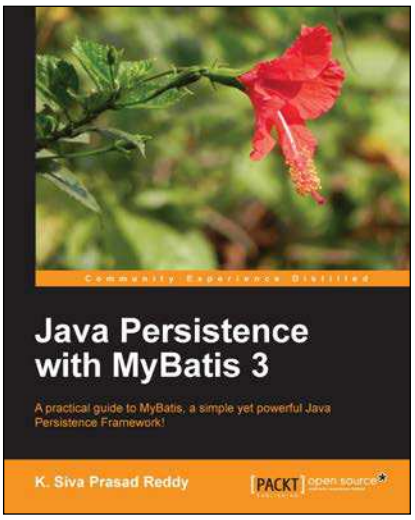
[PACKT] open source
community experience distilled
PUBLISHING

Java Persistence with MyBatis 3

K. Siva Prasad Reddy



Chapter No. 3
"SQL Mappers Using XML"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "SQL Mappers Using XML"

A synopsis of the book's content

Information on where to buy this book

About the Author

K. Siva Prasad Reddy is a Senior Software Engineer living in Hyderabad, India and has more than six years' experience in developing enterprise applications with Java and JavaEE technologies. Siva is a Sun Certified Java Programmer and has a lot of experience in server-side technologies such as Java, JavaEE, Spring, Hibernate, MyBatis, JSF (PrimeFaces), and WebServices (SOAP/REST).

Siva normally shares the knowledge he has acquired on his blog `www.sivalabs.in`. If you want to find out more information about his work, you can follow him on Twitter (`@sivalabs`) and GitHub (`https://github.com/sivaprasadreddy`).

I would like to thank my wife Neha, as she supported me in every step of the process and without her, this wouldn't have been possible. I thank my parents and my sister for their moral support in helping me complete this dream.

Java Persistence with MyBatis 3

For many software systems, saving and retrieving data from a database is a crucial part of the process. In Java land there are many tools and frameworks for implementing the data persistence layer and each of them follow a different approach. MyBatis, a simple yet powerful Java persistence framework, took the approach of eliminating the boilerplate code and leveraging the power of SQL and Java while still providing powerful features.

This MyBatis book will take you through the process of installing, configuring, and using MyBatis. Concepts in every chapter are explained through simple and practical examples with step-by-step instructions.

By the end of the book, you will not only gain theoretical knowledge but also gain hands-on practical understanding and experience on how to use MyBatis in your real projects.

This book can also be used as a reference or to relearn the concepts that have been discussed in each chapter. It has illustrative examples, wherever necessary, to make sure it is easy to follow.

What This Book Covers

Chapter 1, Getting Started with MyBatis, introduces MyBatis persistence framework and explains the advantages of using MyBatis instead of plain JDBC. We will also look at how to create a project, install MyBatis framework dependencies with and without the Maven build tool, configure, and use MyBatis.

Chapter 2, Bootstrapping MyBatis, covers how to bootstrap MyBatis using XML and Java API-based configuration. We will also learn various MyBatis configuration options such as type aliases, type handlers, global settings, and so on.

Chapter 3, SQL Mappers Using XML, goes in-depth into writing SQL mapped statements using the Mapper XML files. We will learn how to configure simple statements, statements with one-to-one, one-to-many relationships and mapping results using ResultMaps. We will also learn how to build dynamic queries, paginated results, and custom ResultSet handling.

Chapter 4, SQL Mappers Using Annotations, covers writing SQL mapped statements using annotations. We will learn how to configure simple statements, statements with one-to-one and one-to-many relationships. We will also look into building dynamic queries using SqlProvider annotations.

Chapter 5, Integration with Spring, covers how to integrate MyBatis with Spring framework. We will learn how to install Spring libraries, register MyBatis beans in Spring ApplicationContext, inject SqlSession and Mapper beans, and use Spring's annotation-based transaction handling mechanism with MyBatis.

3
SQL Mappers Using XML

Relational databases and SQL are time-tested and proven data storage mechanisms. Unlike other ORM frameworks such as Hibernate, MyBatis encourages the use of SQL instead of hiding it from developers, thereby utilizing the full power of SQL provided by the database server. At the same time, MyBatis eliminates the pain of writing boilerplate code and makes using SQL easy.

Embedding SQL queries directly inside the code is a bad practice and hard to maintain. MyBatis configures SQL statements using Mapper XML files or annotations. In this chapter, we will see how to configure mapped statements in Mapper XML files in detail; we will cover the following topics:

- Mapper XMLs and Mapper interfaces
- Mapped statements
    - Configuring `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements
- ResultMaps
    - Simple ResultMaps
    - One-to-one mapping using a nested `Select` query
    - One-to-one mapping using nested results mapping
    - One-to-many mapping using a nested `Select` query
    - One-to-many mapping using nested results mapping
- Dynamic SQL
    - The `If` condition
    - The `choose` (`when`, `otherwise`) condition
    - The `trim` (`where`, `set`) condition
    - The `foreach` loop
- MyBatis recipes

Mapper XMLs and Mapper interfaces

In the previous chapters, we have seen some basic examples of how to configure mapped statements in Mapper XML files and how to invoke them using the `SqlSession` object.

Let us now see how the `findStudentById` mapped statement can be configured in `StudentMapper.xml`, which is in the `com.mybatis3.mappers` package, using the following code:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

  <mapper namespace="com.mybatis3.mappers.StudentMapper">
  <select id="findStudentById" parameterType="int"
  resultType="Student">
  select stud_id as studId, name, email, dob from Students where
stud_id=#{studId}
  </select>
</mapper>
```

We can invoke the mapped statement as follows:

```
public Student findStudentById(Integer studId)
{
  SqlSession sqlSession = MyBatisUtil.getSqlSession();
  try
  {
    Student student =
sqlSession.selectOne("com.mybatis3.mappers.StudentMapper.
findStudentById", studId);
    return student;
  } finally {
    sqlSession.close();
  }
}
```

We can invoke mapped statements such as the previous one using string literals (`namespace` and `statement id`), but this exercise is error prone. You need to make sure to pass the valid input type parameter and assign the result to a valid return type variable by checking it in the Mapper XML file.

MyBatis provides a better way of invoking mapped statements by using Mapper interfaces. Once we have configured the mapped statements in the Mapper XML file, we can create a Mapper interface with a fully qualified name that is the same as the namespace and add the method signatures with matching statement IDs, input parameters, and return types.

For the preceding `StudentMapper.xml` file, we can create a Mapper interface `StudentMapper.java` as follows:

```
package com.mybatis3.mappers;
public interface StudentMapper
{
  Student findStudentById(Integer id);
}
```

In the `StudentMapper.xml` file, the namespace should be the same as the fully qualified name of the `StudentMapper` interface that is `com.mybatis3.mappers.StudentMapper`. Also, the statement `id`, `parameterType`, and `returnType` values in `StudentMapper.xml` should be the same as the method name, argument type, and return type in the `StudentMapper` interface respectively.

Using Mapper interfaces, you can invoke mapped statements in a type safe manner as follows:

```
public Student findStudentById(Integer studId)
{
  SqlSession sqlSession = MyBatisUtil.getSqlSession();
  try {
    StudentMapper studentMapper =
sqlSession.getMapper(StudentMapper.class);
    return studentMapper.findStudentById(studId);
  } finally {
    sqlSession.close();
  }
}
```

Even though Mapper interfaces are enabled to invoke mapped statements in a type safe manner, it is our responsibility to write Mapper interfaces with correct, matching method names, argument types, and return types. If the Mapper interface methods do not match the mapped statements in XML, you will get exceptions at runtime. Actually, specifying `parameterType` is optional; MyBatis can determine `parameterType` by using `Reflection API`. But from a readability perspective, it would be better to specify the `parameterType` attribute. If the `parameterType` attribute has not been mentioned, the developer will have to switch between Mapper XML and Java code to know what type of input parameter is being passed to that statement.

Mapped statements

MyBatis provides various elements to configure different types of statements, such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. Let us see how to configure mapped statements in detail.

The INSERT statement

An `INSERT` query can be configured in a Mapper XML file using the `<insert>` element as follows:

```
<insert id="insertStudent" parameterType="Student">
  INSERT INTO STUDENTS(STUD_ID,NAME,EMAIL, PHONE)
  VALUES(#{studId},#{name},#{email},#{phone})
</insert>
```

Here, we are giving an ID `insertStudent` that can be uniquely identified along with the namespace `com.mybatis3.mappers.StudentMapper.insertStudent`. The `parameterType` attribute value should be a fully qualified class name or type alias name.

We can invoke this statement as follows:

```
int count =
sqlSession.insert("com.mybatis3.mappers.StudentMapper.insertStuden
t", student);
```

The `sqlSession.insert()` method returns the number of rows affected by the `INSERT` statement.

Instead of invoking the mapped statement using `namespace` and the statement `id`, you can create a Mapper interface and invoke the method in a type safe manner as follows:

```
package com.mybatis3.mappers;
public interface StudentMapper
{
  int insertStudent(Student student);
}
```

You can invoke the `insertStudent` mapped statement as follows:

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
int count = mapper.insertStudent(student);
```

Autogenerated keys

In the preceding `INSERT` statement, we are inserting the value for the `STUD_ID` column that is an `auto_generated` primary key column. We can use the `useGeneratedKeys` and `keyProperty` attributes to let the database generate the `auto_increment` column value and set that generated value into one of the input object properties as follows:

```
<insert id="insertStudent" parameterType="Student"
useGeneratedKeys="true" keyProperty="studId">
  INSERT INTO STUDENTS(NAME, EMAIL, PHONE)
  VALUES(#{name},#{email},#{phone})
</insert>
```

Here the `STUD_ID` column value will be autogenerated by MySQL database, and the generated value will be set to the `studId` property of the `student` object.

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
mapper.insertStudent(student);
```

Now you can obtain the `STUD_ID` value of the inserted `STUDENT` record as follows:

```
int studentId = student.getStudId();
```

Some databases such as Oracle don't support `AUTO_INCREMENT` columns and use `SEQUENCE` to generate the primary key values.

Assume we have a `SEQUENCE` called `STUD_ID_SEQ` to generate the `STUD_ID` primary key values. Use the following code to generate the primary key:

```
<insert id="insertStudent" parameterType="Student">
  <selectKey keyProperty="studId" resultType="int" order="BEFORE">
    SELECT ELEARNING.STUD_ID_SEQ.NEXTVAL FROM DUAL
```

```
    </selectKey>
    INSERT INTO STUDENTS(STUD_ID,NAME,EMAIL, PHONE)
    VALUES(#{studId},#{name},#{email},#{phone})
  </insert>
```

Here we used the `<selectKey>` subelement to generate the primary key value and stored it in the `studId` property of the `Student` object. The attribute `order="BEFORE"` indicates that MyBatis will get the primary key value, that is, the next value from the sequence and store it in the `studId` property before executing the `INSERT` query.

We can also set the primary key value using a trigger where we will obtain the next value from the sequence and set it as the primary key column value before executing the `INSERT` query.

If you are using this approach, the `INSERT` mapped statement will be as follows:

```
<insert id="insertStudent" parameterType="Student">
  INSERT INTO STUDENTS(NAME,EMAIL, PHONE)
  VALUES(#{name},#{email},#{phone})
  <selectKey keyProperty="studId" resultType="int" order="AFTER">
    SELECT ELEARNING.STUD_ID_SEQ.CURRVAL FROM DUAL
  </selectKey>
</insert>
```

The UPDATE statement

An `UPDATE` statement can be configured in the Mapper XML file using the `<update>` element as follows:

```
<update id="updateStudent" parameterType="Student">
  UPDATE STUDENTS SET NAME=#{name}, EMAIL=#{email}, PHONE=#{phone}
  WHERE STUD_ID=#{studId}
</update>
```

We can invoke this statement as follows:

```
int noOfRowsUpdated =
sqlSession.update("com.mybatis3.mappers.StudentMapper.updateStudent",
student);
```

The `sqlSession.update()` method returns the number of rows affected by this `UPDATE` statement.

Instead of invoking the mapped statement using `namespace` and the statement `id`, you can create a Mapper interface and invoke the method in a type safe way as follows:

```
package com.mybatis3.mappers;
public interface StudentMapper
{
  int updateStudent(Student student);
}
```

You can invoke the `updateStudent` statement using the Mapper interface as follows:

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
int noOfRowsUpdated = mapper.updateStudent(student);
```

The DELETE statement

A `DELETE` statement can be configured in the Mapper XML file using the `<delete>` element as follows:

```
<delete id="deleteStudent" parameterType="int">
  DELETE FROM STUDENTS WHERE STUD_ID=#{studId}
</delete>
```

We can invoke this statement as follows:

```
int studId =1;
int noOfRowsDeleted =
sqlSession.delete("com.mybatis3.mappers.StudentMapper.deleteStuden
t", studId);
```

The `sqlSession.delete()` method returns the number of rows affected by this `delete` statement.

