Oracle Berkeley DB Java Edition

Java Collections Tutorial

Release 3.3

Legal Notice

Published 6/4/2008

Table of Contents

Preface

Welcome to the Berkeley DB Java Edition (JE) Collections API. This document provides a tutorial that introduces the collections API. The goal of this document is to provide you with an efficient mechanism with which you can quickly become efficient with this API. As such, this document is intended for Java developers and senior software architects who are looking for transactionally-protected backing of their Java collections. No prior experience with JE technologies 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;
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;
```

In situations in this book, programming examples are updated from one chapter to the next in this book. 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);
```

For More Information

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

- Getting Started with Berkeley DB Java Edition [http://www.oracle.com/technology/documentation/berkeley-db/je/GettingStartedGuide/BerkeleyDB-JE-GSG.pdf]
- Berkeley DB Java Edition Getting Started with Transaction Processing [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]

Chapter 1. Introduction

The JE Collections API is a Java framework that extends the well known Java Collections [http://java.sun.com/j2se/1.5.0/docs/guide/collections/] design pattern such that collections can now be stored, updated and queried in a transactional manner. The JE Collections API is a layer on top of JE.

Together the JE Collections API and Berkeley DB Java Edition provide an embedded data management solution with all the benefits of a full transactional storage and the simplicity of a well known Java API. Java programmers who need fast, scalable, transactional data management for their projects can quickly adopt and deploy the JE Collections API with confidence.

This framework was first known as Greybird DB [http://greybird-db.sourceforge.net/] written by Mark Hayes. Mark collaborated with us to permanently incorporate his excellent work into our distribution and to support it as an ongoing part of Berkeley DB and Berkeley DB Java Edition. The repository of source code that remains at SourceForge at version 0.9.0 is considered the last version before incorporation and will remain intact but will not be updated to reflect changes made as part of Berkeley DB or Berkeley DB Java Edition.

Features

JE provides a Java API that can be roughly described as a map and cursor interface, where the keys and values are represented as byte arrays. The JE Collections API is a layer on top of JE. It adds significant new functionality in several ways.

- An implementation of the Java Collections interfaces (Map, SortedMap, Set, SortedSet, and Iterator) is provided.
- Transactions are supported using the conventional Java transaction-per-thread model, where the current transaction is implicitly associated with the current thread.
- Transaction runner utilities are provided that automatically perform transaction retry and exception handling.
- Keys and values are represented as Java objects rather than byte arrays. Bindings are used to map between Java objects and the stored byte arrays.
- The tuple data format is provided as the simplest data representation, and is useful for keys as well as simple compact values.
- The serial data format is provided for storing arbitrary Java objects without writing custom binding code. Java serialization is extended to store the class descriptions separately, making the data records much more compact than with standard Java serialization.
- Custom data formats and bindings can be easily added. XML data format and XML bindings could easily be created using this feature, for example.

Note that the JE Collections API does not support caching of programming language objects nor does it keep track of their stored status. This is in contrast to "persistent object" approaches

such as those defined by ODMG [http://www.odmg.org/] and JDO (JSR 12). Such approaches have benefits but also require sophisticated object caching. For simplicity the JE Collections API treats data objects by value, not by reference, and does not perform object caching of any kind. Since the JE Collections API is a thin layer, its reliability and performance characteristics are roughly equivalent to those of Berkeley DB, and database tuning is accomplished in the same way as for any Berkeley DB database.

Developing a JE Collections Application

There are several important choices to make when developing an application using the JE Collections API.

1. Choose the Format for Keys and Values

   For each database you may choose a binding format for the keys and values. For example, the tuple format is useful for keys because it has a deterministic sort order. The serial format is useful for values if you want to store arbitrary Java objects. In some cases a custom format may be appropriate. For details on choosing a binding format see Using Data Bindings (page 77).

2. Choose the Binding for Keys and Values

   With the serial data format you do not have to create a binding for each Java class that is stored since Java serialization is used. But for other formats a binding must be defined that translates between stored byte arrays and Java objects. For details see Using Data Bindings (page 77).

3. Choose Secondary Indices and Foreign Key Indices

   Any database that has unique keys may have any number of secondary indices. A secondary index has keys that are derived from data values in the primary database. This allows lookup and iteration of objects in the database by its index keys. A foreign key index is a special type of secondary index where the index keys are also the primary keys of another primary database. For each index you must define how the index keys are derived from the data values using a `SecondaryKeyCreator`. For details see the `SecondaryDatabase`, `SecondaryConfig` and `SecondaryKeyCreator` classes.

4. Choose the Collection Interface for each Database

   The standard Java Collection interfaces are used for accessing databases and secondary indices. The Map and Set interfaces may be used for any type of database. The Iterator interface is used through the Set interfaces. For more information on the collection interfaces see Using Stored Collections (page 82).

Any number of bindings and collections may be created for the same database. This allows multiple views of the same stored data. For example, a data store may be viewed as a Map of keys to values, a Set of keys, or a Collection of values. String values, for example, may be used with the built-in binding to the String class, or with a custom binding to another class that represents the string values differently.

It is sometimes desirable to use a Java class that encapsulates both a data key and a data value. For example, a Part object might contain both the part number (key) and the part name (value). Using the JE Collections API this type of object is called an "entity". An entity binding is used to translate between the Java object and the stored data key and value. Entity bindings may be used with all Collection types.

Please be aware that the provided JE Collections API collection classes do not conform completely to the interface contracts defined in the `java.util` package. For example, all iterators must be explicitly closed and the `size()` method is not available. The differences between the JE Collections API collections and the standard Java collections are documented in Stored Collections Versus Standard Java Collections (page 83).

Tutorial Introduction

Most of the remainder of this document illustrates the use of the JE Collections API by presenting a tutorial that describes usage of the API. This tutorial builds a shipment database, a familiar example from classic database texts.

The examples illustrate the following concepts of the JE Collections API:

- Object-to-data *bindings*
- The database *environment*
- *Databases* that contain key/value records
- *Secondary index* databases that contain index keys
- Java *collections* for accessing databases and indices
- *Transactions* used to commit or undo database changes

The examples build on each other, but at the same time the source code for each example stands alone.

- The Basic Program (page 6)
- Using Secondary Indices and Foreign keys (page 32)
- Using Entity Classes (page 46)
- Using Tuples (page 58)
- Using Serializable Entities (page 68)

The shipment database consists of three database stores: the part store, the supplier store, and the shipment store. Each store contains a number of records, and each record consists of a key and a value.

Store	Key	Value
Part	Part Number	Name, Color, Weight, City
Supplier	Supplier Number	Name, Status, City
Shipment	Part Number, Supplier Number	Quantity

In the example programs, Java classes containing the fields above are defined for the key and value of each store: `PartKey`, `PartData`, `SupplierKey`, `SupplierData`, `ShipmentKey` and `ShipmentData`. In addition, because the Part's Weight field is itself composed of two fields — the weight value and the unit of measure — it is represented by a separate `Weight` class. These classes will be defined in the first example program.

In general the JE Collections API uses bindings to describe how Java objects are stored. A binding defines the stored data syntax and the mapping between a Java object and the stored data. The example programs show how to create different types of bindings, and explains the characteristics of each type.

The following tables show the record values that are used in all the example programs in the tutorial.

Number	Name	Color	Weight	City
P1	Nut	Red	12.0 grams	London
P2	Bolt	Green	17.0 grams	Paris
P3	Screw	Blue	17.0 grams	Rome
P4	Screw	Red	14.0 grams	London
P5	Cam	Blue	12.0 grams	Paris
P6	Cog	Red	19.0 grams	London

Number	Name	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Part Number	Supplier Number	Quantity
P1	S1	300
P1	S2	300
P2	S1	200
P2	S2	400
P2	S3	200
P2	S4	200
P3	S1	400
P4	S1	200
P4	S4	300
P5	S1	100
P5	S4	400
P6	S1	100

Chapter 2. The Basic Program

The Basic example is a minimal implementation of the shipment program. It writes and reads the part, supplier and shipment databases.

The complete source of the final version of the example program is included in the Berkeley DB distribution.

Defining Serialized Key and Value Classes

The key and value classes for each type of shipment record — Parts, Suppliers and Shipments — are defined as ordinary Java classes. In this example the serialized form of the key and value objects is stored directly in the database. Therefore these classes must implement the standard Java java.io.Serializable interface. A compact form of Java serialization is used that does not duplicate the class description in each record. Instead the class descriptions are stored in the class catalog store, which is described in the next section. But in all other respects, standard Java serialization is used.

An important point is that instances of these classes are passed and returned by value, not by reference, when they are stored and retrieved from the database. This means that changing a key or value object does not automatically change the database. The object must be explicitly stored in the database after changing it. To emphasize this point the key and value classes defined here have no field setter methods. Setter methods can be defined, but it is important to remember that calling a setter method will not cause the change to be stored in the database. How to store and retrieve objects in the database will be described later.

Each key and value class contains a toString method that is used to output the contents of the object in the example program. This is meant for illustration only and is not required for database objects in general.

Notice that the key and value classes defined below do not contain any references to `com.sleepycat` packages. An important characteristic of these classes is that they are independent of the database. Therefore, they may be easily used in other contexts and may be defined in a way that is compatible with other tools and libraries.

The `PartKey` class contains only the Part's Number field.

Note that `PartKey` (as well as `SupplierKey` below) contain only a single String field. Instead of defining a specific class for each type of key, the String class by itself could have been used. Specific key classes were used to illustrate strong typing and for consistency in the example. The use of a plain String as an index key is illustrated in the next example program. It is up to the developer to use either primitive Java classes such as String and Integer, or strongly typed classes. When there is the possibility that fields will be added later to a key or value, a specific class should be used.

```
import java.io.Serializable;

public class PartKey implements Serializable
{
    private String number;

    public PartKey(String number) {
        this.number = number;
    }

    public final String getNumber() {
        return number;
    }

    public String toString() {
        return "[PartKey: number=" + number + ']';
    }
}
```

The `PartData` class contains the Part's Name, Color, Weight and City fields.

```
import java.io.Serializable;

public class PartData implements Serializable
{
    private String name;
    private String color;
    private Weight weight;
    private String city;

    public PartData(String name, String color, Weight weight, String city)
    {
        this.name = name;
        this.color = color;
        this.weight = weight;
        this.city = city;
    }

    public final String getName()
    {
        return name;
    }

    public final String getColor()
    {
        return color;
    }

    public final Weight getWeight()
```

```
    {
        return weight;
    }

    public final String getCity()
    {
        return city;
    }

    public String toString()
    {
        return "[PartData: name=" + name +
               " color=" + color +
               " weight=" + weight +
               " city=" + city + ']';
    }
}
```

The `Weight` class is also defined here, and is used as the type of the Part's Weight field. Just as in standard Java serialization, nothing special is needed to store nested objects as long as they are all Serializable.

```
import java.io.Serializable;

public class Weight implements Serializable
{
    public final static String GRAMS = "grams";
    public final static String OUNCES = "ounces";

    private double amount;
    private String units;

    public Weight(double amount, String units)
    {
        this.amount = amount;
        this.units = units;
    }

    public final double getAmount()
    {
        return amount;
    }

    public final String getUnits()
    {
        return units;
    }

    public String toString()
```

```
    {
        return "[" + amount + ' ' + units + ']';
    }
}
```

The `SupplierKey` class contains the Supplier's Number field.

```
import java.io.Serializable;

public class SupplierKey implements Serializable
{
    private String number;

    public SupplierKey(String number)
    {
        this.number = number;
    }

    public final String getNumber()
    {
        return number;
    }

    public String toString()
    {
        return "[SupplierKey: number=" + number + ']';
    }
}
```

The `SupplierData` class contains the Supplier's Name, Status and City fields.

```
import java.io.Serializable;

public class SupplierData implements Serializable
{
    private String name;
    private int status;
    private String city;

    public SupplierData(String name, int status, String city)
    {
        this.name = name;
        this.status = status;
        this.city = city;
    }

    public final String getName()
    {
        return name;
    }
```

```
    public final int getStatus()
    {
        return status;
    }

    public final String getCity()
    {
        return city;
    }

    public String toString()
    {
        return "[SupplierData: name=" + name +
               " status=" + status +
               " city=" + city + ']';
    }
}
```

The `ShipmentKey` class contains the keys of both the Part and Supplier.

```
import java.io.Serializable;

public class ShipmentKey implements Serializable
{
    private String partNumber;
    private String supplierNumber;

    public ShipmentKey(String partNumber, String supplierNumber)
    {
        this.partNumber = partNumber;
        this.supplierNumber = supplierNumber;
    }

    public final String getPartNumber()
    {
        return partNumber;
    }

    public final String getSupplierNumber()
    {
        return supplierNumber;
    }

    public String toString()
    {
        return "[ShipmentKey: supplier=" + supplierNumber +
                " part=" + partNumber + ']';
```

```
    }
}
```

The `ShipmentData` class contains only the Shipment's Quantity field. Like `PartKey` and `SupplierKey`, `ShipmentData` contains only a single primitive field. Therefore the Integer class could have been used instead of defining a specific value class.

```
import java.io.Serializable;

public class ShipmentData implements Serializable
{
    private int quantity;

    public ShipmentData(int quantity)
    {
        this.quantity = quantity;
    }

    public final int getQuantity()
    {
        return quantity;
    }

    public String toString()
    {
        return "[ShipmentData: quantity=" + quantity + ']';
    }
}
```

Opening and Closing the Database Environment

This section of the tutorial describes how to open and close the database environment. The database environment manages resources (for example, memory, locks and transactions) for any number of databases. A single environment instance is normally used for all databases.

The `SampleDatabase` class is used to open and close the environment. It will also be used in following sections to open and close the class catalog and other databases. Its constructor is used to open the environment and its `close()` method is used to close the environment. The skeleton for the `SampleDatabase` class follows.

```
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import java.io.File;
import java.io.FileNotFoundException;

public class SampleDatabase
{
    private Environment env;
```

```
    public SampleDatabase(String homeDirectory)
        throws DatabaseException, FileNotFoundException
    {
    }

    public void close()
        throws DatabaseException
    {
    }
}
```

The first thing to notice is that the Environment class is in the com.sleepycat.je package, not the com.sleepycat.collections package. The com.sleepycat.je package contains all core Berkeley DB functionality. The com.sleepycat.collections package contains extended functionality that is based on the Java Collections API. The collections package is layered on top of the com.sleepycat.je package. Both packages are needed to create a complete application based on the JE Collections API.

The following statements create an `Environment` object.

```
public SampleDatabase(String homeDirectory)
        throws DatabaseException, FileNotFoundException
    {
        System.out.println("Opening environment in: " + homeDirectory);

        EnvironmentConfig envConfig = new EnvironmentConfig();
        envConfig.setTransactional(true);
        envConfig.setAllowCreate(true);

        env = new Environment(new File(homeDirectory), envConfig);
    }
```

The `EnvironmentConfig` class is used to specify environment configuration parameters. The first configuration option specified — `setTransactional()` — is set to true to create an environment where transactional (and non-transactional) databases may be opened. While non-transactional environments can also be created, the examples in this tutorial use a transactional environment.

`setAllowCreate()` is set to true to specify that the environment's files will be created if they don't already exist. If this parameter is not specified, an exception will be thrown if the environment does not already exist. A similar parameter will be used later to cause databases to be created if they don't exist.

When an `Environment` object is constructed, a home directory and the environment configuration object are specified. The home directory is the location of the environment's log files that store all database information.

