
iBATIS Data Mapper Developer Guide

Table of Contents

1. iBATIS Data Mapper	2
1.1. What's covered here	2
1.2. Disclaimer	2
2. The Big Picture	2
2.1. What does it do?	3
2.2. How does it work?	3
3. Working with Data Maps	4
3.1. What's in a Data Map definition file, anyway?	4
3.2. Mapped Statements	6
3.2.1. Statement Types	7
3.2.2. Stored Procedures	8
3.2.3. The SQL	8
3.2.4. Statement-type Element Attributes	9
3.3. Parameter Maps and Inline Parameters	12
3.3.1. <parameterMap> attributes	13
3.3.2. <parameter> Elements	13
3.3.3. Inline Parameter Maps	15
3.3.4. Standard Type Parameters	16
3.3.5. Map or IDictionary Type Parameters	16
3.4. Result Maps	17
3.4.1. Extending resultMaps	18
3.4.2. parameterMap attributes	18
3.4.3. Implicit Result Maps	19
3.4.4. Primitive Results (i.e. String, Integer, Boolean)	19
3.4.5. Maps with ResultMaps	20
3.4.6. Complex Properties (i.e. a property of a class defined by the user)	20
3.4.7. Avoiding N+1 Selects (1:1)	21
3.4.8. Complex Collection Properties	21
3.4.9. Avoiding N+1 Selects (1:M and M:N)	22
3.4.10. Composite Keys or Multiple Complex Parameters Properties	22
3.5. Supported Types for Parameter Maps and Result Maps	23
3.5.1.	23
3.5.2.	24
3.6. Cache Models	24
3.6.1. Read-Only vs. Read/Write	24
3.6.2. Cache Types	24
3.6.3. "MEMORY" (com.ibatis.db.sqlmap.cache.memory.MemoryCacheController)	25
3.6.4. "LRU" (com.ibatis.db.sqlmap.cache.lru.LruCacheController)	25
3.6.5. "FIFO" (com.ibatis.db.sqlmap.cache.fifo.FifoCacheController)	26
3.7. Dynamic SQL	26
3.7.1. Binary Conditional Elements	28
3.7.2. Unary Conditional Elements	28
3.7.3. Simple Dynamic SQL Elements	29
4. Java Developer Guide	30
4.1. Installing iBATIS Data Mapper for Java	30
4.1.1. JAR Files and Dependencies	30
4.1.2. Upgrading from version 1.x to version 2.x	31
4.2. Configuring the Data Mapper for Java	32
4.2.1. Data Mapper clients	32
4.2.2. Data Mapper Configuration File (SqlMapConfig.xml)	33
4.2.3. Data Mapper Configuration Elements	33
4.3. Programming with iBATIS Data Mapper: The Java API	37
4.3.1. Configuration	38
4.3.2. Transactions	38

4.3.3. Global (DISTRIBUTED) Transactions	39
4.3.4. Batches	40
4.3.5. Executing Statements via the SqlMapClient API	41
4.4. The One Page JavaBeans Course	44
4.5. Logging SqlMap Activity with Jakarta Commons Logging	45
4.5.1. Log Configuration	45
4.6. SimpleDataSource (com.ibatis.common.jdbc.*)	46
4.7. ScriptRunner (com.ibatis.common.jdbc.*)	47
4.8. Resources (com.ibatis.common.resources.*)	48
5. .NET Developer Guide	49
5.1. Installing the Data Mapper for .NET	49
5.1.1. Setup the Distribution	49
5.1.2. Add Assembly References	49
5.1.3. Add XML File Items	50
5.2. Configuring the Data Mapper for .NET	50
5.2.1. Data Mapper clients	50
5.2.2. Data Mapper Configuration File (SqlMapper.config)	50
5.2.3. Data Mapper Configuration Elements	51
5.3. Programming with iBATIS Data Mapper: The .NET API	54
5.3.1. Building a SqlMapper Instance	55
5.3.2. Exploring the SqlMapper API	56
5.3.3. Using Explicit and Automatic Transactions	58
5.3.4. Coding Examples [TODO: Review from here]	59
5.4. Logging SqlMap Activity with Apache Log4Net	61
5.4.1. Log Configuration	61

1. iBATIS Data Mapper

The iBATIS Data Mapper framework makes it easier to use a database with a Java or .NET application. iBATIS Data Mapper couples objects with stored procedures or SQL statements using a XML descriptor. Simplicity is the biggest advantage of the iBATIS Data Mapper over object relational mapping tools. To use iBATIS Data Mapper you rely on your own objects, XML, and SQL. There is little to learn that you don't already know. With iBATIS Data Mapper you have the full power of both SQL and stored procedures at your fingertips.

1.1. What's covered here

This Guide covers the Java and .NET implementations of iBATIS Data Mapper.

iBATIS Data Mapper for Java version 2.x
iBATIS Data Mapper for .NET version 1.x

Since iBATIS relies on an XML descriptor to create the mappings, much of the material applies to both implementations. When a section pertains to only one implementation, it is labeled "Java Only" or ".NET only". If you are not using that implementation, you may safely skip that section. (No worries, mate.)

For installation instructions, see the Java or .NET Developer Guide (Sections 6 and 7).

A Tutorial is also available. We recommend reviewing the Tutorial for your platform before reading this Guide.

1.2. Disclaimer

iBATIS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

2. The Big Picture

iBATIS is a simple but complete framework that makes it easy for you to map your objects to your SQL state-

ments or stored procedures. The iBATIS team follows a common sense philosophy: obtain 80% of data access functionality using only 20% of the code.

2.1. What does it do?

Developers often create maps between objects within an application. One definition of a Mapper is an "object that sets up communication between two independent objects." A Data Mapper is a "layer of mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself." [Patterns of Enterprise Architecture, ISBN 0-321-12742-0].

You provide the database and the objects; iBATIS provides the mapping layer that goes between the two.

2.2. How does it work?

Your programming platform already provides a capable library for accessing databases, whether through SQL statements or stored procedures. But developers find several things are still hard to do well when using "stock" JDBC or ADO.NET, including:

- Separating SQL code from programming code
- Passing input parameters to the library classes and extracting the output
- Separating data access classes from business logic classes
- Caching often-used data until it changes
- Managing transactions and threading

iBATIS Data Mapper solves these problems -- and many more -- by using XML documents to create a mapping between a plain-old object and a SQL statement or a stored procedure. The "plain-old object" can be a Map or JavaBean (if you are using Java) or a IDictionary or property object (if you are using .NET).

Tip

The object does not need to be part of a special object hierarchy or implement a special interface. (Which is why we call them "plain-old" objects.) Whatever you are already using should work just fine.

Figure 1 diagrams the iBATIS workflow.

Figure 1. iBATIS Data Mapper workflow

Here's a high level description of the workflow diagrammed by Figure 1:

1. Provide a parameter, either as an object or a native type. The parameter can be used to set runtime values in your SQL statement or stored procedure. If a runtime value is not needed, the parameter can be omitted.
2. Execute the mapping by passing the parameter and the name you gave the statement or procedure in your XML descriptor. This step is where the magic happens. The framework will prepare the SQL statement or stored procedure, set any runtime values using your parameter, execute the procedure or statement, and return the result.
3. In the case of an update, the number of rows affected is returned. In the case of a query, a single object, or a collection of objects is returned. Like the parameter, the result object, or collection of objects, can be a plain-old object or a native type. The result can also be given as XML..

So, what does all this look like in your source code? Here's how you might code the insert of a "lineItem" object into your database:

```
Mapper.map().insert("insertLineItem",lineItem); // Java
```

If your database is generating the primary keys, the generated key can be returned from this method call, like

this:

```
MyKey = Mapper.map().Insert("InsertLineItem",LineItem); // .NET
```

Example 1 show how the XML descriptor for "InsertLineItem" might look.

Example 1. The "InsertLineItem" descriptor

```
<insert id="InsertLineItem" parameterClass="LineItem">
  INSERT INTO [LinesItem]
  (Order_Id, LineItem_LineNum, Item_Id, LineItem_Quantity, LineItem_UnitPrice)
  VALUES
  (#Order.Id#, #LineNumber#, #Item.Id#, #Quantity#, #Item.ListPrice#)
  <selectKey resultClass="int" keyProperty="id" >
    SELECT @@IDENTITY AS ID
  </selectKey>
</insert>
```

The <selectKey> stanza returns an autogenerated key from a SQL Server database (for example).

If you need to select multiple rows, iBATIS can return a list of objects, each mapped to a row in the result set:

```
IList productList = Mapper.Map().QueryForList("selectProduct",stCategory); // .NET
```

Or just one, if that's all you need:

```
Object product = Mapper.map().queryForObject("selectProduct",productKey); // Java
```

Of course, there's more, but this is iBATIS from 10,000 meters. (For a longer, gentler introduction, see the Tutorial.) Section 3 describes the Data Map definition files -- where the statement for "InsertLineItem" would be defined. The Developers Guide for your platform (Section 4 or 5) describes the "bootstrap" configuration file that exposes iBATIS to your application.

3. Working with Data Maps

If you want to know how to configure and install iBATIS, see the Developer Guide for your platform (Section 4 or 5). But if you want to know how iBATIS *really* works, continue from here.

The Data Map definition file is where the interesting stuff happens. Here, you define how your application interacts with your database. As mentioned, the Data Map definition is an XML descriptor file. By using a service routine provided by iBATIS, the XML descriptors are rendered into a client object. To access your Data Maps, your application calls the client object and passes in the name of the statement you need.

The real work of using iBATIS is not so much in the application code, but in the XML descriptors that iBATIS renders. Instead of monkeying with application source code, you monkey with XML descriptors instead. The benefit is that the XML descriptors are much better suited to the task of mapping your object properties to database entities.

3.1. What's in a Data Map definition file, anyway?

If you read the Tutorial, you've already seen some simple Data Map examples, like the one shown in Example 2.

Example 2. A simple Data Map (.NET)

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

```
<sqlMap name="LineItem" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://ibatisnet.sf.net/schemaV1/SqlMap.xsd">

<alias>
<typeAlias alias="LineItem" assembly="NPetshop.Domain.dll"
class="NPetshop.Domain.Billing.LineItem" />
</alias>

<statements>

<insert id="InsertLineItem" parameterClass="LineItem">

INSERT INTO [LinesItem]
(Order_Id, LineItem_LineNum, Item_Id, LineItem_Quantity, LineItem_UnitPrice)
VALUES
(#Order.Id#, #LineNumber#, #Item.Id#, #Quantity#, #Item.ListPrice#)

</insert>

</statements>

</sqlMap>
```

This map takes some properties from a LineItem instance and merges the values into the SQL statement. The value-add is that our SQL is separated from our program code, and we can pass our LineItem instance directly to a library method:

```
mapper.Insert("InsertLineItem",LineItem); // C#
```

No fuss, no muss. Likewise, see Example 3 for a simple select statement.

