

Oracle Berkeley DB Java Edition

Getting Started with Berkeley DB Java Edition

Release 3.2



Legal Notice

This documentation is distributed under an open source license. You may review the terms of this license at: <http://www.oracle.com/technology/software/products/berkeley-db/htdocs/jeoslicense.html>

Oracle, Berkeley DB, Berkeley DB Java Edition and Sleepycat are trademarks or registered trademarks of Oracle. All rights to these marks are reserved. No third-party use is permitted without the express prior written consent of Oracle.

Java™ and all Java-based marks are a trademark or registered trademark of Sun Microsystems, Inc, in the United States and other countries.

To obtain a copy of this document's original source code, please submit a request to the Oracle Technology Network forum at: <http://forums.oracle.com/forums/forum.jspa?forumID=273>

Published 11/9/2007

Table of Contents

Preface	v
Conventions Used in this Book	v
For More Information	vi
1. Introduction to Berkeley DB Java Edition	1
Features	1
The JE Application	3
Databases and Database Environments	3
Database Records	4
Putting and Getting Database Records	4
Duplicate Data	4
Replacing and Deleting Entries	5
Secondary Databases	5
Transactions	5
JE Resources	6
Application Considerations	7
JE Backup and Restore	7
JCA Support	8
JMX Support	8
Getting and Using JE	9
JE Exceptions	9
Six Things Everyone Should Know about JE Log Files	10
2. Database Environments	11
Opening Database Environments	11
Closing Database Environments	12
Environment Properties	13
The EnvironmentConfig Class	13
EnvironmentMutableConfig	14
Environment Statistics	16
Database Environment Management Example	16
3. Databases	19
Opening Databases	19
Deferred Write Databases	20
Closing Databases	22
Database Properties	23
Administrative Methods	25
Database Example	26
4. Database Records	30
Using Database Records	30
Reading and Writing Database Records	32
Writing Records to the Database	32
Getting Records from the Database	33
Deleting Records	35
Data Persistence	35
Using the BIND APIs	36
Numerical and String Objects	36
Serializable Complex Objects	38

Usage Caveats	39
Serializing Objects	40
Deserializing Objects	42
Custom Tuple Bindings	43
Using Comparators	47
Writing Comparators	47
Setting Comparators	48
Database Record Example	49
5. Using Cursors	61
Opening and Closing Cursors	61
Getting Records Using the Cursor	62
Searching for Records	64
Working with Duplicate Records	67
Putting Records Using Cursors	69
Deleting Records Using Cursors	71
Replacing Records Using Cursors	72
Cursor Example	73
6. Secondary Databases	78
Opening and Closing Secondary Databases	79
Implementing Key Creators	81
Secondary Database Properties	84
Reading Secondary Databases	84
Deleting Secondary Database Records	85
Using Secondary Cursors	86
Database Joins	87
Using Join Cursors	88
JoinCursor Properties	91
Secondary Database Example	91
Opening Secondary Databases with MyDbEnv	93
Using Secondary Databases with ExampleInventoryRead	96
7. Backing up and Restoring Berkeley DB Java Edition Applications	99
Databases and Log Files	99
Log File Overview	99
Cleaning the Log Files	100
The BTree	100
Database Modifications and Syncs	100
Normal Recovery	101
Performing Backups	101
Performing a Hot Backup	101
Performing an Offline Backup	102
Using the DbBackup Helper Class	102
Performing Catastrophic Recovery	104
Hot Standby	104
8. Administering Berkeley DB Java Edition Applications	106
The JE Properties File	106
Managing the Background Threads	106
The Cleaner Thread	107
The Checkpointer Thread	108
Sizing the Cache	108

The Command Line Tools	109
DbDump	109
DbLoad	111
DbVerify	112

Preface

Welcome to Berkeley DB Java Edition (JE). This document introduces JE, version 3.2. It is intended to provide a rapid introduction to the JE API set and related concepts. The goal of this document is to provide you with an efficient mechanism with which you can evaluate JE 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 Java Edition 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 `Environment.openDatabase()` method returns a `Database` class object."

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

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

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

...

// Open the environment. Allow it to be created if it does not already exist.
Environment myDbEnv;
```

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.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 myDbEnv;
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
myDbEnv = new Environment(new File("/export/dbEnv"), envConfig);
```



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 JE application:

- [Berkeley DB Java Edition Getting Started with Transaction Processing](http://www.oracle.com/technology/documentation/berkeley-db/je/TransactionGettingStarted/BerkeleyDB-JE-Txn.pdf)
[<http://www.oracle.com/technology/documentation/berkeley-db/je/TransactionGettingStarted/BerkeleyDB-JE-Txn.pdf>]
- [Berkeley DB Java Edition Javadoc](http://www.oracle.com/technology/documentation/berkeley-db/je/java/index.html)
[<http://www.oracle.com/technology/documentation/berkeley-db/je/java/index.html>]
- [Berkeley DB Java Edition Collections Tutorial](http://www.oracle.com/technology/documentation/berkeley-db/je/collections/tutorial/BerkeleyDB-JE-Collections.pdf)
[<http://www.oracle.com/technology/documentation/berkeley-db/je/collections/tutorial/BerkeleyDB-JE-Collections.pdf>]
- [Berkeley DB Java Edition Getting Started with the Direct Persistence Layer](http://www.oracle.com/technology/documentation/berkeley-db/je/PersistenceAPI/BerkeleyDB-JE-Persistence-GSG.pdf)
[<http://www.oracle.com/technology/documentation/berkeley-db/je/PersistenceAPI/BerkeleyDB-JE-Persistence-GSG.pdf>]

Chapter 1. Introduction to Berkeley DB Java Edition

Welcome to Berkeley DB Java Edition (JE). JE is a general-purpose, transaction-protected, embedded database written in 100% Java (JE makes no JNI calls). As such, it offers the Java developer safe and efficient in-process storage and management of arbitrary data.

JE requires Java J2SE 1.4.2 or later.



In addition to the APIs described in this book, JE also offers the Direct Persistence Layer (DPL). The DPL is implemented as a layer on top of JE. Using it, you can cause any Java type to be persistent without implementing special interfaces.

We recommend that you use the DPL if all you want is to make classes with relatively static schemas to be persistent.

The only caveat to this recommendation is that the DPL requires Java 1.5, while JE does not. If you want to use Java 1.4, you must use the JE APIs.

You can find a brief introduction to the DPL in the *Berkeley DB Java Edition Getting Started with the Direct Persistence Layer* guide.

Features

JE provides an enterprise-class Java-based data management solution. You use JE through a series of Java APIs. All you need to get started is to add a single jar file to your application's classpath. See [Getting and Using JE \(page 9\)](#) for more information.

JE offers the following major features:

- Large database support. JE databases efficiently scale from one to millions of records. The size of your JE databases are likely to be limited more by hardware resources than by any limits imposed upon you by JE.

Databases are described in [Databases \(page 19\)](#).

- Multiple thread and process support. JE is designed for multiple threads of control. Both read and write operations can be performed by multiple threads. JE uses record-level locking for high concurrency in threaded applications. Further, JE uses timeouts for deadlock detection to help you ensure that two threads of control do not deadlock indefinitely.

Moreover, JE allows multiple processes to access the same databases. However, in this configuration JE requires that there be no more than one process allowed to write to the database. Read-only processes are guaranteed a consistent, although potentially out of date, view of the stored data as of the time that the environment is opened.

- Database records. All database records are organized as simple key/data pairs. Both keys and data can be anything from primitive Java types to the most complex of Java objects.

Database records are described in [Database Records \(page 30\)](#).

- Transactions. Transactions allow you to treat one or more operations on one or more databases as a single unit of work. JE transactions offer the application developer recoverability, atomicity, and isolation for your database operations.

Note that transaction protection is optional. Transactions are described in the *Berkeley DB Java Edition Getting Started with Transaction Processing* guide.

- Indexes. JE allows you to easily create and maintain secondary indices for your primary data through the use of secondary databases. In this way, you can obtain rapid access to your data through the use of an alternative, or secondary, key.

Indexes are described in [Secondary Databases \(page 78\)](#).

- In-memory cache. The cache allows for high speed database access for both read and write operations by avoiding unnecessary disk I/O. The cache will grow on demand up to a pre-configured maximum size. To improve your application's performance immediately after startup time, you can preload your cache in order to avoid disk I/O for production requests of your data.

Cache management is described in [Sizing the Cache \(page 108\)](#).

- Log files. JE databases are stored in one or more sequential numerically-named log files in the environment directory. The log files are write-once and are portable across platforms with different endianness.

Note that unlike other database implementations, there is no distinction between database files (that is, the "material database") and log files. Instead JE employs a log-based storage system to protect database modifications. Before any change is made to a database, JE writes information about the change to the log file.

Note that JE's log files are not binary compatible with Berkeley DB's database files. However, both products provide dump and load utilities, and the files that these operate on are compatible across product lines.

JE's log files are described in more detail in [Backing up and Restoring Berkeley DB Java Edition Applications \(page 99\)](#). For information on using JE's dump and load utilities, see [The Command Line Tools \(page 109\)](#). Finally, for a short list of things to know about log files while you are learning JE, see [Six Things Everyone Should Know about JE Log Files \(page 10\)](#).

- Background threads. JE provides several threads that manage internal resources for you. The checkpointer is responsible for flushing database data to disk that was written to cache as the result of a transaction commit (this is done in order to shorten recovery time). The compressor thread removes subtrees from the database that are empty because of deletion activity. Finally, the cleaner thread is responsible for cleaning and removing unneeded log files, thereby helping you to save on disk space.

Background thread management is described in [Managing the Background Threads \(page 106\)](#).

- Database environments. Database environments provide a unit of encapsulation and management for one or more databases. In addition, an environment is the unit of management for internal resources such as the in-memory cache and the background threads. Environments are also used to manage concurrency and transactions. Note that all applications using JE are required to use database environments.

Database environments are described in [Database Environments \(page 11\)](#).

- Backup and restore. JE's backup procedure consists of simply copying JE's log files to a safe location for storage. To recover from a catastrophic failure, you copy your archived log files back to your production location on disk and reopen the JE environment.

Note that JE always performs *normal recovery* when it opens a database environment. Normal recovery brings the database to a consistent state based on change information found in the database log files.

JE's backup and recovery mechanisms are described in [Backing up and Restoring Berkeley DB Java Edition Applications \(page 99\)](#).

- JCA. JE provides support for the Java Connector Architecture. See [JCA Support \(page 8\)](#) for more information.
- JMX. JE provides support for Java Management Extensions. See [JMX Support \(page 8\)](#) for more information.

The JE Application

This section provides a brief overview to the major concepts and operations that comprise a JE application. This section is concluded with a summary of the decisions that you need to make when building a JE application.

Note that the core JE classes are all contained in the `com.sleepycat.je` package. In addition, this book describes some classes that are found in `com.sleepycat.je.bind`. The bind APIs are used for converting Java objects in and out of `byte` arrays.

