Spring Recipes: A Problem-Solution Approach

Copyright © 2008 by Gary Mak

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-979-2

ISBN-10 (pbk): 1-59059-979-9

ISBN-13 (electronic): 978-1-4302-0624-8

ISBN-10 (electronic): 1-4302-0624-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the US and other countries. Apress, Inc., is not affiliated with Sun Microsystems, Inc., and this book was written without endorsement from Sun Microsystems, Inc.

Lead Editors: Steve Anglin, Tom Welsh
Technical Reviewers: Sam Brannen, Kris Lander
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editor: Damon Larson
Associate Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor: Dina Quan
Proofreader: April Eddy
Indexer: Ron Strauss
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 7

Spring JDBC Support

In this chapter, you will learn how Spring can simplify your database access tasks. Data access is a common requirement for most enterprise applications, which usually require accessing data stored in relational databases. As an essential part of Java SE, JDBC (Java Database Connectivity) defines a set of standard APIs for you to access relational databases in a vendor-independent fashion.

The purpose of JDBC is to provide APIs through which you can execute SQL statements against a database. However, when using JDBC, you have to manage database-related resources by yourself and handle database exceptions explicitly. To make JDBC easier to use, Spring establishes a JDBC accessing framework by defining an abstract layer on top of the JDBC APIs.

As the heart of the Spring JDBC framework, JDBC templates are designed to provide template methods for different types of JDBC operations. Each template method is responsible for controlling the overall process and allows you to override particular tasks of the process. In this way, you can minimize your database access effort while retaining as much flexibility as possible.

Besides the JDBC template approach, the Spring JDBC framework supplies another more object-oriented approach for you to organize your data access logic. It enables you to model each database operation as a fine-grained operation object. Compared to the JDBC template approach, the JDBC operation object approach is just an alternative approach to organizing your data access logic. Some developers may prefer the former, while others like the latter better.

Upon finishing this chapter, you will be able to use the Spring JDBC framework to access relational databases. As part of the Spring data access module, the Spring JDBC framework is consistent with other parts of the module. So, learning about the JDBC framework is an ideal introduction to the entire data access module.

7-1. Problems with Direct JDBC

Suppose you are going to develop an application for vehicle registration, whose major functions are the basic CRUD (create, read, update, and delete) operations on vehicle records. These records will be stored in a relational database and accessed with JDBC. First, you design the following `Vehicle` class, which represents a vehicle in Java:

```
package com.apress.springrecipes.vehicle;

public class Vehicle {

    private String vehicleNo;
    private String color;
    private int wheel;
    private int seat;

    // Constructors, Getters and Setters
    ...
}
```

Setting Up the Application Database

Before developing your vehicle registration application, you have to set up the database for it. For the sake of low memory consumption and easy configuration, I have chosen Apache Derby (`http://db.apache.org/derby/`) as my database engine. Derby is an open source relational database engine provided under the Apache License and implemented in pure Java.

■ **Note** You can download the Apache Derby binary distribution (e.g., v10.3) from the Apache Derby web site and extract it to a directory of your choice to complete the installation.

Derby can run in either the embedded mode or the client/server mode. For testing purposes, the client/server mode is more appropriate because it allows you to inspect and edit data with any visual database tools that support JDBC—for example, the Eclipse Data Tools Platform (DTP).

■ **Note** To start the Derby server in the client/server mode, just execute the `startNetworkServer` script for your platform (located in the `bin` directory of the Derby installation).

After starting up the Derby network server on localhost, you can connect to it with the JDBC properties shown in Table 7-1.

■ **Note** You require Derby's client JDBC driver `derbyclient.jar` (located in the `lib` directory of the Derby installation) to connect to the Derby server.

Table 7-1. *JDBC Properties for Connecting to the Application Database*

Property	Value
Driver class	`org.apache.derby.jdbc.ClientDriver`
URL	`jdbc:derby://localhost:1527/vehicle;create=true`
Username	`app`
Password	`app`

The first time you connect to this database, the database instance `vehicle` will be created if it did not exist before, because you specified `create=true` in the URL. You can provide any values for the username and password, as Derby disables authentication by default. Next, you have to create the `VEHICLE` table for storing vehicle records with the following SQL statement. By default, this table will be created in the `APP` database schema.

```
CREATE TABLE VEHICLE (
    VEHICLE_NO    VARCHAR(10)    NOT NULL,
    COLOR         VARCHAR(10),
    WHEEL         INT,
    SEAT          INT,
    PRIMARY KEY (VEHICLE_NO)
);
```

Understanding the Data Access Object Design Pattern

A typical design mistake made by inexperienced developers is to mix different types of logic (e.g., presentation logic, business logic, and data access logic) in a single large module. This reduces the module's reusability and maintainability because of the tight coupling it introduces. The general purpose of the *Data Access Object (DAO)* pattern is to avoid these problems by separating data access logic from business logic and presentation logic. This pattern recommends that data access logic be encapsulated in independent modules called data access objects.

For your vehicle registration application, you can abstract the data access operations to insert, update, delete, and query a vehicle. These operations should be declared in a DAO interface to allow for different DAO implementation technologies.

```
package com.apress.springrecipes.vehicle;

public interface VehicleDao {

    public void insert(Vehicle vehicle);
    public void update(Vehicle vehicle);
    public void delete(Vehicle vehicle);
    public Vehicle findByVehicleNo(String vehicleNo);
}
```

Most parts of the JDBC APIs declare throwing `java.sql.SQLException`. But since this interface aims to abstract the data access operations only, it should not depend on the

implementation technology. So, it's unwise for this general interface to declare throwing the JDBC-specific `SQLException`. A common practice when implementing a DAO interface is to wrap this kind of exception with a runtime exception.

