

iBatis

Data Mapper
(a.k.a SQL Maps)

Version 2.0

Developer Guide

June 29, 2006



Table of Contents

Introduction.....	3
Data Mapper.....	3
Installation.....	4
Upgrading from 1.x.....	5
The SQL Map XML Configuration File	7
The <properties> Element.....	8
The <settings> Element.....	8
The <resultObjectFactory> Element.....	10
The <typeAlias> Element.....	11
The <transactionManager> Element.....	11
The <dataSource> Element.....	12
The <sqlMap> Element.....	14
The SQL Map XML File	15
Mapped Statements.....	16
Statement Types.....	16
The SQL.....	18
Stored Procedures.....	19
Parameter Maps and Inline Parameters.....	24
Inline Parameter Maps.....	27
Primitive Type Parameters.....	28
Map Type Parameters.....	28
Result Maps.....	29
Implicit Result Maps.....	31
Primitive Results.....	31
Complex Properties	32
Avoiding N+1 Selects (1:1).....	33
Complex Collection Properties	34
Avoiding N+1 Selects (1:M and M:N).....	35
Composite Keys or Multiple Complex Parameters Properties	36
Supported Types for Parameter Maps and Result Maps.....	38
Creating custom Type Handlers.....	39
Caching Mapped Statement Results.....	40
Read-Only vs. Read/Write.....	40
Serializable Read/Write Caches.....	40
Cache Types.....	41
Dynamic Mapped Statements.....	44
Dynamic Element.....	45
Binary Conditional Elements.....	45
Unary Conditional Elements	46
Other Elements.....	47
Simple Dynamic SQL Elements.....	49
Programming with Data Mapper: The API.....	50
Configuration.....	50
Transactions.....	50
Batches.....	53
Executing Statements via the SqlMapClient API.....	54
Logging SqlMap Activity.....	59
The One Page JavaBeans Course.....	61
Resources (com.ibatis.common.resources.*)	63
SimpleDataSource (com.ibatis.common.jdbc.*).....	64

Introduction

The iBATIS Data Mapper framework will help you to significantly reduce the amount of Java code that you normally need to access a relational database. iBATIS simply maps JavaBeans to SQL statements using a very simple XML descriptor. *Simplicity* is the key advantage of iBATIS over other frameworks and object relational mapping tools. To use the iBATIS Data Mapper 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 Data Mapper you have the full power of real SQL at your fingertips.

Data Mapper (com.ibatis.sqlmap.*)

Concept

The iBATIS Data Mapper API allows programmers to easily map JavaBeans objects to PreparedStatement parameters and ResultSets. The philosophy behind Data Mapper is simple: provide a simple framework to provide 80% of JDBC functionality using only 20% of the code.

How does it work?

Data Mapper 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, ...
2. Execute the mapped statement. This step is where the magic happens. The Data Mapper 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 iBATIS Data Mapper 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 -cp* 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 Data Mapper 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 iBATIS has no other dependencies. The optional JAR file libraries can be downloaded from the websites below. They are categorized by function. The following is a summary of when you would need to use the optional packages.

Description	When to Use	Dependency
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.	JDBC 2.0 Extensions http://java.sun.com/products/jdbc/download.html JTA 1.0.1a http://java.sun.com/products/jta/ Xerces 2.4.0 http://xml.apache.org/xerces2-j/
iBATIS Backward Compatibility	If you're using the old iBATIS (1.x) DAO framework, or the old Data Mapper (1.x) you can continue to do so by simply including the JAR files in this directory.	iBATIS DAO 1.3.1 http://sourceforge.net/projects/ibatisdb/
Runtime Bytecode Enhancement	If you want to enable CGLIB 2.0 bytecode enhancement to improve lazy loading and reflection performance.	CGLIB 2.0 http://cglib.sf.net
DataSource Implementation	If you want to use the Jakarta DBCP connection pool.	DBCP 1.1 http://jakarta.apache.org/commons/dbcp/
Distributed Caching	If you want to use OSCache for centralized or distributed caching support.	OSCache 2.0.1 http://www.opensymphony.com/oscache/
Logging Solution	If you want to use Log4J logging.	Log4J 1.2.8 http://logging.apache.org/log4j/docs/
Logging Solution	If you want to use Jakarta Commons Logging	Jakarta Commons Logging http://jakarta.apache.org/commons/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 Data Mapper 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 <i>After release 1.2.9b, this file was split into the following 3 files</i> 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 (optional) 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 Data Mapper framework.

The SQL Map XML Configuration File (<http://ibatis.apache.org/dtd/sql-map-config-2.dtd>)

Data Mapper is configured using a central XML configuration file, which provides configuration details for DataSources, Data Mapper 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.apache.org//DTD SQL Map Config 2.0//EN"
  "http://ibatis.apache.org/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="128"
    maxSessions="10"
    maxTransactions="5"
    useStatementNamespaces="false"
    defaultStatementTimeout="5"
  />

  <!-- This element declares a factory class that iBATIS will use for creating result objects.
       This element is optional (more detail later in this document). -->
  <resultObjectFactory type="com.mydomain.MyResultObjectFactory" >
    <property name="someProperty" value="someValue"/>
  </resultObjectFactory>

