iBATIS SQL Maps

Developer Guide

Version 2.0

June 17, 2004

Introduction

The SQL Maps framework will help you to 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. S*implicity* is the biggest advantage of SQL Maps over other frameworks and object relational mapping tools. 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 real SQL at your fingertips.

SQL Maps (com.ibatis.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 JDBC functionality using only 20% of the code.

How does it work?

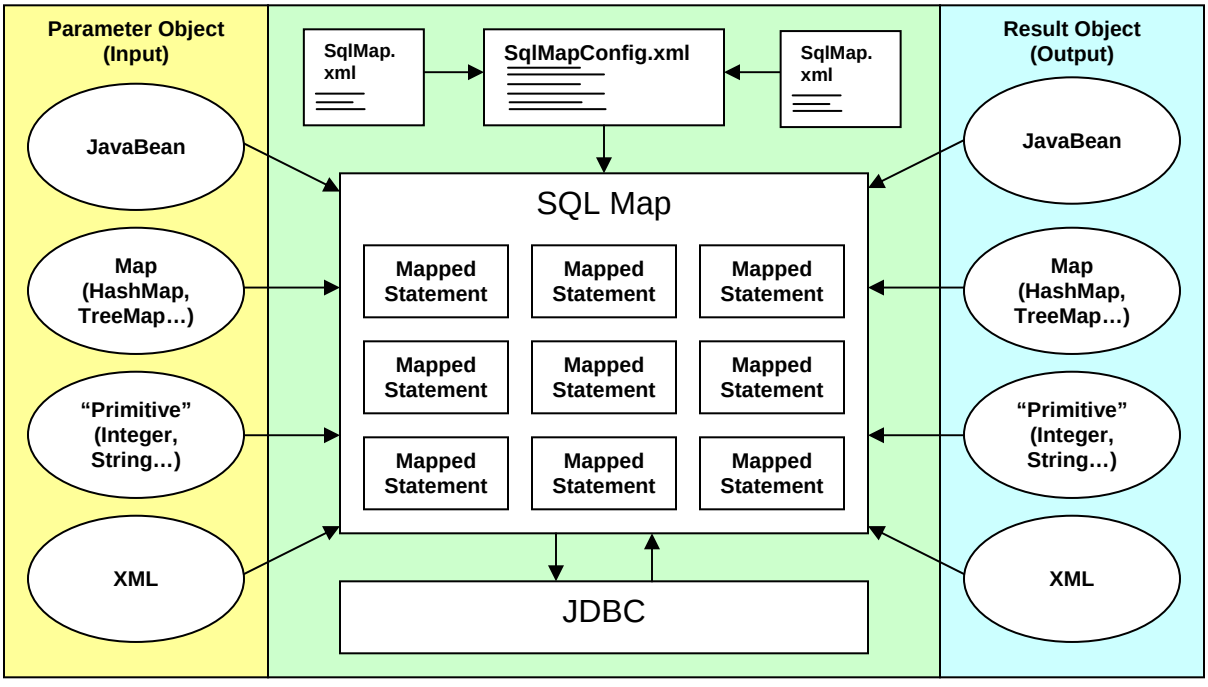
SQL Maps provides a very simple framework for using XML descriptors to map JavaBeans, Map implementations, primitive wrapper types (String, Integer…) and even XML documents to an SQL statement. The following is a high level description of the lifecycle:

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

The diagram below illustrates the flow as described.



Installation

Installing the SQL Maps framework is simply a matter of placing the appropriate JAR files on the classpath. This can either be the classpath specified at JVM startup time (*java* argument), or it could be the /WEB-INF/lib directory of a web application. A full discussion of the Java classpath is beyond the scope of this document. If you're new to Java and/or the classpath, please refer to the following resources:

http://java.sun.com/j2se/1.4/docs/tooldocs/win32/classpath.html
http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ClassLoader.html
http://java.sun.com/j2se/1.4.2/docs/

iBATIS comes with the following JAR files that should be on the classpath:

File Name	Description	Required
ibatis-common.jar	iBATIS Common Utilities	YES
ibatis-sqlmap.jar	iBATIS SQL Maps Framework	YES
ibatis-dao.jar	iBATIS Data Access Objects Framework.	NO

JAR Files and Dependencies

When a framework has too many dependencies, it makes it difficult to integrate into an application and with other frameworks. One of the key focus points of 2.0 was dependency management and reduction. Therefore, if you're running JDK 1.4, then the only real dependency is on the Jakarta Commons Logging framework. The optional JAR file libraries are organized into a package structure found in the /lib/optional directory of the distribution. They are categorized by function. The following is a summary of when you would need to use the optional packages.

Description	When to Use	Directories
Legacy JDK Support	If you're running less than JDK 1.4 and if your app server also doesn't already supply these JARs, then you will need these optional packages.	/lib/optional/**jdbc** /lib/optional/**jta** /lib/optional/**xml**
iBATIS Backward Compatibility	If you're using the old iBATIS (1.x) DAO framework, or the old SQL Maps (1.x) you can continue to do so by simply including the JAR files in this directory.	/lib/optional/**compatibility**
Runtime Bytecode Enhancement	If you want to enable CGLIB 2.0 bytecode enhancement to improve lazy loading and reflection performance.	/lib/optional/**enhancement**
DataSource Implementation	If you want to use the Jakarta DBCP connection pool.	/lib/optional/**dbcp**
Distributed Caching	If you want to use OSCache for centralized or distributed caching support.	/lib/optional/**caching**
Logging Solution	If you want to use Log4J logging.	/lib/optional/**logging**

Upgrading from 1.x

Should you Upgrade?

The best way to determine if you should upgrade is to try it. There are a few upgrade paths.

1. Version 2.0 has maintained nearly complete backward compatibility with the 1.x releases, so for some people simply replacing the JAR files might be enough. This approach yields the fewest benefits, but is also the simplest. You don't need to change your XML files or your Java code. Some incompatibilities may be found though.

2. The second option is to convert your XML files to the 2.0 specification, but continue using the 1.x Java API. This is a safe solution in that fewer compatibility issues will occur between the mapping files (there are a few). An Ant task is included with the framework to convert your XML files for you (described below).

3. The third option is to convert your XML files (as in #2) and your Java code. There is no tool for converting Java code, and therefore it must be done by hand.

