goes here
}
```

Custom Tuple Bindings

If you want to store complex objects in your database, then you can use tuple bindings to do this. While they are more work to write and maintain than if you were to use serialization, the `byte` array conversion is faster. In addition, custom tuple bindings should allow you to create `byte` arrays that are smaller than those created by serialization. Custom tuple bindings also allow you to optimize your BTree comparisons, whereas serialization does not.

For information on using serialization to store complex objects, see Serializable Complex Objects (page 76).

To store complex objects using a custom tuple binding:

1. Implement the class whose instances that you want to store. Note that you do not have to implement the Serializable interface.
2. Write a tuple binding using the `com.sleepycat.bind.tuple.TupleBinding` class.
3. Open (create) your database. Unlike serialization, you only need one.
4. Create an entry binding that uses the tuple binding that you implemented in step 2.
5. Instantiate an instance of the object that you want to store, and place it in a `DatabaseEntry` using the entry binding that you created in the previous step.

For example, suppose you want to your keys to be instances of the following class:

```
package db.GettingStarted;

public class MyData2 {
    private long longData;
    private Double doubleData;
    private String description;

    public MyData2() {
        longData = 0;
        doubleData = new Double(0.0);
        description = "";
    }

    public void setLong(long data) {
        longData = data;
    }

    public void setDouble(Double data) {
        doubleData = data;
```

```
    }

    public void setString(String data) {
        description = data;
    }

    public long getLong() {
        return longData;
    }

    public Double getDouble() {
        return doubleData;
    }

    public String getString() {
        return description;
    }
}
```

In this case, you need to write a tuple binding for the `MyData2` class. When you do this, you must implement the `TupleBinding.objectToEntry()` and `TupleBinding.entryToObject()` abstract methods. Remember the following as you implement these methods:

- You use `TupleBinding.objectToEntry()` to convert objects to `byte` arrays. You use `com.sleepycat.bind.tuple.TupleOutput` to write primitive data types to the `byte` array. Note that `TupleOutput` provides methods that allows you to work with numerical types (`long`, `double`, `int`, and so forth) and not the corresponding `java.lang` numerical classes.
- The order that you write data to the `byte` array in `TupleBinding.objectToEntry()` is the order that it appears in the array. So given the `MyData2` class as an example, if you write `description`, `doubleData`, and then `longData`, then the resulting byte array will contain these data elements in that order. This means that your records will sort based on the value of the `description` data member and then the `doubleData` member, and so forth. If you prefer to sort based on, say, the `longData` data member, write it to the byte array first.
- You use `TupleBinding.entryToObject()` to convert the `byte` array back into an instance of your original class. You use `com.sleepycat.bind.tuple.TupleInput` to get data from the `byte` array.
- The order that you read data from the `byte` array must be exactly the same as the order in which it was written.

For example:

```
package db.GettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.bind.tuple.TupleInput;
import com.sleepycat.bind.tuple.TupleOutput;
```

```
public class MyTupleBinding extends TupleBinding {

    // Write a MyData2 object to a TupleOutput
    public void objectToEntry(Object object, TupleOutput to) {

        MyData2 myData = (MyData2)object;

        // Write the data to the TupleOutput (a DatabaseEntry).
        // Order is important. The first data written will be
        // the first bytes used by the default comparison routines.
        to.writeDouble(myData.getDouble().doubleValue());
        to.writeLong(myData.getLong());
        to.writeString(myData.getString());
    }

    // Convert a TupleInput to a MyData2 object
    public Object entryToObject(TupleInput ti) {

        // Data must be read in the same order that it was
        // originally written.
        Double theDouble = new Double(ti.readDouble());
        long theLong = ti.readLong();
        String theString = ti.readString();

        MyData2 myData = new MyData2();
        myData.setDouble(theDouble);
        myData.setLong(theLong);
        myData.setString(theString);

        return myData;
    }
}
```

In order to use the tuple binding, instantiate the binding and then use:

- `MyTupleBinding.objectToEntry()` to convert a MyData2 object to a `DatabaseEntry`.
- `MyTupleBinding.entryToObject()` to convert a `DatabaseEntry` to a `MyData2` object.

For example:

```
package db.GettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.db.DatabaseEntry;

...

TupleBinding keyBinding = new MyTupleBinding();
```

```
MyData2 theKeyData = new MyData2();
theKeyData.setLong(123456789l);
theKeyData.setDouble(new Double(12345.6789));
theKeyData.setString("My key data");

DatabaseEntry myKey = new DatabaseEntry();

try {
    // Store theKeyData in the DatabaseEntry
    keyBinding.objectToEntry(theKeyData, myKey);

    ...
    // Database put and get activity omitted for clarity
    ...

    // Retrieve the key data
    theKeyData = (MyData2) keyBinding.entryToObject(myKey);
} catch (Exception e) {
    // Exception handling goes here
}
```

Database Usage Example

In MyDbs Class (page 66) we created a class that opens and closes databases for us. We now make use of that class to load inventory data into two databases that we will use for our inventory system.

Again, remember that you can find the complete implementation for these functions in:

```
DB_INSTALL/examples_java/db/GettingStarted
```

where *DB_INSTALL* is the location where you placed your DB distribution.

Note that in this example, we are going to save two types of information. First there are a series of inventory records that identify information about some food items (fruits, vegetables, and desserts). These records identify particulars about each item such as the vendor that the item can be obtained from, how much the vendor has in stock, the price per unit, and so forth.

We also want to manage vendor contact information, such as the vendor's address and phone number, the sales representative's name and his phone number, and so forth.

Example 8.1. Inventory.java

All Inventory data is encapsulated in an instance of the following class. Note that because this class is not serializable, we need a custom tuple binding in order to place it on a `DatabaseEntry` object. Because the `TupleInput` and `TupleOutput` classes used by custom tuple bindings support Java numerical types and not Java numerical classes, we use `int` and `float` here instead of the corresponding `Integer` and `Float` classes.

```
// File Inventory.java
package db.GettingStarted;

public class Inventory {

    private String sku;
    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; }

}
```

Example 8.2. Vendor.java

The data for vendor records are stored in instances of the following class. Notice that we are using serialization with this class for no other reason than to demonstrate serializing a class instance.

```
// File Vendor.java
package db.GettingStarted;

import java.io.Serializable;

public class Vendor implements Serializable {

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

    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;
    }

    ...
    // Corresponding getter methods omitted for brevity.
    // See examples/je/gettingStarted/Vendor.java
    // for a complete implementation of this class.

}
```

Because we will not be using serialization to convert our `Inventory` objects to a `DatabaseEntry` object, we need a custom tuple binding:

Example 8.3. InventoryBinding.java

```
// File InventoryBinding.java
package db.GettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.bind.tuple.TupleInput;
import com.sleepycat.bind.tuple.TupleOutput;

public class InventoryBinding extends TupleBinding {

    // Implement this abstract method. Used to convert
    // a DatabaseEntry to an Inventory object.
    public Object entryToObject(TupleInput ti) {

        String sku = ti.readString();
        String itemName = ti.readString();
        String category = ti.readString();
        String vendor = ti.readString();
        int vendorInventory = ti.readInt();
        float vendorPrice = ti.readFloat();

        Inventory inventory = new Inventory();
        inventory.setSku(sku);
        inventory.setItemName(itemName);
        inventory.setCategory(category);
        inventory.setVendor(vendor);
        inventory.setVendorInventory(vendorInventory);
        inventory.setVendorPrice(vendorPrice);

        return inventory;
    }

    // Implement this abstract method. Used to convert a
```

```
    // Inventory object to a DatabaseEntry object.
    public void objectToEntry(Object object, TupleOutput to) {

        Inventory inventory = (Inventory)object;

        to.writeString(inventory.getSku());
        to.writeString(inventory.getItemName());
        to.writeString(inventory.getCategory());
        to.writeString(inventory.getVendor());
        to.writeInt(inventory.getVendorInventory());
        to.writeFloat(inventory.getVendorPrice());
    }
}
```

In order to store the data identified above, we write the `ExampleDatabaseLoad` application. This application loads the inventory and vendor databases for you.

Inventory information is stored in a `Database` dedicated for that purpose. The key for each such record is a product SKU. The inventory data stored in this database are objects of the `Inventory` class (see Inventory.java (page 84) for more information). `ExampleDatabaseLoad` loads the inventory database as follows:

1. Reads the inventory data from a flat text file prepared in advance for this purpose.
2. Uses `java.lang.String` to create a key based on the item's SKU.
3. Uses an `Inventory` class instance for the record data. This object is stored on a `DatabaseEntry` object using `InventoryBinding`, a custom tuple binding that we implemented above.
4. Saves each record to the inventory database.

Vendor information is also stored in a `Database` dedicated for that purpose. The vendor data stored in this database are objects of the `Vendor` class (see Vendor.java (page 86) for more information). To load this `Database`, `ExampleDatabaseLoad` does the following:

1. Reads the vendor data from a flat text file prepared in advance for this purpose.
2. Uses the vendor's name as the record's key.
3. Uses a `Vendor` class instance for the record data. This object is stored on a `DatabaseEntry` object using `com.sleepycat.bind.serial.SerialBinding`.

Example 8.4. Stored Class Catalog Management with MyDbs

Before we can write `ExampleDatabaseLoad`, we need to update `MyDbs.java` to support the class catalogs that we need for this application.

To do this, we start by importing an additional class to support stored class catalogs:

```
// File: MyDbs.java
package db.GettingStarted;
```

```
import com.sleepycat.bind.serial.StoredClassCatalog;

import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.DatabaseType;

import java.io.FileNotFoundException;
```

We also need to add two additional private data members to this class. One supports the database used for the class catalog, and the other is used as a handle for the class catalog itself.

```
public class MyDbs {

    // The databases that our application uses
    private Database vendorDb = null;
    private Database inventoryDb = null;
    private Database classCatalogDb = null;

    // Needed for object serialization
    private StoredClassCatalog classCatalog;

    private String vendordb = "VendorDB.db";
    private String inventorydb = "InventoryDB.db";
    private String classcatalogdb = "ClassCatalogDB.db";

    // Our constructor does nothing
    public MyDbs() {}
```

Next we need to update the `MyDbs.setup()` method to open the class catalog database and create the class catalog.

```
    // The setup() method opens all our databases
    // for us.
    public void setup(String databasesHome)
        throws DatabaseException {

        DatabaseConfig myDbConfig = new DatabaseConfig();

        ...
        // Database configuration omitted for brevity
        ...

        // Now open, or create and open, our databases
        // Open the vendors and inventory databases
        try {
            vendordb = databasesHome + "/" + vendordb;
            vendorDb = new Database(vendordb,
```

```
                                    null,
                                    myDbConfig);

            inventorydb = databasesHome + "/" + inventorydb;
            inventoryDb = new Database(inventorydb,
                                       null,
                                       myDbConfig);

            // Open the class catalog db. This is used to
            // optimize class serialization.
            classcatalogdb = databasesHome + "/" + classcatalogdb;
            classCatalogDb = new Database(classcatalogdb,
                                          null,
                                          myDbConfig);

        } catch(FileNotFoundException fnfe) {
            System.err.println("MyDbs: " + fnfe.toString());
            System.exit(-1);
        }
    }
```

Finally we need a getter method to return the class catalog. Note that we do not provide a getter for the catalog database itself – our application has no need for that.

We also update our `close()` to close our class catalog.