  <!-- 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>
</sqlMapConfig>
```

```

</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 Data Mapper 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	<p>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 <i>increase</i> performance.</p> <p><i>Example: maxRequests="256"</i> <i>Default: 512</i></p>
maxSessions	<p>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.</p> <p><i>Example: maxSessions="64"</i> <i>Default: 128</i></p>

maxTransactions	<p>This is the maximum number of threads that can enter <code>SqlMapClient.startTransaction()</code> 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 <code>maxSessions</code> and always much less than <code>maxRequests</code>. Often reducing the maximum number of concurrent transactions can <i>increase</i> performance.</p> <p><i>Example: <code>maxTransactions="16"</code></i> <i>Default: 32</i></p>
cacheModelsEnabled	<p>This setting globally enables or disables all cache models for an <code>SqlMapClient</code>. This can come in handy for debugging.</p> <p><i>Example: <code>cacheModelsEnabled="true"</code></i> <i>Default: true (enabled)</i></p>
lazyLoadingEnabled	<p>This setting globally enables or disables all lazy loading for an <code>SqlMapClient</code>. This can come in handy for debugging.</p> <p><i>Example: <code>lazyLoadingEnabled="true"</code></i> <i>Default: true (enabled)</i></p>
enhancementEnabled	<p>This setting enables runtime bytecode enhancement to facilitate optimized JavaBean property access as well as enhanced lazy loading.</p> <p><i>Example: <code>enhancementEnabled="true"</code></i> <i>Default: false (disabled)</i></p>
useStatementNamespaces	<p>With this setting enabled, you must always refer to mapped statements by their fully qualified name, which is the combination of the <code>sqlMap</code> name and the statement name. For example:</p> <p><code>queryForObject("sqlMapName.statementName");</code></p> <p><i>Example: <code>useStatementNamespaces="false"</code></i> <i>Default: false (disabled)</i></p>
defaultStatementTimeout	<p>(iBATIS versions 2.2.0 and later)</p> <p>This setting is an integer value that will be applied as the JDBC query timeout for all statements. This value can be overridden with the "statement" attribute of any mapped statement. If not specified, not query timeout will be set unless specified on the "statement" attribute of a mapped statement. The specified value is the number of seconds the driver will wait for a statement to finish. Note that not all drivers support this setting.</p>

The <resultObjectFactory> Element

Important: this feature is available in iBATIS versions 2.2.0 and higher.

The *resultObjectFactory* element allows you to specify a factory class for creating objects resulting from the execution of SQL statements. This element is optional – if you don't specify the element, iBATIS will use internal mechanisms to create result objects (`class.newInstance()`).

iBATIS creates result objects in these cases:

1. When mapping rows returned from a `ResultSet` (the most common case)
2. When you use a nested select statement on a *result* element in a *resultMap*. If the nested select statement declares a *parameterClass*, then iBATIS will create and populate an instance of the class before executing the nested select
3. When executing stored procedures – iBATIS will create objects for OUTPUT parameters
4. When processing nested result maps. If the nested result map is used in conjunction with the `groupBy` support for avoiding N+1 queries, then the object will typically be an implementation of type `Collection`, `List`, or `Set`. You can provide custom implementations of these interfaces through the result object factory if you wish. In a 1:1 join with a nested result map, then iBATIS will create an instance of the specified domain object through this factory.

If you choose to implement a factory, your factory class must implement the interface **`com.ibatis.sqlmap.engine.mapping.result.ResultObjectFactory`**, and your class must have a public default constructor. The `ResultObjectFactory` interface has two methods – one to create an object, and one to accept any property values specified in the configuration.

For example, suppose you specify a *resultObjectFactory* configuration element like this:

```
<resultObjectFactory type="com.mydomain.MyResultObjectFactory" >
  <property name="someProperty" value="someValue"/>
</resultObjectFactory>
```

Then you should code a result object factory class like this:

```
package com.mydomain;

import com.ibatis.sqlmap.engine.mapping.result.ResultObjectFactory;

public class MyResultObjectFactory implements ResultObjectFactory {

    public MyResultObjectFactory() {
        super();
    }

    public Object createInstance(String statementId, Class clazz)
        throws InstantiationException, IllegalAccessException {

        // create and return instances of clazz here...

    }

    public void setProperty(String name, String value) {
        // save property values here...
    }
}
```

iBATIS will call the *setProperty* method once for each property specified in the configuration. All properties will be set before any call to the *createInstance* method is processed.

iBATIS will call the *createInstance* method every time an object needs to be created according to the cases mentioned above. If you return `null` from the *createInstance* method, then iBATIS will attempt to create

the object through its normal means (`class.newInstance()`). If you return `null` from a request to create `java.util.Collection` or `java.util.List`, then iBATIS will create `java.util.ArrayList`. If you return `null` from a request to create `java.util.Set`, then iBATIS will create `java.util.HashSet`. iBATIS passes the current statement id to let you know the context in which the object create is requested.

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: Data Mapper 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 Data Mapper must manage the transactions. This setting is also useful for non-transactional databases (e.g. Read-only).