4. The final option is to not upgrade at all. If you have difficulty, don't be afraid to leave your working systems on the 1.x release. It's probably not a bad idea to leave your old applications on 1.x and start only new applications on 2.0. Of course, if an old application is being heavily refactored beyond the point of recognition anyway, you might as well upgrade SQL Maps too.

Converting XML Configuration Files from 1.x to 2.x

The 2.0 framework includes an XML document converter that runs via the Ant build system. Converting your XML documents is completely optional as 1.x code will automatically transform old XML files on the fly. Still, it's a good idea to convert your files once you're comfortable with the idea of upgrading. You will experience fewer compatibility issues and you'll be able to take advantage of some of the new features (even if you're still using the 1.x Java API).

The Ant task looks like this in your build.xml file:

```
<taskdef name="convertSqlMaps"
      classname="com.ibatis.db.sqlmap.upgrade.ConvertTask"
      classpathref="classpath"/>

<target name="convert">
  <convertSqlMaps todir="D:/targetDirectory/" overwrite="true">
    <fileset dir="D/sourceDirectory/">
      <include name="**/maps/*.xml"/>
    </fileset>
  </convertSqlMaps>
</target>
```

As you can see, it works exactly like the Ant copy task, and in fact it extends the Ant copy task, so you can really do anything with this task that Copy can do (see the Ant Copy task documentation for details).

JAR Files: Out with the Old, In with the New

When upgrading, it's a good idea to remove all existing (old) iBATIS files and dependencies, and replace them with the new files. Be sure not to remove any that your other components or frameworks might still need. Note that most of the JAR files are optional depending on your circumstances. Please see the discussion above for more information about JAR files and dependencies.

The following table summarizes the old files and the new ones.

Old Files	New Files
ibatis-db.jar *After release 1.2.9b, this file was split into the following 3 files* ibatis-common.jar ibatis-dao.jar ibatis-sqlmap.jar	ibatis-common.jar ***(required)*** ibatis-sqlmap.jar ***(required)*** ibatis-dao.jar ***(optional DAO framework)***
commons-logging.jar commons-logging-api.jar commons-collections.jar commons-dbcp.jar commons-pool.jar oscache.jar jta.jar jdbc2_0-stdext.jar xercesImpl.jar xmlParserAPIs.jar jdom.jar	commons-logging-1-0-3.jar ***(required)*** commons-collections-2-1.jar ***(optional)*** commons-dbcp-1-1.jar ***(optional)*** commons-pool-1-1.jar ***(optional)*** oscache-2-0-1.jar ***(optional)*** jta-1-0-1a.jar ***(optional)*** jdbc2_0-stdext.jar ***(optional)*** xercesImpl-2-4-0.jar ***(optional)*** xmlParserAPIs-2-4-0.jar ***(optional)*** xalan-2-5-2.jar ***(optional)*** log4j-1.2.8.jar ***(optional)*** cglib-full-2-0-rc2.jar ***(optional)***

The rest of the guide will introduce you to using the SQL Maps framework.

The SQL Map XML Configuration File (http://www.ibatis.com/dtd/sql-map-config-2.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:

SqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig
    PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-config-2.dtd">

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

  <!-- 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 SqlMapClient configuration details, primarily to do with transaction
       management. They are all optional (more detail later in this document). -->
  <settings
   cacheModelsEnabled="true"
   enhancementEnabled="true"
   lazyLoadingEnabled="true"
   maxRequests="32"
   maxSessions="10"
   maxTransactions="5"
   useStatementNamespaces="false"
  />

  <!-- Type aliases allow you to use a shorter name for long fully qualified class names. -->
  <typeAlias alias="order" type="testdomain.Order"/>

  <!-- Configure a datasource to use with this SQL Map using SimpleDataSource.
       Notice the use of the properties from the above resource -->
  <transactionManager type="JDBC" >
    <dataSource type="SIMPLE">
      <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="JDBC.DefaultAutoCommit" value="true" />
      <property name="Pool.MaximumActiveConnections" value="10"/>
      <property name="Pool.MaximumIdleConnections" value="5"/>
      <property name="Pool.MaximumCheckoutTime" value="120000"/>
      <property name="Pool.TimeToWait" value="500"/>
      <property name="Pool.PingQuery" value="select 1 from ACCOUNT"/>
      <property name="Pool.PingEnabled" value="false"/>
      <property name="Pool.PingConnectionsOlderThan" value="1"/>
      <property name="Pool.PingConnectionsNotUsedFor" value="1"/>
    </dataSource>
  </transactionManager>

  <!-- 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… -->
  <sqlMap resource="examples/sqlmap/maps/Person.xml" />

</sqlMapConfig>
```

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 properties can be loaded from the classpath (use the *resource* attribute) or from any valid URL (use the *url* attribute). For example, to load a fixed path file, use:

<properties url="file:///c:/config/my.properties" />

The <settings> Element

The <settings> element allows you to configure various options and optimizations for the SqlMapClient 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:

maxRequests	This is the maximum number of threads that can execute an SQL statement at a time. Threads beyond the set value will be blocked until another thread completes execution. Different DBMS have different limits, but no database is without these limits. This should usually be at least 10 times maxTransactions (see below) and should always be greater than both maxSessions and maxTransactions. Often reducing the maximum number of concurrent requests can *increase* performance. *Example: maxRequests="256"* *Default: 512*
maxSessions	This is the number of sessions (or clients) that can be active at a given time. A session is either an explicit session, requested programmatically, or it is automatic whenever a thread makes use of an SqlMapClient instance (e.g. executes a statement etc.). This should always be greater than or equal to maxTransactions and less than maxRequests. Reducing the maximum number of concurrent sessions can reduce the overall memory footprint. *Example: maxSessions="64"* *Default: 128*

maxTransactions	This is the maximum number of threads that can enter SqlMapClient.startTransaction() at a time. Threads beyond the set value will be blocked until another thread exits. Different DBMS have different limits, but no database is without these limits. This value should always be less than or equal to maxSessions and always much less than maxRequests. Often reducing the maximum number of concurrent transactions can *increase* performance. *Example: maxTransactions="16"* *Default: 32*
cacheModelsEnabled	This setting globally enables or disables all cache models for an SqlMapClient. This can come in handy for debugging. *Example: cacheModelsEnabled="true"* *Default: true (enabled)*
lazyLoadingEnabled	This setting globally enables or disables all lazy loading for an SqlMapClient. This can come in handy for debugging. *Example: lazyLoadingEnabled="true"* *Default: true (enabled)*
enhancementEnabled	This setting enables runtime bytecode enhancement to facilitate optimized JavaBean property access as well as enhanced lazy loading. *Example: enhancementEnabled="true"* *Default: false (disabled)*
useStatementNamespaces	With this setting enabled, you must always refer to mapped statements by their fully qualified name, which is the combination of the sqlMap name and the statement name. For example: queryForObject("sqlMapName.statementName"); *Example: useStatementNamespaces="false"* *Default: false (disabled)*

The <typeAlias> Element

The *typeAlias* element simply allows you to specify a shorter name to refer to what is usually a long, fully qualified classname. For example:

```
<typeAlias alias="shortname" type="com.long.class.path.Class"/>
```

There are some predefined aliases used in the SQL Map Config file. They are:

Transaction Manager Aliases	
JDBC	com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransactionConfig
JTA	com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig
EXTERNAL	com.ibatis.sqlmap.engine.transaction.external.ExternalTransactionConfig
Data Source Factory Aliases	
SIMPLE	com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory
DBCP	com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory
JNDI	com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory

The <transactionManager> Element

1.0 Conversion Note: SQL Maps 1.0 allowed multiple datasources to be configured. This became cumbersome and introduced some bad practices. Therefore 2.0 only allows a single datasource. For multiple deployments/configurations it is recommended that you use multiple properties files that are either configured differently by the system, or passed in as a parameter when building the SQL Map (see the java API section below).

The <transactionManager> element allows you to configure the transaction management services for an SQL Map. The *type* attribute indicates which transaction manager to use. The value can either be a class name or a type alias. The three transaction managers included with the framework are: JDBC, JTA and EXTERNAL.

JDBC - This allows JDBC to control the transaction via the usual Connection commit() and rollback() methods.

JTA - This transaction manager uses a JTA global transaction such that the SQL Map activities can be included as part of a wider scope transaction that possibly involves other databases or transactional resources. This configuration requires a UserTransaction property set to locate the user transaction from a JNDI resource. *See the JNDI datasource example below for an example of this configuration.*

EXTERNAL – This allows you to manage transactions on your own. You can still configure a data source, but transactions will not be committed or rolled back as part of the framework lifecycle. This means that some part of your application external to SQL Maps must manage the transactions. This setting is also useful for non-transactional databases (e.g. read-only).

The <dataSource> Element

Included as part of the transaction manager configuration is a dataSource element and a set of properties to configure a DataSource for use with your SQL Map. There are currently three datasource factories provided with the framework, but you can also write your own. The included DataSourceFactory implementations are discussed in further detail below and example configurations are provided for each.

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. It is based on the iBATIS SimpleDataSource connection pool implementation.

```
<transactionManager type="JDBC">
  <dataSource  type="SIMPLE">
      <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"/>
      <!-- OPTIONAL PROPERTIES BELOW -->
      <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"/>
  </dataSource>
</transactionManager>
```

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:

```
<transactionManager type="JDBC">
  <dataSource type="DBCP">
    <property name="JDBC.Driver" value="${driver}"/>
    <property name="JDBC.ConnectionURL" value="${url}"/>
    <property name="JDBC.Username" value="${username}"/>
    <property name="JDBC.Password" value="${password}"/>
    <!-- OPTIONAL PROPERTIES BELOW -->
    <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>
</transactionManager>
```

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:

```
<transactionManager type="JDBC" >
   <dataSource type="JNDI">
      <property name="DataSource" value="java:comp/env/jdbc/jpetstore"/>
   </dataSource>
</transactionManager>
```

The above configuration will use normal JDBC transaction management. But with a container managed resource, you might also want to configure it for global transactions as follows:

```
<transactionManager type="JTA" >
   <property name="UserTransaction" value="java:/ctx/con/UserTransaction"/>
   <dataSource type="JNDI">
      <property name="DataSource" value="java:comp/env/jdbc/jpetstore"/>
   </dataSource>
</transactionManager>
```

Notice the *UserTransaction* property that points to the JNDI location where the UserTransaction instance can be found. This is required for JTA transaction management so that your SQL Map take part in a wider scoped transaction involving other databases and transactional resources.

The <sqlMap> Element

The sqlMap 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 SqlMapClient instance, must be declared. The SQL Map XML files will be loaded as a stream resource from the classpath or from a URL. You must specify any and all SQL Maps (as many as there are). Here are some examples:

```
<!-- CLASSPATH RESOURCES -->
<sqlMap resource="com/ibatis/examples/sql/Customer.xml" />
<sqlMap resource="com/ibatis/examples/sql/Account.xml" />
<sqlMap resource="com/ibatis/examples/sql/Product.xml" />

<!-- URL RESOURCES -->
<sqlMap url="file:///c:/config/Customer.xml " />
<sqlMap url="file:///c:/config/Account.xml " />
<sqlMap url="file:///c:/config/Product.xml" />
```

The next several sections detail the structure of these SQL Map XML files.

The SQL Map XML File (http://www.ibatis.com/dtd/sql-map-config-2.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.

```
<sqlMap   id="Product">

        <cacheModel   id="productCache"   type="LRU">
                <flushInterval hours="24"/>
                <property name="size" value="1000" />
        </cacheModel>

        <typeAlias alias="product" type="com.ibatis.example.Product" />

        <parameterMap   id="productParam" class="product">
                <parameter property="id"/>
        </parameterMap>

        <resultMap    id="productResult" class="product">
                <result property="id" column="PRD_ID"/>
                <result property="description" column="PRD_DESCRIPTION"/>
        </resultMap>

        <select  id="getProduct"  parameterMap="productParam"
                 resultMap="productResult"  cacheModel="product-cache">
            select * from PRODUCT where PRD_ID = ?
        </select>

</sqlMap>
```

TOO MUCH? Although the framework is doing a lot for you, that might seem like a lot of extra work (XML) for a simple select statement. Worry not. Here's a shorthand version of the above.