Instead of invoking the mapped statement using `namespace` and the statement `id`, you can create a Mapper interface and invoke the method in a type safe way as follows:

```
package com.mybatis3.mappers;
public interface StudentMapper
{
  int deleteStudent(int studId);
}
```

You can invoke the `deleteStudent` statement using the Mapper interface as follows:

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
int noOfRowsDeleted = mapper.deleteStudent(studId);
```

The SELECT statement

The true power of MyBatis will be known only by finding out how flexible MyBatis is for mapping `SELECT` query results to JavaBeans.

Let us see how a simple `select` query can be configured, using the following code:

```
<select id="findStudentById" parameterType="int"
resultType="Student">
  SELECT STUD_ID, NAME, EMAIL, PHONE
  FROM STUDENTS
  WHERE STUD_ID=#{studId}
</select>
```

We can invoke this statement as follows:

```
int studId =1;
Student student = sqlSession.selectOne("com.mybatis3.mappers.
StudentMapper.findStudentById", studId);
```

The `sqlSession.selectOne()` method returns the object of the type configured for the `resultType` attribute. If the query returns multiple rows for the `sqlSession.selectOne()` method, `TooManyResultsException` will be thrown.

Instead of invoking the mapped statement using `namespace` and the statement `id`, you can create a Mapper interface and invoke the method in a type safe manner as follows:

```
package com.mybatis3.mappers;
public interface StudentMapper
{
  Student findStudentById(Integer studId);
}
```

You can invoke the `findStudentById` statement using the Mapper interface as follows:

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
Student student = mapper.findStudentById(studId);
```

If you check the property values of the `Student` object, you will observe that the `studId` property value is not populated with the `stud_id` column value. This is because MyBatis automatically populates the JavaBeans properties with the column values that have a matching column name. That is why, the properties `name`, `email`, and `phone` get populated but the `studId` property does not get populated.

To resolve this, we can give alias names for the columns to match with the Java Beans property names as follows:

```
<select id="findStudentById" parameterType="int"
resultType="Student">
  SELECT STUD_ID AS studId, NAME,EMAIL, PHONE
  FROM STUDENTS
  WHERE STUD_ID=#{studId}
</select>
```

Now the `Student` bean will get populated with all the `stud_id`, `name`, `email`, and `phone` columns properly.

Now let us see how to execute a `SELECT` query that returns multiple rows as shown in the following code:

```
<select id="findAllStudents" resultType="Student">
  SELECT STUD_ID AS studId, NAME,EMAIL, PHONE
  FROM STUDENTS
</select>
```

```
List<Student> students =
sqlSession.selectList("com.mybatis3.mappers.StudentMapper.findAllS
tudents");
```

The Mapper interface `StudentMapper` can also be used as follows:

```
package com.mybatis3.mappers;
public interface StudentMapper
{
  List<Student> findAllStudents();
}
```

Using the previous code, you can invoke the `findAllStudents` statement with the Mapper interface as follows:

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
List<Student> students = mapper.findAllStudents();
```

If you observe the preceding `SELECT` query mappings, we are giving the alias name for `stud_id` in all the mapped statements.

Instead of repeating the alias names everywhere, we can use `ResultMaps`, which we are going to discuss in a moment.

Instead of `java.util.List`, you can also use other types of collections, such as `Set`, `Map`, and `SortedSet`. Based on the type of the collection, MyBatis will use an appropriate collection implementation as follows:

- For the `List`, `Collection`, or `Iterable` types, `java.util.ArrayList` will be returned
- For the `Map` type, `java.util.HashMap` will be returned
- For the `Set` type, `java.util.HashSet` will be returned
- For the `SortedSet` type, `java.util.TreeSet` will be returned

ResultMaps

ResultMaps are used to map the SQL `SELECT` statement's results to JavaBeans properties. We can define ResultMaps and reference this `resultMap` query from several `SELECT` statements. The MyBatis ResultMaps feature is so powerful that you can use it for mapping simple `SELECT` statements to complex `SELECT` statements with one-to-one and one-to-many associations.

Simple ResultMaps

A simple `resultMap` query that maps query results to the `Student` JavaBeans is as follows:

```
<resultMap id="StudentResult" type="com.mybatis3.domain.Student">
  <id property="studId" column="stud_id"/>
  <result property="name" column="name"/>
  <result property="email" column="email"/>
  <result property="phone" column="phone"/>
</resultMap>

<select id="findAllStudents" resultMap="StudentResult" >
  SELECT * FROM STUDENTS
</select>

<select id="findStudentById" parameterType="int"
resultMap="StudentResult">
  SELECT * FROM STUDENTS WHERE STUD_ID=#{studId}
</select>
```

The `id` attribute of `resultMap` `StudentResult` should be unique within the `namespace`, and the type should be a fully qualified name or alias name of the return type.

The `<result>` sub-elements are used to map a `resultset` column to a JavaBeans property.

The `<id>` element is similar to `<result>` but is used to map the identifier property that is used for comparing objects.

In the `<select>` statement, we have used the `resultMap` attribute instead of `resultType` to refer the `StudentResult` mapping. When a `resultMap` attribute is configured for a `<select>` statement, MyBatis uses the column for property mappings in order to populate the JavaBeans properties.

We can use either `resultType` or `resultMap` for a `SELECT` mapped statement, but not both.

Let us see another example of a `<select>` mapped statement showing how to populate query results into `HashMap` as follows:

```
<select id="findStudentById" parameterType="int" resultType="map">
  SELECT * FROM STUDENTS WHERE STUD_ID=#{studId}
</select>
```

In the preceding `<select>` statement, we configured `resultType` to be `map`, that is, the alias name for `java.util.HashMap`. In this case, the column names will be the key and the column value will be the value.

```
HashMap<String,Object> studentMap = sqlSession.selectOne("com.
mybatis3.mappers.StudentMapper.findStudentById", studId);
System.out.println("stud_id :"+studentMap.get("stud_id"));
System.out.println("name :"+studentMap.get("name"));
System.out.println("email :"+studentMap.get("email"));
System.out.println("phone :"+studentMap.get("phone"));
```

Let us see another example using `resultType="map"` that returns multiple rows.

```
<select id="findAllStudents" resultType="map">
  SELECT STUD_ID, NAME, EMAIL, PHONE FROM STUDENTS
</select>
```

As `resultType="map"` and the statement return multiple rows, the final return type would be `List<HashMap<String,Object>>` as shown in the following code:

```
List<HashMap<String,Object>> studentMapList =
sqlSession.selectList("com.mybatis3.mappers.StudentMapper.findAllS
tudents");

for(HashMap<String,Object> studentMap : studentMapList)
```

```
{
  System.out.println("studId :"+studentMap.get("stud_id"));
  System.out.println("name :"+studentMap.get("name"));
  System.out.println("email :"+studentMap.get("email"));
  System.out.println("phone :"+studentMap.get("phone"));
}
```

Extending ResultMaps

We can extend one `<resultMap>` query from another `<resultMap>` query, thereby inheriting the column to do property mappings from the one that is being extended.