The <transactionManager> element also allows an optional attribute *commitRequired* that can be *true* or *false*. Normally iBATIS will not commit transactions unless an insert, update, or delete operation has been performed. **This is true even if you explicitly call the `commitTransaction()` method.** This behavior creates problems in some cases. If you want iBATIS to always commit transactions, even if no insert,

update, or delete operation has been performed, then set the value of the *commitRequired* attribute to *true*. Examples of where this attribute is useful include:

1. If you call a stored procedures that updates data as well as returning rows. In that case you would call the procedure with the `queryForList()` operation – so iBATIS would not normally commit the transaction. But then the updates would be rolled back.
2. In a WebSphere environment when you are using connection pooling and you use the JNDI `<dataSource>` and the JDBC or JTA transaction manager. WebSphere requires all transactions on pooled connections to be committed or the connection will not be returned to the pool.

Note that the *commitRequired* attribute has no effect when using the EXTERNAL transaction manager.

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"/>
    <property name="Driver.DriverSpecificProperty" value="SomeValue"/>
  </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="DBJndiContext" 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:comp/UserTransaction"/>
  <dataSource type="JNDI">
    <property name="DBJndiContext" 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 Data Mapper (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://ibatis.apache.org/dtd/sql-map-config-2.dtd>)

In the examples above, we saw the most simple forms of Data Mapper. 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 Data Mapper 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"]
                [timeout="5"]>
    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
<code><statement></code>	id parameterClass resultClass parameterMap resultMap cacheModel resultSetType fetchSize xmlResultName remapResults timeout	All dynamic elements	insert update delete All query methods
<code><insert></code>	id parameterClass parameterMap timeout	All dynamic elements <code><selectKey></code>	insert update delete

<update>	id parameterClass parameterMap timeout	All dynamic elements	insert update delete
<delete>	id parameterClass parameterMap timeout	All dynamic elements	insert update delete

<select>	id parameterClass resultClass parameterMap resultMap cacheModel resultSetType fetchSize xmlResultName remapResults timeout	All dynamic elements	All query methods
<procedure>	id parameterClass resultClass parameterMap resultMap cacheModel fetchSize xmlResultName remapResults timeout	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. Data Mapper 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" >
    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" >
        SELECT @@IDENTITY AS ID
    </selectKey>
</insert>

```

The selectKey statement is executed before the insert statement if it is placed before the insert SQL, otherwise the selectKey statement is executed after the insert statement. In the previous examples, the Oracle example shows that the selectKey will be executed **before** the insert statement (as is appropriate for a sequence). The SQL Server example shows that the selectKey statement will be executed **after** the insert statement (as is appropriate for an identity column).

With iBATIS versions 2.2.0 and later, you can explicitly state the order of execution of the statements if you wish. The *selectKey* element supports an attribute *type* that can be used to explicitly set the execution order. The value of the *type* attribute can be either “pre” or “post” - meaning that the statement will be executed before or after the insert statement. If you specify the *type* attribute, then the value you specify will be used regardless of the position of the *selectKey* element. For example, in the following statement the *selectKey* statement will be executed before the insert statement, even though the element is placed after the insert statement.

```

<insert id="insertProduct-ORACLE-type-specified" parameterClass="com.domain.Product">
    insert into PRODUCT (PRD_ID,PRD_DESCRIPTION)
    values (#id#,#description#)
    <selectKey resultClass="int" type="pre" >
        SELECT STOCKIDSEQUENCE.NEXTVAL AS ID FROM DUAL
    </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" implementation="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>
```

remapResults

The remapResults attribute is available on <statement>, <select>, and <procedure> mapped statements. It's an optional attribute and the default is false.

The `remapResults` attribute should be set to true when a query has a variable set of return columns. For example consider the following queries:

```
SELECT $fieldList$  
FROM   table
```

In the former example the list of column is dynamic, even though the table is always the same.