The following statement closes the environment. The environment should always be closed when database work is completed to free allocated resources and to avoid having to run recovery

later. Closing the environment does not automatically close databases, so databases should be closed explicitly before closing the environment.

```
    public void close()
        throws DatabaseException
    {
        env.close();
    }
```

The following getter method returns the environment for use by other classes in the example program. The environment is used for opening databases and running transactions.

```
public class SampleDatabase
{
    ...
    public final Environment getEnvironment()
    {
        return env;
    }
    ...
}
```

Opening and Closing the Class Catalog

This section describes how to open and close the Java class catalog. The class catalog is a specialized database store that contains the Java class descriptions of the serialized objects that are stored in the database. The class descriptions are stored in the catalog rather than storing them redundantly in each database record. A single class catalog per environment must be opened whenever serialized objects will be stored in the database.

The `SampleDatabase` class is extended to open and close the class catalog. The following additional imports and class members are needed.

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

...

public class SampleDatabase
{
    private Environment env;
    private static final String CLASS_CATALOG = "java_class_catalog";
    ...
    private StoredClassCatalog javaCatalog;
```

```
    ...
}
```

While the class catalog is itself a database, it contains metadata for other databases and is therefore treated specially by the JE Collections API. The `StoredClassCatalog` class encapsulates the catalog store and implements this special behavior.

The following statements open the class catalog by creating a `Database` and a `StoredClassCatalog` object. The catalog database is created if it does not already exist.

```
    public SampleDatabase(String homeDirectory)
        throws DatabaseException, FileNotFoundException
    {
        ...
        DatabaseConfig dbConfig = new DatabaseConfig();
        dbConfig.setTransactional(true);
        dbConfig.setAllowCreate(true);

        Database catalogDb = env.openDatabase(null, CLASS_CATALOG, dbConfig);

        javaCatalog = new StoredClassCatalog(catalogDb);
        ...
    }
    ...
    public final StoredClassCatalog getClassCatalog() {
        return javaCatalog;
    }
```

The `DatabaseConfig` class is used to specify configuration parameters when opening a database. The first configuration option specified — `setTransactional()` — is set to true to create a transactional database. While non-transactional databases can also be created, the examples in this tutorial use transactional databases.

`setAllowCreate()` is set to true to specify that the database will be created if it does not already exist. If this parameter is not specified, an exception will be thrown if the database does not already exist.

The first parameter of the `openDatabase()` method is an optional transaction that is used for creating a new database. If null is passed, auto-commit is used when creating a database.

The second parameter of `openDatabase()` specifies the database name and must not be a null.

The last parameter of `openDatabase()` specifies the database configuration object.

Lastly, the `StoredClassCatalog` object is created to manage the information in the class catalog database. The `StoredClassCatalog` object will be used in the sections following for creating serial bindings.

The `getClassCatalog` method returns the catalog object for use by other classes in the example program.

When the environment is closed, the class catalog is closed also.

```
    public void close()
        throws DatabaseException
    {
        javaCatalog.close();
        env.close();
    }
```

The `StoredClassCatalog.close()` method simply closes the underlying class catalog database and in fact the `Database.close()` method may be called instead, if desired. The catalog database, and all other databases, must be closed before closing the environment.

Opening and Closing Databases

This section describes how to open and close the Part, Supplier and Shipment databases. A *database* is a collection of records, each of which has a key and a value. The keys and values are stored in a selected format, which defines the syntax of the stored data. Two examples of formats are Java serialization format and tuple format. In a given database, all keys have the same format and all values have the same format.

The `SampleDatabase` class is extended to open and close the three databases. The following additional class members are needed.

```
public class SampleDatabase
{
    ...
    private static final String SUPPLIER_STORE = "supplier_store";
    private static final String PART_STORE = "part_store";
    private static final String SHIPMENT_STORE = "shipment_store";
    ...
    private Database supplierDb;
    private Database partDb;
    private Database shipmentDb;
    ...
}
```

For each database there is a database name constant and a `Database` object.

The following statements open the three databases by constructing a Database object.

```
    public SampleDatabase(String homeDirectory)
        throws DatabaseException, FileNotFoundException
    {
        ...
        DatabaseConfig dbConfig = new DatabaseConfig();
        dbConfig.setTransactional(true);
        dbConfig.setAllowCreate(true);
        ...
        partDb = env.openDatabase(null, PART_STORE, dbConfig);
        supplierDb = env.openDatabase(null, SUPPLIER_STORE, dbConfig);
```

```
        shipmentDb = env.openDatabase(null, SHIPMENT_STORE, dbConfig);
        ...
    }
```

The database configuration object that was used previously for opening the catalog database is reused for opening the three databases above. The databases are created if they don't already exist. The parameters of the `openDatabase()` method were described earlier when the class catalog database was opened.

The following statements close the three databases.

```
    public void close()
        throws DatabaseException
    {
        partDb.close();
        supplierDb.close();
        shipmentDb.close();
        javaCatalog.close();
        env.close();
    }
```

All databases, including the catalog database, must be closed before closing the environment.

The following getter methods return the databases for use by other classes in the example program.

```
public class SampleDatabase
{
    ...
    public final Database getPartDatabase()
    {
        return partDb;
    }

    public final Database getSupplierDatabase()
    {
        return supplierDb;
    }

    public final Database getShipmentDatabase()
    {
        return shipmentDb;
    }
    ...
}
```

Creating Bindings and Collections

Bindings translate between stored records and Java objects. In this example, Java serialization bindings are used. Serial bindings are the simplest type of bindings because no mapping of

fields or type conversion is needed. Tuple bindings — which are more difficult to create than serial bindings but have some advantages — will be introduced later in the Tuple example program.

Standard Java *collections* are used to access records in a database. Stored collections use bindings transparently to convert the records to objects when they are retrieved from the collection, and to convert the objects to records when they are stored in the collection.

An important characteristic of stored collections is that they do *not* perform object caching. Every time an object is accessed via a collection it will be added to or retrieved from the database, and the bindings will be invoked to convert the data. Objects are therefore always passed and returned by value, not by reference. Because Berkeley DB is an embedded database, efficient caching of stored raw record data is performed by the database library.

The `SampleViews` class is used to create the bindings and collections. This class is separate from the `SampleDatabase` class to illustrate the idea that a single set of stored data can be accessed via multiple bindings and collections, or *views*. The skeleton for the `SampleViews` class follows.

```
import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.ClassCatalog;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.collections.StoredEntrySet;
import com.sleepycat.collections.StoredMap;
...

public class SampleViews
{
    private StoredMap partMap;
    private StoredMap supplierMap;
    private StoredMap shipmentMap;

    ...
    public SampleViews(SampleDatabase db)
    {
    }
}
```

A `StoredMap` field is used for each database. The StoredMap class implements the standard Java `Map` interface, which has methods for obtaining a `Set` of keys, a `Collection` of values, or a `Set` of `Map.Entry` key/value pairs. Because databases contain key/value pairs, any Berkeley DB database may be represented as a Java map.

The following statements create the key and data bindings using the `SerialBinding` class.

```
    public SampleViews(SampleDatabase db)
    {
        ClassCatalog catalog = db.getClassCatalog();
        EntryBinding partKeyBinding =
            new SerialBinding(catalog, PartKey.class);
        EntryBinding partValueBinding =
```

```
            new SerialBinding(catalog, PartData.class);
        EntryBinding supplierKeyBinding =
            new SerialBinding(catalog, SupplierKey.class);
        EntryBinding supplierValueBinding =
            new SerialBinding(catalog, SupplierData.class);
        EntryBinding shipmentKeyBinding =
            new SerialBinding(catalog, ShipmentKey.class);
        EntryBinding shipmentValueBinding =
            new SerialBinding(catalog, ShipmentData.class);
        ...
    }
```

The first parameter of the `SerialBinding` constructor is the class catalog, and is used to store the class descriptions of the serialized objects.

The second parameter is the base class for the serialized objects and is used for type checking of the stored objects. If `null` or `Object.class` is specified, then any Java class is allowed. Otherwise, all objects stored in that format must be instances of the specified class or derived from the specified class. In the example, specific classes are used to enable strong type checking.

The following statements create standard Java maps using the `StoredMap` class.

```
    public SampleViews(SampleDatabase db)
    {
        ...
        partMap =
            new StoredMap(db.getPartDatabase(),
                          partKeyBinding, partValueBinding, true);
        supplierMap =
            new StoredMap(db.getSupplierDatabase(),
                          supplierKeyBinding, supplierValueBinding, true);
        shipmentMap =
            new StoredMap(db.getShipmentDatabase(),
                          shipmentKeyBinding, shipmentValueBinding, true);
    ...
    }
```

The first parameter of the `StoredMap` constructor is the database. In a StoredMap, the database keys (the primary keys) are used as the map keys. The Index example shows how to use secondary index keys as map keys.

The second and third parameters are the key and value bindings to use when storing and retrieving objects via the map.

The fourth and last parameter specifies whether changes will be allowed via the collection. If false is passed, the collection will be read-only.

The following getter methods return the stored maps for use by other classes in the example program. Convenience methods for returning entry sets are also included.

```
public class SampleViews
{
    ...
    public final StoredMap getPartMap()
    {
        return partMap;
    }

    public final StoredMap getSupplierMap()
    {
        return supplierMap;
    }

    public final StoredMap getShipmentMap()
    {
        return shipmentMap;
    }

    public final StoredEntrySet getPartEntrySet()
    {
        return (StoredEntrySet) partMap.entrySet();
    }

    public final StoredEntrySet getSupplierEntrySet()
    {
        return (StoredEntrySet) supplierMap.entrySet();
    }

    public final StoredEntrySet getShipmentEntrySet()
    {
        return (StoredEntrySet) shipmentMap.entrySet();
    }
    ...
}
```

Note that StoredMap and StoredEntrySet are returned rather than just returning Map and Set. Since StoredMap implements the Map interface and StoredEntrySet implements the Set interface, you may ask why Map and Set were not returned directly.

`StoredMap`, `StoredEntrySet`, and other stored collection classes have a small number of extra methods beyond those in the Java collection interfaces. The stored collection types are therefore returned to avoid casting when using the extended methods. Normally, however, only a Map or Set is needed, and may be used as follows.

```
    SampleDatabase sd = new SampleDatabase(new String("/home"));
    SampleViews views = new SampleViews(sd);
    Map partMap = views.getPartMap();
    Set supplierEntries = views.getSupplierEntrySet();
```

Implementing the Main Program

The main program opens the environment and databases, stores and retrieves objects within a transaction, and finally closes the environment databases. This section describes the main program shell, and the next section describes how to run transactions for storing and retrieving objects.

The `Sample` class contains the main program. The skeleton for the `Sample` class follows.

```
import com.sleepycat.je.DatabaseException;
import java.io.FileNotFoundException;

public class Sample
{
    private SampleDatabase db;
    private SampleViews views;

    public static void main(String args)
    {
    }

    private Sample(String homeDir)
        throws DatabaseException, FileNotFoundException
    {
    }

    private void close()
        throws DatabaseException
    {
    }

    private void run()
        throws Exception
    {
    }
}
```

The main program uses the `SampleDatabase` and `SampleViews` classes that were described in the preceding sections. The `main` method will create an instance of the `Sample` class, and call its `run()` and `close()` methods.

The following statements parse the program's command line arguments.

```
    public static void main(String[] args)
    {
        System.out.println("\nRunning sample: " + Sample.class);
        String homeDir = "./tmp";
        for (int i = 0; i < args.length; i += 1)
        {
            String arg = args[i];
            if (args[i].equals("-h") && i < args.length - 1)
            {
                i += 1;
                homeDir = args[i];
            }
            else
            {
                System.err.println("Usage:\n java " +
                                   Sample.class.getName() +
                                  "\n  [-h <home-directory>]");
                System.exit(2);
            }
        }
        ...
    }
```

The usage command is:

```
java com.sleepycat.examples.bdb.shipment.basic.Sample
     [-h <home-directory> ]
```

The `-h` command is used to set the `homeDir` variable, which will later be passed to the `SampleDatabase()` constructor. Normally all Berkeley DB programs should provide a way to configure their database environment home directory.

The default for the home directory is `./tmp` — the tmp subdirectory of the current directory where the sample is run. The home directory must exist before running the sample. To re-create the sample database from scratch, delete all files in the home directory before running the sample.

The home directory was described previously in Opening and Closing the Database Environment (page 11).

Of course, the command line arguments shown are only examples and a real-life application may use different techniques for configuring these options.

The following statements create an instance of the `Sample` class and call its `run()` and `close()` methods.

```
    public static void main(String args)
    {
        ...
        Sample sample = null;
        try
        {
            sample = new Sample(homeDir);
            sample.run();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        finally
        {
            if (sample != null)
            {
                try
                {
                    sample.close();
                }
                catch (Exception e)
                {
                    System.err.println("Exception during database close:");
                    e.printStackTrace();
                }
            }
        }
    }
```

The `Sample()` constructor will open the environment and databases, and the `run()` method will run transactions for storing and retrieving objects. If either of these throws an exception, then the program was unable to run and should normally terminate. (Transaction retries are handled at a lower level and will be described later.) The first `catch` statement handles such exceptions.

The `finally` statement is used to call the `close()` method since an attempt should always be made to close the environment and databases cleanly. If an exception is thrown during close and a prior exception occurred above, then the exception during close is likely a side effect of the prior exception.

The `Sample()` constructor creates the `SampleDatabase` and `SampleViews` objects.

```
    private Sample(String homeDir)
        throws DatabaseException, FileNotFoundException
    {
        db = new SampleDatabase(homeDir);
```

```
        views = new SampleViews(db);
    }
```

Recall that creating the `SampleDatabase` object will open the environment and all databases.

To close the database the `Sample.close()` method simply calls `SampleDatabase.close()`.

```
     private void close()
        throws DatabaseException
    {
        db.close();
    }
```

The `run()` method is described in the next section.

Using Transactions

JE transactional applications have standard transactional characteristics: recoverability, atomicity and integrity (this is sometimes also referred to generically as *ACID properties*). The JE Collections API provides these transactional capabilities using a *transaction-per-thread* model. Once a transaction is begun, it is implicitly associated with the current thread until it is committed or aborted. This model is used for the following reasons.

- The transaction-per-thread model is commonly used in other Java APIs such as J2EE.
- Since the Java collections API is used for data access, there is no way to pass a transaction object to methods such as `Map.put`.

The JE Collections API provides two transaction APIs. The lower-level API is the `CurrentTransaction` class. It provides a way to get the transaction for the current thread, and to begin, commit and abort transactions. It also provides access to the Berkeley DB core API `Transaction` object. With `CurrentTransaction`, just as in the com.sleepycat.je API, the application is responsible for beginning, committing and aborting transactions, and for handling deadlock exceptions and retrying operations. This API may be needed for some applications, but it is not used in the example.

The example uses the higher-level `TransactionRunner` and `TransactionWorker` APIs, which are build on top of `CurrentTransaction`. `TransactionRunner.run()` automatically begins a transaction and then calls the `TransactionWorker.doWork()` method, which is implemented by the application.

The `TransactionRunner.run()` method automatically detects deadlock exceptions and performs retries by repeatedly calling the `TransactionWorker.doWork()` method until the operation succeeds or the maximum retry count is reached. If the maximum retry count is reached or if another exception (other than `DeadlockException`) is thrown by `TransactionWorker.doWork()`, then the transaction will be automatically aborted. Otherwise, the transaction will be automatically committed.

Using this high-level API, if `TransactionRunner.run()` throws an exception, the application can assume that the operation failed and the transaction was aborted; otherwise, when an exception

is not thrown, the application can assume the operation succeeded and the transaction was committed.

The `Sample.run()` method creates a `TransactionRunner` object and calls its `run()` method.