Implementing the DAO with JDBC

To access the database with JDBC, you create an implementation for this DAO interface (e.g., `JdbcVehicleDao`). Because your DAO implementation has to connect to the database to execute SQL statements, you may establish database connections by specifying the driver class name, database URL, username, and password. However, in JDBC 2.0 or higher, you can obtain database connections from a preconfigured `javax.sql.DataSource` object without knowing about the connection details.

```
package com.apress.springrecipes.vehicle;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

public class JdbcVehicleDao implements VehicleDao {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (?, ?, ?, ?)";
        Connection conn = null;
        try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setString(1, vehicle.getVehicleNo());
            ps.setString(2, vehicle.getColor());
            ps.setInt(3, vehicle.getWheel());
            ps.setInt(4, vehicle.getSeat());
            ps.executeUpdate();
            ps.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (conn != null) {
                try {
```

```
                conn.close();
            } catch (SQLException e) {}
        }
    }
}

public Vehicle findByVehicleNo(String vehicleNo) {
    String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
    Connection conn = null;
    try {
        conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, vehicleNo);

        Vehicle vehicle = null;
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            vehicle = new Vehicle(rs.getString("VEHICLE_NO"),
                    rs.getString("COLOR"), rs.getInt("WHEEL"),
                    rs.getInt("SEAT"));
        }
        rs.close();
        ps.close();
        return vehicle;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {}
        }
    }
}

public void update(Vehicle vehicle) {}

public void delete(Vehicle vehicle) {}
}
```

The vehicle insert operation is a typical JDBC update scenario. Each time this method is called, you obtain a connection from the data source and execute the SQL statement on this connection. Your DAO interface doesn't declare throwing any checked exceptions, so if a `SQLException` occurs, you have to wrap it with an unchecked `RuntimeException`. Finally, don't forget to release the connection in the `finally` block. Failing to do so may cause your application to run out of connections.

Here, the update and delete operations will be skipped because they are much the same as the insert operation from a technical point of view. For the query operation, you have to

extract the data from the returned result set to build a vehicle object in addition to executing the SQL statement.

Configuring a Data Source in Spring

The `javax.sql.DataSource` interface is a standard interface defined by the JDBC specification. There are many data source implementations provided by different vendors and projects. It is very easy to switch between different data source implementations because they implement the common `DataSource` interface. As a Java application framework, Spring also provides several convenient but less powerful data source implementations. The simplest one is `DriverManagerDataSource`, which opens a new connection every time it's requested.

■ **Note** To access a database instance running on the Derby server, you have to include `derbyclient.jar` (located in the `lib` directory of the Derby installation) in your classpath.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
            value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
            value="jdbc:derby://localhost:1527/vehicle;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>

    <bean id="vehicleDao"
        class="com.apress.springrecipes.vehicle.JdbcVehicleDao">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

`DriverManagerDataSource` is not an efficient data source implementation, as it opens a new connection for the client every time it's requested. Another data source implementation provided by Spring is `SingleConnectionDataSource`. As its name indicates, this maintains only a single connection that's reused all the time and never closed. Obviously, it is not suitable in a multithreaded environment.

Spring's own data source implementations are mainly used for testing purposes. However, many production data source implementations support connection pooling. For example, the DBCP (Database Connection Pooling Services) module of the Apache Commons Library has

several data source implementations that support connection pooling. Of these, `BasicDataSource` accepts the same connection properties as `DriverManagerDataSource` and allows you to specify the initial connection size and maximum active connections for the connection pool.

■ **Note** To use the data source implementations provided by DBCP, you have to include `commons-dbcp.jar` and `commons-pool.jar` (located in the `lib/jakarta-commons` directory of the Spring installation) in your classpath.

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName"
        value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url"
        value="jdbc:derby://localhost:1527/vehicle;create=true" />
    <property name="username" value="app" />
    <property name="password" value="app" />
    <property name="initialSize" value="2" />
    <property name="maxActive" value="5" />
</bean>
```

Many Java EE application servers build in data source implementations that you can configure from the server console. If you have a data source configured in an application server and exposed for JNDI lookup, you can use `JndiObjectFactoryBean` to look it up.

```
<bean id="dataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/VehicleDS" />
</bean>
```

In Spring 2.x, a JNDI lookup can be simplified by the `jndi-lookup` element defined in the `jee` schema.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">

    <jee:jndi-lookup id="dataSource" jndi-name="jdbc/VehicleDS" />
    ...
</beans>
```

Running the DAO

The following `Main` class tests your DAO by using it to insert a new vehicle to the database. If it succeeds, you can query the vehicle from the database immediately.

```
package com.apress.springrecipes.vehicle;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        Vehicle vehicle = new Vehicle("TEM0001", "Red", 4, 4);
        vehicleDao.insert(vehicle);

        vehicle = vehicleDao.findByVehicleNo("TEM0001");
        System.out.println("Vehicle No: " + vehicle.getVehicleNo());
        System.out.println("Color: " + vehicle.getColor());
        System.out.println("Wheel: " + vehicle.getWheel());
        System.out.println("Seat: " + vehicle.getSeat());
    }
}
```

Now you are able to implement a DAO using JDBC directly. However, as you can see from the preceding DAO implementation, most of the JDBC code is similar and needs to be repeated for each database operation. Such redundant code will make your DAO methods much longer and less readable.

7-2. Using a JDBC Template to Update a Database

Problem

To implement a JDBC update operation, you have to perform the following tasks, most of which are redundant:

1. Obtain a database connection from the data source.
2. Create a `PreparedStatement` object from the connection.
3. Bind the parameters to the `PreparedStatement` object.
4. Execute the `PreparedStatement` object.
5. Handle `SQLException`.
6. Clean up the statement object and connection.

Solution

The `JdbcTemplate` class declares a number of overloaded `update()` template methods to control the overall update process. Different versions of the `update()` method allow you to override different task subsets of the default process. The Spring JDBC framework predefines several callback interfaces to encapsulate different task subsets. You can implement one of these callback interfaces and pass its instance to the corresponding `update()` method to complete the process.

How It Works

Updating a Database with a Statement Creator

The first callback interface to introduce is `PreparedStatementCreator`. You implement this interface to override the statement creation task (task 2) and the parameter binding task (task 3) of the overall update process. To insert a vehicle into the database, you implement the `PreparedStatementCreator` interface as follows:

```
package com.apress.springrecipes.vehicle;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import org.springframework.jdbc.core.PreparedStatementCreator;

public class InsertVehicleStatementCreator implements PreparedStatementCreator {

    private Vehicle vehicle;

    public InsertVehicleStatementCreator(Vehicle vehicle) {
        this.vehicle = vehicle;
    }

    public PreparedStatement createPreparedStatement(Connection conn)
            throws SQLException {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (?, ?, ?, ?)";
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, vehicle.getVehicleNo());
        ps.setString(2, vehicle.getColor());
        ps.setInt(3, vehicle.getWheel());
        ps.setInt(4, vehicle.getSeat());
        return ps;
    }
}
```

When implementing the `PreparedStatementCreator` interface, you will get the database connection as the `createPreparedStatement()` method's argument. All you have to do in this

method is to create a PreparedStatement object on this connection and bind your parameters to this object. Finally, you have to return the PreparedStatement object as the method's return value. Notice that the method signature declares throwing SQLException, which means that you don't need to handle this kind of exception yourself.