```
SELECT *  
FROM   $sometable$
```

In the former example the table could be different. Because of the usage of `*` in the select clause, the resulting column names could be different as well. Dynamic elements could also cause the column list to change from one query execution to the next one.

Since the overhead to introspect/determine the resultset metadata is not trivial, iBATIS will remember what was returned the last time the query was run. This creates problems in situations similar to the examples above, hence the possibility to do metadata introspection with every query execution.

So if the return columns can change set `remapResults` to true, else set `remapResults` to false to avoid the overhead of metadata introspection.

resultSetType

To specify the resultSetType of the SQL statement. It can either be:

- **FORWARD_ONLY**: cursor may move only forward;
- **SCROLL_INSENSITIVE**: cursor is scrollable but generally not sensitive to changes made by others;
- **SCROLL_SENSITIVE**: cursor is scrollable and generally sensitive to changes made by others.

Note that resultSetType is generally not required and that different JDBC drivers may behave differently using the same resultSetType setting (e.g. Oracle does not support **SCROLL_SENSITIVE**).

fetchSize

Sets the fetchSize on the SQL statement that will be executed. It gives the JDBC driver a hint to do prefetching in order to minimize round-trips to the database server.

timeout (iBATIS versions 2.2.0 and later only)

Sets the JDBC query timeout for this statement. Any value specified here will override the value specified in the “defaultStatementTimeout” setting in the SQLMapConfig.xml file. If you specify a default timeout and decide that you don't want a timeout for a particular statement, set the timeout value to 0. The specified value is the number of seconds the driver will wait for a statement to finish. Note that not all drivers support this setting.

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="-9999"]
    [typeName="{REF or user-defined type}"]
    [resultMap=someResultMap]
    [mode=IN|OUT|INOUT]
    [typeHandler=someTypeHandler]
    [numericScale=2]/>
  <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 Data Mapper 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.

typeName

The *typeName* attribute is used to explicitly specify a REF type or a user-defined type.

From the javadoc the following is stated:

The *typeName* attribute... “should be used for a user-defined or REF output parameter. Examples of user-defined types include: STRUCT, DISTINCT, JAVA_OBJECT, and named array types. ... For a user-defined parameter, the fully-qualified SQL type name of the parameter should also be given, while a REF parameter requires that the fully-qualified type name of the referenced type be given. A JDBC driver that does not need the type code and type name information may ignore it. To be portable, however, applications should always provide these values for user-defined and REF parameters. Although it is intended for user-defined and REF parameters, this *attribute* may be used to register a parameter of any JDBC type. If the parameter does not have a user-defined or REF type, the *typeName* parameter is ignored.”

* *italicized* words were substituted to put the explanation in context of this documentation

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.

resultMap

Specify the *resultMap* element when you expect an instance of `java.sql.ResultSet` as the value of a stored procedure output parameter. This will enable iBATIS to do normal result set to object mapping.

mode

The *mode* attribute specifies the mode of a stored procedure parameter. Valid values are IN, OUT, or INOUT.

typeHandler

The *typeHandler* attribute is used to specify a custom type handler that will be used for this property instead of the default iBATIS type system. If specified, this value should be the fully qualified name of a class that implements either the `com.ibatis.sqlmap.engine.type.TypeHandler` interface or the `com.ibatis.sqlmap.client.extensions.TypeHandlerCallback` interface. This value overrides any global type handler that might otherwise be applied to this property. There is further detail on custom type handlers later in this document.

numericScale

(numericScale is available in iBATIS versions 2.2.0 and later only)

The *numericScale* attribute is used to specify the scale (digits to the right of the decimal point) for NUMERIC or DECIMAL stored procedure output parameters. If you specify OUT or INOUT for the *mode* attribute, and the *jdbcType* is DECIMAL or NUMERIC, then you should also specify a value for *numericScale*. The value specified for this attribute must be an integer greater than zero.

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.

Inline Parameter Map Syntax

iBATIS supports two different syntaxes for in line parameter maps – a simple syntax, and a more advanced and more complete syntax..

The simple syntax is as follows:

```
#propertyName# - OR -
#propertyName:jdbcType# - OR -
#propertyName:jdbcType:nullValue#
```

Examples of this syntax are as above. The *propertyName* element is the name of a property in the parameter object (or the value of the parameter object itself if the parameter object is a simple value like String, Integer, etc.). The *jdbcType* element is used to specify the JDBC type of the parameter. The value must be one of the types listed in `java.sql.Types` (VARCHAR, INTEGER, etc.) Generally the

jdbcType element is needed if there is a possibility that the value could be NULL, or to specify the use of DATE or TIME fields (as opposed to TIMESTAMP fields). The *nullValue* element is used to specify the NULL replacement value as described above. Note that you cannot specify *nullValue* unless you also specify *jdbcType*.

This syntax is appropriate in almost all situations unless you need access to some of the advanced options of a formal parameter map (for example, when calling a stored procedure).

The more advanced syntax is as follows:

```
#propertyName,javaType=?,jdbcType=?,mode=?,nullValue=?,handler=?,numericScale=?#
```

Where “?” is a value you specify for the attribute.

The advanced syntax gives you access to most values of a formal parameter map. The *propertyName* element is required, all other values are optional. The values can be specified in any order, except that the *propertyName* element must be first. Values allowed for the different attributes are exactly what is allowed when using a formal parameter map. Also note that with this syntax, the *handler* attribute will use an aliased name for the type handler if such an alias is registered. An example of this syntax used to call a stored procedure is as follows:

```
<procedure id="callProcedure" parameterClass="com.mydomain.MyParameter">
    {call MyProcedure
      (#parm1,jdbcType=INTEGER,mode=IN#, #parm2,jdbcType=INTEGER,mode=IN#,
      #parm3,jdbcType=DECIMAL,mode=OUT,numericScale=2#)}
</procedure>
```

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 Data Mapper (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 Data Mapper (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 Data Mapper. 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"]
  [groupBy="some property list"]>
  <result property="propertyName" column="COLUMN_NAME"
    [columnIndex="1"] [javaType="int"] [jdbcType="NUMERIC"]
    [nullValue="-999999"] [select="someOtherStatement"]
    [resultMap="someOtherResultMap"]
    [typeHandler="com.mydomain.MyTypehandler"]
  />
  <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* element also supports the attribute *groupBy*. The *groupBy* attribute is used to specify a list of properties in this *resultMap* that are used to identify unique rows in the returned result set. Rows with equal values for the specified properties will only generate one result object. Use *groupBy* in combination with nested resultMaps to solve the N+1 query problem (see following discussion for examples).

The resultMap can contain any number of result mappings that map JavaBean properties 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 *result* 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.

resultMap

The *resultMap* attribute is used to describe a nested resultMap that can be reused in the result mapping. This can be used in 1:1 relationships or 1:N relationships. If you expect a 1:N relationship, then the related property should be a Collection (List, Set, Collection, etc.), and you should also specify the *groupBy* attribute on the *resultMap* element to denote how iBATIS will group the rows. In 1:1 relationships, the related property can be of any type and the *groupBy* attribute may, or may not, be specified. It is also possible to use the *groupBy* attribute when some joins are 1:N and some are 1:1.

typeHandler

The *typeHandler* attribute is used to specify a custom type handler that will be used for this property instead of the default iBATIS type system. If specified, this value should be the fully qualified name of a class that implements either the `com.ibatis.sqlmap.engine.type.TypeHandler` interface or the `com.ibatis.sqlmap.client.extensions.TypeHandlerCallback` interface. This value overrides any global type handler that might otherwise be applied to this property. There is further detail on custom type handlers 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 `queryForList()`). 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 `queryForList()`). 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>
```



```

</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>