Example 3. A simple Data Map (Java)

```
<?xml version="1.0" encoding="UTF-8" ?>
<sqlMap namespace="Product">
<select id="selectProduct"
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>
```

In Example 3, we use SQL aliasing to map columns to our object properties and an iBatis inline parameter (see sidebar) to insert a runtime value. Easy as pie.

A Quick Glance at Inline Parameters

Although further details are elsewhere (see Section TODO:), here is a quick intro to inline-parameters. Inline parameters can be used inside of any type of SQL statement. For example:

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

In our example, the inline parameters are #id# and #description#. Each represents an object property that will be used to populate the statement parameter in-place. Here, the Product class is expected to have id and description properties that can be read for a value to be placed in the statement in place of the corresponding property token.

So for a statement that is passed a Product with id=5 and description="dog", the statement might be executed as:

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

But, what if you wanted some ice cream with that pie? And maybe a cherry on top? What if we wanted to cache the result of the select. Or, what if we didn't want to use SQL aliasing or named parameters. (Say, because we were using pre-existing SQL that we didn't want to touch.) Example 4 shows a Data Map that specifies a cache, and uses a <parameterMap> and a <resultMap> to keep our SQL pristine.

Example 4. A Data Map definition file with some bells and whistles (Java)

```
<?xml version="1.0" encoding="UTF-8" ?>  
<sqlMap namespace="Product">  
  
    <typeAlias alias="product" type="com.ibatis.example.Product" />  
  
    <cacheModel id="productCache" type="LRU">  
        <flushInterval hours="24"/>  
        <property name="size" value="1000" />  
    </cacheModel>  
  
    <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">  
        select * from PRODUCT where PRD_ID = ?  
    </select>  
</sqlMap>
```

In Example 4, <parameterMap> maps the SQL "?" to the product id property. The <resultMap> maps the columns to our object properties. The <cacheModel> keeps the result of the last one thousand of these queries in active memory for up to 24 hours.

Example 4 is longer and more complex than Example 3, but considering what you get in return, it seems like a fair trade. (A bargain even.)

Many "agile" developers would start with something like Example 3 and add features like caching later. If you changed the Data Map from Example 3 to Example 4, you would not have to touch your application source code at all. You can start simple and add complexity only when it is needed.

A single Data Map definition file can contain as many Cache Models, Type Aliases, Result Maps, Parameter Maps, and Mapped Statements (including stored procedures), as you like. Everything is loaded into the same configuration, so you can define elements in one Data Map and then use them in another. Use discretion and organize the statements and maps appropriately for your application by finding some logical way to group them.

3.2. Mapped Statements

Mapped Statements can hold any SQL statement and can use Parameter Maps and Result Maps for input and output. (A stored procedure is a specialized form of a statement. See section 3.2.1 and 3.2.2 for more.)

If the case is simple, the Mapped Statement can reference the parameter and result classes directly. Mapped Statements also support caching, through reference to a Cache Model element. Example 5 shows the syntax for a statement element.

Example 5. Statement element syntax

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

In Example 5, the [bracketed] parts are optional and in some cases only certain combinations are allowed. So it is perfectly legal to have a Mapped Statement as simple as shown by Example 6.

Example 6. A simplistic Mapped Statement

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

Example 6 is obviously unlikely, but it does show that you can use iBatis to execute arbitrary SQL statements. More likely, you will use the object mapping features with Parameter Maps (Section 3.4) and Result Maps (Section 3.5), where the magic happens.

3.2.1. Statement Types

The <statement> element is a general “catch all” element that can be used for any type of SQL statement. Generally it is a good idea to use one of the more specific statement-type elements. The more specific elements use attributes that are more intuitive. The specific elements may also provide additional features that the general <statement> element does not. Table 1 summarizes the statement-type elements and their supported attributes and features.

Table 1. The six statement-type elements

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

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

The various attributes used by statement-type elements are covered in Section 3.2.4.

3.2.2. Stored Procedures

iBATIC Data Mapper treats stored procedures as yet another statement type. Example 7 shows a simple Data Map hosting a stored procedure.

Example 7. A Data Map using a stored procedure

```

<!-- Java -->
<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>

<!-- .NET -->
<procedure id="SwapEmailAddresses" parameterMap="swap-params">
  ps_swap_email_address
</procedure>
...
<parameterMap id="swap-params">
  <parameter property="email1" column="First_Email" />
  <parameter property="email2" column="Second_Email" />
</parameterMap>

```

The idea behind Example 7 is that calling the stored procedure "swapEmailAddress" would exchange two email addresses between two columns in a database table and also in the parameter object (here, a Map or HashTable). The parameter object is only modified if the parameter mappings mode attribute is set to "INOUT" or "OUT". Otherwise they are left unchanged. Of course, immutable parameter objects (e.g. String) cannot be modified.

Note

For Java, always be sure to use the standard JDBC stored procedure syntax. See the JDBC CallableStatement documentation for details.

3.2.3. The SQL

If you are not using stored procedures, the most important part of a statement-type element is the SQL. You can use any SQL statement that is valid for your database system. Since iBATIC passes the SQL through to the standard libraries (JDBC or ADO.NET), you can use any statement with iBATIC that you could use without iBATIC. You can use whatever functions your database system supports, and even send multiple statements, so

long as your driver or provider supports them.

If standard, static SQL isn't enough, iBATIS can help you build a dynamic SQL statement. See Section 3.7 for more about Dynamic SQL.

3.2.3.1. Escaping XML symbols

Because you are combining SQL and XML in a single document, conflicts can occur. The most common conflict is the greater-than and less-than symbols (<>). SQL statements use these symbols as operators, but they are reserved symbols in XML. A simple solution is to "escape" the SQL statements that uses XML reserved symbols within a CDATA element. Example 8 demonstrates.

Example 8. Using CDATA to "escape" SQL code

```
<statement id="selectPersonsByAge" parameterClass="int" resultClass="person"> <![CDATA[ SELECT * FROM PER
```

3.2.3.2. Auto-Generated Keys

Many database systems support auto-generation of primary key fields, as a vendor extension. Some vendors pre-generate keys (e.g. Oracle), some vendors post-generate keys (e.g. MS-SQL Server). In either case, you can obtain a pre-generated key using a <selectKey> stanza within an <insert> element. Example 9 shows an <insert> statement for either approach.

Example 9. <insert> statements using <selectKey> stanzas

```
<!--Oracle SEQUENCE Example --> <insert id="insertProduct-ORACLE" parameterClass="product"> <selectKey res
```

3.2.4. Statement-type Element Attributes

The six statement-type elements take various attributes. See Section 3.2.1 for a table itemizing which attributes each element-type accepts. The individual attributes are described in Sections 3.2.4.1 through 3.2.4.7.

3.2.4.1. id

The required id attribute provides a name for this statement, which must be unique within this <SqlMap>.

3.2.4.2. parameterMap

A Parameter Map defines an ordered list of values that match up with the "?" placeholders of a standard, parameterized query statement. Example 10 shows a <parameterMap> and a corresponding <statement>.the structure of a <resultMap> element.

Example 10. A parameterMap and corresponding statement

```
<parameterMap id="insert-product-param" class="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 Example 9, the Parameter Map describes two parameters that will match, in order, two placeholders in the

SQL statement. The first “?” is replaced by the value of the “id” property. The second is replaced with the “description” property.

iBatis also supports named, inline parameters, which most developers prefer. However, Parameter Maps are useful when the SQL must be kept in a standard form or when extra information needs to be provided. For more about Parameter Maps see Section 3.4.

3.2.4.3. parameterClass

If a parameterMap attribute is not specified (see Section 3.2.4.2), you may specify a parameterClass instead and use inline parameters (see Section 3.3.3). The value of the parameterClass attribute can be a Type Alias or the fully qualified name of a class. Example 11 shows a statement using a fully-qualified name versus an alias.

Example 11. Ways to specify a parameterClass

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

<!-- typeAlias (defined elsewhere) -->
<statement id="statementName" parameterClass="product">
  insert into PRODUCT values (#id#, #description#, #price#)
</statement>
```

3.2.4.4. resultMap

A Result Map lets you control how data is extracted from the result of a query, and how the columns are mapped to object properties. Example 12 shows a <resultMap> element and a corresponding <statement> element.

Example 12. A <resultMap> and corresponding <statement>

```
<resultMap id="select-product-result" class="product">
  <result property="id" column="PRD_ID"/>
  <result property="description" column="PRD_DESCRIPTION"/>
</resultMap>

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

In Example 12, the result of the SQL query will be mapped to an instance of the Product class using the "select-product-result" <resultMap>. The <resultMap> says to populate the “id” property from the “PRD_ID” column, and to populate the “description” property from the “PRD_DESCRIPTION” column.

Tip

In Example 12, note that using “select *” is supported. If you want all the columns, you don't need to map them all individually. (Though many developers consider it a best practice to always specify the columns expected.)

For more about Result Maps, see Section 3.4.

3.2.4.5. resultClass

If a resultMap is not specified, you may specify a resultClass instead. The value of the resultClass attribute can be a Type Alias or the fully qualified name of a class. The class specified will be automatically mapped to the

columns in the result, based on the result metadata. Example 13 shows a `<statement>` element with a `resultClass` attribute.

Example 13. A `<statement>` element with `resultClass` attribute

```
<statement id="selectPerson" parameterClass="int" resultClass="person">
  SELECT
  ER_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 Example 13, 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 is instantiated and populated by matching the object property names to column names from the query.

Using a SQL aliases to map columns to properties saves defining a `<parameterMap>` element, but there are limitations. 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 a slight performance consequence (from accessing the result metadata). You can overcome these limitations with an explicit Result Map (Section 3.4).

3.2.4.6. `cacheModel`

If you want to cache the result of a query, you can specify a `Cache Model` as part of the `<statement>` element. Example 14 shows a `<cacheModel>` element and a corresponding `<statement>`.