Databases and Database Environments

To use a JE database, you must first create or open a JE database environment. Database environments require you to identify the directory on disk where the environment lives. This location must exist before you create the environment.

You open a database environment by instantiating an `Environment` object. Your `Environment` instance is called an *environment handle*.

Once you have opened an environment, you can use it to open any number of databases within that environment. Each such database is encapsulated by a `Database` object. You are required to provide a string that uniquely identifies the database when you open it. Like environments, the `Database` instance is sometimes referred to as a *database handle*.

You use the environment handle to manage database environments and database opens through methods available on the `Environment` class. You use the database handle to manage individual databases through methods available on the `Database` class. Further, you use environment handles to close environments, and you use database handles to close databases.

Note that for both databases and environments, you can optionally allow JE to create them if they do not exist at open time.

Environments are described in greater detail in [Database Environments \(page 11\)](#). Databases are described in greater detail in [Databases \(page 19\)](#).

Database Records

Database records are represented as simple key/data pairs. Both record keys and record data must be byte arrays and are passed to and returned from JE using `DatabaseEntry` instances. `DatabaseEntry` only supports storage of Java byte arrays. Complex objects must be marshalled using either Java serialization, or more efficiently with the bind APIs provided with JE

Database records and byte array conversion are described in [Database Records \(page 30\)](#).

Putting and Getting Database Records

You store records in a `Database` by calling one of the put methods on a `Database` handle. JE 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 \(page 19\)](#). Cursors are described in [Using Cursors \(page 61\)](#).

Duplicate Data

At creation time, databases can be configured to allow duplicate data. Remember that JE 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.

Replacing and Deleting Entries

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, you can use either `Database.delete()` or `Cursor.delete()`. If duplicate data is not allowed in your database, then these two methods 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 Databases

Secondary Databases provide a mechanism by which you can automatically create and maintain secondary keys or indices. That is, you can access a database record using a key other than the one used to store the record in the first place.

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, JE uses this class to determine what the secondary key should be.

When a primary record is created, modified, or deleted, JE 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 \(page 78\)](#).

Transactions

Transactions provide a high level of safety for your database operations by allowing you to manage one or more database operations as if they were a single unit of work. Transactions provide your database operations with recoverability, atomicity, and isolation.

Transactions provide recoverability by allowing JE to undo any transactional-protected operations that may have been in progress at the time of an application or system failure.

Transactions provide atomicity by allowing you to group many database operations into a single unit of work. Either all operations succeed or none of them do. This means that if one write operation fails for any reason, then all other writes contained within that transaction also fail. This ensures that the database is never partially updated as the result of an only partially successful chain of read/write operations.

Transactions provide isolation by ensuring that the transaction will never write to a record that is currently in use (for either read or write) by another transaction. Similarly, any record to which the transaction has written can not be read outside of the transaction until the transaction ends. Note that this is only the default behavior; you can configure your `Database`, `Cursor`, or `Transaction` handle to relax its isolation guarantees.

Essentially, transactional isolation provides a transaction with the same unmodified view of the database that it would have received had the operations been performed in a single-threaded application.

Transactions may be long or short lived, they can encompass as many database operations as you want, and they can span databases so long as all participating databases reside in the same environment.

Transaction usage results in a performance penalty for the application because they generally require more disk I/O than do non-transactional operations. Therefore, while most applications will use transactions for database writes, their usage is optional. In particular, processes that are performing read-only access to JE databases might not use transactions. Also, applications that use JE for an easily recreated cache might also choose to avoid transactions.

Using transactions with your JE applications is described in detail in the *Berkeley DB Java Edition Getting Started with Transaction Processing* guide.

JE Resources

JE has some internal resources that you may want to manage. Most important of these is the in-memory cache. You should carefully consider how large the JE cache needs to be. If you set this number too low, JE will perform potentially unnecessary disk I/O which will result in a performance hit. If you set it too high, then you are potentially wasting RAM that could be put to better purposes.

Note that the size that you configure for the in-memory cache is a maximum size. At application startup, the cache starts out fairly small (only about 7% of the maximum allowed size for the cache). It then grows as is required by your application's database operations. Also, the cache is not pinned in memory - it can be paged out by your operating system's virtual memory system.

Beyond the cache, JE uses several background threads to clean the JE log files, to compress the database by removing unneeded subtrees, and to flush database changes seen in the cache to the backing data files. For the majority of JE applications, the default behavior

for the background threads should be acceptable and you will not need to manage their behavior. Note that background threads are started no more than once per process upon environment open.

For more information on sizing the cache and on the background threads, see [Administering Berkeley DB Java Edition Applications \(page 106\)](#).

Application Considerations

When building your JE application, be sure to think about the following things:

- What data do you want to store? What is best used for the primary key? What is the best representation for primary record data? Think about the most efficient way to move your keys and data in and out of byte arrays. See [Database Records \(page 30\)](#) for more information.
- Does the nature of your data require duplicate record support? Remember that duplicate support can be configured only at database creation time. See [Opening Databases \(page 19\)](#) for more information.

If you are supporting duplicate records, you may also need to think about duplicate comparators (not just key comparators). See [Using Comparators \(page 47\)](#) for more information.

- What secondary indexes do you need? How can you compute your secondary indexes based on the data and keys stored in your primary database? Indexes are described in [Secondary Databases \(page 78\)](#).
- What cache size do you need? See [Sizing the Cache \(page 108\)](#) for information on how to size your cache.
- Does your application require transactions (most will). Transactions are described in the *Berkeley DB Java Edition Getting Started with Transaction Processing* guide.

JE Backup and Restore

To backup your database, copy the `.jdb` files starting from the lowest numbered log file to the highest numbered log file to your backup media. Be sure to copy the bytes of the individual database files in order from the lowest to the highest. You do not have to close your database or otherwise cease database operations when you do this.

Restoring a JE database from a backup consists of closing your JE environment, copying archived log files back into your environment directory and then opening your JE environment again.

Note that whenever a JE environment is opened, JE runs *normal recovery*. This involves bringing your database into a consistent state given the changed data found in the database. If you are using transactions during normal operations, then JE automatically runs checkpoints for you so as to limit the time required to run this recovery. In any case,

running normal recovery is a routine operation, while performing database restores is not.

For more information on JE backup and restores, and on checkpoints, see [Backing up and Restoring Berkeley DB Java Edition Applications \(page 99\)](#).

JCA Support

JCA is the *Java Connector Architecture*. This architecture provides a standard for connecting the J2EE platform to legacy enterprise information systems (EIS), such as ERP systems, database systems, and legacy applications not written in Java. JE supports this architecture.

Users who want to run JE within a J2EE Application Server can use the JCA Resource Adapter to connect to JE through a standard API. The JE Resource Adapter supports all three J2EE application server transaction types:

- No transaction.
- Local transactions.
- XA transactions.

JCA also includes the Java Transaction API (JTA), which means that JE supports 2 phase commit (XA). Therefore, JEs can participate in distributed transactions managed by either a J2EE server or the applications direct use of the JTA API.

The JE distribution includes an example showing JCA usage in a simple EJB. The Resource Adaptor has been tested using JBoss 3.2.6, and the Sun Java System Application Server, version 8.1. Instructions for how to build the Resource Adapter and run a simple "smoke test" example for each of the application servers can be found here:

```
JE_HOME/examples/jca/HOWTO-jboss.txt
```

and

```
JE_HOME/examples/jca/HOWTO-sjsas.txt
```

JMX Support

JMX is the *Java Management Extensions*. This extension provides tools for managing and monitoring devices, applications, and service-driven networks. JE supports this extension.

The JE distribution supplies an MBean that can be deployed for monitoring a JE environment in any JMX server (such as an J2EE application server). Alternatively, applications can use JE helper classes to add JE monitoring to their own JMX MBean implementations.

For information on how to deploy the standalone JE JMX MBean, or on how to use JE helper classes to build an application-specific MBean, see:

```
JE_HOME/examples/jmx/README.txt
```

Getting and Using JE

You can obtain JE by visiting the JE download page:

<http://www.oracle.com/technology/software/products/berkeley-db/je/index.html>.

To install JE, simply untar or unzip the distribution to the directory of your choice. If you use unzip, make sure to specify the `-U` option in order to preserve case.

For more information on installing JE, see *JE_HOME/docs/relnotes.html*, where *JE_HOME* is the directory where you unpacked JE.

You can use JE with your application by adding *JE_HOME/lib/je-<version>.jar* to your application's classpath.

Beyond this manual, you can find documentation for JE at *JE_HOME/docs/index.html* directory. In particular, complete Javadoc for the JE API set is available at *JE_HOME/docs/java/index.html*.

JE Exceptions

Before describing the Java API usage, it is first useful to examine the exceptions thrown by those APIs. So, briefly, this section describes the exceptions that you can expect to encounter when writing JE applications.

All of the JE APIs throw `DatabaseException`. `DatabaseException` extends `java.lang.Exception`. Also, the following classes are subclasses of `DatabaseException`:

- `DatabaseNotFoundException`

Thrown whenever an operation requires a database, and that database cannot be found.

- `DeadlockException`

Thrown whenever a transaction is selected to resolve a deadlock. Upon receiving this exception, any open cursors must be closed and the enclosing transaction aborted. Transactions are described in the *Berkeley DB Java Edition Getting Started with Transaction Processing* guide.

- `RunRecoveryException`

Thrown whenever JE experiences a catastrophic error such that recovery needs to be run on the database. If you receive this exception, you must reopen your environment so as to allow normal recovery to run. See [Databases and Log Files \(page 99\)](#) for more information on how normal recovery works.

Note that when reopening your environment, you should stop all database read and write activities, close all your cursors, close all your databases, and then close and reopen your environment.

Note that `DatabaseException` and its subclasses belong to the `com.sleepycat.je` package.

Six Things Everyone Should Know about JE Log Files

JE log files are not like the log files of other database systems. Nor are they like the log files or database files created by Berkeley DB C Edition. In this guide you will learn more about log files as you go along, but it is good to keep the following points in mind as you begin using JE.

1. JE log files are "append only". Record insertions, deletions, and updates are always added at the end of the current file. The first file is named `00000000.jdb`. When that file grows to a certain size (10 MB by default) a new file named `00000001.jdb` becomes the current file, and so on.
2. There are no separate database files. Unlike Berkeley DB C Edition, databases are not stored in files that are separate from the transaction log. The transaction log and the database records are stored together in a single sequential log consisting of multiple log files.
3. The JE cleaner is responsible for reclaiming unused disk space. When the records in a log file are superseded by deletions or updates recorded in a later log file, the older log file is no longer fully utilized. The cleaner, which operates by default as a separate thread, determines the least utilized log files, copies any still utilized records in those files to the end of the current log file, and finally deletes the now completely un-utilized log file.

See [The Cleaner Thread \(page 107\)](#) for more information on the cleaner.