Now you can use this statement creator to simplify the vehicle insert operation. First of all, you have to create an instance of the JdbcTemplate class and pass in the data source for this template to obtain a connection from it. Then you just make a call to the update() method and pass in your statement creator for the template to complete the update process.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(Vehicle vehicle) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.update(new InsertVehicleStatementCreator(vehicle));
    }
}
```

Typically, it is better to implement the PreparedStatementCreator interface and other callback interfaces as inner classes if they are used within one method only. This is because you can get access to the local variables and method arguments directly from the inner class, instead of passing them as constructor arguments. The only constraint on such variables and arguments is that they must be declared as final.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(final Vehicle vehicle) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.update(new PreparedStatementCreator() {

            public PreparedStatement createPreparedStatement(Connection conn)
                    throws SQLException {
                String sql = "INSERT INTO VEHICLE "
                        + "(VEHICLE_NO, COLOR, WHEEL, SEAT) "
                        + "VALUES (?, ?, ?, ?)";
                PreparedStatement ps = conn.prepareStatement(sql);
                ps.setString(1, vehicle.getVehicleNo());
                ps.setString(2, vehicle.getColor());
                ps.setInt(3, vehicle.getWheel());
                ps.setInt(4, vehicle.getSeat());
```

```
                return ps;
            }
        });
    }
}
```

Now you can delete the preceding `InsertVehicleStatementCreator` class, as it will not be used anymore.

Updating a Database with a Statement Setter

The second callback interface, `PreparedStatementSetter`, as its name indicates, performs only the parameter binding task (task 3) of the overall update process.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementSetter;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (?, ?, ?, ?)";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.update(sql, new PreparedStatementSetter() {

                    public void setValues(PreparedStatement ps)
                            throws SQLException {
                        ps.setString(1, vehicle.getVehicleNo());
                        ps.setString(2, vehicle.getColor());
                        ps.setInt(3, vehicle.getWheel());
                        ps.setInt(4, vehicle.getSeat());
                    }
                });
    }
}
```

Another version of the `update()` template method accepts a SQL statement and a `PreparedStatementSetter` object as arguments. This method will create a `PreparedStatement` object for you from your SQL statement. All you have to do with this interface is to bind your parameters to the `PreparedStatement` object.

Updating a Database with a SQL Statement and Parameter Values

Finally, the simplest version of the `update()` method accepts a SQL statement and an object array as statement parameters. It will create a `PreparedStatement` object from your SQL statement and bind the parameters for you. Therefore, you don't have to override any of the tasks in the update process.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (?, ?, ?, ?)";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.update(sql, new Object[] { vehicle.getVehicleNo(),
                vehicle.getColor(),vehicle.getWheel(), vehicle.getSeat() });
    }
}
```

Of the three different versions of the `update()` method introduced, the last is the simplest, as you don't have to implement any callback interfaces. In contrast, the first is the most flexible because you can do any preprocessing of the `PreparedStatement` object before its execution. In practice, you should always choose the simplest version that meets all your needs.

There are also other overloaded `update()` methods provided by the `JdbcTemplate` class. Please refer to its javadoc for details.

Batch Updating a Database

Suppose you would like to insert a batch of vehicles into the database. If you call the `insert()` method multiple times, it will be very slow as the SQL statement will be compiled repeatedly. So, it would be better to add a new method to the DAO interface for inserting a batch of vehicles.

```
package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
    ...
    public void insertBatch(List<Vehicle> vehicles);
}
```

The `JdbcTemplate` class also offers the `batchUpdate()` template method for batch update operations. It requires a SQL statement and a `BatchPreparedStatementSetter` object as arguments. In this method, the statement is compiled only once and executed multiple times. If your database driver supports JDBC 2.0, this method automatically makes use of the batch update features to increase performance.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BatchPreparedStatementSetter;
import org.springframework.jdbc.core.JdbcTemplate;
```

```
public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insertBatch(final List<Vehicle> vehicles) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (?, ?, ?, ?)";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {

                    public int getBatchSize() {
                        return vehicles.size();
                    }

                    public void setValues(PreparedStatement ps, int i)
                            throws SQLException {
                        Vehicle vehicle = vehicles.get(i);
                        ps.setString(1, vehicle.getVehicleNo());
                        ps.setString(2, vehicle.getColor());
                        ps.setInt(3, vehicle.getWheel());
                        ps.setInt(4, vehicle.getSeat());
                    }
                });
    }
}
```

You can test your batch insert operation with the following code snippet in the `Main` class:

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        Vehicle vehicle1 = new Vehicle("TEM0002", "Blue", 4, 4);
        Vehicle vehicle2 = new Vehicle("TEM0003", "Black", 4, 6);
        vehicleDao.insertBatch(
                Arrays.asList(new Vehicle[] { vehicle1, vehicle2 }));
    }
}
```

7-3. Using a JDBC Template to Query a Database

Problem

To implement a JDBC query operation, you have to perform the following tasks, two of which (task 5 and task 6) are additional as compared to an update operation:

1. Obtain a database connection from the data source.
2. Create a `PreparedStatement` object from the connection.
3. Bind the parameters to the `PreparedStatement` object.
4. Execute the `PreparedStatement` object.
5. Iterate the returned result set.
6. Extract data from the result set.
7. Handle `SQLException`.
8. Clean up the statement object and connection.

Solution

The `JdbcTemplate` class declares a number of overloaded `query()` template methods to control the overall query process. You can override the statement creation task (task 2) and the parameter binding task (task 3) by implementing the `PreparedStatementCreator` and `PreparedStatementSetter` interfaces, just as you did for the update operations. Moreover, the Spring JDBC framework supports multiple ways for you to override the data extraction task (task 6).

