iBATIS Database Layer

Developer's Guide

Version 1.2.8

By Clinton Begin
October 12, 2003

Introduction

Welcome to the iBATIS Database Layer! This framework will help you design and implement better persistence layers for your Java applications. There are two main components to the framework, which are: SQL Maps and Data Access Objects. In addition to these, there are a number of utilities included that you might find useful.

SQL Maps

SQL Maps are by far the most exciting feature of the framework and really provide the core value of the iBATIS Database Layer. Using SQL Maps you can significantly reduce the amount of Java code that you normally need to access a relational database. SQL Maps simply map JavaBeans to SQL statements using a very simple XML descriptor. The biggest advantage of SQL Maps over other frameworks and object relational mapping tools is *simplicity*. To use SQL Maps you need only be familiar with JavaBeans, XML and SQL. There is very little else to learn. There is no complex scheme required to join tables or execute complex queries. Using SQL Maps you have the full power of SQL at your fingertips.

Data Access Objects (DAO)

When developing robust Java applications, it is often a good idea to isolate the specifics of your persistence implementation behind a common API. Data Access Objects allow you to create simple components that provide access to your data without revealing the specifics of the implementation to the rest of your application. Using DAOs you can allow your application to be dynamically configured to use different persistence mechanisms. If you have a complex application with a number of different databases and persistence approaches involved, DAOs can help you create a consistent API for the rest of your application to use.

Utilities

The iBATIS Database Layer includes a number of useful utilities including SimpleDataSource, a lightweight implementation of a JDBC 2.0 DataSource (JDBC 3.0 available as well). The ScriptRunner class also comes in handy for preparing databases for anything from unit testing to automated deployment.

Examples

There are a number of simple examples included with the framework in the examples.zip file. In addition there are a number of examples of working applications available at http://www.ibatis.com. This includes the very popular JPetStore application, which is an example of an online pet store. More examples will be coming soon!

About this Document

This document discusses the most important features of the iBATIS Database Layer. There are other features of the framework that are undocumented. If a feature is undocumented it is considered unsupported and not durable to change. Any undocumented features can change without notice, so you may want to avoid them. This document is constantly evolving as fast as the framework. Please ensure that you always have the matching version. If you find any errors or something that is difficult to understand, please email clinton.begin@ibatis.com.

SQL Maps (com.ibatis.db.sqlmap.*)

Concept

The SQL Map API allows programmers to easily map JavaBeans objects to PreparedStatement parameters and ResultSets. The Philosophy behind SQL Maps is simple: provide a simple framework to provide 80% of the functionality of JDBC in 20% of the code.

How does it work?

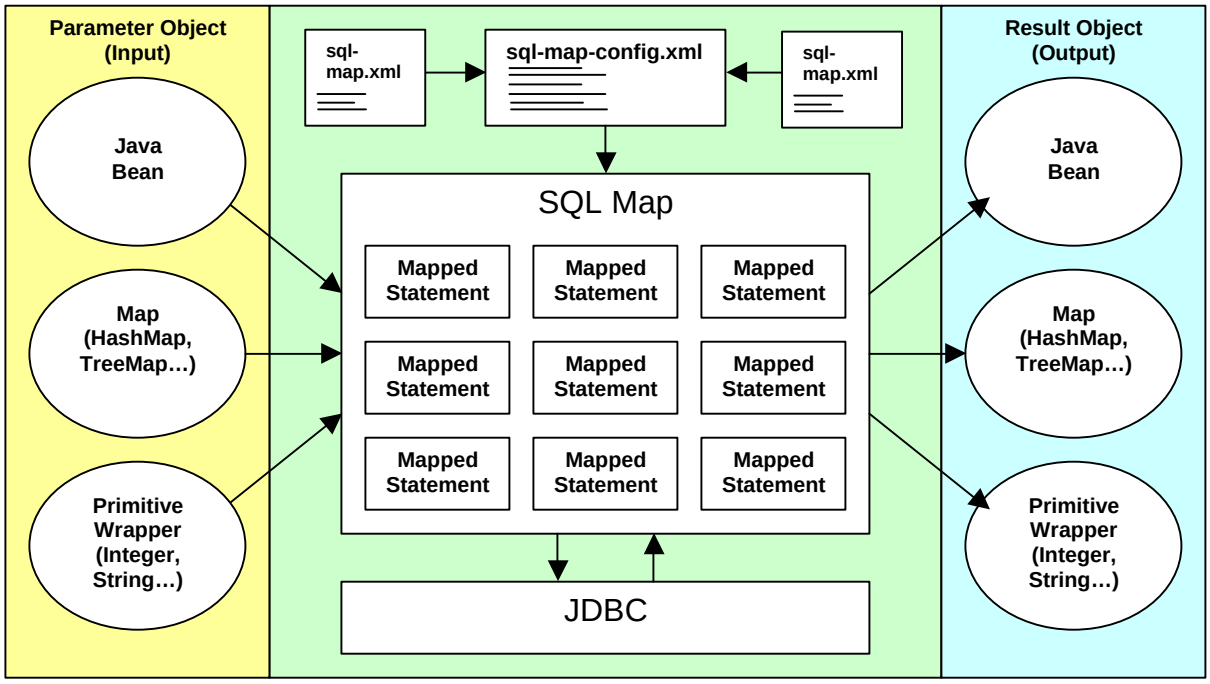
SQL Maps provides a very simple framework for using XML descriptors to map JavaBeans, Map implementations and even primitive wrapper types (String, Integer…) to a JDBC PreparedStatement. The idea is very simple. The general steps involved are:

1) Provide an object as a parameter (either a JavaBean, Map or primitive wrapper). The parameter object will be used setting input values in an update statement, or query values in a where clause (etc.).

2) Execute the mapped statement. This step is where the magic happens. The SQL Maps framework will create a PreparedStatement instance, set any parameters using the provided parameter object, execute the statement and build a result object from the ResultSet.

3) In the case of an updated, the number of rows effected is returned. In the case of a query, a single object, or a collection of objects is returned. Like parameters, result objects can be a JavaBean, a Map, or a primitive type wrapper.

The diagram below illustrates the flow as described.



Fast Track

The first section of this document will take you through a number of "Fast Tracks", which make up a quick walkthrough of a simple use of SQL Maps. Following the walkthrough, the topics will be discussed in full detail.

Fast Track: Preparing to Use SQL Maps

The SQL Maps framework is very tolerant of bad database models and even bad object models. Despite this, it is recommended that you use best practices when designing your database (proper normalization) and your object model. By doing so, you will get good performance and a clean design.

The easiest place to start is to analyze what you're working with. What are your business objects? What are your database tables? How do they relate to each other? For the first example, let's just use a simple Person class that follows the typical JavaBeans pattern.

Person.java

```
package examples.domain;

//imports implied….

public class Person {
   private int id;
   private String firstName;
   private String lastName;
   private Date birthDate;
   private double weightInKilograms;
   private double heightInMeters;

   public int getId () {
      return id;
   }
   public void setId (int id) {
      this.id = id;
   }

   //…let's assume we have the other getters and setters to save space…
}
```

Now how does this Person class map to our database? SQL Maps doesn't restrict you from having relationships such as table-per-class or multiple-tables-per-class or multiple-classes-per-table. Because you have the full power of SQL at your fingertips, there are very few restrictions. For this example, let's use the following simple table which would be suitable for a table-per-class relationship:

Person.sql

```
CREATE   TABLE   PERSON(
   PER_ID              NUMBER       (5, 0)    NOT NULL,
   PER_FIRST_NAME      VARCHAR      (40)      NOT NULL,
   PER_LAST_NAME       VARCHAR      (40)      NOT NULL,
   PER_BIRTH_DATE      DATETIME               ,
   PER_WEIGHT_KG       NUMBER       (4, 2)    NOT NULL,
   PER_HEIGHT_M        NUMBER       (4, 2)    NOT NULL,
   PRIMARY KEY (PER_ID)
)
```

Fast Track: The SQL Map Configuration File

Once we're comfortable with what we're working with, the best place to start is the SQL Map configuration file. This file will act as the root configuration for our SQL Map implementation.

The configuration file is an XML file. Within it we will configure properties, JDBC DataSources and SQL Maps. It gives you a nice convenient location to centrally configure your DataSource which can be any number of different implementations. The framework comes provided with DataSource implementations including iBATIS SimpleDataSource, Jakarta DBCP (Commons), and any DataSource that can be looked up via a JNDI context (e.g. from within an app server). These are described in more detail later in this document. For now, we'll just use Jakarta DBCP. The structure is simple and for the example above, might look like this:

SqlMapConfigExample.xml

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

<!DOCTYPE sql-map-config
   PUBLIC "-//iBATIS.com//DTD SQL Map Config 1.0//EN"
   "http://www.ibatis.com/dtd/sql-map-config.dtd">

<!-- Always ensure to use the correct XML header as above! -->

<sql-map-config>

  <!-- The properties (name=value) in the file specified here can be used placeholders in this config
       file (e.g. "${driver}". The file is relative to the classpath and is completely optional. -->

  <properties resource="examples/sqlmap/maps/SqlMapConfigExample.properties" />

  <!-- These settings control SqlMap configuration details, primarily to do with transaction
       management. They are all optional (more detail later in this document). -->

  <settings  maxExecute="300"
             maxExecutePerConnection="1"
             maxTransactions="10"
             statementCacheSize="75"
             useGlobalTransactions="false"
             useBeansMetaClasses="true"/>

  <!-- Configure a datasource to use with this SQL Map using Jakarta DBCP.
       Notice the use of the properties from the above resource -->

  <datasource name="basic" default = "true"
         factory-class="com.ibatis.db.sqlmap.datasource.DbcpDataSourceFactory">
    <property name="JDBC.Driver" value="${driver}"/>
    <property name="JDBC.ConnectionURL" value="${url}"/>
    <property name="JDBC.Username" value="${username}"/>
    <property name="JDBC.Password" value="${password}"/>
    <property name="Pool.MaximumActiveConnections" value="10"/>
    <property name="Pool.MaximumIdleConnections" value="5"/>
    <property name="Pool.MaximumWait" value="60000"/>
  </datasource>

  <!-- Identify all SQL Map XML files to be loaded by this SQL map. Notice the paths
       are relative to the classpath. For now, we only have one… -->

  <sql-map resource="examples/sqlmap/maps/Person.xml" />

</sql-map-config>
```

SqlMapConfigExample.properties

```
# This is just a simple properties file that simplifies automated configuration
# of the SQL Maps configuration file (e.g. by Ant builds or continuous
# integration tools for different environments… etc.)
# These values can be used in any property value in the file above (e.g. "${driver}")
# Using a properties file such as this is completely optional.

driver=oracle.jdbc.driver.OracleDriver
url=jdbc:oracle:thin:@localhost:1521:oracle1
username=jsmith
password=test
```

Fast Track: The SQL Map File(s)

Now that we have a DataSource configured and our central configuration file is ready to go, we will need to provide the actual SQL Map file which contains our SQL code and the mappings for parameter objects and result objects (input and output respectively).

Continuing with our example above, let's build an SQL Map file for the Person class and the PERSON table. We'll start with the general structure of an SQL document, and a simple select statement:

Person.xml

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

<!DOCTYPE sql-map
  PUBLIC "-//iBATIS.com//DTD SQL Map 1.0//EN"
  "http://www.ibatis.com/dtd/sql-map.dtd">

<sql-map name="Person">

    <mapped-statement name="getPerson" result-class="examples.domain.Person">
        SELECT
                PER_ID              as id,
                PER_FIRST_NAME      as firstName,
                PER_LAST_NAME       as lastName,
                PER_BIRTH_DATE      as birthDate,
                PER_WEIGHT_KG       as weightInKilograms,
                PER_HEIGHT_M        as heightInMeters
        FROM PERSON
        WHERE PER_ID = #value#
    </mapped-statement>

</sql-map>
```

The above example shows the simplest form of SQL Map. It uses a feature of the SQL Maps framework that automatically maps the columns of a ResultSet to JavaBeans properties (or Map keys etc.) based on name matching. The #value# token is an input parameter. More specifically, the use of "value" implies that we are using a simple primitive wrapper type (e.g. Integer; but we're not limited to this).

Although very simple, there are some limitations of using this approach. There is no way to specify the types of the output columns (if necessary), there is no way to automatically load related data (complex properties) and there is also a slight performance implication in that this approach requires accessing the ResultSetMetaData. By using a result-map, we can overcome all of these limitations. But, for now simplicity is our goal, and we can always change to a different approach later (without changing the Java source code).

Most database applications don't simply read from the database, they also have to modify data in the database. We've already seen an example of how a simple SELECT looks in a mapped statement, but what about INSERT, UPDATE and DELETE? The good news is that it's no different. Below we will complete our Person SQL Map with more statements to provide a complete set of statements for accessing and modifying data.

Person.xml

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

<!DOCTYPE sql-map
  PUBLIC "-//iBATIS.com//DTD SQL Map 1.0//EN"
  "http://www.ibatis.com/dtd/sql-map.dtd">

<sql-map name="Person">

    <!-- Use primitive wrapper type (e.g. Integer) as parameter and allow results to
        be auto-mapped results to Person object (JavaBean) properties -->
    <mapped-statement name="getPerson" result-class="examples.domain.Person">
        SELECT
                PER_ID              as id,
                PER_FIRST_NAME      as firstName,
                PER_LAST_NAME       as lastName,
                PER_BIRTH_DATE      as birthDate,
                PER_WEIGHT_KG       as weightInKilograms,
                PER_HEIGHT_M        as heightInMeters
        FROM PERSON
        WHERE PER_ID = #value#
    </mapped-statement>

    <!-- Use Person object (JavaBean) properties as parameters for insert.  Each of the
        parameters in the #hash# symbols is a JavaBeans property.  -->
    <mapped-statement name="insertPerson" >
        INSERT INTO
        PERSON  (PER_ID, PER_FIRST_NAME, PER_LAST_NAME,
                  PER_BIRTH_DATE, PER_WEIGHT_KG, PER_HEIGHT_M)
        VALUES (#id#, #firstName#, #lastName#,
                  #birthDate#, #weightInKilograms#, #heightInMeters#)
    </mapped-statement>

    <!-- Use Person object (JavaBean) properties as parameters for update. Each of the
        parameters in the #hash# symbols is a JavaBeans property.  -->
    <mapped-statement name="updatePerson" >
        UPDATE PERSON
        SET (PER_ID = PER_FIRST_NAME = #firstName#,
            PER_LAST_NAME = #lastName#, PER_BIRTH_DATE = #birthDate#,
            PER_WEIGHT_KG = #weightInKilograms#,
            PER_HEIGHT_M = #heightInMeters#)
        WHERE PER_ID = #id#
   </mapped-statement>

    <!-- Use Person object (JavaBean) "id" properties as parameters for delete. Each of the
        parameters in the #hash# symbols is a JavaBeans property.  -->
    <mapped-statement name="deletePerson" >
        DELETE PERSON
        WHERE PER_ID = #id#
   </mapped-statement>

</sql-map>
```

Fast Track: Programming with the SQL Map Framework

Now that we are all configured and mapped, all we need to do is code it in our Java application. The first step is to configure the SQL Map. This is very simply a matter of loading our SQL Map configuration XML file that we created before. To simplify loading the XML file, we can make use of the Resources class included with the framework.

```
String resource = "com/ibatis/example/sql-map-config.xml";
Reader reader = Resources.getResourceAsReader (resource);
SqlMap sqlMap =  XmlSqlMapBuilder.buildSqlMap(reader);
```

The SqlMap object is thread safe and is meant to be long-lived. For a given run of an application, you need only configure it once. This makes it a good candidate for a static member of a base class (e.g. base DAO class), or if you prefer to have it more centrally configured and globally available, you could wrap it up in a convenience class of your own. For example:

```
private MyAppSqlConfig {

   private static final SqlMap sqlMap;

   static {
      try {
         String resource = "com/ibatis/example/sql-map-config.xml";
         Reader reader = Resources.getResourceAsReader (resource);
         sqlMap =  XmlSqlMapBuilder.buildSqlMap(reader);
       } catch (Exception e) {
         // If you get an error at this point, it matters little what it was.  It is going to be
         // unrecoverable and we will want the app to blow up good so we are aware of the
         // problem.  You should always log such errors and re-throw them in such a way that
         // you can be made immediately aware of the problem.
         e.printStackTrace();
         throw new RuntimeException ("Error initializing MyAppSqlConfig class.  Cause: " + e);
       }
   }

   public static getSqlMapInstance () {
      return sqlMap;
   }

}
```

Reading Objects from the Database

Now that the SqlMap instance is initialized and easily accessible, we can make use of it. Let's first use it to get a Person object from the database. (For this example, let's assume there's 10 PERSON records in the database ranging from PER_ID 1 to 10).

To get a Person object from the database, we simply need the SqlMap instance, the name of the mapped statement and a Person ID. Let's load Person #5.

```
…
SqlMap sqlMap = MyAppSqlMapConfig.getSqlMapInstance(); // as coded above
…
Integer personPk = new Integer(5);
Person person = (Person) sqlMap.executeQueryForObject ("getPerson", personPk);
…
```

Writing Objects to the Database

We now have a Person object from the database. Let's modify some data. We'll change the person's height and weight.

```
…
person.setHeightInMeters(1.83);          // person as read from the database above
person.setWeightInKilograms(86.36);
…
sqlMap.executeUpdate("updatePerson", person);
…
```

If we wanted to delete this Person, it's just as easy.

```
…
sqlMap.executeUpdate("deletePerson", person);
…
```

Inserting a new Person is similar.

