

```
        this.sessionFactory = sessionFactory;
    }

    // ...

}
```

Last example we will show here is for typical JDBC support. You would have the `DataSource` injected into an initialization method where you would create a `JdbcTemplate` and other data access support classes like `SimpleJdbcCall` etc using this `DataSource`.


```
@Repository
public class JdbcMovieFinder implements MovieFinder {

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void init(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // ...

}
```



Please see the specific coverage of each persistence technology for details on how to configure the application context to take advantage of these annotations.

19. Data access with JDBC

19.1 Introduction to Spring Framework JDBC

The value-add provided by the Spring Framework JDBC abstraction is perhaps best shown by the sequence of actions outlined in the table below. The table shows what actions Spring will take care of and which actions are the responsibility of you, the application developer.

Table 19.1. Spring JDBC - who does what?

| Action | Spring | You |
|--|--------|-----|
| Define connection parameters. | | X |
| Open the connection. | X | |
| Specify the SQL statement. | | X |
| Declare parameters and provide parameter values | | X |
| Prepare and execute the statement. | X | |
| Set up the loop to iterate through the results (if any). | X | |

| Action | Spring | You |
|--|--------|-----|
| Do the work for each iteration. | | X |
| Process any exception. | X | |
| Handle transactions. | X | |
| Close the connection, statement and resultset. | X | |

The Spring Framework takes care of all the low-level details that can make JDBC such a tedious API to develop with.

19.1.1 Choosing an approach for JDBC database access

You can choose among several approaches to form the basis for your JDBC database access. In addition to three flavors of the `JdbcTemplate`, a new `SimpleJdbcInsert` and `SimpleJdbcCall` approach optimizes database metadata, and the RDBMS Object style takes a more object-oriented approach similar to that of JDO Query design. Once you start using one of these approaches, you can still mix and match to include a feature from a different approach. All approaches require a JDBC 2.0-compliant driver, and some advanced features require a JDBC 3.0 driver.

- *JdbcTemplate* is the classic Spring JDBC approach and the most popular. This "lowest level" approach and all others use a `JdbcTemplate` under the covers.
- *NamedParameterJdbcTemplate* wraps a `JdbcTemplate` to provide named parameters instead of the traditional JDBC "?" placeholders. This approach provides better documentation and ease of use when you have multiple parameters for an SQL statement.
- *SimpleJdbcInsert* and *SimpleJdbcCall* optimize database metadata to limit the amount of necessary configuration. This approach simplifies coding so that you only need to provide the name of the table or procedure and provide a map of parameters matching the column names. This only works if the database provides adequate metadata. If the database doesn't provide this metadata, you will have to provide explicit configuration of the parameters.
- *RDBMS Objects including MappingSqlQuery, SqlUpdate and StoredProcedure* requires you to create reusable and thread-safe objects during initialization of your data access layer. This approach is modeled after JDO Query wherein you define your query string, declare parameters, and compile the query. Once you do that, execute methods can be called multiple times with various parameter values passed in.

19.1.2 Package hierarchy

The Spring Framework's JDBC abstraction framework consists of four different packages, namely `core`, `datasource`, `object`, and `support`.

The `org.springframework.jdbc.core` package contains the `JdbcTemplate` class and its various callback interfaces, plus a variety of related classes. A subpackage named `org.springframework.jdbc.core.simple` contains the `SimpleJdbcInsert` and `SimpleJdbcCall` classes. Another subpackage named `org.springframework.jdbc.core.namedparam` contains the `NamedParameterJdbcTemplate` class and the related support classes. See [Section 19.2, "Using the JDBC core classes to control basic JDBC processing and error handling"](#), [Section 19.4, "JDBC batch operations"](#), and [Section 19.5, "Simplifying JDBC operations with the SimpleJdbc classes"](#).

The `org.springframework.jdbc.datasource` package contains a utility class for easy `DataSource` access, and various simple `DataSource` implementations that can be used for testing and running unmodified JDBC code outside of a Java EE container. A subpackage named `org.springframework.jdbc.datasource.embedded` provides support for

creating embedded databases using Java database engines such as HSQL, H2, and Derby. See [Section 19.3, “Controlling database connections”](#) and [Section 19.8, “Embedded database support”](#).