Example 14. A `<cacheModel>` element with its corresponding `<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="selectProductList" parameterClass="int" cacheModel="product-cache">
  select * from PRODUCT where PRD_CAT_ID = #value#
</statement>
```

In Example 14, a cache is defined for products that uses a WEAK reference type and flushes every 24 hours or whenever associated update statements are executed. For more about Cache Models, see Section 3.6.

3.2.4.7. `xmlResultName`

If you prefer results returned as XML, you can use the `XMLResultName` to specify the enclosing element. Example 15 shows a `<select>` that specifies a `xmlResultName`.

Example 15. A `<select>` element with `<xmlResultName>` attribute

```
<select id="selectPerson" 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, only one of which is required
  FROM PERSON
  WHERE PER_ID = #value#
</select>
```

Example 16 shows a XML object that might be returned by the <select> statement shown by Example 15.

Example 16. A result XML object

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

3.3. Parameter Maps and Inline Parameters

Most SQL statements are useful because we can pass them values at runtime. Someone wants a database record with the ID 42, and we need to merge that ID number into a select statement. Both JDBC and ADO.NET support using question marks ("?") as replaceable parameters. A list of one or more parameters are passed at runtime, and each placeholder is replaced in turn. Simple, but labor intensive, since developers spend a lot of time counting symbols to make sure everything is in sync.

Note

Sections 3.0 and 3.1.2 introduce the iBATIS inline parameters, which automatically map properties to named parameters. Many, if not most, iBATIS developers prefer this approach. But others do prefer to stick to the standard, anonymous approach to SQL parameters. Sometimes people need to retain the purity of the SQL statements, other times because extra information needs to be passed.

A Parameter Map defines an ordered list of values that match up with the placeholders of a parameterized query statement. While the attributes specified by the map still need to be in the correct order, each parameter is named. You can populate the underlying class in any order, and the Parameter Map ensure each value is passed in the correct order.

Note

Dynamic Mapped Statements (Section 3.7) can't use Parameter Maps. Being dynamic, the number of parameters will change and defeat the purpose of a Parameter Map.

Parameter Maps can be provided as an external element and "inline". Example 17 shows an external Parameter Map.

Example 17. An external Parameter Map

```
<!-- Java -->
<parameterMap id="parameterMapName" [class="className"]>[type="String"]
  <parameter property="propertyName" [jdbcType="VARCHAR"] [javaType="String"]
    [nullValue="NUMERIC"] [null="-9999999"]/>
  <parameter ..... />
  <parameter ..... />
</parameterMap>

<!-- .NET -->
<parameterMap id="parameterMapName" [class="className"]>
  <parameter property="propertyName" [dbType="VARCHAR"]
    [nullValue="NUMERIC"] [null="-9999999"]/>
  <parameter ..... />
  <parameter ..... />database columns
</parameterMap>
```

In Example 17, the parts in [brackets] are optional. The `parameterMap` element only requires the `id` attribute. The class attribute is optional but recommended. The class attribute helps to validate the incoming parameter and optimizes performance. Example 18 shows a typical `<parameterMap>`.

Example 18. A typical `<parameterMap>` element

```
<parameterMap id="insert-product-param" class="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 Data Map definition file where they are defined. You can refer to a Parameter Map in another Data Map definition file by prefixing the `id` of the Parameter Map with the `id` of the Data Map (set in the `<sqlMap>` root tag). If the Parameter Map in Example 18 were in a `SqlMap` named "Product", it could be referenced from another file using "Product.insert-product-param".

3.3.1. `<parameterMap>` attributes

The `<parameterMap>` element accepts two attributes, `id` which is required, and `class` which is optional

3.3.1.1. `id`

The required `id` attributes provides a unique identifier for the `<parameterMap>` within this Data Map.

3.3.1.2. `class`

The optional `class` attribute specifies a

Note

The parameter classes must be a `JavaBean` or `Map` instance (if you are using `Java`), or a `property object` or `IDictionary` instance (if you are using `.NET`).

3.3.2. `<parameter>` Elements

The `<parameterMap>` element holds one or more parameter stanzas that map object properties to placeholders in a SQL statement. Section 3.3.2.1 through 3.3.2.11 describe each of the attributes.

3.3.2.1. property

The property attribute of <parameterMap> is the name of an object property (get method) of the parameter object. It may also be the name of an entry in a Map (Java) or IDictionary (.NET). The name can be used more than once depending on the number of times it is needed in the statement. (In an update, you might set a column that is also part of the where clause.)

3.3.2.2. jdbcType | dbType

The jdbcType attribute (Java) or dbType attribute (.NET) is used to explicitly specify the database column type of the parameter to be set by this property. For certain operations, some JDBC drivers and ADO.NET providers are not able to determine the type of a column, and the type must be specified.

Note

A perfect example is the Java 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 iBATIS 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 and .NET only have one Date value type (java.util.Date and System.DateTime), most SQL databases have more than one. Usually, a database has at least three different types (DATE, DATETIME, TIMESTAMP). You might need to specify the column type, in order for the value to map correctly.

Note

Most drivers/providers only need the type specified for nullable columns. In this case, you only need to specify the type for the columns that are nullable.

Note

When using Oracle, you will get an “Invalid column type” error if you attempt to set a null value to a column without specifying its type.

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). Section 3.5 describes the types that are supported by the framework.

3.3.2.3. 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.

3.3.2.4. javaType (Java only)

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.

3.3.2.5. mode (Java only)

TODO:

3.3.2.6. column (.NET only)

TODO:

3.3.2.7. output (.NET only)

TODO:

3.3.2.8. size (.NET only)

TODO:

3.3.2.9. precision (.NET only)

TODO:

3.3.2.10. scale (.NET only)

TODO:

3.3.2.11. direction (.NET only)

TODO:

3.3.3. Inline Parameter Maps

If you prefer to use inline parameters (see Sections 3.0 and 3.2.2), you can add extra type information inline too. The inline parameter map syntax lets you embed the property name, the property type, and the column type, into a parametrized SQL statement (i.e. coded directly into the SQL). Example 19 shows Example 18 coded with inline parameters.

Example 19. A <statement> using inline parameters

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

Example 20 shows how you can declare types inline.

Example 20. A <statement> using an inline parameter map

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

Example 21 shows how you can declaring types and null value replacements.

Example 21. A <statement> using an inline parameter map with null value replacements

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

```
</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

For "round-trip" transparency of null values, you must also specify database columns and null value replacements in your Result Map (see Section 3.5).

Note

Inline parameter maps are handy for small jobs, but when there are a lot of type descriptors and null value replacements in a complex statement, an industrial-strength, external parameterMap can be easier.

3.3.4. Standard Type Parameters

In practice, you will find that many statements take a single parameter, often an Integer or a String. Rather than wrap a single value in another object, you can use the standard library object (String, Integer, et cetera) as the parameter directly. Example 22 shows a statement using a standard type parameter.

Example 22. A <statement> using standard type parameters

```
<!-- Java -->
<statement id="insertProduct" parameter="java.lang.Integer">
  select * from PRODUCT where PRD_ID = #value#
</statement>

<!-- .NET -->
<statement id="insertProduct" parameter="System.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, a standard 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, you can use another moniker if you like.) Result Maps support primitive types as results as well. For more information about using primitive types as parameters, see Section 3.4, "Result Maps" and the "Programming iBATIS Data Mapper" section in the Developers Guide for your platform (Section 4 or 5).

For your convenience, primitive types are aliased by the framework. For example, "int" can be used in place of "java.lang.Integer" or "System.Integer". For a complete list, see Section 3.5, "Supported Types for Parameter Maps and Result Maps".

3.3.5. Map or IDictionary Type Parameters

You can also pass a Map (Java) or IDictionary (.NET) instance as a parameter object. This would usually be a HashMap (for Java) or a Hashtable (for .NET). Example 23 shows a <statement> using a Map or IDictionary for a parameterClass.

Example 23. A <statement> using a Map or IDictionary for a parameterClass

```
<!-- Java -->
<statement id="insertProduct" parameterClass="java.util.Map">
```

```
select * from PRODUCT
where PRD_CAT_ID = #catId#
and PRD_CODE = #code#
</statement>

<!-- .NET -->
<statement id="insertProduct" parameterClass="System.Collections.IDictionary">
select * from PRODUCT
where PRD_CAT_ID = #catId#
and PRD_CODE = #code#
</statement>
```

In Example 23, notice that the SQL in this Mapped Statement looks like any other. There is no difference in how the inline parameters are used. In If a Map instance is passed, the Map must contain keys named “catId” and “code”. The values referenced by those keys would be of the appropriate type for the column, just as they would if passed from a properties object.

Result Maps support Map and IDictionary types as results too. For more information about using Map and IDictionary types as parameters, see Section 3.5, "Result Maps" and "Programming iBATIS Data Mapper" in your platform's Developer Guide (Section 5 or 6).

For your convenience, Map and IDictionary types are aliased by the framework. So, “map” can be used in place of “java.util.Map”, and "HashTable" can be used in place of "System.Collections.HashTable". For a complete list of aliases, see Section 3.5, “Supported Types for Parameter Maps and Result Maps”.

3.4. Result Maps

Section 3.3 describes Parameter Maps and Inline parameters, which map object properties to parameters in a database query. Result Maps finish the job by mapping the result of a database query (a set of columns) to object properties. Next to Mapped Statements, the Result Map is probably one of the most commonly used and most important features to understand.

A Result Map lets you control how data is extracted from the result of a query, and how the columns are mapped to object properties. A Result Map can describe the column type, a null value replacement, and complex property mappings (including Collections). Example 24 shows the structure of a <resultMap> element.

Example 24. The structure of a <resultMap> element.

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

In Example 24, the [brackets] indicate optional attributes. The id attribute is required and provides a name for the statement to reference. The class attribute is also required, and specifies a Type Alias or the fully qualified name of a class. This is the class that will be instantiated and populated based on the result mappings it contains.

The resultMap can contain any number of property mappings that map object properties to the columns of a ResultSet. The property mappings are applied in the order that they are defined by the element. The columns will be read in the explicit order specified in the Result Map. This ensures consistent results between different drivers and providers.

Note

As with parameter classes, the result class must be a JavaBean or Map instance (if you are using Java),

or a property object or IDictionary instance (if you are using .NET).

3.4.1. Extending resultMap

The optional extends attribute can be set to the name of another resultMap upon which to base this resultMap. All properties of the "super" resultMap will be included as part of this resultMap, and values from the "super" resultMap are set before any values specified by this resultMap. The effect is similar to extending a class.

Tip

The "super" resultMap must be defined in the file *before* the extending resultMap. The classes for the super and sub resultMaps need not be the same, and do not need to be related in any way.

3.4.2. parameterMap attributes

Sections x.x.x through x.x.x describe the <parameterMap> attributes.

3.4.2.1. property element

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.

3.4.2.2. 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.

3.4.2.3. 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.

3.4.2.4. 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).

3.4.2.5. 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.

3.4.2.6. 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:

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="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).

3.4.2.7. 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.

3.4.3. 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="selectProduct" resultClass="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.