```
import com.sleepycat.collections.TransactionRunner;
import com.sleepycat.collections.TransactionWorker;
...
public class Sample
{
    private SampleDatabase db;
    ...
    private void run()
        throws Exception
    {
        TransactionRunner runner = new TransactionRunner(db.getEnvironment());
        runner.run(new PopulateDatabase());
        runner.run(new PrintDatabase());
    }
    ...
    private class PopulateDatabase implements TransactionWorker
    {
        public void doWork()
            throws Exception
        {
        }
    }

    private class PrintDatabase implements TransactionWorker
    {
        public void doWork()
            throws Exception
        {
        }
    }
}
```

The `run()` method is called by `main()` and was outlined in the previous section. It first creates a `TransactionRunner`, passing the database environment to its constructor.

It then calls `TransactionRunner.run()` to execute two transactions, passing instances of the application-defined `PopulateDatabase` and `PrintDatabase` nested classes. These classes implement the `TransactionWorker.doWork()` method and will be fully described in the next two sections.

For each call to `TransactionRunner.run()`, a separate transaction will be performed. The use of two transactions in the example — one for populating the database and another for printing its contents — is arbitrary. A real-life application should be designed to create transactions for each group of operations that should have ACID properties, while also taking into account the impact of transactions on performance.

The advantage of using `TransactionRunner` is that deadlock retries and transaction begin, commit and abort are handled automatically. However, a `TransactionWorker` class must be implemented for each type of transaction. If desired, anonymous inner classes can be used to implement the `TransactionWorker` interface.

Adding Database Items

Adding (as well as updating, removing, and deleting) information in the database is accomplished via the standard Java collections API. In the example, the `Map.put` method is used to add objects. All standard Java methods for modifying a collection may be used with the JE Collections API.

The `PopulateDatabase.doWork()` method calls private methods for adding objects to each of the three database stores. It is called via the `TransactionRunner` class and was outlined in the previous section.

```
import java.util.Map;
import com.sleepycat.collections.TransactionWorker;
...
public class Sample
{
    ...
    private SampleViews views;
    ...
    private class PopulateDatabase implements TransactionWorker
    {
        public void doWork()
            throws Exception
        {
            addSuppliers();
            addParts();
            addShipments();
        }
    }
    ...

    private void addSuppliers()
    {
    }

    private void addParts()
    {
    }

    private void addShipments()
    {
    }
}
```

The addSuppliers(), addParts() and addShipments() methods add objects to the Suppliers, Parts and Shipments stores. The Map for each store is obtained from the SampleViews object.

```
    private void addSuppliers()
    {
        Map suppliers = views.getSupplierMap();
        if (suppliers.isEmpty())
        {
            System.out.println("Adding Suppliers");
            suppliers.put(new SupplierKey("S1"),
                          new SupplierData("Smith", 20, "London"));
            suppliers.put(new SupplierKey("S2"),
                          new SupplierData("Jones", 10, "Paris"));
            suppliers.put(new SupplierKey("S3"),
                          new SupplierData("Blake", 30, "Paris"));
            suppliers.put(new SupplierKey("S4"),
                          new SupplierData("Clark", 20, "London"));
            suppliers.put(new SupplierKey("S5"),
                          new SupplierData("Adams", 30, "Athens"));
        }
    }

    private void addParts()
    {
        Map parts = views.getPartMap();
        if (parts.isEmpty())
        {
            System.out.println("Adding Parts");
            parts.put(new PartKey("P1"),
                      new PartData("Nut", "Red",
                                   new Weight(12.0, Weight.GRAMS),
                                   "London"));
            parts.put(new PartKey("P2"),
                      new PartData("Bolt", "Green",
                                   new Weight(17.0, Weight.GRAMS),
                                   "Paris"));
            parts.put(new PartKey("P3"),
                      new PartData("Screw", "Blue",
                                   new Weight(17.0, Weight.GRAMS),
                                   "Rome"));
            parts.put(new PartKey("P4"),
                      new PartData("Screw", "Red",
                                   new Weight(14.0, Weight.GRAMS),
                                   "London"));
            parts.put(new PartKey("P5"),
                      new PartData("Cam", "Blue",
                                   new Weight(12.0, Weight.GRAMS),
                                   "Paris"));
            parts.put(new PartKey("P6"),
```

```
                    new PartData("Cog", "Red",
                                 new Weight(19.0, Weight.GRAMS),
                                 "London"));
        }
    }

    private void addShipments()
    {
        Map shipments = views.getShipmentMap();
        if (shipments.isEmpty())
        {
            System.out.println("Adding Shipments");
            shipments.put(new ShipmentKey("P1", "S1"),
                          new ShipmentData(300));
            shipments.put(new ShipmentKey("P2", "S1"),
                          new ShipmentData(200));
            shipments.put(new ShipmentKey("P3", "S1"),
                          new ShipmentData(400));
            shipments.put(new ShipmentKey("P4", "S1"),
                          new ShipmentData(200));
            shipments.put(new ShipmentKey("P5", "S1"),
                          new ShipmentData(100));
            shipments.put(new ShipmentKey("P6", "S1"),
                          new ShipmentData(100));
            shipments.put(new ShipmentKey("P1", "S2"),
                          new ShipmentData(300));
            shipments.put(new ShipmentKey("P2", "S2"),
                          new ShipmentData(400));
            shipments.put(new ShipmentKey("P2", "S3"),
                          new ShipmentData(200));
            shipments.put(new ShipmentKey("P2", "S4"),
                          new ShipmentData(200));
            shipments.put(new ShipmentKey("P4", "S4"),
                          new ShipmentData(300));
            shipments.put(new ShipmentKey("P5", "S4"),
                          new ShipmentData(400));
        }
    }
}
```

The key and value classes used above were defined in the Defining Serialized Key and Value Classes (page 6).

In each method above, objects are added only if the map is not empty. This is a simple way of allowing the example program to be run repeatedly. In real-life applications another technique — checking the `Map.containsKey` method, for example — might be used.

Retrieving Database Items

Retrieving information from the database is accomplished via the standard Java collections API. In the example, the `Set.iterator` method is used to iterate all `Map.Entry` objects for each store. All standard Java methods for retrieving objects from a collection may be used with the JE Collections API.

The `PrintDatabase.doWork()` method calls `printEntries()` to print the map entries for each database store. It is called via the `TransactionRunner` class and was outlined in the previous section.

```
import java.util.Iterator;
...
public class Sample
{
    ...
    private SampleViews views;
    ...
    private class PrintDatabase implements TransactionWorker
    {
        public void doWork()
            throws Exception
        {
            printEntries("Parts",
                         views.getPartEntrySet().iterator());
            printEntries("Suppliers",
                         views.getSupplierEntrySet().iterator());
            printEntries("Shipments",
                         views.getShipmentEntrySet().iterator());
        }
    }
    ...

    private void printEntries(String label, Iterator iterator)
    {
    }
    ...
}
```

The `Set` of `Map.Entry` objects for each store is obtained from the `SampleViews` object. This set can also be obtained by calling the `Map.entrySet` method of a stored map.

The `printEntries()` prints the map entries for any stored map. The `Object.toString` method of each key and value is called to obtain a printable representation of each object.

```
    private void printEntries(String label, Iterator iterator)
    {
        System.out.println("\n--- " + label + " ---");
        while (iterator.hasNext())
        {
```

```
            Map.Entry entry = (Map.Entry) iterator.next();
            System.out.println(entry.getKey().toString());
            System.out.println(entry.getValue().toString());
        }
    }
```

This is one of a small number of behavioral differences between standard Java collections and stored collections. For a complete list see Using Stored Collections (page 82).

The output of the example program is shown below.

```
Adding Suppliers
Adding Parts
Adding Shipments

--- Parts ---
PartKey: number=P1
PartData: name=Nut color=Red weight=[12.0 grams] city=London
PartKey: number=P2
PartData: name=Bolt color=Green weight=[17.0 grams] city=Paris
PartKey: number=P3
PartData: name=Screw color=Blue weight=[17.0 grams] city=Rome
PartKey: number=P4
PartData: name=Screw color=Red weight=[14.0 grams] city=London
PartKey: number=P5
PartData: name=Cam color=Blue weight=[12.0 grams] city=Paris
PartKey: number=P6
PartData: name=Cog color=Red weight=[19.0 grams] city=London

--- Suppliers ---
SupplierKey: number=S1
SupplierData: name=Smith status=20 city=London
SupplierKey: number=S2
SupplierData: name=Jones status=10 city=Paris
SupplierKey: number=S3
SupplierData: name=Blake status=30 city=Paris
SupplierKey: number=S4
SupplierData: name=Clark status=20 city=London
SupplierKey: number=S5
SupplierData: name=Adams status=30 city=Athens

--- Shipments ---
ShipmentKey: supplier=S1 part=P1
ShipmentData: quantity=300
ShipmentKey: supplier=S2 part=P1
ShipmentData: quantity=300
ShipmentKey: supplier=S1 part=P2
ShipmentData: quantity=200
ShipmentKey: supplier=S2 part=P2
ShipmentData: quantity=400
```

```
ShipmentKey: supplier=S3 part=P2
ShipmentData: quantity=200
ShipmentKey: supplier=S4 part=P2
ShipmentData: quantity=200
ShipmentKey: supplier=S1 part=P3
ShipmentData: quantity=400
ShipmentKey: supplier=S1 part=P4
ShipmentData: quantity=200
ShipmentKey: supplier=S4 part=P4
ShipmentData: quantity=300
ShipmentKey: supplier=S1 part=P5
ShipmentData: quantity=100
ShipmentKey: supplier=S4 part=P5
ShipmentData: quantity=400
ShipmentKey: supplier=S1 part=P6
ShipmentData: quantity=100
```

Handling Exceptions

Exception handling was illustrated previously in Implementing the Main Program (page 20) and Using Transactions (page 23) exception handling in a JE Collections API application in more detail.

There are two exceptions that must be treated specially: `RunRecoveryException` and `DeadlockException`.

`RunRecoveryException` is thrown when the only solution is to shut down the application and run recovery. All applications must catch this exception and follow the recovery procedure.

When `DeadlockException` is thrown, the application should normally retry the operation. If a deadlock continues to occur for some maximum number of retries, the application should give up and try again later or take other corrective actions. The JE Collections API provides two APIs for transaction execution.

- When using the `CurrentTransaction` class directly, the application must catch `DeadlockException` and follow the procedure described previously.
- When using the `TransactionRunner` class, retries are performed automatically and the application need only handle the case where the maximum number of retries has been reached. In that case, `TransactionRunner.run` will throw `DeadlockException`.

When using the `TransactionRunner` class there are two other considerations.

- First, if the application-defined `TransactionWorker.doWork` method throws an exception the transaction will automatically be aborted, and otherwise the transaction will automatically be committed. Applications should design their transaction processing with this in mind.
- Second, please be aware that `TransactionRunner.run` unwraps exceptions in order to discover whether a nested exception is a `DeadlockException`. This is particularly important since all Berkeley DB exceptions that occur while calling a stored collection method are wrapped with

a `RuntimeExceptionWrapper`. This wrapping is necessary because Berkeley DB exceptions are checked exceptions, and the Java collections API does not allow such exceptions to be thrown.

When calling `TransactionRunner.run`, the unwrapped (nested) exception will be unwrapped and thrown automatically. If you are not using `TransactionRunner` or if you are handling exceptions directly for some other reason, use the `ExceptionUnwrapper.unwrap` method to get the nested exception. For example, this can be used to discover that an exception is a `RunRecoveryException` as shown below.

```
import com.sleepycat.je.RunRecoveryException;
import com.sleepycat.util.ExceptionUnwrapper;
...
    catch (Exception e)
    {
        e = ExceptionUnwrapper.unwrap(e);
        if (e instanceof RunRecoveryException)
        {
            // follow recovery procedure
        }
    }
```

Chapter 3. Using Secondary Indices and Foreign keys

In the Basic example, each store has a single *primary key*. The Index example extends the Basic example to add the use of *secondary keys* and *foreign keys*.

The complete source of the final version of the example program is included in the Berkeley DB distribution.

Opening Secondary Key Indices

Secondary indices or *secondary databases* are used to access a primary database by a key other than the primary key. Recall that the Supplier Number field is the primary key of the Supplier database. In this section, the Supplier City field will be used as a secondary lookup key. Given a city value, we would like to be able to find the Suppliers in that city. Note that more than one Supplier may be in the same city.

Both primary and secondary databases contain key-value records. The key of an index record is the secondary key, and its value is the key of the associated record in the primary database. When lookups by secondary key are performed, the associated record in the primary database is transparently retrieved by its primary key and returned to the caller.

Secondary indices are maintained automatically when index key fields (the City field in this case) are added, modified or removed in the records of the primary database. However, the application must implement a `SecondaryKeyCreator` that extracts the index key from the database record.

It is useful to contrast opening an secondary index with opening a primary database (as described earlier in Opening and Closing Databases (page 15).

- A primary database may be associated with one or more secondary indices. A secondary index is always associated with exactly one primary database.
- For a secondary index, a `SecondaryKeyCreator` must be implemented by the application to extract the index key from the record of its associated primary database.
- A primary database is represented by a `Database` object and a secondary index is represented by a `SecondaryDatabase` object. The `SecondaryDatabase` class extends the `Database` class.
- When a `SecondaryDatabase` is created it is associated with a primary `Database` object and a `SecondaryKeyCreator`.

The `SampleDatabase` class is extended to open the Supplier-by-City secondary key index.

```
import com.sleepycat.bind.serial.SerialSerialKeyCreator;
import com.sleepycat.je.SecondaryConfig;
import com.sleepycat.je.SecondaryDatabase;
...
public class SampleDatabase
```

```
{
    ...
    private static final String SUPPLIER_CITY_INDEX = "supplier_city_index";
    ...
    private SecondaryDatabase supplierByCityDb;
    ...
    public SampleDatabase(String homeDirectory)
        throws DatabaseException, FileNotFoundException
    {
        ...
        SecondaryConfig secConfig = new SecondaryConfig();
        secConfig.setTransactional(true);
        secConfig.setAllowCreate(true);
        secConfig.setSortedDuplicates(true);

        secConfig.setKeyCreator(
            new SupplierByCityKeyCreator(javaCatalog,
                                         SupplierKey.class,
                                         SupplierData.class,
                                         String.class));

        supplierByCityDb = env.openSecondaryDatabase(null,
                                                     SUPPLIER_CITY_INDEX,
                                                     supplierDb,
                                                     secConfig);
    ...
    }
}
```

A `SecondaryConfig` object is used to configure the secondary database. The `SecondaryConfig` class extends the `DatabaseConfig` class, and most steps for configuring a secondary database are the same as for configuring a primary database. The main difference in the example above is that the `SecondaryConfig.setSortedDuplicates()` method is called to allow duplicate index keys. This is how more than one Supplier may be in the same City. If this property is not specified, the default is that the index keys of all records must be unique.

For a primary database, duplicate keys are not normally used since a primary database with duplicate keys may not have any associated secondary indices. If primary database keys are not unique, there is no way for a secondary key to reference a specific record in the primary database.

Opening a secondary key index requires creating a `SecondaryKeyCreator`. The `SupplierByCityKeyCreator` class implements the `SecondaryKeyCreator` interface and will be defined below.

The `SecondaryDatabase` object is opened last. If you compare the `openSecondaryDatabase()` and `openDatabase()` methods you'll notice only two differences:

- openSecondaryDatabase() has an extra parameter for specifying the associated primary database. The primary database is supplierDb in this case.
- The last parameter of openSecondaryDatabase() is a SecondaryConfig instead of a DatabaseConfig.

How to use the secondary index to access records will be shown in a later section.

The application-defined SupplierByCityKeyCreator class is shown below. It was used above to configure the secondary database.