The `org.springframework.jdbc.object` package contains classes that represent RDBMS queries, updates, and stored procedures as thread-safe, reusable objects. See [Section 19.6, “Modeling JDBC operations as Java objects”](#). This approach is modeled by JDO, although objects returned by queries are naturally *disconnected* from the database. This higher level of JDBC abstraction depends on the lower-level abstraction in the `org.springframework.jdbc.core` package.

The `org.springframework.jdbc.support` package provides `SQLException` translation functionality and some utility classes. Exceptions thrown during JDBC processing are translated to exceptions defined in the `org.springframework.dao` package. This means that code using the Spring JDBC abstraction layer does not need to implement JDBC or RDBMS-specific error handling. All translated exceptions are unchecked, which gives you the option of catching the exceptions from which you can recover while allowing other exceptions to be propagated to the caller. See [Section 19.2.3, “SQLExceptionTranslator”](#).

19.2 Using the JDBC core classes to control basic JDBC processing and error handling

19.2.1 JdbcTemplate

The `JdbcTemplate` class is the central class in the JDBC core package. It handles the creation and release of resources, which helps you avoid common errors such as forgetting to close the connection. It performs the basic tasks of the core JDBC workflow such as statement creation and execution, leaving application code to provide SQL and extract results. The `JdbcTemplate` class executes SQL queries, update statements and stored procedure calls, performs iteration over `ResultSet`s and extraction of returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the `org.springframework.dao` package.

When you use the `JdbcTemplate` for your code, you only need to implement callback interfaces, giving them a clearly defined contract. The `PreparedStatementCreator` callback interface creates a prepared statement given a `Connection` provided by this class, providing SQL and any necessary parameters. The same is true for the `CallableStatementCreator` interface, which creates callable statements. The `RowCallbackHandler` interface extracts values from each row of a `ResultSet`.

The `JdbcTemplate` can be used within a DAO implementation through direct instantiation with a `DataSource` reference, or be configured in a Spring IoC container and given to DAOs as a bean reference.



The `DataSource` should always be configured as a bean in the Spring IoC container. In the first case the bean is given to the service directly; in the second case it is given to the prepared template.

All SQL issued by this class is logged at the `DEBUG` level under the category corresponding to the fully qualified class name of the template instance (typically `JdbcTemplate`, but it may be different if you are using a custom subclass of the `JdbcTemplate` class).

Examples of JdbcTemplate class usage

This section provides some examples of `JdbcTemplate` class usage. These examples are not an exhaustive list of all of the functionality exposed by the `JdbcTemplate`; see the attendant javadocs for that.

Querying (SELECT)

Here is a simple query for getting the number of rows in a relation:

```
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from t_actor", Integer.class);
```

A simple query using a bind variable:

```
int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(
    "select count(*) from t_actor where first_name = ?", Integer.class, "Joe");
```

Querying for a `String`:

```
String lastName = this.jdbcTemplate.queryForObject(
    "select last_name from t_actor where id = ?",
    new Object[]{1212L}, String.class);
```

Querying and populating a *single* domain object:

```
Actor actor = this.jdbcTemplate.queryForObject(
    "select first_name, last_name from t_actor where id = ?",
    new Object[]{1212L},
    new RowMapper<Actor>() {
        public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setFirstName(rs.getString("first_name"));
            actor.setLastName(rs.getString("last_name"));
            return actor;
        }
    });
```

Querying and populating a number of domain objects:

```
List<Actor> actors = this.jdbcTemplate.query(
    "select first_name, last_name from t_actor",
    new RowMapper<Actor>() {
        public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setFirstName(rs.getString("first_name"));
            actor.setLastName(rs.getString("last_name"));
            return actor;
        }
    });
```

If the last two snippets of code actually existed in the same application, it would make sense to remove the duplication present in the two `RowMapper` anonymous inner classes, and extract them out into a single class (typically a `static` nested class) that can then be referenced by DAO methods as needed. For example, it may be better to write the last code snippet as follows:

```
public List<Actor> findAllActors() {
    return this.jdbcTemplate.query( "select first_name, last_name from t_actor", new ActorMapper())
}

private static final class ActorMapper implements RowMapper<Actor> {

    public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
        Actor actor = new Actor();
```

```
        actor.setFirstName(rs.getString("first_name"));
        actor.setLastName(rs.getString("last_name"));
        return actor;
    }
}
```