4. Cleaning does not start immediately and never produces 100% utilization. Until you have written enough data to create several log files, and some of that data is obsoleted through deletions and updates, you will not notice any log files being deleted by the cleaner. By default cleaning occurs in the background and maintains the log files at 50% utilization. You can configure a higher utilization value, but configuring too high a utilization value will reduce overall performance.
5. Cleaning is not automatically performed when closing the environment. If you wish to reduce unused disk space to a minimum at a particular point in time, you must explicitly call a method to perform log cleaning. See the [Closing Database Environments \(page 12\)](#) for more information.
6. Log file deletion only occurs after a checkpoint. The cleaner prepares log files to be deleted, but file deletion must be performed after a checkpoint to ensure that the files are no longer referenced. Checkpoints occur on their own schedule, which is every 20 MB of log written, by default. This is part of the reason that you will not see log files being deleted until after several files have been created.

Chapter 2. Database Environments

Berkeley DB Java Edition requires that all databases be placed in a database environment. 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.

Finally, you also use the database environment for administrative and configuration activities related to your database log files and the in-memory cache. See [Administering Berkeley DB Java Edition Applications \(page 106\)](#) for more information.

For information on managing databases using database environments, see [Databases \(page 19\)](#). To find out how to use environments with a transaction-protected application, see the *Berkeley DB Java Edition Getting Started with Transaction Processing* guide.

Opening Database Environments

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

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

```
package je.gettingStarted;

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

import java.io.File;

...

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

try {
    EnvironmentConfig envConfig = new EnvironmentConfig();
```

```

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

```

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

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

Note that opening your environment causes normal recovery to be run. This causes your databases to be brought into a consistent state relative to the changed data found in your log files. See [Databases and Log Files \(page 99\)](#) for more information.

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 [Database Modifications and Syncs \(page 100\)](#).

```

import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;

...

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

```

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



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

See [The Cleaner Thread \(page 107\)](#) for details on the cleaner threads.

See the *Getting Started with Berkeley DB Java Edition* guide for information on the cleaner threads.

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

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

import com.sleepycat.je.Environment;

...

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

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

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

- `EnvironmentConfig.setTransactional()`

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

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

For example:

```
package je.gettingStarted;

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

import java.io.File;

...

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

EnvironmentMutableConfig

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

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

- `setCachePercent()`

Determines the percentage of JVM memory available to the JE cache. See [Sizing the Cache \(page 108\)](#) for more information.

- `setCacheSize()`

Determines the total amount of memory available to the database cache. See [Sizing the Cache \(page 108\)](#) 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 *Berkeley DB Java Edition Getting Started with Transaction Processing* guide.

- `setTxnWriteNoSync()`

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

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

For example:

```
package je.gettingStarted;

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

import java.io.File;

...

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

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

Environment Statistics

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

However, one statistic that is very important (especially for long-running applications) is `EnvironmentStats.getNCacheMiss()`. This statistic returns the total number of requests for database objects that were not serviceable from the cache. This number is important to the application administrator who is attempting to determine the proper size for the in-memory cache. See [Sizing the Cache \(page 108\)](#) for details.

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

```
import com.sleepycat.je.Environment;  
  
...  
  
long cacheMisses = myEnv.getStats(null).getNCacheMiss();  
  
...
```

Note that `Environment.getStats()` can only obtain statistics from with your application's process. In order for the application administrator to obtain this statistic, you must either use JMX to retrieve the statistic (see [JMX Support \(page 8\)](#)) or you must print it for examination (for example, log the value once a minute).

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

Database Environment Management Example

This example provides a complete class that can open and close an environment. It is both extended and used in subsequent examples in this book to open and close both environments and databases. We do this so as to make the example code shorter and easier to manage. You can find this class in:

```
JE_HOME/examples/je/gettingStarted/MyDbEnv.java
```

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

Example 2.1. Database Environment Management Class

First we write the normal class declarations. We also set up some private data members that are used to manage environment creation. We use the class constructor to instantiate the `EnvironmentConfig` object that is used to configure our environment when we open it.

```
// File MyDbEnv.java
package je.gettingStarted;

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

import java.io.File;

public class MyDbEnv {

    private Environment myEnv;

    public MyDbEnv() {}
```

Next we need a method to open the environment. This is responsible for instantiating our `Environment` object. Remember that instantiation is what opens the environment (or creates it if the creation property is set to `true` and the environment does not currently exist).

```
public void setup(File envHome, boolean readOnly)
    throws DatabaseException {

    // Instantiate an environment configuration object
    EnvironmentConfig myEnvConfig = new EnvironmentConfig();
    // Configure the environment for the read-only state as identified by
    // the readOnly parameter on this method call.
    myEnvConfig.setReadOnly(readOnly);
    // If the environment is opened for write, then we want to be able to
    // create the environment if it does not exist.
    myEnvConfig.setAllowCreate(!readOnly);

    // Instantiate the Environment. This opens it and also possibly
    // creates it.
    myEnv = new Environment(envHome, myEnvConfig);
}
```

Next we provide a getter method that allows us to retrieve the `Environment` directly. This is needed for later examples in this guide.

```
// Getter methods
public Environment getEnv() {
```

```
        return myEnv;
    }
```

Finally, we need a method to close our `Environment`. We wrap this operation in a `try` block so that it can be used gracefully in a `finally` statement.

```
    // Close the environment
    public void close() {
        if (myEnv != null) {
            try {
                myEnv.close();
            } catch(DatabaseException dbe) {
                System.err.println("Error closing environment" +
                    dbe.toString());
            }
        }
    }
}
```

This completes the `MyDbEnv` class. While not particularly useful as it currently exists, we will build upon it throughout this book so that it will eventually open and close all of the databases required by our applications.

We can now use `MyDbEnv` to open and close a database environment from the appropriate place in our application. For example:

```
package je.gettingStarted;

import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;

import java.io.File;

...

MyDbEnv exampleDbEnv = new MyDbEnv();

try {
    exampleDbEnv.setup(new File("/directory/currently/exists"), true);
    ...
} catch(DatabaseException dbe) {
    // Error code goes here
} finally {
    exampleDbEnv.close();
}
```

Chapter 3. Databases

In Berkeley DB Java Edition, 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 \(page 30\)](#) for details on this class). So, fundamentally, using a JE `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.

Note that on disk, databases are stored in sequentially numerically named log files in the directory where the opening environment is located. JE log files are described [Databases and Log Files \(page 99\)](#).

Opening Databases

You open a database by using the `Environment.openDatabase()` method (environments are described in [Database Environments \(page 11\)](#)). This method creates and returns a `Database` object handle. You must provide `Environment.openDatabase()` with a database name.

You can optionally provide `Environment.openDatabase()` with a `DatabaseConfig()` object. `DatabaseConfig()` allows you to set properties for the database, such as whether it can be created if it does not currently exist, whether you are opening it read-only, and whether the database is to support transactions.

Note that by default, JE does not create databases if they do not already exist. To override this behavior, set the [creation property](#) to true.

Finally, if you configured your environment and database to support transactions, you can optionally provide a transaction object to the `Environment.openDatabase()`. Transactions are described in the *Berkeley DB Java Edition Getting Started with Transaction Processing* guide.

The following code fragment illustrates a database open:

```
package je.gettingStarted;

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

import java.io.File;
...

Environment myDbEnvironment = null;
Database myDatabase = null;

...

try {
    // Open the environment. Create it if it does not already exist.
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setAllowCreate(true);
    myDbEnvironment = new Environment(new File("/export/dbEnv"), envConfig);

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

Deferred Write Databases

By default, JE's databases are all persistent. That is, the data they contain is stored on disk so that it can be accessed across program runs. However, it is possible to configure JE's databases so that they are by default *not persistent*. JE calls databases configured in this way to be *deferred write databases*.

Deferred write databases are essentially in-memory only databases. Therefore, they are particularly useful for applications that want databases which are truly temporary.

Note that deferred write databases do not always avoid disk I/O. It is particularly important to realize that deferred write databases can page to disk if the cache is not large enough to hold the database's entire contents. Therefore, deferred write database performance is best if your in-memory cache is large enough to hold the database's entire data-set.

Beyond that, you can deliberately cause data modifications made to a deferred write database to be made persistent. If `Database.sync()` is called before the application is shutdown, the contents of the deferred write database is saved to disk.

In short, upon reopening an environment and a deferred write database, the database is guaranteed to be in at least the same state it was in at the time of the last database sync. (It is possible that due to a full in-memory cache, the database will page to disk and so the database might actually be in a state sometime *after* the last sync.) This means that if the deferred write database is never sync'd, it is entirely possible for it to open as an empty database.

Because modifications made to a deferred write databases can be made persistent at a time that is easily controlled by the programmer, these types of databases are also useful for applications that perform a great deal of database modifications, record additions, deletions, and so forth. By delaying the data write, you delay the disk I/O. Depending on your workload, this can improve your data throughput by quite a lot.

Be aware that you lose modifications to a deferred write database only if you (1) do not call sync, (2) close the deferred write database and (3) also close the environment. If you only close the database but leave the environment opened, then all operations performed on that database since the time of the last environment open may be retained.

All other rules of behavior pertain to deferred write databases as they do to normal databases. Deferred write databases must be named and created just as you would a normal database. If you want to delete the deferred write database, you must remove it just as you would a normal database. This is true even if the deferred write database is empty because its name persists in the environment's namespace until such a time as the database is removed.

Note that determining whether a database is deferred write is a configuration option. It is therefore possible to switch a database between "normal" mode and deferred write database. You might want to do this if, for example, you want to load a lot of data to the database. In this case, loading data to the database while it is in deferred write state is faster than in "normal" state, because you can avoid a lot of the normal disk I/O overhead during the load process. Once the load is complete, sync the database, close it, and then reopen it as a normal database. You can then continue operations as if the database had been created as a "normal" database.

To configure a database as deferred write, set `DatabaseConfig.setDeferredWrite()` to `true` and then open the database with that `DatabaseConfig` option.

For example, the following code fragment opens and closes a deferred write database:

```
package je.gettingStarted;

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

```
import java.io.File;
...

Environment myDbEnvironment = null;
Database myDatabase = null;

...

try {
    // Open the environment. Create it if it does not already exist.
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setAllowCreate(true);
    myDbEnvironment = new Environment(new File("/export/dbEnv"), envConfig);

    // Open the database. Create it if it does not already exist.
    DatabaseConfig dbConfig = new DatabaseConfig();
    dbConfig.setAllowCreate(true);
    // Make it deferred write
    dbConfig.setDeferredWrite(true);
    myDatabase = myDbEnvironment.openDatabase(null,
                                              "sampleDatabase",
                                              dbConfig);

    ...
    // do work
    ...
    // Do this if you want the work to persist across
    // program runs
    // myDatabase.sync();

    // then close the database and environment here
    // (see the next section)

} catch (DatabaseException dbe) {
    // Exception handling goes here
}
```

Closing Databases

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

Closing a database causes it to become unusable until it is opened again. If any cursors are opened for the database, JE warns you about the open cursors, and then closes them for you. Active cursors during a database close can cause unexpected results, especially if any of those cursors are writing to the database in another thread. You should always make sure that all your database accesses have completed before closing your database.