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

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

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

<!-- .NET -->
<resultMap id="select-product-result" class="System.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):

```
<!-- Java -->
<statement id="selectProductCount" resultClass="java.lang.Integer">
  select count(1) as value
  from PRODUCT
</statement>

<!-- .NET -->
<statement id="selectProductCount" resultClass="java.lang.Integer">
  select count(1) as value
  from PRODUCT
</statement>
```

3.4.5. Maps with ResultMaps

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

```
<!-- Java -->
<resultMap id="select-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>

<!-- .NET -->
<resultMap id="select-product-result" class="HashTable">
  <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:

```
<!-- Java -->
<statement id="selectProductCount" resultClass="java.util.HashMap">
  select * from PRODUCT
</statement>

<!-- .NET -->
<statement id="selectProductCount" resultClass="HashTable">
  select * from PRODUCT
</statement>
```

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

3.4.6. 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="select-product-result" class="product">
  <result property="id" column="PRD_ID"/>
  <result property="description" column="PRD_DESCRIPTION"/>
  <result property="category" column="PRD_CAT_ID" select="selectCategory"/>
```

```
</resultMap>
<resultMap id="select-category-result" class="category">
  <result property="id" column="CAT_ID" />
  <result property="description" column="CAT_DESCRIPTION" />
</resultMap>
<statement id="selectProduct" parameterClass="int" resultMap="select-product-result">
  select * from PRODUCT where PRD_ID = #value#
</statement>
<statement id="selectCategory" parameterClass="int" resultMap="select-category-result">
  select * from CATEGORY where CAT_ID = #value#
</statement>
```

In the above example, an instance of Product has a 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 iBatis Data Mapper 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-categoryresult Result Map will instantiate a Category and populate it. The whole Category instance then gets set into the Product's category property.

3.4.7. 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="select-product-result" class="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="selectProduct" parameterClass="int" resultMap="select-product-result">
  select *
  from PRODUCT, CATEGORY
  where PRD_CAT_ID=CAT_ID
  and PRD_ID = #value#
</statement>
```

3.4.7.1. 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 Data Mapper configuration 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. "==").

3.4.8. 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 iBATIS Data Mapper 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="select-category-result" class="category">
  <result property="id" column="CAT_ID"/>
  <result property="description" column="CAT_DESCRIPTION"/>
  <result property="productList" column="CAT_ID" select="selectProductsByCatId"/>
</resultMap>

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

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

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

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

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

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

3.4.9.1. 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 Data Mapper configuration 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. “==”).

3.4.10. 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="select-order-result" class="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="selectOrderPayments"/>
</resultMap>

<statement id="selectOrderPayments"
  resultMap="select-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 iBatis Data Mapper 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 Data Map definition file that they are defined in. You can refer to a Result Map in another Data Map definition file by prefixing the name of the Result Map with the name of the SQL Map (set in the <sqlMap> root tag).

Note

If you are using the Microsoft SQL Server 2000 Driver for JDBC you may need to add Select-Method=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>)

3.5. Supported Types for Parameter Maps and Result Maps

The Java types supported by the iBatis Data Mapper framework for parameters are shown in Section 3.5.1 (Java) and Section 3.5.2 (.NET)

Table 2 shows the Supported Types for Parameter Maps and Result Maps for Java.

Table 2. Supported Types for Parameter Maps and Result Maps (Java only)

Java Type	JavaBean/Map Property	Mapping	Result Class/Parameter Class***	Type Alias**
[:TODO:]				

* 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 Data Mapper 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.

Table 3 shows the Supported Types for Parameter Maps and Result Maps for .NET.

Table 3. Supported Types for Parameter Maps and Result Maps (.NET only)

Common Type	Object/Map Mapping	Property	Result Class***	Class/Parameter	Type Alias**
[<i>TODO</i> :]					

3.6. Cache Models

Some values in a database are know to change more slowly than others. To improve performance, many developers like to cache often-used data, to avoid making unnecessary trips back to the database. iBatis provides its own caching system, that you configure through a <cacheModel> element.

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 Data Mapper configuration file. Cache models are configured using the cacheModel element as follows:

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

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

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

3.6.1. 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).

3.6.2. 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:

3.6.3. **“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".

Table 4. Reference types that can be used for a MEMORY cache

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).

3.6.4. **“LRU” (com.ibatis.db.sqlmap.cache.lru.LruCacheController)**

The LRU cache implementation uses an Least Recently Used algorithm to determines how objects are automati-

cally 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!

3.6.5. “FIFO” (com.ibatis.db.sqlmap.cache.fifo.FifoCacheController)

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

The FIFO implementation is configured as follows:

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

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

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/>

3.7. Dynamic SQL

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 iBATIS 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">
    (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 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.

3.7.1. 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.

3.7.1.1. 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)

Table 5.

<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.
<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>

3.7.2. Unary Conditional Elements

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

3.7.2.1. Unary Conditional Attributes:

prepend – the overridable SQL part that will be prepended to the statement (optional)

property – the property to be checked (required)

Table 6.

<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>

3.7.2.2. Parameter Present:

These elements check for parameter object existence.

3.7.2.3. Parameter Present Attributes:

prepend – the overridable SQL part that will be prepended to the statement (optional)

Table 7.

<isParameterPresent>	Checks to see if the parameter object is present (not null).
<isNotParameterPresent>	Checks to see if the parameter object is not present (null). Example Usage: <pre data-bbox="804 667 1393 741"><isNotParameterPresent prepend="AND"> EMPLOYEE_TYPE = 'DEFAULT' </isNotParameterPresent></pre>

3.7.2.4. Iterate:

This tag will iterate over a collection and repeat the body content for each item in a List

3.7.2.5. Iterate Attributes:

prepend – the overridable SQL part that will be prepended to the statement (optional)

property – a property of type java.util.List that is to be iterated over (required)

open – the string with which to open the entire block of iterations, useful for brackets (optional)

close – the string with which to close the entire block of iterations, useful for brackets (optional)

conjunction – the string to be applied in between each iteration, useful for AND and OR (optional)

Table 8.

<iterate>	Iterates over a property that is of type java.util.List Example Usage: <pre data-bbox="804 1368 1406 1458"><iterate prepend="AND" property="userNameList" open="(" close=")" conjunction="OR"> username=#userNameList[]# </iterate></pre> <p data-bbox="804 1480 1406 1621">Note: It is very important to include the square brackets[] at the end of the List property name when using the Iterate element. These brackets distinguish this object as an List to keep the parser from simply outputting the List as a string.</p>
-----------	--

3.7.3. Simple Dynamic SQL Elements

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

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

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

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

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

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

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

4. Java Developer Guide

4.1. Installing iBATIS Data Mapper for Java

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

Table 9. JAR files used by iBATIS Data Mapper

<i>Filename</i>	<i>Description</i>	<i>Required</i>
ibatis-common.jar	iBATIS Common Utilities	YES
ibatis-sqlmap.jar	iBATIS Data Mapper Framework	YES
ibatis-dao-1-x-x-b.jar	Legacy iBATIS Data Access Objects for backward compatibility.	NO
ibatis-compact	Legacy iBATIS Java API for backward compatibility.	NO

4.1.1. JAR Files and Dependencies

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

tional packages.

Table 10. When to use optional packages

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

4.1.2. Upgrading from version 1.x to version 2.x

4.1.2.1. 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 iBATIS Data Mapper too.

4.1.2.2. 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:

Example 25. Ant TaskDef

```
<target>
  <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).

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

When upgrading, it's a good idea to remove all existing (old) iBatis SQL Map 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.

Table 11. Old files versus new files

Old Files	New Files
.ibatis-db.jar After release 1.2.9b, this file was split into .ibatis-common.jar.ibatis-dao.jar .ibatis-sqlmap.jar	.ibatis-common.jar (required) .ibatis-sqlmap.jar (required).ibatis-dao-1-2-9-b.jar (opti .ibatis-compat (optional 1.x compatibility)
commons-logging.jar commons-logging-api.jar commons-collection commons-pool.jar oscache.jar jta.jar dbc2 xmlParserAPIs.jar jjdom.jar	commons-logging-1-0-3.jar (required) commons-collections-2-1.jar (optional) commons-dbcp-1-1 commons-pool-1-1.jar (optional) oscache-2-0-1.jar (opti jta-1-0-1a.jar (optional) jdbc2_0-stdext.jar (optional) xercesImpl-2-4-0.jar (optional) xmlParserAPIs-2-4-0.jar xalan-2-5-2.jar (optional) log4j-1.2.8.jar (optional) cglib-full-2-0-rc2.jar (optional)

4.2. Configuring the Data Mapper for Java

iBatis Data Mapper is configured using a central XML descriptor, which provides the detail for your data source, data maps, and other features like caching, transactions, and thread management. At runtime, your application code calls a iBatis library routine which reads and parses the main configuration file. Other XML descriptors may be incorporated by reference, but each Data Mapper client instance "boots" from a single configuration file.

4.2.1. Data Mapper clients

Each Data Mapper client (instance of SqlMapClient) is created by reading a single configuration file. Each configuration file can specify only one database or datasource. However, you can use multiple Data Mapper clients

in your application. Just create another configuration file and pass the name of that file when the Data Mapper client is created. The configuration files might use a different account with the same database, or reference different databases on different servers. You can even read from one client and write to another, if that's what you need to do.

4.2.2. Data Mapper Configuration File (SqlMapConfig.xml)

A sample configuration file is shown in the following example:

Example 26. SqlMapConfig.xml for Java

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

<sqlMapConfig>

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

  <settings
    cacheModelsEnabled="true"
    enhancementEnabled="true"
    maxSessions="64"
    maxTransactions="8"
    maxRequests="128"/>

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

  <sqlMap resource="com.ibatis/jpetstore/persistence/sqlmapdao/sql/Account.xml"/>
  <sqlMap resource="com.ibatis/jpetstore/persistence/sqlmapdao/sql/Category.xml"/>
  <sqlMap resource="com.ibatis/jpetstore/persistence/sqlmapdao/sql/Product.xml"/>
  <sqlMap resource="com.ibatis/jpetstore/persistence/sqlmapdao/sql/Sequence.xml"/>
  <sqlMap resource="com.ibatis/jpetstore/persistence/sqlmapdao/sql/LineItem.xml"/>
  <sqlMap resource="com.ibatis/jpetstore/persistence/sqlmapdao/sql/Order.xml"/>
  <sqlMap resource="com.ibatis/jpetstore/persistence/sqlmapdao/sql/Item.xml"/>

</sqlMapConfig>
```

4.2.3. Data Mapper Configuration Elements

The next few sections describe the elements of the Data Mapper configuration document for Java.

4.2.3.1. The <properties> Element

Sometimes the values we use in an XML configuration file occur in more than one element. Often, there are values that change when we move the application from one server to another. To help you manage configuration values, you can specify a standard properties file (name=value) as part of a Data Mapper configuration. Each named value in the properties file becomes a "shell" variable that can be used throughout the Data Mapper configuration, including the Data Map definition files (see Section 3). For example, if the properties file contains

```
username=iBATIS
```

then any element in the Data Mapper configuration, including the data maps, can use the variable `${username}` to insert the value "iBATIS". For example:

```
<dataSource connectionString="user id=${username};"
```

Properties are handy during building, testing, and deployment. Properties make it easy to reconfigure your application for multiple environments or use automated tools for configuration (e.g. Ant or NAnt).

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" />
```

4.2.3.1.1. <properties> attributes

There can be only one <properties> element, which can accept one of two attributes: resources and url.

resource - A properties file found somewhere on the classpath.

url - A properties found on the local file system or other uniform locaiton.