Updating (INSERT/UPDATE/DELETE) with jdbcTemplate

You use the `update(..)` method to perform insert, update and delete operations. Parameter values are usually provided as var args or alternatively as an object array.

```
this.jdbcTemplate.update(
    "insert into t_actor (first_name, last_name) values (?, ?)",
    "Lionel", "Watling");
```

```
this.jdbcTemplate.update(
    "update t_actor set last_name = ? where id = ?",
    "Banjo", 5276L);
```

```
this.jdbcTemplate.update(
    "delete from actor where id = ?",
    Long.valueOf(actorId));
```

Other jdbcTemplate operations

You can use the `execute(..)` method to execute any arbitrary SQL, and as such the method is often used for DDL statements. It is heavily overloaded with variants taking callback interfaces, binding variable arrays, and so on.

```
this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");
```

The following example invokes a simple stored procedure. More sophisticated stored procedure support is [covered later](#).

```
this.jdbcTemplate.update(
    "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",
    Long.valueOf(unionId));
```

JdbcTemplate best practices

Instances of the `JdbcTemplate` class are *threadsafe once configured*. This is important because it means that you can configure a single instance of a `JdbcTemplate` and then safely inject this *shared* reference into multiple DAOs (or repositories). The `JdbcTemplate` is stateful, in that it maintains a reference to a `DataSource`, but this state is *not* conversational state.

A common practice when using the `JdbcTemplate` class (and the associated `NamedParameterJdbcTemplate` classes) is to configure a `DataSource` in your Spring configuration file, and then dependency-inject that shared `DataSource` bean into your DAO classes; the `JdbcTemplate` is created in the setter for the `DataSource`. This leads to DAOs that look in part like the following:

```
public class JdbcCorporateEventDao implements CorporateEventDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
```

```

        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // JDBC-backed implementations of the methods on the CorporateEventDao follow...
}

```

The corresponding configuration might look like this.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="corporateEventDao" class="com.example.JdbcCorporateEventDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <context:property-placeholder location="jdbc.properties"/>

</beans>

```

An alternative to explicit configuration is to use component-scanning and annotation support for dependency injection. In this case you annotate the class with `@Repository` (which makes it a candidate for component-scanning) and annotate the `DataSource` setter method with `@Autowired`.

```

@Repository
public class JdbcCorporateEventDao implements CorporateEventDao {

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // JDBC-backed implementations of the methods on the CorporateEventDao follow...
}

```

The corresponding XML configuration file would look like the following:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans

```

```

    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- Scans within the base package of the application for @Component classes to configure as beans -->
    <context:component-scan base-package="org.springframework.docs.test" />

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}" />
        <property name="url" value="${jdbc.url}" />
        <property name="username" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
    </bean>

    <context:property-placeholder location="jdbc.properties"/>

</beans>

```

If you are using Spring's `JdbcDaoSupport` class, and your various JDBC-backed DAO classes extend from it, then your sub-class inherits a `setDataSource(...)` method from the `JdbcDaoSupport` class. You can choose whether to inherit from this class. The `JdbcDaoSupport` class is provided as a convenience only.

Regardless of which of the above template initialization styles you choose to use (or not), it is seldom necessary to create a new instance of a `JdbcTemplate` class each time you want to execute SQL. Once configured, a `JdbcTemplate` instance is threadsafe. You may want multiple `JdbcTemplate` instances if your application accesses multiple databases, which requires multiple `DataSources`, and subsequently multiple differently configured `JdbcTemplates`.

19.2.2 NamedParameterJdbcTemplate

The `NamedParameterJdbcTemplate` class adds support for programming JDBC statements using named parameters, as opposed to programming JDBC statements using only classic placeholder (`'?'`) arguments. The `NamedParameterJdbcTemplate` class wraps a `JdbcTemplate`, and delegates to the wrapped `JdbcTemplate` to do much of its work. This section describes only those areas of the `NamedParameterJdbcTemplate` class that differ from the `JdbcTemplate` itself; namely, programming JDBC statements using named parameters.

```

// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

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

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(*) from T_ACTOR where first_name = :first_name";

    SqlParameterSource namedParameters = new MapSqlParameterSource("first_name", firstName);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);
}

```

Notice the use of the named parameter notation in the value assigned to the `sql` variable, and the corresponding value that is plugged into the `namedParameters` variable (of type `MapSqlParameterSource`).