Remember that for the same reason, you should always close all your databases before closing the environment to which they belong.

Cursors are described in [Using Cursors \(page 61\)](#) later in this manual.

The following illustrates database and environment close:

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

...

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

    if (myDbEnvironment != null) {
        myDbEnvironment.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.

The database properties that you can set 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 [Using Comparators \(page 47\)](#).

- `DatabaseConfig.setDuplicateComparator()`

Sets the class that is used to compare two duplicate records in the database. For more information, see [Using Comparators \(page 47\)](#).

- `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 \(page 78\)](#).

- `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.setTransactional()`

If `true`, the database supports transactions. Default is `false`. Note that a database cannot support transactions if the environment is non-transactional.

For example:

```
package je.gettingStarted;

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

...
// Environment open omitted for brevity
...

Database myDatabase = null;
try {
    DatabaseConfig dbConfig = new DatabaseConfig();
    dbConfig.setAllowCreate(true);
    dbConfig.setSortedDuplicates(true);
    myDatabase =
        myDbEnv.openDatabase(null,
                            "sampleDatabase",
                            dbConfig);
} catch (DatabaseException dbe) {
    // 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.getEnvironment()`

Returns the `Environment` that contains this database.

```
Environment theEnv = myDatabase.getEnvironment();
```

- `Database.preload()`

Preloads the database into the in-memory cache. Optionally takes a `long` that identifies the maximum number of bytes to load into the cache. If this parameter is not supplied, the maximum memory usage allowed by the evictor thread is used.

```
myDatabase.preload(10485761); // 1024*1024
```

- `Environment.getDatabaseNames()`

Returns a list of `Strings` of all the databases contained by the environment.

```
import java.util.List;
...
List myDbNames = myDbEnv.getDatabaseNames();
for(int i=0; i < myDbNames.size(); i++) {
    System.out.println("Database Name: " + (String)myDbNames.get(i));
}
```

- `Environment.removeDatabase()`

Deletes the database. The database must be closed when you perform this action on it.

```
String dbName = myDatabase.getDatabaseName();
myDatabase.close();
myDbEnv.removeDatabase(null, dbName);
```

- `Environment.renameDatabase()`

Renames the database. The database must be closed when you perform this action on it.

```
String oldName = myDatabase.getDatabaseName();
String newName = new String(oldName + ".new", "UTF-8");
myDatabase.close();
myDbEnv.renameDatabase(null, oldName, newName);
```

- `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());
```

Database Example

In [Database Environment Management Example \(page 16\)](#) we created a class that manages an `Environment`. We now extend that class to allow it to open and manage multiple databases. Again, remember that you can find this class in:

```
JE_HOME/je/gettingStarted/MyDbEnv.java
```

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

Example 3.1. Database Management with MyDbEnv

First, we need to import a few additional classes, and setup some global variables to support databases. The databases that we are configuring and creating here are used by applications developed in examples later in this guide.

```
// File MyDbEnv.java

package je.gettingStarted;

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

import java.io.File;
```

```
public class MyDbEnv {  
  
    private Environment myEnv;  
    private Database vendorDb;  
    private Database inventoryDb;  
  
    public MyDbEnv() {}  
}
```

Next we need to update the `MyDbEnv.setup()` method to instantiate a `DatabaseConfig` object. We also need to set some properties on that object. These property values are determined by the value of the `readOnly` parameter. We want our databases to be read-only if the environment is also read-only. We also want to allow our databases to be created if the databases are not read-only.

```
public void setup(File envHome, boolean readOnly)  
    throws DatabaseException {  
  
    // Instantiate an environment and database configuration object  
    EnvironmentConfig myEnvConfig = new EnvironmentConfig();  
    DatabaseConfig myDbConfig = new DatabaseConfig();  
    // Configure the environment and databases for the read-only  
    // state as identified by the readOnly parameter on this  
    // method call.  
    myEnvConfig.setReadOnly(readOnly);  
    myDbConfig.setReadOnly(readOnly);  
    // If the environment is opened for write, then we want to be  
    // able to create the environment and databases if  
    // they do not exist.  
    myEnvConfig.setAllowCreate(!readOnly);  
    myDbConfig.setAllowCreate(!readOnly);  
  
    // Instantiate the Environment. This opens it and also possibly  
    // creates it.  
    myEnv = new Environment(envHome, myEnvConfig);  
  
    // Now create and open our databases.  
    vendorDb = myEnv.openDatabase(null,  
                                  "VendorDB",  
                                  myDbConfig);  
  
    inventoryDb = myEnv.openDatabase(null,  
                                     "InventoryDB",  
                                     myDbConfig);  
}
```

Next we need some additional getter methods used to return our database handles.

```
// Getter methods  
public Environment getEnvironment() {  
    return myEnv;  
}
```

```

    }

    public Database getVendorDB() {
        return vendorDb;
    }

    public Database getInventoryDB() {
        return inventoryDb;
    }

```

Finally, we need to update the `MyDbEnv.close()` method to close our databases.

```

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

```

We can now use `MyDbEnv` to open and close both database environments and databases from the appropriate place in our application. For example:

```

package je.gettingStarted;

import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Database;

import java.io.File;

...

MyDbEnv exampleDbEnv = new MyDbEnv();

try {
    exampleDbEnv.setup(new File("/directory/currently/exists"), true);
    Database vendorDb = exampleDbEnv.getVendorDB();
    Database inventoryDB = exampleDbEnv.getInventoryDB();

    ...

} catch(DatabaseException dbe) {

```

```
    // Error code goes here
  } finally {
    exampleDbEnv.close();
  }
```

Chapter 4. Database Records

JE 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 JE 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 \(page 36\)](#) 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 JE 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 JE using `DatabaseEntry` objects as `byte` arrays. Using `DatabaseEntry`s allows JE 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 je.gettingStarted;

import com.sleepycat.je.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 je.gettingStarted;

import com.sleepycat.je.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, JE 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 \(page 36\)](#).

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*.

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

JE 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 \(page 61\)](#).

Writing Records to the Database

Database records are stored in the internal BTree based on whatever sorting routine is available to the database. Records are sorted first by their key. If the database supports duplicate records, then the records for a specific key are sorted by their data.

By default, JE sorts both keys and the data portion of duplicate records using unsigned byte-by-byte lexicographic comparisons. This default comparison works well for the majority of cases. However, in some case performance benefits can be realized by overriding the default comparison routine. See [Using Comparators \(page 47\)](#) for more information.

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 je.gettingStarted;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;

...

// Environment and database opens omitted for clarity.
// Environment and database 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 \(page 61\)](#).

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 je.gettingStarted;

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

...

// Environment and database opens omitted for clarity.
// Environment and 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 \(page 61\)](#).

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

For example:

```
package je.gettingStarted;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;

...

// Environment and database opens omitted for clarity.
// Environment and 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.

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, JE ensures that the data will not be lost due to application or system failure. Transaction usage is described in the *Berkeley DB Java Edition 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 JE 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.

Note that by default, a sync is run every time you close an environment. You can also run a sync by calling the `Environment.sync()` method.

For a brief description of how JE manages its data in the cache and in the log files, and how sync works, see [Databases and Log Files \(page 99\)](#).

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, JE provides the Bind APIs. While these APIs are described in detail in the *Berkeley DB Java Edition 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 je.gettingStarted;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.je.DatabaseEntry;

...

// 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(1234567891);
    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 je.gettingStarted;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.tuple.TupleBinding;
```

```
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.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, JE 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 JE 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 you 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.
- You are using custom comparators. In JE, comparators are instantiated and called internally whenever databases are not accessible. Because serial bindings depend on the class catalog, a serial binding cannot be used during these times. As a result, attempting to use a serial binding with a custom comparator will result in a `NullPointerException` during environment open or close.

For information on building your own custom tuple binding, see [Custom Tuple Bindings \(page 43\)](#).

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 je.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 je.gettingStarted;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.StoredClassCatalog;
import com.sleepycat.bind.serial.SerialBinding;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseEntry;

...

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

// The data data
MyData data2Store = new MyData();
data2Store.setLong(1234567891);
data2Store.setDouble(1234.9876543);
data2Store.setDescription("A test instance of this class");

try {
    // Environment open omitted for brevity

    // Open the database that you will use to store your data
    DatabaseConfig myDbConfig = new DatabaseConfig();
    myDbConfig.setAllowCreate(true);
    myDbConfig.setSortedDuplicates(true);
    Database myDatabase = myDbEnv.openDatabase(null, "myDb", 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 = myDbEnv.openDatabase(null, "classDb", 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 je.gettingStarted;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.StoredClassCatalog;
import com.sleepycat.bind.serial.SerialBinding;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.LockMode;

...

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

try {

```

```

// Environment open omitted for brevity.

// Open the database that stores your data
DatabaseConfig myDbConfig = new DatabaseConfig();
myDbConfig.setAllowCreate(false);
Database myDatabase = myDbEnv.openDatabase(null, "myDb", myDbConfig);

// Open the database that stores your class information.
Database myClassDb = myDbEnv.openDatabase(null, "classDb", 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 \(page 38\)](#).

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 your keys to be instances of the following class:

```
package je.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 je.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 je.gettingStarted;

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

...

TupleBinding keyBinding = new MyTupleBinding();

MyData2 theKeyData = new MyData2();
theKeyData.setLong(1234567891);
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  
}
```

Using Comparators

Internally, JE databases are organized as BTrees. This means that most database operations (inserts, deletes, reads, and so forth) involve BTree node comparisons. This comparison most frequently occurs based on database keys, but if your database supports duplicate records then comparisons can also occur based on the database data.

By default, JE performs all such comparisons using a byte-by-byte lexicographic comparison. This mechanism works well for most data. However, in some cases you may need to specify your own comparison routine. One frequent reason for this is to perform a language sensitive lexical ordering of string keys.

Writing Comparators

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).

JE passes 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 JE). In this case, your comparator would look like the following:

```
package je.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, "UTF-8");  
        String s2 = new String(b2, "UTF-8");
```

```

        return s1.compareTo(s2);
    }
}

```

Setting Comparators

You specify a `Comparator` using the following methods. Note that by default these methods can only be used at database creation time, and they are ignored for normal database opens. Also, note that JE uses the no-argument constructor for these comparators. Further, it is not allowable for there to be a mutable state in these comparators or else unpredictable results will occur.

- `DatabaseConfig.setBtreeComparator()`

Sets the Java `Comparator` class used to compare two keys in the database.

- `DatabaseConfig.setDuplicateComparator()`

Sets the Java `Comparator` class used to compare the data on two duplicate records in the database. This comparator is used only if the database supports duplicate records.

You can use the above methods to set a database's comparator after database creation time if you explicitly indicate that the comparator is to be overridden. You do this by using the following methods:

 If you override your comparator, the new comparator must preserve the sort order implemented by your original comparator. That is, the new comparator and the old comparator must return the same value for the comparison of any two valid objects. Failure to observe this constraint will cause unpredictable results for your application.

If you want to change the fundamental sort order for your database, back up the contents of the database, delete the database, recreate it, and then reload its data.

- `DatabaseConfig.setOverrideBtreeComparator()`

If set to `true`, causes the database's Btree comparator to be overridden with the `Comparator` specified on `DatabaseConfig.setBtreeComparator()`. This method can be used to change the comparator post-environment creation.

- `DatabaseConfig.setOverrideDuplicateComparator()`

If set to `true`, causes the database's duplicates comparator to be overridden with the `Comparator` specified on `DatabaseConfig.setDuplicateComparator()`.

For example, to use the `Comparator` described in the previous section:

```

package je.gettingStarted;

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

```

```

import java.util.Comparator;

...