```
public class SampleDatabase
{
...
    private static class SupplierByCityKeyCreator
        extends SerialSerialKeyCreator
    {
        private SupplierByCityKeyCreator(StoredClassCatalog catalog,
                                         Class primaryKeyClass,
                                         Class valueClass,
                                         Class indexKeyClass)
        {
            super(catalog, primaryKeyClass, valueClass, indexKeyClass);
        }

        public Object createSecondaryKey(Object primaryKeyInput,
                                         Object valueInput)
        {
            SupplierData supplierData = (SupplierData) valueInput;
            return supplierData.getCity();
        }
    }
...
}
```

In general, a key creator class must implement the SecondaryKeyCreator interface. This interface has methods that operate on the record data as raw bytes. In practice, it is easiest to use an abstract base class that performs the conversion of record data to and from the format defined for the database's key and value. The base class implements the SecondaryKeyCreator interface and has abstract methods that must be implemented in turn by the application.

In this example the SerialSerialKeyCreator base class is used because the database record uses the serial format for both its key and its value. The abstract methods of this class have key and value parameters of type Object which are automatically converted to and from the raw record data by the base class.

To perform the conversions properly, the key creator must be aware of all three formats involved: the key format of the primary database record, the value format of the primary database record, and the key format of the index record. The SerialSerialKeyCreator constructor is given the base classes for these three formats as parameters.

The `SerialSerialKeyCreator.createSecondaryKey` method is given the key and value of the primary database record as parameters, and it returns the key of the index record. In this example, the index key is a field in the primary database record value. Since the record value is known to be a `SupplierData` object, it is cast to that class and the city field is returned.

Note that the `primaryKeyInput` parameter is not used in the example. This parameter is needed only when an index key is derived from the key of the primary database record. Normally an index key is derived only from the primary database record value, but it may be derived from the key, value or both.

The following getter methods return the secondary database object for use by other classes in the example program. The secondary database object is used to create Java collections for accessing records via their secondary keys.

```
public class SampleDatabase
{
    ...
    public final SecondaryDatabase getSupplierByCityDatabase()
    {
        return supplierByCityDb;
    }
    ...
}
```

The following statement closes the secondary database.

```
public class SampleDatabase
{
    ...
    public void close()
        throws DatabaseException {

        supplierByCityDb.close();
        partDb.close();
        supplierDb.close();
        shipmentDb.close();
        javaCatalog.close();
        env.close();
    }
    ...
}
```

Secondary databases must be closed before closing their associated primary database.

Opening Foreign Key Indices

This section builds on the prior section describing secondary key indices to show how to open foreign key indices. A *foreign key index* is a secondary key index that also provides integrity constraints. When the primary key of a record in one database is embedded in the value of a record in another database, integrity constraints ensure that the record in the first database

exists, i.e, that there are no "dangling pointers". In this example the Shipment's PartNumber and SupplierNumber fields will be used as foreign keys.

When a foreign key index is defined, an "delete action" parameter is specified. This parameter determines what action is taken by the Berkeley DB Java Edition API when the record is deleted to which a foreign key refers. For example, consider what happens to a Shipment record when a Part or Supplier record that is referred to by that Shipment is deleted. There are three possibilities.

- `ForeignKeyDeleteAction.ABORT` specifies that the transaction should be aborted by throwing an exception. The effect is that deleting a Part or Supplier that is referred to by one or more Shipments will not be possible. The Berkeley DB Java Edition will automatically throw a `DatabaseException`, which should normally cause the transaction to be aborted during exception processing. This is the default delete action if none is specified.

- `ForeignKeyDeleteAction.NULLIFY` specifies that the Part or Supplier Number field in the Shipment record should be cleared, or set to a null or empty value. The effect is that the deleted Part or Supplier will no longer be referenced by any Shipment record. This option applies when the foreign key field is optional, i.e, when the application allows it to be set to a null or empty value. When using this option, the application must implement the `nullifyForeignKey()` method of the `ForeignKeyNullifier` interface.

- `ForeignKeyDeleteAction.CASCADE` specifies that the Shipment record should be deleted also. The effect is that deleting a Part or Supplier will delete all Shipments for that Part or Supplier. This option applies when the deleted record is considered the "parent" or "owner" of the record containing the foreign key, and is used in this example. Since deleting the Shipment record could cause other deletions if other records contain the foreign key of the Shipment, and so on, the term "cascade" is used to describe the effect.

The `SampleDatabase` class is extended to open the Shipment-by-Part and Shipment-by-Supplier secondary key indices.

```
import com.sleepycat.bind.serial.SerialSerialKeyCreator;
import com.sleepycat.je.ForeignKeyDeleteAction;
import com.sleepycat.je.SecondaryConfig;
import com.sleepycat.je.SecondaryDatabase;
...
public class SampleDatabase
{
    ...
    private static final String SHIPMENT_PART_INDEX = "shipment_part_index";
    private static final String SHIPMENT_SUPPLIER_INDEX =
        "shipment_supplier_index";
    ...
    private SecondaryDatabase shipmentByPartDb;
    private SecondaryDatabase shipmentBySupplierDb;
    ...
    public SampleDatabase(String homeDirectory)
        throws DatabaseException, FileNotFoundException
    {
```

```
        ...
        SecondaryConfig secConfig = new SecondaryConfig();
        secConfig.setTransactional(true);
        secConfig.setAllowCreate(true);
        secConfig.setSortedDuplicates(true);
        ...
        secConfig.setForeignKeyDatabase(partDb);
        secConfig.setForeignKeyDeleteAction(ForeignKeyDeleteAction.CASCADE);
        secConfig.setKeyCreator(
            new ShipmentByPartKeyCreator(javaCatalog,
                                         ShipmentKey.class,
                                         ShipmentData.class,
                                         PartKey.class));
        shipmentByPartDb = env.openSecondaryDatabase(null,
                                                     SHIPMENT_PART_INDEX,
                                                     shipmentDb,
                                                     secConfig);

        secConfig.setForeignKeyDatabase(supplierDb);
        secConfig.setForeignKeyDeleteAction(ForeignKeyDeleteAction.CASCADE);
        secConfig.setKeyCreator(
            new ShipmentBySupplierKeyCreator(javaCatalog,
                                             ShipmentKey.class,
                                             ShipmentData.class,
                                             SupplierKey.class));
        shipmentBySupplierDb = env.openSecondaryDatabase(null,
                                                     SHIPMENT_SUPPLIER_INDEX,
                                                     shipmentDb,
                                                     secConfig);
    ...
    }
}
```

If you compare these statements for opening foreign key indices to the statements used in the previous section for opening a secondary index, you'll notice that the only significant difference is that the `setForeignKeyDatabase()` and `setForeignKeyDeleteAction()` methods are called. `setForeignKeyDatabase()` specifies the foreign database that contains the records to which the foreign keys refer; this configures the secondary database as a foreign key index. `setForeignKeyDeleteAction()` specifies the delete action.

The application-defined `ShipmentByPartKeyCreator` and `ShipmentBySupplierKeyCreator` classes are shown below. They were used above to configure the secondary database objects.

```
public class SampleDatabase
{
...
    private static class ShipmentByPartKeyCreator
        extends SerialSerialKeyCreator
    {
```

```
        private ShipmentByPartKeyCreator(StoredClassCatalog catalog,
                                         Class primaryKeyClass,
                                         Class valueClass,
                                         Class indexKeyClass)
        {
            super(catalog, primaryKeyClass, valueClass, indexKeyClass);
        }

        public Object createSecondaryKey(Object primaryKeyInput,
                                         Object valueInput)
        {
            ShipmentKey shipmentKey = (ShipmentKey) primaryKeyInput;
            return new PartKey(shipmentKey.getPartNumber());
        }
    }

    private static class ShipmentBySupplierKeyCreator
        extends SerialSerialKeyCreator
    {
        private ShipmentBySupplierKeyCreator(StoredClassCatalog catalog,
                                             Class primaryKeyClass,
                                             Class valueClass,
                                             Class indexKeyClass)
        {
            super(catalog, primaryKeyClass, valueClass, indexKeyClass);
        }

        public Object createSecondaryKey(Object primaryKeyInput,
                                         Object valueInput)
        {
            ShipmentKey shipmentKey = (ShipmentKey) primaryKeyInput;
            return new SupplierKey(shipmentKey.getSupplierNumber());
        }
    }
    ...
}
```

The key creator classes above are almost identical to the one defined in the previous section for use with a secondary index. The index key fields are different, of course, but the interesting difference is that the index keys are extracted from the key, not the value, of the Shipment record. This illustrates that an index key may be derived from the primary database record key, value, or both.

Note that the `SerialSerialKeyCreator.nullifyForeignKey` method is not overridden above. This is because `ForeignKeyDeleteAction.NULLIFY` was not used when creating the `SecondaryDatabase` objects. If it were used, implementing the `nullifyForeignKey()` methods would be needed to set the part number and supplier number to null in the Shipment key. But record keys cannot be changed! And in fact, the primary key is not passed to the `SerialSerialKeyCreator.nullifyForeignKey()` method, only the primary value is passed.

Therefore, if a foreign index key is derived from the primary key, `ForeignKeyDeleteAction.NULLIFY` may not be used.

The following getter methods return the secondary database objects for use by other classes in the example program.

```
public class SampleDatabase
{
    ...
    public final SecondaryDatabase getShipmentByPartDatabase()
    {
        return shipmentByPartDb;
    }

    public final SecondaryDatabase getShipmentBySupplierDatabase()
    {
        return shipmentBySupplierDb;
    }
    ...
}
```

The following statements close the secondary databases.

```
public class SampleDatabase
{
    ...
    public void close()
        throws DatabaseException {

        supplierByCityDb.close();
        shipmentByPartDb.close();
        shipmentBySupplierDb.close();
        partDb.close();
        supplierDb.close();
        shipmentDb.close();
        javaCatalog.close();
        env.close();
    }
    ...
}
```

Secondary databases must be closed before closing their associated primary database.

Creating Indexed Collections

In the prior Basic example, bindings and Java collections were created for accessing databases via their primary keys. In this example, bindings and collections are added for accessing the same databases via their index keys. As in the prior example, serial bindings and the Java `Map` class are used.

When a map is created from a SecondaryDatabase, the keys of the map will be the index keys. However, the values of the map will be the values of the primary database associated with the index. This is how index keys can be used to access the values in a primary database.

For example, the Supplier's City field is an index key that can be used to access the Supplier database. When a map is created using the supplierByCityDb() method, the key to the map will be the City field, a String object. When Map.get is called passing the City as the key parameter, a SupplierData object will be returned.

The SampleViews class is extended to create an index key binding for the Supplier's City field and three Java maps based on the three indices created in the prior section.

```
import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.collections.StoredEntrySet;
import com.sleepycat.collections.StoredMap;
...

public class SampleViews
{
    ...
    private StoredMap supplierByCityMap;
    private StoredMap shipmentByPartMap;
    private StoredMap shipmentBySupplierMap;
    ...

    public SampleViews(SampleDatabase db)
    {
        ClassCatalog catalog = db.getClassCatalog();
        ...
        EntryBinding cityKeyBinding =
            new SerialBinding(catalog, String.class);
        ...
        supplierByCityMap =
            new StoredMap(db.getSupplierByCityDatabase(),
                          cityKeyBinding, supplierValueBinding, true);
        shipmentByPartMap =
            new StoredMap(db.getShipmentByPartDatabase(),
                          partKeyBinding, shipmentValueBinding, true);
        shipmentBySupplierMap =
            new StoredMap(db.getShipmentBySupplierDatabase(),
                          supplierKeyBinding, shipmentValueBinding, true);
    ...
    }
}
```

In general, the indexed maps are created here in the same way as the unindexed maps were created in the Basic example. The differences are:

- The first parameter of the `StoredMap` constructor is a `SecondaryDatabase` rather than a `Database`.
- The second parameter is the index key binding rather than the primary key binding.

For the `supplierByCityMap`, the `cityKeyBinding` must first be created. This binding was not created in the Basic example because the City field is not a primary key.

Like the bindings created earlier for keys and values, the `cityKeyBinding` is a `SerialBinding`. Unlike the bindings created earlier, it is an example of creating a binding for a built-in Java class, `String`, instead of an application-defined class. Any serializable class may be used.

For the `shipmentByPartMap` and `shipmentBySupplierMap`, the `partKeyBinding` and `supplierKeyBinding` are used. These were created in the Basic example and used as the primary key bindings for the `partMap` and `supplierMap`.

The value bindings — `supplierValueBinding` and `shipmentValueBinding` — were also created in the Basic example.

This illustrates that bindings and formats may and should be reused where appropriate for creating maps and other collections.

The following getter methods return the stored maps for use by other classes in the example program. Convenience methods for returning entry sets are also included.

```
public class SampleViews
{
    ...
    public final StoredMap getShipmentByPartMap()
    {
        return shipmentByPartMap;
    }

    public final StoredMap getShipmentBySupplierMap()
    {
        return shipmentBySupplierMap;
    }

    public final StoredMap getSupplierByCityMap()
    {
        return supplierByCityMap;
    }

    public final StoredEntrySet getShipmentByPartEntrySet()
    {
        return (StoredEntrySet) shipmentByPartMap.entrySet();
    }

    public final StoredEntrySet getShipmentBySupplierEntrySet()
    {
```

```
        return (StoredEntrySet) shipmentBySupplierMap.entrySet();
    }

    public final StoredEntrySet getSupplierByCityEntrySet()
    {
        return (StoredEntrySet) supplierByCityMap.entrySet();
    }
    ...
}
```

Retrieving Items by Index Key

Retrieving information via database index keys can be accomplished using the standard Java collections API, using a collection created from a `SecondaryDatabase` rather than a `Database`. However, the standard Java API does not support *duplicate keys*: more than one element in a collection having the same key. All three indices created in the prior section have duplicate keys because of the nature of the city, part number and supplier number index keys. More than one supplier may be in the same city, and more than one shipment may have the same part number or supplier number. This section describes how to use extended methods for stored collections to return all values for a given key.

Using the standard Java collections API, the `Map.get` method for a stored collection with duplicate keys will return only the first value for a given key. To obtain all values for a given key, the `StoredMap.duplicates` method may be called. This returns a `Collection` of values for the given key. If duplicate keys are not allowed, the returned collection will have at most one value. If the key is not present in the map, an empty collection is returned.

The `Sample` class is extended to retrieve duplicates for specific index keys that are present in the database.

```
import java.util.Iterator;
...
public class Sample
{
    ...
    private SampleViews views;
    ...
    private class PrintDatabase implements TransactionWorker
    {
        public void doWork()
            throws Exception
        {
            printEntries("Parts",
                         views.getPartEntrySet().iterator());
            printEntries("Suppliers",
                         views.getSupplierEntrySet().iterator());
            printValues("Suppliers for City Paris",
                        views.getSupplierByCityMap().duplicates(
                                            "Paris").iterator());
```

```
            printEntries("Shipments",
                         views.getShipmentEntrySet().iterator());
            printValues("Shipments for Part P1",
                        views.getShipmentByPartMap().duplicates(
                                            new PartKey("P1")).iterator());
            printValues("Shipments for Supplier S1",
                        views.getShipmentBySupplierMap().duplicates(
                                            new
                                            SupplierKey("S1")).iterator());
        }
    }

    private void printValues(String label, Iterator iterator)
    {
        System.out.println("\n--- " + label + " ---");
        while (iterator.hasNext())
        {
                System.out.println(iterator.next().toString());
        }
     }
    ...
}
```

The `StoredMap.duplicates` method is called passing the desired key. The returned value is a standard Java `Collection` containing the values for the specified key. A standard Java `Iterator` is then obtained for this collection and all values returned by that iterator are printed.

Another technique for retrieving duplicates is to use the collection returned by `Map.entrySet`. When duplicate keys are present, a `Map.Entry` object will be present in this collection for each duplicate. This collection can then be iterated or a subset can be created from it, all using the standard Java collection API.

Note that we did not discuss how duplicates keys can be explicitly added or removed in a collection. For index keys, the addition and deletion of duplicate keys happens automatically when records containing the index key are added, updated, or removed.

While not shown in the example program, it is also possible to create a store with duplicate keys in the same way as an index with duplicate keys — by calling `DatabaseConfig.setSortedDuplicates()` method. In that case, calling `Map.put` will add duplicate keys. To remove all duplicate keys, call `Map.remove`. To remove a specific duplicate key, call `StoredMap.duplicates` and then call `Collection.remove` using the returned collection. Duplicate values may also be added to this collection using `Collection.add`.

The output of the example program is shown below.

