

Simpler J2EE Data Access Using iBATIS

Sandeep Jain

Contents

How to read an XML Document Type Definition	4
iBATIS: The Basics.....	6
2.1 Introduction.....	6
2.2 Object-Relational Mapping vs. Data Mapping.....	6
2.3 iBATIS Data Mapper and Data Access Objects.....	7
2.4 Things You Need to Create to Run iBATIS.....	7
2.5 An Example of iBATIS at Work.....	7
2.5.1 Install the RDBMS, get the JDBC Connection Settings.....	7
2.5.2 Create your Tables and Add your Data.....	8
2.5.3 Create your SQL-Map File.....	9
2.5.4 Copy the JPetStore Configuration File.....	9
2.5.5 Create your Java File.....	9
2.5.6 Copy your iBATIS Jar Files.....	10
2.5.7 Run your iBATIS Application.....	10
How to Write SQL Queries Using iBATIS.....	12
3.1 Introduction.....	12
3.2 The root element: <sqlMap>.....	12
3.3 The <typeAlias> element.....	12
3.4 Parameter Maps, Result Maps, and SQL Queries.....	12
3.4.1 Explicit Parameter Maps.....	13
3.4.2 The Query Elements <update>, <insert>, and <delete>.....	13
3.4.3 Inline Parameter Maps.....	14
3.4.4 More about the <insert> element.....	15
3.4.5 Explicit Result Maps.....	16
3.4.6 The <select> element.....	16
3.4.7 Implicit Result Maps.....	17
How to Use iBATIS's Java Methods.....	18
4.1 Introduction and Shortcuts.....	18
4.2 The Java API for iBATIS.....	18
4.2.1 Methods for Reading the Configuration File.....	19
4.2.2 Methods for Data Access.....	19
4.2.3 Calling Stored Procedures.....	20
4.2.4 Batch Updates for Efficiency.....	20
4.2.5 Transactions.....	21
How to Configure iBATIS.....	22
5.1 Introduction and Shortcuts.....	22
5.2 Configuration File Definition: The <sqlMapConfig> element.....	22
5.3 The simpler elements: <typeAlias>, <properties>, <sqlMap>.....	22
5.4 The <settings> element.....	23
5.5 The <transactionManager> element.....	25
How to Speed Up Queries by Caching.....	26
6.1 Introduction and Shortcuts.....	26
6.2 DTD Fragment for Caching.....	26

How to Use iBATIS Data Access Objects.....	29
7.1 Introduction and Shortcuts.....	29
7.2 A Working Example of iBATIS DAO.....	29
7.2.1 List of Additional Program Files.....	29
7.2.2 Content of Files.....	30
7.2.3 Copy Jar File for DAO.....	32
7.2.4 Compile and Run the iBATIS DAO based program.....	32
7.3 The Concepts of Data Access Objects.....	32
7.3.1 Context.....	32
7.3.2 Transaction Managers.....	33
7.3.1 Interfaces.....	33
7.3.2 Implementations.....	33
How to Use If-Conditions and Loop within iBATIS SQL Queries.....	34
8.1 Introduction and Shortcuts.....	34
8.2 Conditional elements.....	34
8.2.1 Unary Conditional Elements.....	35
8.2.2 Binary Conditional Elements.....	36
8.2.3 Other Conditional Elements.....	37
8.3 Looping: the <iterate> element.....	37
iBATIS in Your J2EE Architecture.....	39
9.1 Introduction and Shortcuts.....	39
9.2 Running the JPetStore Example.....	39
9.3 Recommended Architecture Diagram.....	39

Chapter 1

How to read an XML Document Type Definition

Once you start using iBATIS, you will find that you spend most of your time filling in a certain type of XML file. It follows that this book is mostly about that XML file.

The technique chosen to explain the XML file to you is to present the DTD, or Document Type Definition of the XML file to you in fragments. If you already know how to read a DTD, you can skip this chapter. If not, this chapter will explain it to you.

A DTD is a document that defines the structure of the XML file, and lets you tell the difference between files with correct and incorrect elements.

One fragment of the XML file you will be using most of the time looks like this (we'll talk more about this fragment in a subsequent chapter):

```
<!ELEMENT selectKey (#PCDATA | include) *>
  <!ATTLIST selectKey
    resultClass CDATA #IMPLIED
    keyProperty CDATA #IMPLIED
  >

<!ELEMENT include EMPTY>
  <!ATTLIST include
    refid CDATA #REQUIRED
  >
```

The first line says that `<selectKey>` is an element; the brackets after `<selectKey>` indicate that it is an element that has children, and the children could be parsed character data ("PCDATA") or the `<include>` element. Parsed character data is any string inside which the XML processing program can search for XML elements. Hence, the XML for the above DTD fragment could look like this:

```
<selectKey resultClass="com.mycompany.MyClass" keyProperty="some_property">
  <include refid="some_refid"/>
</selectKey>
```

The `EMPTY` keyword in the `<include>` element of the DTD means that the element `<include>` will not have any children, although it does have an attribute, `refid`. The attributes `resultClass` and `keyProperty` are tagged with the keywords `CDATA #IMPLIED`. `CDATA` means unparsed character data. This means that the values for `resultClass` and `keyProperty` will not be examined from within by the XML processing program for XML tags. `#IMPLIED` means that specifying a value for this attribute is optional. If you do not supply a value, the XML

processing program is expected to supply its own value. In contrast, the attribute for the <include> element, *refid*, is required because of the tag CDATA #REQUIRED.

We will look at another DTD fragment from the iBATIS XML DTD before we start talking about iBATIS:

```
<!ELEMENT cacheModel (flushInterval?, flushOnExecute*, property*)+>
  <!ATTLIST cacheModel
    id CDATA #REQUIRED
    type CDATA #REQUIRED
    readOnly (true | false) #IMPLIED
    serialize (true | false) #IMPLIED
  >
```

We want to pay attention here to the child elements of the element <cacheModel>, and to the attributes *readOnly* and *serialize*. The question-mark after the element <flushInterval> means that the element <flushInterval> can occur at most once and optionally not at all, as a child element of the element <cacheModel>. The * after the child element <flushOnExecute> means that it may occur zero or more times (as many as required), as a child element of <cacheModel>. If this * had been a +, it would have meant that the element had to occur at least once, and could occur as many times as needed.

The text (true | false) after the *readOnly* and *serialize* attributes means that the values of these attributes have to be either "true" or "false", and the #IMPLIED keyword means that if a value is not supplied, the XML processing program is expected to supply it automatically.

The + at the end of the <cacheModel> element declaration means that of the children defined, at least one must be present.

Since the child elements are separated by commas in the DTD, in the XML file, they must occur in the same order as the one by which they are presented in the <cacheModel> element declaration.

Chapter 2

iBATIS: The Basics

2.1 Introduction

This chapter gives you an overview of the central concepts of iBATIS. It also helps you get up and running with a very simple program that uses iBATIS to fetch data from a relational database and print it to the console. You should make sure to run this program. After you have run it, you will find that everything else falls into place in a simple way. If you do not run it, you may find the text very theoretical.

2.2 Object-Relational Mapping vs. Data Mapping

Today's server based software applications, in almost every case, store their data in relational databases. A relational database models data as a set of tables, linked to each other by common columns. The software application that relies on the database models data in another form. If the application is written in an object-oriented language such as C# or Java, data is represented as variables contained in Objects, where Objects can either contain other Objects, or inherit their data members from parent Objects. In this book we will use Java for all our examples, and also say 'Java' instead of 'programming language', to be concrete. iBATIS was originally written for Java, and then extended to C#.

Because of the difference in the way that the software application models the data, and how the relational database models the data, there is a need to 'map' the models from one to the other. The following are the two commonest ways of doing this:

1. Object-Relational Mapping (O/R Mapping)

The best known O/R Mapping tool is Hibernate. It defines direct mappings between the variables within objects, to the columns within relational tables. These mappings are stored in XML files. To query the database, you use a query language defined by Hibernate.

2. Data Mapping

iBATIS is the most popular example of a data mapping tool. Instead of mapping application objects to relational tables, you map the variables within the application object to the variables in the SQL query. The query language continues to be SQL, with some minor syntax variations introduced to allow you to put the values of your application variables inside the SQL query. Since this book is on iBATIS, you may detect a bias for the latter approach.

2.3 iBATIS Data Mapper and Data Access Objects

iBATIS contains two software modules that can be used independently of each other (although they work well when used together). What we have been talking about all this while, and will be discussing for the most part in this book, is the Data Mapper. The Data Mapper is also known as iBATIS SQL-Maps. We will refer to it as SQL-Maps in this book.

We explain the other item, Data Access Objects, in Chapter 7: <TODO: Updated Name> How to Set up a Common Interface Layer to Accommodate Tools other than iBATIS. Chapter 7 is the only chapter that discusses Data Access Objects.