Alternatively, you can pass along named parameters and their corresponding values to a `NamedParameterJdbcTemplate`

instance by using the `Map`-based style. The remaining methods exposed by the `NamedParameterJdbcOperations` and implemented by the `NamedParameterJdbcTemplate` class follow a similar pattern and are not covered here.

The following example shows the use of the `Map`-based style.

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

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

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(*) from T_ACTOR where first_name = :first_name";

    Map<String, String> namedParameters = Collections.singletonMap("first_name", firstName);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);
}
```

One nice feature related to the `NamedParameterJdbcTemplate` (and existing in the same Java package) is the `SqlParameterSource` interface. You have already seen an example of an implementation of this interface in one of the previous code snippet (the `MapSqlParameterSource` class). An `SqlParameterSource` is a source of named parameter values to a `NamedParameterJdbcTemplate`. The `MapSqlParameterSource` class is a very simple implementation that is simply an adapter around a `java.util.Map`, where the keys are the parameter names and the values are the parameter values.

Another `SqlParameterSource` implementation is the `BeanPropertySqlParameterSource` class. This class wraps an arbitrary `JavaBean` (that is, an instance of a class that adheres to [the JavaBean conventions](#)), and uses the properties of the wrapped `JavaBean` as the source of named parameter values.

```
public class Actor {

    private Long id;
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public Long getId() {
        return this.id;
    }

    // setters omitted...
}
```

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
```



```
public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActors(Actor exampleActor) {

    // notice how the named parameters match the properties of the above 'Actor' class
    String sql = "select count(*) from T_ACTOR where first_name = :firstName and last_name = :lastName";

    SqlParameterSource namedParameters = new BeanPropertySqlParameterSource(exampleActor);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);
}
```

Remember that the `NamedParameterJdbcTemplate` class wraps a classic `JdbcTemplate` template; if you need access to the wrapped `JdbcTemplate` instance to access functionality only present in the `JdbcTemplate` class, you can use the `getJdbcOperations()` method to access the wrapped `JdbcTemplate` through the `JdbcOperations` interface.

See also [the section called "JdbcTemplate best practices"](#) for guidelines on using the `NamedParameterJdbcTemplate` class in the context of an application.

19.2.3 SQLExceptionTranslator

`SQLExceptionTranslator` is an interface to be implemented by classes that can translate between `SQLExceptions` and Spring's own `org.springframework.dao.DataAccessException`, which is agnostic in regard to data access strategy. Implementations can be generic (for example, using SQLState codes for JDBC) or proprietary (for example, using Oracle error codes) for greater precision.

`SQLExceptionCodesSQLExceptionTranslator` is the implementation of `SQLExceptionTranslator` that is used by default. This implementation uses specific vendor codes. It is more precise than the `SQLState` implementation. The error code translations are based on codes held in a JavaBean type class called `SQLExceptionCodes`. This class is created and populated by an `SQLExceptionCodesFactory` which as the name suggests is a factory for creating `SQLExceptionCodes` based on the contents of a configuration file named `sql-error-codes.xml`. This file is populated with vendor codes and based on the `DatabaseProductName` taken from the `DatabaseMetaData`. The codes for the actual database you are using are used.

The `SQLExceptionCodesSQLExceptionTranslator` applies matching rules in the following sequence:



The `SQLExceptionCodesFactory` is used by default to define Error codes and custom exception translations. They are looked up in a file named `sql-error-codes.xml` from the classpath and the matching `SQLExceptionCodes` instance is located based on the database name from the database metadata of the database in use.

- Any custom translation implemented by a subclass. Normally the provided concrete `SQLExceptionCodesSQLExceptionTranslator` is used so this rule does not apply. It only applies if you have actually provided a subclass implementation.
- Any custom implementation of the `SQLExceptionTranslator` interface that is provided as the `customSQLExceptionTranslator` property of the `SQLExceptionCodes` class.
- The list of instances of the `CustomSQLExceptionCodesTranslation` class, provided for the `customTranslations` property of the `SQLExceptionCodes` class, are searched for a match.
- Error code matching is applied.

- Use the fallback translator. `SQLExceptionSubclassTranslator` is the default fallback translator. If this translation is not available then the next fallback translator is the `SQLStateSQLExceptionTranslator`.