How It Works

Extracting Data with a Row Callback Handler

`RowCallbackHandler` is the primary interface that allows you to process the current row of the result set. One of the `query()` methods iterates the result set for you and calls your `RowCallbackHandler` for each row. So, the `processRow()` method will be called once for each row of the returned result set.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowCallbackHandler;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        final Vehicle vehicle = new Vehicle();
        jdbcTemplate.query(sql, new Object[] { vehicleNo },
                new RowCallbackHandler() {
                    public void processRow(ResultSet rs) throws SQLException {
                        vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
```

```
                    vehicle.setColor(rs.getString("COLOR"));
                    vehicle.setWheel(rs.getInt("WHEEL"));
                    vehicle.setSeat(rs.getInt("SEAT"));
                }
            });
        return vehicle;
    }
}
```

As there will be one row returned for the SQL query at maximum, you can create a vehicle object as a local variable and set its properties by extracting data from the result set. For a result set with more than one row, you should collect the objects as a list.

Extracting Data with a Row Mapper

The `RowMapper` interface is more general than `RowCallbackHandler`. Its purpose is to map a single row of the result set to a customized object, so it can be applied to a single-row result set as well as a multiple-row result set. From the viewpoint of reuse, it's better to implement the `RowMapper` interface as a normal class than as an inner class. In the `mapRow()` method of this interface, you have to construct the object that represents a row and return it as the method's return value.

```
package com.apress.springrecipes.vehicle;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class VehicleRowMapper implements RowMapper {

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Vehicle vehicle = new Vehicle();
        vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
        vehicle.setColor(rs.getString("COLOR"));
        vehicle.setWheel(rs.getInt("WHEEL"));
        vehicle.setSeat(rs.getInt("SEAT"));
        return vehicle;
    }
}
```

As mentioned, `RowMapper` can be used for either a single-row or multiple-row result set. When querying for a unique object like in `findByVehicleNo()`, you have to make a call to the `queryForObject()` method of `JdbcTemplate`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
```

```
public class JdbcVehicleDao implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        Vehicle vehicle = (Vehicle) jdbcTemplate.queryForObject(sql,
                new Object[] { vehicleNo }, new VehicleRowMapper());
        return vehicle;
    }
}
```

Spring 2.5 comes with a convenient `RowMapper` implementation, `BeanPropertyRowMapper`, which can automatically map a row to a new instance of the specified class. It first instantiates this class, and then maps each column value to a property by matching their names. It supports matching a property name (e.g., `vehicleNo`) to the same column name or the column name with underscores (e.g., `VEHICLE_NO`).

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        Vehicle vehicle = (Vehicle) jdbcTemplate.queryForObject(sql,
                new Object[] { vehicleNo } ,
                BeanPropertyRowMapper.newInstance(Vehicle.class));
        return vehicle;
    }
}
```

Querying for Multiple Rows

Now let's see how to query for a result set with multiple rows. For example, suppose you need a `findAll()` method in the DAO interface to get all vehicles.

```
package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
    ...
    public List<Vehicle> findAll();
}
```

Without the help of `RowMapper`, you can still call the `queryForList()` method and pass in a SQL statement. The returned result will be a list of maps. Each map stores a row of the result set with the column names as the keys.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public List<Vehicle> findAll() {
        String sql = "SELECT * FROM VEHICLE";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        List<Vehicle> vehicles = new ArrayList<Vehicle>();
        List<Map> rows = jdbcTemplate.queryForList(sql);
        for (Map row : rows) {
            Vehicle vehicle = new Vehicle();
            vehicle.setVehicleNo((String) row.get("VEHICLE_NO"));
            vehicle.setColor((String) row.get("COLOR"));
            vehicle.setWheel((Integer) row.get("WHEEL"));
            vehicle.setSeat((Integer) row.get("SEAT"));
            vehicles.add(vehicle);
        }
        return vehicles;
    }
}
```

You can test your `findAll()` method with the following code snippet in the `Main` class:

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        List<Vehicle> vehicles = vehicleDao.findAll();
        for (Vehicle vehicle : vehicles) {
            System.out.println("Vehicle No: " + vehicle.getVehicleNo());
            System.out.println("Color: " + vehicle.getColor());
            System.out.println("Wheel: " + vehicle.getWheel());
            System.out.println("Seat: " + vehicle.getSeat());
        }
    }
}
```

If you use a RowMapper object to map the rows in a result set, you will get a list of mapped objects from the query() method.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public List<Vehicle> findAll() {
        String sql = "SELECT * FROM VEHICLE";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        List<Vehicle> vehicles = jdbcTemplate.query(sql,
                BeanPropertyRowMapper.newInstance(Vehicle.class));
        return vehicles;
    }
}
```

Querying for a Single Value

Finally, let's see how to query for a single-row and single-column result set. As an example, add the following operations to the DAO interface:

```
package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
    ...
    public String getColor(String vehicleNo);
    public int countAll();
}
```

To query for a single string value, you can call the overloaded queryForObject() method, which requires an argument of java.lang.Class type. This method will help you to map the result value to the type you specified. For integer values, you can call the convenient method queryForInt().

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public String getColor(String vehicleNo) {
        String sql = "SELECT COLOR FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
```

```
        String color = (String) jdbcTemplate.queryForObject(sql,
                new Object[] { vehicleNo }, String.class);
        return color;
    }

    public int countAll() {
        String sql = "SELECT COUNT(*) FROM VEHICLE";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        int count = jdbcTemplate.queryForInt(sql);
        return count;
    }
}
```

You can test these two methods with the following code snippet in the `Main` class:

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        int count = vehicleDao.countAll();
        System.out.println("Vehicle Count: " + count);
        String color = vehicleDao.getColor("TEM0001");
        System.out.println("Color for [TEM0001]: " + color);
    }
}
```

7-4. Simplifying JDBC Template Creation

Problem

It's not efficient to create a new instance of `JdbcTemplate` every time you use it, as you have to repeat the creation statement and spend the cost of creating a new object.

Solution

The `JdbcTemplate` class is designed to be thread-safe, so you can declare a single instance of it in the IoC container and inject this instance into all your DAO instances. Furthermore, the Spring JDBC framework offers a convenient class, `JdbcDaoSupport`, to simplify your DAO implementation. This class declares a `jdbcTemplate` property, which can be injected from the IoC container or created automatically from a data source. Your DAO can extend this class to have this property inherited.

How It Works

Injecting a JDBC Template

Until now, you have created a new instance of `JdbcTemplate` in each DAO method. Actually, you can have it injected at the class level and use this injected instance in all DAO methods. For simplicity's sake, the following code only shows the change to the `insert()` method.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {

    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (?, ?, ?, ?)";

        jdbcTemplate.update(sql, new Object[] { vehicle.getVehicleNo(),
                vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });

    }
    ...
}
```

A JDBC template requires a data source to be set. You can inject this property by either a setter method or a constructor argument. Then, you can inject this JDBC template into your DAO.