```
<resultMap type="Student" id="StudentResult">
  <id property="studId" column="stud_id"/>
  <result property="name" column="name"/>
  <result property="email" column="email"/>
  <result property="phone" column="phone"/>
</resultMap>

<resultMap type="Student" id="StudentWithAddressResult"
extends="StudentResult">
  <result property="address.addrId" column="addr_id"/>
  <result property="address.street" column="street"/>
  <result property="address.city" column="city"/>
  <result property="address.state" column="state"/>
  <result property="address.zip" column="zip"/>
  <result property="address.country" column="country"/>
</resultMap>
```

The `resultMap` query with the ID `StudentWithAddressResult` extends the `resultMap` with the ID `StudentResult`.

Now you can use `StudentResult resultMap` if you want to map only the `Student` data as shown in the following code:

```
<select id="findStudentById" parameterType="int"
resultMap="StudentResult">
  SELECT * FROM STUDENTS WHERE STUD_ID=#{studId}
</select>
```

If you want to map the query results with `Student` along with the `Address` data, you can use `resultMap` with the ID `StudentWithAddressResult` as follows:

```
<select id="selectStudentWithAddress" parameterType="int"
resultMap="StudentWithAddressResult">
  SELECT STUD_ID, NAME, EMAIL, PHONE, A.ADDR_ID, STREET, CITY,
```

```
  STATE, ZIP, COUNTRY
  FROM STUDENTS S LEFT OUTER JOIN ADDRESSES A ON
  S.ADDR_ID=A.ADDR_ID
  WHERE STUD_ID=#{studId}
</select>
```

One-to-one mapping

In our sample domain model, each student has an associated address. The `STUDENTS` table has an `ADDR_ID` column that is a foreign key to the `ADDRESSES` table.

The `STUDENTS` table's sample data is as follows:

STUD_ID	NAME	E-MAIL	PHONE	ADDR_ID
1	John	john@gmail.com	123-456-7890	1
2	Paul	paul@gmail.com	111-222-3333	2

The `ADDRESSES` table's sample data is as follows:

ADDR_ID	STREET	CITY	STATE	ZIP	COUNTRY
1	Naperville	CHICAGO	IL	60515	USA
2	Elgin	CHICAGO	IL	60515	USA

Let us see how to fetch `Student` details along with `Address` details.

The `Student` and `Address` JavaBeans are created as follows:

```
public class Address
{
  private Integer addrId;
  private String street;
  private String city;
  private String state;
  private String zip;
  private String country;
  // setters & getters
}
public class Student
{
  private Integer studId;
  private String name;
```

```
  private String email;
  private PhoneNumber phone;
  private Address address;
  //setters & getters
}

<resultMap type="Student" id="StudentWithAddressResult">
  <id property="studId" column="stud_id"/>
  <result property="name" column="name"/>
  <result property="email" column="email"/>
  <result property="phone" column="phone"/>
  <result property="address.addrId" column="addr_id"/>
  <result property="address.street" column="street"/>
  <result property="address.city" column="city"/>
  <result property="address.state" column="state"/>
  <result property="address.zip" column="zip"/>
  <result property="address.country" column="country"/>
</resultMap>

<select id="selectStudentWithAddress" parameterType="int"
  resultMap="StudentWithAddressResult">
  SELECT STUD_ID, NAME, EMAIL, A.ADDR_ID, STREET, CITY, STATE,
  ZIP, COUNTRY
  FROM STUDENTS S LEFT OUTER JOIN ADDRESSES A ON
  S.ADDR_ID=A.ADDR_ID
  WHERE STUD_ID=#{studId}
</select>
```

We can set the properties of a nested object using the dot notation. In the preceding `resultMap`, Student's address property values are set by `address` column values using dot notation. Likewise, we can refer the properties of nested objects to any depth. We can access the nested object properties as follows:

```
public interface StudentMapper
{
  Student selectStudentWithAddress(int studId);
}

int studId = 1;
StudentMapper studentMapper =
sqlSession.getMapper(StudentMapper.class);
Student student = studentMapper.selectStudentWithAddress(studId);
System.out.println("Student :"+student);
System.out.println("Address :"+student.getAddress());
```

The preceding example shows one way of mapping a one-to-one association. However with this approach, if the `address` results need to be mapped to the `Address` object values in other `Select` mapped statements, we'll need to repeat the mappings for each statement.

MyBatis provides better approaches for mapping one-to-one associations using the `Nested ResultMap` and `Nested Select` statements, which is what we are going to discuss next.

One-to-one mapping using nested ResultMap

We can get `Student` along with the `Address` details using a nested ResultMap as follows:

```
<resultMap type="Address" id="AddressResult">
  <id property="addrId" column="addr_id"/>
  <result property="street" column="street"/>
  <result property="city" column="city"/>
  <result property="state" column="state"/>
  <result property="zip" column="zip"/>
  <result property="country" column="country"/>
</resultMap>

<resultMap type="Student" id="StudentWithAddressResult">
  <id property="studId" column="stud_id"/>
  <result property="name" column="name"/>
  <result property="email" column="email"/>
  <association property="address" resultMap="AddressResult"/>
</resultMap>

<select id="findStudentWithAddress" parameterType="int"
resultMap="StudentWithAddressResult">
  SELECT STUD_ID, NAME, EMAIL, A.ADDR_ID, STREET, CITY, STATE,
ZIP, COUNTRY
  FROM STUDENTS S LEFT OUTER JOIN ADDRESSES A ON
S.ADDR_ID=A.ADDR_ID
  WHERE STUD_ID=#{studId}
</select>
```

The `<association>` element can be used to load the `has-one` type of associations. In the preceding example, we used the `<association>` element, referencing another `<resultMap>` that is declared in the same XML file.

We can also use `<association>` with an inline `resultMap` query as follows:

```
<resultMap type="Student" id="StudentWithAddressResult">
  <id property="studId" column="stud_id"/>
  <result property="name" column="name"/>
  <result property="email" column="email"/>
  <association property="address" javaType="Address">
    <id property="addrId" column="addr_id"/>
    <result property="street" column="street"/>
    <result property="city" column="city"/>
    <result property="state" column="state"/>
    <result property="zip" column="zip"/>
    <result property="country" column="country"/>
  </association>
</resultMap>
```

Using the nested ResultMap approach, the association data will be loaded using a single query (along with `joins` if required).

One-to-one mapping using nested Select

We can get `Student` along with the `Address` details using a nested `Select` query as follows:

```
<resultMap type="Address" id="AddressResult">
  <id property="addrId" column="addr_id"/>
  <result property="street" column="street"/>
  <result property="city" column="city"/>
  <result property="state" column="state"/>
  <result property="zip" column="zip"/>
  <result property="country" column="country"/>
</resultMap>

<select id="findAddressById" parameterType="int"
resultMap="AddressResult">
  SELECT * FROM ADDRESSES WHERE ADDR_ID=#{id}
</select>