```
    // getter methods
     public Database getVendorDB() {
         return vendorDb;
     }

     public Database getInventoryDB() {
         return inventoryDb;
     }

     public StoredClassCatalog getClassCatalog() {
         return classCatalog;
     }
```

Finally, we need our `close()` method:

```


     // Close the databases
     public void close() {
         try {
             if (vendorDb != null) {
                 vendorDb.close();
             }
```

```
            if (inventoryDb != null) {
                inventoryDb.close();
            }

            if (classCatalogDb != null) {
                classCatalogDb.close();
            }
        } catch(DatabaseException dbe) {
            System.err.println("Error closing MyDbs: " +
                               dbe.toString());
            System.exit(-1);
        }
    }
}
```

So far we have identified the data that we want to store in our databases and how we will convert that data in and out of `DatabaseEntry` objects for database storage. We have also updated `MyDbs` to manage our databases for us. Now we write `ExampleDatabaseLoad` to actually put the inventory and vendor data into their respective databases. Because of the work that we have done so far, this application is actually fairly simple to write.

Example 8.5. ExampleDatabaseLoad.java

First we need the usual series of import statements:

```
// File: ExampleDatabaseLoad.java
package db.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.bind.EntryBinding;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
```

Next comes the class declaration and the private data members that we need for this class. Most of these are setting up default values for the program.

Note that two `DatabaseEntry` objects are instantiated here. We will reuse these for every database operation that this program performs. Also a `MyDbEnv` object is instantiated here. We can do this because its constructor never throws an exception. See Stored Class Catalog Management with MyDbs (page 88) for its implementation details.

Finally, the `inventory.txt` and `vendors.txt` file can be found in the GettingStarted examples directory along with the classes described in this extended example.

```
public class ExampleDatabaseLoad {

    private static String myDbsPath = "./";
    private static File inventoryFile = new File("./inventory.txt");
    private static File vendorsFile = new File("./vendors.txt");

    // DatabaseEntries used for loading records
    private static DatabaseEntry theKey = new DatabaseEntry();
    private static DatabaseEntry theData = new DatabaseEntry();

    // Encapsulates the databases.
    private static MyDbs myDbs = new MyDbs();
```

Next comes the `usage()` and `main()` methods. Notice the exception handling in the `main()` method. This is the only place in the application where we catch exceptions. For this reason, we must catch `DatabaseException` which is thrown by the `com.sleepycat.db.*` classes.

Also notice the call to `MyDbs.close()` in the `finally` block. This is the only place in the application where `MyDbs.close()` is called. `MyDbs.close()` is responsible for closing all open `Database` handles for you.

```
    private static void usage() {
        System.out.println("ExampleDatabaseLoad [-h <database home>]");
        System.out.println("      [-s <selections file>] [-v <vendors file>]");
        System.exit(-1);
    }

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

Next we write the `ExampleDatabaseLoad.run()` method. This method is responsible for initializing all objects. Because our environment and databases are all opened using the `MyDbs.setup()` method, `ExampleDatabaseLoad.run()` method is only responsible for calling `MyDbs.setup()` and then calling the `ExampleDatabaseLoad` methods that actually load the databases.

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

        myDbs.setup(myDbsPath); // path to the environment home

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

This next method loads the vendor database. This method uses serialization to convert the `Vendor` object to a `DatabaseEntry` object.

```
    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 database. The vendor's name is the
        // key, and the data is a Vendor class object.

        // Need a serial binding for the data
        EntryBinding dataBinding =
            new SerialBinding(myDbs.getClassCatalog(), Vendor.class);

        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]);

            // The key is the vendor's name.
            // ASSUMES THE VENDOR'S NAME IS UNIQUE!
            String vendorName = theVendor.getVendorName();
            try {
                theKey = new DatabaseEntry(vendorName.getBytes("UTF-8"));
            } catch (IOException willNeverOccur) {}
```

```
            // Convert the Vendor object to a DatabaseEntry object
            // using our SerialBinding
            dataBinding.objectToEntry(theVendor, theData);

            // Put it in the database.
            myDbs.getVendorDB().put(null, theKey, theData);
        }
    }
```

Now load the inventory database. This method uses our custom tuple binding (see InventoryBinding.java (page 87)) to convert the `Inventory` object to a `DatabaseEntry` object.

```
    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 database. The item's sku is the
        // key, and the data is an Inventory class object.

        // Need a tuple binding for the Inventory class.
        TupleBinding inventoryBinding = new InventoryBinding();

        for (int i = 0; i < inventoryArray.size(); i++) {
            String[] sArray = (String[])inventoryArray.get(i);
            String sku = sArray[1];
            try {
                theKey = new DatabaseEntry(sku.getBytes("UTF-8"));
            } catch (IOException willNeverOccur) {}

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

            // Place the Vendor object on the DatabaseEntry object using
            // our the tuple binding we implemented in
            // InventoryBinding.java
            inventoryBinding.objectToEntry(theInventory, theData);
```

```
            // Put it in the database. Note that this causes our
            // secondary database to be automatically updated for us.
            myDbs.getInventoryDB().put(null, theKey, theData);
        }
    }
```

The remainder of this application provides utility methods to read a flat text file into an array of strings and parse the command line options:

```
    private static void parseArgs(String args[]) {
        // Implementation omitted for brevity.
    }

    private List loadFile(File theFile, int numFields) {
        List records = new ArrayList();
        // Implementation omitted for brevity.
        return records;
    }

    protected ExampleDatabaseLoad() {}
}
```

From the perspective of this document, these things are relatively uninteresting. You can see how they are implemented by looking at `ExampleDatabaseLoad.java` in:

```
DB_INSTALL/examples_java/db/GettingStarted
```

where *DB_INSTALL* is the location where you placed your DB distribution.

Chapter 9. Using Cursors

Cursors provide a mechanism by which you can iterate over the records in a database. Using cursors, you can get, put, and delete database records. If a database allows duplicate records, then cursors are the easiest way that you can access anything other than the first record for a given key.

This chapter introduces cursors. It explains how to open and close them, how to use them to modify databases, and how to use them with duplicate records.

Opening and Closing Cursors

To use a cursor, you must open it using the `Database.openCursor()` method. When you open a cursor, you can optionally pass it a `CursorConfig` object to set cursor properties. The cursor properties that you can set allows you to control the isolation level that the cursor will obey. See the *Berkeley DB Getting Started with Transaction Processing* guide for more information.

For example:

```
package db.GettingStarted;

import com.sleepycat.db.Cursor;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseException;

import java.io.FileNotFoundException;

...
Database myDatabase = null;
Cursor myCursor = null;

try {
    myDatabase = new Database("myDB", null, null);

    myCursor = myDatabase.openCursor(null, null);
} catch (FileNotFoundException fnfe) {
    // Exception handling goes here ...
} catch (DatabaseException dbe) {
    // Exception handling goes here ...
}
```

To close the cursor, call the `Cursor.close()` method. Note that if you close a database that has cursors open in it, then it will throw an exception and close any open cursors for you. For best results, close your cursors from within a `finally` block.

```
package db.GettingStarted;

import com.sleepycat.db.Cursor;
import com.sleepycat.db.Database;
```

```
...
try {
    ...
} catch ... {
} finally {
    try {
        if (myCursor != null) {
            myCursor.close();
        }

        if (myDatabase != null) {
            myDatabase.close();
        }
    } catch(DatabaseException dbe) {
        System.err.println("Error in close: " + dbe.toString());
    }
}
```

Getting Records Using the Cursor

To iterate over database records, from the first record to the last, simply open the cursor and then use the `Cursor.getNext()` method. For example:

```
package db.GettingStarted;

import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.Cursor;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;

...

Cursor cursor = null;
try {
    ...
    Database myDatabase = null;
    // Database open omitted for brevity
    ...

    // Open the cursor.
    cursor = myDatabase.openCursor(null, null);

    // Cursors need a pair of DatabaseEntry objects to operate. These hold
    // the key and data found at any given position in the database.
    DatabaseEntry foundKey = new DatabaseEntry();
    DatabaseEntry foundData = new DatabaseEntry();
```

```
    // To iterate, just call getNext() until the last database record has been
    // read. All cursor operations return an OperationStatus, so just read
    // until we no longer see OperationStatus.SUCCESS
    while (cursor.getNext(foundKey, foundData, LockMode.DEFAULT) ==
        OperationStatus.SUCCESS) {
        // getData() on the DatabaseEntry objects returns the byte array
        // held by that object. We use this to get a String value. If the
        // DatabaseEntry held a byte array representation of some other data
        // type (such as a complex object) then this operation would look
        // considerably different.
        String keyString = new String(foundKey.getData(), "UTF-8");
        String dataString = new String(foundData.getData(), "UTF-8");
        System.out.println("Key | Data : " + keyString + " | " +
                           dataString + "");
    }
} catch (DatabaseException de) {
    System.err.println("Error accessing database." + de);
} finally {
    // Cursors must be closed.
    cursor.close();
}
```

To iterate over the database from the last record to the first, instantiate the cursor, and then use `Cursor.getPrev()` until you read the first record in the database. For example:

```
package db.GettingStarted;

import com.sleepycat.db.Cursor;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;

...

Cursor cursor = null;
Database myDatabase = null;
try {
    ...
    // Database open omitted for brevity
    ...

    // Open the cursor.
    cursor = myDatabase.openCursor(null, null);

    // Get the DatabaseEntry objects that the cursor will use.
    DatabaseEntry foundKey = new DatabaseEntry();
```

```
    DatabaseEntry foundData = new DatabaseEntry();

    // Iterate from the last record to the first in the database
    while (cursor.getPrev(foundKey, foundData, LockMode.DEFAULT) ==
        OperationStatus.SUCCESS) {

        String theKey = new String(foundKey.getData(), "UTF-8");
        String theData = new String(foundData.getData(), "UTF-8");
        System.out.println("Key | Data : " +  theKey + " | " + theData + "");
    }
} catch (DatabaseException de) {
    System.err.println("Error accessing database." + de);
} finally {
    // Cursors must be closed.
    cursor.close();
}
```

Searching for Records

You can use cursors to search for database records. You can search based on just a key, or you can search based on both the key and the data. You can also perform partial matches if your database supports sorted duplicate sets. In all cases, the key and data parameters of these methods are filled with the key and data values of the database record to which the cursor is positioned as a result of the search.

Also, if the search fails, then cursor's state is left unchanged and `OperationStatus.NOTFOUND` is returned.

The following `Cursor` methods allow you to perform database searches:

- `Cursor.getSearchKey()`

  Moves the cursor to the first record in the database with the specified key.

- `Cursor.getSearchKeyRange()`

  Identical to `Cursor.getSearchKey()` unless you are using the BTree access. In this case, the cursor moves to the first record in the database whose key is greater than or equal to the specified key. This comparison is determined by the comparator that you provide for the database. If no comparator is provided, then the default lexicographical sorting is used.

  For example, suppose you have database records that use the following Strings as keys:

  ```
  Alabama
  Alaska
  Arizona
  ```

  Then providing a search key of `Alaska` moves the cursor to the second key noted above. Providing a key of `Al` moves the cursor to the first key (`Alabama`), providing a search key of

`Alas` moves the cursor to the second key (`Alaska`), and providing a key of `Ar` moves the cursor to the last key (`Arizona`).