```
<sqlMap   id="Product">

    <select  id="getProduct"  parameterClass=" com.ibatis.example.Product"
                              resultClass="com.ibatis.example.Product">
        select
           PRD_ID as id,
           PRD_DESCRIPTION as description
        from PRODUCT
        where PRD_ID = #id#
    </select>

</sqlMap>
```

Now, these statements aren't exactly equal in terms of the SQL Map behavior –there are some differences. First, the latter statement does not define a cache, and therefore every request will hit the database. Second, the latter statement uses auto-mapping features of the framework, which can create some overhead. However, both of these statements would be executed *exactly* the same way from your Java code and therefore you can start with the simpler solution first and move to the more advanced mapping as needed in the future. Simplest solution first is best practice in many modern methodologies.

A single SQL Map XML file can contain as many cache models, parameter maps, result maps and statements as you like. Use discretion and organize the statements and maps appropriately for your application (group them logically).

Mapped Statements

The SQL Maps concept is centered around mapped statements. Mapped statements can be any SQL statement and can have parameter maps (input) and result maps (output). If the case is simple, the mapped statement can be configured directly to a class for parameters and results. The mapped statement can also be configured to use a cache model to cache popular results in memory.

```
<statement     id="statementName"
               [parameterClass="some.class.Name"]
               [resultClass="some.class.Name"]
               [parameterMap="nameOfParameterMap"]
               [resultMap="nameOfResultMap"]
               [cacheModel="nameOfCache"]
               >
  select * from PRODUCT where PRD_ID = [?|#propertyName#]
  order by [$simpleDynamic$]
</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:

```
<statement id="insertTestProduct" >
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (1, "Shih Tzu")
</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 describe the structure and attributes and how they effect the mapped statement.

Statement Types

The <statement> element is a general "catch all" statement that can be used for any type of SQL statement. Generally it is a good idea to use one of the more specific statement elements. The more specific elements provide a more intuitive XML DTD and sometimes provides additional features that a normal <statement> element cannot. The following table summarizes the statement elements and their supported attributes and features:

Statement Element	Attributes	Child Elements	Methods
<statement>	id parameterClass resultClass parameterMap resultMap cacheModel xmlResultName	All dynamic elements	insert update delete All query methods
<insert>	id parameterClass parameterMap	All dynamic elements <selectKey>	insert update delete
<update>	id parameterClass parameterMap	All dynamic elements	insert update delete
<delete>	id parameterClass parameterMap	All dynamic elements	insert update delete

<select>	id parameterClass resultClass parameterMap resultMap cacheModel	All dynamic elements	All query methods
<procedure>	id parameterClass resultClass parameterMap resultMap xmlResultName	All dynamic elements	insert update delete All query methods

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:

```
<statement id="getPersonsByAge" parameterClass=”int” resultClass="examples.domain.Person">
    <![CDATA[
        SELECT *
        FROM PERSON
        WHERE AGE  >  #value#
    ]]>
</statement>
```

Auto-Generated Keys

Many relational database systems support auto-generation of primary key fields. This feature of the RDBMS is often (if not always) proprietary. SQL Maps supports auto-generated keys via the <selectKey> stanza of the <insert> element. Both pre-generated keys (e.g. Oracle) and post-generated (MS-SQL Server) keys are supported. Here are a couple of examples:

```
<!—Oracle SEQUENCE Example -->
<insert id="insertProduct-ORACLE" parameterClass="com.domain.Product">
    <selectKey resultClass="int" keyProperty="id" >
        SELECT STOCKIDSEQUENCE.NEXTVAL AS ID FROM DUAL
    </selectKey>
    insert into PRODUCT (PRD_ID,PRD_DESCRIPTION)
    values  (#id#,#description#)
</insert>

<!— Microsoft SQL Server IDENTITY Column Example -->
<insert id="insertProduct-MS-SQL" parameterClass="com.domain.Product">
    insert into PRODUCT (PRD_DESCRIPTION)
    values  (#description#)
    <selectKey resultClass="int" keyProperty="id" >
        SELECT @@IDENTITY AS ID
    </selectKey>
</insert>
```

Stored Procedures

Stored procedures are supported via the <procedure> statement element. The following example shows how a stored procedure would be used with output parameters.

```
<parameterMap id="swapParameters" class="map" >
   <parameter property="email1" jdbcType="VARCHAR" javaType="java.lang.String" mode="INOUT"/>
   <parameter property="email2" jdbcType="VARCHAR" javaType="java.lang.String" mode="INOUT"/>
</parameterMap>

<procedure id="swapEmailAddresses" parameterMap="swapParameters" >
   {call swap_email_address (?, ?)}
</procedure>
```

Calling the above procedure would swap two email addresses between two columns (database table) and also in the parameter object (Map). The parameter object is only modified if the parameter mappings *mode* attribute is set to "INOUT" or "OUT". Otherwise they are left unchanged. Obviously immutable parameter objects (e.g. String) cannot be modified.

Note! Always be sure to use the standard JDBC stored procedure syntax. See the JDBC CallableStatement documentation for more information.

parameterClass

The value of the parameterClass attribute is the fully qualified name of a Java class (i.e. including package). The parameterClass attribute is optional, but highly recommended. It is used to limit parameters passed to the statement, as well as to optimize the performance of the framework. If you're using a parameterMap, there is no need to use the parameterClass attribute. 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:

```
<statement id="statementName"   parameterClass=" examples.domain.Product">
   insert into PRODUCT values (#id#, #description#, #price#)
</statement>
```

IMPORTANT: Although optional for backward compatibility, it is highly recommended to always provide a parameter class (unless of course there are no required parameters). You will achieve better performance by providing the class, because the framework is capable of optimizing itself if it knows the type in advance.

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

parameterMap

The value of the parameterMap attribute is the name of a defined parameterMap element (see below). The parameterMap attribute is rarely used in favor of the parameterClass attribute (above) and inline parameters (described below). However, this is a good approach if XML purity and consistency is your concern, or you need a more descriptive parameterMap (e.g. for stored procedures).

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

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

```
<parameterMap id="insert-product-param" class="com.domain.Product">
        <parameter property="id"/>
        <parameter property="description"/>
</parameterMap>

<statement id="insertProduct"  parameterMap="insert-product-param">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?);
</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.

