Berkeley DB stores data as simple key/data pairs, where the key and data item are both byte arrays. In the DBMS application that you will build in the assignments, you'll need to be able to take a collection of fields and convert them to byte array, and vice versa. The process of converting data into a form that stored in the database is called marshalling. The reverse process is called unmarshalling.

In the DBMS programming assignments, you will perform marshalling/unmarshalling by using the TupleOutput and TupleInput classes from Berkeley DB’s Java API. In this document, we'll look at how this is done.

An Example of Marshalling Data
To begin, let's look at some code that we've given you in the Catalog class for marshalling table metadata – i.e., data about the names and types of the columns that belong to the table. The Catalog.putMetadata() method takes a Table object as a parameter, and it creates a key/data pair for that table's metadata that can be stored in the catalog (a Berkeley DB database). The key is a single value (the table's name), so we don't need to perform any marshalling for it. However, the data item contains a number of different components, so we need to marshall them into a single byte array.

We begin the putMetadata() method with a check that ensures that the Table object has the necessary column information. We then create a TupleOutput object:

```java
tuple = new TupleOutput();
```

This object will allow us to write the individual components of the data item into a byte array, filling the array from left to right.

We start by writing the number of columns in the table. This has to come first, so that we will know how much column information to read in during unmarshalling. We assume that there are no more than 256 columns in any one table, and thus we store this number as a single byte using the TupleInput.writeByte method:

```java
tuple.writeByte(table.numColumns());
```

An Example of Marshalling Data
To begin, let's look at some code that we've given you in the Catalog class for marshalling table metadata – i.e., data about the names and types of the columns that belong to the table. The Catalog.putMetadata() method takes a Table object as a parameter, and it creates a key/data pair for that table's metadata that can be stored in the catalog (a Berkeley DB database). The key is a single value (the table's name), so we don't need to perform any marshalling for it. However, the data item contains a number of different components, so we need to marshall them into a single byte array.

We begin the putMetadata() method with a check that ensures that the Table object has the necessary column information. We then create a TupleOutput object:

```java
tuple = new TupleOutput();
```

This object will allow us to write the individual components of the data item into a byte array, filling the array from left to right.

We start by writing the number of columns in the table. This has to come first, so that we will know how much column information to read in during unmarshalling. We assume that there are no more than 256 columns in any one table, and thus we store this number as a single byte using the TupleInput.writeByte method:

```java
tuple.writeByte(table.numColumns());
```
We then use a for loop to write the metadata for each column:

```java
for (int i = 0; i < table.numColumns(); i++) {
    Column col = table.getColumn(i);

    /* Write the length of the column name, followed by the name. */
    String colName = col.getName();
    tuple.writeByte(colName.length());
    tuple.writeBytes(colName);

    /* Write the rest of the column information. */
    tuple.writeByte(col.getType());
    tuple.writeInt(col.getLength());
    tuple.writeBoolean(col.isNotNull());
    tuple.writeBoolean(col.isPrimaryKey());
}
```

Note that we store the following information for each column:

- the length of the column's name (stored as a byte); this will be used during unmarshalling so that we know how many bytes to read for the name
- the actual column name (stored as a collection of bytes using the writeBytes method – note the s at the end of the method name – which takes a string and writes it into the byte array)
- the type of column (stored as a one-byte integer that tells us whether it is a CHAR, VARCHAR, etc.; see Column.java for the actual integers that we use)
- the length of the column (stored as a four-byte integer using the writeInt method)
- whether the column is a NOT NULL column (stored as a boolean using the writeBoolean method)
- whether the column is the primary key (also stored as a boolean)

It's worth noting that we designed the format of the byte arrays for the table metadata with the awareness that we will always unmarshall all of the metadata for a given table. When you design the format for byte arrays that you will use to store the data values in a given tuple, you'll need to do so in a way that allows you to efficiently access the value of an arbitrary column without unmarshalling the entire tuple. See the notes on record formats in the lecture notes on storage and indexing for a reminder of some of the possible options for doing this.