- `Cursor.getSearchBoth()`

  Moves the cursor to the first record in the database that uses the specified key and data.

- `Cursor.getSearchBothRange()`

  Moves the cursor to the first record in the database whose key matches the specified key and whose data is greater than or equal to the specified data. If the database supports duplicate records, then on matching the key, the cursor is moved to the duplicate record with the smallest data that is greater than or equal to the specified data.

  For example, suppose your database uses BTree and it has database records that use the following key/data pairs:

```
Alabama/Athens
Alabama/Florence
Alaska/Anchorage
Alaska/Fairbanks
Arizona/Avondale
Arizona/Florence
```

  then providing:

a search key of ...	and a search data of ...	moves the cursor to ...
Alaska	Fa	Alaska/Fairbanks
Arizona	Fl	Arizona/Florence
Alaska	An	Alaska/Anchorage

For example, assuming a database containing sorted duplicate records of U.S. States/U.S Cities key/data pairs (both as Strings), then the following code fragment can be used to position the cursor to any record in the database and print its key/data values:

```
package db.GettingStarted;

import com.sleepycat.db.Cursor;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;

...

// For this example, hard code the search key and data
String searchKey = "Alaska";
String searchData = "Fa";
```

```
Cursor cursor = null;
Database myDatabase = null;
try {
    ...
    // Database open omitted for brevity
    ...

    // Open the cursor.
    cursor = myDatabase.openCursor(null, null);

    DatabaseEntry theKey =
         new DatabaseEntry(searchKey.getBytes("UTF-8"));
    DatabaseEntry theData =
         new DatabaseEntry(searchData.getBytes("UTF-8"));

    // Open a cursor using a database handle
    cursor = myDatabase.openCursor(null, null);

    // Perform the search
    OperationStatus retVal = cursor.getSearchBothRange(theKey, theData,
                                                       LockMode.DEFAULT);
    // NOTFOUND is returned if a record cannot be found whose key
    // matches the search key AND whose data begins with the search data.
    if (retVal == OperationStatus.NOTFOUND) {
        System.out.println(searchKey + "/" + searchData +
                           " not matched in database " +
                           myDatabase.getDatabaseName());
    } else {
        // Upon completing a search, the key and data DatabaseEntry
        // parameters for getSearchBothRange() are populated with the
        // key/data values of the found record.
        String foundKey = new String(theKey.getData(), "UTF-8");
        String foundData = new String(theData.getData(), "UTF-8");
        System.out.println("Found record " + foundKey + "/" + foundData +
                           "for search key/data: " + searchKey +
                           "/" + searchData);
    }

} catch (Exception e) {
    // Exception handling goes here
} finally {
   // Make sure to close the cursor
   cursor.close();
}
```

Working with Duplicate Records

A record is a duplicate of another record if the two records share the same key. For duplicate records, only the data portion of the record is unique.

Duplicate records are supported only for the BTree or Hash access methods. For information on configuring your database to use duplicate records, see Allowing Duplicate Records (page 139).

If your database supports duplicate records, then it can potentially contain multiple records that share the same key. By default, normal database get operations will only return the first such record in a set of duplicate records. Typically, subsequent duplicate records are accessed using a cursor. The following `Cursor` methods are interesting when working with databases that support duplicate records:

- `Cursor.getNext()`, `Cursor.getPrev()`

  Shows the next/previous record in the database, regardless of whether it is a duplicate of the current record. For an example of using these methods, see Getting Records Using the Cursor (page 97).

- `Cursor.getSearchBothRange()`

  Useful for seeking the cursor to a specific record, regardless of whether it is a duplicate record. See Searching for Records (page 99) for more information.

- `Cursor.getNextNoDup()`, `Cursor.getPrevNoDup()`

  Gets the next/previous non-duplicate record in the database. This allows you to skip over all the duplicates in a set of duplicate records. If you call `Cursor.getPrevNoDup()`, then the cursor is positioned to the last record for the previous key in the database. For example, if you have the following records in your database:

  ```
  Alabama/Athens
  Alabama/Florence
  Alaska/Anchorage
  Alaska/Fairbanks
  Arizona/Avondale
  Arizona/Florence
  ```

  and your cursor is positioned to `Alaska/Fairbanks`, and you then call `Cursor.getPrevNoDup()`, then the cursor is positioned to Alabama/Florence. Similarly, if you call `Cursor.getNextNoDup()`, then the cursor is positioned to the first record corresponding to the next key in the database.

  If there is no next/previous key in the database, then `OperationStatus.NOTFOUND` is returned, and the cursor is left unchanged.

- Gets the next record that shares the current key. If the cursor is positioned at the last record in the duplicate set and you call `Cursor.getNextDup()`, then `OperationStatus.NOTFOUND` is returned and the cursor is left unchanged. Likewise, if you call `getPrevDup()` and the cursor

is positioned at the first record in the duplicate set, then `OperationStatus.NOTFOUND` is returned and the cursor is left unchanged.

- `Cursor.count()`

  Returns the total number of records that share the current key.

For example, the following code fragment positions a cursor to a key and displays it and all its duplicates. Note that the following code fragment assumes that the database contains only String objects for the keys and data.

```
package db.GettingStarted;

import com.sleepycat.db.Cursor;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;

...

Cursor cursor = null;
Database myDatabase = null;
try {
    ...
    // Database open omitted for brevity
    ...

    // Create DatabaseEntry objects
    // searchKey is some String.
    DatabaseEntry theKey = new DatabaseEntry(searchKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();

    // Open a cursor using a database handle
    cursor = myDatabase.openCursor(null, null);

    // Position the cursor
    // Ignoring the return value for clarity
    OperationStatus retVal = cursor.getSearchKey(theKey, theData,
                                                 LockMode.DEFAULT);

    // Count the number of duplicates. If the count is greater than 1,
    // print the duplicates.
    if (cursor.count() > 1) {
        while (retVal == OperationStatus.SUCCESS) {
            String keyString = new String(theKey.getData(), "UTF-8");
            String dataString = new String(theData.getData(), "UTF-8");
            System.out.println("Key | Data : " +  keyString + " | " +
                               dataString + "");
```

```
            retVal = cursor.getNextDup(theKey, theData, LockMode.DEFAULT);
        }
    }
} catch (Exception e) {
    // Exception handling goes here
} finally {
   // Make sure to close the cursor
   cursor.close();
}
```

Putting Records Using Cursors

You can use cursors to put records into the database. DB's behavior when putting records into the database differs depending on the flags that you use when writing the record, on the access method that you are using, and on whether your database supports sorted duplicates.

Note that when putting records to the database using a cursor, the cursor is positioned at the record you inserted.

- `Cursor.putNoDupData()`

  If the provided key already exists in the database, then this method returns `OperationStatus.KEYEXIST`.

  If the key does not exist, then the order that the record is put into the database is determined by the insertion order in use by the database. If a comparison function has been provided to the database, the record is inserted in its sorted location. Otherwise (assuming BTree), lexicographical sorting is used, with shorter items collating before longer items.

  This flag can only be used for the BTree and Hash access methods, and only if the database has been configured to support sorted duplicate data items (`DB_DUPSORT` was specified at database creation time).

  This flag cannot be used with the Queue or Recno access methods.

  For more information on duplicate records, see Allowing Duplicate Records (page 139).

- `Cursor.putNoOverwrite()`

  If the provided key already exists in the database, then this method returns .

  If the key does not exist, then the order that the record is put into the database is determined by the BTree (key) comparator in use by the database.

- `Cursor.putKeyFirst()`

  For databases that do not support duplicates, this method behaves exactly the same as if a default insertion was performed. If the database supports duplicate records, and a duplicate sort function has been specified, the inserted data item is added in its sorted location. If

the key already exists in the database and no duplicate sort function has been specified, the inserted data item is added as the first of the data items for that key.

- `Cursor.putKeyLast()`

  Behaves exactly as if `Cursor.putKeyFirst()` was used, except that if the key already exists in the database and no duplicate sort function has been specified, the inserted data item is added as the last of the data items for that key.

For example:

```
package db.GettingStarted;

import com.sleepycat.db.Cursor;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.OperationStatus;

...

// Create the data to put into the database
String key1str = "My first string";
String data1str = "My first data";
String key2str = "My second string";
String data2str = "My second data";
String data3str = "My third data";

Cursor cursor = null;
Database myDatabase = null;
try {
    ...
    // Database open omitted for brevity
    ...

    DatabaseEntry key1 = new DatabaseEntry(key1str.getBytes("UTF-8"));
    DatabaseEntry data1 = new DatabaseEntry(data1str.getBytes("UTF-8"));
    DatabaseEntry key2 = new DatabaseEntry(key2str.getBytes("UTF-8"));
    DatabaseEntry data2 = new DatabaseEntry(data2str.getBytes("UTF-8"));
    DatabaseEntry data3 = new DatabaseEntry(data3str.getBytes("UTF-8"));

    // Open a cursor using a database handle
    cursor = myDatabase.openCursor(null, null);

    // Assuming an empty database.

    OperationStatus retVal = cursor.put(key1, data1); // SUCCESS
    retVal = cursor.put(key2, data2); // SUCCESS
    retVal = cursor.put(key2, data3); // SUCCESS if dups allowed,
                                      // KEYEXIST if not.
```

```
} catch (Exception e) {
    // Exception handling goes here
} finally {
   // Make sure to close the cursor
   cursor.close();
}
```

Deleting Records Using Cursors

To delete a record using a cursor, simply position the cursor to the record that you want to delete and then call

For example:

```
package db.GettingStarted;

import com.sleepycat.db.Cursor;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;

...

Cursor cursor = null;
Database myDatabase = null;
try {
    ...
    // Database open omitted for brevity
    ...
    // Create DatabaseEntry objects
    // searchKey is some String.
    DatabaseEntry theKey = new DatabaseEntry(searchKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();

    // Open a cursor using a database handle
    cursor = myDatabase.openCursor(null, null);

    // Position the cursor. Ignoring the return value for clarity
    OperationStatus retVal = cursor.getSearchKey(theKey, theData,
                                                 LockMode.DEFAULT);

    // Count the number of records using the given key. If there is only
    // one, delete that record.
    if (cursor.count() == 1) {
            System.out.println("Deleting " +
                               new String(theKey.getData(), "UTF-8") +
                               "|" +
                               new String(theData.getData(), "UTF-8"));
```

```
            cursor.delete();
    }
} catch (Exception e) {
    // Exception handling goes here
} finally {
   // Make sure to close the cursor
   cursor.close();
}
```

Replacing Records Using Cursors

You replace the data for a database record by using `Cursor.putCurrent()`.

```
import com.sleepycat.db.Cursor;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;

...
Cursor cursor = null;
Database myDatabase = null;
try {
    ...
    // Database open omitted for brevity
    ...
    // Create DatabaseEntry objects
    // searchKey is some String.
    DatabaseEntry theKey = new DatabaseEntry(searchKey.getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();

    // Open a cursor using a database handle
    cursor = myDatabase.openCursor(null, null);

    // Position the cursor. Ignoring the return value for clarity
    OperationStatus retVal = cursor.getSearchKey(theKey, theData,
                                                 LockMode.DEFAULT);

    // Replacement data
    String replaceStr = "My replacement string";
    DatabaseEntry replacementData =
        new DatabaseEntry(replaceStr.getBytes("UTF-8"));
    cursor.putCurrent(replacementData);
} catch (Exception e) {
    // Exception handling goes here
} finally {
   // Make sure to close the cursor
   cursor.close();
}
```

Note that you cannot change a record's key using this method; the key parameter is always ignored when you replace a record.