```
Person newPerson = new Person();
newPerson.setId(11);       // you would normally get the ID from a sequence or custom table
newPerson.setFirstName("Clinton");
newPerson.setLastName("Begin");
newPerson.setBirthDate (null);
newPerson.setHeightInMeters(1.83);
newPerson.setWeightInKilograms(86.36);
…
sqlMap.executeUpdate ("insertPerson", newPerson);
…
```

End of Fast Track

This is the end of the quick walkthrough. The next several sections will discuss the features of the SqlMap framework in more detail.

Side Bar: Object Graph Navigation (JavaBeans Properties, Maps, Lists)

Throughout this document you will be seeing objects accessed through a special syntax that might be familiar to anyone who has used Struts or any other JavaBeans compatible framework. If you are unfamiliar with JavaBeans, please see the JavaBeans discussion later in this document. The SqlMaps framework allows object graphs to be navigated via JavaBeans properties, Maps (key/value) and Lists. Consider the following navigation (includes a List, a Map and a JavaBean):

```
Employee emp = getSomeEmployeeFromSomewhere();
((Address) ( (Map)emp.getDepartmentList().get(3) ).get ("address")).getCity();
```

This property of the employee object could be navigated in an SqlMap property (ResultMap, ParameterMap etc…) as follows (given the employee object as above):

```
"departmentList[3].address.city"
```

The SQL Map XML Configuration File (http://www.ibatis.com/dtd/sql-map-config.dtd)

SQL Maps are configured using a central XML configuration file, which provides configuration details for DataSources, SQL Maps and other options like thread management.. The following is an example of the SQL Map configuration file:

sql-map-config.xml

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

<!DOCTYPE sql-map-config
   PUBLIC "-//iBATIS.com//DTD SQL Map Config 1.0//EN"
   "http://www.ibatis.com/dtd/sql-map-config.dtd">

<!-- Always ensure to use the correct XML header as above! -->

<sql-map-config>

  <!-- The properties (name=value) in the file specified here can be used placeholders in this config
       file (e.g. "${driver}". The file is relative to the classpath and is completely optional. -->

  <properties resource="examples/sqlmap/maps/SqlMapConfigExample.properties" />

  <!-- These settings control SqlMap configuration details, primarily to do with transaction
       management. They are all optional (more detail later in this document). -->

  <settings  maxExecute="300"
             maxExecutePerConnection="1"
             maxTransactions="10"
             statementCacheSize="75"
             useGlobalTransactions="false"
             useBeansMetaClasses="true" />

  <!-- Configure a datasource to use with this SQL Map using Jakarta DBCP
      notice the use of the properties from the above resource -->

  <datasource name="basic" default = "true"
         factory-class="com.ibatis.db.sqlmap.datasource.DbcpDataSourceFactory">
    <property name="JDBC.Driver" value="${driver}"/>
    <property name="JDBC.ConnectionURL" value="${url}"/>
    <property name="JDBC.Username" value="${username}"/>
    <property name="JDBC.Password" value="${password}"/>
    <property name="Pool.MaximumActiveConnections" value="10"/>
    <property name="Pool.MaximumIdleConnections" value="5"/>
    <property name="Pool.MaximumWait" value="60000"/>
  </datasource>

  <!-- Identify all SQL Map XML files to be loaded by this SQL map. Notice the paths
       are relative to the classpath. -->

  <sql-map resource="examples/sqlmap/maps/Person.xml" />
  <sql-map resource="examples/sqlmap/maps/Account.xml" />
  <sql-map resource="examples/sqlmap/maps/Order.xml" />
  <sql-map resource="examples/sqlmap/maps/LineItem.xml" />
  <sql-map resource="examples/sqlmap/maps/Product.xml" />

</sql-map-config>
```

The following sections of this document discuss the various sections of the SQL Map configuration file.

The <properties> Element

The SQL Map can have a single <properties> element that allows a standard Java properties file (name=value) to be associated with the SQL Map XML configuration document. By doing so, each named value in the properties file can become a variable that can be referred to in the SQL Map configuration file and all SQL Maps referenced within. For example, if the properties file contains the following:

driver=org.hsqldb.jdbcDriver

Then the SQL Map configuration file or each SQL Map referenced by the configuration document can use the placeholder ${driver} as a value that will be replaced by org.hsqldb.jdbcDriver. For example:

<property name="JDBC.Driver" value="**${driver}**"/>

This comes in handy during building, testing and deployment. It makes it easy to reconfigure your app for multiple environments or use automated tools for configuration (e.g. Ant).

The <settings> Element

The <settings> element allows you to configure various options and optimizations for the SqlMap instance that will be built using this XML file. The settings element and all of its attributes are completely optional. The attributes supported and their various behaviors are described in the following table:

maxExecute	This is the maximum number of threads that can enter MappedStatement.executeXXXXX() at a time. This includes threads that pass through the corresponding SqlMap.executeXXXXX() methods. Threads beyond the set value will be blocked until another thread exits. Different DBMS have different limits. *Example: maxExecute="100"* *Default: 0 (unlimited)*
maxExecutePerConnection	This is the maximum number of threads that can enter MappedStatement.executeXXXXX() at a time, for a given connection. This includes threads that pass through the corresponding SqlMap.executeXXXXX() methods. Threads beyond the set value will be blocked until another thread exits. The only time this value will have an effect is when Connection objects are shared (e.g. shared read connections). Different DBMS have different limits. *Example: maxExecutePerConnection="25"* *Default: 0 (unlimited)*
maxTransactions	This is the maximum number of threads that can enter SqlMap.startTransaction() at a time. Threads beyond the set value will be blocked until another thread exits. Different DBMS have different limits. *Example: maxTransactions="300"* *Default: 0 (none)*

statementCacheSize	This is the maximum number of PreparedStatements that will be cached. Cached statements are eliminated on a least-recently used basis. Different DBMS have different limits. *Example: statementCacheSize="175"* *Default: 0 (no statement cache)*
cacheModelsEnabled	This setting globally enables or disables all cache models for an SqlMap. This can come in handy for debugging. *Example: cacheModelsEnabled="true"* *Default: true (enabled)*
driverHintsEnabled	This setting enables hints to the driver, such as expected result set sizes, which can improve performance. Some drivers (PostgreSQL) do not support this. *Example: driverHintsEnabled="true"* *Default: false (disabled)*
useBeansMetaClasses	This setting controls a very important JavaBeans performance optimization. When enabled, the SqlMaps framework will use the CGLIB library for accessing JavaBeans properties instead of normal Java Reflection. The result is a performance increase, but at the expense of some initialization time. If for some reason the initialization time is unbearable, or if you are using an odd JDK that isn't compatible with CGLIB, you can disable this setting. IMPORTANT: This property is global to the JVM instance. If you disable this setting, it will be disabled for all SqlMap instances within the JVM. If set more than once, the last setting will be used. As a best practice, make sure all of you SqlMap config files set this consistently (all true, all false or all left as default). *Example: useBeansMetaClasses="true"* *Default: false (disabled)*
useFullyQualifiedStatementNames	With this setting enabled, you must always refer to mapped statements by their fully qualified name, which is the combination of the sql-map name and the mapped-statement name. For example: sqlMap.executeQuery("sqlMapName.statementName"…); *Example: useFullyQualifiedStatementNames="false"* *Default: false (disabled)*

useGlobalTransactions	This setting enables or disables support for global transactions, which are usually managed declaratively via a JTA compliant application server. This allows for easy integration in distributed environments where more than one database (or resource) may be involved in a transaction. *Example: useGlobalTransactions="true"* *Default: false (disabled)*
userTransactionJndiName	This very important setting will enable global (distributed) transaction management via a UserTransaction that is looked up based on the JNDI name specified as the value of this setting. This setting also requires useGlobalTransactions to be enabled as well (see above). If this setting is left unset, global transactions (UserTransaction) can still be leveraged, but must be programmatically controlled externally by the developer. *Example:* *userTransactionJndiName=java:comp/UserTransaction* *Default: unset (disabled)* *Requires: useGlobalTransactions="true"*
startTransactionBeforeConnection	Some application servers, such as WebSphere using Oracle, will require a different order of events when managing global transactions. If you encounter strange behavior or error messages relating to connections, resources and global transactions, try setting this to *false*. *Example:* *startTransactionBeforeConnection=false* *Default: true (connection started within transaction)* *Requires:* *useGlobalTransactions="true"* *and* *userTransactionJndiName=[something]*

The <datasource> Element

The <datasource> tag allows you to configure a factory that will take a set of properties, instantiate a DataSource and configure it for use with your SQL Maps. You can configure as many <datasource> instances as you like, but an SqlMap instance may only use one at a time. There are currently three datasource factories provided with the framework, but you can also write your own. The included DataSourceFactory implementations are:

SimpleDataSourceFactory

The SimpleDataSource factory provides a basic implementation of a pooling DataSource that is ideal for providing connections in cases where there is no container provided DataSource:

```
<datasource      name="hsql"
          factory-class="com.ibatis.db.sqlmap.datasource.SimpleDataSourceFactory"
          default="true">
     <property name="JDBC.Driver" value="org.postgresql.Driver"/>
     <property name="JDBC.ConnectionURL" value="jdbc:postgresql://server:5432/dbname"/>
```

```
    <property name="JDBC.Username" value="user"/>
    <property name="JDBC.Password" value="password"/>
    <!-- The following are optional -->
    <property name="Pool.MaximumActiveConnections" value="10"/>
    <property name="Pool.MaximumIdleConnections" value="5"/>
    <property name="Pool.MaximumCheckoutTime" value="120000"/>
    <property name="Pool.TimeToWait" value="10000"/>
    <property name="Pool.PingQuery" value="select * from dual"/>
    <property name="Pool.PingEnabled" value="false"/>
    <property name="Pool.PingConnectionsOlderThan" value="0"/>
    <property name="Pool.PingConnectionsNotUsedFor" value="0"/>
    <property name="Pool.QuietMode" value="true"/>
</datasource>
```

DbcpDataSourceFactory

This implementation uses Jakarta DBCP (Database Connection Pool) to provide connection pooling services via the DataSource API. This DataSource is ideal where the application/web container cannot provide a DataSource implementation, or you're running a standalone application. An example of the configuration parameters that must be specified for the DbcpDataSourceFactory are as follows:

```
<datasource name="basic" default = "true"
        factory-class="com.ibatis.db.sqlmap.datasource.DbcpDataSourceFactory">
  <property name="JDBC.Driver" value="${driver}"/>
  <property name="JDBC.ConnectionURL" value="${url}"/>
  <property name="JDBC.Username" value="${username}"/>
  <property name="JDBC.Password" value="${password}"/>
  <!-- The following are optional -->
  <property name="Pool.MaximumActiveConnections" value="10"/>
  <property name="Pool.MaximumIdleConnections" value="5"/>
  <property name="Pool.MaximumWait" value="60000"/>
  <!-- Use of the validation query can be problematic.  If you have difficulty, try without it.  -->
  <property name="Pool.ValidationQuery" value="select * from ACCOUNT"/>
  <property name="Pool.LogAbandoned" value="false"/>
  <property name="Pool.RemoveAbandoned" value="false"/>
  <property name="Pool.RemoveAbandonedTimeout" value="50000"/>
</datasource>
```

IMPORTANT NOTE: *DBCP support was added by request. However, there are currently some known issues with DBCP that you will want to be aware of if you are going to use it in an production application. These can be found at:*

http://issues.apache.org/bugzilla/buglist.cgi?bug_status=NEW&product=Commons&component=Dbcp

JndiDataSourceFactory

This implementation will retrieve a DataSource implementation from a JNDI context from within an application container. This is typically used when an application server is in use and a container managed connection pool and associated DataSource implementation are provided. The standard way to access a JDBC DataSource implementation is via a JNDI context. JndiDataSourceFactory provides functionality to access such a DataSource via JNDI. The configuration parameters that must be specified in the datasource stanza are as follows:

```
<datasource      name="somedb"
        factory-class="com.ibatis.db.sqlmap.datasource.JndiDataSourceFactory"
        default="true">
    <property name="DBFullJndiContext" value="java:comp/env/jdbc/jpetstore"/>
    <!-- The following can be used instead of DBFullJndiContext -->
    <property name="DBInitialContext" value="java:comp/env"/>
    <property name="DBLookup" value="/jdbc/jpetstore"/>
</datasource>
```

The <sql-map> Element

The <sql-map> element is used to explicitly include an SQL Map or another SQL Map Configuration file. Each SQL Map XML file that is going to be used by this SqlMap instance, must be declared. The SQL Map XML files will be loaded as a stream resource from the classpath. You must specify any and all SQL Maps (as many as there are) relative to your classpath. Here are some examples:

```
<sql-map resource="com/ibatis/examples/sql/Customer.xml" />
<sql-map resource="com/ibatis/examples/sql/Account.xml" />
<sql-map resource="com/ibatis/examples/sql/Product.xml" />
```

The SQL Map XML File Structure (http://www.ibatis.com/dtd/sql-map.dtd)

In the examples above, we saw the most simple forms of SQL Maps. There are other options available within the SQL Map document structure. Here is an example of a mapped statement that makes use of more features.

```
<sql-map    name="Product">
        <cache-model   name="product-cache" implementation="LRU">
                <flush-interval hours="24"/>
                <cache-property name="cache-size" value="1000" />
        </cache-model>

        <parameter-map   name="get-product-param">
                <property   name="id"/>
        </parameter-map>

        <result-map    name="get-product-result" class="com.ibatis.example.Product">
                <property   name="id" column="PRD_ID"/>
                <property   name="description" column="PRD_DESCRIPTION"/>
        </result-map>

        <mapped-statement   name="getProduct"  parameter-map="get-product-param"
                            result-map="get-product-result"   cache-model="product-cache">
                select * from PRODUCT where PRD_ID = ?
        </mapped-statement>
</sql-map>
```

The example above shows us a number of new features in the SQL Map file. The following sections will take you through the sections in more detail.

Note! A single SQL Map XML file can contain as many Mapped Statements, Parameter Maps and Result Maps as you like. Use discretion and organize the statements and maps appropriately for your application (group them logically).

Note! The name of an SQL Map is global and must be unique within a single group of SQL Maps to be used.

Mapped Statements

Mapped statements come in two flavors: Update and Query. However, there is nothing in the XML descriptor to distinguish between the two. In fact, it is not until the client code calls the API that the differentiation is made (due to the different return types: row count vs. result set). The complete structure of a mapped statement is as follows:

```
<mapped-statement name="statementName"
                  [parameter-class="some.class.Name"]
                  [result-class="some.class.Name"]
                  [parameter-map="nameOfParameterMap"]
                  [result-map="nameOfResultMap"]
                  [inline-parameters="true|false"]
                  [cache-model="nameOfCache"]
                  [is-stored-procedure="true|false"]
                  >
   select * from PRODUCT where PRD_ID = [?|#propertyName#]
   order by [$simpleDynamic$]
</mapped-statement>
```

In the above statement, the **[bracketed]** parts are optional and in some cases only certain combinations are allowed. So it is perfectly legal to have a Mapped Statement with as simple as this:

```
<mapped-statement name="insertTestProduct" >
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (1, "Shih Tzu")
</mapped-statement>
```

The above example is obviously unlikely, however this can come in handy if you want to simply make use of the SQL Map framework for executing arbitrary SQL statements. However, it will be more common to make use of the JavaBeans mapping features using Parameter Maps and Result Maps, as that is where the true power is. The next several sections describes the structure and attributes and how they effect the mapped statement.

The SQL

The SQL is obviously the most important part of the map. It can be any SQL that is valid for your database and JDBC driver. You can use any functions available and even send multiple statements as long as your driver supports it. Because you are combining SQL and XML in a single document, there is potential for conflicting special characters. The most common obviously is the greater-than and less-than symbols (<>). These are commonly required in SQL and are reserved symbols in XML. There is a simple solution to deal with these and any other special XML characters you might need to put in your SQL. By using a standard XML CDATA section, none of the special characters will be parsed and the problem is solved. For example:

```
<mapped-statement name="getPersonsByAge" result-class="examples.domain.Person">
    <![CDATA[
        SELECT *
        FROM PERSON
        WHERE AGE  >  #value#
    ]]>
</mapped-statement>
```

parameter-class

The value of the parameter-class attribute is the fully qualified name of a Java class (i.e. including package). The parameter-class attribute is optional and is used to limit the mapped statement to accept parameters of only a single type. For example, if you only wanted to allow objects of type (i.e. instanceof) "examples.domain.Product" to be passed in as a parameter, you could do something like this:

```
<mapped-statement name="statementName"
                  parameter-class=" examples.domain.Product">
  insert into PRODUCT values (#id#, #description#, #price#)
</mapped-statement>
```

Without a parameter-class specified, any JavaBean with appropriate properties (get/set methods) will be accepted as a parameter, which can be very useful in some situations.

result-class

The value of the result-class attribute is the fully qualified name of a Java class (i.e. including package). The result-class attribute allows us to specify a class that will be auto-mapped to our JDBC ResultSet based on the ResultSetMetaData. Wherever a property on the JavaBean and a column of the ResultSet match, the property will be populated with the column value. This makes query mapped statements very short and sweet indeed! For example:

```
<mapped-statement name="getPerson" result-class="examples.domain.Person">
        SELECT
                PER_ID                  as id,
                PER_FIRST_NAME          as firstName,
                PER_LAST_NAME           as lastName,
                PER_BIRTH_DATE          as birthDate,
                PER_WEIGHT_KG           as weightInKilograms,
                PER_HEIGHT_M            as heightInMeters
        FROM PERSON
        WHERE PER_ID = #value#
</mapped-statement>
```

In the example above, the Person class has properties including: *id, firstName, lastName, birthDate, weightInKilograms and heightInMeters*. Each of these corresponds with the column aliases described by the SQL select statement (using the "as" keyword –a standard SQL feature). Column aliases are only required if the database column names don't match, which in general they should not. When executed, a Person object will be instantiated and the results from the result set will be mapped to the instance based on the property names and column names.