<resultMap type="Student" id="StudentWithAddressResult">
  <id property="studId" column="stud_id"/>
  <result property="name" column="name"/>
  <result property="email" column="email"/>
  <association property="address" column="addr_id"
select="findAddressById"/>
</resultMap>
```

```
<select id="findStudentWithAddress" parameterType="int"
resultMap="StudentWithAddressResult">
  SELECT * FROM STUDENTS WHERE STUD_ID=#{Id}
</select>
```

In this approach, the `<association>` element's `select` attribute is set to the statement `id` `findAddressById`. Here, two separate SQL statements will be executed against the database, the first one called `findStudentById` to load student details and the second one called `findAddressById` to load its address details.

The `addr_id` column value will be passed as input to the `selectAddressById` statement.

We can invoke the `findStudentWithAddress` mapped statement as follows:

```
StudentMapper mapper = sqlSession.getMapper(StudentMapper.class);
Student student = mapper.selectStudentWithAddress(studId);
System.out.println(student);
System.out.println(student.getAddress());
```

One-to-many mapping

In our sample domain model, a tutor can teach one or more courses. This means that there is a one-to-many relationship between the tutor and course.

We can map one-to-many types of results to a collection of objects using the `<collection>` element.

The `TUTORS` table's sample data is as follows:

TUTOR_ID	NAME	EMAIL	PHONE	ADDR_ID
1	John	john@gmail.com	123-456-7890	1
2	Ying	ying@gmail.com	111-222-3333	2

The `COURSES` table's sample data is as follows:

COURSE_ID	NAME	DESCRIPTION	START_DATE	END_DATE	TUTOR_ID
1	JavaSE	Java SE	2013-01-10	2013-02-10	1
2	JavaEE	JavaEE6	2013-01-10	2013-03-10	2
3	MyBatis	MyBatis	2013-01-10	2013-02-20	2

In the preceding table data, the tutor John teaches one course whereas the tutor Ying teaches two courses.

The JavaBeans for `Course` and `Tutor` are as follows:

```
public class Course
{
  private Integer courseId;
  private String name;
  private String description;
  private Date startDate;
  private Date endDate;
  private Integer tutorId;

  //setters & getters
}

public class Tutor
{
  private Integer tutorId;
  private String name;
  private String email;
  private Address address;
  private List<Course> courses;
  /setters & getters
}
```

Now let us see how we can get the tutor's details along with the list of courses he/she teaches.

The `<collection>` element can be used to map multiple course rows to a list of course objects. Similar to one-to-one mapping, we can map one-to-many relationships using a nested ResultMap and nested `Select` approaches.

One-to-many mapping with nested ResultMap

We can get the tutor along with the courses' details using a nested ResultMap as follows:

```
<resultMap type="Course" id="CourseResult">
  <id column="course_id" property="courseId"/>
  <result column="name" property="name"/>
  <result column="description" property="description"/>
  <result column="start_date" property="startDate"/>
  <result column="end_date" property="endDate"/>
```

```
</resultMap>

<resultMap type="Tutor" id="TutorResult">
  <id column="tutor_id" property="tutorId"/>
  <result column="tutor_name" property="name"/>
  <result column="email" property="email"/>
  <collection property="courses" resultMap="CourseResult"/>
</resultMap>

<select id="findTutorById" parameterType="int"
resultMap="TutorResult">
  SELECT T.TUTOR_ID, T.NAME AS TUTOR_NAME, EMAIL, C.COURSE_ID,
  C.NAME, DESCRIPTION, START_DATE, END_DATE
  FROM TUTORS T LEFT OUTER JOIN ADDRESSES A ON T.ADDR_ID=A.ADDR_ID
  LEFT OUTER JOIN COURSES C ON T.TUTOR_ID=C.TUTOR_ID
  WHERE T.TUTOR_ID=#{tutorId}
</select>
```

Here we are fetching the tutor along with the courses' details using a single `Select` query with `JOINS`. The `<collection>` element's `resultMap` is set to the `resultMap` ID `CourseResult` that contains the mapping for the `Course` object's properties.

One-to-many mapping with nested select

We can get the tutor along with the courses' details using a nested `select` query as follows:

```
<resultMap type="Course" id="CourseResult">
  <id column="course_id" property="courseId"/>
  <result column="name" property="name"/>
  <result column="description" property="description"/>
  <result column="start_date" property="startDate"/>
  <result column="end_date" property="endDate"/>
</resultMap>

<resultMap type="Tutor" id="TutorResult">
  <id column="tutor_id" property="tutorId"/>
  <result column="tutor_name" property="name"/>
  <result column="email" property="email"/>
  <association property="address" resultMap="AddressResult"/>
  <collection property="courses" column="tutor_id"
select="findCoursesByTutor"/>
  </resultMap>

<select id="findTutorById" parameterType="int"
```

```
resultMap="TutorResult">
  SELECT T.TUTOR_ID, T.NAME AS TUTOR_NAME, EMAIL
  FROM TUTORS T WHERE T.TUTOR_ID=#{tutorId}
  </select>

<select id="findCoursesByTutor" parameterType="int"
resultMap="CourseResult">
  SELECT * FROM COURSES WHERE TUTOR_ID=#{tutorId}
</select>
```

In this approach, the `<association>` element's `select` attribute is set to the statement ID `findCoursesByTutor` that triggers a separate SQL query to load the courses' details. The `tutor_id` column value will be passed as input to the `findCoursesByTutor` statement.

```
public interface TutorMapper
{
  Tutor findTutorById(int tutorId);
}
TutorMapper mapper = sqlSession.getMapper(TutorMapper.class);
Tutor tutor = mapper.findTutorById(tutorId);
System.out.println(tutor);
List<Course> courses = tutor.getCourses();
for (Course course : courses)
{
  System.out.println(course);
}
```

A nested select approach may result in N+1 select problems. First, the main query will be executed (1), and for every row returned by the first query, another select query will be executed (*N* queries for *N* rows). For large datasets, this could result in poor performance.

Dynamic SQL

Sometimes, static SQL queries may not be sufficient for application requirements. We may have to build queries dynamically, based on some criteria.

For example, in web applications there could be search screens that provide one or more input options and perform searches based on the chosen criteria. While implementing this kind of search functionality, we may need to build a dynamic query based on the selected options. If the user provides any value for input criteria, we'll need to add that field in the `WHERE` clause of the query.

MyBatis provides first-class support for building dynamic SQL queries using elements such as `<if>`, `<choose>`, `<where>`, `<foreach>`, and `<trim>`.

The If condition

The `<if>` element can be used to conditionally embed SQL snippets. If the test condition is evaluated to `true`, then only the SQL snippet will be appended to the query.

Assume we have a Search Courses Screen that has a `Tutor` dropdown, the `CourseName` text field, and the `StartDate` and `End Date` input fields as the search criteria.

Assume that `Tutor` is a mandatory field and that the rest of the fields are optional.

When the user clicks on the search button, we need to display a list of courses that meet the following criteria:

- Courses by the selected `Tutor`
- Courses whose name contain the entered course name; if nothing has been provided, fetch all the courses
- Courses whose start date and end date are in between the provided `StartDate` and `EndDate` input fields

We can create the mapped statement for searching the courses as follows:

```
<resultMap type="Course" id="CourseResult">
  <id column="course_id" property="courseId"/>
  <result column="name" property="name"/>
  <result column="description" property="description"/>
  <result column="start_date" property="startDate"/>
  <result column="end_date" property="endDate"/>
</resultMap>

<select id="searchCourses" parameterType="hashmap"
resultMap="CourseResult">
<![CDATA[
  SELECT * FROM COURSES
  WHERE TUTOR_ID= #{tutorId}
  <if test="courseName != null">
    AND NAME LIKE #{courseName}
  </if>
  <if test="startDate != null">
    AND START_DATE >= #{startDate}
  </if>
```

```
    <if test="endDate != null">
      AND END_DATE <= #{endDate}
    </if>
  ]]>
  </select>

  public interface CourseMapper
  {
    List<Course> searchCourses(Map<String, Object> map);
  }

  public void searchCourses()
  {
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("tutorId", 1);
    map.put("courseName", "%java%");
    map.put("startDate", new Date());
    CourseMapper mapper = sqlSession.getMapper(CourseMapper.class);
    List<Course> courses = mapper.searchCourses(map);
    for (Course course : courses) {
      System.out.println(course);

  }
```

This will generate the query `SELECT * FROM COURSES WHERE TUTOR_ID= ? AND NAME like ? AND START_DATE >= ?`. This will come in handy while preparing a dynamic SQL query based on the given criteria.

MyBatis uses **OGNL** (**Object Graph Navigation Language**) expressions for building dynamic queries.

The choose, when, and otherwise conditions

Sometimes, search functionality could be based on the search type. First, the user needs to choose whether he wants to search by `Tutor` or `Course Name` or `Start Dates` and `End Dates`, and then based on the selected search type, the input field will appear. In such scenarios, we should apply only one of the conditions.

MyBatis provides the `<choose>` element to support this kind of dynamic SQL preparation.

Now let us write a SQL mapped statement to get the courses by applying the search criteria. If no search criteria is selected, the courses starting from today onwards should be fetched as follows:

```
<select id="searchCourses" parameterType="hashmap"
resultMap="CourseResult">
  SELECT * FROM COURSES
  <choose>
    <when test="searchBy == 'Tutor'">
      WHERE TUTOR_ID= #{tutorId}
    </when>
    <when test="searchBy == 'CourseName'">
      WHERE name like #{courseName}
    </when>
    <otherwise>
      WHERE TUTOR start_date  &gt;= now()
    </otherwise>
  </choose>
</select>
```

MyBatis evaluates the `<choose>` test conditions and uses the clause with the first condition that evaluates to `TRUE`. If none of the conditions are true, the `<otherwise>` clause will be used.

The where condition

At times, all the search criteria might be optional. In cases where at least one of the search conditions needs to be applied, then only the `WHERE` clause should be appended. Also, we need to append `AND` or `OR` to the conditions only if there are multiple conditions. MyBatis provides the `<where>` element to support building these kinds of dynamic SQL statements.

In our example Search Courses screen, we assume that all the search criteria is optional. So, the `WHERE` clause should be there only if any of the search criteria has been provided.

```
<select id="searchCourses" parameterType="hashmap"
resultMap="CourseResult">
  SELECT * FROM COURSES
  <where>
    <if test=" tutorId != null ">
      TUTOR_ID= #{tutorId}
    </if>
    <if test="courseName != null">
      AND name like #{courseName}
```

```
      </if>
      <if test="startDate != null">
        AND start_date  &gt;= #{startDate}
      </if>
      <if test="endDate != null">
        AND end_date  &lt;= #{endDate}
      </if>
    </where>
  </select>
```

The `<where>` element inserts `WHERE` only if any content is returned by the inner conditional tags. Also, it removes the `AND` or `OR` prefixes if the `WHERE` clause begins with `AND` or `OR`.

In the preceding example, if none of the `<if>` conditions are `True`, `<where>` won't insert the `WHERE` clause. If at least one of the `<if>` conditions is `True`, `<where>` will insert the `WHERE` clause followed by the content returned by the `<if>` tags.

If the `tutor_id` parameter is `null` and the `courseName` parameter is not `null`, `<where>` will take care of stripping out the `AND` prefix and adding `NAME like #{courseName}`.

The trim condition

The `<trim>` element works similar to `<where>` but provides additional flexibility on what prefix/suffix needs to be prefixed/suffixed and what prefix/suffix needs to be stripped off.

```
<select id="searchCourses" parameterType="hashmap"
resultMap="CourseResult">
  SELECT * FROM COURSES
  <trim prefix="WHERE" prefixOverrides="AND | OR">
    <if test=" tutorId != null ">
      TUTOR_ID= #{tutorId}
    </if>
    <if test="courseName != null">
      AND name like #{courseName}
    </if>
  </trim>
</select>
```

Here `<trim>` will insert `WHERE` if any of the `<if>` conditions are `true` and remove the `AND` or `OR` prefixes just after `WHERE`.

The foreach loop

Another powerful dynamic SQL builder tag is `<foreach>`. It is a very common requirement for iterating through an array or list and for building `AND/OR` conditions or an `IN` clause.

Suppose we want to find out all the courses taught by the tutors whose `tutor_id` IDs are `1`, `3`, and `6`. We can pass a list of `tutor_id` IDs to the mapped statement and build a dynamic query by iterating through the list using `<foreach>`.

```
<select id="searchCoursesByTutors" parameterType="map"
resultMap="CourseResult">
  SELECT * FROM COURSES
  <if test="tutorIds != null">
    <where>
      <foreach item="tutorId" collection="tutorIds">
        OR tutor_id=#{tutorId}
      </foreach>
    </where>
  </if>
</select>

public interface CourseMapper
{
  List<Course> searchCoursesByTutors(Map<String, Object> map);
}

public void searchCoursesByTutors()
{
  Map<String, Object> map = new HashMap<String, Object>();
  List<Integer> tutorIds = new ArrayList<Integer>();
  tutorIds.add(1);
  tutorIds.add(3);
  tutorIds.add(6);
  map.put("tutorIds", tutorIds);
  CourseMapper mapper =
  sqlSession.getMapper(CourseMapper.class);
  List<Course> courses = mapper.searchCoursesByTutors(map);
  for (Course course : courses)
  {
    System.out.println(course);
  }
}
```

Let us see how to use `<foreach>` to generate the `IN` clause:

```
<select id="searchCoursesByTutors" parameterType="map"
resultMap="CourseResult">
  SELECT * FROM COURSES
  <if test="tutorIds != null">
    <where>
      tutor_id IN
      <foreach item="tutorId" collection="tutorIds"
        open="(" separator="," close=")">
        #{tutorId}
      </foreach>
    </where>
  </if>
</select>
```

The set condition

The `<set>` element is similar to the `<where>` element and will insert `SET` if any content is returned by the inner conditions.