When replacing the data portion of a record, if you are replacing a record that is a member of a sorted duplicates set, then the replacement will be successful only if the new record sorts identically to the old record. This means that if you are replacing a record that is a member of a sorted duplicates set, and if you are using the default lexicographic sort, then the replacement will fail due to violating the sort order. However, if you provide a custom sort routine that, for example, sorts based on just a few bytes out of the data item, then potentially you can perform a direct replacement and still not violate the restrictions described here.

Under these circumstances, if you want to replace the data contained by a duplicate record, and you are not using a custom sort routine, then delete the record and create a new record with the desired key and data.

Cursor Example

In Database Usage Example (page 84) we wrote an application that loaded two `Database` objects with vendor and inventory information. In this example, we will use those databases to display all of the items in the inventory database. As a part of showing any given inventory item, we will look up the vendor who can provide the item and show the vendor's contact information.

To do this, we create the `ExampleDatabaseRead` application. This application reads and displays all inventory records by:

1. Opening the inventory, vendor, and class catalog `Database` objects. We do this using the `MyDbs` class. See Stored Class Catalog Management with MyDbs (page 88) for a description of this class.

2. Obtaining a cursor from the inventory `Database`.

3. Steps through the `Database`, displaying each record as it goes.

4. To display the Inventory record, the custom tuple binding that we created in InventoryBinding.java (page 87) is used.

5. `Database.get()` is used to obtain the vendor that corresponds to the inventory item.

6. A serial binding is used to convert the `DatabaseEntry` returned by the `get()` to a Vendor object.

7. The contents of the Vendor object are displayed.

We implemented the `Vendor` class in Vendor.java (page 86). We implemented the `Inventory` class in Inventory.java (page 84).

The full implementation of `ExampleDatabaseRead` can be found in:

```
DB_INSTALL/examples_java/db/GettingStarted
```

where *DB_INSTALL* is the location where you placed your DB distribution.

Example 9.1. ExampleDatabaseRead.java

To begin, we import the necessary classes:

```
// file ExampleDatabaseRead.java
package db.GettingStarted;

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

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.db.Cursor;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;
```

Next we declare our class and set up some global variables. Note a `MyDbs` object is instantiated here. We can do this because its constructor never throws an exception. See Database Example (page 65) for its implementation details.

```
public class ExampleDatabaseRead {

    private static String myDbsPath = "./";

    // Encapsulates the database environment and databases.
    private static MyDbs myDbs = new MyDbs();

    private static TupleBinding inventoryBinding;
    private static EntryBinding vendorBinding;
```

Next we create the `ExampleDatabaseRead.usage()` and `ExampleDatabaseRead.main()` methods. We perform almost all of our exception handling from `ExampleDatabaseRead.main()`, and so we must catch `DatabaseException` because the `com.sleepycat.db.*` APIs throw them.

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

    public static void main(String args[]) {
        ExampleDatabaseRead edr = new ExampleDatabaseRead();
        try {
            edr.run(args);
        } catch (DatabaseException dbe) {
            System.err.println("ExampleDatabaseRead: " + dbe.toString());
            dbe.printStackTrace();
```

```
        } finally {
            myDbs.close();
        }
        System.out.println("All done.");
    }
```

In `ExampleDatabaseRead.run()`, we call `MyDbs.setup()` to open our databases. Then we create the bindings that we need for using our data objects with `DatabaseEntry` objects.

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

        myDbs.setup(myDbsPath);

        // Setup our bindings.
        inventoryBinding = new InventoryBinding();
        vendorBinding =
             new SerialBinding(myDbs.getClassCatalog(),
                               Vendor.class);

        showAllInventory();
    }
```

Now we write the loop that displays the `Inventory` records. We do this by opening a cursor on the inventory database and iterating over all its contents, displaying each as we go.

```
    private void showAllInventory()
        throws DatabaseException {
        // Get a cursor
        Cursor cursor = myDbs.getInventoryDB().openCursor(null, null);

        // DatabaseEntry objects used for reading records
        DatabaseEntry foundKey = new DatabaseEntry();
        DatabaseEntry foundData = new DatabaseEntry();

        try { // always want to make sure the cursor gets closed
            while (cursor.getNext(foundKey, foundData,
                        LockMode.DEFAULT) == OperationStatus.SUCCESS) {
                Inventory theInventory =
                    (Inventory)inventoryBinding.entryToObject(foundData);
                displayInventoryRecord(foundKey, theInventory);
            }
        } catch (Exception e) {
            System.err.println("Error on inventory cursor:");
            System.err.println(e.toString());
            e.printStackTrace();
        } finally {
            cursor.close();
```

```
        }
    }
```

We use `ExampleDatabaseRead.displayInventoryRecord()` to actually show the record. This method first displays all the relevant information from the retrieved Inventory object. It then uses the vendor database to retrieve and display the vendor. Because the vendor database is keyed by vendor name, and because each inventory object contains this key, it is trivial to retrieve the appropriate vendor record.

```
    private void displayInventoryRecord(DatabaseEntry theKey,
                                        Inventory theInventory)
        throws DatabaseException {

        String theSKU = new String(theKey.getData(), "UTF-8");
        System.out.println(theSKU + ":");
        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: ");

        DatabaseEntry searchKey = null;
        try {
            searchKey =
                new DatabaseEntry(theInventory.getVendor().getBytes("UTF-8"));
        } catch (IOException willNeverOccur) {}
        DatabaseEntry foundVendor = new DatabaseEntry();

        if (myDbs.getVendorDB().get(null, searchKey, foundVendor,
                LockMode.DEFAULT) != OperationStatus.SUCCESS) {
            System.out.println("Could not find vendor: " +
                theInventory.getVendor() + ".");
            System.exit(-1);
        } else {
            Vendor theVendor =
                (Vendor)vendorBinding.entryToObject(foundVendor);
            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 remainder of this application provides a utility method used to parse the command line options. From the perspective of this document, this is relatively uninteresting. You can see how this is implemented by looking at:

```
DB_INSTALL/examples_java/db/GettingStarted
```

where *DB_INSTALL* is the location where you placed your DB distribution.

Chapter 10. Secondary Databases

Usually you find database records by means of the record's key. However, the key that you use for your record will not always contain the information required to provide you with rapid access to the data that you want to retrieve. For example, suppose your `Database` contains records related to users. The key might be a string that is some unique identifier for the person, such as a user ID. Each record's data, however, would likely contain a complex object containing details about people such as names, addresses, phone numbers, and so forth. While your application may frequently want to query a person by user ID (that is, by the information stored in the key), it may also on occasion want to locate people by, say, their name.

Rather than iterate through all of the records in your database, examining each in turn for a given person's name, you create indexes based on names and then just search that index for the name that you want. You can do this using secondary databases. In DB, the `Database` that contains your data is called a *primary database*. A database that provides an alternative set of keys to access that data is called a *secondary database* In a secondary database, the keys are your alternative (or secondary) index, and the data corresponds to a primary record's key.

You create a secondary database by using a `SecondaryConfig` class object to identify an implementation of a `SecondaryKeyCreator` class object that is used to create keys based on data found in the primary database. You then pass this `SecondaryConfig` object to the `SecondaryDatabase` constructor.

Once opened, DB manages secondary databases for you. Adding or deleting records in your primary database causes DB to update the secondary as necessary. Further, changing a record's data in the primary database may cause DB to modify a record in the secondary, depending on whether the change forces a modification of a key in the secondary database.

Note that you can not write directly to a secondary database. To change the data referenced by a `SecondaryDatabase` record, modify the primary database instead. The exception to this rule is that delete operations are allowed on the `SecondaryDatabase` object. See Deleting Secondary Database Records (page 121) for more information.

Secondary database records are updated/created by DB only if the `SecondaryKeyCreator.createSecondaryKey()` method returns `true`. If `false` is returned, then DB will not add the key to the secondary database, and in the event of a record update it will remove any existing key.

See Implementing Key Creators (page 116) for more information on this interface and method.

When you read a record from a secondary database, DB automatically returns the data and optionally the key from the corresponding record in the primary database.

Opening and Closing Secondary Databases

You manage secondary database opens and closes using the `SecondaryDatabase` constructor. Just as is the case with primary databases, you must provide the `SecondaryDatabase()` constructor with the database's name and, optionally, other properties such as whether duplicate

records are allowed, or whether the secondary database can be created on open. In addition, you must also provide:

- A handle to the primary database that this secondary database is indexing. Note that this means that secondary databases are maintained only for the specified `Database` handle. If you open the same `Database` multiple times for write (such as might occur when opening a database for read-only and read-write in the same application), then you should open the `SecondaryDatabase` for each such `Database` handle.
- A `SecondaryConfig` object that provides properties specific to a secondary database. The most important of these is used to identify the key creator for the database. The key creator is responsible for generating keys for the secondary database. See Secondary Database Properties (page 120) for details.

Primary databases *must not* support duplicate records. Secondary records point to primary records using the primary key, so that key must be unique.

So to open (create) a secondary database, you:

1. Open your primary database.
2. Instantiate your key creator.
3. Instantiate your `SecondaryConfig` object.
4. Set your key creator object on your `SecondaryConfig` object.
5. Open your secondary database, specifying your primary database and your `SecondaryConfig` at that time.

For example:

```
package db.GettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseType;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.SecondaryDatabase;
import com.sleepycat.db.SecondaryConfig;

import java.io.FileNotFoundException;

...