2.4 Things You Need to Create to Run iBATIS

To give you a concrete picture of iBATIS, we list here the end products of your effort in working with iBATIS:

1. Java file(s) with calls to methods exposed by iBATIS

iBATIS has a small set of methods that you will typically use - less than twenty. You will write at least one Java file, and most likely more, that will call some of these methods from the iBATIS API.

2. SQL-Mapping file(s)

A SQL-Mapping file is an XML file in which you write the SQL queries that your application will use. In addition, you define here the mapping from the variables in your Java objects to place-holders in your SQL queries, and the mapping from the results returned by the SQL query, back to a Java object. You will see this at work in the next chapter.

3. Configuration file

This is another XML file, a single XML file for your application, where you define configuration settings, such as, among other things, the names of the SQL-Mapping files in (2) above. Once you understand these files, you will have understood the essentials of iBATIS.

4. Relational Database

Since the purpose of iBATIS is to access data in a relational database, you need to have done your data modeling and design, and created the relational tables you will need. You can then interact with the data using iBATIS.

2.5 An Example of iBATIS at Work

This example will give you a feel for iBATIS. The seven section headings seem like a lot. But you have to do this kind of thing to use any data access framework. Once you get it running, you will find that it is easy to climb the iBATIS ladder and learn whatever else you need to, for your current application.

2.5.1 Install the RDBMS, get the JDBC Connection Settings

Perhaps the trickiest part of the process is to get the JDBC connection settings for your RDBMS. If you have an RDBMS installed, and don't know the connection settings, search the web for something like "oracle jdbc", or whatever your RDBMS is. I am going to assume from here that you do not have an RDBMS installed to begin with, that you need to install one for running iBATIS, and that you are going to install it on the same machine as the one on which you are running your Java code.

The most popular open source relational database products are MySQL and PostgreSQL. I experimented with the latest versions (MySQL 5.0 and PostgreSQL 8.1) at the time of writing this book, and found that PostgreSQL

8.1 was easier to use than MySQL 5.0. So I'll build my instructions around PostgreSQL, and you can adapt them to your database.

Installing PostgreSQL 8.1

Obtain the PostgreSQL installer from postgresql.org, and install it. Allow the installer to install it as a service. This makes it easier to work with, and you can always turn it off. Remember the account name and password that you choose. These are part of what you need to connect with the database. The JDBC driver comes built in. Also, select the option to install the PostgreSQL administration tool. You will find that it is convenient for administration, and also as a general purpose SQL client.

Database Connection Settings

If you have PostgreSQL installed, the JDBC connection settings you need are as follows:

```
driver=org.postgresql.Driver
url=jdbc:postgresql://localhost:5432:your_dbname
username=your_username
password=your_password
```

Copy these into a file named
database.properties

We will get to the directory structure shortly. Incidentally, if you are using Oracle, the database.properties file will be:

```
driver=oracle.jdbc.driver.OracleDriver
url=jdbc:oracle:thin:@localhost:1521:your_dbname
username=your_username
password=your_password
```

2.5.2 Create your Tables and Add your Data

Our simple database has a single table:

CARS	
NAME	FROM
Honda	Japan
Ford	USA
Volvo	Sweden
GM	USA
Toyota	Japan
Peugeot	France
Hyundai	South Korea
Nissan	Japan

NAME and FROM are the two columns. The SQL code to create the table and add the data is:

```
create table CARS ( NAME VARCHAR, COUNTRY VARCHAR );

insert into CARS(NAME, COUNTRY) values( 'Honda', 'Japan' );
insert into CARS(NAME, COUNTRY) values( 'Ford', 'USA' );
insert into CARS(NAME, COUNTRY) values( 'Volvo', 'Sweden' );
insert into CARS(NAME, COUNTRY) values( 'GM', 'USA' );
```

```

insert into CARS (NAME, COUNTRY) values ( 'Toyota', 'Japan' );
insert into CARS (NAME, COUNTRY) values ( 'Peugeot', 'France' );
insert into CARS (NAME, COUNTRY) values ( 'Hyundai', 'South Korea' );
insert into CARS (NAME, COUNTRY) values ( 'Nissan', 'Japan' );

```

If you are following along on your computer, you can insert the data now, and verify that it has been added properly. That's all we need to do with the database.

2.5.3 Create your SQL-Map File

Now you create a 'SQL-Map' file (we'll explain what this is in the next chapter). First, create a root level directory, say `ibatisapp` (if you are using Unix), or on the C drive if you are using Windows. Create a file in this directory called `sqlMapCars.xml`. Also put the `database.properties` file you created earlier, into this directory.

Put the following code in `sqlMapCars.xml`.

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap namespace="Cars">
  <select id="selectAllNames" resultClass="java.lang.String">
    select NAME from CARS;
  </select>
</sqlMap>

```

2.5.4 Copy the JPetStore Configuration File

Now you need a file to configure iBATIS. This is called the SQL Map Configuration file. Chapter 5 is devoted to this. For now, it would be better to just copy it from the JPetStore example released with iBATIS. If you are at the root of the JPetStore folder, the file is located at:

```
JPetStore\src\com\ibatis\jpetstore\persistence\sqlmapdao\sql
```

Copy this file to your `ibatisapp` directory. You can leave the file as is, except remove all the `<sqlMap>` elements at the bottom of the file, and replace with a single `sqlMap` element:

```
<sqlMap resource="sqlMapCars.xml"/>
```

2.5.5 Create your Java File

Create a file named `IbatisExerciser.java` in the `ibatisapp` directory, as follows:

```

import java.util.List;
import java.io.IOException;
import java.io.Reader;
import java.sql.SQLException;
import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.*;

class IbatisExerciser {

    void doIbatis() {
        // Read the config file with the overall configuration settings
        String configFile = "sql-map-config.xml";

        Reader reader = null;
        try {
            reader = Resources.getResourceAsReader(configFile);
        } catch (IOException ioe) {

```

```

        ioe.printStackTrace();
    }
    SqlMapClient sqlMapClient = SqlMapClientBuilder.buildSqlMapClient( reader );

    List carNames = null;
    // Query the database using iBATIS
    try {
        carNames = sqlMapClient.queryForList("selectAllNames", null);
    } catch ( SQLException sqle ) {
        sqle.printStackTrace();
    }

    // Print result to screen
    for( int i=0; i < carNames.size(); i++ ) {
        System.out.println( (String)carNames.get( i ) );
    }
}

public static void main( String args[] ) {
    IbatisExerciser exerciser = new IbatisExerciser();
    exerciser.doIbatis();
}
}

```

The first three lines in `IbatisExerciser.java` have the job of reading the configuration file, and creating an instance of the interface `SqlMapClient`, which you will use to send a query to iBATIS. The central line of the program is:

```
List carNames = sqlMapClient.queryForList("selectAllNames", null);
```

The first argument in `queryForList` finds the correct *select* query in the file `sqlMapCars.xml`. The second argument can be used to send parameter values to the *select* statement, but is null in this case because the *select* statement in the file `sqlMapCars.xml` has no *where* clause. The *resultClass* attribute in `sqlMapCars.xml` is `java.lang.String`, indicating that each value returned by the *select* statement will be a `String`. Since the query is a `queryForList`, a `List` of all the car names will be returned by iBATIS, and printed on the screen by your program.

2.5.6 Copy your iBATIS Jar Files

You need to copy the following files: `ibatis-sqlmap-2.jar`, `ibatis-common-2.jar`, `postgresql-8.1-404.jdbc3.jar` to the `ibatisapp` directory. If you are using some other database than PostgreSQL, copy the JDBC driver jar file for that database. You do not strictly need to copy the files: just include them in the classpath. But we are giving these steps to make the process easily repeatable.

2.5.7 Run your iBATIS Application

To compile your Java code, type this at the command line:

```
javac -classpath ibatis-common-2.jar;ibatis-sqlmap-2.jar IbatisExerciser.java
```

To run your program, type this at the command line:

```
java -cp .;ibatis-common-2.jar;ibatis-sqlmap-2.jar;postgresql-8.1-404.jdbc3.jar IbatisExerciser
```

Type the name of your other JDBC driver jar file, if it is not PostgreSQL, or if you have a more recent version

of PostgreSQL. We're assuming that you have the Java 2 SDK (1.4) installed, and that the bin directory within your j2sdk folder is part of the path variable in your environment setup. If it is not, you can give the full path name to run javac and java, for example, `\j2sdk1.4.2_05\bin\javac` and `\j2sdk1.4.2_05\bin\java`.

That's it! In the remainder of this book, we will study most of the options you can exercise as you interact with your database using iBATIS.

How to Write SQL Queries Using iBATIS

3.1 Introduction

This chapter teaches you how to write SQL queries using iBATIS. The SQL queries (insert, update, delete, and select statements) get written in one or more XML files called SQL-Map files. They use Parameter Maps and Result Maps to map SQL variables values to Java variable values. In the next sections, we understand Parameter and Result Maps, and how they help us write SQL queries.

3.2 The root element: <sqlMap>

<sqlMap> is the root element. The other elements should come in between <sqlMap> and </sqlMap>. Optionally, you can supply a namespace attribute:

```
<sqlMap namespace="Cars">
```

Then, if namespaces are enabled in your SQL Map Configuration file, you can address the contents of the SQL Map file by namespace.

3.3 The <typeAlias> element

Before diving into the details of the SQL-Maps file, we want to discuss one simple and important element, the <typeAlias> element. It looks like this:

```
<typeAlias alias="car" type="com.myrentalagency.CarData"/>
```

To keep our examples simple, we are not dividing our Java files into packages, so for us, it would say:

```
<typeAlias alias="car" type="CarData"/>
```

The result of this is, if there are places in your XML file where you need to put the text: com.myrentalagency.CarData, you can instead put "car", and thanks to the <typeAlias> element, iBATIS will interpret it correctly. Since "CarData" is short enough, we will not in general use the above alias.

3.4 Parameter Maps, Result Maps, and SQL Queries

Parameter Maps and Result Maps are the central ideas of iBATIS. A Parameter Map defines how the variables in one or more Java objects are to be set as parameters in a SQL query. A Result Map defines how the resultant data returned by a SQL query is put into the variable values of one or more Java objects.

3.4.1 Explicit Parameter Maps

```
<!ELEMENT parameterMap (parameter+)>
<!ATTLIST parameterMap
  id CDATA #REQUIRED
  class CDATA #REQUIRED
>
```

The element `<parameterMap>` contains one or more `<parameter>` elements. The `id` attribute identifies this `parameterMap` uniquely. The `class` attribute is either the fully qualified name of a Java class that this `parameterMap` is mapping, or the alias of such a class (an alias is a short-hand name for a class – we will discuss aliases shortly).

```
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter
  property CDATA #REQUIRED
  javaType CDATA #IMPLIED
  jdbcType CDATA #IMPLIED
  nullValue CDATA #IMPLIED
  mode (IN | OUT | INOUT) #IMPLIED
>
```

Example:

```
<parameterMap id="carParamMap" class="java.util.Map">
  <parameter property="carName" javaType="java.lang.String" jdbcType="VARCHAR">
    <parameter property="fromCountry" javaType="java.lang.String"
      jdbcType="VARCHAR">

```

The Java class that the Parameter Map is mapping from has to be a `JavaBean`, or a `Java Map`. The `property` attribute above is the name of the property of the `JavaBean` or `Map`. The `javaType` is the type of the property, for example `java.lang.Integer`. This has to be given as a fully qualified name or an alias. (For `java.lang.Integer`, the alias is `int`; for `java.lang.String`, the alias is `string`. A table of possible Java and JDBC types, and built in aliases, is given later in this chapter). The `jdbcType` is the type of the JDBC variable in the SQL query, to which the Java variable is being mapped. `nullValue` means, if the value of the variable being passed to the SQL query is null, what should we automatically substitute it with? `mode` specifies the directions in which the argument is carrying data. It will in most cases take the default value of `IN`, except when dealing with stored procedures.

You will notice that the `property` attribute is the only required one. There is a complicated set of rules for when the other attributes are best used. Our simple advice is: whenever you use a Parameter Map, specify the `property`, `javaType`, and `jdbcType` attributes as a rule, and the others if required.

The Parameter Map defined by the `parameterMap` element is called an explicit Parameter Map. There is another type of Parameter Map called an inline Parameter Map that achieves the same effect, but takes less space to write. We will go over these later.

3.4.2 The Query Elements `<update>`, `<insert>`, and `<delete>`

The DTD fragments for the query elements are complicated. We will start with examples directly. The key to using iBATIS well is to understand the Parameter and Result Mappings. We will also in this section look at “inline” Parameter Maps and “implicit” Result Maps.

Examples:

```
<insert id="insertCar" parameterMap="carParamMap">
  insert into CARS ( NAME, COUNTRY ) values (?,?);
</insert>
```

The parameterMap attribute is as defined in the Parameter Maps section above. The question marks are filled in the same sequence as the order of the parameters is defined in the parameterMap element. So, it is necessary to make sure that the order matches, else you will get incorrect results. The semi-colon at the end is necessary. The structure of update and delete is exactly the same as the default structure for <insert>. For the <update> query, we will need to rewrite the parameter map for cars:

```
<parameterMap id="carParamMap2" class="java.util.Map">
  <parameter property="fromCountry" javaType="java.lang.String"
             jdbcType="VARCHAR">
  <parameter property="carName" javaType="java.lang.String" jdbcType="VARCHAR">
</parameterMap>
```

The country name comes before the carName now, since the question mark for COUNTRY is before the question mark for NAME.

```
<update id="updateCars" parameterMap="carParamMap2">
  update CARS set COUNTRY = ? WHERE NAME=?;
</update>
```

For this delete instruction, we use the parameter map from Section 3.4.1.

```
<delete id="deleteCar" parameterMap="carParamMap">
  delete from CARS WHERE NAME=? AND COUNTRY=?;
</delete>
```

This shows you the use of explicit parameter maps with the insert, update, and delete statements. Now we will cover inline parameter maps.

3.4.3 Inline Parameter Maps

Inline parameter maps are a shorthand for the explicit parameter maps, to be used when the parameter mapping is relatively simple. With inline parameter maps, you do not declare a separate <parameterMap> element. You declare the map directly in the SQL element. For example:

```
<insert id="insertCar" parameterClass="CarData">
  insert into CARS ( NAME, COUNTRY ) values (#carName#, #fromCountry#);
</insert>
```

This assumes that you have a JavaBean named CarData, with a property named carName, and a property named fromCountry. If your bean is part of a package, you have to give the fully qualified package name. These get mapped via the inline parameter map shown above to the columns NAME and COUNTRY in the table CARS.

You can add more complexity and functionality into inline Parameter Maps, but the version shown above is the best form of usage, for inline Parameter Maps.

For the update and insert statements, the above inline parameter map would look like this:

```
<update id="updateCars" parameterClass="CarData">
```

```

    update CARS set NAME = #carName# where COUNTRY = #fromCountry#;
</update>

<delete id="deleteItem" parameterClass="CarData">
    delete from CARS where NAME=#carName# and COUNTRY=#fromCountry#;
</delete>

```

Java Primitive Wrappers as Parameters

With an inline parameter map, you can specify a Java wrapper around a primitive type, such as `java.lang.Integer`, `java.lang.String`, etc., directly as a parameter. An example follows:

```

<delete id="deleteCar" parameterClass="java.lang.String">
    delete from CARS WHERE NAME=#value#
</delete>

```

You can specify only one parameter for the `parameterClass`. In that case, within the query, the character string between the two `#` signs will be assumed to be the thing that should be replaced by the value coming in through the `parameterClass` attribute.

If you want to specify multiple primitive parameters with an inline Parameter Map, the solution would be to use the `Map` type as the parameter class (this is the `java.util.Map` type – its use is distinct from the use of the word "Map" in "Parameter Map").

The `java.util.Map` as the Parameter Class Object

This would work as follows:

```

<insert id="insertCar" parameterClass="java.util.HashMap">
    insert into CARS ( NAME, COUNTRY ) values (#carName#, #fromCountry#);
</insert>

```

`carName` and `fromCountry` have to be properties in your map, and the values of these properties will go into the corresponding placeholders above.

This is the best way in which to use Parameter Maps. It will allow you to deal with a large proportion of the cases where you need to write a parameter map (assuming you're passing more than a single primitive parameter). It is simpler than using explicit parameter maps, because you can access the values by name instead of by question marks. If you have to use explicit parameter maps, you need to keep track of the order in which the variables must go into the question marks.

3.4.4 More about the `<insert>` element

The examples above should have explained how the `<insert>`, `<delete>`, and `<update>` elements should be used. There is an additional feature along with the `<insert>` element. It is the `<selectKey>` element. Its usage is as follows:

```

<insert id="insertItem" parameterClass="com.mycompany.Item">
    <selectKey resultClass="int">
        SELECT ITEMKEYS.NEXTVAL AS KEYVALUE FROM DUAL
    </selectKey>
    insert into ITEMS ( ITEM_ID, ITEM_NAME, ITEM_COST )
    values (#keyvalue#, #itemName#, #itemCost#)
</insert>

```

We have used an "ITEM" class example, since our CARS database does not have a numerical key. The `<selectKey>` element gives you the value of an auto-generated key. The above example is for the Oracle database. The iBATIS documentation has a similar example for the SQL Server database.

3.4.5 Explicit Result Maps

```
<!ELEMENT resultMap (result+)>
  <!ATTLIST resultMap
    id CDATA #REQUIRED
    class CDATA #REQUIRED
    extends CDATA #IMPLIED
  >
```

Similar to the `<parameterMap>` element, the `<resultMap>` element consists of one or more `<result>` elements. The `<resultMap>` element is identified by an `id`, and specifies a Java class to which the query result is to be mapped. This Java class has to be a Java Map or a JavaBean.