4.2.3.2. The <settings> Element

There are a number of minimums, maximums, and other settings used by the framework. The settings appropriate for one application may not be appropriate for another. The <settings> element lets you configure these options and optimizations for the SqlMapClient instance that is created from the XML document. All of the settings have defaults, and you can omit the setting element or any of its attributes. The setting attributes and their behavior they control are described in Table 9.

Table 12. Attributes of the settings element

<i>cacheModelsEnabled</i>	<p>This setting globally enables or disables all cache models for an SqlMapClient. This can come in handy for debugging.</p> <pre>Example: cacheModelsEnabled="true" Default: true (enabled)</pre>
<i>useStatementNamespaces (Java)</i>	<p>With this setting enabled, you must always refer to mapped statements by their fully qualified name, which is the combination of the sqlMap name and the statement name. For example:</p> <pre>queryForObject("sqlMapName.statementName"); (See the Developers Guide for your platform (Section 4 or 5) Example: useStatementNamespaces="false" Default: false (disabled)</pre>
<i>maxRequests</i>	<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> <pre>Example: maxRequests="256" Default: 512</pre>
<i>maxSessions</i>	<p>This is the number of sessions (or clients) that can be active at a given time. A session is either an explicit</p>

	<p>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 following memory footprint.</p> <p>Example: maxSessions="64" Default: 128</p>
<i>maxTransactions</i>	<p>This is the maximum number of threads that can enter SqlMapClient.startTransaction() at a time. Threads beyond the set value will be blocked until another thread exits. Different DBMS have different limits, but no database is without these limits. This value should always be less than or equal to maxSessions and always much less than maxRequests. Often reducing the maximum number of concurrent comes in transactions can increase performance.</p> <p>Example: maxTransactions="16" Default: 32</p>
<i>lazyLoadingEnabled</i>	<p>This setting globally enables or disables all lazy loading for an SqlMapClient. This can come in handy for debugging.</p> <p>Example: lazyLoadingEnabled="true" Default: true (enabled)</p>
<i>enhancementEnabled</i>	<p>This setting enables runtime bytecode enhancement to facilitate optimized JavaBean property access as well as enhanced lazy loading.</p> <p>Example: enhancementEnabled="true" Default: false (disabled)</p>
<i>errorTracingEnabled</i>	TODO:

4.2.3.3. The <typeAlias> Element

The typeAlias element lets you specify a shorter name in lieu of fully-qualified classname. For example:

```
<typeAlias name="Account" assembly="NPetshop.Domain.dll"
  class="NPetshop.Domain.Accounts.Account" />
```

You can then refer to "account" or "Account" where you would normally have to spell-out the fully qualified class name.

4.2.3.3.1. <typeAlias> Attributes

<sect6>

The <typeAlais> element has two required properites:

alias - A unique identifier for this alais

class - The fully-qualified classname, including package reference

</sect6>

4.2.3.3.3. Predefined type aliases

There are several predefined aliases for Java, they are:

Table 13. Transaction Manager Aliases (Java)

```
JDBC = com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransactionConfig
JTA = com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig
EXTERNALcom.ibatis.sqlmap.engine.transaction.external.ExternalTransactionConfig
```

Table 14. Data Source Factory Aliases (Java)

```
SIMPLE = com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory
DBCP = com.ibatis.sqlmap.engine.datasource.DbcDataSourceFactory
JNDI = com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory
```

4.2.3.4. The <transactionManager> Element

The .NET implementation only supports ADO-controlled transaction. The TransactionManager element is not supported by the .NET implementation since there is only one manager available (ADO).

4.2.3.4.1. The <dataSource> Element (within transactionManger)

Three DataSource factories are provided with the Java implementation, and you can also write your own. The bundled DataSourceFactory implementations are discussed in the next three sections.

<sect6>

<title>SimpleDataSourceFactory</title>

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.

Example 27. TransactionManager element for SimpleDataSourceFactory (Java)

```
<transactionManager type="JDBC"> <dataSource type="SIMPLE"> <property name="JDBC.Driver" value="org.postg
```

```
</sect6>
```

```
<sect6>
```

```
<title>DbcpDataSourceFactory</title>
```

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:

Example 28. TransactionManager element for DbcpDataSourceFactory (Java)

```
<transactionManager type="JDBC"> <dataSource type="DBCP"> <property name="JDBC.Driver" value="{driver}"/
```

```
</sect6>
```

```
<sect6>
```

```
<title>JndiDataSourceFactory</title>
```

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:

Example 29. `TransactionManager` element for `JndiDataSourceFactory` (Java)

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

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:

Example 30. `TransactionManager` element configured for global transactions (Java)

```
<transactionManager type="JTA" > <property name="UserTransaction" value="java:/ctx/con/UserTransaction"/>
```

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 Data Map can take part in a wider scoped transaction involving other databases and transactional resources.

```
</sect6>
```

4.2.3.5. The `<sqlMap>` Element

On a daily basis, most of your work will be with the Data Maps. The Data Maps define the actual SQL statements or stored procedures used by your application. The parameter and result objects are also defined as part of the Data Map. As your application grows, you may have several varieties of Data Map. To help you keep your Data Maps organized, you can create any number of Data Map definition files and incorporate them by reference into the Data Mapper configuration. All of the definition files used by a Data Mapper instance must be listed in the configuration file.

Example 32 shows `<sqlMap>` elements for loading a set of Data Map definitions, either from the classpath or an URL.

Example 31. Data Map elements

```
<!-- CLASSPATH RESOURCES (Java) -->
<sqlMap resource="com/ibatis/jpetstore/persistence/sqlmapdao/sql/Category.xml" />
<sqlMap resource="com/ibatis/jpetstore/persistence/sqlmapdao/sql/Product.xml" />

<!-- URL RESOURCES (absolute) -->
<sqlMap url="file:///c:/config/Category.xml" />
<sqlMap url="file:///c:/config/Product.xml" />

<!-- URL RESOURCES (relative) -->
<sqlMap file="maps/Category.xml" />
<sqlMap file="Mmps/Product.xml" />
```

Section 3 describes the Data Map definition files.

4.3. Programming with iBATIS Data Mapper: The Java API

The `SqlMap` client API is meant to be simple and minimal. It provides the programmer with the ability to do

four primary functions: configure a Data 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.

4.3.1. Configuration

Configuring iBATIS Data Mapper is trivial once you have created your Data Map definition files and Data Mapper 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);
```

4.3.2. Transactions

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

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

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

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

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

An example of using transactions is as follows:

Example 32. Using transactions

```
private Reader reader = new Resources.getResourceAsReader ("com/ibatis/example/sqlMapconfig.xml");
private SqlMapClient sqlMap = XmlSqlMapBuilder.buildSqlMap(reader);

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

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

Note

Transactions cannot be nested. Calling .startTransaction() from the same thread more than once, before

calling `commit()` or `rollback()`, will cause an exception to be thrown. In other words, each thread can have -at most- one transaction open, per `SqlMapClient` instance.

Note

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

4.3.2.1. 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:

Example 33. Setting up an automatic transaction

```
private Reader reader = new Resources.getResourceAsReader ("com/ibatis/example/sqlMapconfig.xml");
private SqlMapClient sqlMap = XmlSqlMapBuilder.buildSqlMap(reader);

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

Note

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

4.3.3. Global (DISTRIBUTED) Transactions

The iBATIS 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).

4.3.3.1. 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 Data Mapper 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 Data Mapper code to make use of a global transaction.

4.3.3.2. Managed Global Transactions

The iBATIS Data Mapper framework can also manage global transactions for you. To enable support for managed global transactions, you must set the `<transactionManager>` type attribute to “JTA” in your SQL Map configuration file and set the “UserTransaction” property to the full JNDI name of where the SqlMapClient instance will find the UserTransaction instance. See the `<transactionManager>` discussion above for full configuration details.

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

```
try {
    orderSqlMap.startTransaction();
    storeSqlMap.startTransaction();
    orderSqlMap.insertOrder(...);
    orderSqlMap.updateQuantity(...);
    storeSqlMap.commitTransaction();
    orderSqlMap.commitTransaction();
}
finally {
    try {
        storeSqlMap.endTransaction()
    }
    finally {
        orderSqlMap.endTransaction()
    }
}
```

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

Warning

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

4.3.4. 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 iBATIS Data Mapper API, two simple methods allow you to demarcate the boundaries of the batch:

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

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

4.3.5. Executing Statements via the `SqlMapClient` API

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

Example 34. iBATIS Data Mapper Client API

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

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

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

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

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

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

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

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

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

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

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

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

4.3.5.1. `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.

4.3.5.2. `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.

4.3.5.3. `queryForList()`:

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

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

```
public void handleRow (Object object, List list);
```

4.3.5.4. `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.

4.3.5.5. `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).

4.3.5.6. Examples

Example 35. Executing Update (insert, update, delete)

```
sqlMap.startTransaction();
Product product = new Product();
product.setId (1);
product.setDescription ("Shih Tzu");
int rows = sqlMap.insert ("insertProduct", product);
sqlMap.commitTransaction();
```

Example 36. Executing Query for Object (select)

```
sqlMap.startTransaction();
Integer key = new Integer (1);
Product product = (Product)sqlMap.queryForObject ("getProduct", key);
sqlMap.commitTransaction();
```

Example 37. 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 38. Executing Query for List (select)

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

Example 39. Auto-commit

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

Example 40. Executing Query for List (select) With Result Boundaries

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

Example 41. Executing Query with a RowHandler (select)

```
public class MyRowHandler implements RowHandler {
    public void handleRow (Object object, List list) throws
        SQLException {
        Product product = (Product) object;
        product.setQuantity (10000);
        sqlMap.update ("updateProduct", product);
        // Optionally you could add the result object to the list.
        // The list is returned from the queryForList() method.
    }
}
sqlMap.startTransaction();
RowHandler rowHandler = new MyRowHandler();
List list = sqlMap.queryForList ("getProductList", null, rowHandler);
sqlMap.commitTransaction();
}
```

Example 42. Executing Query for Paginated List (select)

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

Example 43. Executing Query for Map

```
sqlMap.startTransaction();
Map map = sqlMap.queryForMap ("getProductList", null, "productCode");
sqlMap.commitTransaction();
Product p = (Product) map.get("EST-93");
```

4.4. The One Page JavaBeans Course

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

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

That said, here is an example of a JavaBean:

Example 44. A typical JavaBean

```
public class Product implements Serializable {
    private String id;
    private String description;

    public String getId() {
        return id;
    }

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

    public String getDescription() {
        return description;
    }

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

Note

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

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

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

Note

Similarly, make sure you only have one method named getXxxx() and setXxxx(). Be judicious with

polymorphic methods. You're better off naming them more specifically anyway. As a Stream: For simple read-only binary or text data.