A Quick Glance at Inline Parameters

Although further details are provided later in the document, here is a quick intro to inline parameters. Inline parameters can be used inside of a mapped statement. For example:

```
<statement id="insertProduct" >
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
    values (#id#, #description#);
</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');
```

resultClass

The value of the resultClass attribute is the fully qualified name of a Java class (i.e. including package). The resultClass 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:

```
<statement id="getPerson" parameterClass="int"  resultClass="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#
</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 do 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 earlier, there are some limitations of using auto-mapping with a resultClass. 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 consequence in that this approach requires accessing the ResultSetMetaData. All of these limitations can be overcome by using an explicit resultMap. Result maps are described in more detail later in this document.

resultMap

The resultMap property is one of the more commonly used and most important attributes to understand. The value of the resultMap attribute is the name of a defined resultMap element (see below). Using the resultMap 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 resultClass attribute (above), the resultMap allows you to describe the column type, a null value replacement and complex property mappings (including other JavaBeans, Collections and primitive type wrappers).

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

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

<statement id="getProduct" resultMap="get-product-result">
        select * from PRODUCT
</statement>
```

In the example above, the ResultSet from the SQL query will be mapped to a Product instance using the resultMap definition. The resultMap 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 supported. There is no need to map all of the returned columns in the ResultSet.

cacheModel

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

```
<cacheModel id="product-cache" imlementation="LRU">
   <flushInterval hours="24"/>
   <flushOnExecute statement="insertProduct"/>
   <flushOnExecute statement="updateProduct"/>
   <flushOnExecute statement="deleteProduct"/>
   <property name="size" value="1000" />
  </cacheModel>

<statement id="getProductList"   parameterClass="int"  cacheModel="product-cache">
  select * from PRODUCT where PRD_CAT_ID = #value#
</statement>
```

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

xmlResultName

When mapping results directly to an XML document, the value of the xmlResultName will be the name of the root element of the XML document. For example:

```
<select id="getPerson" parameterClass="int"  resultClass="xml" xmlResultName="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#
</select>
```

The above select statement would produce a result XML object of the following structure:

```
<person>
  <id>1</id>
  <firstName>Clinton</firstName>
  <lastName>Begin</lastName>
  <birthDate>1900-01-01</birthDate>
  <weightInKilograms>89</weightInKilograms>
  <heightInMeters>1.77</heightInMeters>
</person>
```

Parameter Maps and Inline Parameters

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

```
<parameterMap id="parameterMapName" [class="com.domain.Product"]>
        <parameter property ="propertyName" [jdbcType="VARCHAR"] [javaType="string"]
                                            [nullValue="NUMERIC"]  [null="-9999999"]/>
        <parameter …… />
        <parameter …… />
</parameterMap>
```

The parts in **[brackets]** are optional. The parameterMap itself only requires a ***id*** attribute that is an identifier that statements will use to refer to it. The ***class*** attribute is optional but highly recommended. Similar to the parameterClass attribute of a statement, the *class* attribute allows the framework to validate the incoming parameter as well as optimize the engine for performance.

<parameter> Elements

The parameterMap can contain any number of parameter mappings that map directly to the parameters of a statement. The next few sections describe the attributes of the *property* elements:

property

The property attribute of the parameter map is the name of a JavaBeans property (get method) 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).

jdbcType

The *jdbcType* attribute is used to explicitly specify the database column 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 *jdbcType* attribute of the parameterMap 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 *jdbcType* 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.

javaType

The *javaType* attribute is used to explicitly specify the Java property type of the parameter to be set. Normally this can be derived from a JavaBeans property through reflection, but certain mappings such as Map and XML mappings cannot provide the type to the framework. If the javaType is not set and the framework cannot otherwise determine the type, the type is assumed to be Object.

nullValue

The nullValue attribute can be set to 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.

A <parameterMap> Example

An example of a parameterMap that uses the full structure is as follows

```
<parameterMap   id="insert-product-param" class="com.domain.Product">
        <parameter property="id" jdbcType="NUMERIC" javaType="int" nullValue="-9999999"/>
        <parameter property="description" jdbcType="VARCHAR" nullValue="NO_ENTRY"/>
</parameterMap>

<statement   id="insertProduct"  parameterMap="insert-product-param">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?);
</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:

```
<parameterMap id="insert-product-param" class="com.domain.Product">
        <parameter property="description" />
        <parameter property="id"/>
</parameterMap>

<statement id="insertProduct"  parameterMap="insert-product-param">
    insert into PRODUCT (PRD_DESCRIPTION, PRD_ID) values (?,?);
</statement>
```

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 *id* of the Parameter Map with the *id* of the SQL Map (set in the <sqlMap> 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 parameterMaps is very verbose. There is a more popular 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 parameterMap specified will be parsed for inline parameters. The previous example (i.e. product), implemented with an inline parameter map, would look like this:

```
<statement id="insertProduct" parameterClass="com.domain.Product">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id#, #description#);
</statement>
```

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

```
<statement id="insertProduct" parameterClass="com.domain.Product">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id:NUMERIC#, #description:VARCHAR#);
</statement>
```

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

```
<statement id="insertProduct" parameterClass="com.domain.Product">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id:NUMERIC:-999999#, #description:VARCHAR:NO_ENTRY#);
</statement>
```

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

Note! If you want 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 parameterMap.

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:

```
<statement id="insertProduct" parameter="java.lang.Integer">
  select * from PRODUCT where PRD_ID = #value#
</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. The name **"value"** is simply a placeholder and can be any moniker. 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.

Primitive types are aliased for more concise code. For example, "int" can be used in place of "java.lang.Integer". The aliases are described in the table below titled: "Supported Types for Parameter Maps and Result Maps".

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:

```
<statement id="insertProduct" parameterClass="java.util.Map">
        select * from PRODUCT
        where PRD_CAT_ID = #catId#
        and PRD_CODE = #code#
</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 (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.

Map types are also aliased for more concise code. For example, "map" can be used in place of "java.util.Map". The aliases are described in the table below titled: "Supported Types for Parameter Maps and Result Maps".

Result Maps

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