As stated above, there are some limitations of using auto-mapping with a result-class. There is no way to specify the types of the output columns (if necessary), there is no way to automatically load related data (complex properties) and there is also a slight performance implication in that this approach requires accessing the ResultSetMetaData. For now, simplicity is our goal, and we can always change to a different approach later (without changing the Java source code). These limitations can be overcome by using an explicit result-map. Result maps are described in more detail later in this document.

parameter-map

The parameter-map attribute is rarely used. The value of the parameter-class attribute is the name of a defined parameter-map element (see below). A more common approach (and the default approach) is to use inline parameters (described below). However, this is a good approach if XML purity and consistency is your concern.

Note! Dynamic mapped statements (described below) only support inline parameters and do not work with parameter maps.

The idea of a parameter-map is to define an ordered list of parameters that match up with the value tokens of a JDBC PreparedStatement. For example:

```
<parameter-map name="insert-product-param">
        <property name="id"/>
        <property name="description"/>
</parameter-map>

<mapped-statement name="insertProduct"
parameter-map="insert-product-param">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?);
</mapped-statement>
```

In the example above, the parameter map describes two parameters that will match, in order, the value tokens ("?") in the SQL statement. So the first "?" will be replaced by the value of the "id" property and the second with the "description" property. Parameter maps and their options are described in more detail later in this document.

result-map

The result-map property is one of the most commonly used and most important attributes to understand. The value of the result-map attribute is the name of a defined result-map element (see below). Using the

result-map attribute allows you to control how data is extracted from a result-set and which properties to map to which columns. Unlike the auto-mapping approach using the result-class attribute (above), the result-map allows you to describe the column type, a null value replacement, lazy load options, and complex property mappings (including other JavaBeans, Collections and primitive type wrappers).

The full details of the result-map structure are discussed later in this document, but the following example will demonstrate how the result-map looks related to a mapped-statement.

```
<result-map name="get-product-result" class="com.ibatis.example.Product">
        <property name="id" column="PRD_ID"/>
        <property name="description" column="PRD_DESCRIPTION"/>
</result-map>

<mapped-statement name="getProduct" result-map="get-product-result">
select * from PRODUCT
</mapped-statement>
```

In the example above, the ResultSet from the SQL query will be mapped to a Product instance using the result-map definition. The result-map shows that the "id" property will be populated by the "PRD_ID" column and the "description" property will be populated by the "PRD_DESCRIPTION" column.

Notice that using "select *" is okay. There is no need to map all of the columns in the ResultSet.

inline-parameters (default)

The inline-parameters attribute is rarely used, because it is enabled by default and automatically disabled if an external parameter-map is in use. It's value is either "true" or "false". When set to true or left undefined, inline parameters can be used inside of a mapped statement. Full details of inline parameters are discussed later in this document, but here's an example to get you started.

```
<mapped-statement name="insertProduct" inline-parameters="true">
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
    values (#id#, #description#);
</mapped-statement>
```

In the example above, the inline parameters are *#id#* and *#description#*. Each represents a JavaBeans property that will be used to populate the statement parameter in-place. In the example above, the Product class (that we've used from previous examples) has *id* and *description* properties that will be read for a value to be placed in the statement where the associated property token is located. So for a statement that is passed a Product with *id=5* and *description="dog"*, the statement might be executed as follows:

```
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
    values (5, 'dog');
```

This is done via the standard JDBC APIs, particularly the PreparedStatement API.

cache-model

The cache-model attribute value is the name of a defined cache-model element (see below). A cache-model is used to describe a cache for use with a query mapped statement. Each query mapped statement can use a different cache-model, or the same one. Full details of the cache-model element and its attributes are discussed later. The following example will demonstrate how it looks related to a mapped-statement.

```
<cache-model name="product-cache" imlementation="LRU">
  <flush-interval hours="24"/>
  <flush-on-execute statement="insertProduct"/>
  <flush-on-execute statement="updateProduct"/>
  <flush-on-execute statement="deleteProduct"/>
  <cache-property name="cache-size" value="1000" />
</cache-model>

<mapped-statement name="getProductList"
                  cache-model="product-cache">
  select * from PRODUCT where PRD_CAT_ID = #value#
</mapped-statement>
```

In the above example, a cache is defined for products that uses a STRONG reference type and flushes every 24 hours or whenever associated update statements are executed.

is-stored-procedure

The value of is-stored-procedure is "true" or "false". By default it is set to false. When set to true, the SQL statement will be executed via the CallableStatement API rather than the PreparedStatement API. Most stored procedures can be executed within a mapped-statement, but output parameters are not supported.

Parameter Maps and Inline Parameters

As you've seen above, the parameter-map is responsible for mapping JavaBeans properties to the parameters of a mapped-statement. Although parameter-maps are rare in their external form, understanding them will help you understand inline parameters. Inline parameters are discussed immediately following this section.

```
<parameter-map name="parameterMapName">
        <property name="propertyName" [type="NUMERIC"]  [null="-9999999"]/>
        <property …… />
        <property …… />
</parameter-map>
```

The parts in **[brackets]** are optional. The parameter-map itself only has a *name* attribute that is an identifier that is globally unique to the collection of SQL Map documents. The parameter-map can contain any number of property mappings that map directly to the parameters of a mapped-statement. The next few sections describe the attributes of the *property* elements:

name

The name attribute of the parameter map is the name of a JavaBeans property (get methods) of the parameter object passed to a mapped statement. The name can be used more than once depending on the number of times it is needed in the statement (e.g. where the same property that is updated in the set clause of an SQL update statement, is also used as the key in the where clause).

type

The type attribute is used to explicitly specify the database column type (i.e. not the Java class type) of the parameter to be set by this property. Some JDBC drivers are not able to identify the type of a column for certain operations without explicitly telling the driver the column type. A perfect example of this is the PreparedStatement.setNull(int parameterIndex, int sqlType) method. This method requires the type to be specified. Some drivers will allow the type to be implicit by simply sending Types.OTHER or Types.NULL. However, the behavior is inconsistent and some drivers need the exact type to be specified. For such situations, the SQL Maps API allows the type to be specified using the type attribute of the parameter-map property element.

This attribute is normally only required if the column is nullable. Although, another reason to use the type attribute is to explicitly specify date types. Whereas Java only has one Date value type (java.util.Date), most SQL databases have many –usually at least 3 different types. Because of this you might want to specify explicitly that your column type is DATE versus DATETIME (etc.).

The type attribute can be set to any string value that matches a constant in the JDBC Types class. Although it can be set to any of these, some types are not supported (e.g. blobs). A section later in this document describes the types that are supported by the framework.

Note! Most drivers only need the type specified for nullable columns. Therefore, for such drivers you only need to specify the type for the columns that are nullable.

Note! When using an Oracle driver, you will get an "Invalid column type" error if you attempt to set a null value to a column without specifying its type.

null

The value of the null attribute is any valid value (based on property type). The null attribute is used to specify an outgoing null value replacement. What this means is that when the value is detected in the JavaBeans property, a NULL will be written to the database (the opposite behavior of an inbound null value replacement). This allows you to use a "magic" null number in your application for types that do not support null values (e.g. int, double, float etc.). When these types of properties contain a matching null value (e.g. –9999), a NULL will be written to the database instead of the value.

An example of a real parameter-map that uses the full structure is as follows

```
<parameter-map   name="insert-product-param">
        <property name="id" type="NUMERIC" null="-9999999"/>
        <property name="description" type="VARCHAR" null="NO_ENTRY"/>
</parameter-map>

<mapped-statement   name="insertProduct"
                    parameter-map="insert-product-param">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?);
</mapped-statement>
```

In the above example, the JavaBeans properties *id* and *description* will be applied to the parameters of the Mapped Statement *insertProduct* in the order they are listed. So, *id* will be applied to the first parameter (?) and *description* to the second. If the orders were reversed, the XML would look like the following:

```
<parameter-map name="insert-product-param">
        <property name="description" />
        <property name="id"/>
</parameter-map>

<mapped-statement name="insertProduct"
                        parameter-map="insert-product-param">
    insert into PRODUCT (PRD_DESCRIPTION, PRD_ID) values (?,?);
</mapped-statement>
```

Note! Property Maps do not automatically bind to a particular class type. Therefore based on the above example, any class instance with JavaBeans compliant properties for *id* and *description* would satisfy the property map. If you want it to bind a specific class, you can use the parameter-class attribute (described in an earlier section).

Note! Parameter Map names are always local to the SQL Map XML file that they are defined in. You can refer to a Parameter Map in another SQL Map XML file by prefixing the name of the Parameter Map with

the name of the SQL Map (set in the <sql-map> root tag). For example, to refer to the above parameter map from a different file, the full name to reference would be "Product.insert-product-param".

Inline Parameter Maps

Although very descriptive, the above syntax for declaring parameter-maps is verbose. There is a more popular alternate syntax for Parameter Maps that can simplify the definition and reduce code. This alternate syntax places the JavaBeans property names inline with the Mapped Statement (i.e. coded directly into the SQL). By default, any Mapped Statement that has no explicit parameter-map specified will be parsed for inline parameters. You can explicitly enable or disable inline parameters by specifying inline-parameters="true|false" (described earlier in this document). The previous example (i.e. product), implemented with an inline parameter map, would look like this:

```
<mapped-statement name="insertProduct" inline-parameters="true">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id#, #description#);
</mapped-statement>
```

Declaring types can be accomplished with inline parameters by using the following syntax:

```
<mapped-statement name="insertProduct" inline-parameters="true">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id:NUMERIC#, #description:VARCHAR#);
</mapped-statement>
```

Declaring types and null value replacements can be accomplished with inline parameters by using the following syntax:

```
<mapped-statement name="insertProduct" inline-parameters="true">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id:NUMERIC:-999999#, #description:VARCHAR:NO_ENTRY#);
</mapped-statement>
```

Note! When using inline parameters, you cannot specify the null value replacement without also specifying the type. You must specify both.

Note! If you full transparency of null values, you must also specify null value replacements in your result maps, as discussed later in this document.

Note! If you require a lot of type descriptors and null value replacements, you might be able to achieve cleaner code by using an external parameter-map.

Primitive Type Parameters

It is not always necessary or convenient to write a JavaBean just to use as a parameter. In these cases you are perfectly welcome to use a primitive type wrapper object (String, Integer, Date etc.) as the parameter directly. For example:

```
<mapped-statement name="insertProduct" >
   select * from PRODUCT where PRD_ID = #value#
</mapped-statement>
```

Assuming PRD_ID is a *numeric* type, when a call is made to this mapped statement an java.lang.Integer object can be passed in. The **#value#** parameter will be replaced with the value of the Integer instance. It is important to use the parameter name **"value"** any time you are using a primitive type instead of a JavaBean. Result Maps (discussed below) support primitive types as results as well. See the Result Map section and Programming SQL Maps (API) section below for more information about using primitive types as parameters.

Map Type Parameters

If you are in a situation where it is not necessary or convenient to write a JavaBean class, and a single primitive type parameter won't do (e.g. there are multiple parameters), you can use a Map (e.g. HashMap, TreeMap) as a parameter object. For example:

```
<mapped-statement   name="insertProduct" >
        select * from PRODUCT
        where PRD_CAT_ID = #catId#
        and PRD_CODE = #code#
</mapped-statement>
```

Notice that there is no difference in the mapped statement implementation! In the above example, if a Map instance was passed into the call to the statement, the Map must contain keys named "catId" and "code". The values referenced by those keys would be of the appropriate type, such as Integer and String (perhaps for the example above). Result Maps (discussed below) support Map types as results as well. See the Result Map section and Programming SQL Maps (API) section below for more information about using Map types as parameters.

Result Maps

Result maps are an extremely important component of SQL Maps. The result-map is responsible for mapping JavaBeans properties to the columns of a ResultSet produced by executing a query mapped statement. The structure of a result-map looks like this:

```
<result-map  name="resultMapName"  class="some.domain.Class" [extends="parent-result-map"]>
        <property  name="propertyName"  column="COLUMN_NAME"
                [columnIndex="1"  type="TYPE"]  [null="-999999"]
                [mapped-statement="someOtherStatement"]
                [lazy-load="true|false"]
                />
        <property ……/>
        <property ……/>
        <property ……/>
</result-map>
```

The parts in **[brackets]** are optional. The result-map itself has a ***name*** attribute that is an identifier that is globally unique to the collection of SQL Map documents. The result-map also has a ***class*** attribute that is the fully qualified (i.e. full package) name of a class. This class will be instantiated and populated based on the property mappings it contains. The ***extends*** attribute can be optionally set to the name of another result-map upon which to base a result-map. This is similar to extending a class in Java, all properties of the super result-map will be included as part of the sub result-map. The properties of the super result-map are always inserted before the sub result-map properties and the parent result-map must be defined before the child. The classes for the super/sub result-maps need not be the same, nor do they need to be related at all (they can each use any class).

The result-map can contain any number of property mappings that map JavaBeans to the columns of a ResultSet. These property mappings will be applied in the order that they are defined in the document. The associated class must be a JavaBeans compliant class with appropriate get/set methods for each of the properties.

Note! The columns will be read explicitly in the order specified in the Result Map (this comes in handy for some poorly written JDBC drivers).

The next few sections describe the attributes of the *property* elements:

name

The name attribute of the result map property is the name of a JavaBeans property (get method) of the result object that will be returned by the mapped statement. The name can be used more than once depending on the number of times it is needed to populate the results.

column

The column attribute value is the name of the column in the ResultSet from which the value will be used to populate the JavaBeans property.

columnIndex

As an optional performance enhancement, the columnIndex attribute value is the index of the column in the ResultSet from which the value will be used to populate the JavaBeans property. This is not likely needed in 99% of applications and sacrifices readability for speed. Some JDBC drivers may not realize any performance benefit, while others will speed up dramatically.

type

The type attribute is used to explicitly specify the database column type (i.e. not the Java type) of the ResultSet column that will be used to populate the JavaBean property. Although result maps do not have the same difficulties with null values, specifying the type can be useful for certain mapping types such as Date properties. Because Java only has one Date value type and SQL databases may have many (usually at least 3), specifying the date may become necessary in some cases to ensure that dates (or other types) are set correctly. Similarly, String types may be populated by a VARCHAR, CHAR or CLOB, so specifying the type might be needed in those cases too (driver dependent).

null

The null attribute specifies the value to be used in place of a NULL value in the database. So if a NULL is read from the ResultSet, the JavaBean property will be set to the value specified by the null attribute instead of. The null attribute value can be any value, but must be appropriate for the property type.

If your database has a NULLABLE column, but you want your application to represent NULL with a constant value you can specify it in the result map as follows:

```
<result-map name="get-product-result" class="com.ibatis.example.Product">
        <property name="id" column="PRD_ID"/>
        <property name="description" column="PRD_DESCRIPTION"/>
        <property name="subCode" column="PRD_SUB_CODE" null="-999"/>
</result-map>
```

In the above example, if PRD_SUB_CODE is read as NULL, then the subCode property will be set to the value of -999. This allows you to use a primitive type in your Java class to represent a NULLABLE column in the database. Remember that if you want this to work for queries as well as updates/inserts, you must also specify the NULL value in the parameter map (discussed earlier in this document).

mapped-statement

The mapped-statement attribute is used to describe a relationship between objects and automatically load complex (i.e. user defined) property types. The value of the mapped-statement property must be the name of another mapped statement. The value of the database column (the column attribute) that is defined in the same property element as this mapped-statement attribute will be passed to the related mapped statement as the parameter. Therefore the column must be a supported, primitive type. More information about supported primitive types and complex property mappings/relationships is discussed later in this document.

lazy-load

When using a mapped-statement to load a complex collection property (discussed later), this lazy load attribute can be set to "true" or "false" to enable or disable lazy loading. By default it is enabled. When enabled collections will not be loaded until they are first accessed by the application. (Note: If you're planning on serializing your result objects across a network, you should set lazy-loading="false").

Implicit Result Maps

If you have a very simple requirement that does not require the reuse of an explicitly defined result-map, there is a quick way to implicitly specify a result map by setting a result-class attribute of a mapped statement. The trick is that you must ensure that the result set returned has column names (or labels/aliases) that match up with the write-able property names of your JavaBean. For example, if we consider the Product class described above, we could create a mapped statement with an implicit result map as follows:

```
<mapped-statement name="getProduct" result-class="com.ibatis.example.Product">
        select
           PRD_ID as id,
           PRD_DESCRIPTION as description
        from PRODUCT
        where PRD_ID = #value#
</mapped-statement>
```

The above mapped statement specifies a result-class and declares aliases for each column that match the JavaBean properties of the Product class. This is all that is required, no result map is needed. The tradeoff here is that you don't have an opportunity to specify a column type (normally not required) or a null value (or any other property attributes). The one other condition to keep in mind is that databases are very rarely case sensitive, and therefore implicit result maps are not case sensitive either. So if your JavaBean had two properties, one named firstName and another named firstname, these would be considered identical and you could not use an implicit result map (it would also identify a potential problem with the design of the JavaBean class). Furthermore, there is some performance overhead associated with auto-mapping via a result-class. Accessing ResultSetMetaData can be slow with some JDBC drivers.

Primitive Results (i.e. String, Integer, Boolean)

In addition to supporting JavaBeans compliant classes, Result Maps can conveniently populate a simple Java type wrapper such as String, Integer, Boolean etc. Collections of primitive objects can also be retrieved using the APIs described below (see executeQueryForList()). Primitive types are mapped exactly the same way as a JavaBean, with only one thing to keep in mind. A primitive type can only have one property that must be named "value". For example, if we wanted to load just a list of all product descriptions (Strings) instead of the entire Product class, the map would look like this:

```
<result-map name="get-product-result" class="java.lang.String">
        <property name="value" column="PRD_DESCRIPTION"/>
</result-map>
```

A simpler approach is to simply use a result class in a mapped statement (make note of the column alias "value" using the "as" keyword):

```
<mapped-statement  name="getProductCount" result-class="java.lang.Integer">
    select count(1) as value
    from PRODUCT
</mapped-statement>
```

Map Results

Result Maps can also conveniently populate a Map instance such as HashMap or TreeMap. Collections of such objects (e.g. Lists of Maps) can also be retrieved using the APIs described below (see executeQueryForList()). Map types are mapped exactly the same way as a JavaBean, but instead of setting JavaBeans properties, the keys of the Map are set to reference the values for the corresponding mapped columns. For example, if we wanted to load the values of a product quickly into a Map, we could do the following:

```
<result-map name="get-product-result" class="java.util.HashMap">
        <property name="id" column="PRD_ID"/>
        <property name="code" column="PRD_CODE"/>
        <property name="description" column="PRD_DESCRIPTION"/>
        <property name="suggestedPrice" column="PRD_SUGGESTED_PRICE"/>
</result-map>
```

In the example above, an instance of HashMap would be created and populated with the Product data. The property name attributes (e.g. "id") would be the keys of the HashMap, and the values of the mapped columns would be the values in the HashMap.