You can extend `SQLErrorCodeSQLExceptionTranslator`:

```
public class CustomSQLErrorCodesTranslator extends SQLErrorCodeSQLExceptionTranslator {  
  
    protected DataAccessException customTranslate(String task, String sql, SQLException sqlEx) {  
        if (sqlEx.getErrorCode() == -12345) {  
            return new DeadlockLoserDataAccessException(task, sqlEx);  
        }  
        return null;  
    }  
}
```

In this example, the specific error code `-12345` is translated and other errors are left to be translated by the default translator implementation. To use this custom translator, it is necessary to pass it to the `JdbcTemplate` through the method `setExceptionHandler` and to use this `JdbcTemplate` for all of the data access processing where this translator is needed. Here is an example of how this custom translator can be used:

```
private JdbcTemplate jdbcTemplate;  
  
public void setDataSource(DataSource dataSource) {  
  
    // create a JdbcTemplate and set data source  
    this.jdbcTemplate = new JdbcTemplate();  
    this.jdbcTemplate.setDataSource(dataSource);  
  
    // create a custom translator and set the DataSource for the default translation lookup  
    CustomSQLErrorCodesTranslator tr = new CustomSQLErrorCodesTranslator();  
    tr.setDataSource(dataSource);  
    this.jdbcTemplate.setExceptionHandler(tr);  
  
}  
  
public void updateShippingCharge(long orderId, long pct) {  
    // use the prepared JdbcTemplate for this update  
    this.jdbcTemplate.update("update orders" +  
        " set shipping_charge = shipping_charge * ? / 100" +  
        " where id = ?", pct, orderId);  
}
```

The custom translator is passed a data source in order to look up the error codes in `sql-error-codes.xml`.

19.2.4 Executing statements

Executing an SQL statement requires very little code. You need a `DataSource` and a `JdbcTemplate`, including the convenience methods that are provided with the `JdbcTemplate`. The following example shows what you need to include for a minimal but fully functional class that creates a new table:

```
import javax.sql.DataSource;  
import org.springframework.jdbc.core.JdbcTemplate;  
  
public class ExecuteAStatement {
```

```
private JdbcTemplate jdbcTemplate;

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

public void doExecute() {
    this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");
}
}
```

19.2.5 Running queries

Some query methods return a single value. To retrieve a count or a specific value from one row, use `queryForObject(..)`. The latter converts the returned JDBC `Type` to the Java class that is passed in as an argument. If the type conversion is invalid, then an `InvalidDataAccessApiUsageException` is thrown. Here is an example that contains two query methods, one for an `int` and one that queries for a `String`.

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class RunAQuery {

    private JdbcTemplate jdbcTemplate;

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

    public int getCount() {
        return this.jdbcTemplate.queryForObject("select count(*) from mytable", Integer.class);
    }

    public String getName() {
        return this.jdbcTemplate.queryForObject("select name from mytable", String.class);
    }
}
```

In addition to the single result query methods, several methods return a list with an entry for each row that the query returned. The most generic method is `queryForList(..)` which returns a `List` where each entry is a `Map` with each entry in the map representing the column value for that row. If you add a method to the above example to retrieve a list of all the rows, it would look like this:

```
private JdbcTemplate jdbcTemplate;

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

public List<Map<String, Object>> getList() {
    return this.jdbcTemplate.queryForList("select * from mytable");
}
```

The list returned would look something like this:

```
[{name=Bob, id=1}, {name=Mary, id=2}]
```

19.2.6 Updating the database

The following example shows a column updated for a certain primary key. In this example, an SQL statement has placeholders for row parameters. The parameter values can be passed in as varargs or alternatively as an array of objects. Thus primitives should be wrapped in the primitive wrapper classes explicitly or using auto-boxing.