```
<!ELEMENT result EMPTY>
  <!ATTLIST result
    property CDATA #REQUIRED
    javaType CDATA #IMPLIED
    column CDATA #IMPLIED
    jdbcType CDATA #IMPLIED
    nullValue CDATA #IMPLIED
  >
```

The meanings of the attributes of the result element are virtually identical to those of the parameter element described above. We will go through them briefly.

property is the property name in a JavaBean or Java Map

javaType is the type of the Java variable to which the returned column value is being mapped

column is the name of the SQL column, in the result returned by the query

jdbcType is the type of the SQL column, as a JDBC variable

nullValue is the value to set the Java variable, if the SQL value is NULL.

Example

```
<resultMap id="carsResult" class="CarData">
  <result property="carName" javaType="string" column="ITEM_NAME"
          jdbcType="VARCHAR">
  <result property="fromCountry" javaType="int" column="COUNTRY"
          jdbcType="VARCHAR"/>
</resultMap>
```

With the explanations given above, this example should be self-explanatory.

3.4.6 The `<select>` element

The `<select>` element (has nothing to do with the `<selectKey>` element discussed previously) houses select queries to be used to fetch information from the database.

It's DTD is as follows:

```
<!ELEMENT select ( ... child elements are 'dynamic' - discussed in Chapter 9 ... )>
```

```

<!ATTLIST select
  id CDATA #REQUIRED
  parameterMap CDATA #IMPLIED
  parameterClass CDATA #IMPLIED
  resultMap CDATA #IMPLIED
  resultClass CDATA #IMPLIED
  cacheModel CDATA #IMPLIED
  fetchSize CDATA #IMPLIED
>

```

We have omitted a couple of specialized attributes, which are mentioned in Chapter 10. The main difference between the `<select>` element and the `<insert>`, `<update>`, and `<delete>` elements is, the `<select>` element also has the `resultMap` and `resultClass` attributes. The other elements don't. The `resultMap` attribute is used with explicit result maps. The `resultClass` attribute is used with implicit result maps. These are similar to inline parameter maps. We will discuss them shortly. Chapter 4 is devoted to the `cacheModel` attribute.

A `<select>` element with an explicit result map and an inline parameter map would like this:

```

<select id= "getCarInfo" parameterClass="string" resultMap="carsResult">
  select NAME, COUNTRY from CARS where COUNTRY = #value#;
</select>

```

The `#value#` is filled by the integer from `parameterClass`. The selected rows populate the object defined in the `selectItems` `<resultMap>` described in the previous section, namely a `CarData` object.

3.4.7 Implicit Result Maps

Implicit result maps are similar to inline parameter maps. They allow us to map SQL variables to Java variables, without an explicit mapping construct, namely the explicit result map. They use the `resultClass` attribute instead.

This is what an implicit result map looks like:

```

<select id= getCarInfoImplicit parameterClass="string" resultClass="CarData">
  select NAME as carName, COUNTRY as fromCountry from CARS where
  COUNTRY = #value# ;
</select>

```

Here, the `parameterMap` is inline, and the result map is implicit.

Since `carName` and `fromCountry` are the names of properties in `CarData`, they will automatically get populated, if this form of the select query is used.

There remain two XML elements for SQL that we have not discussed yet: `<statement>` and `<procedure>`. `<statement>` is a general purpose element that can be used in place of any of the others. You are advised not to use the `<statement>` element, and instead use the specific element that describes what you are trying to do. That will be better programming practice, since your colleagues will be able to see right away what you intend your query to do. The other is the `<procedure>` element which lets you call Stored Procedures. We discuss this in Chapter 6.

How to Use iBATIS's Java Methods

4.1 Introduction and Shortcuts

In this chapter, we show you the Java methods that are used with SQL-Maps. There are only a handful of them, so this, the API chapter, is also a short one.

By way of shortcuts to using iBATIS, the following data access methods are good to focus on, to start with:

```
Object insert(String id, Object parameterObject) throws SQLException
```

```
int update(String id, Object parameterObject) throws SQLException
```

```
int delete(String id, Object parameterObject) throws SQLException
```

These three methods are used with the <insert>, <update>, and <delete> elements in the SQL Map respectively, covered in the previous chapter.

If you want to fire a select query, you will almost certainly expect to get back a list of rows. Hence, the following methods are good to know:

```
List queryForList(String id, Object parameterObject) throws SQLException
```

```
PaginatedList queryForPaginatedList(String id, Object parameterObject,  
                                     int pageSize) throws SQLException
```

For both these methods, the 'id' argument will be the id of a <select> element. The returned List will contain objects of the class defined in the Result Map. The PaginatedList class lets you access the List (which may have thousands of entries), one page at a time. For all of the methods, parameterObject is the object containing the parameters to the query, as defined in the Parameter Map.

4.2 The Java API for iBATIS

The following sets of methods constitute the Java API for iBATIS. There are some other methods that can be called, that are not listed here, but these methods give you all the functionality you will need.

4.2.1 Methods for Reading the Configuration File

The methods are:

```
Reader reader = Resources.getResourceAsReader(String configFile)
SqlMapClient sqlMapClient = SqlMapClientBuilder.buildSqlMap( reader );
```

You have seen these methods at work in Section 2.4.5.

4.2.2 Methods for Data Access

These are the Java methods you call in order to exercise the code that you put into the SQL-Map XML file. They are declared in a Java interface called `SqlMapExecutor`, in the package `com.ibatis.sqlmap.client`. `SqlMapExecutor` is inherited by the `SqlMapClient` interface. You can invoke these methods using the `SqlMapClient` instance you get from iBATIS, using the methods shown in Section 4.2.1.

In all the statements below, the Java object `parameterObject` will in almost every case contain the variables named in the SQL-Map XML file as parameters for the given query, and their values. The value of the String `id` has to be the same as the id of the corresponding statement in the SQL-Map file.

Object insert(String id, Object parameterObject) throws SQLException

Inserts a row, and provides the option of returning the primary key of the newly inserted row.

int update(String id, Object parameterObject) throws SQLException

Does an update, and returns the number of rows updated

int delete(String id, Object parameterObject) throws SQLException

Does a delete, and returns the number of rows deleted

Object queryForObject(String id, Object parameterObject) throws SQLException

Runs a SELECT statement in the SQL-Map file. Returns the results of the query in a Java variable of type `Object`, which has to be cast into the appropriate Java variable defined in the result mapping in the SQL-Map XML file.

Object queryForObject(String id, Object parameterObject, Object resultObject) throws SQLException

This method is different from the previous one in that the `resultObject` is passed as a parameter, and gets populated with the result of the query, which is also the return value.

List queryForList(String id, Object parameterObject) throws SQLException

A typical SELECT query returns not one, but many rows. The values of the columns for each of these rows is mapped to a Java object. Hence, when many rows are returned, a collection of Java objects will be required to hold the result. iBATIS uses a `List` type to hold the objects populated by each row.

List queryForList(String id, Object parameterObject, int skip, int max) throws SQLException

This is a variation on the `queryForList` statement that skips the first few rows defined by the number in the parameter 'skip', and returns 'max' number of rows.

PaginatedList queryForPaginatedList(String id, Object parameterObject, int pageSize) throws SQLException

The `PaginatedList` returned by `queryForPaginatedList` is a `List` which holds a number of objects defined by the parameter `pageSize` in each 'page'. `PaginatedList` has a number of functions to let you navigate pages, such as `nextPage()` (moves to the next page), and `isLastPage()` (tells you if the current page is the last page), and

a number of others.

<< TO DO: Check once again how these functions are to be used >>

```
/**
 * Executes a mapped SQL SELECT statement that returns data to populate
 * a number of result objects that will be keyed into a Map.
 * <p/>
 * The parameter object is generally used to supply the input
 * data for the WHERE clause parameter(s) of the SELECT statement.
 *
 * @param id          The name of the statement to execute.
 * @param parameterObject The parameter object (e.g. JavaBean, Map, XML etc.).
 * @param keyProp     The property to be used as the key in the Map.
 * @return A Map keyed by keyProp with values being the result object instance.
 * @throws java.sql.SQLException If an error occurs.
 */
Map queryForMap(String id, Object parameterObject, String keyProp) throws SQLException;
```

```
/**
 * Executes a mapped SQL SELECT statement that returns data to populate
 * a number of result objects from which one property will be keyed into a Map.
 * <p/>
 * The parameter object is generally used to supply the input
 * data for the WHERE clause parameter(s) of the SELECT statement.
 *
 * @param id          The name of the statement to execute.
 * @param parameterObject The parameter object (e.g. JavaBean, Map, XML etc.).
 * @param keyProp     The property to be used as the key in the Map.
 * @param valueProp   The property to be used as the value in the Map.
 * @return A Map keyed by keyProp with values of valueProp.
 * @throws java.sql.SQLException If an error occurs.
 */
Map queryForMap(String id, Object parameterObject, String keyProp, String valueProp) throws SQLException;
```

4.2.3 Calling Stored Procedures

<TODO: Look into this>

4.2.4 Batch Updates for Efficiency

The following methods are used for batch updates:

```
void startBatch() throws SQLException
int executeBatch() throws SQLException
```

They are accessed via the SqlMapClient class. They are used as shown below:

```
try {
    sqlMapClient.startBatch();
    sqlMapClient.insert( id1, parameterObject1 );
```

```

    sqlMapClient.update( id2, parameterObject2 );
    sqlMapClient.update( id3, parameterObject3 );
    sqlMapClient.delete( id4, parameterObject4 );
    sqlMapClient.update( id5, parameterObject5 );
    sqlMapClient.executeBatch();
} catch (SQLException sqle) {
    sqle.printStackTrace();
}

```

If you have many SQL commands that are not selects (don't return values), then you can use `startBatch` and `executeBatch` to send them to the database at once. The main advantage is that it cuts down the time lag due to sending each statement in sequence. `startBatch` starts collecting the statements, and `executeBatch` sends them to the database, and also empties the collection of statements started by `startBatch`.

4.2.5 Transactions

A transaction is when a sequence of queries are run either together in sequence and completed, or, if even one of them fails for some reason, the entire set of queries is “rolled back” so that the database is in the state it was in before the first query ran. Note that this is different from the “batch updates” discussed above. In a batch update, if one of the queries fails, the ones before it are not affected.

Transactions can be of two varieties: those that span multiple databases, and those that are restricted to a single database. Here, we are going to discuss the transactions that are restricted to a single database. It is very simple to run these transactions in iBATIS. The methods are as follows.

```
public void startTransaction() throws SQLException
```

Starts the transaction

```
public void commitTransaction() throws SQLException
```

Commits the transaction

```
public void endTransaction() throws SQLException
```

Should be called regardless of whether the transaction succeeds or fails.

These methods are accessed via the `SqlMapClient` interface. The following code illustrates how the transactions can work.

```

try {
    sqlMapClient.startTransaction();
    sqlMapClient.insert( id1, parameterObject1 );
    sqlMapClient.update( id2, parameterObject2 );
    sqlMapClient.update( id3, parameterObject3 );
    sqlMapClient.delete( id4, parameterObject4 );
    sqlMapClient.update( id5, parameterObject5 );
    sqlMapClient.commitTransaction();
} catch (SQLException sqle) {
    sqle.printStackTrace();
} finally {
    sqlMapClient.endTransaction();
}

```

How to Configure iBATIS

5.1 Introduction and Shortcuts

A single XML configuration file configures iBATIS for your application. You can name it as you please. We discussed it earlier in Section 2.4.4.

The best shortcut for this, as with most other configuration files, is to copy and modify the file rather than trying to write it from scratch. The purpose of this chapter is to give you some insight into what the settings mean. The configuration file from JPetStore is a good one. Section 2.4.4 tells you where you can get it from. You can copy and paste this file, and set the values in the database.properties file (which this one uses) according to your database's JDBC parameters. Then you simply need to create `<sqlMap>` elements for each of your SQL-Map files, and you're done.

If you're ready to read this chapter now, our approach will be to break up the file into DTD fragments, and walk through each one of them.

5.2 Configuration File Definition: The `<sqlMapConfig>` element

The DTD fragment for the root element is:

```
<!ELEMENT sqlMapConfig (properties?, settings?, typeAlias*, typeHandler*,
transactionManager?, sqlMap+)+>
  <!ATTLIST sqlMapConfig
    xmlns:fo CDATA #IMPLIED
  >
```

The root element can contain the elements `<properties>`, `<settings>`, `<typeAlias>`, `<typeHandler>`, `<transactionManager>`, and `<sqlMap>`. We will describe these in turn.

5.3 The simpler elements: `<typeAlias>`, `<properties>`, `<sqlMap>`

The DTD fragments for the simpler elements are as follows.

`<typeAlias>` element:

```
<!ELEMENT typeAlias EMPTY>
  <!ATTLIST typeAlias
    alias CDATA #REQUIRED
    type CDATA #REQUIRED
```

>

The `<typeAlias>` element is identical to the SQL-Map file `<typeAlias>` element. It allows you to associate the fully qualified name of a Java class, to a shorter name or alias. You can have none, or as many `<typeAlias>` elements as you want, in the file.

<properties> element:

```
<!ELEMENT properties EMPTY>
  <!ATTLIST properties
    resource CDATA #IMPLIED
    url CDATA #IMPLIED
  >
```

The `properties` element allows you to identify a property file as a resource from which to pick up key-value pairs. The property names listed in that file are picked up by the SQL-Map Configuration file, and their values can be referred to by using the expression `#{property_name}`, where `property_name` is the name of the property in question. You may have at most one property file.

Example:

```
<properties resource="application.properties">
```

<sqlMap> element:

```
<!ELEMENT sqlMap EMPTY>
<!ATTLIST sqlMap
resource CDATA #IMPLIED
url CDATA #IMPLIED
>
```

The `sqlMap` element lists each SQL-Map file (containing Parameter Maps, Result Maps, and queries, discussed in the previous chapter), that you have created. If you do not list the SQL-Map file here, you will not be able to call its queries from your Java code. You must have at least one `sqlMap` entry, and you may have as many as you want.

Example:

```
<sqlMap resource="sqlMap.xml">
```

5.4 The <settings> element

The DTD fragment for the `<settings>` element is as follows:

```
<!ELEMENT settings EMPTY>
  <!ATTLIST settings
    lazyLoadingEnabled (true | false) #IMPLIED
    cacheModelsEnabled (true | false) #IMPLIED
    enhancementEnabled (true | false) #IMPLIED
    useStatementNamespaces (true | false) #IMPLIED
    maxSessions CDATA #IMPLIED
    maxTransactions CDATA #IMPLIED
```

```
maxRequests CDATA #IMPLIED
>
```

It has no children, a bunch of true/false attributes, and a few “max” values, as you see above. All the attributes have defaults. We describe the attributes here:

lazyLoadingEnabled

If a query generates a very large amount of data, say thousands of rows, you want to be able fetch that data from the database as and when it is required, instead of right away. This reduces the immediate return time of the query. This feature is called 'lazy loading', and putting the setting as 'true' enables lazyLoading. The default setting is 'true'.

cacheModelsEnabled

iBATIS has different algorithms or 'models' for caching data. Setting this to 'true' enables those cache models. The default is 'true'

enhancementEnabled

Bytecode enhancement is a technique for optimizing a program. Setting this to 'true' enables it. The default is 'false'.

useStatementNamespaces

Setting useStatementNamespaces to 'true' requires you to refer to SQL-Map statements via the namespace of the XML file that they belong to.

maxSessions

maxSessions has to be set to a number that indicates the maximum number of user sessions that can be supported at any time.

maxTransactions

maxTransactions defines the maximum number of threads that can enter the startTransaction() method at any one time.

maxRequests

maxRequests defines the maximum number of user requests that can be supported at any time.

An example of the settings element follows:

```
<settings
  lazyLoadingEnabled="true"
  cacheModelsEnabled="true"
  enhancementEnabled="false"
  useStatementNamespaces="false"
  maxSessions="64"
  maxTransactions="28"
  maxRequests="512"
/>
```

5.5 The <transactionManager> element

The DTD fragment for the transaction manager element is as follows:

```
<!ELEMENT transactionManager (property*,dataSource)>
  <!ATTLIST transactionManager
    type CDATA #REQUIRED
    commitRequired (true | false) #IMPLIED
  >

<!ELEMENT dataSource (property*)>
  <!ATTLIST dataSource
    type CDATA #REQUIRED
  >

<!ELEMENT property EMPTY>
  <!ATTLIST property
    name CDATA #REQUIRED
    value CDATA #REQUIRED
  >
```

The <transactionManager> element must have exactly one <dataSource> element, and any number of <property> elements (different from the <properties> element described above). The property element is just a name – value pair. The <dataSource> element contains a number of <property> elements that help describe the connection to the database.

The <transactionManager> element from the JPetStore example is as follows:

```
<transactionManager type="JDBC">
  <dataSource type="SIMPLE">
    <property value="{driver}" name="JDBC.Driver"/>
    <property value="{url}" name="JDBC.ConnectionURL"/>
    <property value="{username}" name="JDBC.Username"/>
    <property value="{password}" name="JDBC.Password"/>
    <property value="15" name="Pool.MaximumActiveConnections"/>
    <property value="15" name="Pool.MaximumIdleConnections"/>
    <property value="1000" name="Pool.MaximumWait"/>
  </dataSource>
</transactionManager>
```

The first four values should be self-explanatory. These are the values you set in the database.properties file in Chapter 2. They are picked up from there.

There are other data source types, and other properties that can be defined for the transaction manager. For getting more detail on the <transactionManager> element, you should look at the iBATIS documentation, by Clinton Begin.

<TODO: Add a bit more detail for the transaction manager>

How to Speed Up Queries by Caching

6.1 Introduction and Shortcuts

If so instructed, iBATIS keeps the Java object generated by a given query in the computer's memory. If an identical query is sent to iBATIS later, iBATIS simply returns the existing Java object, instead of running the query again.. The effect is that the result of the query is received almost immediately, no matter how complex the query.

A good shortcut to the problem of choosing the correct caching strategy is: go with 'LRU'. 'Least Recently Used' is an excellent algorithm for most of the situations you will face. It is described in more detail in Section The next section gives the syntax for defining the cache.

6.2 DTD Fragment for Caching

The following text shows the part of the SQL-Map Document Type Definition relevant to Caching.