```

In iBATIS versions 2.2.0 and above, you can also reuse a result map in a 1:1 query instead of repeating the columns. An example of this usage is 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="category" resultMap="get-category-result" />
</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, 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

iBATIS fully solves the N+1 selects solution. Here is an example:

```
<sqlMap namespace="ProductCategory">
  <resultMap id="categoryResult" class="com.ibatis.example.Category" groupBy="id">
    <result property="id" column="CAT_ID"/>
    <result property="description" column="CAT_DESCRIPTION"/>
    <result property="productList" resultMap="ProductCategory.productResult"/>
  </resultMap>

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

  <select id="getCategory" parameterClass="int" resultMap="categoryResult">
    select C.CAT_ID, C.CAT_DESCRIPTION, P.PRD_ID, P.PRD_DESCRIPTION
    from CATEGORY C
    left outer join PRODUCT P
    on C.CAT_ID = P.PRD_CAT_ID
    where CAT_ID = #value#
  </select>
</sqlMap>
```

When you call...

```
List myList = queryForList("ProductCategory.getCategory", new Integer(1002));
```

...the main query is executed, and the results are stored in the *myList* variable as beans of type "com.ibatis.example.Category". Each object in that List will have a "productList" property that is also a List populated from the same query, but using the "productResult" result map to populate the beans in the child list. So, you end up with a list containing sub-lists, and only one database query is executed.

The important items here are the...

groupBy="id"

...attribute and the...

```
<result property="productList" resultMap="ProductCategory.productResult"/>
```

...property mapping in the "categoryResult" result map. One other important detail is that the result mapping for the productList property is namespace aware - had it been simply "productResult" it would not work.

Using this approach, you can solve any N+1 problem of any depth or breadth.

Important Notes: Combining the `groupBy` behavior with the `queryForPaginatedList()` API is undefined behavior and is likely to return different results than you expect. Please do not attempt to combine these two ideas. If you are using `groupBy`, you should always use the `queryForList` or `queryForObject` methods.

The nested property can be any implementation of `java.util.Collection`, but the getter and setter for the property should be a simple and just provide access to the internal attribute. iBATIS will repeatedly call the `get` method to access the property, and then call the property's `add()` method as it is processing the result set. Do not try to do anything out of the ordinary with the getters and setters (like trying to wrap an internal array in a `List`) – this will likely cause iBATIS to fail. There is a common misconception that iBATIS somehow batches up the objects and calls the `set` method just one time. This is not the case – iBATIS only calls the `set` method if the `get` method returns null – in which case iBATIS will create a default implementation of the property and sets the new object into the result object. The newly created object will always be empty – because iBATIS will then call the `get` method to obtain the property property and call the `add` method.

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

Note that type aliases are case sensitive as of version 2.2.0. So type aliases “string”, “String”, “StrinG” all map to the java type “java.lang.String”.

** 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.*

Creating custom Type Handlers

Type support can be extended in iBATIS through the use of the `TypeHandler` or the `TypeHandlerCallback` interface. The `TypeHandlerCallback` interface is simpler to implement, so we recommend using it over the more complex `TypeHandler` interface. To create your own type handler you need to create a class that implements the `TypeHandlerCallback`. Using a custom type handler you can extend the framework to handle types that are not supported, or handle supported types in a different way. For example, you might use a custom type handler to implement proprietary BLOB support (e.g. Oracle), or you might use it to handle booleans using "Y" and "N" instead of the more typical 0/1.

Here's a simple example of a boolean handler that uses "Yes" and "No":

```
public class YesNoBoolTypeHandlerCallback implements TypeHandlerCallback {

    private static final String YES = "Y";
    private static final String NO = "N";

    public Object getResult(ResultGetter getter)
        throws SQLException {
        String s = getter.getString();
        if (YES.equalsIgnoreCase(s)) {
            return new Boolean (true);
        } else if (NO.equalsIgnoreCase(s)) {
            return new Boolean (false);
        } else {
            throw new SQLException (
                "Unexpected value " + s + " found where " + YES + " or " + NO + " was expected.");
        }
    }

    public void setParameter(ParameterSetter setter, Object parameter)
        throws SQLException {
        boolean b = ((Boolean)parameter).booleanValue();
        if (b) {
            setter.setString(YES);
        } else {
            setter.setString(NO);
        }
    }

    public Object valueOf(String s) {
        if (YES.equalsIgnoreCase(s)) {
            return new Boolean (true);
        } else if (NO.equalsIgnoreCase(s)) {
            return new Boolean (false);
        } else {
            throw new SQLException (
                "Unexpected value " + s + " found where " + YES + " or " + NO + " was expected.");
        }
    }
}
```

In order to declare these types for use in iBATIS you use the following syntax in your `sqlMapConfig.xml`:

```
<typeHandler
    javaType="boolean"
    jdbcType="VARCHAR"
    callback="org.apache.ibatis.sqlmap.extensions.YesNoBoolTypeHandlerCallback"/>
```

After this iBATIS will know to handle translations between the the stated java type and jdbc type with the particular type handler callback that was written. Optionally, you can also specify a type handler for

individual properties by specifying the type handler on the `<result>` mapping, or with an explicit or inline parameter map.