Of course, you can also use an implicit result map with a Map type. For example:

```
<mapped-statement  name="getProductCount" result-class="java.util.HashMap">
    select * from PRODUCT
</mapped-statement>
```

The above would basically give you a Map representation of the returned ResultSet.

Complex Properties (i.e. a property of a class defined by the user)

It is possible to automatically populate properties of complex types (classes created by the user) by associating a result-map property with a mapped statement that knows how to load the appropriate data and class. In the database the data is usually represented via a 1:1 relationship, or a 1:M relationship where the class that holds the complex property is from the "many side" of the relationship and the property itself is from the "one side" of the relationship. Consider the following example:

```
<result-map name="get-product-result" class="com.ibatis.example.Product">
        <property name="id" column="PRD_ID"/>
        <property name="description" column="PRD_DESCRIPTION"/>
        <property name="category" column="PRD_CAT_ID" mapped-statement="getCategory"/>
</result-map>

<result-map name="get-category-result" class="com.ibatis.example.Category">
        <property name="id" column="CAT_ID"/>
        <property name="description" column="CAT_DESCRIPTION"/>
</result-map>

<mapped-statement   name="getProduct"   result-map="get-product-result">
        select * from PRODUCT where PRD_ID = #value#
</mapped-statement>

<mapped-statement   name="getCategory"   result-map="get-category-result">
        select * from CATEGORY where CAT_ID = #value#
</mapped-statement>
```

In the above example, an instance of *Product* has an property called *category* of type *Category*. Since category is a complex user type (i.e. a user defined class), JDBC does not have the means to populate it. By associating another mapped statement with the property mapping, we are providing enough information

for the SQL Map engine to populate it appropriately. Upon executing *getProduct*, the *get-product-result* Result Map will call *getCategory* using the value returned in the *PRD_CAT_ID* column. The *get-category-result* Result Map will instantiate a Category and populate it. The whole Category instance then gets set into the Product's category property.

Complex Collection Properties

It is also possible to load properties that represent lists of complex objects. In the database the data would be represented by a M:M relationship, or a 1:M relationship where the class containing the list is on the "one side" of the relationship and the objects in the list are on the "many side". To load a List of objects, there is no change to the mapped-statement (see example above). The only difference required to cause the SQL Map framework to load the property as a List is that the property on the business object must be of type java.util.List or java.util.Collection. For example, if a Category has a List of Product instances, the mapping would look like this (assume Category has a property called "productList" of type java.util.List):

```
<result-map name="get-category-result" class="com.ibatis.example.Category">
    <property name="id" column="CAT_ID"/>
    <property name="description" column="CAT_DESCRIPTION"/>
    <property name="productList" column="CAT_ID" mapped-statement=" getProductsByCatId"/>
</result-map>

<result-map name="get-product-result" class="com.ibatis.example.Product">
        <property name="id" column="PRD_ID"/>
        <property name="description" column="PRD_DESCRIPTION"/>
</result-map>

<mapped-statement       name="getCategory"
                        inline-parameters="true"
                        result-map="get-category-result">
        select * from CATEGORY where CAT_ID = #value#
</mapped-statement>

<mapped-statement       name="getProductsByCatId"
                        result-map="get-product-result">
        select * from PRODUCT where PRD_CAT_ID = #value#
</mapped-statement>
```

If you are using the Microsoft SQL Server 2000 Driver for JDBC you may need to add SelectMethod=Cursor to the connection url in order to execute multiple statements while in manual transaction mode (see MS Knowledge Base Article 313181: http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B313181).

Composite Keys or Multiple Complex Parameters Properties

You might have noticed that in the above examples there is only a single key being used as specified in the result-map by the *column* attribute. This would suggest that only a single column can be associated to a related mapped statement. However, there is an alternate syntax that allows multiple columns to be passed to the related mapped statement. This comes in handy for situations where a composite key relationship exists, or even if you simply want to use a parameter of some name other than *#value#*. The alternate syntax for the column attribute is simply {param1=column1, param2=column2, …, paramN=columnN}. Consider the example below where the PAYMENT table is keyed by both Customer ID and Order ID:

```
<result-map name="get-order-result" class="com.ibatis.example.Order">
    <property name="id" column="ORD_ID"/>
    <property name="customerId" column="ORD_CST_ID"/>
    …
    <property name="payments" column="{itemId=ORD_ID, custId=ORD_CST_ID}"
```

```
            mapped-statement=" getOrderPayments"/>
</result-map>
```

```
<mapped-statement      name="getOrderPayments"
                       result-map="get-payment-result">
        select * from PAYMENT
        where PAY_ORD_ID  = #itemId#
        and PAY_CST_ID  = #custId#
</mapped-statement>
```

Important! Currently the SQL Map framework does not automatically resolve circular relationships. Be aware of this when implementing parent/child relationships (trees). An easy workaround is to simply define a second result map for one of the cases that does not load the parent object (or vice versa).

Note! Some JDBC drivers (e.g. PointBase Embedded) do not support multiple ResultSets (per connection) open at the same time. Such drivers will not work with complex object mappings because the SQL Map engine requires multiple ResultSet connections.

Note! Result Map names are always local to the SQL Map XML file that they are defined in. You can refer to a Result Map in another SQL Map XML file by prefixing the name of the Result Map with the name of the SQL Map (set in the <sql-map> root tag).

Supported Types for Parameter Maps and Result Maps

The Java types supported by the iBATIS framework for parameters and results are as follows:

Type	JavaBean Property Mapping	result-class/parameter-class
boolean	YES	NO
java.lang.Boolean	YES	YES
byte	YES	NO
java.lang.Byte	YES	YES
short	YES	NO
java.lang.Short	YES	YES
int	YES	NO
java.lang.Integer	YES	YES
long	YES	NO
java.lang.Long	YES	YES
float	YES	NO
java.lang.Float	YES	YES
double	YES	NO
java.lang.Double	YES	YES
java.lang.String	YES	YES
java.util.Date	YES	YES
java.math.BigDecimal	YES	YES
* java.sql.Date	YES	YES
* java.sql.Time	YES	YES
* java.sql.Timestamp	YES	YES

** The use of java.sql. data types is discouraged. It is a best practice to use java.lang.Date instead.*

Note! Primitive types such as *int*, *boolean* and *float* cannot be directly supported as primitive types, as the iBATIS Database Layer is a fully Object Oriented approach. Therefore all parameters and results must be an Object at their highest level. Incidentally the autoboxing features of JDK 1.5 will allow these primitives to be used as well.

Caching Mapped Statement Results

The results from a Query Mapped Statement can be cached simply by specifying the cache-model parameter in the mapped-statement tag (seen above). A cache model is a configured cache that is defined within your SQL map. Cache models are configured using the cache-model element as follows:

```
<cache-model name="product-cache" implementation ="LRU">
   <flush-interval hours="24"/>
   <flush-on-execute statement="insertProduct"/>
   <flush-on-execute statement="updateProduct"/>
   <flush-on-execute statement="deleteProduct"/>
   <cache-property name="cache-size" value="1000" />
</cache-model>
```

The cache model above will create an instance of a cache named "product-cache" that uses a Least Recently Used (LRU) implementation. The value of the implementation attribute is either a fully qualified class name, or an alias for one of the included implementations (see below). Based on the flush elements specified within the cache model, this cache will be flushed every 24 hours. There can be only one flush interval element and it can be set using *hours*, *minutes*, *seconds* or *milliseconds*. In addition the cache will be flushed whenever the *insertProduct*, *updateProduct*, or *deleteProduct* mapped statements are executed. There can be any number of "flush on execute" elements specified for a cache. Some cache implementations may need additional properties, such as the 'cache-size' property demonstrated above. In the case of the LRU cache, the size determines the number of entries to store in the cache. Once a cache model is configured, you can specify the cache model to be used by a mapped statement, for example:

```
<mapped-statement name="getProductList"   cache-model="product-cache">
      select * from PRODUCT where PRD_CAT_ID = #value#
</mapped-statement>
```

The cache model uses a pluggable framework for supporting different types of caches. The implementation is specified in the implementation attribute of the cache-model element (as discussed above). The class name specified must be an implementation of the CacheController interface, or one of the four aliases discussed below. Further configuration parameters can be passed to the implementation via the cache-property elements contained within the body of the cache-model. Currently there are 4 implementations included with the distribution. These are as follows:

"MEMORY" (com.ibatis.db.sqlmap.cache.memory.MemoryCacheController)

The MEMORY cache implementation uses reference types to manage the cache behavior. That is, the garbage collector effectively determines what stays in the cache or otherwise. The MEMORY cache is a good choice for applications that don't have an identifiable pattern of object reuse, or applications where memory is scarce (it will do what it can).

The MEMORY implementation is configured as follows:

```
<cache-model name="product-cache" implementation ="MEMORY">
   <flush-interval hours="24"/>
   <flush-on-execute statement="insertProduct"/>
   <flush-on-execute statement="updateProduct"/>
   <flush-on-execute statement="deleteProduct"/>
   <cache-property name="reference-type" value="WEAK" />
</cache-model>
```

Only a single cache-property is recognized by the MEMORY cache implementation. This property, named 'reference-type' must be set to a value of STRONG, SOFT or WEAK. These values correspond to various memory reference types available in the JVM.

The following table describes the different reference types that can be used for a MEMORY cache. To better understand the topic of reference types, please see the JDK documentation for java.lang.ref for more information about "reachability".

WEAK *(default)*	This reference type is probably the best choice in most cases and is the default if the reference-type is not specified. It will increase performance for popular results, but it will absolutely release the memory to be used in allocating other objects, assuming that the results are not currently in use.
SOFT	This reference type will reduce the likelihood of running out of memory in case the results are not currently in use and the memory is needed for other objects. However, this is not the most aggressive reference type in that regard and memory still might be allocated and made unavailable for more important objects.
STRONG	This reference type will guarantee that the results stay in memory until the cache is explicitly flushed (e.g. by time interval or flush on execute). This is ideal for results that are: 1) very small, 2) absolutely static, and 3) used very often. The advantage is that performance will be very good for this particular query. The disadvantage is that if the memory used by these results is needed, then it will not be released to make room for other objects (possibly more important objects).

A note about backward compatibility: To maintain backward compatibility 3 preconfigured cache-models are made automatically available. They are named the same as the reference types that they represent: STRONG, SOFT and WEAK. These cache models do not have any declarative flush policy associated with them and therefore they must be flushed programmatically. Furthermore, the old approach of specifying reference-type as an attribute (without specifying implementation) is still supported and will result in a MEMORY cache of the appropriate type.

"LRU" (com.ibatis.db.sqlmap.cache.lru.LruCacheController)

The LRU cache implementation uses an Least Recently Used algorithm to determines how objects are automatically removed from the cache. When the cache becomes over full, the object that was accessed least recently will be removed from the cache. This way, if there is a particular object that is often referred to, it will stay in the cache with the least chance of being removed. The LRU cache makes a good choice for applications that have patterns of usage where certain objects may be popular to one or more users over a longer period of time (e.g. navigating back and forth between paginated lists, popular search keys etc.).

The LRU implementation is configured as follows:

```
<cache-model name="product-cache" implementation ="LRU">
   <flush-interval hours="24"/>
   <flush-on-execute statement="insertProduct"/>
   <flush-on-execute statement="updateProduct"/>
   <flush-on-execute statement="deleteProduct"/>
   <cache-property name="cache-size" value="1000" />
</cache-model>
```

Only a single cache-property is recognized by the LRU cache implementation. This property, named 'cache-size' must be set to an integer value representing the maximum number of objects to hold in the cache at once. An important thing to remember here is that an object can be anything from a single String instance to an ArrayList of JavaBeans. So take care not to store too much in your cache and risk running out of memory!

"FIFO" (com.ibatis.db.sqlmap.cache.fifo.FifoCacheController)

The FIFO cache implementation uses an First In First Out algorithm to determines how objects are automatically removed from the cache. When the cache becomes over full, the oldest object will be removed from the cache. The FIFO cache is good for usage patterns where a particular query will be referenced a few times in quick succession, but then possibly not for some time later.

The FIFO implementation is configured as follows:

```
<cache-model name="product-cache" implementation ="FIFO">
   <flush-interval hours="24"/>
   <flush-on-execute statement="insertProduct"/>
   <flush-on-execute statement="updateProduct"/>
   <flush-on-execute statement="deleteProduct"/>
   <cache-property name="cache-size" value="1000" />
</cache-model>
```

Only a single cache-property is recognized by the FIFO cache implementation. This property, named 'cache-size' must be set to an integer value representing the maximum number of objects to hold in the cache at once. An important thing to remember here is that an object can be anything from a single String instance to an ArrayList of JavaBeans. So take care not to store too much in your cache and risk running out of memory!

"OSCACHE" (com.ibatis.db.sqlmap.cache.oscache.OSCacheController)

The OSCACHE cache implementation is a plugin for the OSCache 2.0 caching engine. It is highly configurable, distributable and flexible.

The OSCACHE implementation is configured as follows:

```
<cache-model name="product-cache" implementation ="OSCACHE">
   <flush-interval hours="24"/>
   <flush-on-execute statement="insertProduct"/>
   <flush-on-execute statement="updateProduct"/>
   <flush-on-execute statement="deleteProduct"/>
</cache-model>
```

The OSCACHE implementation does not use any cache-property elements for configuration. Instead, the OSCache instance is configured using the standard *oscache.properties* file which should be located in the root of your classpath. Within that file you can configure algorithms (much like those discussed above), cache size, persistence approach (memory, file etc.), and clustering.

Please refer to the OSCache documentation for more information. OSCache and its documentation can be found at the following Open Symphony website:

http://www.opensymphony.com/oscache/

Note: At the time of this writing, OSCache 2.0 was still in beta.

Dynamic Mapped Statements

A very common problem with working directly with JDBC is dynamic SQL. It is normally very difficult to work with SQL statements that change not only the values of parameters, but which parameters and columns are included at all. The typical solution is usually a mess of conditional if-else statements and horrid string concatenations. The desired result is often a query by example, where a query can be built to find objects that are similar to the example object. The SQL Maps API provides a relatively elegant solution that can be applied to a structure very similar to mapped statements. Here is a simple example:

```
<dynamic-mapped-statement name="dynamicGetAccountList"
                          cache-model="account-cache"
                          result-map="account-result" >

    select * from ACCOUNT

    <dynamic prepend="where">
        <isGreaterThan prepend="and" property="id" compareValue="0">
            ACC_ID = #id#
        </isGreaterThan>
    </dynamic>

    order by ACC_LAST_NAME

</dynamic-mapped-statement>
```

In the above example, there are two possible statements that could be created depending on the state of the "id" property of the parameter bean. If the id parameter is greater than 0, then the statement will be created as follows:

```
select * from ACCOUNT where ACC_ID = ?
```

Or if the id parameter is 0 or less, the statement will look as follows.

```
select * from ACCOUNT
```

The immediate usefulness of this might not become apparent until a more complex situation is encountered. For example, the following is a somewhat more complex example.

```
<dynamic-mapped-statement name="dynamicGetAccountList"
                          result-map="account-result" >
  select * from ACCOUNT
  <dynamic prepend="WHERE">
    <isNotNull prepend="AND" property="firstName">
      (ACC_FIRST_NAME = #firstName#
      <isNotNull prepend="OR" property="lastName">
        ACC_LAST_NAME = #lastName#
      </isNotNull>
      )
    </isNotNull>
    <isNotNull prepend="AND" property="emailAddress">
      ACC_EMAIL like #emailAddress#
    </isNotNull>
    <isGreaterThan prepend="AND" property="id" compareValue="0">
      ACC_ID = #id#
    </isGreaterThan>
  </dynamic>
  order by ACC_LAST_NAME
</dynamic-mapped-statement>
```

Depending on the situation, there could be as many as 16 different SQL queries generated from the above dynamic statement. To code the if-else structures and string concatenations could get quite messy and require hundreds of lines of code.

Using dynamic statements is as simple as inserting some conditional tags around the dynamic parts of your SQL. For example:

```
<dynamic-mapped-statement name="someName"
                          result-map="account-result" >
  select * from ACCOUNT
  <dynamic prepend="where">
    <isGreaterThan prepend="and" property="id" compareValue="0">
      ACC_ID = #id#
    </isGreaterThan>
    <isNotNull prepend="and" property="lastName">
      ACC_LAST_NAME = #lastName#
    </isNotNull>
  </dynamic>
  order by ACC_LAST_NAME
</dynamic-mapped-statement>
```

In the above statement, the <dynamic> element demarcates a section of the SQL that is dynamic. The dynamic section can contain any number of conditional elements (see below) that will determine whether a the contained SQL code will be included in the statement. All of the conditional elements work based on the state of the parameter object passed into the query. Both the dynamic element and the conditional elements have a "prepend" attribute. The prepend attribute is a part of the code that is free to be overridden by the a parent element's prepend if necessary. In the above example the "where" prepend will override the first true conditional prepend. This is necessary to ensure that the SQL statement is built properly. For example, in the case of the first true condition, there is no need for the AND, and in fact it would break the statement. The following sections describe the various kinds of elements, including Binary Conditionals, Unary Conditionals and Iterate.