```
<beans ...>
    ...
    <bean id="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="vehicleDao"
        class="com.apress.springrecipes.vehicle.JdbcVehicleDao">
        <property name="jdbcTemplate" ref="jdbcTemplate" />
    </bean>
</beans>
```

Extending the JdbcDaoSupport Class

The `JdbcDaoSupport` class has a `setDataSource()` method and a `setJdbcTemplate()` method. Your DAO class can extend this class to have these methods inherited. Then you can either inject a JDBC template directly or inject a data source for it to create a JDBC template. The following code fragment is taken from Spring's `JdbcDaoSupport` class:

```
package org.springframework.jdbc.core.support;
...
public abstract class JdbcDaoSupport extends DaoSupport {

    private JdbcTemplate jdbcTemplate;

    public final void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = createJdbcTemplate(dataSource);
        initTemplateConfig();
    }

    public final void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
        initTemplateConfig();
    }

    public final JdbcTemplate getJdbcTemplate() {
        return jdbcTemplate;
    }
    ...
}
```

In your DAO methods, you can simply call the `getJdbcTemplate()` method to retrieve the JDBC template. You also have to delete the `dataSource` and `jdbcTemplate` properties, as well as their setter methods, from your DAO class, as they have already been inherited. Again, for simplicity's sake, only the change to the `insert()` method is shown.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.support.JdbcDaoSupport;

public class JdbcVehicleDao extends JdbcDaoSupport implements VehicleDao {

    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (?, ?, ?, ?)";

        getJdbcTemplate().update(sql, new Object[] { vehicle.getVehicleNo(),
                vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
    }
    ...
}
```

By extending `JdbcDaoSupport`, your DAO class inherits the `setDataSource()` method. You can inject a data source into your DAO instance for it to create a JDBC template.

```
<beans ...>
    ...
    <bean id="vehicleDao"
        class="com.apress.springrecipes.vehicle.JdbcVehicleDao">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

7-5. Using the Simple JDBC Template with Java 1.5

Problem

The `JdbcTemplate` class works fine in most circumstances, but it can be further improved to take advantage of the Java 1.5 features.

Solution

`SimpleJdbcTemplate` is an evolution of `JdbcTemplate` that takes advantage of Java 1.5 features such as autoboxing, generics, and variable-length arguments to simplify its usage.

How It Works

Using a Simple JDBC Template to Update a Database

Many of the methods in the classic `JdbcTemplate` require statement parameters to be passed as an object array. In `SimpleJdbcTemplate`, they can be passed as variable-length arguments, which saves you the trouble of wrapping them in an array. To use `SimpleJdbcTemplate`, you can either instantiate it directly or retrieve its instance by extending the `SimpleJdbcDaoSupport` class.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
        VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (?, ?, ?, ?)";

        getSimpleJdbcTemplate().update(sql, vehicle.getVehicleNo(),
                vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat());
    }
    ...
}
```

SimpleJdbcTemplate offers a convenient batch update method for you to specify a SQL statement and a batch of parameters in the form of List<Object[]> so that you don't need to implement the BatchPreparedStatementSetter interface.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements VehicleDao {
    ...
    public void insertBatch(List<Vehicle> vehicles) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (?, ?, ?, ?)";

        List<Object[]> parameters = new ArrayList<Object[]>();
        for (Vehicle vehicle : vehicles) {
            parameters.add(new Object[] { vehicle.getVehicleNo(),
                    vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
        }
        getSimpleJdbcTemplate().batchUpdate(sql, parameters);
    }
}
```

Using a Simple JDBC Template to Query a Database

When implementing the RowMapper interface, the return type of the mapRow() method is java.lang.Object. ParameterizedRowMapper is a subinterface that takes a type parameter as the return type of the mapRow() method.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.ParameterizedRowMapper;

public class VehicleRowMapper implements ParameterizedRowMapper<Vehicle> {

    public Vehicle mapRow(ResultSet rs, int rowNum) throws SQLException {
        Vehicle vehicle = new Vehicle();
        vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
        vehicle.setColor(rs.getString("COLOR"));
        vehicle.setWheel(rs.getInt("WHEEL"));
        vehicle.setSeat(rs.getInt("SEAT"));
        return vehicle;
    }
}
```

Using SimpleJdbcTemplate with ParameterizedRowMapper can save you the trouble of casting the type of the returned result. For the queryForObject() method, the return type is determined by the ParameterizedRowMapper object's type parameter, which is Vehicle in this

case. Note that the statement parameters must be supplied at the end of the argument list since they are of variable length.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
        VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";

        // No need to cast into Vehicle anymore.
        Vehicle vehicle = getSimpleJdbcTemplate().queryForObject(sql,
                new VehicleRowMapper(), vehicleNo);
        return vehicle;
    }
}
```

Spring 2.5 also comes with a convenient `ParameterizedRowMapper` implementation, `ParameterizedBeanPropertyRowMapper`, which can automatically map a row to a new instance of the specified class.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.ParameterizedBeanPropertyRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
        VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";

        Vehicle vehicle = getSimpleJdbcTemplate().queryForObject(sql,
                ParameterizedBeanPropertyRowMapper.newInstance(Vehicle.class),
                vehicleNo);
        return vehicle;
    }
}
```

When using the classic `JdbcTemplate`, the `findAll()` method has a warning from the Java compiler because of an unchecked conversion from `List` to `List<Vehicle>`. This is because the return type of the `query()` method is `List` rather than the type-safe `List<Vehicle>`. After switching to `SimpleJdbcTemplate` and `ParameterizedBeanPropertyRowMapper`, the warning will be eliminated immediately, as the returned `List` is parameterized with the same type as the `ParameterizedRowMapper` argument.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.ParameterizedBeanPropertyRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
        VehicleDao {
    ...
    public List<Vehicle> findAll() {
        String sql = "SELECT * FROM VEHICLE";

        List<Vehicle> vehicles = getSimpleJdbcTemplate().query(sql,
                ParameterizedBeanPropertyRowMapper.newInstance(Vehicle.class));
        return vehicles;
    }
}
```

When querying for a single value with `SimpleJdbcTemplate`, the return type of the `queryForObject()` method will be determined by the `class` argument (e.g., `String.class`). So, there's no need for you to perform type casting manually. Note that the statement parameters of variable length must also be supplied at the end of the argument list.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
        VehicleDao {
    ...
    public String getColor(String vehicleNo) {
        String sql = "SELECT COLOR FROM VEHICLE WHERE VEHICLE_NO = ?";

        // No need to cast into String anymore.
        String color = getSimpleJdbcTemplate().queryForObject(sql,
                String.class, vehicleNo);
        return color;
    }
}
```

7-6. Using Named Parameters in a JDBC Template

Problem

In classic JDBC usages, SQL parameters are represented by the placeholder ? and bound by position. The trouble with positional parameters is that whenever the parameter order is changed, you have to change the parameter bindings as well. For a SQL statement with many parameters, it is very cumbersome to match the parameters by position.