```
Adding Suppliers
Adding Parts
Adding Shipments

--- Parts ---
```

```
PartKey: number=P1
PartData: name=Nut color=Red weight=[12.0 grams] city=London
PartKey: number=P2
PartData: name=Bolt color=Green weight=[17.0 grams] city=Paris
PartKey: number=P3
PartData: name=Screw color=Blue weight=[17.0 grams] city=Rome
PartKey: number=P4
PartData: name=Screw color=Red weight=[14.0 grams] city=London
PartKey: number=P5
PartData: name=Cam color=Blue weight=[12.0 grams] city=Paris
PartKey: number=P6
PartData: name=Cog color=Red weight=[19.0 grams] city=London

--- Suppliers ---
SupplierKey: number=S1
SupplierData: name=Smith status=20 city=London
SupplierKey: number=S2
SupplierData: name=Jones status=10 city=Paris
SupplierKey: number=S3
SupplierData: name=Blake status=30 city=Paris
SupplierKey: number=S4
SupplierData: name=Clark status=20 city=London
SupplierKey: number=S5
SupplierData: name=Adams status=30 city=Athens

--- Suppliers for City Paris ---
SupplierData: name=Jones status=10 city=Paris
SupplierData: name=Blake status=30 city=Paris

--- Shipments ---
ShipmentKey: supplier=S1 part=P1
ShipmentData: quantity=300
ShipmentKey: supplier=S2 part=P1
ShipmentData: quantity=300
ShipmentKey: supplier=S1 part=P2
ShipmentData: quantity=200
ShipmentKey: supplier=S2 part=P2
ShipmentData: quantity=400
ShipmentKey: supplier=S3 part=P2
ShipmentData: quantity=200
ShipmentKey: supplier=S4 part=P2
ShipmentData: quantity=200
ShipmentKey: supplier=S1 part=P3
ShipmentData: quantity=400
ShipmentKey: supplier=S1 part=P4
ShipmentData: quantity=200
ShipmentKey: supplier=S4 part=P4
ShipmentData: quantity=300
ShipmentKey: supplier=S1 part=P5
```

```
ShipmentData: quantity=100
ShipmentKey: supplier=S4 part=P5
ShipmentData: quantity=400
ShipmentKey: supplier=S1 part=P6
ShipmentData: quantity=100

--- Shipments for Part P1 ---
ShipmentData: quantity=300
ShipmentData: quantity=300

--- Shipments for Supplier S1 ---
ShipmentData: quantity=300
ShipmentData: quantity=200
ShipmentData: quantity=400
ShipmentData: quantity=200
ShipmentData: quantity=100
ShipmentData: quantity=100
```

Chapter 4. Using Entity Classes

In the prior examples, the keys and values of each store were represented using separate classes. For example, a `PartKey` and a `PartData` class were used. Many times it is desirable to have a single class representing both the key and the value, for example, a `Part` class.

Such a combined key and value class is called an *entity class* and is used along with an *entity binding*. Entity bindings combine a key and a value into an entity when reading a record from a collection, and split an entity into a key and a value when writing a record to a collection. Entity bindings are used in place of value bindings, and entity objects are used with collections in place of value objects.

Some reasons for using entities are:

- When the key is a property of an entity object representing the record as a whole, the object's identity and concept are often clearer than with key and value objects that are disjoint.
- A single entity object per record is often more convenient to use than two objects.

Of course, instead of using an entity binding, you could simply create the entity yourself after reading the key and value from a collection, and split the entity into a key and value yourself before writing it to a collection. But this would detract from the convenience of the using the Java collections API. It is convenient to obtain a `Part` object directly from `Map.get` and to add a `Part` object using `Set.add`. Collections having entity bindings can be used naturally without combining and splitting objects each time a collection method is called; however, an entity binding class must be defined by the application.

In addition to showing how to use entity bindings, this example illustrates a key feature of all bindings: Bindings are independent of database storage parameters and formats. Compare this example to the prior Index example and you'll see that the `Sample` and `SampleViews` classes have been changed to use entity bindings, but the `SampleDatabase` class was not changed at all. In fact, the Entity program and the Index program can be used interchangeably to access the same physical database files. This demonstrates that bindings are only a "view" onto the physical stored data.

`Warning:` When using multiple bindings for the same database, it is the application's responsibility to ensure that the same format is used for all bindings. For example, a serial binding and a tuple binding cannot be used to access the same records.

The complete source of the final version of the example program is included in the Berkeley DB distribution.

Defining Entity Classes

As described in the prior section, *entity classes* are combined key/value classes that are managed by entity bindings. In this example the `Part`, `Supplier` and `Shipment` classes are entity classes. These classes contain fields that are a union of the fields of the key and value classes that were defined earlier for each store.

In general, entity classes may be defined in any way desired by the application. The entity binding, which is also defined by the application, is responsible for mapping between key/value objects and entity objects.

The `Part`, `Supplier` and `Shipment` entity classes are defined below.

An important difference between the entity classes defined here and the key and value classes defined earlier is that the entity classes are not serializable (do not implement the `Serializable` interface). This is because the entity classes are not directly stored. The entity binding decomposes an entity object into key and value objects, and only the key and value objects are serialized for storage.

One advantage of using entities can already be seen in the `toString()` method of the classes below. These return debugging output for the combined key and value, and will be used later to create a listing of the database that is more readable than in the prior examples.

```
public class Part
{
    private String number;
    private String name;
    private String color;
    private Weight weight;
    private String city;

    public Part(String number, String name, String color, Weight weight,
                String city)
    {
        this.number = number;
        this.name = name;
        this.color = color;
        this.weight = weight;
        this.city = city;
    }

    public final String getNumber()
    {
        return number;
    }

    public final String getName()
    {
        return name;
    }

    public final String getColor()
    {
        return color;
    }

    public final Weight getWeight()
```

```
    {
        return weight;
    }

    public final String getCity()
    {
        return city;
    }

    public String toString()
    {
        return "Part: number=" + number +
               " name=" + name +
               " color=" + color +
               " weight=" + weight +
               " city=" + city + '.';
    }
}
```

```
public class Supplier
{
    private String number;
    private String name;
    private int status;
    private String city;

    public Supplier(String number, String name, int status, String city)
    {
        this.number = number;
        this.name = name;
        this.status = status;
        this.city = city;
    }

    public final String getNumber()
    {
        return number;
    }

    public final String getName()
    {
        return name;
    }

    public final int getStatus()
    {
        return status;
    }
```

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

    public String toString()
    {
        return "Supplier: number=" + number +
               " name=" + name +
               " status=" + status +
               " city=" + city + '.';
    }
}
```

```
public class Shipment
{
    private String partNumber;
    private String supplierNumber;
    private int quantity;

    public Shipment(String partNumber, String supplierNumber, int quantity)
    {
        this.partNumber = partNumber;
        this.supplierNumber = supplierNumber;
        this.quantity = quantity;
    }

    public final String getPartNumber()
    {
        return partNumber;
    }

    public final String getSupplierNumber()
    {
        return supplierNumber;
    }

    public final int getQuantity()
    {
        return quantity;
    }

    public String toString()
    {
        return "Shipment: part=" + partNumber +
               " supplier=" + supplierNumber +
               " quantity=" + quantity + '.';
    }
}
```

Creating Entity Bindings

Entity bindings are similar to ordinary bindings in that they convert between Java objects and the stored data format of keys and values. In addition, entity bindings map between key/value pairs and entity objects. An ordinary binding is a one-to-one mapping, while an entity binding is a two-to-one mapping.

The `partValueBinding`, `supplierValueBinding` and `shipmentValueBinding` bindings are created below as entity bindings rather than (in the prior examples) serial bindings.

```
import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.EntityBinding;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.bind.serial.SerialSerialBinding;
...

public class SampleViews
{
    ...
    public SampleViews(SampleDatabase db)
    {
        ClassCatalog catalog = db.getClassCatalog();
        SerialBinding partKeyBinding =
            new SerialBinding(catalog, PartKey.class);
        EntityBinding partValueBinding =
            new PartBinding(catalog, PartKey.class, PartData.class);
        SerialBinding supplierKeyBinding =
            new SerialBinding(catalog, SupplierKey.class);
        EntityBinding supplierValueBinding =
            new SupplierBinding(catalog, SupplierKey.class,
                                SupplierData.class);
        SerialBinding shipmentKeyBinding =
            new SerialBinding(catalog, ShipmentKey.class);
        EntityBinding shipmentValueBinding =
            new ShipmentBinding(catalog, ShipmentKey.class,
                                ShipmentData.class);
        SerialBinding cityKeyBinding =
            new SerialBinding(catalog, String.class);
        ...
    }
}
```

The entity bindings will be used in the next section to construct stored map objects.

The `PartBinding` class is defined below.

```
public class SampleViews
{
    ...
    private static class PartBinding extends SerialSerialBinding {
```

```
        private PartBinding(ClassCatalog classCatalog,
                            Class keyClass,
                            Class dataClass)
        {
            super(classCatalog, keyClass, dataClass);
        }

        public Object entryToObject(Object keyInput, Object dataInput)
        {
            PartKey key = (PartKey) keyInput;
            PartData data = (PartData) dataInput;
            return new Part(key.getNumber(), data.getName(), data.getColor(),
                            data.getWeight(), data.getCity());
        }

        public Object objectToKey(Object object)
        {
            Part part = (Part) object;
            return new PartKey(part.getNumber());
        }

        public Object objectToData(Object object)
        {
            Part part = (Part) object;
            return new PartData(part.getName(), part.getColor(),
                                part.getWeight(), part.getCity());
        }
    }
    ...
}
```

In general, an entity binding is any class that implements the `EntityBinding` interface, just as an ordinary binding is any class that implements the `EntryBinding` interface. In the prior examples the built-in `SerialBinding` class (which implements `EntryBinding`) was used and no application-defined binding classes were needed.

In this example, application-defined binding classes are used that extend the `SerialSerialBinding` abstract base class. This base class implements `EntityBinding` and provides the conversions between key/value bytes and key/value objects, just as the `SerialBinding` class does. The application-defined entity class implements the abstract methods defined in the base class that map between key/value objects and entity objects.

Three abstract methods are implemented for each entity binding. The `entryToObject()` method takes as input the key and data objects, which have been deserialized automatically by the base class. As output, it returns the combined `Part` entity.

The `objectToKey()` and `objectToData()` methods take an entity object as input. As output they return the part key or data object that is extracted from the entity object. The key or data will then be serialized automatically by the base class.

The SupplierBinding and ShipmentBinding classes are very similar to the PartBinding class.

```
public class SampleViews
{
    ...
    private static class SupplierBinding extends SerialSerialBinding {
        private SupplierBinding(ClassCatalog classCatalog,
                                Class keyClass,
                                Class dataClass)
        {
            super(classCatalog, keyClass, dataClass);
        }

        public Object entryToObject(Object keyInput, Object dataInput)
        {
            SupplierKey key = (SupplierKey) keyInput;
            SupplierData data = (SupplierData) dataInput;
            return new Supplier(key.getNumber(), data.getName(),
                                data.getStatus(), data.getCity());
        }

        public Object objectToKey(Object object)
        {
            Supplier supplier = (Supplier) object;
            return new SupplierKey(supplier.getNumber());
        }

        public Object objectToData(Object object)
        {
            Supplier supplier = (Supplier) object;
            return new SupplierData(supplier.getName(), supplier.getStatus(),
                                    supplier.getCity());
        }
    }

    private static class ShipmentBinding extends SerialSerialBinding {
        private ShipmentBinding(ClassCatalog classCatalog,
                                Class keyClass,
                                Class dataClass)
        {
            super(classCatalog, keyClass, dataClass);
        }

        public Object entryToObject(Object keyInput, Object dataInput)
        {
            ShipmentKey key = (ShipmentKey) keyInput;
            ShipmentData data = (ShipmentData) dataInput;
            return new Shipment(key.getPartNumber(), key.getSupplierNumber(),
                                data.getQuantity());
```

```
        }

        public Object objectToKey(Object object)
        {
            Shipment shipment = (Shipment) object;
            return new ShipmentKey(shipment.getPartNumber(),
                                   shipment.getSupplierNumber());
        }

        public Object objectToData(Object object)
        {
            Shipment shipment = (Shipment) object;
            return new ShipmentData(shipment.getQuantity());
        }
    }
    ...
}
```

Creating Collections with Entity Bindings

Stored map objects are created in this example in the same way as in prior examples, but using entity bindings in place of value bindings. All value objects passed and returned to the Java collections API are then actually entity objects (`Part`, `Supplier` and `Shipment`). The application no longer deals directly with plain value objects (`PartData`, `SupplierData` and `ShipmentData`).

Since the `partValueBinding`, `supplierValueBinding` and `shipmentValueBinding` were defined as entity bindings in the prior section, there are no source code changes necessary for creating the stored map objects.

```
public class SampleViews
{
    ...
    public SampleViews(SampleDatabase db)
    {
        ...
        partMap =
            new StoredMap(db.getPartDatabase(),
                          partKeyBinding, partValueBinding, true);
        supplierMap =
            new StoredMap(db.getSupplierDatabase(),
                          supplierKeyBinding, supplierValueBinding, true);
        shipmentMap =
            new StoredMap(db.getShipmentDatabase(),
                          shipmentKeyBinding, shipmentValueBinding, true);
      ...
    }
```

Specifying an `EntityBinding` will select a different `StoredMap` constructor, but the syntax is the same. In general, an entity binding may be used anywhere that a value binding is used.

The following getter methods are defined for use by other classes in the example program. Instead of returning the map's entry set (`Map.entrySet`), the map's value set (`Map.values`) is returned. The entry set was convenient in prior examples because it allowed enumerating all key/value pairs in the collection. Since an entity contains the key and the value, enumerating the value set can now be used more conveniently for the same purpose.

```
import com.sleepycat.collections.StoredValueSet;
...
public class SampleViews
{
    ...
    public StoredValueSet getPartSet()
    {
        return (StoredValueSet) partMap.values();
    }

    public StoredValueSet getSupplierSet()
    {
        return (StoredValueSet) supplierMap.values();
    }

    public StoredValueSet getShipmentSet()
    {
        return (StoredValueSet) shipmentMap.values();
    }
    ...
}
```

Notice that the collection returned by the `StoredMap.values` method is actually a `StoredValueSet` and not just a `Collection` as defined by the `Map.values` interface. As long as duplicate keys are not allowed, this collection will behave as a true set and will disallow the addition of duplicates, etc.

Using Entities with Collections

In this example entity objects, rather than key and value objects, are used for adding and enumerating the records in a collection. Because fewer classes and objects are involved, adding and enumerating is done more conveniently and more simply than in the prior examples.

For adding and iterating entities, the collection of entities returned by `Map.values` is used. In general, when using an entity binding, all Java collection methods that are passed or returned a value object will be passed or returned an entity object instead.