Binary Conditional Elements

Binary conditional elements compare a property value to a static vale or another property value. If the result is true, the body content is included in the SQL query.

Binary Conditional Attributes:

prepend – the overridable SQL part that will be prepended to the statement (optional)
property – the property to be compared (required)
compareProperty – the other property to be compared (required or compareValue)
compareValue – the value to be compared (required or compareProperty)

<isEqual>	Checks the equality of a property and a value, or another property.
<isNotEqual>	Checks the inequality of a property and a value, or another property.
<isGreaterThan>	Checks if a property is greater than a value or another property.
<isGreaterEqual>	Checks if a property is greater than or equal to a value or another property.
<isLessThan>	Checks if a property is less than a value or another property.
<isLessEqual>	Checks if a property is less than or equal to a value or another property. Example Usage: **<isLessEqual prepend="AND" property="age" compareValue="18"> ADOLESCENT = 'TRUE' </isLessEqual>**

Unary Conditional Elements

Unary conditional elements check the state of a property for a specific condition.

Unary Conditional Attributes:
prepend – the overridable SQL part that will be prepended to the statement (optional)
property – the property to be checked (required)

<isPropertyAvailable>	Checks if a property is available (i.e is a property of the parameter bean)
<isNotPropertyAvailable>	Checks if a property is unavailable (i.e not a property of the parameter bean)
<isNull>	Checks if a property is null.
<isNotNull>	Checks if a property is not null.
<isEmpty>	Checks to see if the value of a Collection, String or String.valueOf() property is null or empty (“” or size() < 1).
<isNotEmpty>	Checks to see if the value of a Collection, String or String.valueOf() property is not null and not empty (“” or size() < 1). Example Usage: **<isNotEmpty prepend=”AND” property=”firstName” > FIRST_NAME=#firstName# </isNotEmpty>**

Other Elements

Parameter Present: These elements check for parameter object existence.
Parameter Present Attributes:
prepend – the overridable SQL part that will be prepended to the statement (optional)

<isParameterPresent>	Checks to see if the parameter object is present (not null).
<isNotParameterPresent>	Checks to see if the parameter object is not present (null). Example Usage: **<isNotParameterPresent prepend=”AND”> EMPLOYEE_TYPE = ‘DEFAULT’ </isNotParameterPresent>**

Iterate: This tag will iterate over a collection and repeat the body content for each item in a List
Iterate Attributes:
prepend – the overridable SQL part that will be prepended to the statement (optional)
property – a property of type java.util.List that is to be iterated over (required)
open – the string with which to open the entire block of iterations, useful for brackets (optional)
close – the string with which to close the entire block of iterations, useful for brackets (optional)
conjunction – the string to be applied in between each iteration, useful for AND and OR (optional)

<iterate>	Iterates over a property that is of type java.util.List Example Usage: **<iterate prepend="AND" property="userNameList"** **open="(" close=")" conjunction="OR">** **username=#userNameList[]#** **</iterate>** Note: It is very important to include the square brackets[] at the end of the List property name when using the Iterate element. These brackets distinguish this object as an List to keep the parser from simply outputting the List as a string.

Simple Dynamic SQL Elements

Despite the power of the full Dynamic Mapped Statement API discussed above, sometimes you just need a simple, small piece of your SQL to be dynamic. For this, SQL mapped-statements and dynamic-mapped-statements can contain simple dynamic SQL elements to help implement dynamic *order by* clauses, dynamic select columns or pretty much any part of the SQL statement. The concept works much like inline parameter maps, but uses a slightly different syntax. Consider the following example:

```
<mapped-statement  name="getProduct" result-map="get-product-result">
        select * from PRODUCT order by $preferredOrder$
</mapped-statement>
```

In the above example the *preferredOrder* dynamic element will be replaced by the value of the *preferredOrder* property of the parameter object (just like a parameter map). The difference is that this is a fundamental change to the SQL statement itself, which is much more serious than simply setting a parameter value. A mistake made in a Dynamic SQL Element can introduce security, performance and stability risks. Take care to do a lot of redundant checks to ensure that the simple dynamic SQL elements are being used appropriately. Also, be mindful of your design, as there is potential for database specifics to encroach on your business object model. For example, you may not want a column name intended for an order by clause to end up as a property in your business object, or as a field value on your JSP page.

Simple dynamic elements can be included within <dynamic-mapped-statements> and come in handy when there is a need to modify the SQL statement itself. For example:

```
<dynamic-mapped-statement  name="getProduct" result-map="get-product-result">
        SELECT * FROM PRODUCT
        <dynamic prepend="WHERE">
          <isNotEmpty property="description">
            PRD_DESCRIPTION   $operator$   #description#
          </isNotEmpty>
        </dynamic>
</dynamic-mapped-statement>
```

In the above example the *operator* property of the parameter object will be used to replace the $operator$ token. So if the *operator* property was equal to 'like' and the *description* property was equal to '%dog%', then the SQL statement generated would be:

```
SELECT * FROM PRODUCT WHERE PRD_DESCRIPTION LIKE '%dog%'
```

Programming with SQL Maps: The API

The SQL Map API is meant to be simple and minimal. It provides the programmer with the ability to do four primary functions: configure an SQL Map, execute an SQL update, execute a query for a single object, and execute a query for a list of objects.

Configuration

Configuring an SQL Map is trivial once you have created your SQL Map XML definition files and SQL Map configuration file (discussed above). SQL Maps are built using XmlSqlMapBuilder. This class has one primary static method named buildSqlMap(). The buildSqlMap() method simply takes a Reader instance that can read in the contents of an sql-map-config.xml (not necessarily named that).

```
Reader reader = Resources.getResourceAsReader ("com/ibatis/example/sql-map-config.xml");
SqlMap sqlMap =  XmlSqlMapBuilder.buildSqlMap(reader);
```

Validation

While your developing your application, it is probably quite inconvenient to start up your app server just to test your SQL Maps. For this reason, a separate utility is provided for validating SQL Maps. This validator can be run from within the Ant build tool for simple build-time SQL Map validation.

You can specify as many SQL map config files as you like. By default the validator will look for the files relative to the classpath. If you want to specify a full file path, you can do so by simply prefixing the resource path with "file:". Here are a few examples:

```
java com.ibatis.db.sqlmap.XmlSqlMapValidator  SqlMapConfig1.xml  SqlMapConfig2.xml
java com.ibatis.db.sqlmap.XmlSqlMapValidator  com/domain/app/sql/SqlMapConfig1.xml
java com.ibatis.db.sqlmap.XmlSqlMapValidator  file:C:/devt/sql/SqlMapConfig1.xml
```

As mentioned, you can also run the validator from within an Ant build script. Here's an example target using the <java> task:

```
<target name="validateMaps" depends="compile">
 <java classpathref="classpath" classname="com.ibatis.db.sqlmap.XmlSqlMapValidator" >
   <arg value="examples/sqlmap/maps/SqlMapConfigExample.xml"/>
   <arg value="file:C:/devt/sql/SqlMapConfig1.xml"/>
 </java>
</target>
```

Note! The validator currently does not throw an exception in the case of errors. Watch the build log!

Setting the Current DataSource

Although an SQL Map can have many DataSource configurations, each instance of SQL Map may only actively use a single DataSource at a time. To use more than one, you must instantiate a separate SQL Map instance. You can set the current DataSource by calling the following method on the SqlMap class:

```
public void setCurrentDataSource (String dataSourceName) throws SqlMapException
```

The default DataSource can be specified in the SQL Map Configuration file by specifying default="true" in the <datasource> tag. For example:

```
<datasource
        name="somedb"
        factory-class="com.ibatis.db.sqlmap.datasource.JndiDataSourceFactory"
        default="true" />
```

Transactions

By default, calling any executeXxxx() method on an SqlMap instance will auto-commit/rollback. This means that each call to executeXxxx() will be a single unit of work. This is simple indeed, but not ideal if you have a number of statements that must execute as a single unit of work (i.e. either succeed or fail as a group). This is where transactions come into play.

If you're using Global Transactions (configured by the SQL Map configuration file), you can use auto-commit and still achieve unit-of-work behaviour. However, it still might be ideal for performance reasons to demarcate transaction boundaries, as it reduces the traffic on the connection pool and database connection initializations.

The SqlMap has methods that allow you to demarcate transactional boundaries. A transaction can be started, committed and/or rolled back using the following methods on the SqlMap class:

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

By starting a transaction you are retrieving a connection from the connection pool, and opening it to receive SQL queries and updates.

An example of using transactions is as follows:

```
private Reader reader = new Resources.getResourceAsReader ("com/ibatis/example/sql-map-
config.xml");
private SqlMap sqlMap = XmlSqlMapBuilder.buildSqlMap(reader);

public updateItemDescription (String itemId, String newDescription)
      throws SQLException {
   try {
      sqlMap.startTransaction ();

      Item item =  (Item) sqlMap.executeQueryForObject ("getItem", itemId);
      item.setDescription (newDescription);
      sqlMap.executeUpdate ("updateItem", item);

      sqlMap.commitTransaction ();
   } catch (SQLException e) {
      sqlMap.rollbackTransaction ();
      throw (SQLException) e.fillInStackTrace();
   }
}
```

Note! Transactions cannot be nested. Calling .startTransaction() from the same thread more than once, before calling commit() or rollback(), will cause an exception to be thrown. In other words, each thread can have -at most- one transaction open, per SqlMap instance.

Note! SqlMap transactions use Java's ThreadLocal store for storing transactional objects. This means that each thread that calls startTransaction() will get a unique Connection object for their transaction. The only way to return a connection to the DataSource (or close the connection) is to call commitTransaction() or rollbackTransaction(). Not doing so could cause your pool to run out of connections and lock up.

Automatic Transactions

Although using explicit transactions is very highly recommended, there is a simplified semantic that can be used for simple requirements (generally read-only). If you do not explicitly demarcate transactions using the startTransaction(), commitTransaction() and rollbackTransaction() methods, they will all be called automatically for you whenever you execute a statement outside of a transactional block as demonstrated in the above. For example:

```
private Reader reader = new Resources.getResourceAsReader ("com/ibatis/example/sql-map-
config.xml");
private SqlMap sqlMap = XmlSqlMapBuilder.buildSqlMap(reader);

public updateItemDescription (String itemId, String newDescription)
      throws SQLException {
   try {
      Item item =  (Item) sqlMap.executeQueryForObject ("getItem", itemId);
      item.setDescription ("TX1");
      //No transaction demarcated, so transaction will be automatic (implied)
      sqlMap.executeUpdate ("updateItem", item);
      item.setDescription (newDescription);
      item.setDescription ("TX2");
      //No transaction demarcated, so transaction will be automatic (implied)
      sqlMap.executeUpdate ("updateItem", item);
   } catch (SQLException e) {
      throw (SQLException) e.fillInStackTrace();
   }
}
```

Note! Be very careful using automatic transactions, for although they can be attractive, you will run into trouble if your unit of work requires more than a single update to the database. In the above example, if the second call to "updateItem" fails, the item description will still be updated with the first new description of "TX1" (i.e. this is not transactional behavior).

Global (DISTRIBUTED) Transactions

The SQL Maps framework supports global transactions as well. Global transactions, also known as distributed transactions, will allow you to update multiple databases (or other JTA compliant resources) in the same unit of work (i.e. updates to multiple datasources can succeed or fail as a group).

External/Programmatic Global Transactions

You can choose to manage global transactions externally, either programmatically (coded by hand), or by implementing another framework such as the very common EJB. Using EJBs you can declaratively demarcate (set the boundaries of) a transaction in an EJB deployment descriptor. Further discussion of how this is done is beyond the scope of this document. To enable support external or programmatic global transactions, you must set the useGlobalTransactions="true" in your SQL Map configuration file (see above). When using externally controlled global transactions, the SQL Map transaction control methods are somewhat redundant, because the begin, commit and rollback of transactions will be controlled by the external transaction manager. However, there can be a performance benefit to still demarcating your transactions using the SqlMap methods startTransaction(), commitTransaction() and rollbackTransaction() (vs. allowing an automatic transaction to started and committed or rolled back). By continuing to use these methods, you will maintain a consistent programming paradigm, as well as you will be able to reduce the number of requests for connections from the connection pool. Further benefit is that in some cases you may need to change the order in which resources are closed (commitTransaction() or rollbackTransaction()) versus when the global transaction is committed. Different app servers and transaction managers have different rules (unfortunately). Other than these simple considerations, there are really no changes required to your SQL Map code to make use of a global transaction.

Managed Global Transactions

The SQL Map framework can also manage global transactions for you. If you do not want to make use of EJB or program the global transaction yourself, you can simply set the userTransactionJndiName setting in your SQL Map config XML file. The value of this setting must be the full JNDI name of where the SqlMap instance will find the UserTransaction instance. The UserTransaction instance is set in a JNDI directory by the app server for convenient access by the application code. An example setting is:

```
<settings
    useGlobalTransactions="true"
    userTransactionJndiName =”java:comp/UserTransaction”
/>
```

Notice that you must also set useGlobalTransactions=”true” to enable global transactions.

Programming for global transactions is not much different, however there are some small considerations. Here is an example:

```
try {
  orderSqlMap.startTransaction();
  storeSqlMap.startTransaction();

  orderSqlMap.insertOrder(…);
  orderSqlMap.updateQuantity(…);

  storeSqlMap.commitTransaction();
  orderSqlMap.commitTransaction();
} catch (DaoException e) {
  try { storeDaoManager.rollbackTransaction(); } catch (Exception e2) { /* ignore */ }
  try { orderDaoManager.rollbackTransaction(); } catch (Exception e2) { /* ignore */ }
  throw ((DaoException) e.fillInStackTrace());
}
```

In this example, there are two SqlMap instances that we will assume are using two different databases. The first SqlMap (orderSqlMap) that we use to start a transaction will also start the global transaction. After that, all other activity is considered part of the global transaction until that same SqlMap (orderSqlMap) calls commitTransaction() or rollbackTransaction(), at which point the global transaction is committed and all other work is considered done.

Warning! Although this seems simple, it is very important that you don't overuse global (distributed) transactions. There are performance implications, as well as additional complex configuration requirements for your application server and database drivers. Although it looks easy, you might still experience some difficulties. Remember, EJBs have a lot more industry support and tools to help you along, and you still might be better off using EJBs for any work that requires distributed transactions. The JPetStore example app found at www.ibatis.com is an example usage of SQL Map global transactions.

Batches

If you have a great number of non-query (insert/update/delete) statements to execute, you might like to execute them as a batch to minimize network traffic and allow the JDBC driver to perform additional optimization (e.g. compression). Using batches is simple with the SQL Map API, two simple methods allow you to demarcate the boundaries of the batch:

```
sqlMap.startBatch();
//…execute statements in between
sqlMap.endBatch();
```

Upon calling endBatch(), all batched statements will executed through the JDBC driver.

Executing Statements via the SqlMap Class API

The SqlMap class provides an API to execute all mapped statements associated to it. These methods are as follows:

```
public int executeUpdate(String statementName, Object parameterObject)
       throws SQLException

public Object executeQueryForObject(String statementName,
       Object parameterObject)
       throws SQLException

public Object executeQueryForObject(String statementName,
       Object parameterObject, Object resultObject)
       throws SQLException

public List executeQueryForList(String statementName, Object parameterObject)
       throws SQLException

public PaginatedList executeQueryForPaginatedList(String statementName,
       Object parameterObject, int pageSize)
       throws SQLException

public List executeQueryForList(String statementName, Object parameterObject,
       int skipResults, int maxResults)
       throws SQLException

public Map executeQueryForMap (String statementName, Object parameterObject,
       String keyProperty)
       throws SQLException

public Map executeQueryForMap (String statementName, Object parameterObject,
       String keyProperty, String valueProperty)
       throws SQLException

public List executeQueryWithRowHandler (String statementName,
       Object parameterObject, RowHandler rowHandler)
       throws SQLException
```

In each case a the *name* of the Mapped Statement is passed in as the first parameter. This name corresponds to the name attribute of the <mapped-statement> element described above. In addition, a parameter object can always be optionally passed in. A null parameter object can be passed if no parameters are expected, otherwise it is required. For the most part the similarities end there. The remaining differences in behavior is outlined below.

executeUpdate(): This is the only method specifically meant for update statements (a.k.a. non-query). That said, it's not impossible to execute an update statement using one of the query methods below, however this is an odd semantic and obviously driver dependent. In the case of executeUpdate(), the statement is simply executed and the number of rows effected is returned.

executeQueryForObject(): There are two versions of executeQueryForObject(), one that returns a newly allocated object, and another that uses a pre-allocated object that is passed in as a parameter. The latter is useful for objects that are populated by more than one statement.

executeQueryForList(): There are also two versions of executeQueryForList(). The first executes a query and returns all of the results from that query. The second allows for specifying a particular number of results to be skipped (i.e. a starting point) and also the maximum number of records to return. This is valuable when dealing with extremely large data sets that you do not want to return in their entirety.

executeQueryForPaginatedList(): This very useful method returns a list that can manage a subset of data that can be navigated forward and back. This is commonly used in implementing user interfaces that only display a subset of all of the available records returned from a query. An example familiar to most would be a web search engine that finds 10,000 hits, but only displays 100 at a time. The PaginatedList interface includes methods for navigating through pages (nextPage(), previousPage(), gotoPage()) and also checking the status of the page (isFirstPage(), isMiddlePage(), isLastPage(), isNextPageAvailable(), isPreviousPageAvailable(), getPageIndex(), getPageSize()). Although the total number of records available is not accessible from the PaginatedList interface, this should be easily accomplished by simply executing a second statement that counts the expected results. Too much overhead would be associated with the PaginatedList otherwise.

executeQueryForMap(): This method provides an alternative to loading a collection of results into a list. Instead it loads the results into a map keyed by the parameter passed in as the keyProperty. For example, if loading a collection of Employee objects, you might load them into a map keyed by the employeeNumber property. The value of the map can either be the entire employee object, or another property from the employee object as specified in the optional second parameter called valueProperty. For example, you might simply want a map of employee names keyed by the employee number. Do not confuse this method with the concept of using a Map type as a result object. This method can be used whether the result object is a JavaBean or a Map (or a primitive wrapper, but that would likely be useless).

executeQueryWithRowHandler(): Finally there is executeQueryWithRowHandler(). This method allows you to process result sets row by row but using the result object rather than the usual columns and rows. The method is passed the typical name and parameter object, but it also takes a RowHandler. The row handler is an instance of a class that implements the RowHandler interface. The RowHandler interface has only one method as follows:

```
public void handleRow (Object object);
```

This method will be called on the RowHandler for each row returned from the database. This is a very clean, simple and scalable way to process results of a query. For an example usage of RowHandler, see the examples section below.