// Environment open omitted for brevity

try {
    // Get the database configuration object
    DatabaseConfig myDbConfig = new DatabaseConfig();
    myDbConfig.setAllowCreate(true);

    // Set the duplicate comparator class
    myDbConfig.setDuplicateComparator(MyDataComparator.class);

    // Open the database that you will use to store your data
    myDbConfig.setSortedDuplicates(true);
    Database myDatabase = myDbEnv.openDatabase(null, "myDb", myDbConfig);
} catch (DatabaseException dbe) {
    // Exception handling goes here
}

```

Database Record Example

In [Database Example \(page 26\)](#), we created `MyDbEnv`, a class that manages `DatabaseEnvironment` and `Database` opens and closes. We will now write an application that takes advantage of this class to open databases, put a series of records in them, and then close the databases and environment.

Remember that all of the classes and programs presented here can be found in the following directory:

```
JE_HOME/examples/je/gettingStarted
```

where `JE_HOME` is the location where you placed your JE 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 4.1. 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 je.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 4.2. Vendor.java

The data for vendor records are stored in instances of the following class. Notice that we are using serialization with this class simply to demonstrate serializing a class instance.

```
// File Vendor.java
package je.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 4.3. `InventoryBinding.java`

```

// File InventoryBinding.java
package je.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 `ExampleDatabasePut` 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 \(page 49\)](#) for more information).

`ExampleDatabasePut` 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 \(page 51\)](#) for more information). To load this `Database`, `ExampleDatabasePut` 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 4.4. Stored Class Catalog Management with MyDbEnv

Before we can write `ExampleDatabasePut`, we need to update `MyDbEnv.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 MyDbEnv.java
package je.gettingStarted;

```

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

import java.io.File;

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

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 MyDbEnv {

    private Environment myEnv;
    private Database vendorDb;
    private Database inventoryDb;
    private Database classCatalogDb;

    // Needed for object serialization
    private StoredClassCatalog classCatalog;

    public MyDbEnv() {}
```

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

```
public void setup(File envHome, boolean readOnly)
    throws DatabaseException {

    ...
    // Database and environment configuration omitted for brevity
    ...

    // Instantiate the Environment. This opens it and also possibly
    // creates it.
    myEnv = new Environment(envHome, myEnvConfig);

    // Now create and open our databases.
    vendorDb = myEnv.openDatabase(null, "VendorDB", myDbConfig);

    inventoryDb = myEnv.openDatabase(null, "InventoryDB", myDbConfig);

    // Open the class catalog db. This is used to
    // optimize class serialization.
    classCatalogDb =
        myEnv.openDatabase(null,
```

```

        "ClassCatalogDB",
        myDbConfig);

    // Create our class catalog
    classCatalog = new StoredClassCatalog(classCatalogDb);
}

```

Next 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.

```

// Getter methods
public Environment getEnvironment() {
    return myEnv;
}

public Database getVendorDB() {
    return vendorDb;
}

public Database getInventoryDB() {
    return inventoryDb;
}

public StoredClassCatalog getClassCatalog() {
    return classCatalog;
}

```

Finally, we need to update the `MyDbEnv.close()` method to close the class catalog database.

```

// Close the environment
public void close() {
    if (myEnv != null) {
        try {
            vendorDb.close();
            inventoryDb.close();
            classCatalogDb.close()
            myEnv.close();
        } catch(DatabaseException dbe) {
            System.err.println("Error closing MyDbEnv: " +
                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 `MyDbEnv` to manage our databases for us. Now we write `ExampleDatabasePut`

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 4.5. ExampleDatabasePut.java

First we need the usual series of import statements:

```
//File ExampleDatabasePut.java
package je.gettingStarted;

// Bind classes used to move class objects in an out of byte arrays.
import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.bind.tuple.TupleBinding;

// Standard JE database imports
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;

// Most of this is used for loading data from a text file for storage
// in the databases.
import java.io.File;
import java.io.FileInputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
```

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 MyDbEnv \(page 53\)](#) 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 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");

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

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

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.je.*` classes.

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

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

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.err.println("Exception: " + e.toString());
        e.printStackTrace();
    } finally {
        myDbEnv.close();
    }
    System.out.println("All done.");
}
```

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

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

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

    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(myDbEnv.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. These puts are transactionally
        // protected (we're using autocommit).
        myDbEnv.getVendorDB().put(null, theKey, theData);
    }
}
```

```

    }
}

```

Now load the inventory database. This method uses our custom tuple binding (see [InventoryBinding.java \(page 52\)](#)) 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]);
        theInventory.setVendorPrice((new Float(sArray[2])).floatValue());
        theInventory.setVendorInventory(
            (new Integer(sArray[3])).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.
        myDbEnv.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. From the perspective of this

document, these things are relatively uninteresting. You can see how they are implemented by looking at:

```
JE_HOME/examples/je/gettingStarted/ExampleDataPut.java
```

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

```
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 ExampleDatabasePut() {}
}
```

Chapter 5. 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 only mechanism by which you can access anything other than the first duplicate 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. Currently, the only available property tells the cursor to perform uncommitted reads. (For a description of uncommitted reads, see the *Berkeley DB Java Edition Getting Started with Transaction Processing* guide.)

For example:

```
package je.gettingStarted;

import com.sleepycat.je.Cursor;
import com.sleepycat.je.CursorConfig;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;

import java.io.File;

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

try {
    myDbEnvironment = new Environment(new File("/export/dbEnv"), null);
    myDatabase = myDbEnvironment.openDatabase(null, "myDB", null);

    myCursor = myDatabase.openCursor(null, null);
} 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 je.gettingStarted;

import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.Environment;

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

        if (myDatabase != null) {
            myDatabase.close();
        }

        if (myDbEnvironment != null) {
            myDbEnvironment.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 je.gettingStarted;

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

...

Cursor cursor = null;
try {
    ...
    // Database and environment 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 je.gettingStarted;

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

...

Cursor cursor = null;
try {
    ...
    // Database and environment 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()`

Moves the cursor 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 unsigned byte-by-byte 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 you have 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 je.gettingStarted;

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

...

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

Cursor cursor = null;
try {
    ...
    // Database and environment 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

If your database supports duplicate records, then it can potentially contain multiple records that share the same key. Using normal database get operations, you can only ever obtain the first such record in a set of duplicate records. To access subsequent duplicates, use 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 \(page 62\)](#).

- `Cursor.getSearchBothRange()`

Useful for seeking the cursor to a specific record, regardless of whether it is a duplicate record. See [Searching for Records \(page 64\)](#) 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.

- `Cursor.getNextDup()`, `Cursor.getPrevDup()`

Gets the next/previous 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, if the key contains duplicate records, displays all the duplicates. Note that the following code fragment assumes that the database contains only `String` objects for the keys and data.

```
package je.gettingStarted;

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

...

Cursor cursor = null;
try {
    ...
    // Database and environment 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. JE's behavior when putting records into the database differs depending on whether the database supports duplicate records. If duplicates are allowed, its behavior also differs depending on whether a comparator is provided for the database. (Comparators are described in [Using Comparators \(page 47\)](#)).

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

You can use the following methods to put records to the database:

- `Cursor.put()`

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

If the provided key already exists in the database, and the database does not support sorted duplicates, then the existing record data is replaced with the data provided on this method.

If the provided key already exists in the database, and the database does support sorted duplicates, then the order that the record is inserted into the database is determined by the duplicate comparator in use by the database.

- `Cursor.putNoDupData()`

If the provided key and data 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 BTree (key) comparator in use by the database.

- `Cursor.putNoOverwrite()`

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 `BTree (key) comparator` in use by the database.

For example:

```
package je.gettingStarted;

import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.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;
try {
    ...
    // Database and environment 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 `Cursor.delete()`. Note that after deleting a record, the value of `Cursor.getCurrent()` is unchanged until such a time as the cursor is moved again. Also, if you call `Cursor.delete()` two or more times in a row without repositioning the cursor, then all subsequent deletes result in a return value of `OperationStatus.KEYEMPTY`.

For example:

```
package je.gettingStarted;

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

...

Cursor cursor = null;
try {
    ...
    // Database and environment 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()`. This method takes just one argument – the data that you want to write to the current location in the database.

```

import com.sleepycat.je.Cursor;

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

...
Cursor cursor = null;
try {
    ...
    // Database and environment 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 this method cannot be used if the record that you are trying to replace is a member of a duplicate set. This is because records must be sorted by their data and replacement would violate that sort order.

If you want to replace the data contained by a duplicate record, delete the record and create a new record with the desired key and data.

Cursor Example

In [Database Example \(page 26\)](#) 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 `ExampleInventoryRead` application. This application reads and displays all inventory records by:

1. Opening the environment and then the inventory, vendor, and class catalog `Database` objects. We do this using the `MyDbEnv` class. See [Stored Class Catalog Management with MyDbEnv \(page 53\)](#) 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 \(page 52\)](#) 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 \(page 51\)](#). We implemented the `Inventory` class in [Inventory.java \(page 49\)](#).

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

```
JE_HOME/examples/je/gettingStarted/ExampleInventoryRead.java
```

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

Example 5.1. ExampleInventoryRead.java

To begin, we import the necessary classes:

```
// file ExampleInventoryRead.java
package je.gettingStarted;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.bind.tuple.TupleBinding;

import com.sleepycat.je.Cursor;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;

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

Next we declare our class and set up some global variables. Note a `MyDbEnv` object is instantiated here. We can do this because its constructor never throws an exception. See [Database Example \(page 26\)](#) for its implementation details.

```
public class ExampleInventoryRead {

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

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

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

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

```
private static void usage() {
    System.out.println("ExampleInventoryRead [-h <env directory>]");
    System.exit(0);
}

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.");
}

```

In `ExampleInventoryRead.run()`, we call `MyDbEnv.setup()` to open our environment and 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);

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

    // Setup our bindings.
    inventoryBinding = new InventoryBinding();
    vendorBinding =
        new SerialBinding(myDbEnv.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 = myDbEnv.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 `ExampleInventoryRead.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 {

    DatabaseEntry searchKey = null;
    try {
        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: ");

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

    if (myDbEnv.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());
    }
}

```

Chapter 6. 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 JE, 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*, and these are managed using `SecondaryDatabase` class objects. 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, JE manages secondary databases for you. Adding or deleting records in your primary database causes JE to update the secondary as necessary. Further, changing a record's data in the primary database may cause JE 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. While methods exist on `SecondaryDatabase` and `SecondaryCursor` that appear to allow this, they in fact always throw `UnsupportedOperationException`. 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 \(page 85\)](#) for more information.