```
<update id="updateStudent" parameterType="Student">
  update students
  <set>
    <if test="name != null">name=#{name},</if>
    <if test="email != null">email=#{email},</if>
    <if test="phone != null">phone=#{phone},</if>
  </set>
  where stud_id=#{id}
</update>
```

Here, `<set>` inserts the `SET` keyword if any of the `<if>` conditions return text and also strips out the tailing commas at the end.

In the preceding example, if `phone != null`, `<set>` will take care of removing the comma after `phone=#{phone}`.

MyBatis recipes

In addition to simplifying the database programming, MyBatis provides various features that are very useful for implementing some common tasks, such as loading the table rows page by page, storing and retrieving `CLOB/BLOB` type data, and handling enumerated type values, among others. Let us have a look at a few of these features.

Handling enumeration types

MyBatis supports persisting `enum` type properties out of the box. Assume that the `STUDENTS` table has a column `gender` of the type `varchar` to store either `MALE` or `FEMALE` as the value. And, the `Student` object has a `gender` property that is of the type `enum` as shown in the following code:

```
public enum Gender
{
  FEMALE,
  MALE
}
```

By default, MyBatis uses `EnumTypeHandler` to handle `enum` type Java properties and stores the name of the `enum` value. You don't need any extra configuration to do this. You can use `enum` type properties just like primitive type properties as shown in the following code:

```
public class Student
{
  private Integer id;
  private String name;
  private String email;
  private PhoneNumber phone;
  private Address address;
  private Gender gender;
  //setters and getters
}

<insert id="insertStudent" parameterType="Student"
useGeneratedKeys="true" keyProperty="id">
  insert into students(name,email,addr_id, phone,gender)
  values(#{name},#{email},#{address.addrId},#{phone},#{gender})
</insert>
```

When you execute the `insertStudent` statement, MyBatis takes the name of the `Gender` enum (`FEMALE`/`MALE`) and stores it in the `GENDER` column.

If you want to store the ordinal position of the enum instead of the enum name, you will need to explicitly configure it.

So if you want to store `0` for `FEMALE` and `1` for `MALE` in the `gender` column, you'll need to register `EnumOrdinalTypeHandler` in the `mybatis-config.xml` file.

```
<typeHandler
handler="org.apache.ibatis.type.EnumOrdinalTypeHandler"
javaType="com.mybatis3.domain.Gender"/>
```

Be careful to use ordinal values to store in the DB. Ordinal values are assigned to `enum` values based on their order of declaration. If you change the declaration order in `Gender enum`, the data in the database and ordinal values will be mismatched.

Handling the CLOB/BLOB types

MyBatis provides built-in support for mapping `CLOB/BLOB` type columns.

Assume we have the following table to store the `Students` and `Tutors` photographs and their biodata:

```
CREATE TABLE USER_PICS
(
  ID INT(11) NOT NULL AUTO_INCREMENT,
  NAME VARCHAR(50) DEFAULT NULL,
  PIC BLOB,
  BIO LONGTEXT,
  PRIMARY KEY (ID)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=LATIN1;
```

Here, the photograph can be an image of type PNG, JPG, and so on, and the biodata can be a lengthy history about the student/tutor.

By default, MyBatis maps `CLOB` type columns to the `java.lang.String` type and `BLOB` type columns to the `byte[]` type.

```
public class UserPic
{
  private int id;
  private String name;
  private byte[] pic;
  private String bio;
  //setters & getters
}
```

Create the `UserPicMapper.xml` file and configure the mapped statements as follows:

```
<insert id="insertUserPic" parameterType="UserPic">
  INSERT INTO USER_PICS(NAME, PIC,BIO)
  VALUES(#{name},#{pic},#{bio})
</insert>

<select id="getUserPic" parameterType="int" resultType="UserPic">
  SELECT * FROM USER_PICS WHERE ID=#{id}
</select>
```

The following method `insertUserPic()` shows how to insert data into `CLOB/BLOB` type columns:

```
public void insertUserPic()
{
  byte[] pic = null;
  try {
    File file = new File("C:\\Images\\UserImg.jpg");
    InputStream is = new FileInputStream(file);
    pic = new byte[is.available()];
    is.read(pic);
    is.close();
  } catch (FileNotFoundException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  }
  String name = "UserName";
  String bio = "put some lenghty bio here";
  UserPic userPic = new UserPic(0, name, pic , bio);

  SqlSession sqlSession = MyBatisUtil.openSession();
  try {
    UserPicMapper mapper =
    sqlSession.getMapper(UserPicMapper.class);
    mapper.insertUserPic(userPic);
    sqlSession.commit();
  }
  finally {
    sqlSession.close();
  }

}
```

The following method `getUserPic()` shows how to read `CLOB` type data into `String` and `BLOB` type data into `byte[]` properties:

```
public void getUserPic()
{
  UserPic userPic = null;
  SqlSession sqlSession = MyBatisUtil.openSession();
  try {
    UserPicMapper mapper =
sqlSession.getMapper(UserPicMapper.class);
    userPic =  mapper.getUserPic(1);
  }
  finally {
    sqlSession.close();
  }
  byte[] pic = userPic.getPic();
  try {
    OutputStream os = new FileOutputStream(new
File("C:\\Images\\UserImage_FromDB.jpg"));
    os.write(pic);
    os.close();
  } catch (FileNotFoundException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Passing multiple input parameters

MyBatis's mapped statements have the `parameterType` attribute to specify the type of input parameter. If we want to pass multiple input parameters to a mapped statement, we can put all the input parameters in a `HashMap` and pass it to that mapped statement.