Note! The primary difference between coding to the SqlMap API and the MappedStatement API described below is the management of transactions. In the case of the SqlMap API, transactions can be managed for you. If you choose to use the MappedStatement API you will have to manually code and pass a Connection instance into each method call. See the MappedStatement discussion below.

Flushing the Cache

The SqlMap class also provides functionality to flush the MappedStatement results caches. Calling the flushCache() method will result in all cached results from all MappedStatements to be flushed.

Optional PreparedStatement Caching

The JDBC API allows PreparedStatements to be reused. Depending on your database driver, this can either be a blessing or a curse. In some cases performance can be improved dramatically (e.g. Oracle), while in others it can actually slow down your application (e.g PostgreSQL). In some the difference can be inconsistent or make no noticeable difference. For this reason, prepared statements are not cached by default. To enable prepared statement caching, simply set the cache size to a value greater than zero (0) by using the setting in the SQL Map configuration file discussed earlier in this document, or by simply calling the following method:

```
sqlMap.setStatementCacheSize(50);
```

Programming directly to Mapped Statements

Retrieving the MappedStatement

All SQL code is wrapped by an instance of MappedStatement. The MappedStatement class provides methods for executing SQL statements. If you like, you can optionally code to the MappedStatement API directly (instead of using the SqlMap class APIs). The appropriate MappedStatement instance can be retrieved using the following method of the SqlMap class:

```
MappedStatement ms = sqlMap.getMappedStatement("statementName");
```

Executing the MappedStatement

Once you have a reference to your desired MappedStatement you can execute it. Depending on the type of statement that is being executed and the expected result set, you will use one of the following methods of MappedStatement:

```
public int executeUpdate(Connection conn, String statementName,
       Object parameterObject)
       throws SQLException

public Object executeQueryForObject(Connection conn, String statementName,
       Object parameterObject)
       throws SQLException

public Object executeQueryForObject(Connection conn, String statementName,
       Object parameterObject, Object resultObject)
       throws SQLException

public List executeQueryForList(Connection conn, String statementName,
       Object parameterObject)
       throws SQLException

public PaginatedList executeQueryForPaginatedList(String statementName,
       Object parameterObject, int pageSize)
       throws SQLException

public List executeQueryForList(Connection conn, String statementName,
       Object parameterObject, int skipResults, int maxResults)
       throws SQLException

public Map executeQueryForMap (Connection conn, String statementName,
       Object parameterObject, String keyProperty)
       throws SQLException

public Map executeQueryForMap (Connection conn, String statementName,
       Object parameterObject, String keyProperty, String valueProperty)
       throws SQLException

public List executeQueryWithRowHandler (Connection conn, String statementName,
       Object parameterObject, RowHandler rowHandler)
       throws SQLException
```

With the exception of executeQueryForPaginatedList(), in each case a connection and a parameter object are passed in. A null parameter object can be passed if no parameter map is expected, otherwise it is required. The primary difference between the methods is the return type. In the case of executeUpdate(), the number of rows effected is returned. In the case of executeQueryForObject(), the object requested is returned. In the case of executeQueryForList(), a List containing the requested objects is returned.

executeQueryForObject(): There are two versions of executeQueryForObject(), one that returns a newly allocated object, and another that uses a preallocated object that is passed in as a parameter. The latter is useful for objects that are populated by more than one table.

executeQueryForList(): There are also two versions of executeQueryForList(). The first executes a query and returns all of the results from that query. The second allows for specifying a particular number of results to be skipped (i.e. a starting point) and also the maximum number of records to return. This is valuable when dealing with extremely large data sets that you do not want to return in their entirety.

executeQueryForPaginatedList(): This very useful method returns a list that can manage a subset of data that can be navigated forward and back. This is commonly used in implementing user interfaces that only display a subset of all of the available records returned from a query. An example familiar to most would be a web search engine that finds 10,000 hits, but only displays 100 at a time. The PaginatedList interface includes methods for navigating through pages (nextPage(), previousPage(), gotoPage()) and also checking the status of the page (isFirstPage(), isMiddlePage(), isLastPage(), isNextPageAvailable(), isPreviousPageAvailable(), getPageIndex(), getPageSize()). Although the total number of records available is not accessible from the PaginatedList interface, this should be easily accomplished by simply executing a second statement that counts the expected results. Too much overhead would be associated with the PaginatedList otherwise. Note: When executing this method from the MappedStatement directly, you do not need to provide a connection. However, you must have a DataSource associated with the SqlMap that the mapped statement belongs to.

executeQueryForMap(): This method provides an alternative to loading a collection of results into a list. Instead it loads the results into a map keyed by the parameter passed in as the keyProperty. For example, if loading a collection of Employee objects, you might load them into a map keyed by the employeeNumber property. The value of the map can either be the entire employee object, or another property from the employee object as specified in the optional second parameter called valueProperty. For example, you might simply want a map of employee names keyed by the employee number. Do not confuse this method with the concept of using a Map type as a result object. This method can be used whether the result object is a JavaBean or a Map (or a primitive wrapper, but that would likely be useless).

executeQueryWithRowHandler(): The behavior of this method is discussed in the SQL Map API above. An example usage of this method and the RowHandler interface is below.

Note!: Cache-models are ignored for executeQueryForObject if a preallocated Result Object is specified.

Note! The primary difference between coding to the SqlMap API described above and the MappedStatement API, is the management of transactions. In the case of the MappedStatement API you will have to manually code and pass connections into each method call, whereas the SqlMap API manages transactions (connections) for you. See the SqlMap API discussion above for more information.

Example 1: Executing Update (insert, update, delete)

Using the SqlMap API

```
sqlMap.startTransaction();

Product product = new Product();
product.setId (1);
product.setDescription ("Shih Tzu");

int rows = stmt.executeUpdate ("insertProduct", product);

sqlMap.commitTransaction();
```

Using the MappedStatement API

```
Connection conn = simpleDataSource.getConnection();

Product product = new Product();
product.setId (1);
product.setDescription ("Shih Tzu");

MappedStatement stmt = sqlMap.getMappedStatement ("insertProduct");
int rows = stmt.executeUpdate (conn, product);

conn.close();
```

Example 2: Executing Query for Object (select)

Using the SqlMap API

```
sqlMap.startTransaction();

Integer key = new Integer (1);

Product product = (Product) stmt.executeQueryForObject ("getProduct",
                                                        key);
sqlMap.commitTransaction();
```

Using the MappedStatement API

```
Connection conn = simpleDataSource.getConnection();

Integer key = new Integer (1);

MappedStatement stmt = sqlMap.getMappedStatement ("getProduct");
Product product = (Product) stmt.executeQueryForObject (conn, key);
conn.close();
```

Example 3: Executing Query for Object (select) With Preallocated Result Object

Using the SqlMap API

```
sqlMap.startTransaction();

Customer customer = new Customer();

sqlMap.executeQueryForObject("getCust", parameterObject, customer);
sqlMap.executeQueryForObject("getAddr", parameterObject, customer);

sqlMap.commitTransaction();
```

Using the MappedStatement API

```
Connection conn = simpleDataSource.getConnection();

MappedStatement getCustomer = sqlMap.getMappedStatement("getCust");
MappedStatement getAddress = sqlMap.getMappedStatement("getAddr");

Customer customer = new Customer();

getCustomer.executeQueryForObject(conn, parameterObject, customer);
getAddress.executeQueryForObject(conn, parameterObject, customer);

conn.close();
```

Example 4: Executing Query for List (select)

Using the SqlMap API

```
sqlMap.startTransaction();

List list = sqlMap.executeQueryForList ("getProductList", null);

sqlMap.commitTransaction();
```

Using the MappedStatement API

```
Connection conn = simpleDataSource.getConnection();

MappedStatement stmt = sqlMap.getMappedStatement ("getProductList");
List list = stmt.executeQueryForList (conn, null);

conn.close();
```

Example 5: Auto-commit

Using the SqlMap API

```
// When No startTransaction is not called, the statements will
// auto-commit.  Calling commit/rollback is not needed.
int rows = stmt.executeUpdate ("insertProduct", product);
```

Example 6: Executing Query for List (select) With Result Boundaries

Using the SqlMap API

```
sqlMap.startTransaction();

List list = stmt.executeQueryForList ("getProductList", null, 0, 40);

sqlMap.commitTransaction();
```

Using the MappedStatement API

```
Connection conn = simpleDataSource.getConnection();

MappedStatement stmt = sqlMap.getMappedStatement ("getProductList");
List list = stmt.executeQueryForList (conn, null, 0, 40);

conn.close();
```

Example 7: Executing Query with a RowHandler (select)

An Example RowHandler Implementation

```
public class MyRowHandler implements RowHandler {

  public void handleRow (Object object) throws SQLException {
    Product product = (Product) object;
    product.setQuantity (10000);
    sqlMap.executeUpdate ("updateProduct", product);
  }

}
```

Using the SqlMap API

```
sqlMap.startTransaction();

RowHandler rowHandler = new MyRowHandler();
stmt.executeQueryWithRowHandler ("getProductList", null, rowHandler);

sqlMap.commitTransaction();
```

Using the MappedStatement API

```
Connection conn = simpleDataSource.getConnection();

MappedStatement stmt = sqlMap.getMappedStatement ("getProductList");

RowHandler rowHandler = new MyRowHandler();
stmt.executeQueryWithRowHandler (conn, null, rowHandler);

conn.close();
```

Example 8: Executing Query for Paginated List (select)

Using the SqlMap API

```
PaginatedList list = stmt.executeQueryForPaginatedList
                                    ("getProductList", null, 10);
list.nextPage();
list.previousPage();
```

Using the MappedStatement API

```
if (simpleDataSource.getCurrentDataSource() == null) {
  // you don't need to do this, it's just to make a point
  throw new RuntimeException ("No DataSource defined for SqlMap.");
}

MappedStatement stmt = sqlMap.getMappedStatement ("getProductList");
PaginatedList list = stmt.executeQueryForPaginatedList (null, 10);

list.nextPage();
list.previousPage();
```

Example 9: Executing Query for Map

Using the SqlMap API

```
sqlMap.startTransaction();

Map map = stmt.executeQueryForMap ("getProductList", null,
                                    "productCode");

sqlMap.commitTransaction();

Product p = (Product) map.get("EST-93");
```

Using the MappedStatement API

```
Connection conn = simpleDataSource.getConnection();

MappedStatement stmt = sqlMap.getMappedStatement ("getProductList");

Map map = stmt.executeQueryForMap (conn, null, "productCode");

conn.close();

Product p = (Product) map.get("EST-93");
```

Logging SqlMap Activity with Jakarta Commons Logging

The SqlMap framework provides logging information through the use of Jakarta Commons Logging (JCL – *NOT Job Control Language!*). This JCL framework provides logging services in an implementation independent way. You can "plug-in" various logging providers including Log4J and the JDK 1.4 Logging API. The specifics of Jakarta Commons Logging, Log4J and the JDK 1.4 Logging API are beyond the scope of this document. However the example configuration below should get you started. If you would like to know more about these frameworks, you can get more information from the following locations:

Jakarta Commons Logging
- http://jakarta.apache.org/commons/logging/index.html

Log4J
- http://jakarta.apache.org/log4j/docs/index.html

JDK 1.4 Logging API
- http://java.sun.com/j2se/1.4.1/docs/guide/util/logging/

What will be Logged?

Currently, the following events are logged:

- MappedStatement executeUpdate(), executeQueryForObject(), and executeQueryForList() which each include the action and the name of the mapped statement
- When a statement is executed, the JDBC SQL activity will be logged as well. This includes the SQL statement, the PreparedStatement parameters and any ResultSet rows/columns that are read (only those that are read).
- Any exceptions that occur during the execution of the mapped statement.

These can each be configured independently as demonstrated below.

Log Configuration

Configuring the commons logging services is very simply a matter of including one or more extra configuration files and sometimes a new JAR file (e.g. log4j.jar). The following example configuration will configure full logging services using Log4J as a provider. There are 3 steps.

Step 1: Configure Jakarta Commons Logging.

Configuring JCL is very easy. You simply need to place a file named **commons-logging.properties** in the root of your application classpath (i.e. not in a package). The contents of the file simply configure the log factory for creating logging provider instances, and the logging providers themselves. In our case we'll use the default log factory, and include the Log4J logging provider. There are only a couple of lines to include:

commons-logging.properties

```
org.apache.commons.logging.LogFactory=org.apache.commons.logging.impl.LogFactoryImpl
org.apache.commons.logging.Log=org.apache.commons.logging.impl.Log4JLogger
```

Step 2: Add the Log4J JAR file

Because we're using Log4J, we'll need to ensure its JAR file is available to our application. Remember, Commons Logging is an abstraction API. It is not meant to provide its implementations. So to use Log4J, you need to add the JAR file to your application classpath. You can download Log4J from the URL above. For web or enterprise applications you can add the log4j.jar to your WEB-INF/lib directory, or for a standalone application you can simply add it to the JVM -classpath startup parameter.

Step 3: Configure Log4J

Configuring Log4J is simple. Like Commons Logging, you'll again be adding a properties file to your classpath root (i.e. not in a package). This time the file is called log4j.properties and it looks like the following:

log4j.properties

```
 1  # Global logging configuration
 2  log4j.rootLogger=ERROR, stdout
 3
 4  # SqlMap logging configuration...
 5  #log4j.logger.com.ibatis.db.sqlmap.MappedStatement=DEBUG, stdout
 6  #log4j.logger.java.sql.Connection=DEBUG, stdout
 7  #log4j.logger.java.sql.ResultSet=DEBUG, stdout
 8  #log4j.logger.java.sql.Statement=DEBUG, stdout
 9  #log4j.logger.java.sql.PreparedStatement=DEBUG, stdout
10
11  # Console output...
12  log4j.appender.stdout=org.apache.log4j.ConsoleAppender
13  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
14  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
15
16  # Rolling log file output...
17  log4j.appender.fileout.=org.apache.log4j.RollingFileAppender
18  log4j.appender.fileout.File=sqlmap-examples.log
19  log4j.appender.fileout.MaxFileSize=100KB
20  log4j.appender.fileout.MaxBackupIndex=1
21  log4j.appender.fileout.layout=org.apache.log4j.PatternLayout
22  log4j.appender.fileout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n
```

The above file is the minimal configuration that will cause logging to only report on errors. Line 2 of the file is what is shown to be configuring Log4J to only report errors to the stdout appender. An appender is a component that collects output (e.g. console, file, database etc.). To maximize the level of reporting, we could change line 2 as follows:

```
log4j.rootLogger=DEBUG, stdout, fileout
```

By changing line 2 as above, Log4J will now report on all logged events to both the 'stdout' appender (console) and a rolling log file appender called 'fileout'. If you want to tune the logging at a finer level, you can configure each class that logs to the system using the 'SqlMap logging configuration' section of the file above (commented out in lines 5 through 9 above). So if we wanted to log PreparedStatement activity (SQL statements) to the console at the DEBUG level, we would simply change line 9 to the following (notice it's not #commented out anymore):

```
log4j.logger.java.sql.PreparedStatement=DEBUG, stdout
```

The remaining configuration in the log4j.properties file is used to configure the appenders, which is beyond the scope of this document. However, you can find more information at the Log4J website (URL above). Or, you could simply play around with it to see what effects the different configuration options have.

The One Page JavaBeans Course

The SqlMaps framework requires a firm understanding of JavaBeans. Luckily, there's not much to the JavaBeans API as far as it relates to SqlMaps. So here's a quick introduction to JavaBeans if you haven't been exposed to them before.

What is a JavaBean? A JavaBean is a class that adheres to a strict convention for naming methods that access or mutate the state of the class. Another way of saying this is that a JavaBean follows certain conventions for "getting" and "setting" properties. The properties of a JavaBean are defined by its method definitions (not by its fields). Methods that start with the word "set" are write-able properties (e.g. setEngine), whereas methods that start with "get" are readable properties (e.g. getEngine). For boolean properties the readable property method can also start with the word "is" (e.g. isEngineRunning). Set methods should not define a return type (i.e it should be void), and should take only a single parameter of the appropriate type for the property (e.g. String). Get methods should return the appropriate type (e.g. String) and should take no parameters. Although it's usually not enforced, the parameter type of the set method and the return type of the get method should be the same. JavaBeans should also implement the Serializable interface. JavaBeans also support other features (events etc.), but these are unimportant in the context of SQL Maps and usually equally unimportant in the context of a web application.

That said, here is an example of a JavaBean:

```
public class Product implements Serializable {

  private String id;
  private String description;

  public String getId() {
     return id;
  }
  public void setId(String id) {
     this.id = id;
  }

  public String getDescription() {
     return description;
  }
  public void setDescription(String description) {
     this.description = description;
  }
}
```

Note! Don't mix data types of the get and set properties for a given property. For example, for a numeric "account" property, be sure to use the same numeric type for both the getter and setter, as follows:

```
public void setAccount (int acct) {….}
public int getAccount () {….}
```

Notice both use the "int" type. Returning a "long" from the get method, for example, would cause problems.