Secondary database records are updated/created by JE only if the `SecondaryKeyCreator.createSecondaryKey()` method returns `true`. If `false` is returned, then JE 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 \(page 81\)](#) for more information on this interface and method.

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

Opening and Closing Secondary Databases

You manage secondary database opens and closes using the `Environment.openSecondaryDatabase()` method. Just as is the case with primary databases, you must provide `Environment.openSecondaryDatabase()` 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 \(page 84\)](#) 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 je.gettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.SecondaryDatabase;
import com.sleepycat.je.SecondaryConfig;

import java.io.File;
```

```
...

DatabaseConfig myDbConfig = new DatabaseConfig();
SecondaryConfig mySecConfig = new SecondaryConfig();
myDbConfig.setAllowCreate(true);
mySecConfig.setAllowCreate(true);
// Duplicates are frequently required for secondary databases.
mySecConfig.setSortedDuplicates(true);

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

    myEnv = new Environment(new File("/tmp/JEENV"), null);
    myDb = myEnv.openDatabase(null, dbName, myDbConfig);

    // A fake tuple binding that is not actually implemented anywhere
    // in this manual. 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 = myEnv.openSecondaryDatabase(null, secDbName, myDb,
                                          mySecConfig);
} catch (DatabaseException de) {
    // 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();
    }

    if (myEnv != null) {
        myEnv.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 \(page 36\)](#).

`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 je.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 \(page 43\)](#).)

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 je.gettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;

import com.sleepycat.je.SecondaryKeyCreator;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.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 je.gettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.SecondaryDatabase;
import com.sleepycat.je.SecondaryConfig;

...
Environment myEnv = null;
Database myDb = null;
SecondaryDatabase mySecDb = null;
try {
    // Environment and primary database open omitted for brevity
    ...

    TupleBinding myDataBinding = new MyTupleBinding();
    FullNameKeyCreator fnkc = new FullNameKeyCreator(myDataBinding);

    SecondaryConfig mySecConfig = new SecondaryConfig();
    mySecConfig.setKeyCreator(fnkc);

    //Perform the actual open
    String secDbName = "mySecondaryDatabase";
    mySecDb = myEnv.openSecondaryDatabase(null, secDbName, myDb,
                                          mySecConfig);
} catch (DatabaseException de) {
    // Exception handling goes here
} finally {
    try {
        if (mySecDb != null) {
            mySecDb.close();
        }

        if (myDb != null) {

```

```
        myDb.close();
    }

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

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 \(page 23\)](#) 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 \(page 81\)](#) 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 je.gettingStarted;

import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
import com.sleepycat.je.SecondaryDatabase;

...
```

```
try {
    // Omitting all database and environment 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 \(page 86\)](#)).

Deleting Secondary Database Records

In general, you can not modify a secondary database directly. In order to modify a secondary database, you should modify the primary database and simply allow JE 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 JE 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 database supports duplicate records, then only the first record in the matching duplicates set is deleted by this method. To delete all the duplicate records that use a given key, use a `SecondaryCursor`.



`SecondaryDatabase.delete()` causes the previously described delete operations to occur only if the primary database is opened for write access.

For example:

```
package je.gettingStarted;

import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.OperationStatus;
import com.sleepycat.je.SecondaryDatabase;
```

```
...
try {
    // Omitting all database and environment 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, to get the next non-duplicate record, and so forth. For a complete description on cursors and their capabilities, see [Using Cursors \(page 61\)](#).

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 \(page 81\)](#). 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 je.gettingStarted;

import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
import com.sleepycat.je.SecondaryDatabase;
import com.sleepycat.je.SecondaryCursor;
```

```
...
try {
    // Database and environment 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 items. 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 \(page 30\)](#).

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 must be obtained from 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 je.gettingStarted;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.JoinCursor;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
```

```
import com.sleepycat.je.SecondaryCursor;
import com.sleepycat.je.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

// 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 JE databases. In this example, we will extend those examples to use secondary databases. Specifically:

- In [Stored Class Catalog Management with MyDbEnv \(page 53\)](#) we built a class that we can use to open and manage a `JE Environment` and one or more `Database` objects. In [Opening Secondary Databases with MyDbEnv \(page 93\)](#) we will extend that class to also open and manage a `SecondaryDatabase`.
- In [Cursor Example \(page 73\)](#) we built an application to display our inventory database (and related vendor information). In [Using Secondary Databases with ExampleInventoryRead \(page 96\)](#) we will extend that application to show inventory records based on the index we cause to be loaded using `ExampleDatabasePut`.

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 6.1. `ItemNameKeyCreator.java`

This class assumes the primary database uses `Inventory` objects for the record data. The `Inventory` class is described in [Inventory.java \(page 49\)](#).

In our key creator class, we make use of a custom tuple binding called `InventoryBinding`. This class is described in [InventoryBinding.java \(page 52\)](#).

You can find the following class in:

```
JE_HOME/examples/je/gettingStarted/ItemNameKeyCreator.java
```

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

```
package je.gettingStarted;

import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.SecondaryDatabase;
import com.sleepycat.je.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 `MyDbEnv` to manage a secondary database, and to use `ItemNameKeyCreator` to generate keys for that secondary database.

Opening Secondary Databases with MyDbEnv

In [Stored Class Catalog Management with MyDbEnv \(page 53\)](#) we built `MyDbEnv` as an example of a class that encapsulates `Environment` and `Database` opens and closes. We will now extend that class to manage a `SecondaryDatabase`.

Example 6.2. SecondaryDatabase Management with MyDbEnv

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 MyDbEnv.java

package je.gettingStarted;

import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.bind.serial.StoredClassCatalog;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.je.SecondaryConfig;
import com.sleepycat.je.SecondaryDatabase;

import java.io.File;

public class MyDbEnv {

    private Environment myEnv;

    // The databases that our application uses
    private Database vendorDb;
    private Database inventoryDb;
    private Database classCatalogDb;
    private SecondaryDatabase itemNameIndexDb;

    // Needed for object serialization
    private StoredClassCatalog classCatalog;

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

Next we update the `MyDbEnv.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 \(page 52\)](#)). We do all this work together inside of `MyDbEnv.setup()`.

```
public void setup(File envHome, boolean readOnly)
    throws DatabaseException {

    EnvironmentConfig myEnvConfig = new EnvironmentConfig();
    DatabaseConfig myDbConfig = new DatabaseConfig();
    SecondaryConfig mySecConfig = new SecondaryConfig();

    // If the environment is read-only, then
    // make the databases read-only too.
    myEnvConfig.setReadOnly(readOnly);
    myDbConfig.setReadOnly(readOnly);
    mySecConfig.setReadOnly(readOnly);

    // If the environment is opened for write, then we want to be
    // able to create the environment and databases if
    // they do not exist.
    myEnvConfig.setAllowCreate(!readOnly);
    myDbConfig.setAllowCreate(!readOnly);
    mySecConfig.setAllowCreate(!readOnly);

    ...
    // Environment and database opens omitted for brevity
    ...

    // 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 the secondary properties
    mySecConfig.setAllowPopulate(true); // Allow autopopulate
    mySecConfig.setKeyCreator(keyCreator);
    // Need to allow duplicates for our secondary database
    mySecConfig.setSortedDuplicates(true);

    // Now open it
    itemNameIndexDb =
        myEnv.openSecondaryDatabase(
            null,
```

```

        "itemNameIndex", // Index name
        inventoryDb,     // Primary database handle. This is
                        // the db that we're indexing.
        mySecConfig);   // The secondary config
    }

```

Next we need an additional getter method for returning the secondary database.

```

public SecondaryDatabase getNameIndexDB() {
    return itemNameIndexDb;
}

```

Finally, we need to update the `MyDbEnv.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() {
    if (myEnv != null) {
        try {
            //Close the secondary before closing the primaries
            itemNameIndexDb.close();
            vendorDb.close();
            inventoryDb.close();
            classCatalogDb.close();

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

```

That completes our update to `MyDbEnv`. You can find the complete class implementation in:

```

JE_HOME/examples/je/gettingStarted/MyDbEnv.java

```

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

Because we performed all our secondary database configuration management in `MyDbEnv`, we do not need to modify `ExampleDatabasePut` at all in order to create our secondary indices. When `ExampleDatabasePut` calls `MyDbEnv.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 `ExampleInventoryRead` 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.

Using Secondary Databases with `ExampleInventoryRead`

In the previous section we changed `MyDbEnv` to cause a secondary database to be built using inventory item names as the secondary keys. In this section, we will update `ExampleInventoryRead` to allow us to query our inventory records based on the item name. To do this, we will modify `ExampleInventoryRead` 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 `ExampleInventoryRead`, 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 the following class in:

```
JE_HOME/examples/je/gettingStarted/ExampleInventoryRead.java
```

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

Example 6.3. SecondaryDatabase usage with `ExampleInventoryRead`

First we need to import a few additional classes in order to use secondary databases and cursors, and then we add a single global variable:

```
package je.gettingStarted;

import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
import com.sleepycat.je.SecondaryCursor;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.bind.tuple.TupleBinding;

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

public class ExampleInventoryRead {

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

```

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

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

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

    if (locateItem != null) {
        showItem();
    } else {
        showAllInventory();
    }
}

```

Finally, we need to implement `ExampleInventoryRead.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 =
    myDbEnv.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 `ExampleInventoryRead.parseArgs()` to support the `-s` command line switch. To see how this is done, see:

```
JE_HOME/examples/je/gettingStarted/ExampleInventoryRead.java
```

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

Chapter 7. Backing up and Restoring Berkeley DB Java Edition Applications

Fundamentally, you backup your databases by copying JE log files off to a safe storage location. To restore your database from a backup, you copy those files to an appropriate directory on disk and reopen your JE application

Beyond these simple activities, there are some differing backup strategies that you may want to consider. These topics are described in this chapter.

Databases and Log Files

Before describing JE backup and restore, it is necessary to describe some of JE's internal workings. In particular, a high-level understanding of JE log files and the in-memory cache is required. You also need to understand a little about how JE is using its internal data structures in order to understand why checkpoints and/or syncs are required.

You can skip this section so long as you understand that:

- JE databases are stored in log files contained in your environment directory.
- Every time a JE environment is opened, normal recovery is run.
- For transactional applications, checkpoints should be run in order to bound normal recovery time. Checkpoints are normally run by the checkpointer thread. Transactional applications and the checkpointer thread are described in the *Berkeley DB Java Edition Getting Started with Transaction Processing* guide.
- For non-transactional applications, environment syncs must be performed if you want to guarantee the persistence of your database modifications. Environment syncs are manually performed by the application developer. See [Data Persistence \(page 35\)](#) for details.