Solution

Another option when binding SQL parameters in the Spring JDBC framework is to use named parameters. As the term implies, named SQL parameters are specified by name (starting with a colon) rather than by position. Named parameters are easier to maintain and also improve readability. Named parameters will be replaced by placeholders by the framework classes at runtime. Named parameters are only supported in `SimpleJdbcTemplate` and `NamedParameterJdbcTemplate`.

How It Works

When using named parameters in your SQL statement, you can provide the parameter values in a map with the parameter names as the keys.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
        VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        Map<String, Object> parameters = new HashMap<String, Object>();
        parameters.put("vehicleNo", vehicle.getVehicleNo());
        parameters.put("color", vehicle.getColor());
        parameters.put("wheel", vehicle.getWheel());
        parameters.put("seat", vehicle.getSeat());

        getSimpleJdbcTemplate().update(sql, parameters);
    }
    ...
}
```

You can also provide a SQL parameter source, whose responsibility is to offer SQL parameter values for named parameters. There are two implementations of the `SqlParameterSource` interface. The basic one is `MapSqlParameterSource`, which wraps a map as its parameter source.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
        VehicleDao {
```

```
    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        Map<String, Object> parameters = new HashMap<String, Object>();
        ...
        SqlParameterSource parameterSource =
            new MapSqlParameterSource(parameters);

        getSimpleJdbcTemplate().update(sql, parameterSource);
    }
    ...
}
```

Another implementation of `SqlParameterSource` is `BeanPropertySqlParameterSource`, which wraps a normal Java object as a SQL parameter source. For each of the named parameters, the property with the same name will be used as the parameter value.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
        VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        SqlParameterSource parameterSource =
            new BeanPropertySqlParameterSource(vehicle);

        getSimpleJdbcTemplate().update(sql, parameterSource);
    }
    ...
}
```

Named parameters can also be used in batch update. You can provide either a `Map` array or a `SqlParameterSource` array for the parameter values.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;
```

```
public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements VehicleDao {
    ...
    public void insertBatch(List<Vehicle> vehicles) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        List<SqlParameterSource> parameters = new ArrayList<SqlParameterSource>();
        for (Vehicle vehicle : vehicles) {
            parameters.add(new BeanPropertySqlParameterSource(vehicle));
        }

        getSimpleJdbcTemplate().batchUpdate(sql,
                parameters.toArray(new SqlParameterSource[0]));
    }
}
```

7-7. Modeling JDBC Operations As Fine-Grained Objects

Problem

You would like to organize your data access logic in a more object-oriented way by modeling each database operation as a fine-grained operation object.

Solution

Besides the JDBC template approach, the Spring JDBC framework supports modeling each database operation as a fine-grained operation object by extending one of the predefined JDBC operation classes (e.g., `SqlUpdate`, `MappingSqlQuery`, or `SqlFunction`). Before an operation object can be used, you have to perform the following tasks:

- Set the data source for this operation object.
- Set the SQL statement for this operation object.
- Declare the parameter types in correct order for this operation object.
- Call the `compile()` method on this operation object to compile the SQL statement.

The best place to perform these tasks is in an operation object's constructor. An operation object is thread-safe once it has been initialized with these tasks, so you can declare a single instance of it in the Spring IoC container for later use.

How It Works

Update Operation Objects

First, let's consider how to model the vehicle insert operation as an operation object. You can create the `VehicleInsertOperation` class by extending `SqlUpdate` for an update operation.

```
package com.apress.springrecipes.vehicle;

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class VehicleInsertOperation extends SqlUpdate {

    public VehicleInsertOperation(DataSource dataSource) {
        setDataSource(dataSource);
        setSql("INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                + "VALUES (?, ?, ?, ?)");
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.INTEGER));
        declareParameter(new SqlParameter(Types.INTEGER));
        compile();
    }

    public void perform(Vehicle vehicle) {
        update(new Object[] { vehicle.getVehicleNo(), vehicle.getColor(),
                vehicle.getWheel(), vehicle.getSeat() });
    }
}
```

Once an update operation object is properly initialized, you can make a call to its `update()` method and pass the parameter values in an object array. For better encapsulation, and to avoid invalid parameters, you can provide a custom `perform()` method to extract the parameters from a `Vehicle` object. Then you can declare its instance in the IoC container.

```
<bean id="vehicleInsertOperation"
    class="com.apress.springrecipes.vehicle.VehicleInsertOperation">
    <constructor-arg ref="dataSource" />
</bean>
```

The following code in the `Main` class shows how to use this operation object to insert a new vehicle:

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleInsertOperation operation =
            (VehicleInsertOperation) context.getBean("vehicleInsertOperation");
```

```
        Vehicle vehicle = new Vehicle("OBJ0001", "Red", 4, 4);
        operation.perform(vehicle);
    }
}
```

Query Operation Objects

For a query operation object, you can extend the `MappingSqlQuery` class to have the query and mapping logic centralized in a single class. For better encapsulation, you should also write a custom `perform()` method to pack the query parameters in an object array and cast the return type.

```
package com.apress.springrecipes.vehicle;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.MappingSqlQuery;

public class VehicleQueryOperation extends MappingSqlQuery {

    public VehicleQueryOperation(DataSource dataSource) {
        setDataSource(dataSource);
        setSql("SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?");
        declareParameter(new SqlParameter(Types.VARCHAR));
        compile();
    }

    protected Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Vehicle vehicle = new Vehicle();
        vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
        vehicle.setColor(rs.getString("COLOR"));
        vehicle.setWheel(rs.getInt("WHEEL"));
        vehicle.setSeat(rs.getInt("SEAT"));
        return vehicle;
    }

    public Vehicle perform(String vehicleNo) {
        return (Vehicle) findObject(new Object[] { vehicleNo });
    }
}
```

Then declare an instance of this operation object in the Spring IoC container.