Note! Similarly, make sure you only have one method named getXxxx() and setXxxx(). Be judicious with polymorphic methods. You're better off naming them more specifically anyway.

Note! An alternate getter syntax exists for boolean type properties. The get methods may be named in the format of isXxxxx(). Be sure that you either have an "is" method or a "get" method, not both!

Congratulations! You've passed the course!

Data Access Objects (com.ibatis.db.dao.*)

The iBATIS Data Access Objects API can be used to help hide persistence layer implementation details from the rest of your application, and also allow a dynamic, pluggable DAO components to be swapped in and out easily. For example, you could have two implementations of a particular DAO, one that is fully cross platform and uses only SQL 92 compliant statements, and another that is fully optimized for an Oracle database (perhaps calls an Oracle stored procedure). In the picture below, you could imagine that DAOImpelementorA is the Oracle optimized version and DAOImplementorC is the cross platform version (both implement the same interface and can therefore be "plugged-in" to the application). Another example would be a DAO that provides caching services for another DAO. Depending on the situation (e.g. limited database performance vs. limited memory), either the cache DAO could be "plugged in" or the standard un-cached DAO could be used. These examples show the convenience that the DAO pattern provides, however, more important is the safety that DAO provides. The DAO pattern protects your application from possibly being tied to a particular persistence approach. In the event that your current approach becomes unsuitable (or even unavailable), you can simply create new DAO implementations to support a new approach, without having to modify any code in the other layers of your application.

Note! The DAO framework and SQLMaps Framework are completely separate and are not dependent on each other in any way. Please feel free to use either one separately, or both together.

The Components of the Data Access Objects API

There are a number of classes that make up the DAO API. Each has a very specific and important role. The following table lists the classes and a brief description. The next sections will provide more detail on how to use these classes.

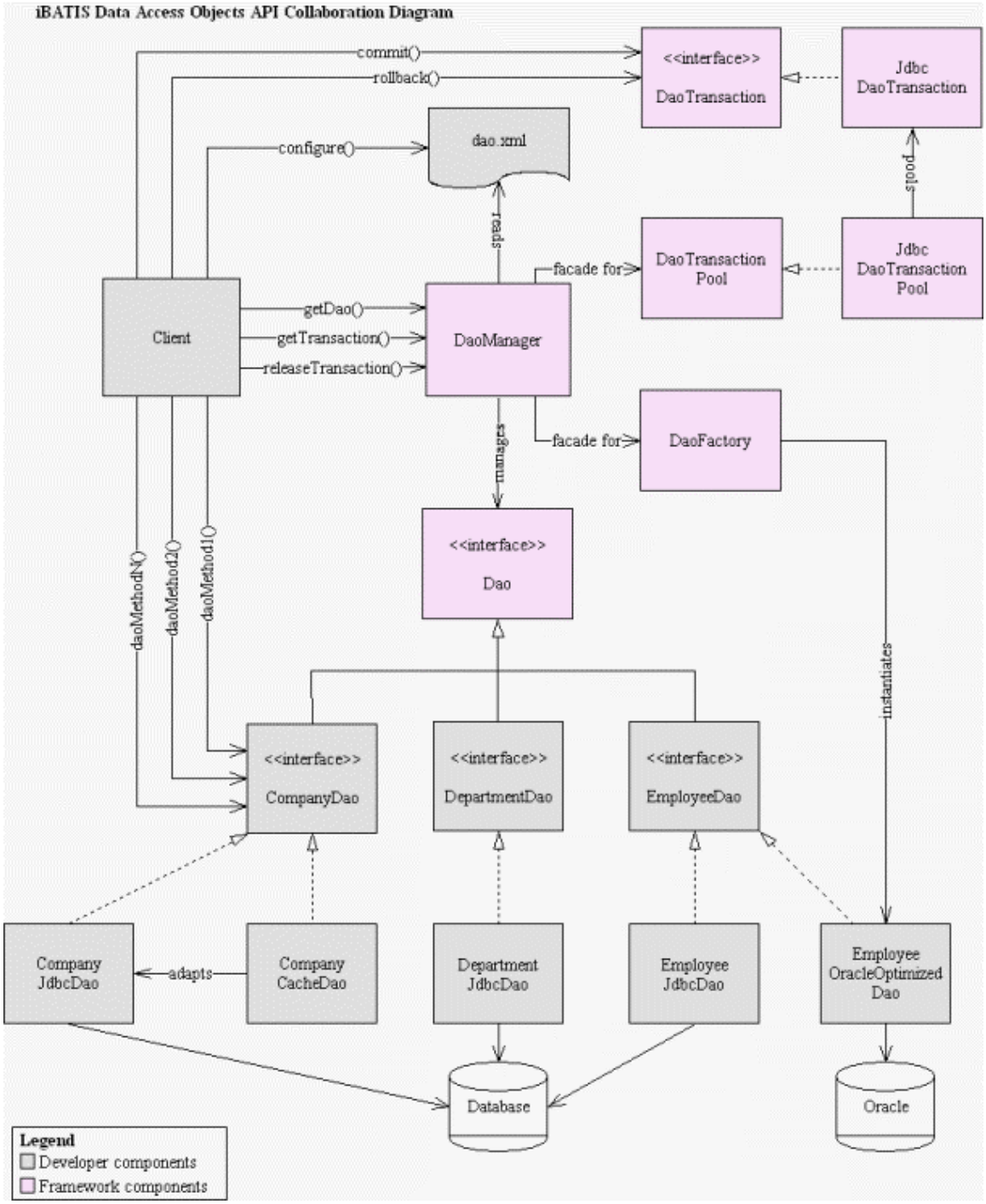
Class/Interface (Patterns)	Description
DaoManager (Façade)	Responsible for configuration of the DAO framework (via dao.xml), instantiating Dao implementations and acts as a façade to the rest of the API.
DaoTransactionPool (Interface)	A generic interface for pools of transactions (connections). Defines methods to get transactions from the pool and release transactions back to the pool. A common implementation would wrap a connection pooling component.
DaoTransaction (Interface)	A generic interface for transactions (connections). Defines methods for commit, rollback and release. A common implementation would wrap a JDBC connection object.
DaoException	All methods and classes in the DAO API throw this exception exclusively. Dao implementations should also throw this method exclusively, avoid throwing any other exception type.
Dao (Marker Interface)	A marker interface for all DAO implementations. This interface must be implemented by all DAO classes. This interface does not declare any methods to be implemented, and only acts as a marker (i.e. something for the DaoFactory to identify the class by).

JDBC Transactions and Pools

jdbc.JndiDaoTransactionPool	An implementation of DaoTransactionPool that wraps a DataSource that is returned from a JNDI context.
jdbc.SimpleDaoTransactionPool	An implementation of DaoTransactionPool that wraps an iBATIS SimpleDataSource object that supports connection pooling.
jdbc.DbcpDaoTransactionPool	An implementation of DaoTransactionPool that wraps a Jakarta DBCP BasicDataSource object that supports connection pooling.
jdbc.JdbcDaoTransaction	A reference implementation of DaoTransaction that wraps a JDBC Connection instance for use with JDBC DAO implementations.

SqlMap Transactions and Pools

jdbc.SqlMapDaoTransactionPool	An implementation of DaoTransactionPool that allows an SqlMap to be "plugged-in" to the DAO framework.
jdbc.SqlMapDaoTransaction	The DaoTransaction implementation used with SqlMapDaoTransactionPool. The SqlMapDaoTransaction wraps the actual SqlMap so DAOs can access it for reading and writing to the database.



dao.xml –The Configuration File (http://www.ibatis.com/dtd/dao.dtd)

The DaoManager class is responsible for configuration of the DAO Framework. The DaoManager is able to parse a special XML file with configuration information for the framework. The configuration XML file specifies the following things: 1) DAO contexts, 2) the DaoTransactionPool implementation for each context, 3) properties for configuration of the DaoTransactionPool, 3) the named Dao implementations for each context. A DAO context is a grouping of related configuration information and DAO implementations. Usually a context is associated with a single database. By configuring multiple contexts, you can easily centralize access configuration to multiple databases. The structure of the DAO configuration file (commonly called dao.xml, but not required) is as follows. Values that you will likely change for your application are highlighted.

```
<dao-config>

 <properties resource="properties/dao.properties"/>

 <context name="JPetStore" default="true">
  <transaction-pool implementation="com.ibatis.common.dao.jdbc.JdbcDaoTransactionPool">
      <property name="JDBC.Driver" value="org.postgresql.Driver"/>
      <property name="JDBC.ConnectionURL" value="jdbc:postgresql://localhost:5432/test"/>
      <!-- ….other transaction pool configuration properties….  -->
  </transaction-pool-factory>
  <dao-factory>
      <dao name="Product" implementation="com.website.application.dao.map.ProductMapDao"/>
      <dao name="Order" implementation="com.website.application.dao.map.OrderMapDao"/>
      <!-- ….other named DAO implementations….  -->
  </dao-factory>
 <extra-properties>
      <property name="an extra property" value="a useful value" />
      <property name="some other extra property" value="another useful value" />
      <!-- ….other extra properties….  -->
  </extra-properties>

 </context>
 <!-- ….other DAO contexts….  -->
</dao-config>
```

In the example above, what we end up with is a single DAO context that can be referred to by name ("JPetStore"). In addition, because we specified this DAO context as the default context (default="true"), this DAO context can be accessed without specifying the name. There can be only one default DAO context. The default DAO context can come in useful when your application only requires a single context. If your application needs more than one context to support multiple databases, then it is recommended to always refer to each context explicitly by name in your application code.

In order to manage multiple configurations for different environments (DEVT, Q/A, PROD), you can also make use of the optional <properties> element that allows you to use placeholders in the place of literal value elements. For example, if you have a properties file with the following values:

```
driver= org.postgresql.Driver
url = jdbc:postgresql://localhost:5432/test
```

Then in the dao.xml file you can use placeholders instead of explicit values. For example (as above):

```
<property name="JDBC.Driver" value="${driver}"/>
<property name="JDBC.ConnectionURL" value="${url}"/>
```

This allows for easier management of different environment configurations. Only one properties resource may be specified per dao.xml.

Within the context (continuing with the example above), the configuration states that we will be using a transaction implementation called "com.ibatis.common.dao.jdbc.JdbcDaoTransactionPool". This particular implementation has certain properties that it needs for configuration, which are specified in the property elements in the body of the transaction-pool element. Different transaction pool implementations will require different properties. See below for more about the JdbcDaoTransactionPool.

Next in the context, is the configuration of the DaoFactory. These mappings map meaningful names to DAO implementations. This is where we can see the "pluggable" nature of Data Access Objects. By simply replacing the implementation class for a given DAO mapping, the persistence approach taken can be completely changed (e.g. from a PreparedStatement to a stored proc via CallableStatement).

Finally, the DAO configuration file allows you to define any extra properties that might be useful for your particular needs. These are defined in the "extra-properties" element in typical name-value pairs.

Note: A common use for extra properties (as the example suggests) is to specify configuration information to other frameworks or APIs such as O/R mappers that are used by your DAOs.

Once you have your configuration XML it needs to be read by the DaoManager. The next section describes how this is done, as well as introduces the rest of the DaoManager methods.

DaoTransactionPool Implementations and Configuration

The DaoTransactionPool implementation is the component that will manage the pool of transaction objects. A DaoTransactionPool object is usually just a wrapper around a specific implementation of a connection pool such as a DataSource implementation from a specific database vendor. Similarly the DaoTransaction object usually wraps a particular implementation such as a JDBC Connection instance.

There are currently four implementations of DaoTransactionPool that come with the iBATIS Database Layer: DbcpDaoTransactionPool, SimpleDaoTransactionPool, JndiDaoTransactionPool and SqlMapDaoTransactionPool. More details are provided for each below.

DbcpDaoTransactionPool

This implementation uses Jakarta DBCP (Database Connection Pool) to provide connection pooling services via the DataSource API. This TransactionPool is ideal where the application/web container cannot provide a DataSource implementation, or you're running a standalone application. The configuration parameters that must be specified in the dao.xml file are as follows:

```
<transaction-pool implementation="com.ibatis.db.dao.jdbc.DbcpDaoTransactionPool">
   <property name="JDBC.Driver" value="${driver}"/>
   <property name="JDBC.ConnectionURL" value="${url}"/>
   <property name="JDBC.Username" value="${username}"/>
   <property name="JDBC.Password" value="${password}"/>
   <!-- The following are optional -->
   <property name="Pool.MaximumActiveConnections" value="10"/>
   <property name="Pool.MaximumIdleConnections" value="5"/>
   <property name="Pool.MaximumWait" value="60000"/>
   <!-- Use of the validation query can be problematic.  If you have difficulty, try without it.  -->
   <property name="Pool.ValidationQuery" value="select * from ACCOUNT"/>
   <property name="Pool.LogAbandoned" value="false"/>
   <property name="Pool.RemoveAbandoned" value="false"/>
   <property name="Pool.RemoveAbandonedTimeout" value="50000"/>
</datasource>
```

SimpleDaoTransactionPool

This implementation wraps a very simple connection pool (SimpleDataSource) and provides access to a pool of standard JDBC Connection objects (wrapped by JdbcDaoTransaction). The SimpleDaoTransactionPool is usually used where the application/web container cannot provide a DataSource implementation. The configuration parameters that must be specified in the dao.xml file are the same as for SimpleDataSource, as follows:

```
<transaction-pool implementation="com.ibatis.db.dao.jdbc.SimpleDaoTransactionPool">
  <property name="JDBC.Driver" value="org.postgresql.Driver"/>
  <property name="JDBC.ConnectionURL" value="jdbc:postgresql://server:5432/dbname"/>
  <property name="JDBC.Username" value="user"/>
  <property name="JDBC.Password" value="password"/>
  <!-- The following are optional -->
  <property name="Pool.MaximumActiveConnections" value="10"/>
  <property name="Pool.MaximumIdleConnections" value="5"/>
  <property name="Pool.MaximumCheckoutTime" value="120000"/>
  <property name="Pool.TimeToWait" value="10000"/>
  <property name="Pool.PingQuery" value="select * from dual"/>
  <property name="Pool.PingEnabled" value="false"/>
  <property name="Pool.PingConnectionsOlderThan" value="0"/>
  <property name="Pool.PingConnectionsNotUsedFor" value="0"/>
  <property name="Pool.QuietMode" value="true"/>
</transaction-pool>
```

JndiDaoTransactionPool

This implementation wraps a DataSource implementation that is returned from a JNDI context. This is typically used when an application server is in use and a container managed connection pool and associated DataSource implementation are provided. The standard way to access a JDBC DataSource implementation is via a JNDI context. JndiDaoTransactionPool provides functionality to access such a DataSource via JNDI. The configuration parameters that must be specified in the dao.xml file are as follows:

```
<transaction-pool implementation="com.ibatis.db.dao.jdbc.JndiDataSourceDaoTransactionPool">
  <property name="DBFullJndiContext" value="java:comp/env/jdbc/jpetstore"/>
  <!-- The following can be used instead of DBFullJndiContext -->
  <property name="DBInitialContext" value="java:comp/env"/>
  <property name="DBLookup" value="/jdbc/jpetstore"/>
</transaction-pool>
```

SqlMapDaoTransactionPool

This is by far the most powerful DaoTransactionPool implementation for use with the SQL Maps API. If your DAOs are wrapping SQL Map calls, you will want to consider this transaction pool. This will allow the SQL Map framework to configure the data source and manage the transactions rather than having to pass transaction instances around between methods. This transaction pool implementation requires only one property as follows:

```
<transaction-pool implementation="com.ibatis.db.dao.jdbc.SqlMapDaoTransactionPool">
  <property name="sql-map-config-file" value="examples/sql-map-config.xml"/>
</transaction-pool>
```

DaoManager -Programming

The DaoManager class is responsible for configuration of the DAO framework (via dao.xml described above). In addition, the DaoManager acts as a central façade to the rest of the DAO API. Particularly it provides methods that allow you to access the DaoFactory and the DaoTransactionPool.

Reading the Configuration File

The dao.xml file is read by the static configure() method of the DaoManager class. The configure() method takes a single Reader instance as a parameter parameter, which can be a simple FileReader that points to a dao.xml file. For example:

```
DaoManager.configure(new FileReader(C:/myapp/dao.xml));
```

More likely though, in a Web application environment (or otherwise), it is typical to load such configuration files from the Classpath. This allows the application to be moved around without having to modify properties to accommodate for the new location (i.e. achieves location transparency). This is simple to do using the com.ibatis.common.resources.Resources class that is provided with the iBATIS Database Layer. For example:

```
DaoManager.configure(Resources.getResourceAsReader("properties/dao.xml"));
```

Getting the DaoManager Instance for a DAO Context

When you call the configure() method, each DAO context you define in your DAO configuration file results in a corresponding DaoManager instance being created and registered with DaoManager. You can then use the various implementations of the DaoManager.getInstance() method. Be sure to retrieve the correct DaoManager instance for the context that you are interested in! You can get a DaoManager instance by calling either getInstance() or getInstance(String name) or getInstance (Dao dao). The parameterless version of getInstance() returns the DaoManager instance associated with the "default" context, as specified in the DAO configuration file. The getInstance(String name) can be called to explicitly return the DaoManager instance associated with the context for the specified name. Finally, the getInstance (Dao dao) method returns the DaoManager that instantiated the DAO provided as the parameter (useful for getting the DaoManager instance from within Dao implementations i.e. reverse lookup). Here are some examples:

```
DaoManager daoManager = DaoManager.getInstance();
```

- OR -

```
DaoManager daoManager = DaoManager.getInstance("JPetStore");
```

- OR -

```
DaoManager daoManager = DaoManager.getInstance(customerJdbcDao);
```

Custom Configuration

There is an alternative to using the centralized (static) configuration management of the DaoManager class. Some environments may not be suited to using the configure()/getInstance() approach described above. So, if you like, you can simply acquire separate instances of the DaoManagers defined in a dao.xml file by using the XmlDaoManagerBuilder class. The API is very similar to the XmlSqlMapBuilder class. The main difference is that a DAO configuration can result in more than one instance of DaoManager, so the build method returns an array of DaoManagers (one for each <context> in the dao.xml file). For example:

```
DaoManager[] daoManagers = XmlDaoManagerBuilder.buildDaoManagers("properties/dao.xml");
```

Getting a Data Access Object

Once you acquire a DaoManager instance, it will allow you to retrieve a Dao implementation by name (as specified in the dao.xml in the dao-factory section). Getting a DataAccess object is simply a matter of calling the getDao() method of a DaoManager instance. For example:

```
ProductDao productDao = (ProductDao) MydaoManager.getDao ("Product");
```

Getting a New Transaction

The DaoManager acts as a façade for the DaoTransactionPool. This allows the client to get and release transaction instances (i.e. a connection to the database) by calling the getTransaction() and releaseTransaction() methods (or Transaction.release()). Because different DaoManager instances can be (and likely are) connecting to different databases, it is important to get the transaction from the appropriate daoManager instance (as defined by your DAO configuration file). For example:

```
DaoTransaction trans = daoManager.getTransaction();
//…queries and database state changes…selects, inserts, updates, deletes…
trans.release();
```

The iBATIS Database Layer includes implementations for DaoTransaction and DaoTransactionPool, which work together to provide basic connection pool facilities (e.g. SimpleDataSource or JNDI) and access to a typical JDBC Connection instance.