Log File Overview

Your JE database is stored on-disk in a series of log files. JE uses no-overwrite log files, which is to say that JE only ever appends data to the end of a log file. It will never delete or modify an existing log file record.

JE log files are named `NNNNNNNN.jdb` where `NNNNNNNN` is an 8-digit hexadecimal number that increases by 1 (starting from `00000000`) for each log file written to disk.

JE creates a new log file whenever the current log file has reached a pre-configured size (10000000 bytes by default). This size is controlled by the `je.log.fileMax` properties parameter. See [The JE Properties File \(page 106\)](#) for information on setting JE properties.

Cleaning the Log Files

Because JE uses no-overwrite log files, the logs must be compacted or cleaned so as to conserve disk space.

JE uses the cleaner background thread to perform this task. When it runs, the cleaner thread picks the log file with the smallest number of active records and scans each log record in it. If the record is no longer active in the database tree, the cleaner does nothing. If the record is still active in the tree, then the cleaner copies the record forward to a newer log file.

Once a log file is no longer needed (that is, it no longer contains active records), then the cleaner thread deletes the log file for you. Or, optionally, the cleaner thread can simply rename the discarded log file with a `del` suffix.

JE uses a minimum log utilization property to determine how much cleaning to perform. The log files contain both obsolete and utilized records. Obsolete records are records that are no longer in use, either because they have been modified or because they have been deleted. Utilized records are those records that are currently in use. The `je.cleaner.minUtilization` property identifies the minimum percentage of log space that must be used by utilized records. If this minimum percentage is not met, then log files are cleaned until the minimum percentage is met.

For information on managing the cleaner thread, see [The Cleaner Thread \(page 107\)](#).

The BTree

JE databases are internally organized as a BTree. In order to operate, JE requires the complete BTree be available to it.

When database records are created, modified, or deleted, the modifications are represented in the BTree's leaf nodes. Beyond leaf node changes, database record modifications can also cause changes to other BTree nodes and structures.

Database Modifications and Syncs

When a write operation is performed in JE, the modified data is written to a leaf node contained in the in-memory cache. If your JE writes are performed without transactions, then the in-memory cache is the only location guaranteed to receive a database modification without further intervention on the part of the application developer.

For some class of applications, this lack of a guaranteed write to disk is ideal. By not writing these modifications to the on-disk logs, the application can avoid most of the overhead caused by disk I/O.

However, if the application requires its data to persist at a specific point in time, then the developer must manually sync database modifications to the on-disk log files (again, this is only necessary for non-transactional applications). This is done using `Environment.sync()`.

Note that syncing the cache causes JE to write all modified objects in the cache to disk. This is probably the most expensive operation that you can perform in JE.

Normal Recovery

Every time a JE environment is opened, normal recovery is run. Because of the way that JE organizes and manages its BTrees, all it needs is leaf nodes in order to recreate the rest of the BTree. Essentially, this is what *normal recovery* is doing - recreating any missing parts of the internal BTree from leaf node information stored in the log files.

Unlike a traditional database system, JE performs recovery for both transactional and non-transactional operations. The integrity of the Btree is guaranteed by JE in the face of both application and OS crashes.

Performing Backups

This section describes how to backup your JE database(s) such that catastrophic recovery is possible for non-transactional applications. Note that this same material is repeated in the *Berkeley DB Java Edition Getting Started with Transaction Processing* guide, but for transactional applications. If you are writing transactional applications, you may want to skip the rest of this chapter and go straight to that book.

To backup your database, you can either take a hot backup or an offline backup. A hot backup is performed while database write operations are in progress.

Do not confuse hot and offline backups with the concept of a full and incremental backup. Both a hot and an offline backup are full backups - you back up the entire database. The only difference between them is how much of the contents of the in-memory cache are contained in them. On the other hand, an incremental backup is a backup of just those log files modified or created since the time of the last backup. Most backup software is capable of performing both full and incremental backups for you.

Performing a Hot Backup

To perform a hot backup of your JE databases, copy all log files (*.jdb files) from your environment directory to your archival location or backup media. The files must be copied in alphabetical order (numerical in effect). You do not have to stop any database operations in order to do this.

To make this process a bit easier, you may want to make use of the DbBackup helper class. See [Using the DbBackup Helper Class \(page 102\)](#) for details.

Note that any modifications made to the database since the time of the last environment sync are not guaranteed to be contained in these log files. In this case, you may want to consider running an offline backup in order to guarantee the availability of all modifications made to your database.

Performing an Offline Backup

An offline backup guarantees that you have captured the database in its entirety, including all contents of your in-memory cache, at the moment that the backup was taken. To do this, you must make sure that no write operations are in progress and all database modifications have been written to your log files on disk. To obtain an offline backup:

1. Stop writing your databases.
2. Run `Environment.sync()` so as to ensure that all database modifications are written to disk. Note that cleanly closing your environment will also ensure that all database modifications are written to disk.
3. Copy all log files (*.jdb) from your environment directory to your archival location or backup media. To make this process a bit easier, you may want to make use of the `DbBackup` helper class. See the next section for details.

You can now resume normal database operations.

Using the DbBackup Helper Class

In order to simplify backup operations, JE provides the `DbBackup` helper class. This class stops and restarts JE background activity in an open environment. It also lets the application create a backup which can support restoring the environment to a specific point in time.

Because you do not have to stop JE write activity in order to take a backup, it is usually necessary to examine your log files twice before you decide that your backup is complete. This is because JE may create a new log file while you are running your backup. A second pass over your log files allows you to ensure that no new files have been created and so you can declare your backup complete.

For example:

time	files in environment	activity
t0	000000001.jdb 000000003.jdb 000000004.jdb	Backup starts copying file 1
t1	000000001.jdb 000000004.jdb 000000005.jdb	JE log cleaner migrates portion of file 3 to newly created file 5 and deletes file 3. Backup finishes file 1, starts copying file 4. Backup MUST include file 5 for a consistent backup!
t2	000000001.jdb 000000004.jdb 000000005.jdb	Backup finishes copying file 4, starts and finishes file 5, has caught up. Backup ends.

DbBackup works around this problem by defining the set of files that must be copied for each backup operation, and freezes all changes to those files. The application can copy that defined set of files and finish operation without checking for the ongoing creation of new files. Also, there will be no need to check for a newer version of the last file on the next backup.

In the example above, if DbBackup was used at t0, the application would only have to copy files 1, 3 and 4 to back up. On a subsequent backup, the application could start its copying at file 5. There would be no need to check for a newer version of file 4.

The following code fragment illustrates this class' usage:

```
package je.gettingStarted;

...
import com.sleepycat.je.util.DbBackup;
...

Environment env = new Environment(...);
DbBackup backupHelper = new DbBackup(env);

// Find the file number of the last file in the previous backup
// persistently, by either checking the backup archive, or saving
// state in a persistent file.
long lastFileCopiedInPrevBackup = ...

// Start backup, find out what needs to be copied.
backupHelper.startBackup();
try {
    String[] filesForBackup =
        backupHelper.getLogFilesInBackupSet(lastFileCopiedInPrevBackup);

    // Copy the files to archival storage.
    myApplicationCopyMethod(filesForBackup)
    // Update our knowlege of the last file saved in the backup set,
    // so we can copy less on the next backup
    lastFileCopiedInPrevBackup = backupHelper.getLastFileInBackupSet();
    myApplicationSaveLastFile(lastFileCopiedInBackupSet);
}
finally {
    // Remember to exit backup mode, or all log files won't be cleaned
    // and disk usage will bloat.
    backupHelper.endBackup();
}
```

Performing Catastrophic Recovery

Catastrophic recovery is necessary whenever your environment and/or database have been lost or corrupted due to a media failure (disk failure, for example). Catastrophic recovery is also required if normal recovery fails for any reason.

In order to perform catastrophic recovery, you must have a full back up of your databases. You will use this backup to restore your database. See [Performing Backups \(page 101\)](#) for information on running back ups.

To perform catastrophic recovery:

1. Shut down your application.
2. Delete the contents of your environment home directory (the one that experienced a catastrophic failure), if there is anything there.
3. Copy your most recent full backup into your environment home directory.
4. If you are using a backup utility that runs incremental backups of your environment directory, copy any log files generated since the time of your last full backup. Be sure to restore all log files in the order that they were written. The order is important because it is possible the same log file appears in multiple archives, and you want to run recovery using the most recent version of each log file.
5. Open the environment as normal. JE's normal recovery will run, which will bring your database to a consistent state relative to the changed data found in your log files.

You are now done restoring your database.

Hot Standby

As a final backup/recovery strategy, you can create a hot standby. Note that using hot standbys requires your application to be able to specify its environment home directory at application startup time. Most application developers allow the environment home directory to be identified using a command line option or a configuration or properties file. If your application has its environment home hard-coded into it, you cannot use hot standbys.

You create a hot standby by periodically backing up your database to an alternative location on disk. Usually this alternative location is on a separate physical drive from where you normally keep your database, but if multiple drives are not available then you should at least put the hot standby on a separate disk partition.

You failover to your hot standby by causing your application to reopen its environment using the hot standby location.

Note that a hot standby should not be used as a substitute for backing up and archiving your data to a safe location away from your operating environment. Even if your data is spread across multiple physical disks, a truly serious catastrophe (fires, malevolent

software viruses, faulty disk controllers, and so forth) can still cause you to lose your data.

To create and maintain a hot standby:

1. Copy all log files (*.jdb) from your environment directory to the location where you want to keep your standby. Either a hot or an offline backup can be used for this purpose, but typically a hot standby is initially created by taking an offline backup of your database. This ensures that you have captured the contents of your in-memory cache.
2. Periodically copy to your standby directory any log files that were changed or created since the time of your last copy. Most backup software is capable of performing this kind of an incremental backup for you.

Note that the frequency of your incremental copies determines the amount of data that is at risk due to catastrophic failures. For example, if you perform the incremental copy once an hour then at most your hot standby is an hour behind your production database, and so you are risking at most an hours worth of database changes.

3. Remove any *.jdb files from the hot standby directory that have been removed or renamed to .del files in the primary directory. This is not necessary for consistency, but will help to reduce disk space consumed by the hot standby.

Chapter 8. Administering Berkeley DB Java Edition Applications

There are a series of tools and parameters of interest to the administrator of a Berkeley DB Java Edition database. These tools and parameters are useful for tuning your JE database's behavior once it is in a production setting, and they are described here. This chapter, however, does not describe backing up and restoring your JE databases. See [Backing up and Restoring Berkeley DB Java Edition Applications \(page 99\)](#) for information on how to perform those procedures.

The JE Properties File

JE applications can be controlled through a Java properties file. This file must be placed in your environment home directory and it must be named `je.properties`.