```
<!ELEMENT cacheModel (flushInterval?, flushOnExecute*, property*)+>
  <!ATTLIST cacheModel
    id CDATA #REQUIRED
    type CDATA #REQUIRED
    readOnly (true | false) #IMPLIED
    serialize (true | false) #IMPLIED
  >

<!ELEMENT flushInterval EMPTY>
  <!ATTLIST flushInterval
    milliseconds CDATA #IMPLIED
    seconds CDATA #IMPLIED
    minutes CDATA #IMPLIED
    hours CDATA #IMPLIED
  >

<!ELEMENT flushOnExecute EMPTY>
  <!ATTLIST flushOnExecute
    statement CDATA #REQUIRED
  >

<!ELEMENT property EMPTY>
  <!ATTLIST property
    name CDATA #REQUIRED
    value CDATA #REQUIRED
  >
```

Syntax Reminder

1. Items in brackets after an element are the child elements of that element. '?' means 0 or 1; '*' means 0 or many; '+' means 1 or many.
2. 'EMPTY' after an element name means it has no child elements.
3. Items in the rows following 'ATTLIST' are the attributes of the element.

Examples

The different caching strategies available to iBATIS are presented as examples in use. You copy these examples to your SQL-Map file, and change the precise settings as you wish.

Example 1

```
<cacheModel type="LRU">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertItem"/>
  <flushOnExecute statement="deleteItem"/>
  <property name="size" value="500"/>
</cacheModel>
```

The above example creates a LRU (Least Recently Used) cache. Every 24 hours, the cache is emptied out (flushed). If the statements with the names *insertItem* or *deleteItem* are executed, the cache will be flushed. This is because these statements would change the underlying data from which the cache is storing information for quick retrieval. The size of the cache is 500, meaning that it can store 500 objects. (Internally, the cached objects are kept in a HashMap, so the 500 means that you can store 500 key-value pairs in the HashMap). The only property entry you can use with the cacheModel element of type LRU is size.

The attributes *readOnly* and *serialize* in the DTD entry for the cacheModel element are left to their default (implied) values, so they are not mentioned here. The default values are **(put here)**

The pattern given in this first example is a good one to use if you have no reason to choose any other. LRU is good caching algorithm. It means that the when the cache becomes full, the cached item that was used the longest time ago will be the one to be ejected from the cache.

Example 2

```
<cacheModel type="FIFO">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertItem"/>
  <flushOnExecute statement="deleteItem"/>
  <property name="size" value="500"/>
</cacheModel>
```

FIFO stands for 'First In First Out'. It means that the first entry that was introduced into the cache will be the first one to be kicked out, should the cache become full. We believe that LRU is superior to FIFO in most cases. You are more likely to organize your cache based on the usage of your users, rather than an arbitrary queue.

Example 3

MEMORY cache (**TODO: to read up**)

Example 4

```
<cacheModel type="OSCACHE">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertItem"/>
  <flushOnExecute statement="deleteItem"/>
</cacheModel>
```

OSCACHE is a separate product devoted entirely to caching. It is open source. You can get it and read about it at <http://www.opensymphony.org/oscache>. To use oscache, you will need to put the oscache.jar file along with your ibatis jar files. To configure it, you set the required settings in the oscache.properties file.

How to Use iBATIS Data Access Objects

7.1 Introduction and Shortcuts

iBATIS SQL-Maps and the iBATIS Data Access Objects (DAO) framework are different and separate products. We've been discussing iBATIS SQL-Maps in all of the preceding chapters. This chapter teaches you how to use the iBATIS Data Access Objects framework.

The purpose of iBATIS Data Access Objects is to serve as an "abstraction layer" behind which SQL-Maps (or possibly other third-party frameworks) work. This translates into two things:

1. If you use the iBATIS DAO framework with iBATIS SQL-Maps, you find yourself pushed to create a good design at your database end. The methods that call iBATIS SQL mapping functionality are supposed to be declared in separate Java interfaces instead of being scattered all over your code. Your application's central logic need only know about the functions declared in these interfaces, and what the argument values and return values mean. That's a good, clean design.
2. You can use other data or object relational mapping software behind the iBATIS DAO framework. Hibernate, direct JDBC, and JTA are supported, with a separate category for other external tools.

We will focus on the benefits mentioned in the first point above, in this chapter, assuming that you want to use the iBATIS DAO framework with iBATIS SQL-Maps.

By way of shortcuts, you should understand that you don't need to understand much DAO theory in order to get it running – and to start benefiting from it – in practice. So, you should run the simple DAO example given in this chapter. If that is enough to get you started on your own, that's great. If not, take a look at the DAO classes and the DAO xml file in the JPetStore example. The directory in JPetStore that has the DAO files is as follows. The explanations given in Section 7.3 are there to give you a deeper understanding of the rationale for the DAO mechanism.

```
jpetstore\src\com\ibatis\jpetstore\persistence
```

7.2 A Working Example of iBATIS DAO

The purpose of this chapter is to give you a working example of how a very simple application looks in practice, when using iBATIS DAO with iBATIS SQL-Maps.

Our goal is to recreate the very simple 'Hello Cars' application of Chapter 2 over here, using Data Access Objects.

7.2.1 List of Additional Program Files

We will need the following additional files.

dao.xml	Configuration file for iBatis DAO
CarDataDao.java	File with a class that is the Java interface for the DAO
CarDataDaoImpl.java	File implementing the interface defined in class CarDataDao
IbatisDaoExerciser.java	Gets the data from CarDataDao and shows it at the "front end" (the command line in this case)
DaoConfig.java	Used to set up the DaoManager instance. Copy and modify from JPetStore example.

And we will need to modify the file SqlMapCars.xml that we created in chapter 2.

All of these files can be put directly in the directory we introduced in Chapter 2: ibatisapp. For bigger applications, you would have a separate directory structure for your DAO files. Also, we're assuming that the files we introduced in Chapter 2 are still in your ibatisapp directory. If you have not put them there, or worked that example through, you should do that first. This example builds on the simple one we introduced in Chapter 2.

7.2.2 Content of Files

The content of the files is as follows.

file: dao.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE daoConfig
  PUBLIC "-//ibatis.com//DTD DAO Configuration 2.0//EN"
  "http://www.ibatis.com/dtd/dao-2.dtd">

<daoConfig>
  <context>
    <transactionManager type="SQLMAP">
      <property name="SqlMapConfigResource" value="sql-map-config.xml"/>
    </transactionManager>

    <dao interface="CarDataDao" implementation="CarDataImplDao"/>
  </context>
</daoConfig>
```

file: CarDataDao.java

```
import java.util.List;

public interface CarDataDao {
  public List getAllCarsList();
  public List getCountriesList();
}
```

file: CarDataDaoImpl.java

```
import com.ibatis.dao.client.DaoManager;
import com.ibatis.dao.client.template.SqlMapDaoTemplate;
```

```

import java.util.List;

public class CarDataDaoImpl extends SqlMapDaoTemplate implements CarDataDao {

    public CarDataDaoImpl(DaoManager daoManager) {
        super(daoManager);
    }

    public List getAllCarsList() {
        return queryByList( "selectAllNames", null );
    }

    public List getCountriesList() {
        return queryByList("selectCountries", null );
    }
}

```

file: IbatisDaoExerciser.java

```

import java.util.List;
import java.sql.SQLException;
import com.ibatis.dao.client.DaoManager;

class IbatisDaoExerciser {
    private DaoManager daoManager = DaoConfig.getDaomanager();
    private CarDataDao carDao;

    public IbatisDaoExerciser() {
        carDao = (CarDataDao) daoManager.getDao(CarDataDao.class);
    }

    void doIbatis() {

        List carNames = null;
        List countries = null;

        try {
            carNames = carDao.getAllCarsList();
            countries = carDao.getCountriesList();
        } catch ( SQLException sqle ) {
            sqle.printStackTrace();
        }

        // Print result to screen
        for( int i=0; i < carNames.size(); i++ ) {
            System.out.println( (String)carNames.get( i ) );
        }

        for( int j=0; j < countries.size(); j++ ) {
            System.out.println( (String)countries.get( j ) );
        }
    }

    public static void main( String args[] ) {
        IbatisExerciser exerciser = new IbatisExerciser();
        exerciser.doIbatis();
    }
}

```

```
}  
}
```

file: DaoConfig.java

Copy this file from the JPetStore example to your ibatisapp directory. In JPetStore 5, the file is in the directory \jpetstore\src\com\ibatis\jpetstore\persistence

Make the following changes:

1. Remove the package declaration, the line that says:
package com.ibatis.jpetstore.persistence;
2. Change the line
String resource = "com.ibatis/jpetstore/persistence/dao.xml";
to
String resource = "dao.xml";

file: SqlMapCars.xml

Modify file SqlMapCars.xml from chapter 2, so that it looks as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>  
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"  
"http://www.ibatis.com/dtd/sql-map-2.dtd">  
  
<sqlMap namespace="Cars">  
  <select id="selectAllNames" resultClass="java.lang.String">  
    select NAME from CARS;  
  </select>  
  <select id="selectCountries" resultClass="java.lang.String">  
    select COUNTRY from CARS;  
  </select>  
</sqlMap>
```

7.2.3 Copy Jar File for DAO

You will need to copy the additional jar file: ibatis-dao-2.jar into your ibatisapp directory.

7.2.4 Compile and Run the iBATIS DAO based program

To compile your program, go into the ibatisapp directory, and type:

To run the program, in the ibatisapp directory, type:

7.3 The Concepts of Data Access Objects

In this section, we take a look at the main concepts behind Data Access Objects. We are going to get an overview here. For more detail, look at the iBATIS Data Access Objects Developer Guide from iBATIS.

7.3.1 Context

A dao.xml file can allow a user to specify more than one context. It is defined within the <context> element

in the dao.xml file (look at the <context> element in the dao.xml file in Section 7.2.3). A context defines one or more transaction managers that the DAO can use, as well as the DAO interfaces that can be used within that context, with the given transaction managers.

7.3.2 Transaction Managers

The transaction manager is defined in the <transactionManager> element in the dao.xml file. The available transaction manager implementations that can be used with iBATIS DAO are JDBC, SQL Maps, JTA (Java Transaction Architecture), Hibernate, and any External transaction manager. In this book, we have made the simplifying assumption that you will be using iBATIS DAO with iBATIS SQL Maps. The dao.xml file for this configuration is particularly simple, as you can see above. If you want additional detail on using iBATIS DAO with other transaction managers, you should look at the iBATIS Data Access Objects Developer Guide.

7.3.1 Interfaces

The CarDataDao.java file above gives an example of an interface definition. If you look at the IbatisDaoExerciser.java file, you see the power of this system. The class that uses the DAO need not anything except the functions defined in the interface file. All the details of accessing the database are hidden from the user of the DAO. That is what allows you to plug in multiple implementations, just by changing what sits underneath this interface file. In my opinion, the power of the DAO comes not from the ability to take out the SQL Maps and plug in Hibernate or JTA, but from the cleanliness that is virtually ensured at your back end, if you use the DAO.

7.3.2 Implementations

The CarDataDaoImpl.java file gives an example of an implementation of the interface defined in the class CarDataDao. An interface has to have an implementation. This implementation has information specific to the particular transaction manager being used. You do need to exercise care in programming the implementation. If you are sending a sequence of thousands of inserts, or if you are going to be selecting thousands of records, you need to know this, and use optimizing tricks, to avoid causing unwanted performance problems.

How to Use If-Conditions and Loops within iBATIS SQL Queries

8.1 Introduction and Shortcuts

So far, you have seen how to send the values of Java variables into SQL variables, and the results of a SQL query back into Java variables.

But you have not been able to change the structure of the SQL statement itself – the content and number of clauses after the WHERE keyword, for example. This chapter will show you how to do this. In iBATIS, this feature is called Dynamic SQL. In this chapter, unlike some of the previous chapters, we will prefer to start with an explanation of the concepts before we dive into the details of how to write the code.

Dynamic SQL instructions go into the SQL-Map file as child elements of the SQL query elements: <select>, <update>, <delete>, <insert>, <procedure>, or <statement>.

There is one Dynamic SQL element for looping, called <iterate>, and the rest of the elements are for evaluating conditionals. We will meet them shortly. To understand Dynamic SQL, you need to become familiar with the concepts: 'properties', and 'values'. In the statement:

```
select * from ITEMS where ITEM_NAME = 'widget';
```

ITEM_NAME is called a property, and 'widget' is called a value. You'll find it convenient to see the column name as the 'property', and the actual value assigned to it as the 'value'.

8.2 Conditional elements

Conditional elements are like the if-then conditions of Java. As mentioned above, SQL query elements such as <update> have the Dynamic SQL elements as children. The DTD fragment for the <update> element is as follows:

```
<!ELEMENT update (#PCDATA | include | dynamic | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual |
isEqual | isGreaterThan | isGreaterEqual | isLessThan | isLessEqual |
isPropertyAvailable | isNotPropertyAvailable)*>
<!ATTLIST update
id CDATA #REQUIRED
parameterMap CDATA #IMPLIED
parameterClass CDATA #IMPLIED
>
```

Everything including and after the <dynamic> element is a Dynamic SQL element. Everything after the <iterate> element is a conditional element. The attributes of the <update> element are not part of Dynamic SQL –

they are for defining the parameter mapping.

The conditional elements fall into 3 categories:

unary: evaluate a single property

binary: compare two properties or a property and a value

other: check to see if the required parameter exists

An explanation of each follows:

8.2.1 Unary Conditional Elements

The unary conditional elements are:

isNotNull, isNotNull, isNotPropertyAvailable, isPropertyAvailable, isEmpty, isNotEmpty

The DTD fragment for isNotNull is

```
<!ELEMENT isNotNull (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual |
isEqual | isGreaterThan | isGreaterEqual | isLessThan | isLessEqual |
isPropertyAvailable | isNotPropertyAvailable)*>
  <!ATTLIST isNotNull
    prepend CDATA #IMPLIED
    open CDATA #IMPLIED
    close CDATA #IMPLIED
    property CDATA #IMPLIED
    removeFirstPrepend CDATA #IMPLIED
  >
```

Two observations:

1. As you can see from the list of child elements for <isNull>, these elements can be nested indefinitely.
2. All the unary conditional elements have a structure identical to <isNotNull>, that is, they all have the same child elements and the same attributes.

The following XML fragment shows an example of the isNotNull element in action:

```
<parameterMap id="namesDynamicParamMap" class="java.util.Map">
  <parameter property="carName" javaType="java.lang.String" jdbcType="VARCHAR"/>
  <parameter property="fromCountry" javaType="java.lang.String"
    jdbcType="VARCHAR"/>
</parameterMap>

<select id="selectNamesUnaryNotNull" resultClass="java.lang.String"
  parameterMap="namesDynamicParamMap">
  select NAME, COUNTRY from CARS
  select NAME from CARS
  <dynamic prepend="where">
    <isNotNull property="carName">
      NAME=#carName#
    </isNotNull>
    <isNotNull property="fromCountry" prepend="AND">
      COUNTRY=#fromCountry#
    </isNotNull>
  </dynamic>
</select>
```

The effect of this dynamic SQL fragment is,

- ◆ If both carName and fromCountry are NULL, the query is:
select NAME from CARS

- ◆ If carName is not NULL and fromCountry is NULL, the query is:
 select NAME from CARS where carName = #carName#
 (#carName# is the value of the String passed from Java for carName).
- ◆ If carName is NULL, and fromCountry is not NULL, the query is:
 select NAME from CARS where fromCountry = #fromCountry#
 (#fromCountry# is the value of the String passed from Java for fromCountry).

The other unary elements work in the same way, and mean the following:

<isNull>	Is true if the property is NULL, the opposite of the above example
<isPropertyAvailable>	Is true if the property exists as one of the properties of the JavaBean or Map passed in as an argument.
<isNotPropertyAvailable>	Is true if the property does not exist as one of the properties of the JavaBean or Map passed in as an argument.
<isEmpty>	Is true of the property has an empty value (like "" for a String), and is not NULL.
<isNotEmpty>	Is true of the property does not have an empty value (like "" for a String), and is not NULL.

8.2.2 Binary Conditional Elements

The binary conditional elements are:

isEqual, isNotEqual, isGreaterThan, isGreaterEqual, isLessThan, isLessEqual

The DTD fragment for isGreaterThan is:

```
<!ELEMENT isGreaterThan (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual |
isEqual | isGreaterThan | isGreaterEqual | isLessThan | isLessEqual |
isPropertyAvailable | isNotPropertyAvailable) *>
  <!ATTLIST isGreaterThan
  prepend CDATA #IMPLIED
  open CDATA #IMPLIED
  close CDATA #IMPLIED
  property CDATA #IMPLIED
  removeFirstPrepend CDATA #IMPLIED
  compareProperty CDATA #IMPLIED
  compareValue CDATA #IMPLIED
  >

<resultMap id="namesDynamicResultMap" class="java.util.Map">
  <result property="nameOfCar" column="NAME" javaType="java.lang.String"
  jdbcType="VARCHAR"/>
  <result property="nameOfCountry" column="COUNTRY" javaType="java.lang.String"
  jdbcType="VARCHAR"/>
</resultMap>