```
<resultMap  id="resultMapName"  class="some.domain.Class" [extends="parent-resultMap"]>
        <result property="propertyName"  column="COLUMN_NAME"
                [columnIndex="1"]  [javaType="int"]  [jdbcType="NUMERIC"]
                [nullValue="-999999"] [select="someOtherStatement"]
                />
        <result ……/>
        <result ……/>
        <result ……/>
</resultMap>
```

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

The resultMap 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, a Map or XML.

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:

property

The *property* 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 property.

columnIndex

As an optional (minimal) 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 maintainability and readability for speed. Some JDBC drivers may not realize any performance benefit, while others will speed up dramatically.

jdbcType

The jdbcType attribute is used to explicitly specify the database column 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).

javaType

The *javaType* attribute is used to explicitly specify the Java property type of the property to be set. Normally this can be derived from a JavaBeans property through reflection, but certain mappings such as Map and XML mappings cannot provide the type to the framework. If the javaType is not set and the framework cannot otherwise determine the type, the type is assumed to be Object.

nullValue

The *nullValue* 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 nullValue attribute instead of NULL. 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:

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

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 nullValue in the parameter map (discussed earlier in this document).

select

The *select* attribute is used to describe a relationship between objects and automatically load complex (i.e. user defined) property types. The value of the 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 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.

Implicit Result Maps

If you have a very simple requirement that does not require the reuse of an explicitly defined resultMap, there is a quick way to implicitly specify a result map by setting a resultClass 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:

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

The above mapped statement specifies a resultClass 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). Since many databases are not case sensitive, 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 resultClass. Accessing ResultSetMetaData can be slow with some poorly written 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 can be named anything you like (usually "value" or "val"). 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:

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

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):

```
<statement  id="getProductCount" resultClass="java.lang.Integer">
    select count(1) as value
    from PRODUCT
</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:

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

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:

```
<statement id="getProductCount" resultClass="java.util.HashMap">
    select * from PRODUCT
</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 resultMap 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:

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

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

<statement id="getProduct" parameterClass="int" resultMap="get-product-result">
        select * from PRODUCT where PRD_ID = #value#
</statement>

<statement id="getCategory" parameterClass="int" resultMap="get-category-result">
        select * from CATEGORY where CAT_ID = #value#
</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.

Avoiding N+1 Selects (1:1)

The problem with the solution above is that whenever you load a Product, two SQL statements are actually being run (one for the Product and one for the Category). This problem seems trivial when loading a single Product, but if you were to run a query that loaded ten (10) Products, a separate query would be run for each Product to load its associated category. This results in eleven (11) queries total: one for the list of Products and one for each Product returned to load each related Category (N+1 or in this case 10+1=11).

The solution is to use a join and nested property mappings instead of a separate select statement. Here's an example using the same situation as above (Products and Categories):

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
        <result property="category.id" column="CAT_ID" />
        <result property="category.description" column="CAT_DESCRIPTION" />
</resultMap>

<statement  id="getProduct"  parameterClass="int" resultMap="get-product-result">
        select *
        from PRODUCT, CATEGORY
        where PRD_CAT_ID=CAT_ID
        and PRD_ID = #value#
</statement>
```

Lazy Loading vs. Joins (1:1)

It's important to note that using a join is not *always* better. If you are in a situation where it is rare to access the related object (e.g. the *category* property of the Product class) then it might actually be faster to avoid the join and the unnecessary loading of all category properties. This is especially true for database designs that involve outer joins or nullable and/or non-indexed columns. In these situations it might be better to use the sub-select solution with the lazy loading and bytecode enhancement options enabled (see SQL Map Config settings). The general rule of thumb is: use the join if you're more likely going to access the associated properties than not. Otherwise, only use it if lazy loading is not an option.

If you're having trouble deciding which way to go, don't worry. No matter which way you go, you can always change it without impacting your Java code. The two examples above would result in exactly the same object graph and are loaded using the exact same method call. The only consideration is that if you were to enable caching, then the using the *separate select* (not the join) solution could result in a cached instance being returned. But more often than not, that won't cause a problem (your app shouldn't be dependent on instance level equality i.e. "==").

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 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):

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

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

<statement id="getCategory"  parameterClass="int" resultMap="get-category-result">
        select * from CATEGORY where CAT_ID = #value#
</statement>

<statement  id="getProductsByCatId"  parameterClass="int" resultMap="get-product-result">
        select * from PRODUCT where PRD_CAT_ID = #value#
</statement>
```

Avoiding N+1 Selects (1:M and M:N)

This is similar to the 1:1 situation above, but is of even greater concern due to the potentially large amount of data involved. The problem with the solution above is that whenever you load a Category, two SQL statements are actually being run (one for the Category and one for the list of associated Products). This problem seems trivial when loading a single Category, but if you were to run a query that loaded ten (10) Categories, a separate query would be run for each Category to load its associated list of Products. This results in eleven (11) queries total: one for the list of Categories and one for each Category returned to load each related list of Products (N+1 or in this case 10+1=11). To make this situation worse, we're dealing with potentially large lists of data.

1:N & M:N Solution? *Currently the feature that resolves this issue not implemented. It will be included in a release in the near future.*

Lazy Loading vs. Joins (1:M and M:N)

As with the 1:1 situation described previously, it's important to note that using a join is not *always* better. This is even more true for collection properties than it was for individual value properties due to the greater amount of data. If you are in a situation where it is rare to access the related object (e.g. the *productList* property of the Category class) then it might actually be faster to avoid the join and the unnecessary loading of the list of products. This is especially true for database designs that involve outer joins or nullable and/or non-indexed columns. In these situations it might be better to use the sub-select solution with the lazy loading and bytecode enhancement options enabled (see SQL Map Config settings). The general rule of thumb is: use the join if you're more likely going to access the associated properties than not. Otherwise, only use it if lazy loading is not an option.

As mentioned earlier, if you're having trouble deciding which way to go, don't worry. No matter which way you go, you can always change it without impacting your Java code. The two examples above would result in exactly the same object graph and are loaded using the exact same method call. The only consideration is that if you were to enable caching, then the using the *separate select* (not the join) solution could result in a cached instance being returned. But more often than not, that won't cause a problem (your app shouldn't be dependent on instance level equality i.e. "==").

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 resultMap 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:

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

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

Optionally you can just specify the column names as long as they're in the same order as the parameters. For example:

```
{ORD_ID, ORD_CST_ID}
```

As usual, this is a slight performance gain with an impact on readability and maintainability.

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), or use a join as described in the "N+1 avoidance" solutions.

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. Again, using a join instead can resolve this.

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 <sqlMap> root tag).

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).

Supported Types for Parameter Maps and Result Maps

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

Java Type	JavaBean/Map Property Mapping	Result Class / Parameter Class***	Type Alias**
boolean	YES	NO	boolean
java.lang.Boolean	YES	YES	boolean
byte	YES	NO	byte
java.lang.Byte	YES	YES	byte
short	YES	NO	short
java.lang.Short	YES	YES	short
int	YES	NO	int/integer
java.lang.Integer	YES	YES	int/integer
long	YES	NO	long
java.lang.Long	YES	YES	long
float	YES	NO	float
java.lang.Float	YES	YES	float
double	YES	NO	double
java.lang.Double	YES	YES	double
java.lang.String	YES	YES	string
java.util.Date	YES	YES	date
java.math.BigDecimal	YES	YES	decimal
* java.sql.Date	YES	YES	N/A
* java.sql.Time	YES	YES	N/A
* java.sql.Timestamp	YES	YES	N/A

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

*** .Type Aliases can be used in place of the full class name when specifying parameter or result classes.*

******** *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 feature 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 cacheModel parameter in the statement tag (seen above). A cache model is a configured cache that is defined within your SQL map. Cache models are configured using the cacheModel element as follows:

```
<cacheModel id="product-cache" type ="LRU" readOnly=”true” serialize=”false”>
   <flushInterval hours="24"/>
   <flushOnExecute statement="insertProduct"/>
   <flushOnExecute statement="updateProduct"/>
   <flushOnExecute statement="deleteProduct"/>
   <property name=”cache-size” value=”1000” />
</cacheModel>
```

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 *type* 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:

```
<statement id=”getProductList”   cacheModel=”product-cache”>
      select * from PRODUCT where PRD_CAT_ID = #value#
</statement>
```

Read-Only vs. Read/Write

The framework supports both read-only and read/write caches. Read-only caches are shared among all users and therefore offer greater performance benefit. However, objects read from a read-only cache should not be modified. Instead, a new object should be read from the database (or a read/write cache) for updating. On the other hand, if there is an intention to use objects for retrieval and modification, a read/write cache is recommended (i.e. required). To use a read-only cache, set *readOnly=”true”* on the cache model element. To use a read/write cache, set *readOnly=”false”*. The default is read-only (*true*).

Serializable Read/Write Caches

As you may agree, caching per-session as described above may offer little benefit to global application performance. Another type of read/write cache that can offer a performance benefit to the entire application (i.e. not just per session) is a serializable read/write cache. This cache will return different instances (copies) of the cached object to each session. Therefore each session can safely modify the instance returned. Realize the difference in semantics here, usually you would expect the same instance to be returned from a cache, but in this case you’ll get a different one. Also note that every object stored by a serializable cache must be *serializable*. This means that you will have difficulty using both lazy loading features combined with a serializable cache, because lazy proxies are not serializable. The best way to figure out what combination of caching, lazy loading and table joining is simply to try it out. To use a serializable cache, set *readOnly=”false” and serialize=”true”*. By default cache models are read-only and non-serializable. Read-only caches will not be serialized (there’s no benefit).

Cache Types

The cache model uses a pluggable framework for supporting different types of caches. The implementation is specified in the *type* attribute of the cacheModel 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 property elements contained within the body of the cacheModel. 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:

```
<cacheModel id="product-cache" type="MEMORY">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertProduct"/>
  <flushOnExecute statement="updateProduct"/>
  <flushOnExecute statement="deleteProduct"/>
  <property name="reference-type" value="WEAK" />
</cacheModel>
```

Only a single 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).

“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:

```
<cacheModel id="product-cache" type="LRU">
   <flushInterval hours="24"/>
   <flushOnExecute statement="insertProduct"/>
   <flushOnExecute statement="updateProduct"/>
   <flushOnExecute statement="deleteProduct"/>
   <property name=”size” value=”1000” />
</cacheModel>
```

Only a single property is recognized by the LRU cache implementation. This property, named ‘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:

```
<cacheModel id="product-cache" type="FIFO">
   <flushInterval hours="24"/>
   <flushOnExecute statement="insertProduct"/>
   <flushOnExecute statement="updateProduct"/>
   <flushOnExecute statement="deleteProduct"/>
   <property name=”size” value=”1000” />
</cacheModel>
```

Only a single property is recognized by the FIFO cache implementation. This property, named ‘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:

```
<cacheModel id="product-cache" type="OSCACHE">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertProduct"/>
  <flushOnExecute statement="updateProduct"/>
  <flushOnExecute statement="deleteProduct"/>
</cacheModel>
```

The OSCACHE implementation does not use any 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/

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 any mapped statement element. Here is a simple example:

```
<select id="dynamicGetAccountList"
                    cacheModel="account-cache"
                    resultMap="account-result" >

      select * from ACCOUNT

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

      order by ACC_LAST_NAME

</select>
```

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.

```
 <statement id="dynamicGetAccountList"
                    resultMap="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
 </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:

```
<statement id="someName"
                resultMap="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
</statement>
```

In the above statement, the <dynamic> element demarcates a section of the SQL that is dynamic. The dynamic element is optional and provides a way to manage a prepend in cases where the prepend (e.g. "WHERE") should not be included unless the contained conditions append to the statement. The statement 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 statements and 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:

```
<statement id="getProduct" resultMap="get-product-result">
        select * from PRODUCT order by $preferredOrder$
</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 <statements> and come in handy when there is a need to modify the SQL statement itself. For example:

```
<statement id="getProduct" resultMap="get-product-result">
        SELECT * FROM PRODUCT
        <dynamic prepend="WHERE">
          <isNotEmpty property="description">
            PRD_DESCRIPTION  $operator$  #description#
          </isNotEmpty>
        </dynamic>
</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 SqlMapClient 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 (including insert and delete), 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). SqlMapClient instances are built using SqlMapClientBuilder. 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 sqlMap-config.xml (not necessarily named that).

```
String resource = "com/ibatis/example/sqlMap-config.xml";
Reader reader = Resources.getResourceAsReader (resource);
SqlMapClient sqlMap =  SqlMapClientBuilder.buildSqlMap(reader);
```

Transactions

By default, calling any executeXxxx() method on an SqlMapClient 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 behavior. 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 SqlMapClient interface 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 SqlMapClient interface:

```
public void startTransaction () throws SQLException
public void commitTransaction () throws SQLException
public void endTransaction () 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/sqlMap-
config.xml");
private SqlMapClient sqlMap = XmlSqlMapBuilder.buildSqlMap(reader);

public updateItemDescription (String itemId, String newDescription)
      throws SQLException {
   try {
      sqlMap.startTransaction ();
      Item item =  (Item) sqlMap.queryForObject ("getItem", itemId);
      item.setDescription (newDescription);
      sqlMap.update ("updateItem", item);
      sqlMap.commitTransaction ();
   } finally {
      sqlMap.endTransaction ();
   }
}
```

Notice how endTransaction() is called regardless of an error. This is an important step to ensure cleanup. The rule is: if you call startTransaction() be absolutely certain to call endTransaction() (whether you commit or not).