The `Sample` class has been changed in this example to add objects using the `Set.add` method rather than the `Map.put` method that was used in the prior examples. Entity objects are constructed and passed to `Set.add`.

```
import java.util.Set;
...
public class Sample
```

```
{
    ...
    private void addSuppliers()
    {
        Set suppliers = views.getSupplierSet();
        if (suppliers.isEmpty())
        {
            System.out.println("Adding Suppliers");
            suppliers.add(new Supplier("S1", "Smith", 20, "London"));
            suppliers.add(new Supplier("S2", "Jones", 10, "Paris"));
            suppliers.add(new Supplier("S3", "Blake", 30, "Paris"));
            suppliers.add(new Supplier("S4", "Clark", 20, "London"));
            suppliers.add(new Supplier("S5", "Adams", 30, "Athens"));
        }
    }

    private void addParts()
    {
        Set parts = views.getPartSet();
        if (parts.isEmpty())
        {
            System.out.println("Adding Parts");
            parts.add(new Part("P1", "Nut", "Red",
                      new Weight(12.0, Weight.GRAMS), "London"));
            parts.add(new Part("P2", "Bolt", "Green",
                      new Weight(17.0, Weight.GRAMS), "Paris"));
            parts.add(new Part("P3", "Screw", "Blue",
                      new Weight(17.0, Weight.GRAMS), "Rome"));
            parts.add(new Part("P4", "Screw", "Red",
                      new Weight(14.0, Weight.GRAMS), "London"));
            parts.add(new Part("P5", "Cam", "Blue",
                      new Weight(12.0, Weight.GRAMS), "Paris"));
            parts.add(new Part("P6", "Cog", "Red",
                      new Weight(19.0, Weight.GRAMS), "London"));
        }
    }

    private void addShipments()
    {
        Set shipments = views.getShipmentSet();
        if (shipments.isEmpty())
        {
            System.out.println("Adding Shipments");
            shipments.add(new Shipment("P1", "S1", 300));
            shipments.add(new Shipment("P2", "S1", 200));
            shipments.add(new Shipment("P3", "S1", 400));
            shipments.add(new Shipment("P4", "S1", 200));
            shipments.add(new Shipment("P5", "S1", 100));
            shipments.add(new Shipment("P6", "S1", 100));
```

```
            shipments.add(new Shipment("P1", "S2", 300));
            shipments.add(new Shipment("P2", "S2", 400));
            shipments.add(new Shipment("P2", "S3", 200));
            shipments.add(new Shipment("P2", "S4", 200));
            shipments.add(new Shipment("P4", "S4", 300));
            shipments.add(new Shipment("P5", "S4", 400));
        }
    }
```

Instead of printing the key/value pairs by iterating over the `Map.entrySet` as done in the prior example, this example iterates over the entities in the `Map.values` collection.

```
import java.util.Iterator;
import java.util.Set;
...
public class Sample
{
    ...
    private class PrintDatabase implements TransactionWorker
    {
        public void doWork()
            throws Exception
        {
            printValues("Parts",
                        views.getPartSet().iterator());
            printValues("Suppliers",
                        views.getSupplierSet().iterator());
            printValues("Suppliers for City Paris",
                        views.getSupplierByCityMap().duplicates(
                                        "Paris").iterator());
            printValues("Shipments",
                        views.getShipmentSet().iterator());
            printValues("Shipments for Part P1",
                        views.getShipmentByPartMap().duplicates(
                                        new PartKey("P1")).iterator());
            printValues("Shipments for Supplier S1",
                        views.getShipmentBySupplierMap().duplicates(
                                        new SupplierKey("S1")).iterator());
        }
    }
    ...
}
```

The output of the example program is shown below.

```
Adding Suppliers
Adding Parts
Adding Shipments

--- Parts ---
```

```
Part: number=P1 name=Nut color=Red weight=[12.0 grams] city=London
Part: number=P2 name=Bolt color=Green weight=[17.0 grams] city=Paris
Part: number=P3 name=Screw color=Blue weight=[17.0 grams] city=Rome
Part: number=P4 name=Screw color=Red weight=[14.0 grams] city=London
Part: number=P5 name=Cam color=Blue weight=[12.0 grams] city=Paris
Part: number=P6 name=Cog color=Red weight=[19.0 grams] city=London

--- Suppliers ---
Supplier: number=S1 name=Smith status=20 city=London
Supplier: number=S2 name=Jones status=10 city=Paris
Supplier: number=S3 name=Blake status=30 city=Paris
Supplier: number=S4 name=Clark status=20 city=London
Supplier: number=S5 name=Adams status=30 city=Athens

--- Suppliers for City Paris ---
Supplier: number=S2 name=Jones status=10 city=Paris
Supplier: number=S3 name=Blake status=30 city=Paris

--- Shipments ---
Shipment: part=P1 supplier=S1 quantity=300
Shipment: part=P1 supplier=S2 quantity=300
Shipment: part=P2 supplier=S1 quantity=200
Shipment: part=P2 supplier=S2 quantity=400
Shipment: part=P2 supplier=S3 quantity=200
Shipment: part=P2 supplier=S4 quantity=200
Shipment: part=P3 supplier=S1 quantity=400
Shipment: part=P4 supplier=S1 quantity=200
Shipment: part=P4 supplier=S4 quantity=300
Shipment: part=P5 supplier=S1 quantity=100
Shipment: part=P5 supplier=S4 quantity=400
Shipment: part=P6 supplier=S1 quantity=100

--- Shipments for Part P1 ---
Shipment: part=P1 supplier=S1 quantity=300
Shipment: part=P1 supplier=S2 quantity=300

--- Shipments for Supplier S1 ---
Shipment: part=P1 supplier=S1 quantity=300
Shipment: part=P2 supplier=S1 quantity=200
Shipment: part=P3 supplier=S1 quantity=400
Shipment: part=P4 supplier=S1 quantity=200
Shipment: part=P5 supplier=S1 quantity=100
Shipment: part=P6 supplier=S1 quantity=100
```

Chapter 5. Using Tuples

JE Collections API *tuples* are sequences of primitive Java data types, for example, integers and strings. The *tuple format* is a binary format for tuples that can be used to store keys and/or values.

Tuples are useful as keys because they have a meaningful sort order, while serialized objects do not. This is because the binary data for a tuple is written in such a way that its raw byte ordering provides a useful sort order. For example, strings in tuples are written with a null terminator rather than with a leading length.

Tuples are useful as keys *or* values when reducing the record size to a minimum is important. A tuple is significantly smaller than an equivalent serialized object. However, unlike serialized objects, tuples cannot contain complex data types and are not easily extended except by adding fields at the end of the tuple.

Whenever a tuple format is used, except when the key or value class is a Java primitive wrapper class, a *tuple binding* class must be implemented to map between the Java object and the tuple fields. Because of this extra requirement, and because tuples are not easily extended, a useful technique shown in this example is to use tuples for keys and serialized objects for values. This provides compact ordered keys but still allows arbitrary Java objects as values, and avoids implementing a tuple binding for each value class.

Compare this example to the prior Entity example and you'll see that the `Sample` class has not changed. When changing a database format, while new bindings are needed to map key and value objects to the new format, the application using the objects often does not need to be modified.

The complete source of the final version of the example program is included in the Berkeley DB distribution.

Using the Tuple Format

Tuples are sequences of primitive Java values that can be written to, and read from, the raw data bytes of a stored record. The primitive values are written or read one at a time in sequence, using the JE Collections API `TupleInput` and `TupleOutput` classes. These classes are very similar to the standard Java `DataInput` and `DataOutput` interfaces. The primary difference is the binary format of the data, which is designed for sorting in the case of tuples.

For example, to read and write a tuple containing two string values, the following code snippets could be used.

```
import com.sleepycat.bind.tuple.TupleInput;
import com.sleepycat.bind.tuple.TupleOutput;
...
TupleInput input;
TupleOutput output;
...
String partNumber = input.readString();
```

```
String supplierNumber = input.readString();
...
output.writeString(partNumber);
output.writeString(supplierNumber);
```

Since a tuple is defined as an ordered sequence, reading and writing order must match. If the wrong data type is read (an integer instead of string, for example), an exception may be thrown or at minimum invalid data will be read.

When the tuple format is used, bindings and key creators must read and write tuples using the tuple API as shown above. This will be illustrated in the next two sections.

Using Tuples with Key Creators

Key creators were used in prior examples to extract index keys from value objects. The keys were returned as deserialized key objects, since the serial format was used for keys. In this example, the tuple format is used for keys and the key creators return keys by writing information to a tuple. The differences between this example and the prior example are:

- The `TupleSerialKeyCreator` base class is used instead of the `SerialSerialKeyCreator` base class.
- For all key input and output parameters, the `TupleInput` and `TupleOutput` classes are used instead of `Object` (representing a deserialized object).
- Instead of returning a key output object, these methods call tuple write methods such as `TupleOutput.writeString`.

In addition to writing key tuples, the `ShipmentByPartKeyCreator` and `ShipmentBySupplierKeyCreator` classes also read the key tuple of the primary key. This is because they extract the index key from fields in the Shipment's primary key. Instead of calling getter methods on the `ShipmentKey` object, as in prior examples, these methods call `TupleInput.readString`. The `ShipmentKey` consists of two string fields that are read in sequence.

The modified key creators are shown below: `SupplierByCityKeyCreator`, `ShipmentByPartKeyCreator` and `ShipmentBySupplierKeyCreator`.

```
import com.sleepycat.bind.serial.TupleSerialKeyCreator;
import com.sleepycat.bind.tuple.TupleInput;
import com.sleepycat.bind.tuple.TupleOutput;
...
public class SampleDatabase
{
    ...
    private static class SupplierByCityKeyCreator
        extends TupleSerialKeyCreator
    {
        private SupplierByCityKeyCreator(StoredClassCatalog catalog,
                                         Class valueClass)
        {
```

```
            super(catalog, valueClass);
        }

        public boolean createSecondaryKey(TupleInput primaryKeyInput,
                                          Object valueInput,
                                          TupleOutput indexKeyOutput)
        {
            SupplierData supplierData = (SupplierData) valueInput;
            String city = supplierData.getCity();
            if (city != null) {
                indexKeyOutput.writeString(supplierData.getCity());
                return true;
            } else {
                return false;
            }
        }
    }

    private static class ShipmentByPartKeyCreator
        extends TupleSerialKeyCreator
    {
        private ShipmentByPartKeyCreator(StoredClassCatalog catalog,
                                         Class valueClass)
        {
            super(catalog, valueClass);
        }

        public boolean createSecondaryKey(TupleInput primaryKeyInput,
                                          Object valueInput,
                                          TupleOutput indexKeyOutput)
        {
            String partNumber = primaryKeyInput.readString();
            // don't bother reading the supplierNumber
            indexKeyOutput.writeString(partNumber);
            return true;
        }
    }

    private static class ShipmentBySupplierKeyCreator
        extends TupleSerialKeyCreator
    {
        private ShipmentBySupplierKeyCreator(StoredClassCatalog catalog,
                                             Class valueClass)
        {
            super(catalog, valueClass);
        }

        public boolean createSecondaryKey(TupleInput primaryKeyInput,
                                          Object valueInput,
```

```
                                            TupleOutput indexKeyOutput)
        {
            primaryKeyInput.readString(); // skip the partNumber
            String supplierNumber = primaryKeyInput.readString();
            indexKeyOutput.writeString(supplierNumber);
            return true;
        }
    }
    ...
}
```

Creating Tuple Key Bindings

Serial bindings were used in prior examples as key bindings, and keys were stored as serialized objects. In this example, a tuple binding is used for each key since keys will be stored as tuples. Because keys are no longer stored as serialized objects, the `PartKey`, `SupplierKey` and `ShipmentKey` classes no longer implement the `Serializable` interface (this is the only change to these classes and is not shown below).

For the `Part` key, `Supplier` key, and `Shipment` key, the `SampleViews` class was changed in this example to create a custom `TupleBinding` instead of a `SerialBinding`. The custom tuple key binding classes are defined further below.

```
import com.sleepycat.bind.tuple.TupleBinding;
...
public class SampleViews
{
    ...
    public SampleViews(SampleDatabase db)
    {
        ...
        ClassCatalog catalog = db.getClassCatalog();
        EntryBinding partKeyBinding =
            new PartKeyBinding();
        EntityBinding partDataBinding =
            new PartBinding(catalog, PartData.class);
        EntryBinding supplierKeyBinding =
            new SupplierKeyBinding();
        EntityBinding supplierDataBinding =
            new SupplierBinding(catalog, SupplierData.class);
        EntryBinding shipmentKeyBinding =
            new ShipmentKeyBinding();
        EntityBinding shipmentDataBinding =
            new ShipmentBinding(catalog, ShipmentData.class);
        EntryBinding cityKeyBinding =
            TupleBinding.getPrimitiveBinding(String.class);
        ...
```

```
    }
}
```

For the City key, however, a custom binding class is not needed because the key class is a primitive Java type, `String`. For any primitive Java type, a tuple binding may be created using the `TupleBinding.getPrimitiveBinding` static method.

The custom key binding classes, `PartKeyBinding`, `SupplierKeyBinding` and `ShipmentKeyBinding`, are defined by extending the `TupleBinding` class. The `TupleBinding` abstract class implements the `EntryBinding` interface, and is used for one-to-one bindings between tuples and objects. Each binding class implements two methods for converting between tuples and objects. Tuple fields are read using the `TupleInput` parameter and written using the `TupleOutput` parameter.

```
import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.bind.tuple.TupleInput;
import com.sleepycat.bind.tuple.TupleOutput;
...
public class SampleViews
{
...

    private static class PartKeyBinding extends TupleBinding
    {
        private PartKeyBinding()
        {
        }

        public Object entryToObject(TupleInput input)
        {
            String number = input.readString();
            return new PartKey(number);
        }

        public void objectToEntry(Object object, TupleOutput output)
        {
            PartKey key = (PartKey) object;
            output.writeString(key.getNumber());
        }
    }
    ...
    private static class SupplierKeyBinding extends TupleBinding
    {
        private SupplierKeyBinding()
        {
        }

        public Object entryToObject(TupleInput input)
        {
            String number = input.readString();
```

```
            return new SupplierKey(number);
        }

        public void objectToEntry(Object object, TupleOutput output)
        {
            SupplierKey key = (SupplierKey) object;
            output.writeString(key.getNumber());
        }
    }
    ...
    private static class ShipmentKeyBinding extends TupleBinding
    {
        private ShipmentKeyBinding()
        {
        }

        public Object entryToObject(TupleInput input)
        {
            String partNumber = input.readString();
            String supplierNumber = input.readString();
            return new ShipmentKey(partNumber, supplierNumber);
        }

        public void objectToEntry(Object object, TupleOutput output)
        {
            ShipmentKey key = (ShipmentKey) object;
            output.writeString(key.getPartNumber());
            output.writeString(key.getSupplierNumber());
        }
    }
    ...
}
```

Creating Tuple-Serial Entity Bindings

In the prior example serial keys and serial values were used, and the `SerialSerialBinding` base class was used for entity bindings. In this example, tuple keys and serial values are used and therefore the `TupleSerialBinding` base class is used for entity bindings.

As with any entity binding, a key and value is converted to an entity in the `TupleSerialBinding.entryToObject` method, and from an entity to a key and value in the `TupleSerialBinding.objectToKey` and `TupleSerialBinding.objectToData` methods. But since keys are stored as tuples, not as serialized objects, key fields are read and written using the `TupleInput` and `TupleOutput` parameters.