```
<bean id="vehicleQueryOperation"
    class="com.apress.springrecipes.vehicle.VehicleQueryOperation">
    <constructor-arg ref="dataSource" />
</bean>
```

In the `Main` class, you can use this operation object to query for a vehicle by vehicle number.

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleQueryOperation operation =
            (VehicleQueryOperation) context.getBean("vehicleQueryOperation");
        Vehicle vehicle = operation.perform("OBJ0001");
        System.out.println("Vehicle No: " + vehicle.getVehicleNo());
        System.out.println("Color: " + vehicle.getColor());
        System.out.println("Wheel: " + vehicle.getWheel());
        System.out.println("Seat: " + vehicle.getSeat());
    }
}
```

Function Operation Objects

The `SqlFunction` operation object type is for querying a single value. For example, the vehicle count operation can be modeled with `SqlFunction`. Its `run()` method returns the result as an integer, which is appropriate in this case. If you want to return a result of some other type, you have to call the `runGeneric()` method.

```
package com.apress.springrecipes.vehicle;

import javax.sql.DataSource;

import org.springframework.jdbc.object.SqlFunction;

public class VehicleCountOperation extends SqlFunction {

    public VehicleCountOperation(DataSource dataSource) {
        setDataSource(dataSource);
        setSql("SELECT COUNT(*) FROM VEHICLE");
        compile();
    }

    public int perform() {
        return run();
    }
}
```

Then declare an instance of this operation object in the Spring IoC container.

```
<bean id="vehicleCountOperation"
    class="com.apress.springrecipes.vehicle.VehicleCountOperation">
    <constructor-arg ref="dataSource" />
</bean>
```

The following code in the `Main` class tests the operation object that queries for the total vehicle count:

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleCountOperation operation =
            (VehicleCountOperation) context.getBean("vehicleCountOperation");
        int count = operation.perform();
        System.out.println("Vehicle Count: " + count);
    }
}
```

7-8. Handling Exceptions in the Spring JDBC Framework

Problem

Many of the JDBC APIs declare throwing `java.sql.SQLException`, a checked exception that must be caught. It's very troublesome to handle this kind of exception every time you perform a database operation. You often have to define your own policy to handle this kind of exception. Failing to do so may lead to inconsistent exception handling.

Solution

The Spring framework offers a consistent data access exception-handling mechanism for its data access module, including the JDBC framework. In general, all exceptions thrown by the Spring JDBC framework are subclasses of `DataAccessException`, a type of `RuntimeException` that you are not forced to catch. It's the root exception class for all exceptions in Spring's data access module.

Figure 7-1 shows only part of the `DataAccessException` hierarchy in Spring's data access module. In total, there are more than 30 exception classes defined for different categories of data access exceptions.

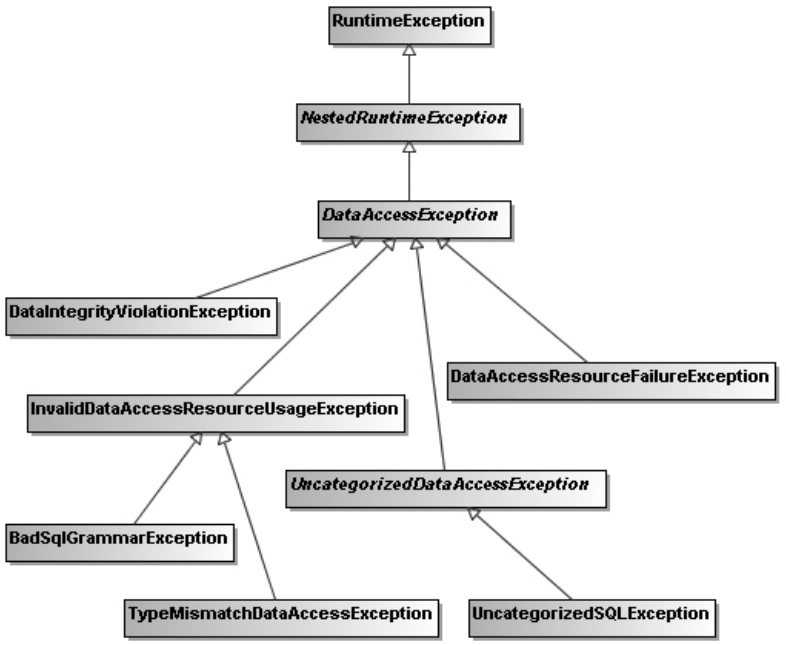

Figure 7-1. *Common exception classes in the DataAccessException hierarchy*

How It Works

Understanding Exception Handling in the Spring JDBC Framework

Until now, you haven't handled JDBC exceptions explicitly when using a JDBC template or JDBC operation objects. To help you understand the Spring JDBC framework's exception-handling mechanism, let's consider the following code fragment in the `Main` class, which inserts a vehicle. What happens if you insert a vehicle with a duplicate vehicle number?

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        Vehicle vehicle = new Vehicle("EX0001", "Green", 4, 4);
        vehicleDao.insert(vehicle);
    }
}
```

If you run the method twice, or the vehicle has already been inserted into the database, it will throw a `DataIntegrityViolationException`, a subclass of `DataAccessException`. In your DAO methods, you neither need to surround the code with a try/catch block nor declare throwing an exception in the method signature. This is because `DataAccessException` (and therefore its subclasses, including `DataIntegrityViolationException`) is an unchecked exception that you are not forced to catch. The direct parent class of `DataAccessException` is `NestedRuntimeException`, a core Spring exception class that wraps another exception in a `RuntimeException`.

When you use the classes of the Spring JDBC framework, they will catch `SQLException` for you and wrap it with one of the subclasses of `DataAccessException`. As this exception is a `RuntimeException`, you are not required to catch it.

But how does the Spring JDBC framework know which concrete exception in the `DataAccessException` hierarchy should be thrown? It's by looking at the `errorCode` and `SQLState` properties of the caught `SQLException`. As a `DataAccessException` wraps the underlying `SQLException` as the root cause, you can inspect the `errorCode` and `SQLState` properties with the following catch block:

```
package com.apress.springrecipes.vehicle;
...
import java.sql.SQLException;

import org.springframework.dao.DataAccessException;

public class Main {

    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        Vehicle vehicle = new Vehicle("EX0001", "Green", 4, 4);
        try {
            vehicleDao.insert(vehicle);
        } catch (DataAccessException e) {
            SQLException sqle = (SQLException) e.getCause();
            System.out.println("Error code: " + sqle.getErrorCode());
            System.out.println("SQL state: " + sqle.getSQLState());
        }
    }
}
```

When you insert the duplicate vehicle again, notice that Apache Derby returns the following error code and SQL state:

```
Error code : -1
SQL state : 23505
```

If you refer to the Apache Derby reference manual, you will find the error code description shown in Table 7-2.