DatabaseConfig myDbConfig = new DatabaseConfig();
myDbConfig.setAllowCreate(true);
myDbConfig.setType(DatabaseType.BTREE);
```

```
SecondaryConfig mySecConfig = new SecondaryConfig();
mySecConfig.setAllowCreate(true);
mySecConfig.setType(DatabaseType.BTREE);
// Duplicates are frequently required for secondary databases.
mySecConfig.setSortedDuplicates(true);

// Open the primary
Database myDb = null;
SecondaryDatabase mySecDb = null;
try {
    String dbName = "myPrimaryDatabase";

    myDb = new Database(dbName, null, myDbConfig);

    // A fake tuple binding that is not actually implemented anywhere.
    // The tuple binding is dependent on the data in use.
    // Tuple bindings are described earlier in this manual.
    TupleBinding myTupleBinding = new MyTupleBinding();

    // Open the secondary.
    // Key creators are described in the next section.
    FullNameKeyCreator keyCreator = new FullNameKeyCreator(myTupleBinding);

    // Get a secondary object and set the key creator on it.
    mySecConfig.setKeyCreator(keyCreator);

    // Perform the actual open
    String secDbName = "mySecondaryDatabase";
    mySecDb = new SecondaryDatabase(secDbName, null, myDb, mySecConfig);
} catch (DatabaseException de) {
    // Exception handling goes here ...
} catch (FileNotFoundException fnfe) {
    // Exception handling goes here ...
}
```

To close a secondary database, call its close() method. Note that for best results, you should close all the secondary databases associated with a primary database before closing the primary.

For example:

```
try {
    if (mySecDb != null) {
        mySecDb.close();
    }

    if (myDb != null) {
        myDb.close();
    }
} catch (DatabaseException dbe) {
```

```
    // Exception handling goes here
}
```

Implementing Key Creators

You must provide every secondary database with a class that creates keys from primary records. You identify this class using the `SecondaryConfig.setKeyCreator()` method.

You can create keys using whatever data you want. Typically you will base your key on some information found in a record's data, but you can also use information found in the primary record's key. How you build your keys is entirely dependent upon the nature of the index that you want to maintain.

You implement a key creator by writing a class that implements the `SecondaryKeyCreator` interface. This interface requires you to implement the `SecondaryKeyCreator.createSecondaryKey()` method.

One thing to remember when implementing this method is that you will need a way to extract the necessary information from the data's `DatabaseEntry` and/or the key's `DatabaseEntry` that are provided on calls to this method. If you are using complex objects, then you are probably using the Bind APIs to perform this conversion. The easiest thing to do is to instantiate the `EntryBinding` or `TupleBinding` that you need to perform the conversion, and then provide this to your key creator's constructor. The Bind APIs are introduced in Using the BIND APIs (page 73).

`SecondaryKeyCreator.createSecondaryKey()` returns a boolean. A return value of `false` indicates that no secondary key exists, and therefore no record should be added to the secondary database for that primary record. If a record already exists in the secondary database, it is deleted.

For example, suppose your primary database uses the following class for its record data:

```
package db.GettingStarted;

public class PersonData {
    private String userID;
    private String surname;
    private String familiarName;

    public PersonData(String userID, String surname, String familiarName) {
        this.userID = userID;
        this.surname = surname;
        this.familiarName = familiarName;
    }

    public String getUserID() {
        return userID;
    }

    public String getSurname() {
        return surname;
    }
```

```
    public String getFamiliarName() {
        return familiarName;
    }
}
```

Also, suppose that you have created a custom tuple binding, `PersonDataBinding`, that you use to convert `PersonData` objects to and from `DatabaseEntry` objects. (Custom tuple bindings are described in Custom Tuple Bindings (page 81).)

Finally, suppose you want a secondary database that is keyed based on the person's full name.

Then in this case you might create a key creator as follows:

```
package db.GettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.db.SecondaryKeyCreator;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.SecondaryDatabase;

import java.io.IOException;

public class FullNameKeyCreator implements SecondaryKeyCreator {

    private TupleBinding theBinding;

    public FullNameKeyCreator(TupleBinding theBinding1) {
            theBinding = theBinding1;
    }

    public boolean createSecondaryKey(SecondaryDatabase secDb,
                                      DatabaseEntry keyEntry,
                                      DatabaseEntry dataEntry,
                                      DatabaseEntry resultEntry) {

        try {
            PersonData pd =
                (PersonData) theBinding.entryToObject(dataEntry);
                String fullName = pd.getFamiliarName() + " " +
                    pd.getSurname();
                resultEntry.setData(fullName.getBytes("UTF-8"));
        } catch (IOException willNeverOccur) {}
        return true;
    }
}
```

Finally, you use this key creator as follows:

```
package db.GettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;

import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.DatabaseType;
import com.sleepycat.db.SecondaryDatabase;
import com.sleepycat.db.SecondaryConfig;

import java.io.FileNotFoundException;

...
Database myDb = null;
SecondaryDatabase mySecDb = null;
try {
    // Primary database open omitted for brevity
...

    TupleBinding myDataBinding = new MyTupleBinding();
    FullNameKeyCreator fnkc = new FullNameKeyCreator(myDataBinding);

    SecondaryConfig mySecConfig = new SecondaryConfig();
    mySecConfig.setKeyCreator(fnkc);
    mySecConfig.setType(DatabaseType.BTREE);

    //Perform the actual open
    String secDbName = "mySecondaryDatabase";
    mySecDb = new SecondaryDatabase(secDbName, null, myDb, mySecConfig);
} catch (DatabaseException de) {
    // Exception handling goes here
} catch (FileNotFoundException fnfe) {
    // Exception handling goes here
} finally {
    try {
        if (mySecDb != null) {
            mySecDb.close();
        }

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

Working with Multiple Keys

Until now we have only discussed indexes as if there is a one-to-one relationship between the secondary key and the primary database record. In fact, it is possible to generate multiple keys for any given record, provided that you take appropriate steps in your key creator to do so.

For example, suppose you had a database that contained information about books. Suppose further that you sometimes want to look up books by author. Because sometimes books have multiple authors, you may want to return multiple secondary keys for every book that you index.

To do this, you write a key creator that implements `SecondaryMultiKeyCreator` instead of `SecondaryKeyCreator`. The key difference between the two is that `SecondaryKeyCreator` uses a single `DatabaseEntry` object as the result, while `SecondaryMultiKeyCreator` returns a set of `DatabaseEntry` objects (using `java.util.Set`). Also, you assign the `SecondaryMultiKeyCreator` implementation using `SecondaryConfig.setMultiKeyCreator()` instead of `SecondaryConfig.setKeyCreator()`.

For example:

```
package db.GettingStarted;

import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.SecondaryDatabase;
import com.sleepycat.db.SecondaryMultiKeyCreator;

import java.util.HashSet;
import java.util.Set;

public class MyMultiKeyCreator implements SecondaryMultiKeyCreator {

    // Constructor not implemented. How this is implemented depends on
    // how you want to extract the data for your keys.
    MyMultiKeyCreator() {
        ...
    }

    // Abstract method that we must implement
    public void createSecondaryKeys(SecondaryDatabase secDb,
        DatabaseEntry keyEntry,    // From the primary
        DatabaseEntry dataEntry,   // From the primary
        Set results)               // Results set
        throws DatabaseException {

        try {
            // Create your keys, adding each to the set

            // Creation of key 'a' not shown
```

```
            results.add(a)

            // Creation of key 'b' not shown
            results.add(b)

        } catch (IOException willNeverOccur) {}
    }
}
```

Secondary Database Properties

Secondary databases accept `SecondaryConfig` objects. `SecondaryConfig` is a subclass of `DatabaseConfig`, so it can manage all of the same properties as does `DatabaseConfig`. See Database Properties (page 60) for more information.

In addition to the `DatabaseConfig` properties, `SecondaryConfig` also allows you to manage the following properties:

- `SecondaryConfig.setAllowPopulate()`

  If true, the secondary database can be auto-populated. This means that on open, if the secondary database is empty then the primary database is read in its entirety and additions/modifications to the secondary's records occur automatically.

- `SecondaryConfig.setKeyCreator()`

  Identifies the key creator object to be used for secondary key creation. See Implementing Key Creators (page 116) for more information.

Reading Secondary Databases

Like a primary database, you can read records from your secondary database either by using the `SecondaryDatabase.get()` method, or by using a `SecondaryCursor`. The main difference between reading secondary and primary databases is that when you read a secondary database record, the secondary record's data is not returned to you. Instead, the primary key and data corresponding to the secondary key are returned to you.

For example, assuming your secondary database contains keys related to a person's full name:

```
package db.GettingStarted;

import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;
import com.sleepycat.db.SecondaryDatabase;

...
SecondaryDatabase mySecondaryDatabase = null;
try {
    // Omitting all database opens
```

```
    ...

    String searchName = "John Doe";
    DatabaseEntry searchKey =
        new DatabaseEntry(searchName.getBytes("UTF-8"));
    DatabaseEntry primaryKey = new DatabaseEntry();
    DatabaseEntry primaryData = new DatabaseEntry();

    // Get the primary key and data for the user 'John Doe'.
    OperationStatus retVal = mySecondaryDatabase.get(null, searchKey,
                                                     primaryKey,
                                                     primaryData,
                                                     LockMode.DEFAULT);
} catch (Exception e) {
    // Exception handling goes here
}
```

Note that, just like `Database.get()`, if your secondary database supports duplicate records then `SecondaryDatabase.get()` only return the first record found in a matching duplicates set. If you want to see all the records related to a specific secondary key, then use a `SecondaryCursor` (described in Using Secondary Cursors (page 122)).

Deleting Secondary Database Records

In general, you will not modify a secondary database directly. In order to modify a secondary database, you should modify the primary database and simply allow DB to manage the secondary modifications for you.

However, as a convenience, you can delete `SecondaryDatabase` records directly. Doing so causes the associated primary key/data pair to be deleted. This in turn causes DB to delete all `SecondaryDatabase` records that reference the primary record.

You can use the `SecondaryDatabase.delete()` method to delete a secondary database record. Note that if your `SecondaryDatabase` contains duplicate records, then deleting a record from the set of duplicates causes all of the duplicates to be deleted as well.

`SecondaryDatabase.delete()` causes the previously described delete operations to occur only if the primary database is opened for write access.

For example:

```
package db.GettingStarted;

import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.OperationStatus;
import com.sleepycat.db.SecondaryDatabase;

...
try {
```

```
    SecondaryDatabase mySecondaryDatabase = null;
    // Omitting all database opens
    ...

    String searchName = "John Doe";
    DatabaseEntry searchKey =
        new DatabaseEntry(searchName.getBytes("UTF-8"));

    // Delete the first secondary record that uses "John Doe" as
    // a key. This causes the primary record referenced by this secondary
    // record to be deleted.
    OperationStatus retVal = mySecondaryDatabase.delete(null, searchKey);
} catch (Exception e) {
    // Exception handling goes here
}
```

Using Secondary Cursors

Just like cursors on a primary database, you can use secondary cursors to iterate over the records in a secondary database. Like normal cursors, you can also use secondary cursors to search for specific records in a database, to seek to the first or last record in the database, to get the next duplicate record, and so forth. For a complete description on cursors and their capabilities, see Using Cursors (page 96).

However, when you use secondary cursors:

- Any data returned is the data contained on the primary database record referenced by the secondary record.
- `SecondaryCursor.getSearchBoth()` and related methods do not search based on a key/data pair. Instead, you search based on a secondary key and a primary key. The data returned is the primary data that most closely matches the two keys provided for the search.

For example, suppose you are using the databases, classes, and key creators described in Implementing Key Creators (page 116). Then the following searches for a person's name in the secondary database, and deletes all secondary and primary records that use that name.

```
package db.GettingStarted;

import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;
import com.sleepycat.db.SecondaryDatabase;
import com.sleepycat.db.SecondaryCursor;

...
try {
    SecondaryDatabase mySecondaryDatabase = null;
    // Database opens omitted for brevity
    ...
```

```
    String secondaryName = "John Doe";
    DatabaseEntry secondaryKey =
        new DatabaseEntry(secondaryName.getBytes("UTF-8"));

    DatabaseEntry foundData = new DatabaseEntry();

    SecondaryCursor mySecCursor =
        mySecondaryDatabase.openSecondaryCursor(null, null);

    OperationStatus retVal = mySecCursor.getSearchKey(secondaryKey,
                                                      foundData,
                                                      LockMode.DEFAULT);
    while (retVal == OperationStatus.SUCCESS) {
        mySecCursor.delete();
        retVal = mySecCursor.getNextDup(secondaryKey,
                                        foundData,
                                        LockMode.DEFAULT);
    }
} catch (Exception e) {
    // Exception handling goes here
}
```

Database Joins

If you have two or more secondary databases associated with a primary database, then you can retrieve primary records based on the intersection of multiple secondary entries. You do this using a `JoinCursor`.