The `SampleViews` class contains the modified entity binding classes that were defined in the prior example: `PartBinding`, `SupplierBinding` and `ShipmentBinding`.

```
import com.sleepycat.bind.serial.TupleSerialBinding;
import com.sleepycat.bind.tuple.TupleInput;
import com.sleepycat.bind.tuple.TupleOutput;
...
public class SampleViews
{
    ...
    private static class PartBinding extends TupleSerialBinding
    {
        private PartBinding(ClassCatalog classCatalog, Class dataClass)
        {
            super(classCatalog, dataClass);
        }
        public Object entryToObject(TupleInput keyInput, Object dataInput)
        {
            String number = keyInput.readString();
            PartData data = (PartData) dataInput;
            return new Part(number, data.getName(), data.getColor(),
                            data.getWeight(), data.getCity());
        }
        public void objectToKey(Object object, TupleOutput output)
        {
            Part part = (Part) object;
            output.writeString(part.getNumber());
        }
        public Object objectToData(Object object)
        {
            Part part = (Part) object;
            return new PartData(part.getName(), part.getColor(),
                                part.getWeight(), part.getCity());
        }
    }
    ...
    private static class SupplierBinding extends TupleSerialBinding
    {
        private SupplierBinding(ClassCatalog classCatalog, Class dataClass)
        {
            super(classCatalog, dataClass);
        }
        public Object entryToObject(TupleInput keyInput, Object dataInput)
        {
            String number = keyInput.readString();
            SupplierData data = (SupplierData) dataInput;
            return new Supplier(number, data.getName(),
                                data.getStatus(), data.getCity());
        }
        public void objectToKey(Object object, TupleOutput output)
        {
            Supplier supplier = (Supplier) object;
```

```
            output.writeString(supplier.getNumber());
        }
        public Object objectToData(Object object)
        {
            Supplier supplier = (Supplier) object;
            return new SupplierData(supplier.getName(), supplier.getStatus(),
                                    supplier.getCity());
        }
    }
    ...
    private static class ShipmentBinding extends TupleSerialBinding
    {
        private ShipmentBinding(ClassCatalog classCatalog, Class dataClass)
        {
            super(classCatalog, dataClass);
        }
        public Object entryToObject(TupleInput keyInput, Object dataInput)
        {
            String partNumber = keyInput.readString();
            String supplierNumber = keyInput.readString();
            ShipmentData data = (ShipmentData) dataInput;
            return new Shipment(partNumber, supplierNumber,
                                data.getQuantity());
        }
        public void objectToKey(Object object, TupleOutput output)
        {
            Shipment shipment = (Shipment) object;
            output.writeString(shipment.getPartNumber());
            output.writeString(shipment.getSupplierNumber());
        }
        public Object objectToData(Object object)
        {
            Shipment shipment = (Shipment) object;
            return new ShipmentData(shipment.getQuantity());
        }
    }
    ...
}
```

Using Sorted Collections

In general, no changes to the prior example are necessary to use collections having tuple keys. Iteration of elements in a stored collection will be ordered by the sort order of the tuples.

Although not shown in the example, all methods of the `SortedMap` and `SortedSet` interfaces may be used with sorted collections. For example, submaps and subsets may be created.

The output of the example program shows that records are sorted by key value.

```
Adding Suppliers
Adding Parts
Adding Shipments

--- Parts ---
Part: number=P1 name=Nut color=Red weight=[12.0 grams] city=London
Part: number=P2 name=Bolt color=Green weight=[17.0 grams] city=Paris
Part: number=P3 name=Screw color=Blue weight=[17.0 grams] city=Rome
Part: number=P4 name=Screw color=Red weight=[14.0 grams] city=London
Part: number=P5 name=Cam color=Blue weight=[12.0 grams] city=Paris
Part: number=P6 name=Cog color=Red weight=[19.0 grams] city=London

--- Suppliers ---
Supplier: number=S1 name=Smith status=20 city=London
Supplier: number=S2 name=Jones status=10 city=Paris
Supplier: number=S3 name=Blake status=30 city=Paris
Supplier: number=S4 name=Clark status=20 city=London
Supplier: number=S5 name=Adams status=30 city=Athens

--- Suppliers for City Paris ---
Supplier: number=S2 name=Jones status=10 city=Paris
Supplier: number=S3 name=Blake status=30 city=Paris

--- Shipments ---
Shipment: part=P1 supplier=S1 quantity=300
Shipment: part=P1 supplier=S2 quantity=300
Shipment: part=P2 supplier=S1 quantity=200
Shipment: part=P2 supplier=S2 quantity=400
Shipment: part=P2 supplier=S3 quantity=200
Shipment: part=P2 supplier=S4 quantity=200
Shipment: part=P3 supplier=S1 quantity=400
Shipment: part=P4 supplier=S1 quantity=200
Shipment: part=P4 supplier=S4 quantity=300
Shipment: part=P5 supplier=S1 quantity=100
Shipment: part=P5 supplier=S4 quantity=400
Shipment: part=P6 supplier=S1 quantity=100

--- Shipments for Part P1 ---
Shipment: part=P1 supplier=S1 quantity=300
```

```
Shipment: part=P1 supplier=S2 quantity=300

--- Shipments for Supplier S1 ---
Shipment: part=P1 supplier=S1 quantity=300
Shipment: part=P2 supplier=S1 quantity=200
Shipment: part=P3 supplier=S1 quantity=400
Shipment: part=P4 supplier=S1 quantity=200
Shipment: part=P5 supplier=S1 quantity=100
Shipment: part=P6 supplier=S1 quantity=100
```

Chapter 6. Using Serializable Entities

In the prior examples that used entities (the Entity and Tuple examples) you may have noticed the redundancy between the serializable value classes and the entity classes. An entity class by definition contains all properties of the value class as well as all properties of the key class.

When using serializable values it is possible to remove this redundancy by changing the entity class in two ways:

- Make the entity class serializable, so it can be used in place of the value class.
- Make the key fields transient, so they are not redundantly stored in the record.

The modified entity class can then serve double-duty: It can be serialized and stored as the record value, and it can be used as the entity class as usual along with the Java collections API. The `PartData`, `SupplierData` and `ShipmentData` classes can then be removed.

Transient fields are defined in Java as fields that are not stored in the serialized form of an object. Therefore, when an object is deserialized the transient fields must be explicitly initialized. Since the entity binding is responsible for creating entity objects, it is the natural place to initialize the transient key fields.

Note that it is not strictly necessary to make the key fields of a serializable entity class transient. If this is not done, the key will simply be stored redundantly in the record's value. This extra storage may or may not be acceptable to an application. But since we are using tuple keys and an entity binding class must be implemented anyway to extract the key from the entity, it is sensible to use transient key fields to reduce the record size. Of course there may be a reason that transient fields are not desired; for example, if an application wants to serialize the entity objects for other purposes, then using transient fields should be avoided.

The complete source of the final version of the example program is included in the Berkeley DB distribution.

Using Transient Fields in an Entity Class

The entity classes in this example are redefined such that they can be used both as serializable value classes and as entity classes. Compared to the prior example there are three changes to the `Part`, `Supplier` and `Shipment` entity classes:

- Each class now implements the `Serializable` interface.
- The key fields in each class are declared as `transient`.
- A package-private `setKey()` method is added to each class for initializing the transient key fields. This method will be called from the entity bindings.

```
import java.io.Serializable;
...
public class Part implements Serializable
{
```

```
    private transient String number;
    private String name;
    private String color;
    private Weight weight;
    private String city;

    public Part(String number, String name, String color, Weight weight,
                String city)
    {
        this.number = number;
        this.name = name;
        this.color = color;
        this.weight = weight;
        this.city = city;
    }

    final void setKey(String number)
    {
        this.number = number;
    }

    public final String getNumber()
    {
        return number;
    }

    public final String getName()
    {
        return name;
    }

    public final String getColor()
    {
        return color;
    }

    public final Weight getWeight()
    {
        return weight;
    }

    public final String getCity()
    {
        return city;
    }

    public String toString()
    {
        return "Part: number=" + number +
```

```
                " name=" + name +
                " color=" + color +
                " weight=" + weight +
                " city=" + city + '.';
    }
}
...
public class Supplier implements Serializable
{
    private transient String number;
    private String name;
    private int status;
    private String city;

    public Supplier(String number, String name, int status, String city)
    {
        this.number = number;
        this.name = name;
        this.status = status;
        this.city = city;
    }

    void setKey(String number)
    {
        this.number = number;
    }

    public final String getNumber()
    {
        return number;
    }

    public final String getName()
    {
        return name;
    }

    public final int getStatus()
    {
        return status;
    }

    public final String getCity()
    {
        return city;
    }

    public String toString()
    {
```

```
        return "Supplier: number=" + number +
               " name=" + name +
               " status=" + status +
               " city=" + city + '.';
    }
}
...
public class Shipment implements Serializable
{
    private transient String partNumber;
    private transient String supplierNumber;
    private int quantity;

    public Shipment(String partNumber, String supplierNumber, int quantity)
    {
        this.partNumber = partNumber;
        this.supplierNumber = supplierNumber;
        this.quantity = quantity;
    }

    void setKey(String partNumber, String supplierNumber)
    {
        this.partNumber = partNumber;
        this.supplierNumber = supplierNumber;
    }

    public final String getPartNumber()
    {
        return partNumber;
    }

    public final String getSupplierNumber()
    {
        return supplierNumber;
    }

    public final int getQuantity()
    {
        return quantity;
    }

    public String toString()
    {
        return "Shipment: part=" + partNumber +
                " supplier=" + supplierNumber +
                " quantity=" + quantity + '.';
    }
}
```

Using Transient Fields in an Entity Binding

The entity bindings from the prior example have been changed in this example to use the entity object both as a value object and an entity object.

Before, the `entryToObject()` method combined the deserialized value object with the key fields to create a new entity object. Now, this method uses the deserialized object directly as an entity, and initializes its key using the fields read from the key tuple.

Before, the `objectToData()` method constructed a new value object using information in the entity. Now it simply returns the entity. Nothing needs to be changed in the entity, since the transient key fields won't be serialized.

```
import com.sleepycat.bind.serial.ClassCatalog;
...
public class SampleViews
{
    ...
    private static class PartBinding extends TupleSerialBinding
    {
        private PartBinding(ClassCatalog classCatalog, Class dataClass)
        {
            super(classCatalog, dataClass);
        }

        public Object entryToObject(TupleInput keyInput, Object dataInput)
        {
            String number = keyInput.readString();
            Part part = (Part) dataInput;
            part.setKey(number);
            return part;
        }

        public void objectToKey(Object object, TupleOutput output)
        {
            Part part = (Part) object;
            output.writeString(part.getNumber());
        }

        public Object objectToData(Object object)
        {
            return object;
        }
    }

    private static class SupplierBinding extends TupleSerialBinding
    {
        private SupplierBinding(ClassCatalog classCatalog, Class dataClass)
        {
```

```
            super(classCatalog, dataClass);
        }

        public Object entryToObject(TupleInput keyInput, Object dataInput)
        {
            String number = keyInput.readString();
            Supplier supplier = (Supplier) dataInput;
            supplier.setKey(number);
            return supplier;
        }

        public void objectToKey(Object object, TupleOutput output)
        {
            Supplier supplier = (Supplier) object;
            output.writeString(supplier.getNumber());
        }

        public Object objectToData(Object object)
        {
            return object;
        }
    }

    private static class ShipmentBinding extends TupleSerialBinding
    {
        private ShipmentBinding(ClassCatalog classCatalog, Class dataClass)
        {
            super(classCatalog, dataClass);
        }

        public Object entryToObject(TupleInput keyInput, Object dataInput)
        {
            String partNumber = keyInput.readString();
            String supplierNumber = keyInput.readString();
            Shipment shipment = (Shipment) dataInput;
            shipment.setKey(partNumber, supplierNumber);
            return shipment;
        }

        public void objectToKey(Object object, TupleOutput output)
        {
            Shipment shipment = (Shipment) object;
            output.writeString(shipment.getPartNumber());
            output.writeString(shipment.getSupplierNumber());
        }

        public Object objectToData(Object object)
        {
            return object;
```

```
        }
    }
}
```

Removing the Redundant Value Classes

The `PartData`, `SupplierData` and `ShipmentData` classes have been removed in this example, and the `Part`, `Supplier` and `Shipment` entity classes are used in their place.

The serial formats are created with the entity classes.

```
public class SampleDatabase
{
    ...
    public SampleDatabase(String homeDirectory)
        throws DatabaseException, FileNotFoundException
    {
        ...
        secConfig.setKeyCreator(new SupplierByCityKeyCreator(javaCatalog,
                                                             Supplier.class));
        ...
        secConfig.setKeyCreator(new ShipmentByPartKeyCreator(javaCatalog,
                                                             Shipment.class));
        ...
        secConfig.setKeyCreator(new ShipmentBySupplierKeyCreator(javaCatalog,
                                                             Shipment.class));
        ...
    }
}
```

The index key creator uses the entity class as well.

```
public class SampleDatabase
{
    ...

    private static class SupplierByCityKeyCreator
        extends TupleSerialKeyCreator
    {
        private SupplierByCityKeyCreator(ClassCatalog catalog,
                                         Class valueClass)
        {
            super(catalog, valueClass);
        }

        public boolean createSecondaryKey(TupleInput primaryKeyInput,
                                          Object valueInput,
                                          TupleOutput indexKeyOutput)
        {
            Supplier supplier = (Supplier) valueInput;
```

```
            String city = supplier.getCity();
            if (city != null) {
                indexKeyOutput.writeString(supplier.getCity());
                return true;
            } else {
                return false;
            }
        }
    }
}
```

Chapter 7. Summary

In summary, the JE Collections API tutorial has demonstrated how to create different types of bindings, as well as how to use the basic facilities of the JE Collections API: the environment, databases, secondary indices, collections, and transactions. The final approach illustrated by the last example program, Serializable Entity, uses tuple keys and serial entity values. Hopefully it is clear that any type of object-to-data binding may be implemented by an application and used along with standard Java collections.

The following table summarizes the differences between the examples in the tutorial.

Example	Key	Value	Entity	Comments
The Basic Program (page 6)	Serial	Serial	No	The shipment program
Using Secondary Indices and Foreign keys (page 32)	Serial	Serial	No	Secondary indices and foreign keys
Using Entity Classes (page 46)	Serial	Serial	Yes	Combining the key and value in a single object
Using Tuples (page 58)	Tuple	Serial	Yes	Compact ordered keys
Using Serializable Entities (page 68)	Tuple	Serial	Yes	One serializable class for entities and values

Having completed this tutorial, you may want to explore how other types of bindings can be implemented. The bindings shown in this tutorial are all *external bindings*, meaning that the data classes themselves contain none of the binding implementation. It is also possible to implement *internal bindings*, where the data classes implement the binding.

Internal bindings are called *marshalled bindings* in the JE Collections API, and in this model each data class implements a marshalling interface. A single external binding class that understands the marshalling interface is used to call the internal bindings of each data object, and therefore the overall model and API is unchanged. To learn about marshalled bindings, see the `marshal` and `factory` examples that came with your JE distribution (you can find them in `<INSTALL_DIR>/examples/collections/ship` where `<INSTALL_DIR>` is the location where you unpacked your JE distribution). These examples continue building on the example programs used in the tutorial. The Marshal program is the next program following the Serializable Entity program, and the Factory program follows the Marshal program. The source code comments in these examples explain their differences.

Appendix A. API Notes and Details

This appendix contains information useful to the collections programmer that is too detailed to easily fit into the format of a tutorial. Specifically, this appendix contains the following information:

Using Data Bindings

Data bindings determine how keys and values are represented as stored data (byte arrays) in the database, and how stored data is converted to and from Java objects.

The selection of data bindings is, in general, independent of the selection of collection views. In other words, any binding can be used with any collection.

In this document, bindings are described in the context of their use for stored data in a database. However, bindings may also be used independently of a database to operate on an arbitrary byte array. This allows using bindings when data is to be written to a file or sent over a network, for example.

Selecting Binding Formats

For the key and value of each stored collection, you may select one of the following types of bindings.

Binding Format	Ordered	Description
`SerialBinding`	No	The data is stored using a compact form of Java serialization, where the class descriptions are stored separately in a catalog database. Arbitrary Java objects are supported.
`TupleBinding`	Yes	The data is stored using a series of fixed length primitive values or zero terminated character arrays (strings). Class/type evolution is not supported.
Custom binding format	User-defined	The data storage format and ordering is determined by the custom binding implementation.

As shown in the table above, the tuple format supports built-in ordering (without specifying a custom comparator), while the serial format does not. This means that when a specific key order is needed, tuples should be used instead of serial data. Alternatively, a custom Btree comparator should be specified using `DatabaseConfig.setBtreeComparator()`. Note that a custom Btree comparator will usually execute more slowly than the default byte-by-byte comparison. This makes using tuples an attractive option, since they provide ordering along with optimal performance.