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 completely 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 determine 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 determine 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, ...), 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 Data Mapper 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"
      open="(" close=")">
      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.

Dynamic Element

The dynamic tag is a simple tag that is meant only to wrap other dynamic sql elements and provide for a way to attach an overall prepend, open or close to the resulting body content. When using this tag the `removeFirstPrepend` attribute functionality is enforced. So, the first content producing nested tag will have it's prepend removed.

Binary Conditional Attributes:

- prepend – the overridable SQL part that will be prepended to the statement (optional)
- open – the string with which to open the entire resulting body content (optional)
- close – the string with which to close the entire resulting body content (optional)

<code><dynamic></code>	Wrapper tag that allows for an overall prepend, open and close.
------------------------------	---

Binary Conditional Elements

Binary conditional elements compare a property value to a static value 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)

removeFirstPrepend – removes the prepend of the first nested content producing tag (true/false, optional)
 open – the string with which to open the entire resulting body content (optional)
 close – the string with which to close the entire resulting body content (optional)

<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: <pre><isLessEqual prepend="AND" property="age" compareValue="18"> ADOLESCENT = 'TRUE' </isLessEqual></pre>

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)
 removeFirstPrepend – removes the prepend of the first nested content producing tag (true/false, optional)
 open – the string with which to open the entire resulting body content (optional)
 close – the string with which to close the entire resulting body content (optional)

<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: <pre><isNotEmpty prepend="AND" property="firstName" > FIRST_NAME=#firstName# </isNotEmpty></pre>

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)
- removeFirstPrepend – removes the prepend of the first nested content producing tag (true|false, optional)
- open – the string with which to open the entire resulting body content (optional)
- close – the string with which to close the entire resulting body content (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).
	<p>Example Usage:</p> <pre><isNotParameterPresent prepend="AND"> EMPLOYEE_TYPE = 'DEFAULT' </isNotParameterPresent></pre>

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.Collection, or java.util.Iterator, or an array that is to be iterated over (optional – the parameter object is assumed to be a collection if the property is not specified. See below for more information.)
- 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)
- removeFirstPrepend – removes the prepend of the first nested content producing tag (true|false|iterate, optional – see below for more information)

<iterate>	<p>Iterates over a property that is an implementation java.util.Collection, or java.util.Iterator, or is an array.</p> <p>Example Usage:</p> <pre> <iterate prepend="AND" property="userNameList" open="(" close=")" conjunction="OR"> username=#userNameList[]# </iterate> </pre> <p>It is also possible to use the iterate when the collection is passed in as a parameter to your mapped statement.</p> <p>Example Usage:</p> <pre> <iterate prepend="AND" open="(" close=")" conjunction="OR"> username=#[]# </iterate> </pre> <p>Note: It is very important to include the square brackets[] at the end of the property name when using the Iterate element. These brackets distinguish this object as a collection to keep the parser from simply outputting the collection as a string.</p>
-----------	--

Further <iterate> tag usage notes:

Note that, in the first example, "userNameList[]" becomes an operator that refers to the current item in the list. You can use this operator to select properties from list items like this:

```

<iterate prepend="AND" property="userList"
  open="(" close=")" conjunction="OR">

  firstname=#userList[].firstName# and
  lastname=#userList[].lastName#

</iterate>

```

As of iBATIS version 2.2.0, iterate tags can also be nested to create complex conditions. Here is an example:

```

<dynamic prepend="where">
  <iterate property="orConditions" conjunction="or">
    (
      <iterate property="orConditions[].conditions"
        conjunction="and">
        $orConditions[].conditions[].condition$
        #orConditions[].conditions[].value#
      </iterate>
    )
  </iterate>
</dynamic>

```


This assumes that the parameter object has a property “orConditions” that is a List of objects. And each of the object in that List contains a List property called “conditions”. So we have Lists within Lists in the parameter object.

Notice that the phrase “orConditions[].conditions[].condition” means “get the condition property from the current element in the inner list, which is the conditions property of the current object in the outer loop. There is no restriction to level of nesting iterate tags. Also, the “current item” operators can be used as input to any of the other dynamic tags.

The *removeFirstPrepend* function with the <iterate> tag is somewhat different than the other tags. If you specify *true* for *removeFirstPrepend*, then the first nested attribute that produces content will have its *prepend* removed. This will happen once for the entire loop. This is the correct behavior in most circumstances.

In some circumstances, it may be desirable to have the *removeFirstPrepend* function work for each iteration of the loop, rather than just one time. In this case, specify *iterate* as the value for *removeFirstPrepend*. This function is only valid with iBATIS versions 2.2.0 and higher.