Note

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

Congratulations! You've passed the course!

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 iBATIS 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 (ResultMap, ParameterMap etc...) as follows (given the employee object as above): "departmentList[3].address.city"

4.5. Logging SqlMap Activity with Jakarta Commons Logging

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

Table 15. Logging Frameworks supported by iBATIS Data Mapper

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/

4.5.1. Log Configuration

Configuring the commons logging services is very simply a matter of including one or more extra configuration files (e.g. "log4j.properties") and sometimes a new JAR file (e.g. "log4j.jar"). The following example configuration will configure full logging services using Log4J as a provider. There are 2 steps.

4.5.1.1. Step 1: Add the Log4J JAR file

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

4.5.1.2. Step 2: Configure Log4J

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

Example 45. `log4j.properties`

```
1 # Global logging configuration
2 log4j.rootLogger=ERROR, stdout
3 # SqlMap logging configuration...
4 #log4j.logger.com.ibatis=DEBUG
5 #log4j.logger.com.ibatis.common.jdbc.SimpleDataSource=DEBUG
6 #log4j.logger.com.ibatis.common.jdbc.ScriptRunner=DEBUG
7 #log4j.logger.com.ibatis.sqlmap.engine.impl.SqlMapClientDelegate=DEBUG
8 #log4j.logger.java.sql.Connection=DEBUG
9 #log4j.logger.java.sql.Statement=DEBUG
10 #log4j.logger.java.sql.PreparedStatement=DEBUG
11 #log4j.logger.java.sql.ResultSet=DEBUG
12 # Console output...
13 log4j.appender.stdout=org.apache.log4j.ConsoleAppender
14 log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
15 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 `'Data Mapper logging configuration'` section of the file above (commented out in lines 5 through 12 above). So if we wanted to log `PreparedStatement` activity (SQL statements) to the console at the `DEBUG` level, we would simply change line 11 to the following (notice it's not `#commented out anymore`):

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

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

4.6. SimpleDataSource (com.ibatis.common.jdbc.*)

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

Note

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

Note

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

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

Table 16. SimpleDataSource properties

Property Name	Required	Default	Description
[<i>:TODO:</i>]			

Example 46. Using SimpleDataSource

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

4.7. ScriptRunner (com.ibatis.common.jdbc.*)

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

Example 47. Script: initializeHere are some examples:-db.sql

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

Example 48. Usage: Using an Existing Connection

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

Example 49. Usage: Using a New Connection

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

Example 50. Usage 2: Using a New Connection from Properties

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

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

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

A few methods that you may find useful are:

```
// if you want the script runner to stop running after a single error
scriptRunner.setStopOnError (true);
// if you want to log output to somewhere other than System.out
scriptRunner.setLogWriter (new PrintWriter(...));
```

4.8. 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 Data Mapper configuration file (e.g. "sqlMap-config.xml") from the classpath.
- Loading the DAO Manager configuration file (e.g. "dao.xml") from the classpath
- Loading various *.properties files from the classpath.
- Etc.

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

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

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

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

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

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

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

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

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

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

5. .NET Developer Guide

This guide explains how to install, configure, and use the iBatis Data Mapper with your .NET applications. This guide assumes that you are using Microsoft Visual Studio .NET (VSN). If you are using another IDE, please adapt these instructions accordingly.

5.1. Installing the Data Mapper for .NET

There are three steps to using iBatis Data Mapper with your application for the first time.

- 1.
- 2.
- 3.

5.1.1. Setup the Distribution

The official site for iBatis Data Mapper for .NET is our SourceForge site <<http://ibatisnet.sf.net/>>. To download the distribution, follow the link to the Files area, and select the iBatis.NET V1.0 or later release. The distribution is in the form of a ZIP archive. You can extract the distribution using a utility, like WinZip, or (if you must) the extractor built into newer versions of Windows. We suggest that you create an "ibatisnet" folder in your VSN project directory and extract the distribution there.

Under the distribution's "source" folder are five other folders that make up the iBatis.NET distribution, as shown in Table 17.

Table 17. Folders found in the iBatis.NET distribution

External-Bin	Precompiled assemblies provided for your convenience.
IBatisNet-Common	Assembly of classes shared by DataAccess and DataMapper
IBatisNet-DataAccess	The Data Access Objects framework (see separate DAO Guide)
iBatisNet-DataMapper	The Data Mapper framework
IBatisNet-Test	NUnit tests for the solution

You can load the "IBatisNet.sln" solution file into Visual Studio and build the solution to generate the needed assemblies. There are four projects in the solution, and all should succeed. The assemblies we need will be created under "\source\IBatisNet.DataMapper\bin\Debug".

5.1.2. Add Assembly References

Switching to your own solution, open the project that will be using the iBatis.NET Data Mapper. Depending on how you organize your solutions, this might not be the project for your Windows or Web application. It may be a library project that your application projects reference. To your project, you need to add two references:

- 1.
- 2.

If you have built the IBatisNet solution as described in Section 5.1.1, both assemblies should be in the bin/debug folder under the IBatisNet-DataMapper project.

5.1.3. Add XML File Items

You will need to add two or more XML file items to your Windows or Web application project (and Test project if you have one). These files are:

SqlMap.config - The Data Mapper configuration file (usually one)

SqlMap.xml - The Data Map definition file (usually one or more, with various names)

The "SqlMapper.config" file must be placed where the framework can find it at runtime. The default location differs by the type of project, as shown in Table 18

Table 18. Where to place the SqlMap.config file

Windows or Test projects (using NUnit or equivalent)	Place in /debug/bin, with the assembly (.dll) files
Web projects	Place in the project root, with the "web.config" file

The "SqlMapper.config" file includes a reference to your Data Map definition files (e.g. "SqlMap.xml"). For a Web project, you may wish to put the map definitions in a folder called "Maps" or "Resources", and then refer to the maps in the form "Maps/Account.xml". For a Windows or Test project, you can do the same, and use a relative reference like "../Maps/Account.xml" in the "SqlMapper.config" file.

Of course, the "SqlMapper.config" file must be in the correct format, which is described in Section 5.2. The format for the Data Map descriptions ("SqlMap.xml" files) is covered by Section 3.

5.2. Configuring the Data Mapper for .NET

iBATIS Data Mapper is configured using a central XML descriptor file, usually named "SqlMapper.config", which provides the detail for your data source, data maps, and other features like caching, transactions, and thread management. At runtime, your application code calls a iBATIS library routine (see Section 5.3) which reads and parses the main configuration file. Other XML descriptors may be incorporated by reference, but each Data Mapper client instance "boots" from a single configuration file.

5.2.1. Data Mapper clients

Each Data Mapper client (instance of SqlMapper) is created by reading a single configuration file. Each configuration file can specify only one database or datasource. However, you can use multiple Data Mapper clients in your application. Just create another configuration file and pass the name of that file when the Data Mapper client is created. The configuration files might use a different account with the same database, or reference different databases on different servers. You can even read from one client and write to another, if that's what you need to do.

5.2.2. Data Mapper Configuration File (SqlMapper.config)

A sample configuration file for .NET is shown in Example 51.

Example 51. SqlMap.Config for .NET

```
<?xml version="1.0" encoding="utf-8"?>
<sqlMapConfig
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ibatisnet.sf.net/schemaV1/SqlMapConfig.xsd">
  <properties file="database.properties"/> <!-- TODO: implementation -->
```

```
<settings>
  <setting useFullyQualifiedStatementNames="false" />
  <setting cacheModelsEnabled="true" />
</settings>

<providers file="providers.config" />

<database>
  <dataSource name="NPetshop" default="true"
    connectionString="
      user id=${username};
      password=${password};
      data source=${datasource};
      database=${database};
      connection reset=false;
      connection lifetime=5;
      min pool size=1;
      max pool size=50" />
</database>

<sqlMaps>
  <sqlMap file="Maps/Category.xml" />
  <sqlMap file="Maps/Product.xml" />
  <sqlMap file="Maps/Item.xml" />
  <sqlMap file="Maps/Account.xml" />
  <sqlMap file="Maps/Sequence.xml" />
  <sqlMap file="Maps/LineItem.xml" />
  <sqlMap file="Maps/Order.xml" />
</sqlMaps>
The
</sqlMapConfig>
```

5.2.3. Data Mapper Configuration Elements

Sections 5.1.3.1 through 5.1.3.8 describes the elements of the Data Mapper configuration document for .NET.

5.2.3.1. The `<properties>` Element [TODO: implementation]

Sometimes the values we use in an XML configuration file occur in more than one element. Often, there are values that change when we move the application from one server to another. To help you manage configuration values, you can specify a standard properties file (name=value) as part of a Data Mapper configuration. Each named value in the properties file becomes a "shell" variable that can be used throughout the Data Mapper configuration, including the Data Map definition files (see Section 3). For example, if the properties file contains

```
username=iBatis
```

then any element in the Data Mapper configuration, including the data maps, can use the variable `${username}` to insert the value "iBatis". For example:

```
<dataSource connectionString="user id=${username};"
```

Properties are handy during building, testing, and deployment. Properties make it easy to reconfigure your application for multiple environments or use automated tools for configuration (e.g. NAnt).

5.2.3.1.1. `<properties>` attributes

There can be only one `<properties>` element which can accept one required attribute, "file", being the path to a properties file.

5.2.3.2. The `<settings>` Element

There are a number of minimums, maximums, and other settings used by the framework. The settings appropriate for one application may not be appropriate for another. The `<settings>` element lets you configure these options and optimizations for the SqlMapper instance that is created from the XML document. All of the settings have defaults, and you can omit the setting element or any of its attributes. The setting attributes and the behavior they control are described in Table 17.

Table 19. Attributes of the settings element

<i>cacheModelsEnabled</i>	<p>This setting globally enables or disables all cache models for an SqlMapClient. This can come in handy for debugging.</p> <p>Example: <code>cacheModelsEnabled="true"</code> Default: true (enabled)</p>
<i>useStatementNamespaces</i>	<p>With this setting enabled, you must always refer to mapped statements by their fully qualified name, which is the combination of the sqlMap name and the statement name. For example:</p> <p><code>queryForObject("sqlMapName.statementName");</code> Example: <code>useStatementNamespaces="false"</code> Default: false (disabled)</p>

5.2.3.3. The <typeAlias> Element

The typeAlias element lets you specify a shorter name in lieu of fully-qualified classname. For example:

```
<typeAlias alias="LineItem" assembly="NPetshop.Domain.dll" class="NPetshop.Domain.Billing.LineItem" />
```

You can then refer to "account" or "Account" where you would normally have to spell-out the fully qualified class name.

Note

In the .NET implementation, zero or more <typeAlias> elements can appear in the Data Map definition file, within an enclosing <alias> element.

5.2.3.3.1. <typeAlias> attributes

The <typeAlias> element has three attributes:

- alias* - A unique identifier for this element
- assembly* - The name of the assembly where the class resides
- class* - The fully-qualified classname, including namespace reference

5.2.3.3.2. Predefined type aliases

The .NET platform predefines some aliases that you can use in your Data Mapper configuration file, as shown in Table 18.

Table 20. Predefined Aliases

<i>Alias</i>	Classname
HashTable	System.Collections.HashTable

5.2.3.4. The <providers> Element

Under ADO.NET, a database system is accessed through a provider. A database system can use a custom provider or a generic ODBC provider. iBATIS.NET uses a pluggable approach to installing providers. Each provider is represented by an XML descriptor element. The list of providers you might want to use can be kept in a separate XML descriptor file. The iBATIS.NET distribution includes a standard "providers.config" file with

a set of six prewritten provider elements:

SqlServer 1.0
SqlServer 1.1
OleDb 1.1 (Access)
ODBC 1.1
Oracle 9.2
ByteFx (MySQL)

The standard "providers.config" file can be found under "\source\IBatisNet.Test\bin\Debug" in the iBatis.NET distribution (see Section 5.1). To use it, you must copy it to your project and include a relative file reference in the <providers> element.

A provider may require libraries that you do not have installed, and so the provider element allows you to disable unused providers. One provider can also be marked as the "default" and will be used if another is not specified by your configuration (see Section 5.2.3.5).

The standard "provider.config" file has SqlServer 1.1 set as the default, and the SqlServer 1.0 provider disabled. The Oracle and ByteFx providers are also disabled. (Oracle is proprietary software, and ByteFx ships under a more restrictive license.)

Important

ByteFx is the recommended provider if you are using MySQL. You may download ByteFx from the MySQLNet SourceForge site (<http://sf.net/projects/mysqlnet/>). If the ByteFx license is acceptable to you, you may install it as a reference within your application and enable the ByteFx provider. ByteFx is slated to join the MySQL project and will be available there as well.

Tip

Be sure to review the "providers.config" file and confirm that the provider you intend to use is enabled! (Set the enabled attribute to true.)

5.2.3.5. The <database> Element

The <database> element encloses elements which configure the database system for use by the framework. These are the <provider>, <datasource>, and <transactionManager> elements.

5.2.3.5.1. The <provider> Element

If the default provider is being used, the <provider> element is optional. Or, if several providers are available, one may be selected using the provider element without altering the "providers.config" file.

```
<provider name="OleDb1.1" />
```

5.2.3.5.2. The <datasource> element

The <datasource> element specifies ODBC datasource or connection string. Example 52 shows sample elements for SQL Server, Oracle, Access, and MySQL.

Example 52. Sample <datasource> and <provider> elements (.NET)

```
<!-- The ${properties} are defined in an external file, -->  
<!-- but the values could also be coded inline. -->  
  
<!-- Connecting to SQL Server -->  
<database>  
  <provider name="sqlServer1.1" />  
  <datasource name="NPetstore" default="true"  
    connectionString="data source=(local)\NetSDK;database=${database};" />  
</database>
```

```
user id=${username};password=${password};connection reset=false;
connection lifetime=5; min pool size=1; XML descriptor filesmax pool size=50"/>
</database>

<!-- Connecting to Oracle -->
TODO:

<!-- Connecting to Access -->
<database>
  <provider name="OleDb1.1" />
  <dataSource name="NPetstore" default="true"
    connectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=${database}" />
</database>

<!-- Connecting to a MySQL database -->
<database>
  <provider name="ByteFx" />
  <dataSource name="NPetstore" default="true"
    connectionString="Host=${host};Database=${database};
    Password=${password};Username=${username}" />
</database>
```

5.2.3.5.3. The <transactionManager> Element [TODO: implementation]

TODO:

5.2.3.6. The <sqlMap> Element

On a daily basis, most of your work will be with the Data Maps, which are covered by Section 3. The Data Maps define the actual SQL statements or stored procedures used by your application. The parameter and result objects are also defined as part of the Data Map. As your application grows, you may have several varieties of Data Map. To help you keep your Data Maps organized, you can create any number of Data Map definition files and incorporate them by reference into the Data Mapper configuration. All of the definition files used by a Data Mapper instance must be listed in the configuration file.

Example 53 shows <sqlMap> elements for loading a set of Data Map definition. Note that the <sqlMap> elements are nested in a <dataMaps> element.

Example 53. SqlMap elements

```
<dataMaps>
  <sqlMap file="Maps/Category.xml" />
  <sqlMap file="Maps/Product.xml" />
</dataMaps>
```

For more about Data Map definition files, see Section 3.

5.3. Programming with iBATIS Data Mapper: The .NET API

The SqlMapper API provides four core functions:

1. build a SqlMapper instance from a configuration file
2. execute an update query (including insert and delete).
3. execute a select query for a single object
4. execute a select query for a list of objects

The API also provides support for retrieving paginated lists and managing transactions.

5.3.1. Building a SqlMapper Instance

An XML document is a wonderful tool for describing a database configuration (Section 5.2) or defining a set of data mappings (Section 3), but you can't "execute" XML. In order to use the iBatis.NET configuration and definitions in your .NET application, you need a class you can call.

The framework provides service methods that you can call which read the configuration file (and any of its definition files) and builds a "SqlMapper" object. The SqlMapper object provides access to the rest of the framework. The SqlMapper is designed to be multi-threaded and long-lived, and so makes for a good singleton. Example 54 shows a singleton Mapper that you can use in your own applications.

Example 54. A Mapper singleton you can use in your own applications

```
using IBatisNet.DataMapper;

namespace MyApp.Library.Services
{
    /// <summary>
    /// Gateway class to provide access to the data maps.
    /// If you are not using managed transactions, this may be the
    /// only class that needs to cite IBatisNet in a using clause.
    /// </summary>
    internal class Mapper
    {
        private static volatile SqlMapper mapper = null;

        protected static void Configure (object obj)
        {
            mapper = (SqlMapper) obj;
        }

        protected static void InitMapper ()
        {
            ConfigureHandler handler = new ConfigureHandler (Configure);
            map = SqlMapper.ConfigureAndWatch(handler);
        }

        public static SqlMapper Get ()
        {
            if (mapper == null)
            {
                lock (typeof (Mapper))
                {
                    if (mapper == null) // double-check
                        InitMapper ();
                }
            }
            return mapper;
        }
    }
}
```

To obtain the SqlMapper instance, just call

```
SqlMapper mapper = Mapper.Get();
```

anywhere in your application, and specify one of the SqlMapper methods (see Section 5.3.2) . Here's an example:

```
IList list = Mapper.Get().QueryForList ("PermitNoForYearList", values);
```

The first time "Mapper.Get()" is called, it will read the default configuration file, "SqlMap.config". On subsequent calls, it will reuse the cached "mapper" instance. The ConfigureAndWatch method monitors changes to

the configuration files. If the configuration or definitions files do change, the `SqlMapper` will be safely reloaded. This is particularly useful in development, when you might make a change to a Data Map definition and want to see it take effect without restarting a debugging session. Likewise, in production, it can allow you to make changes to the definitions without reloading the rest of the application.

If for some reason you do not want to monitor changes to the configuration, you can use the `Configure` method instead:

```
mapper = SqlMapper.Configure(handler);
```

5.3.1.1. Multiple Databases

If you need access to more than one database from the same application, create a Data Mapper configuration class for that database and another "Mapper" class to go with it. In the new "Mapper" class, change the call to `ConfigureAndWatch` to

```
mapper = SqlMapper.ConfigureAndWatch("anotherConfig.config", handler);
```

and substitute the name of your configuration class. Each database then has their own singleton you can call from your application:

```
SqlMapper sqlServer = SqlServerMapper.Get();  
SqlMapper access = AccessMapper.Get();
```

5.3.2. Exploring the `SqlMapper` API

The `SqlMapper` instance acts like a facade to provide access the rest of the Data Mapper framework. The Data Mapper API methods are shown in Example 54.

Example 55. The Data Mapper API for .NET

```
<!-- Query API -- see Sections 5.3.2.1 through 5.3.2.5 -->  
public object Insert(string statementName, object parameterObject);  
public int Update(string statementName, object parameterObject);  
public int Delete(string statementName, object parameterObject);  
  
public object QueryForObject(string statementName, object parameterObject);  
public object QueryForObject(string statementName, object parameterObject, object resultObject);  
  
public IList QueryForList(string statementName, object parameterObject);  
public void QueryForList(string statementName, object parameterObject, IList resultObject);  
public IList QueryForList(string statementName, object parameterObject, int skipResults, int maxResults);  
  
public IList QueryForRowDelegate(string statementName, object parameterObject, RowDelegate rowDelegate);  
  
public PaginatedList QueryForPaginatedList(String statementName, object parameterObject, int pageSize);  
  
public IDictionary QueryForDictionary(string statementName, object parameterObject, string keyProperty)  
public IDictionary QueryForDictionary(string statementName, object parameterObject, string keyProperty, string valueProperty)  
public IDictionary QueryForMap(string statementName, object parameterObject, string keyProperty)  
public IDictionary QueryForMap(string statementName, object parameterObject, string keyProperty, string valueProperty)  
  
<!-- Transaction API -- see Section 5.3.3 -->  
public void BeginTransaction()  
public void CommitTransaction()  
public void RollBackTransaction()
```

Note that each of the API methods accept the name of the Mapped Statement is passed in as the first parameter. The `statementName` parameter corresponds to the id of the Mapped Statement in the Data Map definition (see Section 3.2.4.1). In each case, a `parameterObject` also may be passed. If the Mapped Statement expects no parameters, a null `parameterObject` may be passed. If a statement does expect parameters, then a valid `parameterObject` is required. Section 5.3.2.1 through 5.3.2.5 describe how the API methods work.

5.3.2.1. Insert, Update, Delete

```
public object Insert(string statementName, object parameterObject);
public int Update(string statementName, object parameterObject);
public int Delete(string statementName, object parameterObject);
```

If a Mapped Statement uses one of the <insert>, <update>, or <delete> statement-types, then it should use the corresponding API method. The <insert> element supports a nested <selectKey> element for generating primary keys (see Section 3.2.3.2). If the <selectKey> stanza is used, then Insert returns the generated key; otherwise null. Both Update and Delete return the number of rows effected.

5.3.2.2. QueryForObject