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 SqlMapClient instance.

Note! SqlMapClient 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 endTransaction(). 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 endTransaction() 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/sqlMap-
config.xml");
private SqlMapClient sqlMap = XmlSqlMapBuilder.buildSqlMap(reader);

public updateItemDescription (String itemId, String newDescription)
      throws SQLException {
   try {
      Item item =  (Item) sqlMap.queryForObject ("getItem", itemId);
      item.setDescription ("TX1");
      //No transaction demarcated, so transaction will be automatic (implied)
      sqlMap.update ("updateItem", item);
      item.setDescription (newDescription);
      item.setDescription ("TX2");
      //No transaction demarcated, so transaction will be automatic (implied)
      sqlMap.update("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 <transactionManager> *type* attribute to "EXTERNAL" 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 SqlMapClient methods startTransaction(), commitTransaction() and endTransaction() (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 endTransaction()) 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. To enable support for managed global transactions, you must set the <transactionManager> *type* attribute to "JTA" in your SQL Map configuration file and set the "UserTransaction" property to the full JNDI name of where the SqlMapClient instance will find the UserTransaction instance. See the <transactionManager> discussion above for full configuration details.

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();
} finally {
  try {
    storeSqlMap.endTransaction()
  } finally {
    orderSqlMap.endTransaction()
  }
}
```

In this example, there are two SqlMapClient instances that we will assume are using two different databases. The first SqlMapClient (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 SqlMapClient (orderSqlMap) calls commitTransaction() and endTransaction(), 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 Session 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.executeBatch();
```

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

Executing Statements via the SqlMapClient API

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

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

public int update(String statementName, Object parameterObject)
       throws SQLException

public int delete(String statementName, Object parameterObject)
       throws SQLException

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

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

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

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

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

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

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

public Map queryForMap (String statementName, Object parameterObject,
       String keyProperty, String valueProperty)
       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 <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 are outlined below.

insert(), update(), delete(): These methods are 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.

queryForObject(): 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.

queryForList(): There are three versions of queryForList(). 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.

Finally there is a queryForList() method that takes a row handler. 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, List list);
```

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. The list parameter is an instance of the List interface that will be returned from the queryForList() method. You can optionally add none, some or all of the result objects to the list. Obviously if you're dealing with millions of rows, it's not a good idea to put them all in a list.

queryForPaginatedList(): 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.

queryForMap(): 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).

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

```
sqlMap.startTransaction();

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

int rows = sqlMap.insert ("insertProduct", product);

sqlMap.commitTransaction();
```

Example 2: Executing Query for Object (select)

```
sqlMap.startTransaction();

Integer key = new Integer (1);

Product product = (Product)sqlMap.queryForObject ("getProduct", key);

sqlMap.commitTransaction();
```

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

```
sqlMap.startTransaction();

Customer customer = new Customer();

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

sqlMap.commitTransaction();
```

Example 4: Executing Query for List (select)

```
sqlMap.startTransaction();

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

sqlMap.commitTransaction();
```

Example 5: Auto-commit

```
// When startTransaction is not called, the statements will
// auto-commit.  Calling commit/rollback is not needed.
int rows = sqlMap.insert ("insertProduct", product);
```

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

```
sqlMap.startTransaction();

List list = sqlMap.queryForList ("getProductList", null, 0, 40);

sqlMap.commitTransaction();
```

Example 7: Executing Query with a RowHandler (select)

```
  public class MyRowHandler implements RowHandler {

    public void handleRow (Object object, List list) throws
SQLException {
      Product product = (Product) object;
      product.setQuantity (10000);
      sqlMap.update ("updateProduct", product);
      // Optionally you could add the result object to the list.
      // The list is returned from the queryForList() method.
    }

  }

  sqlMap.startTransaction();

  RowHandler rowHandler = new MyRowHandler();
  List list = sqlMap.queryForList ("getProductList", null, rowHandler);

  sqlMap.commitTransaction();
```

Example 8: Executing Query for Paginated List (select)

```
  PaginatedList list =
           sqlMap.queryForPaginatedList ("getProductList", null, 10);

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

Example 9: Executing Query for Map

```
  sqlMap.startTransaction();

  Map map = sqlMap.queryForMap ("getProductList", null, "productCode");

  sqlMap.commitTransaction();

  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/

Log Configuration

Configuring the commons logging services is very simply a matter of including one or more extra configuration files (e.g. log4j.properties) 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 2 steps.

Step 1: 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 or use the JAR included with the iBATIS framework. 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 2: 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=DEBUG
6  #log4j.logger.com.ibatis.common.jdbc.SimpleDataSource=DEBUG
7  #log4j.logger.com.ibatis.common.jdbc.ScriptRunner=DEBUG
8  #log4j.logger.com.ibatis.sqlmap.engine.impl.SqlMapClientDelegate=DEBUG
9  #log4j.logger.java.sql.Connection=DEBUG
10 #log4j.logger.java.sql.Statement=DEBUG
11 #log4j.logger.java.sql.PreparedStatement=DEBUG
12 #log4j.logger.java.sql.ResultSet=DEBUG
13
14 # Console output...
15 log4j.appender.stdout=org.apache.log4j.ConsoleAppender
16 log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
17 log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %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
```

By changing line 2 as above, Log4J will now report on all logged events to the 'stdout' appender (console). 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 12 above). So if we wanted to log PreparedStatement activity (SQL statements) to the console at the DEBUG level, we would simply change line 11 to the following (notice it's not #commented out anymore):

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

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 mutates 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.), and must have a no-argument constructor, 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!

Okay, Two Pages

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

Throughout this document you may have seen objects accessed through a special syntax that might be familiar to anyone who has used Struts or any other JavaBeans compatible framework. 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 SqlMapClient property (ResultMap, ParameterMap etc…) as follows (given the employee object as above):

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

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. sqlMap-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.sqlMap-config.xml):

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

SimpleDataSource (com.ibatis.common.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).
*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 sqlMap-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.common.jdbc.*)

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(…));
```

CLINTON BEGIN MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.

© 2004 Clinton Begin. All rights reserved. iBATIS and iBATIS logos are trademarks of Clinton Begin.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.