The DaoTransaction interface declares three methods: commit(), rollback() and release(). The first two (commit and rollback) methods should be used just like the Connection.commit() and Connection.rollback() methods of the JDBC API. The third (release) returns the transaction back to the pool. When programming with Data Access Objects, it is important to use a database that supports transactions. Some databases (e.g. MySQL) do not support transactions and are therefore not recommended. (They will "work", but they won't *work*.)

Using the DaoManager Transaction Management

Although you can retrieve a transaction and deal with it manually, a better way is to simply allow the DaoManager to hold the transaction for you. By using a few simple methods of the DaoManager class, you can demarcate transactions and avoid having to pass a transaction object around to all of your DAOs. For example:

```
DaoManager daoManager = DaoManager.getInstance();
try {
   daoManager.startTransaction();
   ProductDao productDao = (ProductDao) daoManager.getDao ("Product");
   Product product = productDao.getProduct (5);
   product.setDescription ("New description.");
   productDao.updateProduct(product);
   daoManager.commitTransaction();
} catch (DaoException e) {
   daoManager.rollbackTransaction();
   throw (DaoException) e.fillInStackTrace();
}
```

The big question in when allowing the DaoManager to manage your transactions is: "How do I get my transaction to my DAO?" The answer is simple. Each DAO can easily find it's own DaoManager instance by calling the DaoManager.getInstance() method and passing itself in as a parameter. For example:

```
DaoManager daoManager = DaoManager.getInstance(this);
```

Once you have the DaoManager instance, the current local transaction can be easily retrieved by calling the following method:

```
DaoTransaction trans = daoManager.getLocalTransaction();
```

A complete example of a DAO method that finds it's DaoManager and the local transaction, which in this case is a JdbcDaoTransaction (wrapped JDBC Connection) is as follows:

```
public int updateProduct (Product product) throws DaoException {
    try {
        // Find the DaoManager for this DAO (i.e. "Who's my daddy?")
        DaoManager daoManager = DaoManager.getInstance(this);

        // We know in this case it's an JdbcDaoTransaction
        JdbcDaoTransaction trans = (JdbcDaoTransaction) daoManager.getLocalTransaction();

        // JdbcDaoTransaction can give us the related Connection instance.
        Connection conn = trans.getConnection();

        //…prepare statement
        //…set params
        //…int rows = execute update
        //…etc. etc.

        // Execute and return the effected rows.
        return rows;
    } catch (SQLException e) {
        throw new DaoException ("Error executing updateProduct(). Cause: " + e);
    }

}
```

Another example that uses an SqlMapDaoTransaction is as follows:

```
public int updateProduct (Product product) throws DaoException {
    try {
        // Find the DaoManager for this DAO (i.e. "Who's my daddy?")
        DaoManager daoManager = DaoManager.getInstance(this);

        // We know in this case it's an SqlMapDaoTransaction
        SqlMapDaoTransaction trans = (SqlMapDaoTransaction) daoManager.getLocalTransaction();

        // SqlMapDaoTransaction can give us the related SqlMap instance.
        SqlMap sqlMap = trans.getSqlMap();

        // Execute and return the effected rows.
        return sqlMap.executeUpdate ("updateProduct", product);
    } catch (SQLException e) {
        throw new DaoException ("Error executing updateProduct(). Cause: " + e);
    }

}
```

Implementing the Dao Interface (i.e. Creating Your Data Access Objects)

The Dao interface is very simple and very flexible because it does not declare any methods. It is intended to act as a marker interface only (as per the Marker Interface pattern –Grand98). In other words, by extending the Dao interface, all that is really achieved for the class (that implements your Dao interface) is the ability to be instantiated by DaoFactory and managed by DaoManager. There are no limitations to the methods that you use in your Dao interfaces, however, the following method contract is recommended for all methods of a Dao:

- Must throw exceptions of type DaoException
- Must only throw exceptions of type DaoException
- Must not get DaoTransactions internally, but rather receive them as a method parameter or allow them to be managed by DaoManager
- A DaoTransaction must be passed in as the first (possibly the only) parameter of the method, or get the local transaction from the DaoManager transaction management services.

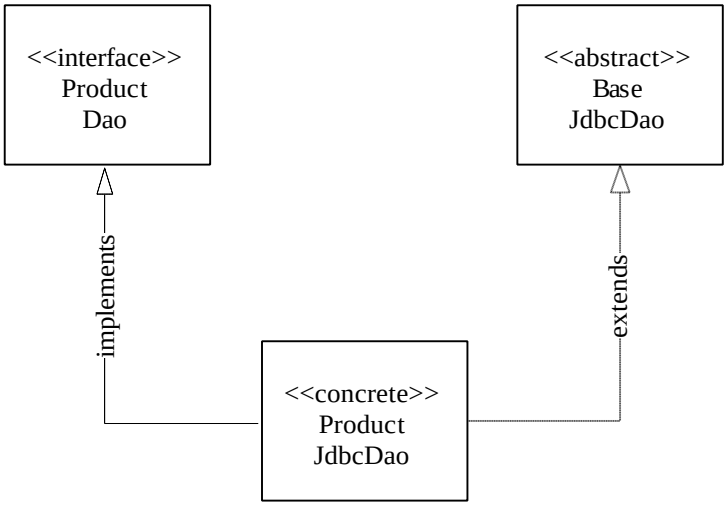
An example of a good Dao interface using manual transaction management is:

```
public interface ProductDao extends Dao {

        //Updates
        public int updateProduct (DaoTransaction trans, Product product)
           throws DaoException;
        public int insertProduct (DaoTransaction trans, Product product)
           throws DaoException;
        public int deleteProduct (DaoTransaction trans, int productId)
           throws DaoException;

        //Queries
        public Product getProduct (DaoTransaction trans, int productId)
           throws DaoException;
        public List getProductListByProductDescription (DaoTransaction trans, String name)
           throws DaoException;
}
```

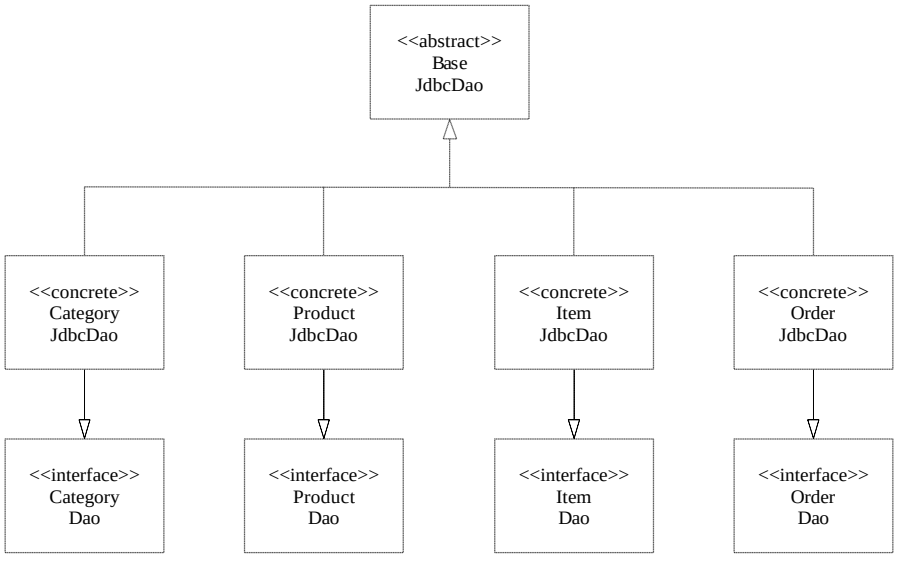
An example of a good Dao interface using the built-in transaction management of the DaoManager class (e.g. daoManager.startTransaction(),daoManager.commitTransaction() etc.) is:

```
public interface ProductDao extends Dao {

        //Updates
        public int updateProduct (Product product)
           throws DaoException;
        public int insertProduct (Product product)
           throws DaoException;
        public int deleteProduct (int productId)
           throws DaoException;

        //Queries
        public Product getProduct (int productId)
           throws DaoException;
        public List getProductListByProductDescription (String name)
           throws DaoException;
}
```

When implementing your Dao classes for your Dao interfaces, it is recommended to use a tripartite approach that includes the Dao interface, an abstract (base) class and a concrete class. The advantage to having the base class is that it can contain common methods that simplify the usage of your persistence approach (e.g. wrapping up exception handling, transaction acquisition etc.). For example:



So a complete set of Data Access Objects that use a tripartite approach would look like this:



Resources (com.ibatis.common.resources.*)

The Resources class provides methods that make it very easy to load resources from the classpath. Dealing with ClassLoaders can be challenging, especially in an application server/container. The Resources class attempts to simplify dealing with this sometimes tedious task.

Common uses of the resources file are:

- Loading the SQL Map configuration file (e.g. sql-map-config.xml) from the classpath.
- Loading the DAO Manager configuration file (e.g. dao.xml) from the classpath
- Loading various *.properties files from the classpath.
- Etc.

There are many different ways to load a resource, including:

- As a Reader: For simple read-only text data.
- As a Stream: For simple read-only binary or text data.
- As a File: For read/write binary or text files.
- As a Properties File: For read-only configuration properties files.
- As a URL: For read-only generic resources

The various methods of the Resources class that load resources using any one of the above schemes are as follows (in order):

```
Reader getResourceAsReader(String resource);
Stream getResourceAsStream(String resource);
File getResourceAsFile(String resource);
Properties getResourceAsProperties(String resource);
Url getResourceAsUrl(String resource);
```

In each case the ClassLoader used to load the resources will be the same as that which loaded the Resources class, or when that fails, the system class loader will be used. In the event you are in an environment where the ClassLoader is troublesome (e.g. within certain app servers), you can specify the ClassLoader to use (e.g. use the ClassLoader from one of your own application classes). Each of the above methods has a sister method that takes a ClassLoader as the first parameter. They are:

```
Reader getResourceAsReader (ClassLoader classLoader, String resource);
Stream getResourceAsStream (ClassLoader classLoader, String resource);
File getResourceAsFile (ClassLoader classLoader, String resource);
Properties getResourceAsProperties (ClassLoader classLoader, String resource);
Url getResourceAsUrl (ClassLoader classLoader, String resource);
```

The resource named by the *resource* parameter should be the full package name plus the full file/resource name. For example, if you have a resource on your classpath such as 'com.domain.mypackage.MyPropertiesFile.properties', you could load as a Properties file using the Resources class using the following code (notice that the resource does not start with a slash "/"):

```
String resource = "com/domain/mypackage/MyPropertiesFile.properties";
Properties props = Resources.getResourceAsProperties (resource);
```

Similarly you could load your SqlMap configuration file from the classpath as a Reader. Say it's in a simple properties package on our classpath (properties.sql-map-config.xml):

```
String resource = "properties/sql-map-config.xml";
Reader reader = Resources.getResourceAsReader(resource);
SqlMap sqlMap = XmlSqlMapBuilder.buildSqlMap(reader);
```

SimpleDataSource (com.ibatis.db.jdbc.*)

The SimpleDataSource class is a *simple* implementation of a JDBC 2.0 compliant DataSource. It supports a convenient set of connection pooling features and is completely synchronous (no spawned threads) which makes it a very lightweight and portable connection pooling solution. SimpleDataSource is used exactly like any other JDBC DataSource implementation, and is documented as part of the JDBC Standard Extensions API, which can be found here: http://java.sun.com/products/jdbc/jdbc20.stdext.javadoc/

Note!: The JDBC 2.0 API is now included as a standard part of J2SE 1.4.x

Note!: SimpleDataSource is quite convenient, efficient and effective. However, for large enterprise or mission critical applications, it is recommended that you use an enterprise level DataSource implementation (such as those that come with App Servers and commercial O/R mapping tools).

The constructor of SimpleDataSource requires a Properties parameter that takes a number of configuration properties. The following table names and describes the properties. Only the "JDBC." properties are required.

Property Name	Required	Default	Description
JDBC.Driver	Yes	n/a	The usual JDBC driver class name.
JDBC.ConnectionURL	Yes	n/a	The usual JDBC connection URL.
JDBC.Username	Yes	n/a	The username to log into the database.
JDBC.Password	Yes	n/a	The password to log into the database.
JDBC.DefaultAutoCommit	No	driver dependant	The default autocommit setting for all connections created by the pool.
Pool.MaximumActiveConnections	No	10	Maximum number of connections that can be open at any given time.
Pool.MaximumIdleConnections	No	5	The number of idle connections that will be stored in the pool.
Pool.MaximumCheckoutTime	No	20000	The maximum length of time (milliseconds) that a connection can be "checked out" before it becomes a candidate for forced collection.
Pool.TimeToWait	No	20000	If a client is forced to wait for a connection (because they are all in use), this is the maximum length of time in (milliseconds) that the thread will wait before making a repeat attempt to acquire a connection. It is entirely possible that within this time a connection will be returned to the pool and notify this thread. Hence, the thread may not have to wait as long as this property specifies (it is simply the maximum).
Pool.PingQuery	No	n/a	The ping query will be run against the database to test the connection. In an environment where connections are not reliable, it is useful to use a ping query to guarantee that the pool will always return a good connection. However, this can have a significant impact on performance. Take care in configuring the ping query and be sure to do a lot of testing.

SimpleDataSource (continued…)

Pool.PingEnabled	No	false	Enable or disable ping query. For most applications a ping query will not be necessary.
Pool.PingConnectionsOlderThan	No	0	Connections that are older than the value (milliseconds) of this property will be tested using the ping query. This is useful if your database environment commonly drops connections after a period of time (e.g. 12 hours).
Pool.PingConnectionsNotUsedFor	No	0	Connections that have been inactive for longer than the value (milliseconds) of this property will be tested using the ping query. This is useful if your database environment commonly drops connections after they have been inactive for a period of time (e.g. after 12 hours of inactivity).
Pool.QuietMode	No	true	When disabled, SimpleDataSource will log all activity (e.g. to System.out).
*Driver.**	No	N/A	Many JDBC drivers support additional features configured by sending extra properties. To send such properties to your JDBC driver, you can specify them by prefixing them with "Driver." and then the name of the property. For example, if your driver has a property called "compressionEnabled", then you can set it in the SimpleDataSource properties by setting "Driver.compressionEnabled=true". Note: These properties also work within the dao.xml and sql-map-config.xml files.

Example: Using SimpleDataSource

```
DataSource dataSource = new SimpleDataSource(props); //properties usually loaded from a file
Connection conn = dataSource.getConnection();
//…..database queries and updates
conn.commit();
conn.close();   //connections retrieved from SimpleDataSource will return to the pool when closed
```

ScriptRunner (com.ibatis.db.util.*)

The ScriptRunner class is a very useful utility for running SQL scripts that may do such things as create database schemas or insert default or test data. Rather than discuss the ScriptRunner in length, consider the following examples that shows how simple it is to use.

Example Script: initialize-db.sql

```
-- Creating Tables – Double hyphens are comment lines
CREATE TABLE SIGNON (USERNAME VARCHAR   NOT NULL, PASSWORD VARCHAR   NOT
NULL, UNIQUE(USERNAME));
-- Creating Indexes
CREATE UNIQUE INDEX PK_SIGNON ON SIGNON(USERNAME);
-- Creating Test Data
INSERT INTO SIGNON VALUES('username','password');
```

Example Usage 1: Using an Existing Connection

```
Connection conn = getConnection();  //some method to get a Connection
ScriptRunner runner = new ScriptRunner ();
runner.runScript(conn, Resources.getResourceAsReader("com/some/resource/path/initialize.sql"));
conn.close();
```

Example Usage 2: Using a New Connection

```
ScriptRunner runner = new ScriptRunner ("com.some.Driver", "jdbc:url://db", "login", "password");
runner.runScript(conn, new FileReader("/usr/local/db/scripts/ initialize-db.sql"));
```

Example Usage 2: Using a New Connection from Properties

```
Properties props = getProperties ();  // some properties from somewhere
ScriptRunner runner = new ScriptRunner (props);
runner.runScript(conn, new FileReader("/usr/local/db/scripts/ initialize-db.sql"));
```

The properties file (Map) used in the example above must contain the following properties:

```
driver=org.hsqldb.jdbcDriver
url=jdbc:hsqldb:.
username=dba
password=whatever
stopOnError=true
```

A few methods that you may find useful are:

```
// if you want the script runner to stop running after a single error
scriptRunner.setStopOnError (true);

// if you want to log output to somewhere other than System.out
scriptRunner.setLogWriter (new PrintWriter(…));
```