Throughout this document we have presented a class that stores inventory information on grocery That class is fairly simple with a limited number of data members, few of which would be interesting from a query perspective. But suppose, instead, that we were storing information on something with many more characteristics that can be queried, such as an automobile. In that case, you may be storing information such as color, number of doors, fuel mileage, automobile type, number of passengers, make, model, and year, to name just a few.

In this case, you would still likely be using some unique value to key your primary entries (in the United States, the automobile's VIN would be ideal for this purpose). You would then create a class that identifies all the characteristics of the automobiles in your inventory. You would also have to create some mechanism by which you would move instances of this class in and out of Java `byte` arrays. We described the concepts and mechanisms by which you can perform these activities in Database Records (page 68).

To query this data, you might then create multiple secondary databases, one for each of the characteristics that you want to query. For example, you might create a secondary for color, another for number of doors, another for number of passengers, and so forth. Of course, you will need a unique key creator for each such secondary database. You do all of this using the concepts and techniques described throughout this chapter.

Once you have created this primary database and all interesting secondaries, what you have is the ability to retrieve automobile records based on a single characteristic. You can, for example, find all the automobiles that are red. Or you can find all the automobiles that have four doors. Or all the automobiles that are minivans.

The next most natural step, then, is to form compound queries, or joins. For example, you might want to find all the automobiles that are red, and that were built by Toyota, and that are minivans. You can do this using a `JoinCursor` class instance.

Using Join Cursors

To use a join cursor:

- Open two or more secondary cursors. These cursors for secondary databases that are associated with the same primary database.
- Position each such cursor to the secondary key value in which you are interested. For example, to build on the previous description, the cursor for the color database is positioned to the `red` records while the cursor for the model database is positioned to the `minivan` records, and the cursor for the make database is positioned to `Toyota`.
- Create an array of secondary cursors, and place in it each of the cursors that are participating in your join query.
- Obtain a join cursor. You do this using the `Database.join()` method. You must pass this method the array of secondary cursors that you opened and positioned in the previous steps.
- Iterate over the set of matching records using `JoinCursor.getNext()` until `OperationStatus` is not `SUCCESS`.
- Close your join cursor.
- If you are done with them, close all your secondary cursors.

For example:

```
package db.GettingStarted;

import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.JoinCursor;
import com.sleepycat.db.LockMode;
import com.sleepycat.db.OperationStatus;
import com.sleepycat.db.SecondaryCursor;
import com.sleepycat.db.SecondaryDatabase;

...

// Database and secondary database opens omitted for brevity.
// Assume a primary database handle:
```

```
//   automotiveDB
// Assume 3 secondary database handles:
//   automotiveColorDB  -- index based on automobile color
//   automotiveTypeDB  -- index based on automobile type
//   automotiveMakeDB   -- index based on the manufacturer
Database automotiveDB = null;
SecondaryDatabase automotiveColorDB = null;
SecondaryDatabase automotiveTypeDB = null;
SecondaryDatabase automotiveMakeDB = null;

// Query strings:
String theColor = "red";
String theType = "minivan";
String theMake = "Toyota";

// Secondary cursors used for the query:
SecondaryCursor colorSecCursor = null;
SecondaryCursor typeSecCursor = null;
SecondaryCursor makeSecCursor = null;

// The join cursor
JoinCursor joinCursor = null;

// These are needed for our queries
DatabaseEntry foundKey = new DatabaseEntry();
DatabaseEntry foundData = new DatabaseEntry();

// All cursor operations are enclosed in a try block to ensure that they
// get closed in the event of an exception.

try {
    // Database entries used for the query:
    DatabaseEntry color = new DatabaseEntry(theColor.getBytes("UTF-8"));
    DatabaseEntry type = new DatabaseEntry(theType.getBytes("UTF-8"));
    DatabaseEntry make = new DatabaseEntry(theMake.getBytes("UTF-8"));

    colorSecCursor = automotiveColorDB.openSecondaryCursor(null, null);
    typeSecCursor = automotiveTypeDB.openSecondaryCursor(null, null);
    makeSecCursor = automotiveMakeDB.openSecondaryCursor(null, null);

    // Position all our secondary cursors to our query values.
    OperationStatus colorRet =
        colorSecCursor.getSearchKey(color, foundData, LockMode.DEFAULT);
    OperationStatus typeRet =
        typeSecCursor.getSearchKey(type, foundData, LockMode.DEFAULT);
    OperationStatus makeRet =
        makeSecCursor.getSearchKey(make, foundData, LockMode.DEFAULT);

    // If all our searches returned successfully, we can proceed
```

```
    if (colorRet == OperationStatus.SUCCESS &&
        typeRet == OperationStatus.SUCCESS &&
        makeRet == OperationStatus.SUCCESS) {

        // Get a secondary cursor array and populate it with our
        // positioned cursors
        SecondaryCursor[] cursorArray = {colorSecCursor,
                                         typeSecCursor,
                                         makeSecCursor};

        // Create the join cursor
        joinCursor = automotiveDB.join(cursorArray, null);

        // Now iterate over the results, handling each in turn
        while (joinCursor.getNext(foundKey, foundData, LockMode.DEFAULT) ==
                        OperationStatus.SUCCESS) {

            // Do something with the key and data retrieved in
            // foundKey and foundData
        }
    }
} catch (DatabaseException dbe) {
    // Error reporting goes here
} catch (Exception e) {
    // Error reporting goes here
} finally {
    try {
        // Make sure to close out all our cursors
        if (colorSecCursor != null) {
            colorSecCursor.close();
        }
        if (typeSecCursor != null) {
            typeSecCursor.close();
        }
        if (makeSecCursor != null) {
            makeSecCursor.close();
        }
        if (joinCursor != null) {
            joinCursor.close();
        }
    } catch (DatabaseException dbe) {
        // Error reporting goes here
    }
}
```

JoinCursor Properties

You can set `JoinCursor` properties using the `JoinConfig` class. Currently there is just one property that you can set:

- `JoinConfig.setNoSort()`

  Specifies whether automatic sorting of input cursors is disabled. The cursors are sorted from the one that refers to the least number of data items to the one that refers to the most.

  If the data is structured so that cursors with many data items also share many common elements, higher performance will result from listing those cursors before cursors with fewer data items. Turning off sorting permits applications to specify cursors in the proper order given this scenario.

  The default value is `false` (automatic cursor sorting is performed).

  For example:

```
// All database and environments omitted
JoinConfig config = new JoinConfig();
config.setNoSort(true);
JoinCursor joinCursor = myDb.join(cursorArray, config);
```

Secondary Database Example

In previous chapters in this book, we built applications that load and display several DB databases. In this example, we will extend those examples to use secondary databases. Specifically:

- In Stored Class Catalog Management with MyDbs (page 88) we built a class that we can use to open several `Database` objects. In Opening Secondary Databases with MyDbs (page 128) we will extend that class to also open and manage a `SecondaryDatabase`.
- In Cursor Example (page 108) we built an application to display our inventory database (and related vendor information). In Using Secondary Databases with ExampleDatabaseRead (page 132) we will extend that application to show inventory records based on the index we cause to be loaded using `ExampleDatabaseLoad`.

Before we can use a secondary database, we must implement a class to extract secondary keys for us. We use `ItemNameKeyCreator` for this purpose.

Example 10.1. ItemNameKeyCreator.java

This class assumes the primary database uses `Inventory` objects for the record data. The `Inventory` class is described in Inventory.java (page 84).

In our key creator class, we make use of a custom tuple binding called `InventoryBinding`. This class is described in InventoryBinding.java (page 87).

You can find `InventoryBinding.java` in:

```
DB_INSTALL/examples_java/db/GettingStarted
```

where *DB_INSTALL* is the location where you placed your DB distribution.

```
package db.GettingStarted;

import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.SecondaryDatabase;
import com.sleepycat.db.SecondaryKeyCreator;
import com.sleepycat.bind.tuple.TupleBinding;

import java.io.IOException;

public class ItemNameKeyCreator implements SecondaryKeyCreator {

    private TupleBinding theBinding;

    // Use the constructor to set the tuple binding
    ItemNameKeyCreator(TupleBinding binding) {
        theBinding = binding;
    }

    // Abstract method that we must implement
    public boolean createSecondaryKey(SecondaryDatabase secDb,
        DatabaseEntry keyEntry,    // From the primary
        DatabaseEntry dataEntry,   // From the primary
        DatabaseEntry resultEntry) // set the key data on this.
        throws DatabaseException {

        try {
            // Convert dataEntry to an Inventory object
            Inventory inventoryItem =
                (Inventory) theBinding.entryToObject(dataEntry);
            // Get the item name and use that as the key
            String theItem = inventoryItem.getItemName();
            resultEntry.setData(theItem.getBytes("UTF-8"));
        } catch (IOException willNeverOccur) {}

        return true;
    }
}
```

Now that we have a key creator, we can use it to generate keys for a secondary database. We will now extend `MyDbs` to manage a secondary database, and to use `ItemNameKeyCreator` to generate keys for that secondary database.

Opening Secondary Databases with MyDbs

In Stored Class Catalog Management with MyDbs (page 88) we built `MyDbs` as an example of a class that encapsulates `Database` opens and closes. We will now extend that class to manage a `SecondaryDatabase`.

Example 10.2. SecondaryDatabase Management with MyDbs

We start by importing two additional classes needed to support secondary databases. We also add a global variable to use as a handle for our secondary database.

```
// File MyDbs.java
package db.GettingStarted;

import java.io.FileNotFoundException;

import com.sleepycat.bind.serial.StoredClassCatalog;
import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.DatabaseType;
import com.sleepycat.db.SecondaryConfig;
import com.sleepycat.db.SecondaryDatabase;

public class MyDbs {

    // The databases that our application uses
    private Database vendorDb = null;
    private Database inventoryDb = null;
    private Database classCatalogDb = null;
    private SecondaryDatabase itemNameIndexDb = null;

    private String vendordb = "VendorDB.db";
    private String inventorydb = "InventoryDB.db";
    private String classcatalogdb = "ClassCatalogDB.db";
    private String itemnameindexdb = "ItemNameIndexDB.db";

    // Needed for object serialization
    private StoredClassCatalog classCatalog;

    // Our constructor does nothing
    public MyDbs() {}
```

