Using the BIND APIs

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

- Single field numerical and string objects

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

- Complex objects that implement Java serialization.

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

- Non-serialized complex objects.

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

Numerical and String Objects

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

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

To store primitive data using the Bind APIs:

1. Create an `EntryBinding` object.

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

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

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

```
package je.gettingStarted;

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

```
import com.sleepycat.je.DatabaseEntry;

...

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

    // Now build the DatabaseEntry using a TupleBinding
    Long myLong = new Long(1234567891);
    DatabaseEntry theData = new DatabaseEntry();
    EntryBinding myBinding = TupleBinding.getPrimitiveBinding(Long.class);
    myBinding.objectToEntry(myLong, theData);

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

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

```
package je.gettingStarted;

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

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

...

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

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

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

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

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

Serializable Complex Objects

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

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

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

The easiest way for you to solve this problem is to use the Bind APIs to perform the serialization for you. Doing so causes the extra object information to be saved off to a unique `Database` dedicated for that purpose. This means that you do not have to duplicate that information on each record in the `Database` that your application is using to store its information.

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

Usage Caveats

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

- If you need to sort based on the objects your are storing. The sort order is meaningless for the byte arrays that you obtain through serialization. Consequently, you should not use serialization for keys if you care about their sort order. You should also not use serialization for record data if your `Database` supports duplicate records and you care about sort order.
- You want to minimize the size of your byte arrays. Even when using the Bind APIs to perform the serialization the resulting `byte` array may be larger than necessary. You can achieve more compact results by building your own custom tuple binding.
- You want to optimize for speed. In general, custom tuple bindings are faster than serialization at moving data in and out of `byte` arrays.
- You are using custom comparators. In JE, comparators are instantiated and called internally whenever databases are not accessible. Because serial bindings depend on the class catalog, a serial binding binding cannot be used during these times. As a result, attempting to use a serial binding with a custom comparator will result in a `NullPointerException` during environment open or close.

For information on building your own custom tuple binding, see Custom Tuple Bindings.

Serializing Objects

To store a serializable complex object using the Bind APIs:

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

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

```
package je.gettingStarted;

import java.io.Serializable;

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

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

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

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

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

    public long getLong() {
        return longData;
    }

    public double getDouble() {
        return doubleData;
    }

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

You can then store instances of this class as follows:

```
package je.gettingStarted;

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

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

...

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

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

try {
    // Environment open omitted for brevity

    // Open the database that you will use to store your data
    DatabaseConfig myDbConfig = new DatabaseConfig();
    myDbConfig.setAllowCreate(true);
    myDbConfig.setSortedDuplicates(true);
```

```
    Database myDatabase = myDbEnv.openDatabase(null, "myDb", myDbConfig);

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

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

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

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

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

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

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

Deserializing Objects

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

```
package je.gettingStarted;

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

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

...

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

try {
    // Environment open omitted for brevity.

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

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

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

    // Create the binding
```

```
    EntryBinding dataBinding = new SerialBinding(classCatalog,
                                                 MyData.class);

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

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

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

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

Custom Tuple Bindings

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

For information on using serialization to store complex objects, see Serializable Complex Objects.

To store complex objects using a custom tuple binding:

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

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

```
package je.gettingStarted;

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

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

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

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

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

    public long getLong() {
```

```
        return longData;
    }

    public Double getDouble() {
        return doubleData;
    }

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

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

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

For example:

```
package je.gettingStarted;

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

public class MyTupleBinding extends TupleBinding {

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

        MyData2 myData = (MyData2)object;

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

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

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

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

```
        return myData;
    }
}
```

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

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

For example:

```
package je.gettingStarted;

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

...

TupleBinding keyBinding = new MyTupleBinding();

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

DatabaseEntry myKey = new DatabaseEntry();

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

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

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