MyBatis provides another way of passing multiple input parameters to a mapped statement. Suppose we want to find students with the given `name` and `email`.

```
Public interface StudentMapper
{
  List<Student> findAllStudentsByNameEmail(String name, String
email);
}
```

MyBatis supports passing multiple input parameters to a mapped statement and referencing them using the `#{param}` syntax.

```
<select id="findAllStudentsByNameEmail" resultMap="StudentResult"
>
  select stud_id, name,email, phone from Students
  where name=#{param1} and email=#{param2}
 </select>
```

Here `#{param1}` refers to the first parameter `name` and `#{param2}` refers to the second parameter `email`.

```
StudentMapper studentMapper = sqlSession.getMapper(StudentMapper.
class);
studentMapper.findAllStudentsByNameEmail(name, email);
```

Multiple results as a map

If we have a mapped statement that returns multiple rows and we want the results in a `HashMap` with some property value as the key and the resulting object as the value, we can use `sqlSession.selectMap()` as follows:

```
<select id=" findAllStudents" resultMap="StudentResult">
  select * from Students
</select>

Map<Integer, Student> studentMap =
sqlSession.selectMap("com.mybatis3.mappers.StudentMapper.
findAllStudents", "studId");
```

Here `studentMap` will contain `studId` values as keys and `Student` objects as values.

Paginated ResultSets using RowBounds

Sometimes, we may need to work with huge volumes of data, such as with tables with millions of records. Loading all these records may not be possible due to memory constraints, or we may need only a fragment of data. Typically in web applications, pagination is used to display large volumes of data in a page-by-page style.

MyBatis can load table data page by page using `RowBounds`. The `RowBounds` object can be constructed using the `offset` and `limit` parameters. The parameter `offset` refers to the starting position and `limit` refers to the number of records.

Suppose if you want to load and display 25 student records per page, you can use the following query:

```
<select id="findAllStudents" resultMap="StudentResult">
  select * from Students
</select>
```

Then, you can load the first page (first 25 records) as follows:

```
int offset =0 , limit =25;
RowBounds rowBounds = new RowBounds(offset, limit);
List<Student> = studentMapper.getStudents(rowBounds);
```

To display the second page, use `offset=25` and `limit=25`; for the third page, use `offset=50` and `limit=25`.

Custom ResultSet processing using ResultSetHandler

MyBatis provides great support with plenty of options for mapping the query results to JavaBeans. But sometimes, we may come across scenarios where we need to process the SQL query results by ourselves for special purposes. MyBatis provides `ResultHandler` plugin that enables the processing of the ResultSet in whatever way we like.

Suppose that we want to get the student details in a `HashMap` where `stud_id` is used as a key and name is used as a value.

As of `mybatis-3.2.2`, MyBatis doesn't have support for getting the result as `HashMap`, with one property value as the key and another property value as the value, using the `resultMap` element. `sqlSession.selectMap()` returns a map with the given property value as the key and the result object as the value. We can't configure it to use one property as the key and another property as the value.

For `sqlSession.select()` methods, we can pass an implementation of `ResultHandler` that will be invoked for each record in the ResultSet.

```
public interface ResultHandler
{
  void handleResult(ResultContext context);
}
```

Now let us see how we can use `ResultHandler` to process the ResultSet and return customized results.

```
public Map<Integer, String> getStudentIdNameMap()
{
  final Map<Integer, String> map = new HashMap<Integer, String>();
  SqlSession sqlSession = MyBatisUtil.openSession();
  try {

sqlSession.select("com.mybatis3.mappers.StudentMapper.findAllStude
  nts",
  new ResultHandler() {
    @Override
    public void handleResult(ResultContext context) {
      Student student = (Student) context.getResultObject();
      map.put(student.getStudId(), student.getName());
    }
  }
);
} finally {
  sqlSession.close();
}
return map;
}
```

In the preceding code, we are providing an inline implementation of `ResultHandler`. Inside the `handleResult()` method, we are getting the current result object using `context.getResultObject()` that is a `Student` object because we configured `resultMap="StudentResult"` for the `findAllStudents` mapped statement. As the `handleResult()` method will be called for every row returned by the query, we are extracting the `studId` and `name` values from the `Student` object and populating the map.

Cache

Caching data that is loaded from the database is a common requirement for many applications to improve their performance. MyBatis provides in-built support for caching the query results loaded by mapped `SELECT` statements. By default, the first-level cache is enabled; this means that if you'll invoke the same `SELECT` statement within the same `SqlSession` interface, results will be fetched from the cache instead of the database.

We can add global second-level caches by adding the `<cache/>` element in SQL Mapper XML files.

When you'll add the `<cache/>` element the following will occur:

- All results from the `<select>` statements in the mapped statement file will be cached
- All the `<insert>`, `<update>`, and `<delete>` statements in the mapped statement file will flush the cache
- The cache will use a Least Recently Used (`LRU`) algorithm for eviction
- The cache will not flush on any sort of time-based schedule (no Flush Interval)
- The cache will store `1024` references to lists or objects (whatever the query method returns)
- The cache will be treated as a read/write cache; this means that the objects retrieved will not be shared and can safely be modified by the caller without it interfering with other potential modifications by other callers or threads

You can also customize this behavior by overriding the default attribute values as follows:

```
<cache eviction="FIFO" flushInterval="60000" size="512"
readOnly="true"/>
```

A description for each of the attributes is as follows:

- `eviction`: This is the cache eviction policy to be used. The default value is `LRU`. The possible values are `LRU` (least recently used), `FIFO`(first in first out), `SOFT`(soft reference), `WEAK`(weak reference).
- `flushInterval`: This is the cache flush interval in milliseconds. The default is not set. So, no flush interval is used and the cache is only flushed by calls to the statements.
- `size`: This represents the maximum number of elements that can be held in the cache. The default is `1024`, and you can set it to any positive integer.
- `readOnly`: A read-only cache will return the same instance of the cached object to all the callers. A read-write cache will return a copy (via serialization) of the cached object. The default is `false` and the possible values are `true` and `false`.

A cache configuration and cache instance are bound to the namespace of the SQL Mapper file, so all the statements in the same `namespace` table as the cache are bound by it.

The default cache configuration for a mapped statement is:

```
<select ... flushCache="false" useCache="true"/>
<insert ... flushCache="true"/>
<update ... flushCache="true"/>
<delete ... flushCache="true"/>
```

You can override this default behavior for any specific mapped statements; for example, by not using a cache for a `select` statement by setting the `useCache="false"` attribute.

In addition to in-built Cache support, MyBatis provides support for integration with popular third-party Cache libraries, such as `Ehcache`, `OSCache`, and `Hazelcast`. You can find more information on integrating third-party Cache libraries on the official MyBatis website `https://code.google.com/p/mybatis/wiki/Caches`.

Summary

In this chapter, we learned how to write SQL mapped statements using the Mapper XML files. We discussed how to configure simple statements, statements with one-to-one and one-to-many relationships, and how to map the results using `ResultMap`. We also looked into building dynamic queries, paginated results, and custom ResultSet handling. In the next chapter, we will discuss how to write mapped statements using annotations.

Where to buy this book

You can buy Java Persistence with MyBatis 3 from the Packt Publishing website: `http://www.packtpub.com/java-persistence-with-mybatis-3/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/java-persistence-with-mybatis-3/book