Next we update the `MyDbs.setup()` method to open the secondary database. As a part of this, we have to pass an `ItemNameKeyCreator` object on the call to open the secondary database. Also, in order to instantiate `ItemNameKeyCreator`, we need an `InventoryBinding` object (we described this class in InventoryBinding.java (page 87)). We do all this work together inside of `MyDbs.setup()`.

```
    public void setup(String databasesHome)
        throws DatabaseException {
        DatabaseConfig myDbConfig = new DatabaseConfig();
        SecondaryConfig mySecConfig = new SecondaryConfig();

        myDbConfig.setErrorStream(System.err);
```

```
        mySecConfig.setErrorStream(System.err);
        myDbConfig.setErrorPrefix("MyDbs");
        mySecConfig.setErrorPrefix("MyDbs");
        myDbConfig.setType(DatabaseType.BTREE);
        mySecConfig.setType(DatabaseType.BTREE);
        myDbConfig.setAllowCreate(true);
        mySecConfig.setAllowCreate(true);

        // Now open, or create and open, our databases
        // Open the vendors and inventory databases
        try {
            vendordb = databasesHome + "/" + vendordb;
            vendorDb = new Database(vendordb,
                                    null,
                                    myDbConfig);

            inventorydb = databasesHome + "/" + inventorydb;
            inventoryDb = new Database(inventorydb,
                                       null,
                                       myDbConfig);

            // Open the class catalog db. This is used to
            // optimize class serialization.
            classcatalogdb = databasesHome + "/" + classcatalogdb;
            classCatalogDb = new Database(classcatalogdb,
                                          null,
                                          myDbConfig);
        } catch(FileNotFoundException fnfe) {
            System.err.println("MyDbs: " + fnfe.toString());
            System.exit(-1);
        }

        // Create our class catalog
        classCatalog = new StoredClassCatalog(classCatalogDb);

        // Need a tuple binding for the Inventory class.
        // We use the InventoryBinding class
        // that we implemented for this purpose.
        TupleBinding inventoryBinding = new InventoryBinding();

        // Open the secondary database. We use this to create a
        // secondary index for the inventory database

        // We want to maintain an index for the inventory entries based
        // on the item name. So, instantiate the appropriate key creator
        // and open a secondary database.
        ItemNameKeyCreator keyCreator =
            new ItemNameKeyCreator(new InventoryBinding());
```

```
        // Set up additional secondary properties
        // Need to allow duplicates for our secondary database
        mySecConfig.setSortedDuplicates(true);
        mySecConfig.setAllowPopulate(true); // Allow autopopulate
        mySecConfig.setKeyCreator(keyCreator);
        // Now open it
        try {
            itemnameindexdb = databasesHome + "/" + itemnameindexdb;
            itemNameIndexDb = new SecondaryDatabase(itemnameindexdb,
                                                    null,
                                                    inventoryDb,
                                                    mySecConfig);
        } catch(FileNotFoundException fnfe) {
            System.err.println("MyDbs: " + fnfe.toString());
            System.exit(-1);
        }
    }
```

Next we need an additional getter method for returning the secondary database.

```
    public SecondaryDatabase getNameIndexDB() {
        return itemNameIndexDb;
    }
```

Finally, we need to update the `MyDbs.close()` method to close the new secondary database. We want to make sure that the secondary is closed before the primaries. While this is not necessary for this example because our closes are single-threaded, it is still a good habit to adopt.

```
    public void close() {
        try {
            if (itemNameIndexDb != null) {
                itemNameIndexDb.close();
            }

            if (vendorDb != null) {
                vendorDb.close();
            }

            if (inventoryDb != null) {
                inventoryDb.close();
            }

            if (classCatalogDb != null) {
                classCatalogDb.close();
            }

        } catch(DatabaseException dbe) {
            System.err.println("Error closing MyDbs: " +
```

```
                                    dbe.toString());
            System.exit(-1);
        }
    }
}
```

That completes our update to `MyDbs`. You can find the complete class implementation in:

```
DB_INSTALL/examples_java/db/GettingStarted
```

where *DB_INSTALL* is the location where you placed your DB distribution.

Using Secondary Databases with ExampleDatabaseRead

Because we performed all our secondary database configuration management in `MyDbs`, we do not need to modify `ExampleDatabaseLoad` at all in order to create our secondary indices. When `ExampleDatabaseLoad` calls `MyDbs.setup()`, all of the necessary work is performed for us.

However, we still need to take advantage of the new secondary indices. We do this by updating `ExampleDatabaseRead` to allow us to query for an inventory record based on its name. Remember that the primary key for an inventory record is the item's SKU. The item's name is contained in the `Inventory` object that is stored as each record's data in the inventory database. But our new secondary index now allows us to easily query based on the item's name.

For this update, we modify `ExampleDatabaseRead` to accept a new command line switch, `-s`, whose argument is the name of an inventory item. If the switch is present on the command line call to `ExampleDatabaseRead`, then the application will use the secondary database to look up and display all the inventory records with that item name. Note that we use a `SecondaryCursor` to seek to the item name key and then display all matching records.

Remember that you can find `ExampleDatabaseRead.java` in:

```
DB_INSTALL/examples_java/db/GettingStarted
```

where *DB_INSTALL* is the location where you placed your DB distribution.

Example 10.3. SecondaryDatabase usage with ExampleDatabaseRead

First we need to import an additional class in order to use the secondary cursor:

```
package db.GettingStarted;

import java.io.IOException;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.db.Cursor;
import com.sleepycat.db.DatabaseEntry;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.LockMode;
```

```
import com.sleepycat.db.OperationStatus;
import com.sleepycat.db.SecondaryCursor;
```

Next we add a single global variable:

```
    public class ExampleDatabaseRead {

    private static String myDbsPath = "./";

    // Encapsulates the database environment and databases.
    private static MyDbs myDbs = new MyDbs();

    private static TupleBinding inventoryBinding;
    private static EntryBinding vendorBinding;

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

Next we update `ExampleDatabaseRead.run()` to check to see if the `locateItem` global variable has a value. If it does, then we show just those records related to the item name passed on the `-s` switch.

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

        myDbs.setup(myDbsPath);

        // Setup our bindings.
        inventoryBinding = new InventoryBinding();
        vendorBinding =
             new SerialBinding(myDbs.getClassCatalog(),
                               Vendor.class);

            if (locateItem != null) {
                showItem();
            } else {
                showAllInventory();
            }
    }
```

Finally, we need to implement `ExampleDatabaseRead.showItem()`. This is a fairly simple method that opens a secondary cursor, and then displays every primary record that is related to the secondary key identified by the `locateItem` global variable.

```
    private void showItem() throws DatabaseException {
        SecondaryCursor secCursor = null;
        try {
            // searchKey is the key that we want to find in the
            // secondary db.
```

```
            DatabaseEntry searchKey =
                new DatabaseEntry(locateItem.getBytes("UTF-8"));

            // foundKey and foundData are populated from the primary
            // entry that is associated with the secondary db key.
            DatabaseEntry foundKey = new DatabaseEntry();
            DatabaseEntry foundData = new DatabaseEntry();

            // open a secondary cursor
            secCursor =
                myDbs.getNameIndexDB().openSecondaryCursor(null, null);

            // Search for the secondary database entry.
            OperationStatus retVal =
                secCursor.getSearchKey(searchKey, foundKey,
                    foundData, LockMode.DEFAULT);

            // Display the entry, if one is found. Repeat until no more
            // secondary duplicate entries are found
            while(retVal == OperationStatus.SUCCESS) {
                Inventory theInventory =
                    (Inventory)inventoryBinding.entryToObject(foundData);
                displayInventoryRecord(foundKey, theInventory);
                retVal = secCursor.getNextDup(searchKey, foundKey,
                    foundData, LockMode.DEFAULT);
            }
        } catch (Exception e) {
            System.err.println("Error on inventory secondary cursor:");
            System.err.println(e.toString());
            e.printStackTrace();
        } finally {
            if (secCursor != null) {
                secCursor.close();
            }
        }

    }
```

The only other thing left to do is to update `ExampleDatabaseRead.parseArgs()` to support the `-s` command line switch. To see how this is done, see `ExampleDatabaseRead.java` in:

```
DB_INSTALL/examples_java/db/GettingStarted
```

where *DB_INSTALL* is the location where you placed your DB distribution.

Chapter 11. Database Configuration

This chapter describes some of the database and cache configuration issues that you need to consider when building your DB database. In most cases, there is very little that you need to do in terms of managing your databases. However, there are configuration issues that you need to be concerned with, and these are largely dependent on the access method that you are choosing for your database.

The examples and descriptions throughout this document have mostly focused on the BTree access method. This is because the majority of DB applications use BTree. For this reason, where configuration issues are dependent on the type of access method in use, this chapter will focus on BTree only. For configuration descriptions surrounding the other access methods, see the *Berkeley DB Programmer's Reference Guide*.

Setting the Page Size

Internally, DB stores database entries on pages. Page sizes are important because they can affect your application's performance.

DB pages can be between 512 bytes and 64K bytes in size. The size that you select must be a power of 2. You set your database's page size using `DatabaseConfig.setPageSize()`.

Note that a database's page size can only be selected at database creation time.

When selecting a page size, you should consider the following issues:

- Overflow pages.
- Locking
- Disk I/O.

These topics are discussed next.

Overflow Pages

Overflow pages are used to hold a key or data item that cannot fit on a single page. You do not have to do anything to cause overflow pages to be created, other than to store data that is too large for your database's page size. Also, the only way you can prevent overflow pages from being created is to be sure to select a page size that is large enough to hold your database entries.

Because overflow pages exist outside of the normal database structure, their use is expensive from a performance perspective. If you select too small of a page size, then your database will be forced to use an excessive number of overflow pages. This will significantly harm your application's performance.

For this reason, you want to select a page size that is at least large enough to hold multiple entries given the expected average size of your database entries. In BTree's case, for best results select a page size that can hold at least 4 such entries.

You can see how many overflow pages your database is using by obtaining a `DatabaseStats` object using the `Database.getStats()` method, or by examining your database using the `db_stat` command line utility.

Locking

Locking and multi-threaded access to DB databases is built into the product. However, in order to enable the locking subsystem and in order to provide efficient sharing of the cache between databases, you must use an *environment*. Environments and multi-threaded access are not fully described in this manual (see the Berkeley DB Programmer's Reference Manual for information), however, we provide some information on sizing your pages in a multi-threaded/multi-process environment in the interest of providing a complete discussion on the topic.

If your application is multi-threaded, or if your databases are accessed by more than one process at a time, then page size can influence your application's performance. The reason why is that for most access methods (Queue is the exception), DB implements page-level locking. This means that the finest locking granularity is at the page, not at the record.

In most cases, database pages contain multiple database records. Further, in order to provide safe access to multiple threads or processes, DB performs locking on pages as entries on those pages are read or written.

As the size of your page increases relative to the size of your database entries, the number of entries that are held on any given page also increase. The result is that the chances of two or more readers and/or writers wanting to access entries on any given page also increases.

When two or more threads and/or processes want to manage data on a page, lock contention occurs. Lock contention is resolved by one thread (or process) waiting for another thread to give up its lock. It is this waiting activity that is harmful to your application's performance.

It is possible to select a page size that is so large that your application will spend excessive, and noticeable, amounts of time resolving lock contention. Note that this scenario is particularly likely to occur as the amount of concurrency built into your application increases.

Oh the other hand, if you select too small of a page size, then that that will only make your tree deeper, which can also cause performance penalties. The trick, therefore, is to select a reasonable page size (one that will hold a sizeable number of records) and then reduce the page size if you notice lock contention.

You can examine the number of lock conflicts and deadlocks occurring in your application by examining your database environment lock statistics. Either use the method, or use the `db_stat` command line utility. The number of unavailable locks that your application waited for is held in the lock statistic's `st_lock_wait` field.

IO Efficiency

Page size can affect how efficient DB is at moving data to and from disk. For some applications, especially those for which the in-memory cache can not be large enough to hold the entire working dataset, IO efficiency can significantly impact application performance.