The tuple binding uses less space and executes faster than the serial binding. But once a tuple is written to a database, the order of fields in the tuple may not be changed and fields may not be deleted. The only type evolution allowed is the addition of fields at the end of the tuple, and this must be explicitly supported by the custom binding implementation.

The serial binding supports the full generality of Java serialization including type evolution. But serialized data can only be accessed by Java applications, its size is larger, and its bindings are slower to execute.

Selecting Data Bindings

There are two types of binding interfaces. Simple entry bindings implement the `EntryBinding` interface and can be used for key or value objects. Entity bindings implement the `EntityBinding` interface and are used for combined key and value objects called entities.

Simple entry bindings map between the key or value data stored by Berkeley DB and a key or value object. This is a simple one-to-one mapping.

Simple entry bindings are easy to implement and in some cases require no coding. For example, a `SerialBinding` can be used for keys or values without writing any additional code. A tuple binding for a single-item tuple can also be used without writing any code; see the `TupleBinding.getPrimitiveBinding()` method.

Entity bindings must divide an entity object into its key and value data, and then combine the key and value data to re-create the entity object. This is a two-to-one mapping.

Entity bindings are useful when a stored application object naturally has its primary key as a property, which is very common. For example, an Employee object would naturally have an EmployeeNumber property (its primary key) and an entity binding would then be needed. Of course, entity bindings are more complex to implement, especially if their key and data formats are different.

Note that even when an entity binding is used a key binding is also usually needed. For example, a key binding is used to create key objects that are passed to the `Map.get()` method. A key object is passed to this method even though it may return an entity that also contains the key.

Implementing Bindings

There are two ways to implement bindings. The first way is to create a binding class that implements one of the two binding interfaces, `EntryBinding` or `EntityBinding`. For tuple bindings and serial bindings there are a number of abstract classes that make this easier. For example, you can extend `TupleBinding` to implement a simple binding for a tuple key or value. Abstract classes are also provided for entity bindings and are named after the format names of the key and value. For example, you can extend `TupleSerialBinding` to implement an entity binding with a tuple key and serial value.

Another way to implement bindings is with marshalling interfaces. These are interfaces which perform the binding operations and are implemented by the key, value or entity classes themselves. With marshalling you use a binding which calls the marshalling interface and you implement the marshalling interface for each key, value or entity class. For example, you can use `TupleMarshalledBinding` along with key or value classes that implement the `MarshalledTupleEntry` interface.

Using Bindings

Bindings are specified whenever a stored collection is created. A key binding must be specified for map, key set and entry set views. A value binding or entity binding must be specified for map, value set and entry set views.

Any number of bindings may be created for the same stored data. This allows multiple views over the same data. For example, a tuple might be bound to an array of values or to a class with properties for each object.

It is important to be careful of bindings that only use a subset of the stored data. This can be useful to simplify a view or to hide information that should not be accessible. However, if you

write records using these bindings you may create stored data that is invalid from the application's point of view. It is up to the application to guard against this by creating a read-only collection when such bindings are used.

Secondary Key Creators

Secondary Key Creators are needed whenever database indices are used. For each secondary index (`SecondaryDatabase`) a key creator is used to derive index key data from key/value data. Key creators are objects whose classes implement the `SecondaryKeyCreator` interface.

Like bindings, key creators may be implemented using a separate key creator class or using a marshalling interface. Abstract key creator classes and marshalling interfaces are provided in the com.sleepycat.bind.tuple and com.sleepycat.bind.serial packages.

Unlike bindings, key creators fundamentally operate on key and value data, not necessarily on the objects derived from the data by bindings. In this sense key creators are a part of a database definition, and may be independent of the various bindings that may be used to view data in a database. However, key creators are not prohibited from using higher level objects produced by bindings, and doing so may be convenient for some applications. For example, marshalling interfaces, which are defined for objects produced by bindings, are a convenient way to define key creators.

Using the JE Collections API

An `Environment` manages the resources for one or more data stores. A `Database` object represents a single database and is created via a method on the environment object. `SecondaryDatabase` objects represent an index associated with a primary database. Primary and secondary databases are then used to create stored collection objects, as described in Using Stored Collections (page 82).

Using Transactions

Once you have an environment, one or more databases, and one or more stored collections, you are ready to access (read and write) stored data. For a transactional environment, a transaction must be started before accessing data, and must be committed or aborted after access is complete. The JE Collections API provides several ways of managing transactions.

The recommended technique is to use the `TransactionRunner` class along with your own implementation of the `TransactionWorker` interface. `TransactionRunner` will call your `TransactionWorker` implementation class to perform the data access or work of the transaction. This technique has the following benefits:

- Transaction exceptions will be handled transparently and retries will be performed when deadlocks are detected.
- The transaction will automatically be committed if your `TransactionWorker.doWork()` method returns normally, or will be aborted if `doWork()` throws an exception.
- `TransactionRunner` can be used for non-transactional environments as well, allowing you to write your application independently of the environment.

If you don't want to use `TransactionRunner`, the alternative is to use the `CurrentTransaction` class.

1. Obtain a CurrentTransaction instance by calling the `CurrentTransaction.getInstance` method. The instance returned can be used by all threads in a program.

2. Use `CurrentTransaction.beginTransaction()`, `CurrentTransaction.commitTransaction()` and `CurrentTransaction.abortTransaction()` to directly begin, commit and abort transactions.

If you choose to use CurrentTransaction directly you must handle the `DeadlockException` exception and perform retries yourself. Also note that CurrentTransaction may only be used in a transactional environment.

The JE Collections API supports transaction auto-commit. If no transaction is active and a write operation is requested for a transactional database, auto-commit is used automatically.

The JE Collections API also supports transaction dirty-read via the `StoredCollections` class. When dirty-read is enabled for a collection, data will be read that has been modified by another transaction but not committed. Using dirty-read can improve concurrency since reading will not wait for other transactions to complete. For a non-transactional container, dirty-read has no effect. See `StoredCollections` for how to create a dirty-read collection.

Transaction Rollback

When a transaction is aborted (or rolled back) the application is responsible for discarding references to any data objects that were modified during the transaction. Since the JE Collections API treats data by value, not by reference, neither the data objects nor the JE Collections API objects contain status information indicating whether the data objects are 1- in sync with the database, 2- dirty (contain changes that have not been written to the database), 3- stale (were read previously but have become out of sync with changes made to the database), or 4- contain changes that cannot be committed because of an aborted transaction.

For example, a given data object will reflect the current state of the database after reading it within a transaction. If the object is then modified it will be out of sync with the database. When the modified object is written to the database it will then be in sync again. But if the transaction is aborted the object will then be out of sync with the database. References to objects for aborted transactions should no longer be used. When these objects are needed later they should be read fresh from the database.

When an existing stored object is to be updated, special care should be taken to read the data, then modify it, and then write it to the database, all within a single transaction. If a stale data object (an object that was read previously but has since been changed in the database) is modified and then written to the database, database changes may be overwritten unintentionally.

When an application enforces rules about concurrent access to specific data objects or all data objects, the rules described here can be relaxed. For example, if the application knows that a certain object is only modified in one place, it may be able to reliably keep a current copy of that object. In that case, it is not necessary to reread the object before updating it. That

said, if arbitrary concurrent access is to be supported, the safest approach is to always read data before modifying it within a single transaction.

Similar concerns apply to using data that may have become stale. If the application depends on current data, it should be read fresh from the database just before it is used.

Access Method Restrictions

The BTREE access method is always used for JE Databases. Sorted duplicates — more then one record for a single key — are optional.

The restrictions imposed by the access method on the database model are:

- If duplicates are allowed then more than one value may be associated with the same key. This means that the data store cannot be strictly considered a map — it is really a multi-map. See Using Stored Collections (page 82) for implications on the use of the collection interfaces.
- If duplicate keys are allowed for a data store then the data store may not have secondary indices.
- With sorted duplicates, all values for the same key must be distinct.

See Using Stored Collections (page 82) for more information on how access methods impact the use of stored collections.

Using Stored Collections

When a stored collection is created it is based on either a `Database` or a `SecondaryDatabase`. When a database is used, the primary key of the database is used as the collection key. When a secondary database is used, the index key is used as the collection key. Indexed collections can be used for reading elements and removing elements but not for adding or updating elements.

Stored Collection and Access Methods

The use of stored collections is constrained in certain respects as described below.

- All iterators for stored collections implement the `ListIterator` interface as well as the `Iterator` interface. `ListIterator.hasPrevious()` and `ListIterator.previous()` work in all cases.
- `ListIterator.add()` throws `UnsupportedOperationException` if duplicates are not allowed.
- `ListIterator.add()` inserts a duplicate in sorted order if sorted duplicates are configured.
- `ListIterator.set()` throws `UnsupportedOperationException` if sorted duplicates are configured, since updating with sorted duplicates would change the iterator position.
- `ListIterator.nextIndex()` and `ListIterator.previousIndex()` always throw `UnsupportedOperationException`.

- `Map.Entry.setValue()` throws `UnsupportedOperationException` if duplicates are sorted.
- When duplicates are allowed the `Collection` interfaces are modified in several ways as described in the next section.

Stored Collections Versus Standard Java Collections

Stored collections have the following differences with the standard Java collection interfaces. Some of these are interface contract violations.

The Java collections interface does not support duplicate keys (multi-maps or multi-sets). When the access method allows duplicate keys, the collection interfaces are defined as follows.

- `Map.entrySet()` may contain multiple `Map.Entry` objects with the same key.
- `Map.keySet()` always contains unique keys, it does not contain duplicates.
- `Map.values()` contains all values including the values associated with duplicate keys.
- `Map.put()` appends a duplicate if the key already exists rather than replacing the existing value, and always returns null.
- `Map.remove()` removes all duplicates for the specified key.
- `Map.get()` returns the first duplicate for the specified key.
- `StoredMap.duplicates()` is an additional method for returning the values for a given key as a `Collection`.

Other differences are:

- Collection.size() and Map.size() always throws `UnsupportedOperationException`. This is because the number of records in a database cannot be determined reliably or cheaply.
- Because the size() method cannot be used, the bulk operation methods of standard Java collections cannot be passed stored collections as parameters, since the implementations rely on size(). However, the bulk operation methods of stored collections can be passed standard Java collections as parameters. `storedCollection.addAll(standardCollection)` is allowed while `standardCollection.addAll(storedCollection)` is *not* allowed. This restriction applies to the standard collection constructors that take a Collection parameter (copy constructors), the Map.putAll() method, and the following Collection methods: addAll(), containsAll(), removeAll() and retainAll().
- `Comparator` objects cannot be used and the `SortedMap.comparator()` and `SortedSet.comparator()` methods always return null. The `Comparable` interface is not supported. However, Comparators that operate on byte arrays may be specified using `DatabaseConfig.setBtreeComparator`.
- The `Object.equals()` method is not used to determine whether a key or value is contained in a collection, to locate a value by key, etc. Instead the byte array representation of the keys and values are used. However, the equals() method *is* called for each key and value

when comparing two collections for equality. It is the responsibility of the application to make sure that the equals() method returns true if and only if the byte array representations of the two objects are equal. Normally this occurs naturally since the byte array representation is derived from the object's fields.

Other Stored Collection Characteristics

The following characteristics of stored collections are extensions of the definitions in the `java.util` package. These differences do not violate the Java collections interface contract.

- All stored collections are thread safe (can be used by multiple threads concurrently). Locking is handled by the Berkeley DB Java Edition environment. To access a collection from multiple threads, creation of synchronized collections using the `Collections` class is not necessary. Iterators, however, should always be used only by a single thread.
- All stored collections may be read-only if desired by passing false for the writeAllowed parameter of their constructor. Creation of immutable collections using the `Collections` class is not necessary.
- A stored collection is partially read-only if a secondary index is used. Specifically, values may be removed but may not be added or updated. The following methods will throw `UnsupportedOperationException` when an index is used: `Collection.add()`, `ListIterator.set()` and `Map.Entry.setValue()`.
- `SortedMap.entrySet()` and `SortedMap.keySet()` return a `SortedSet`, not just a `Set` as specified in Java collections interface. This allows using the `SortedSet` methods on the returned collection.
- `SortedMap.values()` returns a `SortedSet`, not just a `Collection`, whenever the keys of the map can be derived from the values using an entity binding. Note that the sorted set returned is not really a set if duplicates are allowed, since it is technically a collection; however, the `SortedSet` methods (for example, subSet()), can still be used.
- For `SortedSet` and `SortedMap` views, additional subSet() and subMap() methods are provided that allow control over whether keys are treated as inclusive or exclusive values in the key range.
- Keys and values are stored by value, not by reference. This is because objects that are added to collections are converted to byte arrays (by bindings) and stored in the database. When they are retrieved from the collection they are read from the database and converted from byte arrays to objects. Therefore, the object reference added to a collection will not be the same as the reference later retrieved from the collection.
- A runtime exception, `RuntimeExceptionWrapper`, is thrown whenever database exceptions occur which are not runtime exceptions. The `RuntimeExceptionWrapper.getCause()` method can be called to get the underlying exception.
- All iterators for stored collections implement the `ListIterator` interface as well as the `Iterator` interface. This is to allow use of the `ListIterator.hasPrevious()` and

`ListIterator.previous()` methods, which work for all collections since Berkeley DB provides bidirectional cursors.

- All stored collections have a `StoredCollection.iterator(boolean)` method that allows creating a read-only iterator for a writable collection. For the standard `Collection.iterator()` method, the iterator is read-only only when the collection is read-only.
- Iterator stability for stored collections is greater than the iterator stability defined by the Java collections interfaces. Stored iterator stability is the same as the cursor stability defined by Berkeley DB.
- When an entity binding is used, updating (setting) a value is not allowed if the key in the entity is not equal to the original key. For example, calling `Map.put()` is not allowed when the key parameter is not equal to the key of the entity parameter. `Map.put()`, `ListIterator.set()`, and `Map.Entry.setValue()` will throw `IllegalArgumentException` in this situation.
- The `StoredMap.append(java.lang.Object)` extension method allows adding a new record with an automatically assigned key. An application-defined `PrimaryKeyAssigner` is used to assign the key value.

Why Java Collections for Berkeley DB Java Edition

The Java collections interface was chosen as the best Java API for JE given these requirements:

1. Provide the Java developer with an API that is as familiar and easy to use as possible.
2. Provide access to all, or a large majority, of the features of the underlying Berkeley DB Java Edition storage system.
3. Compared to the JE API, provide a higher-level API that is oriented toward Java developers.
4. For ease of use, support object-to-data bindings, per-thread transactions, and some traditional database features such as foreign keys.
5. Provide a thin layer that can be thoroughly tested and which does not significantly impact the reliability and performance of JE.

Admittedly there are several things about the Java Collections API that don't quite fit with JE or with any transactional database, and therefore there are some new rules for applying the Java Collections API. However, these disadvantages are considered to be smaller than the disadvantages of the alternatives:

- A new API not based on the Java Collections API could have been designed that maps well to JE but is higher-level. However, this would require designing an entirely new model. The exceptions for using the Java Collections API are considered easier to learn than a whole new model. A new model would also require a long design stabilization period before being as complete and understandable as either the Java Collections API or the JE API.
- The ODMG API or another object persistence API could have been implemented on top of JE. However, an object persistence implementation would add much code and require a long

stabilization period. And while it may work well for applications that require object persistence, it would probably never perform well enough for many other applications.

Serialized Object Storage

Serialization of an object graph includes class information as well as instance information. If more than one instance of the same class is serialized as separate serialization operations then the class information exists more than once. To eliminate this inefficiency the `StoredClassCatalog` class will store the class format for all database records stored using a `SerialBinding`. Refer to the `ship` sample code for examples (the class `SampleDatabase` in `<INSTALL_DIR>/examples/collections/ship/basic/SampleDatabase.java` is a good place to start).