The for loop completes the process of marshalling the metadata into the TupleOutput object's byte array. We then create the two DatabaseEntry objects needed for the key/data pair – one for the key, and one for the data item. Because we're using the table's name as the key, we create the key DatabaseEntry object as follows:

```java
String tableName = table.getName();
DatabaseEntry key = null;
try {
    key = new DatabaseEntry(tableName.getBytes("UTF-8"));
} catch (UnsupportedEncodingException e) {
    System.err.println(e);
    DBMS.abort();
}
```
Note that we use the version of the `DatabaseEntry` constructor that takes just an array of bytes, and we're using the `String` class's `getBytes()` method to convert the characters in the table's name to an array of bytes from the UTF-8 character encoding. We use a `try-catch` block to handle the unlikely case that the table's name includes characters that cannot be represented in UTF-8.

The data item's `DatabaseEntry` is created from the data that we marshalled into the `TupleOutput` object's byte array:

```java
DatabaseEntry data = new DatabaseEntry(tuple.getBufferBytes(), 0, tuple.getBufferLength());
```

Here we use a different `DatabaseEntry` constructor that takes three arguments: an array of bytes containing the values from our `TupleOutput` object's byte array (obtained using the `getBufferBytes()` method); an index 0 specifying that we want to start at the beginning of that array; and the number of bytes that we want to use from that array (obtained using the `getBufferLength()` method). We need to use this three-argument constructor because the `TupleOutput` object's byte array may be longer than the number of bytes that we actually wrote into it, and we only want the data item to include the bytes that we wrote.

Finally, we put the key/data pair into the Berkeley DB database that we are using for the catalog. We use the `putNoOverwrite` method to ensure that we don't overwrite the metadata for an existing table with the same name.

**Exercise 1: Practice With Marshalling**

To practice the process of marshalling data, try writing a code fragment that creates and puts a key/data pair in which the key is the string "654321" and the data item is a marshalled form of the following fields:

```java
String name = "John Doe";
int SSN = 123456;
double balance = 134.5;
boolean joint = false;
```

To simplify things, you can make the same assumption that we made with the table metadata – that it is not necessary to be able to "jump" to a particular field, but rather that you will always unmarshall the entire data item. Model your code on the code from the `Catalog.putMetadata()` method, and consult the API documentation for Berkeley DB's Java API as needed.

**An Example of Unmarshalling Data**

Now let's look at the code that we have given you for unmarshalling table metadata. The `Catalog.getMetadata()` method takes a `Table` object as a parameter, and it gets the key/data pair for that table's metadata from the catalog and unmarshalls it so that the column information can be added to the `Table` object.

We begin by preparing the `DatabaseEntry` objects for the key/data pair that will be retrieved from the catalog database. The `DatabaseEntry` for the key is again based on the name of the table, so we prepare it using the same steps taken in the `putMetadata()` method (see above).
In this case, the data item will be read from the database, so we prepare an initially empty DatabaseEntry for it:

```java
DatabaseEntry data = new DatabaseEntry();
```

We then perform the get operation needed to retrieve the key/data pair from the catalog database. Assuming that the operation succeeds, we then create a TupleInput object from the database's byte array, which we obtain using the `getData()` method:

```java
TupleInput tuple = new TupleInput(data.getData());
```

This object will allow us to unmarshall the values from the byte array, reading them in the same order that they were written in put Metadata.

We begin by reading the number of columns using the `readByte()` method:

```java
int numColumns = tuple.readByte();
```

We then use a `for` loop to read in the metadata for each column. These values are used to initialize the corresponding Column object, which is then added to the Table object:

```java
for (int i = 0; i < numColumns; i++) {
    /* the column name */
    int colNameLength = tuple.readByte();
    String colName = tuple.readBytes(colNameLength);
    Column col = new Column(colName, table);

    /* the column info */
    col.setType(tuple.readByte());
    col.setLength(tuple.readInt());
    ColumnOptions opts = new ColumnOptions(tuple.readBoolean(),
                                           tuple.readBoolean());
    col.applyOpts(opts);
    col.setIndex(i);

    /* add the column to the table */
    table.addColumn(col);
}
```

Note that each value is read using the TupleInput method that corresponds to the TupleOutput method used to write it: `readByte` to read a value written by `writeByte`, `readBytes` to read a value written by `writeBytes`, etc.

**Exercise 2: Practice With Unmarshalling**

To practice the process of unmarshalling data, try writing a code fragment that gets and unmarshalls the key/data pair that you created in Exercise 1.