```
public object QueryForObject(string statementName, object parameterObject);
public object QueryForObject(string statementName, object parameterObject, object resultObject);
```

If a Mapped Statement is expected to select a single row, then call it using QueryForObject. Since the Mapped Statement definition specifies the result class expected, the framework can both create and populate the result class for you. Alternatively, if you need to manage the result object yourself, say because it is being populated by more than one statement, you can use the alternate form and pass your resultObject as the third parameter.

5.3.2.3. QueryForList

```
public IList QueryForList(string statementName, object parameterObject);
public void QueryForList(string statementName, object parameterObject, IList resultObject);
public IList QueryForList(string statementName, object parameterObject, int skipResults, int maxResults);
```

If a Mapped Statement is expected to select multiple rows, then call it using QueryForList . Each entry in the list will be an result object populated from the corresponding row of the query result. If you need to manage the resultObject yourself, then it can be passed as the third parameter.

If you need to obtain a partial result, the third form takes the number of records to skip (the starting point) and the maximum number to return, as the skipResults and maxResults parameters. The PaginatedList method (Section 5.3.2.5) provides the same functionality but in a more convenient wrapper. The QueryWithRowDelegate method (Section 5.3.2.4) also works with multiple rows, but provides a post-processing feature.

5.3.2.4. QueryWithRowDelegate

```
public delegate void RowDelegate(object obj, IList list);
public IList QueryWithRowDelegate(string statementName, object parameterObject, RowDelegate rowDelegate);
```

No matter how well our database is designed, or how cleverly we describe our maps, the result objects we get back may not be ideal. You may need to to perform some post-processing task on the result objects. You might even want to omit an entry omitted from the list. Or, you might want to use the result object to create some other, more useful object. To save filtering the result objects from to one list to another, you can pass iBATIS.NET a RowDelegate to do the dirty work. The delegate will be passed in turn each of the result objects and the IList reference. Your delegate can then cope with the object and add to the list, if appropriate.

Important

It is your responsibility to add the objects you want returned to the list. If an object isn't added it is not returned.

5.3.2.5. QueryForPaginatedList

```
public PaginatedList QueryForPaginatedList(String statementName, object parameterObject, int pageSize);
```

We live in an age of information overflow. A database query often returns more hits than users want to see at

once, and our requirements may say that we need to offer a long list of results a "page" at a time. If the query returns 100 hits, we might need to present the hits to the user in sets of ten, and let them move back and forth between the sets. Since this is such a common requirement, the framework provides a convenience method.

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.

Tip

The `PaginatedList` method is convenient, but note that a larger set will first be returned by the database provider and the smaller set extracted by the framework. The higher the page, the larger set that will be returned and thrown away. For very large sets, you may want to use a stored procedure or your own query that used "skipResults" and "maxResults" as parameters. Unfortunately, the semantics for the returning partial data sets is not standardized, so `PaginatedList` is the best we can do within the scope of the framework.

5.3.2.6. QueryForDictionary, QueryForMap

```
public IDictionary QueryForDictionary(string statementName, object parameterObject, string keyProperty)
public IDictionary QueryForDictionary(string statementName, object parameterObject, string keyProperty, string valueProperty)
public IDictionary QueryForMap(string statementName, object parameterObject, string keyProperty)
public IDictionary QueryForMap(string statementName, object parameterObject, string keyProperty, string valueProperty)
```

The `QueryForList` methods return the result objects within a `IList` instance. Alternatively, the `QueryForDictionary` returns a `IDictionary` instance. The value of each entry is one of the result object. The key to each entry is indicated by the `keyProperty` parameter. This is the name of the one of the properties of the result object, the value of which is used as the key for each entry. For example, If you needed a set of `Employee` objects, you might want them returned as a `Dictionary` keyed by each object's `EmployeeNumber` property.

If you don't need the entire result object in your `Dictionary`, you can add the `valueProperty` parameter to indicate which result object property should be the value of an entry. For example, you might just want the `EmployeeName` keyed by `EmployeeNumber`.

Important

You do not need to use this method just to obtain an `IDictionary` result object. As explained in Section 3.x.x., the result object for any query can be a property object or a `IDictionary` instance. This method is used to create a *new* `IDictionary` result object from a property object or (another) `IDictionary` object.

The `QueryforMap` methods provide the same functionality but under a different name, for the sake of continuity with the Java implementation. (The .NET `IDictionary` interface is equivalent to the Java `Map` interface.)

5.3.3. Using Explicit and Automatic Transactions

By default, calling any of the API methods (see Section 5.3.2) on an `SqlMapper` instance will auto-commit/rollback as a single transaction. This means that each call to to these methods will be a single unit of work. For many cases, this simple approach may be sufficient. But it is not ideal if you have a number of statements that must execute as a single unit of work, which is to say, succeed or fail as a group. For cases like these, you can use *explicit transactions*.

The Data Mapper API includes methods to demarcate transactional boundaries. A transaction can be started, committed and/or rolled back. You can call the transaction methods from the `SqlMapper` instance (see Section 5.3.1).

```
public void BeginTransaction()
public void CommitTransaction()
public void RollBackTransaction()
```

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 shown as Example 55.

Example 56. Using explicit transactions

[TODO:]

Note

Transactions cannot be nested. An exception will startTransaction(), commitTransaction() and endTransaction() be thrown if you call BeginTransaction from the same thread more than once, or call CommitTransaction or RollbackTransaction first. In other words, each thread can have -at most- one transaction open, per SqlMapper instance.

Note

SqlMap transactions use ThreadLocal store for storing transactional objects. This means that each thread that calls BeginTransaction 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. Not doing so could cause your pool to run out of connections and lock up.

5.3.3.1. Automatic Transactions

It is better to use explicit transactions. But for simple requirements (generally read-only, you can use a simplified semantic. If you do not demarcate query API statements using the *Transaction methods, the transactions will all be called automatically for you whenever you execute a statement outside of a transactional block. See Example 56 for an automatic transaction.

Example 57. Relying on automatic transactions

[TODO:]

Note

Be careful to consider transactions when framing your queries. Automatic transactions are convenient, but you will run into trouble if your unit of work requires more than a single update to the database. In Example 56, if the second call to “updateItem” fails, the item description will still be updated with the first new description of “TX1”. Not what a user might expect.

5.3.4. Coding Examples [TODO: Review from here]

Example 58. Executing Update (insert, update, delete)

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

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

Example 59. Executing Query for Object (select)

```
sqlMap.startTransaction();
Integer key = new Integer (1);
Product product = (Product)sqlMap.queryForObject ("getProduct", key);
sqlMap.commitTransaction();
```

Example 60. 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 61. Executing Query for List (select)

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

Example 62. Auto-commit

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

Example 63. Executing Query for List (select) With Result Boundaries

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

Example 64. Executing Query with a RowHandler (select)

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

```
List list = sqlMap.queryForList ("getProductList", null, rowHandler);
sqlMap.commitTransaction();
}
```

Example 65. Executing Query for Paginated List (select)

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

Example 66. Executing Query for Map

```
sqlMap.startTransaction();
Map map = sqlMap.queryForMap ("getProductList", null, "productCode");
sqlMap.commitTransaction();
Product p = (Product) map.get("EST-93");
```

5.4. Logging SqlMap Activity with Apache Log4Net

The iBATIS Data Mapper framework provides logging information through the use of Apache Log4Net (<http://logging.apache.org/log4net/>). The specifics of Log4Net are beyond the scope of this document. This section provides a sample configuration to help you get started.

5.4.1. Log Configuration

The framework uses Log4Net internally and will automatically include the assembly when your project is built. To use Log4Net with your own application, you merely need to provide your own Log4Net configuration. You can do this by adding a configuration file for your assembly that includes a log4Net element. The configuration file is named after your assembly but adds a .Config extension, and is stored in the same folder as your assembly. Example xx shows the configuration for the framework's Test project.

Example 67. A sample Log4Net configuration block (IBatisNet.Test.dll.Config)

```
<configuration>
  <!-- Register a section handler for the log4net section -->
  <configSections>
    <section name="log4net" type="System.Configuration.IgnoreSectionHandler" />
  </configSections>
  <appSettings>
    <!-- To enable internal log4net logging specify the following appSettings key -->
    <!-- <add key="log4net.Internal.Debug" value="true"/> -->

    <!-- To test MySql set value="MySql" (to do)-->
    <!-- To test Oracle set value="Oracle" (to do)-->
    <!-- To test MS Sql Server set value="MSSQL" -->
    <add key="database" value="MSSQL"/>
    <!-- To test MS SqlServer via SqlClient value="SqlClient" -->
    <!-- To test MS SqlServer via OleDb value="OleDb" -->
    <!-- To test MS SqlServer via Odbc value="Odbc" -->
    <add key="providerType" value="SqlClient"/>
  </appSettings>

  <!-- This section contains the log4net configuration settings -->
  <log4net>
    <!-- Define some output appenders -->
    <appender name="RollingLogFileAppender" type="log4net.Appender.RollingFileAppender">
      <param name="File" value="log.txt" />
      <param name="AppendToFile" value="true" />
      <param name="MaxSizeRollBackups" value="2" />
      <param name="MaximumFileSize" value="100KB" />
    </appender>
  </log4net>
</configuration>
```

```
<param name="RollingStyle" value="Size" />
<param name="StaticLogFileName" value="true" />
<layout type="log4net.Layout.PatternLayout">
  <param name="Header" value="[Header]\r\n" />
  <param name="Footer" value="[Footer]\r\n" />
  <param name="ConversionPattern" value="%d [%t] %-5p %c [%x] - %m%n" />
</layout>
</appender>
<appender name="ConsoleAppender" type="log4net.Appender.ConsoleAppender">
  <layout type="log4net.Layout.PatternLayout">
    <param name="ConversionPattern" value="%d [%t] %-5p %c [%x] &lt;%X{auth}&gt; - %m%n" />
  </layout>
</appender>

<!-- Set root logger level to ERROR and its appenders -->
<root>
  <level value="ERROR" />
  <appender-ref ref="RollingLogFileAppender" />
  <appender-ref ref="ConsoleAppender" />
</root>

<!-- Print only messages of level DEBUG or above in the packages -->
<logger name="IBatisNet.DataMapper.Configuration.Cache.CacheModel">
  <level value="DEBUG" />
</logger>
<logger name="IBatisNet.DataMapper.Configuration.Statements.PreparedStatementFactory">
  <level value="DEBUG" />
</logger>
<logger name="IBatisNet.DataMapper.LazyLoadList">
  <level value="DEBUG" />
</logger>
</log4net>
</configuration>
```

[TODO: Explanation of logging setup]