```
import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAnUpdate {

    private JdbcTemplate jdbcTemplate;

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

    public void setName(int id, String name) {
        this.jdbcTemplate.update("update mytable set name = ? where id = ?", name, id);
    }
}
```

19.2.7 Retrieving auto-generated keys

An `update()` convenience method supports the retrieval of primary keys generated by the database. This support is part of the JDBC 3.0 standard; see Chapter 13.6 of the specification for details. The method takes a `PreparedStatementCreator` as its first argument, and this is the way the required insert statement is specified. The other argument is a `KeyHolder`, which contains the generated key on successful return from the update. There is not a standard single way to create an appropriate `PreparedStatement` (which explains why the method signature is the way it is). The following example works on Oracle but may not work on other platforms:

```
final String INSERT_SQL = "insert into my_test (name) values(?)";
final String name = "Rob";

KeyHolder keyHolder = new GeneratedKeyHolder();
jdbcTemplate.update(
    new PreparedStatementCreator() {
        public PreparedStatement createPreparedStatement(Connection connection) throws SQLException {
            PreparedStatement ps = connection.prepareStatement(INSERT_SQL, new String[] {"id"});
            ps.setString(1, name);
            return ps;
        }
    },
    keyHolder);

// keyHolder.getKey() now contains the generated key
```

19.3 Controlling database connections

19.3.1 DataSource

Spring obtains a connection to the database through a `DataSource`. A `DataSource` is part of the JDBC specification and

is a generalized connection factory. It allows a container or a framework to hide connection pooling and transaction management issues from the application code. As a developer, you need not know details about how to connect to the database; that is the responsibility of the administrator that sets up the datasource. You most likely fill both roles as you develop and test code, but you do not necessarily have to know how the production data source is configured.

When using Spring's JDBC layer, you obtain a data source from JNDI or you configure your own with a connection pool implementation provided by a third party. Popular implementations are Apache Jakarta Commons DBCP and C3P0. Implementations in the Spring distribution are meant only for testing purposes and do not provide pooling.

This section uses Spring's `DriverManagerDataSource` implementation, and several additional implementations are covered later.



Only use the `DriverManagerDataSource` class should only be used for testing purposes since it does not provide pooling and will perform poorly when multiple requests for a connection are made.

You obtain a connection with `DriverManagerDataSource` as you typically obtain a JDBC connection. Specify the fully qualified classname of the JDBC driver so that the `DriverManager` can load the driver class. Next, provide a URL that varies between JDBC drivers. (Consult the documentation for your driver for the correct value.) Then provide a username and a password to connect to the database. Here is an example of how to configure a `DriverManagerDataSource` in Java code:

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
dataSource.setUrl("jdbc:hsqldb:hsqldb://localhost:");
dataSource.setUsername("sa");
dataSource.setPassword("");
```

Here is the corresponding XML configuration:

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="{jdbc.driverClassName}"/>
  <property name="url" value="{jdbc.url}"/>
  <property name="username" value="{jdbc.username}"/>
  <property name="password" value="{jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

The following examples show the basic connectivity and configuration for DBCP and C3P0. To learn about more options that help control the pooling features, see the product documentation for the respective connection pooling implementations.

DBCP configuration:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="{jdbc.driverClassName}"/>
  <property name="url" value="{jdbc.url}"/>
  <property name="username" value="{jdbc.username}"/>
  <property name="password" value="{jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

C3P0 configuration:

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">
    <property name="driverClass" value="{jdbc.driverClassName}"/>
    <property name="jdbcUrl" value="{jdbc.url}"/>
    <property name="user" value="{jdbc.username}"/>
    <property name="password" value="{jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

19.3.2 DataSourceUtils

The `DataSourceUtils` class is a convenient and powerful helper class that provides `static` methods to obtain connections from JNDI and close connections if necessary. It supports thread-bound connections with, for example, `DataSourceTransactionManager`.

19.3.3 SmartDataSource

The `SmartDataSource` interface should be implemented by classes that can provide a connection to a relational database. It extends the `DataSource` interface to allow classes using it to query whether the connection should be closed after a given operation. This usage is efficient when you know that you will reuse a connection.

19.3.4 AbstractDataSource

`AbstractDataSource` is an `abstract` base class for Spring's `DataSource` implementations that implements code that is common to all `DataSource` implementations. You extend the `AbstractDataSource` class if you are writing your own `DataSource` implementation.

19.3.5 SingleConnectionDataSource

The `SingleConnectionDataSource` class is an implementation of the `SmartDataSource` interface that wraps a *single Connection* that is *not* closed after each use. Obviously, this is not multi-threading capable.

If any client code calls `close` in the assumption of a pooled connection, as when using persistence tools, set the `suppressClose` property to `true`. This setting returns a close-suppressing proxy wrapping the physical connection. Be aware that you will not be able to cast this to a native Oracle `Connection` or the like anymore.