<select id="selectNamesBinaryGreaterThan" resultClass="java.lang.String"
  parameterMap="namesDynamicParamMap">
  select NAME, COUNTRY from CARS
  <dynamic prepend="where">
    <isGreaterThan property="carName" compareProperty="fromCountry">
      COUNTRY=#fromCountry#
```

```

        </isGreaterThan>
    </dynamic>
</select>

```

The other binary elements work in the same way, and mean the following:

<isGreaterEqual>	Is true if the property is greater than or equal to the value, or other property being compared
<isEqual>	Is true if the property is equal to the value, or other property being compared
<isNotEqual>	Is true if the property is not equal to the value, or other property being compared
<isLessThan>	Is true if the property is less than the value, or other property being compared
<isLessEqual>	Is true if the property is less than or equal to the value, or other property being compared

8.2.3 Other Conditional Elements

The other conditional elements are:

isParameterPresent, isNotParameterPresent

The DTD fragment for isParameterPresent is:

```

<!ELEMENT isParameterPresent (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual |
isEqual | isGreaterThan | isGreaterEqual | isLessThan | isLessEqual |
isPropertyAvailable | isNotPropertyAvailable)*>
  <!ATTLIST isParameterPresent
    prepend CDATA #IMPLIED
    open CDATA #IMPLIED
    close CDATA #IMPLIED
    removeFirstPrepend CDATA #IMPLIED
  >

```

8.3 Looping: the <iterate> element

A single element is available for looping (such as while loops in Java): the <iterate> element. Its DTD fragment is as follows:

```

<!ELEMENT iterate (#PCDATA | include | iterate | isParameterPresent |
isNotParameterPresent | isEmpty | isNotEmpty | isNotNull | isNull | isNotEqual |
isEqual | isGreaterThan | isGreaterEqual | isLessThan | isLessEqual |
isPropertyAvailable | isNotPropertyAvailable)*>
  <!ATTLIST iterate

```

```
prepend CDATA #IMPLIED
property CDATA #IMPLIED
removeFirstPrepend CDATA #IMPLIED
open CDATA #IMPLIED
close CDATA #IMPLIED
conjunction CDATA #IMPLIED
>
```

Its children are identical to those of the conditional elements, and the attributes are the same as with the unary conditional elements, except for the additional attribute: conjunction.

It works as follows:

iBATIS in Your J2EE Architecture

9.1 Introduction and Shortcuts

This chapter takes us to a fuzzier aspect of iBATIS than we have so far seen: how iBATIS should fit into our J2EE “architecture”. If you are intimidated by the idea of J2EE architecture, don't be. Richard Feynman, physicist and “amateur” computer scientist, had this to say about computer architecture:

“Don't be put off by the word 'architecture'; it's just a big word for how we arrange things, only we're arranging electronic components rather than bricks and columns”.

He was talking about the arrangement of transistors on a computer chip. In software, architecture reduces to arranging words on a computer screen, and computer files in the computer's memory.

We are going to look at two software architectures in this chapter. The first is the JPetStore 5 architecture. I'm going to invite you to run this program, play with it, and then take a look at my comments about the architecture. The second is an architecture that I've used on two J2EE projects, where iBATIS was used as the back end. These two J2EE projects were very successful. Incidentally, we used the JPetStore 4 architecture on these projects, but JPetStore 4 is no longer being distributed by iBATIS.

The best shortcut for you at this point would be to download and run the JPetStore application, then experiment with it, and try changing some of the files, to see what the effect is. We describe how to run the JPetStore 5 example in the next section.

9.2 Running the JPetStore Example

To run the JPetStore example, you do the following:

1. Install Tomcat or another servlet container or application server, and see that it is working correctly.
2. Set the JAVA_HOME environment variable on your system to point to the directory that has your Java SDK installed (at least JDK 1.4), and set the CLASSPATH variable to point to some directory or jar file (no specific one is required, since the JPetStore build utility appends to the CLASSPATH).
3. Go to the jpetstore\build directory and run the build command.
4. This will create a 'wars' directory within the build directory, containing a file called jpetstore.war.
5. Copy this file to (for Tomcat) the webapps folder in the tomcat directory.
6. Start the web server, and point your browser to <http://localhost:8080/jpetstore>. You should now be running the example.
7. You can play with various files, then build, and run the application again.
8. The database used with this example is HSQLDB, which is internally packaged with the example.

9.3 Recommended Architecture Diagram

The diagram I recommend for your J2EE architecture is given below. This is slightly different from the JPetStore 5 architecture in that the diagram below uses Singleton objects at the business logic layer. The JPetStore 5 does

not. I have found that Singleton based business objects work very well for web applications, since you can never store state information in them. In my experience. Storage of state information is best done explicitly, at the web layer using the provisions that the Servlet API provides for this purpose.

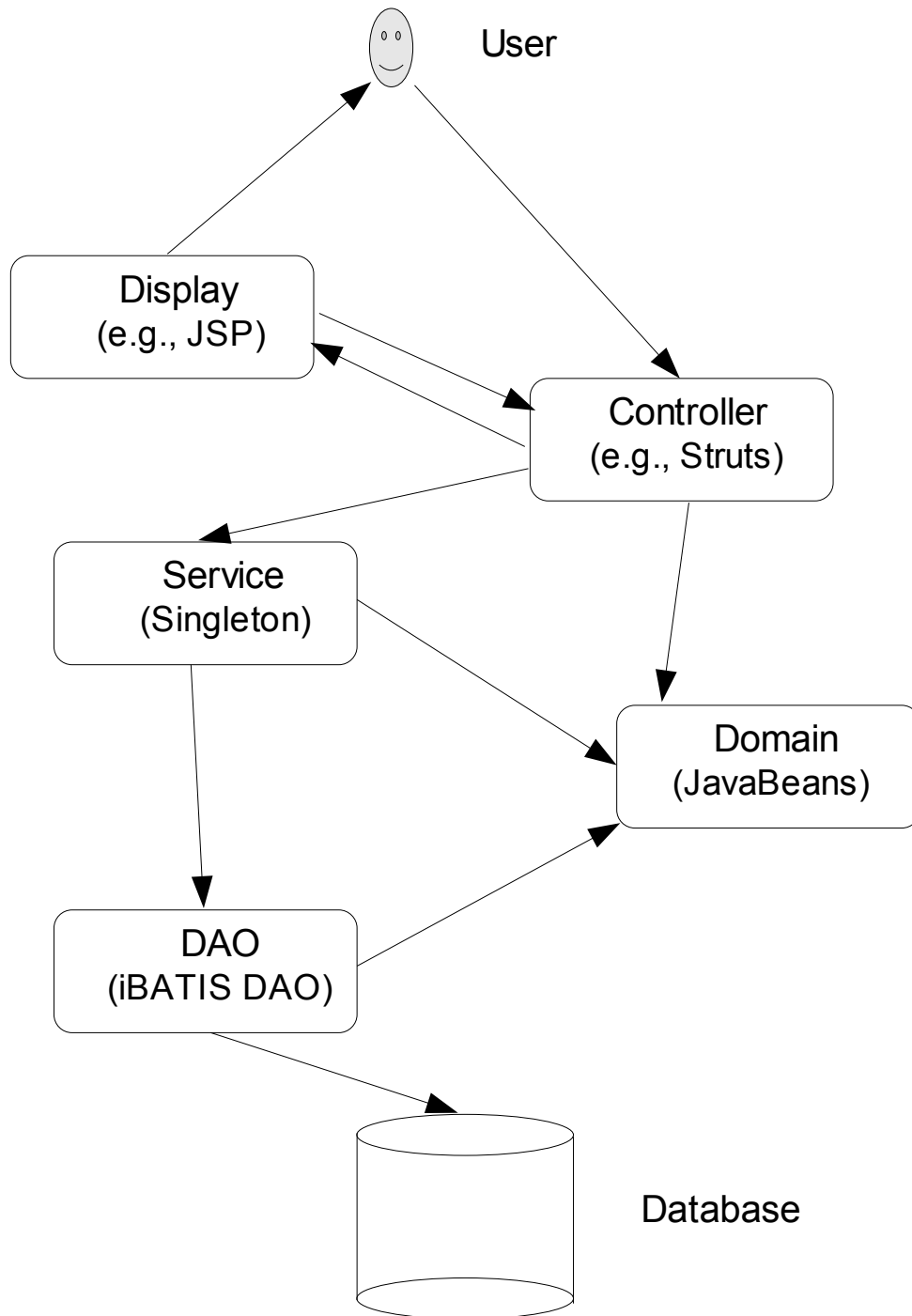


Figure 1: J2EE Architecture

Perhaps it looks complicated, but it is not. The user sends his or her request to the Controller. The Controller may package the request information into one of the Domain objects (which are just JavaBeans), and sends the request on to a Singleton Service object. The value of the Singleton has been explained earlier. The Service object does some processing on the request, and sends it on to the DAO (Data Access Objects), again possibly

packaging the information into a Domain object. What we are calling a Domain object here (to be consistent with JPetStore nomenclature) is also called a Value Object.