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">
        <isEmpty property="description">
            PRD_DESCRIPTION $operator$ #description#
        </isEmpty>
    </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 Data Mapper: 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 execute method on SqlMapClient instance (e.g. queryForObject() or insert()) will auto-commit or auto-rollback. This means that each call to any execution method 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 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 = SqlMapClientBuilder.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 = SqlMapClientBuilder.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 Data Mapper 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 ibatis.apache.org 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, simple methods allow you to demarcate the boundaries of the batch:

```
try {
    sqlMap.startTransaction();
    sqlMap.startBatch();
    // ... execute statements in between
    int rowsUpdated = sqlMap.executeBatch(); //optional
    sqlMap.commitTransaction();
} finally {
    sqlMap.endTransaction();
}
```

Upon calling `executeBatch()`, all batched statements will be executed through the JDBC driver. Calling `executeBatch()` is optional because the commit operation will execute the batch automatically if there is an open batch. So you can call `executeBatch()` if you want to know the number of rows that were affected, or you can skip it and just call `commitTransaction()`.

If you have a large number of operations to perform in a batch, you might want to issue periodic commits throughout the batch. For example, if you're inserting 1000 rows, you might want to commit every 100 rows to keep from creating huge transactions. If you would like to issue periodic commits it is important to know that you should call `startBatch()` after each periodic commit – because the commit will execute and end the batch. Here is an example:

```
try {
    int totalRows = 0;
    sqlMap.startTransaction();

    sqlMap.startBatch();
    // ... insert 100 rows
    totalRows += sqlMap.executeBatch(); //optional
    sqlMap.commitTransaction();

    sqlMap.startBatch();
    // ... insert 100 rows
    totalRows += sqlMap.executeBatch(); //optional
    sqlMap.commitTransaction();

    sqlMap.startBatch();
    // ... insert 100 rows
    totalRows += sqlMap.executeBatch(); //optional
    sqlMap.commitTransaction();

    // etc.

} finally {
    sqlMap.endTransaction();
}
```

Important notes about batches:

1. A batch should ALWAYS be nested inside an explicit transaction. If you fail to use an explicit transaction, then iBATIS will execute each statement individually as if you had never started a batch.

2. You may execute any mapped statement within the batch demarcations. If you execute different mapped statements (i.e. inserts, then updates), iBATIS will break the batch into “sub batches” based on the generated SQL of the last statement executed. For example, consider the following code:

```
try {
    sqlMap.startTransaction();
    sqlMap.startBatch();

    sqlMap.insert("myInsert", parameterObject1);
    sqlMap.insert("myInsert", parameterObject2);
    sqlMap.insert("myInsert", parameterObject3);
    sqlMap.insert("myInsert", parameterObject4);

    sqlMap.update("myUpdate", parameterObject5);
    sqlMap.update("myUpdate", parameterObject6);

    sqlMap.insert("myInsert", parameterObject7);
    sqlMap.insert("myInsert", parameterObject8);
    sqlMap.insert("myInsert", parameterObject9);

    sqlMap.executeBatch();
    sqlMap.commitTransaction();
} finally {
    sqlMap.endTransaction();
}
```

iBATIS will execute this batch in three sub batches – one for the first four insert statements, another for the next two update statements, and another for the last three insert statements. Note that even though the last three insert statements are the same as the first four, iBATIS will still execute a different sub batch because the update statements were in between.

3. The `executeBatch()` method returns an `int` – the total number of records updated in the batch. If there are sub batches, iBATIS will add the number of rows updated in each sub batch to the total. Note that it is entirely legal for the JDBC driver to fail to return the number of records updated in a batch - in which case the `executeBatch()` method will return 0 even though records have been updated. The Oracle driver is a good example of a driver that behaves this way.
4. As of iBATIS versions 2.2.0 and higher, you can use a different method to execute batches - *`executeBatchDetailed`*. This method functions the same as the regular `executeBatch` method (requires an explicit transaction, uses sub batches, etc.), but it returns more detailed information about the row counts. The *`executeBatchDetailed`* method returns a List of `BatchResult` objects – one for each sub batch. Each `BatchResult` object contains information about the statement associated with the sub batch as well as the `int[]` returned from the JDBC driver when the sub batch was executed. If a `java.sql.BatchUpdateException` occurs, the method will throw `BatchException` which contains information about the statement that caused the exception, as well as the List of `BatchResult` objects from any previous successful sub batch.

Executing Statements via the `SqlMapClient` API

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

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

public Object insert(String statementName) throws SQLException

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

```
public int update(String statementName) throws SQLException

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

public int delete(String statementName) throws SQLException

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

public Object queryForObject(String statementName) 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) throws SQLException

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

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

void queryWithRowHandler (String statementName,
    Object parameterObject, RowHandler rowHandler)
    throws SQLException

void queryWithRowHandler (String statementName, RowHandler rowHandler)
    throws SQLException

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

public PaginatedList queryForPaginatedList(String statementName,
    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

public void flushDataCache()

public void flushDataCache(String cacheId)
```

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 a statement element described above (<insert>, <update>, <select>, etc.). 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. As of iBATIS 2.2.0, many of the methods also have overloads without the parameter object if no parameters are expected. 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 four versions of `queryForList()`. The first executes a query and returns all of the results from that query. The second is like the first, but does not accept a parameter object. The third 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. The fourth is like the third, but does not accept a parameter object.