This is primarily a test class. For example, it enables easy testing of code outside an application server, in conjunction with a simple JNDI environment. In contrast to `DriverManagerDataSource`, it reuses the same connection all the time, avoiding excessive creation of physical connections.

19.3.6 DriverManagerDataSource

The `DriverManagerDataSource` class is an implementation of the standard `DataSource` interface that configures a plain JDBC driver through bean properties, and returns a new `Connection` every time.

This implementation is useful for test and stand-alone environments outside of a Java EE container, either as a `DataSource` bean in a Spring IoC container, or in conjunction with a simple JNDI environment. Pool-assuming `Connection.close()` calls will simply close the connection, so any `DataSource`-aware persistence code should work. However, using JavaBean-style connection pools such as `commons-dbcp` is so easy, even in a test environment, that it is almost always

preferable to use such a connection pool over `DriverManagerDataSource`.

19.3.7 TransactionAwareDataSourceProxy

`TransactionAwareDataSourceProxy` is a proxy for a target `DataSource`, which wraps that target `DataSource` to add awareness of Spring-managed transactions. In this respect, it is similar to a transactional JNDI `DataSource` as provided by a Java EE server.



It is rarely desirable to use this class, except when already existing code that must be called and passed a standard JDBC `DataSource` interface implementation. In this case, it's possible to still have this code be usable, and at the same time have this code participating in Spring managed transactions. It is generally preferable to write your own new code using the higher level abstractions for resource management, such as `JdbcTemplate` or `DataSourceUtils`.

(See the `TransactionAwareDataSourceProxy` javadocs for more details.)

19.3.8 DataSourceTransactionManager

The `DataSourceTransactionManager` class is a `PlatformTransactionManager` implementation for single JDBC datasources. It binds a JDBC connection from the specified data source to the currently executing thread, potentially allowing for one thread connection per data source.

Application code is required to retrieve the JDBC connection through `DataSourceUtils.getConnection(DataSource)` instead of Java EE's standard `DataSource.getConnection`. It throws unchecked `org.springframework.dao` exceptions instead of checked `SQLExceptions`. All framework classes like `JdbcTemplate` use this strategy implicitly. If not used with this transaction manager, the lookup strategy behaves exactly like the common one - it can thus be used in any case.

The `DataSourceTransactionManager` class supports custom isolation levels, and timeouts that get applied as appropriate JDBC statement query timeouts. To support the latter, application code must either use `JdbcTemplate` or call the `DataSourceUtils.applyTransactionTimeout(...)` method for each created statement.

This implementation can be used instead of `JtaTransactionManager` in the single resource case, as it does not require the container to support JTA. Switching between both is just a matter of configuration, if you stick to the required connection lookup pattern. JTA does not support custom isolation levels!

19.3.9 NativeJdbcExtractor

Sometimes you need to access vendor specific JDBC methods that differ from the standard JDBC API. This can be problematic if you are running in an application server or with a `DataSource` that wraps the `Connection`, `Statement` and `ResultSet` objects with its own wrapper objects. To gain access to the native objects you can configure your `JdbcTemplate` or `OracleLobHandler` with a `NativeJdbcExtractor`.

The `NativeJdbcExtractor` comes in a variety of flavors to match your execution environment:

- `SimpleNativeJdbcExtractor`
- `C3P0NativeJdbcExtractor`
- `CommonsDbcpNativeJdbcExtractor`

- JBossNativeJdbcExtractor
- WebLogicNativeJdbcExtractor
- WebSphereNativeJdbcExtractor
- XAPoolNativeJdbcExtractor

Usually the `SimpleNativeJdbcExtractor` is sufficient for unwrapping a `Connection` object in most environments. See the javadocs for more details.

19.4 JDBC batch operations

Most JDBC drivers provide improved performance if you batch multiple calls to the same prepared statement. By grouping updates into batches you limit the number of round trips to the database.