Table 7-2. *Apache Derby's Error Code Description*

SQL State	Message Text
23505	The statement was aborted because it would have caused a duplicate key value in a unique or primary key constraint or unique index identified by '*<value>*' defined on '*<value>*'.

Then how does the Spring JDBC framework know that state `23505` should be mapped to `DataIntegrityViolationException`? The error code and SQL state are database specific, which means different database products may return different codes for the same kind of error. Moreover, some database products will specify the error in the `errorCode` property, while others (like Derby) will do so in the `SQLState` property.

As an open Java application framework, Spring understands the error codes of most popular database products. Because of the large number of error codes, however, it can only maintain mappings for the most frequently encountered errors. The mapping is defined in the `sql-error-codes.xml` file, located in the `org.springframework.jdbc.support` package. The following snippet for Apache Derby is taken from this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
    ...
    <bean id="Derby"
        class="org.springframework.jdbc.support.SQLErrorCodes">
        <property name="databaseProductName">
            <value>Apache Derby</value>
        </property>
        <property name="useSqlStateForTranslation">
            <value>true</value>
        </property>
        <property name="badSqlGrammarCodes">
            <value>
                42802,42821,42X01,42X02,42X03,42X04,42X05,42X06,42X07,42X08
            </value>
        </property>
        <property name="dataAccessResourceFailureCodes">
            <value>04501,08004,42Y07</value>
        </property>
        <property name="dataIntegrityViolationCodes">
            <value>22001,22005,23502,23503,23505,23513,X0Y32</value>
        </property>
        <property name="cannotAcquireLockCodes">
            <value>40XL1</value>
        </property>
        <property name="deadlockLoserCodes">
            <value>40001</value>
```

```
        </property>
    </bean>
</beans>
```

Note that the `databaseProductName` property is used to match the database product name returned by `Connection.getMetaData().getDatabaseProductName()`. This enables Spring to know which type of database is currently connecting. The `useSqlStateForTranslation` property means that the `SQLState` property, rather than the `errorCode` property, should be used to match the error code. Finally, the `SQLErrorCodes` class defines several categories for you to map database error codes. The code `23505` lies in the `dataIntegrityViolationCodes` category.

Customizing Data Access Exception Handling

The Spring JDBC framework only maps well-known error codes. Sometimes you may wish to customize the mapping yourself. For example, you might decide to add more codes to an existing category or define a custom exception for particular error codes.

In Table 7-2, the error code `23505` indicates a duplicate key error in Apache Derby. It is mapped by default to `DataIntegrityViolationException`. Suppose you would like to create a custom exception type, `DuplicateKeyException`, for this kind of error. It should extend `DataIntegrityViolationException`, as it is also a kind of data integrity violation error. Remember that for an exception to be thrown by the Spring JDBC framework, it must be compatible with the root exception class `DataAccessException`.

```
package com.apress.springrecipes.vehicle;

import org.springframework.dao.DataIntegrityViolationException;

public class DuplicateKeyException extends DataIntegrityViolationException {

    public DuplicateKeyException(String msg) {
        super(msg);
    }

    public DuplicateKeyException(String msg, Throwable cause) {
        super(msg, cause);
    }
}
```

By default, Spring will look up an exception from the `sql-error-codes.xml` file located in the `org.springframework.jdbc.support` package. However, you can override some of the mappings by providing a file with the same name in the root of the classpath. If Spring is able to find your custom file, it will look up an exception from your mapping first. However, if it does not find a suitable exception there, Spring will look up the default mapping.

For example, suppose you would like to map your custom `DuplicateKeyException` type to error code `23505`. You have to add the binding via a `CustomSQLErrorCodesTranslation` bean, and then add this bean to the `customTranslations` category.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
    <bean id="Derby"
        class="org.springframework.jdbc.support.SQLErrorCodes">
        <property name="databaseProductName">
            <value>Apache Derby</value>
        </property>
        <property name="useSqlStateForTranslation">
            <value>true</value>
        </property>
        <property name="customTranslations">
            <list>
                <ref local="duplicateKeyTranslation" />
            </list>
        </property>
    </bean>

    <bean id="duplicateKeyTranslation"
        class="org.springframework.jdbc.support.CustomSQLErrorCodesTranslation">
        <property name="errorCodes">
            <value>23505</value>
        </property>
        <property name="exceptionClass">
            <value>
                com.apress.springrecipes.vehicle.DuplicateKeyException
            </value>
        </property>
    </bean>
</beans>
```

Now if you remove the try/catch block surrounding the vehicle insert operation and insert a duplicate vehicle, the Spring JDBC framework will throw a `DuplicateKeyException` instead.

However, if you are not satisfied with the basic code-to-exception mapping strategy used by the `SQLErrorCodes` class, you may further implement the `SQLExceptionTranslator` interface and inject its instance into a JDBC template via the `setExceptionTranslator()` method.

7-9. Summary

In this chapter, you have learned how to use the Spring JDBC framework to simplify your database access tasks. When using JDBC directly without Spring's support, most of the JDBC code is similar and needs to be repeated for each database operation. Such redundant code will make your DAO methods much longer and less readable.

As the heart of the Spring JDBC framework, the JDBC template provides template methods for different types of JDBC operations. Each template method is responsible for controlling the overall process, and allows you to override particular tasks of the process by implementing a predefined callback interface.

The `JdbcTemplate` class is the classic Spring JDBC template. In order to take advantage of the Java 1.5 features, Spring provides `SimpleJdbcTemplate`, which makes use of features such as autoboxing, generics, and variable-length arguments to simplify its usage. Also, this class supports the usage of named parameters in SQL statements.

Besides the JDBC template approach, the Spring JDBC framework supports a more object-oriented approach by modeling each database operation as a fine-grained operation object. You can choose either approach according to your preferences and needs.

Spring offers a consistent data access exception-handling mechanism for its data access module, including the JDBC framework. If a `SQLException` is thrown, Spring will wrap it with an appropriate exception in the `DataAccessException` hierarchy. `DataAccessException` is an unchecked exception, so you are not forced to catch and handle every instance of it.

In the next chapter, you will learn about using the powerful transaction management features to manage transactions for your Spring applications.