queryWithRowHandler(): 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 valueObject);
```

This method will be called on the `RowHandler` for each row returned from the database. The `valueObject` passed into the method is the resolved Java object for the current row. This is a very clean, simple and scalable way to process results of a query. With this method you can respond to each object produced from a query individually – rather than having iBATIS fill a list and return the list in whole. This would be an efficient way to deal with huge result sets and could result in memory savings.

For an example usage of `RowHandler`, see the examples section below.

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).

flushDataCache(): These methods provide a programmatic way of flushing the data caches. The method without arguments will flush all data caches. The method taking a `cacheId` as argument will only flush the named data cache. Note that for the latter you always need to specify the `cacheId` using namespaces (even if you set `useStatementNamespaces` to `false`).

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


```
sqlMap.startTransaction();

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

Integer primKey = (Integer)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");

sqlMap.commitTransaction();
```

Example 5: Auto-commit

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

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

```
sqlMap.startTransaction();

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

sqlMap.commitTransaction();
```

Example 7: Executing Query with a RowHandler (select)

```
public class MyRowHandler implements RowHandler {
    private SqlMapClient sqlMap;

    public MyRowHandler(SqlMapClient sqlMap) {
        this.sqlMap = sqlMap;
    }

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

sqlMap.startTransaction();

RowHandler rowHandler = new MyRowHandler(sqlMap);
sqlMap.queryWithRowHandler ("getProductList", rowHandler);

sqlMap.commitTransaction();
```

Example 8: Executing Query for Paginated List (select)

```
PaginatedList list =
    sqlMap.queryForPaginatedList ("getProductList", 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

The SqlMap framework provides logging information through the use of an internal log factory. The internal log factory will delegate logging information to one of the following log implementations:

1. Jakarta Commons Logging (JCL – *NOT Job Control Language!*)
2. Log4J
3. JDK logging (JRE 1.4 or greater required)

The logging solution chosen is based on a runtime introspection by the internal iBATIS log factory. The iBATIS log factory will use the first logging implementation it finds (implementations are searched in the above order). If iBATIS finds none of the above implementations, then logging will be disabled.

Many environments ship JCL as a part of the application server classpath (good examples include Tomcat and WebSphere). It is important to know that in such environments, iBATIS will use JCL as the logging implementation. In an environment like WebSphere this will mean that your Log4J configuration will be ignored because WebSphere supplies its own proprietary implementation of JCL. This can be very frustrating because it will appear that iBATIS is ignoring your Log4J configuration (in fact, iBATIS *is* ignoring your Log4J configuration because iBATIS will use JCL in such environments).

If your application is running in an environment where JCL is included in the classpath but you would rather use one of the other logging implementations you can select a different logging implementation by calling one of the following methods (this is only available in iBATIS version 2.2.0 and later):

```
com.ibatis.common.logging.LogFactory.selectLog4JLogging();  
com.ibatis.common.logging.LogFactory.selectJavaLogging();
```

If you choose to call one of these methods, you should do so before calling any other iBATIS method. Also, these methods will only switch to the requested log implementation if that implementation is available on the runtime classpath. For example, if you try to select Log4J logging and Log4J is not available at runtime, then iBATIS will ignore the request to use Log4J and will use its normal algorithm for discovering logging implementations.

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

iBATIS logs most of its activity using log classes that are not in the iBATIS packages. To see iBATIS logging statements, you should enable logging on classes in the java.sql package – specifically the following classes:

- java.sql.Connection
- java.sql.PreparedStatement
- java.sql.ResultSet
- java.sql.Statement

Again, how you do this is dependent on the logging implementation in use. We'll show how to do it with Log4J.

Configuring the logging services is 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. To use Log4J, you need to add the JAR file to your application classpath. You can download Log4J from the URL above. For web or enterprise applications you can add the log4j.jar to your WEB-INF/lib directory, or for a standalone application you can simply add it to the JVM -classpath startup parameter.

Step 2: Configure Log4J

Configuring Log4J is simple – you create a file 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.sqlmap.engine.cache.CacheModel=DEBUG
8 #log4j.logger.com.ibatis.sqlmap.engine.impl.SqlMapClientImpl=DEBUG
9 #log4j.logger.com.ibatis.sqlmap.engine.builder.xml.SqlMapParser=DEBUG
10 #log4j.logger.com.ibatis.common.util.StopWatch=DEBUG
11 #log4j.logger.java.sql.Connection=DEBUG
12 #log4j.logger.java.sql.Statement=DEBUG
13 #log4j.logger.java.sql.PreparedStatement=DEBUG
14 #log4j.logger.java.sql.ResultSet=DEBUG
15
16 # Console output...
17 log4j.appender.stdout=org.apache.log4j.ConsoleAppender
18 log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
19 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 14 above). So if we wanted to log PreparedStatement activity (SQL statements) to the console at the DEBUG level, we would simply change line 14 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 Data Mapper framework requires a firm understanding of JavaBeans. Luckily, there's not much to the JavaBeans API as far as it relates to Data Mapper. 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 writeable 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 Data Mapper 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 `getXxx()` and `setXxx()`. 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 Data Mapper 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 as follows (given the employee object as above):

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

Important: this syntax only applies to properties used with iBATIS' support for dynamic SQL elements. It will not work with properties in <result> or <parameter> mappings.

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 "/>

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.stdxext.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 dependent	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).
<i>Driver.*</i>	No	n/a	<p>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”.</p> <p>Note: These properties also work within the dao.xml and sqlMap-config.xml files.</p>

Example: Using SimpleDataSource

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

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.