The parameters set in this file take precedence over the configuration behavior coded into the JE application by your application developers.

Usually you will use this file to control the behavior of JE's background threads, and to control the size of your in-memory cache. These topics, and the properties parameters related to them, are described in this chapter. Beyond the properties described here, there are other properties identified throughout this manual that may be of interest to you. However, the definitive identification of all the property parameters available to you is in the sample `example.properties` located in the directory where your JE distribution was unpacked.

Managing the Background Threads

JE uses some background threads to keep your database resources within pre-configured limits. If they are going to run, the background threads are started once per application per process. That is, if your application opens the same environment multiple times, the background threads will be started just once for that process. See the following list for the default conditions that gate whether an individual thread is run. Note that you can prevent a background thread from running by using the appropriate `je.properties` parameter, but this is not recommended for production use and those parameters are not described here.

The background threads are:

- Cleaner thread.

Responsible for cleaning and deleting unused log files. See [The Cleaner Thread \(page 107\)](#) for more information.

This thread is run only if the environment is opened for write access.

- Compressor thread.

Responsible for cleaning up the internal BTree as database records are deleted. The compressor thread ensures that the BTree does not contain unused nodes. There is no need for you to manage the compressor and so it is not described further in this manual.

This thread is run only if the environment is opened for write access.

- Checkpointer thread.

Responsible for running checkpoints on your environment. See [The Checkpointer Thread \(page 108\)](#) for more information.

This thread always runs.

The Cleaner Thread

The cleaner thread is responsible for cleaning, or compacting, your log files for you. Log file cleaning is described in [Cleaning the Log Files \(page 100\)](#).

The following two properties may be of interest to you when managing the cleaner thread:

- `je.cleaner.minUtilization`

Identifies the percentage of the log file space that must be used for utilized records. If the percentage of log file space used by utilized records is too low, then the cleaner removes obsolete records until this threshold is reached. Default is 50%.

- `je.cleaner.expunge`

Identifies the cleaner's behavior in the event that it is able to remove a log file. If `true`, the log files that have been cleaned are deleted from the file system. If `false`, the log files that have been cleaned are renamed from `NNNNNNNNN.jdb` to `NNNNNNNNN.del`. You are then responsible for deleting the renamed files.

Note that the cleaner thread runs only if the environment is opened for write access. Also, be aware that the cleaner is not guaranteed to finish running before the environment is closed, which can result in unexpectedly large log files. See [Closing Database Environments \(page 12\)](#) for more information.

The Checkpointer Thread

Automatically runs checkpoints. Checkpoints and the administration of this thread are described in the *Berkeley DB Java Edition Getting Started with Transaction Processing* guide.

Sizing the Cache

By default, your cache is limited to a percentage of the JVM maximum memory as specified by the `-Xmx` parameter. You can change this percentage by using the `je.maxMemoryPercent` property or through `EnvironmentMutableConfig.setCachePercent()`. That is, the maximum amount of memory available to your cache is normally calculated as:

```
je.maxMemoryPercent * JVM_maximum_memory
```

You can find out what the value for this property is by using `EnvironmentConfig.getCachePercent()`.

Note that you can cause JE to use a fixed maximum cache size by using `je.maxMemory` or by using `EnvironmentConfig.setCacheSize()`.

Also, not every JVM is capable of identifying the amount of memory requested via the `-Xmx` parameter. For those JVMs you must use `je.maxMemory` to change your maximum cache size. The default maximum memory available to your cache in this case is 38M.

Of the amount of memory allowed for your cache, 93% is used for the internal BTree and the other 7% is used for internal buffers. When your application first starts up, the 7% for buffers is immediately allocated. The remainder of the cache grows lazily as your application reads and writes data.

In order for your application to start up successfully, the Java virtual machine must have enough memory available to it (as identified by the `-Xmx` command line switch) for both your application and 7% of your maximum cache value. In order for your application to run continuously (all the while loading data into the cache), you must make sure your JVM has enough memory for your application plus the maximum cache size.

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 the 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).

In order to determine how frequently your application is going to disk for database records not found in the cache, you can examine the value returned by `EnvironmentStats.getNCacheMiss()`.

`EnvironmentStats.getNCacheMiss()` identifies the total number of requests for database objects that were not serviceable from the cache. This value is cumulative since the application started. The faster this number grows, the more your application is going to disk to service database operations. Upon application startup you can expect this value to grow quite rapidly. However, as time passes and your cache is seeded with your most

frequently accessed database records, what you want is for this number's growth to be zero or at least very small.

Note that this statistic can only be collected from within the application itself or using the JMX extension (see [JMX Support \(page 8\)](#)).

For more information on collecting this statistic, see [Environment Statistics \(page 16\)](#).

The Command Line Tools

JE ships with several command line tools that you can use to help you manage your databases. They are:

- DbDump

Dumps a database to a user-readable format.

- DbLoad

Loads a database from the output produced by DbDump

- DbVerify

Verifies the structure of a database.

DbDump

Dumps a database to a flat-text representation. Options are:

-f

Identifies the file to which the output from this command is written. The console (standard out) is used by default.

-h

Identifies the environment's directory. This parameter is required.

-l

Lists the databases contained in the environment. If the **-s** is not provided, then this argument is required.

-p

Prints database records in human-readable format.

-r

Salvage data from a possibly corrupt file. When used on an uncorrupted database, this option should return data equivalent to a normal dump, but most likely in a different order.

This option causes the ensuing output to go to a file named *dbname.dump* where *dbname* is the name of the database you are dumping. The file is placed in the current working directory.

-R

Aggressively salvage data from a possibly corrupt file. This option differs from the `-r` option in that it will return all possible data from the file at the risk of also returning already deleted or otherwise nonsensical items. Data dumped in this fashion will almost certainly have to be edited by hand or other means before the data is ready for reload into another database.

This option causes the ensuing output to go to a file named `dbname.dump` where `dbname` is the name of the database you are dumping. The file is placed in the current working directory.

-s

Identifies the database to be dumped. If this option is not specified, then the `-l` is required.

-v

Prints progress information to the console for `-r` or `-R` mode.

-V

Prints the database version number and then quits. All other command line options are ignored.

For example:

```
> java com.sleepycat.je.util.DbDump -h . -p -s VendorDB
VERSION=3
format=print
type=btree
database=VendorDB
dupsort=false
HEADER=END
Mom's Kitchen
sr\01\01xpt\00\0d53 Yerman Ct.t\00\0c763 554 9200t\00\0bMiddle Townt\00
\0eMaggie Kultgent\00\10763 554 9200 x12t\00\02MNT\00\0dMom's Kitchent\00
\0555432
Off the Vine
sr\01\01xpt\00\10133 American Ct.t\00\0c563 121 3800t\00\0aCentennialt\00
\08Bob Kingt\00\10563 121 3800 x54t\00\02IAt\00\0cOff the Vinet\00\0552002
Simply Fresh
sr\01\01xpt\00\1115612 Bogart Lanet\00\0c420 333 3912t\00\08Harrigant\00
\0fCheryl Swedbergt\00\0c420 333 3952t\00\02WIt\00\0cSimply Fresht\00\0
553704
The Baking Pan
sr\01\01xpt\00\0e1415 53rd Ave.t\00\0c320 442 2277t\00\07Dutchint\00\09
Mike Roant\00\0c320 442 6879t\00\02MNT\00\0eThe Baking Pant\00\0556304
The Pantry
sr\01\01xpt\00\111206 N. Creek Wayt\00\0c763 555 3391t\00\0bMiddle Town
t\00\0fSully Beckstromt\00\0c763 555 3391t\00\02MNT\00\0aThe Pantryt\00
\0555432
TriCounty Produce
```

```
sr\01\01xpt\00\12309 S. Main Streett\00\0c763 555 5761t\00\0bMiddle Townt
\00\0dMort Dufresnet\00\0c763 555 5765t\00\02Mnt\00\11TriCounty Producet
\00\0555432
DATA=END
>
```

DbLoad

Loads a database from the output produced by DbDump. Options are:

-c

Specifies configuration options. The options supplied here override the corresponding options that appear in the data that is being loaded. This option takes values of the form *name=value*, where *name* is the configuration option that you are overriding and *value* is the new value for the option.

The following options can be specified:

- database

The name of the database to be loaded. This option duplicates the functionality of this command's `-s` command line option.

- dupsort

Indicates whether duplicates are allowed in the database. A value of `true` allows duplicates in the database.

-f

Identifies the file from which the database is to be loaded.

-n

Do not overwrite existing keys in the database when loading into an already existing database. If a key/data pair cannot be loaded into the database for this reason, a warning message is displayed on the standard error output, and the key/data pair are skipped

-h

Identifies the environment's directory. This parameter is required.

-l

Allows loading databases that were dumped with the Berkeley DB C product, when the dump file contains parameters not known to JE.

-s

Overrides the database name, causing the data to be loaded into a database that uses the name supplied to this parameter.

-T

Causes a flat text file to be loaded into the database.

The input must be paired lines of text, where the first line of the pair is the key item, and the second line of the pair is its corresponding data item.

A simple escape mechanism, where newline and backslash (\) characters are special, is applied to the text input. Newline characters are interpreted as record separators. Backslash characters in the text will be interpreted in one of two ways: If the backslash character precedes another backslash character, the pair will be interpreted as a literal backslash. If the backslash character precedes any other character, the two characters following the backslash will be interpreted as a hexadecimal specification of a single character; for example, \0a is a newline character in the ASCII character set.

For this reason, any backslash or newline characters that naturally occur in the text input must be escaped to avoid misinterpretation by `db_load`.

`-v`

Report periodic load status to the console.

`-V`

Prints the database version number and then quits. All other command line options are ignored.

For example:

```
> java com.sleepycat.je.util.DbDump -h . -s VendorDB -f vendordb.txt
> java com.sleepycat.je.util.DbLoad -h . -f vendordb.txt
>
```

DbVerify

Examines the identified database for errors. Options are:

`-h`

Identifies the environment's directory. This parameter is required.

`-q`

Suppress the printing of any error descriptions. Instead, simply exit success or failure.

`-s`

Identifies the database to be verified. This parameter is required.

`-V`

Prints the database version number and then quits. All other command line options are ignored.

`-v`

Report intermediate statistics every *N* leaf nodes, where *N* is the value that you provide this parameter.

For example:

```
> java com.sleepycat.je.util.DbVerify -h . -s VendorDB
```

```
<BtreeStats>  
<BottomInternalNodesByLevel total="1">  
  <Item level="1" count="1"/>  
</BottomInternalNodesByLevel>  
<InternalNodesByLevel total="1">  
  <Item level="2" count="1"/>  
</InternalNodesByLevel>  
<LeafNodes count="6"/>  
<DeletedLeafNodes count="0"/>  
<DuplicateCountLeafNodes count="0"/>  
<MainTreeMaxDepth depth="2"/>  
<DuplicateTreeMaxDepth depth="0"/>  
</BtreeStats>
```