Most operating systems use an internal block size to determine how much data to move to and from disk for a single I/O operation. This block size is usually equal to the filesystem's block size. For optimal disk I/O efficiency, you should select a database page size that is equal to the operating system's I/O block size.

Essentially, DB performs data transfers based on the database page size. That is, it moves data to and from disk a page at a time. For this reason, if the page size does not match the I/O block size, then the operating system can introduce inefficiencies in how it responds to DB's I/O requests.

For example, suppose your page size is smaller than your operating system block size. In this case, when DB writes a page to disk it is writing just a portion of a logical filesystem page. Any time any application writes just a portion of a logical filesystem page, the operating system brings in the real filesystem page, over writes the portion of the page not written by the application, then writes the filesystem page back to disk. The net result is significantly more disk I/O than if the application had simply selected a page size that was equal to the underlying filesystem block size.

Alternatively, if you select a page size that is larger than the underlying filesystem block size, then the operating system may have to read more data than is necessary to fulfill a read request. Further, on some operating systems, requesting a single database page may result in the operating system reading enough filesystem blocks to satisfy the operating system's criteria for read-ahead. In this case, the operating system will be reading significantly more data from disk than is actually required to fulfill DB's read request.

While transactions are not discussed in this manual, a page size other than your filesystem's block size can affect transactional guarantees. The reason why is that page sizes larger than the filesystem's block size causes DB to write pages in block size increments. As a result, it is possible for a partial page to be written as the result of a transactional commit. For more information, see http://www.oracle.com/technology/documentation/berkeley-db/db/ref/transapp/reclimit.html.

Page Sizing Advice

Page sizing can be confusing at first, so here are some general guidelines that you can use to select your page size.

In general, and given no other considerations, a page size that is equal to your filesystem block size is the ideal situation.

If your data is designed such that 4 database entries cannot fit on a single page (assuming BTree), then grow your page size to accommodate your data. Once you've abandoned matching your filesystem's block size, the general rule is that larger page sizes are better.

The exception to this rule is if you have a great deal of concurrency occurring in your application. In this case, the closer you can match your page size to the ideal size needed for your application's data, the better. Doing so will allow you to avoid unnecessary contention for page locks.

Selecting the Cache Size

Cache size is important to your application because if it is set to too small of a value, your application's performance will suffer from too much disk I/O. On the other hand, if your cache is too large, then your application will use more memory than it actually needs. Moreover, if your application uses too much memory, then on most operating systems this can result in your application being swapped out of memory, resulting in extremely poor performance.

You select your cache size using either `DatabaseConfig.setCacheSize()`, or `EnvironmentConfig.setCacheSize()`, depending on whether you are using a database environment or not. You cache size must be a power of 2, but it is otherwise limited only by available memory and performance considerations.

Selecting a cache size is something of an art, but fortunately you can change it any time, so it can be easily tuned to your application's changing data requirements. The best way to determine how large your cache needs to be is to put your application into a production environment and watch to see how much disk I/O is occurring. If your application is going to disk quite a lot to retrieve database records, then you should increase the size of your cache (provided that you have enough memory to do so).

You can use the `db_stat` command line utility with the `-m` option to gauge the effectiveness of your cache. In particular, the number of pages found in the cache is shown, along with a percentage value. The closer to 100% that you can get, the better. If this value drops too low, and you are experiencing performance problems, then you should consider increasing the size of your cache, assuming you have memory to support it.

BTree Configuration

In going through the previous chapters in this book, you may notice that we touch on some topics that are specific to BTree, but we do not cover those topics in any real detail. In this section, we will discuss configuration issues that are unique to BTree.

Specifically, in this section we describe:

- Allowing duplicate records.
- Setting comparator callbacks.

Allowing Duplicate Records

BTree databases can contain duplicate records. One record is considered to be a duplicate of another when both records use keys that compare as equal to one another.

By default, keys are compared using a lexicographical comparison, with shorter keys collating higher than longer keys. You can override this default using the `DatabaseConfig.setBtreeComparator()` method. See the next section for details.

By default, DB databases do not allow duplicate records. As a result, any attempt to write a record that uses a key equal to a previously existing record results in the previously existing record being overwritten by the new record.

Allowing duplicate records is useful if you have a database that contains records keyed by a commonly occurring piece of information. It is frequently necessary to allow duplicate records for secondary databases.

For example, suppose your primary database contained records related to automobiles. You might in this case want to be able to find all the automobiles in the database that are of a particular color, so you would index on the color of the automobile. However, for any given color there will probably be multiple automobiles. Since the index is the secondary key, this means that multiple secondary database records will share the same key, and so the secondary database must support duplicate records.

Sorted Duplicates

Duplicate records can be stored in sorted or unsorted order. You can cause DB to automatically sort your duplicate records by setting `DatabaseConfig.setSortedDuplicates()` to `true`. Note that this property must be set prior to database creation time and it cannot be changed afterwards.

If sorted duplicates are supported, then the `java.util.Comparator` implementation identified to `DatabaseConfig.setDuplicateComparator()` is used to determine the location of the duplicate record in its duplicate set. If no such function is provided, then the default lexicographical comparison is used.

Unsorted Duplicates

For performance reasons, BTrees should always contain sorted records. (BTrees containing unsorted entries must potentially spend a great deal more time locating an entry than does a BTree that contains sorted entries). That said, DB provides support for suppressing automatic sorting of duplicate records because it may be that your application is inserting records that are already in a sorted order.

That is, if the database is configured to support unsorted duplicates, then the assumption is that your application will manually perform the sorting. In this event, expect to pay a significant performance penalty. Any time you place records into the database in a sort order not know to DB, you will pay a performance penalty

That said, this is how DB behaves when inserting records into a database that supports non-sorted duplicates:

- If your application simply adds a duplicate record using `Database.put()`, then the record is inserted at the end of its sorted duplicate set.
- If a cursor is used to put the duplicate record to the database, then the new record is placed in the duplicate set according to the actual method used to perform the put. The relevant methods are:
  - `Cursor.putAfter()`

    The data is placed into the database as a duplicate record. The key used for this operation is the key used for the record to which the cursor currently refers. Any key provided on the call is therefore ignored.

    The duplicate record is inserted into the database immediately after the cursor's current position in the database.
  - `Cursor.putBefore()`

    Behaves the same as `Cursor.putAfter()` except that the new record is inserted immediately before the cursor's current location in the database.
  - `Cursor.putKeyFirst()`

    If the key already exists in the database, and the database is configured to use duplicates without sorting, then the new record is inserted as the first entry in the appropriate duplicates list.
  - `Cursor.putKeyLast()`

    Behaves identically to `Cursor.putKeyFirst()` except that the new duplicate record is inserted as the last record in the duplicates list.

Configuring a Database to Support Duplicates

Duplicates support can only be configured at database creation time. You do this by specifying the appropriate `DatabaseConfig` method before the database is opened for the first time.

The methods that you can use are:

- `DatabaseConfig.setUnsortedDuplicates()`

  The database supports non-sorted duplicate records.
- `DatabaseConfig.setSortedDuplicates()`

  The database supports sorted duplicate records.

The following code fragment illustrates how to configure a database to support sorted duplicate records:

```
package db.GettingStarted;

import java.io.FileNotFoundException;

import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseException;
import com.sleepycat.db.DatabaseType;

...

Database myDb = null;

try {
    // Typical configuration settings
    DatabaseConfig myDbConfig = new DatabaseConfig();
    myDbConfig.setType(DatabaseType.BTREE);
    myDbConfig.setAllowCreate(true);

    // Configure for sorted duplicates
    myDbConfig.setSortedDuplicates(true);

   // Open the database
   myDb = new Database("mydb.db", null, myDbConfig);
} catch(DatabaseException dbe) {
    System.err.println("MyDbs: " + dbe.toString());
    System.exit(-1);
} catch(FileNotFoundException fnfe) {
    System.err.println("MyDbs: " + fnfe.toString());
    System.exit(-1);
}
```

Setting Comparison Functions

By default, DB uses a lexicographical comparison function where shorter records collate before longer records. For the majority of cases, this comparison works well and you do not need to manage it in any way.

However, in some situations your application's performance can benefit from setting a custom comparison routine. You can do this either for database keys, or for the data if your database supports sorted duplicate records.

Some of the reasons why you may want to provide a custom sorting function are:

- Your database is keyed using strings and you want to provide some sort of language-sensitive ordering to that data. Doing so can help increase the locality of reference that allows your database to perform at its best.
- You are using a little-endian system (such as x86) and you are using integers as your database's keys. Berkeley DB stores keys as byte strings and little-endian integers do not sort well when

viewed as byte strings. There are several solutions to this problem, one being to provide a custom comparison function. See http://www.oracle.com/technology/documentation/berkeley-db/db/ref/am_misc/faq.html for more information.

- You you do not want the entire key to participate in the comparison, for whatever reason. In this case, you may want to provide a custom comparison function so that only the relevant bytes are examined.

Creating Java Comparators

You set a BTree's key comparator using `DatabaseConfig.setBtreeComparator()`. You can also set a BTree's duplicate data comparison function using `DatabaseConfig.setDuplicateComparator()`.

If the database already exists when it is opened, the comparator provided to these methods must be the same as that historically used to create the database or corruption can occur.

You override the default comparison function by providing a Java `Comparator` class to the database. The Java `Comparator` interface requires you to implement the `Comparator.compare()` method (see http://java.sun.com/j2se/1.4.2/docs/api/java/util/Comparator.html for details).

DB hands your `Comparator.compare()` method the `byte` arrays that you stored in the database. If you know how your data is organized in the `byte` array, then you can write a comparison routine that directly examines the contents of the arrays. Otherwise, you have to reconstruct your original objects, and then perform the comparison.

For example, suppose you want to perform unicode lexical comparisons instead of UTF-8 byte-by-byte comparisons. Then you could provide a comparator that uses `String.compareTo()`, which performs a Unicode comparison of two strings (note that for single-byte roman characters, Unicode comparison and UTF-8 byte-by-byte comparisons are identical – this is something you would only want to do if you were using multibyte unicode characters with DB). In this case, your comparator would look like the following:

```
package db.GettingStarted;

import java.util.Comparator;

public class MyDataComparator implements Comparator {

    public MyDataComparator() {}

    public int compare(Object d1, Object d2) {

        byte[] b1 = (byte[])d1;
        byte[] b2 = (byte[])d2;

        String s1 = new String(b1);
        String s2 = new String(b2);
        return s1.compareTo(s2);
```

```
    }
}
```

To use this comparator:

```
package db.GettingStarted;

import java.io.FileNotFoundException;
import java.util.Comparator;
import com.sleepycat.db.Database;
import com.sleepycat.db.DatabaseConfig;
import com.sleepycat.db.DatabaseException;

...

Database myDatabase = null;
try {
    // Get the database configuration object
    DatabaseConfig myDbConfig = new DatabaseConfig();
    myDbConfig.setAllowCreate(true);

    // Set the duplicate comparator class
    MyDataComparator mdc = new MyDataComparator();
    myDbConfig.setDuplicateComparator(mdc);

    // Open the database that you will use to store your data
    myDbConfig.setSortedDuplicates(true);
    myDatabase = new Database("myDb", null, myDbConfig);
} catch (DatabaseException dbe) {
    // Exception handling goes here
} catch (FileNotFoundException fnfe) {
    // Exception handling goes here
}
```