19.4.1 Basic batch operations with the JdbcTemplate

You accomplish `JdbcTemplate` batch processing by implementing two methods of a special interface, `BatchPreparedStatementSetter`, and passing that in as the second parameter in your `batchUpdate` method call. Use the `getBatchSize` method to provide the size of the current batch. Use the `setValues` method to set the values for the parameters of the prepared statement. This method will be called the number of times that you specified in the `getBatchSize` call. The following example updates the actor table based on entries in a list. The entire list is used as the batch in this example:

```
public class JdbcActorDao implements ActorDao {
    private JdbcTemplate jdbcTemplate;

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

    public int[] batchUpdate(final List<Actor> actors) {
        int[] updateCounts = jdbcTemplate.batchUpdate("update t_actor set first_name = ?, " +
            "last_name = ? where id = ?",
            new BatchPreparedStatementSetter() {
                public void setValues(PreparedStatement ps, int i) throws SQLException {
                    ps.setString(1, actors.get(i).getFirstName());
                    ps.setString(2, actors.get(i).getLastName());
                    ps.setLong(3, actors.get(i).getId().longValue());
                }

                public int getBatchSize() {
                    return actors.size();
                }
            });
        return updateCounts;
    }

    // ... additional methods
}
```

If you are processing a stream of updates or reading from a file, then you might have a preferred batch size, but the last batch might not have that number of entries. In this case you can use the `InterruptibleBatchPreparedStatementSetter` interface, which allows you to interrupt a batch once the input source is exhausted. The `isBatchExhausted` method allows you to signal the end of the batch.

19.4.2 Batch operations with a List of objects

Both the `JdbcTemplate` and the `NamedParameterJdbcTemplate` provides an alternate way of providing the batch update. Instead of implementing a special batch interface, you provide all parameter values in the call as a list. The framework loops over these values and uses an internal prepared statement setter. The API varies depending on whether you use named parameters. For the named parameters you provide an array of `SqlParameterSource`, one entry for each member of the batch. You can use the `SqlParameterSource.createBatch` method to create this array, passing in either an array of JavaBeans or an array of Maps containing the parameter values.

This example shows a batch update using named parameters:

```
public class JdbcActorDao implements ActorDao {
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

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

    public int[] batchUpdate(final List<Actor> actors) {
        SqlParameterSource[] batch = SqlParameterSourceUtils.createBatch(actors.toArray());
        int[] updateCounts = namedParameterJdbcTemplate.batchUpdate(
            "update t_actor set first_name = :firstName, last_name = :lastName where id = :id",
            batch);
        return updateCounts;
    }

    // ... additional methods
}
```

For an SQL statement using the classic "?" placeholders, you pass in a list containing an object array with the update values. This object array must have one entry for each placeholder in the SQL statement, and they must be in the same order as they are defined in the SQL statement.

The same example using classic JDBC "?" placeholders:

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;

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

    public int[] batchUpdate(final List<Actor> actors) {
        List<Object[]> batch = new ArrayList<Object[]>();
        for (Actor actor : actors) {
            Object[] values = new Object[] {
                actor.getFirstName(),
                actor.getLastName(),
                actor.getId();
            };
            batch.add(values);
        }
        int[] updateCounts = jdbcTemplate.batchUpdate(
            "update t_actor set first_name = ?, last_name = ? where id = ?",
            batch);
        return updateCounts;
    }
}
```

```
// ... additional methods  
}
```

All of the above batch update methods return an int array containing the number of affected rows for each batch entry. This count is reported by the JDBC driver. If the count is not available, the JDBC driver returns a -2 value.

19.4.3 Batch operations with multiple batches

The last example of a batch update deals with batches that are so large that you want to break them up into several smaller batches. You can of course do this with the methods mentioned above by making multiple calls to the `batchUpdate` method, but there is now a more convenient method. This method takes, in addition to the SQL statement, a Collection of objects containing the parameters, the number of updates to make for each batch and a `ParameterizedPreparedStatementSetter` to set the values for the parameters of the prepared statement. The framework loops over the provided values and breaks the update calls into batches of the size specified.

This example shows a batch update using a batch size of 100:

```
public class JdbcActorDao implements ActorDao {  
  
    private JdbcTemplate jdbcTemplate;  
  
    public void setDataSource(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
  
    public int[][] batchUpdate(final Collection<Actor> actors) {  
        int[][] updateCounts = jdbcTemplate.batchUpdate(  
            "update t_actor set first_name = ?, last_name = ? where id = ?",  
            actors,  
            100,  
            new ParameterizedPreparedStatementSetter<Actor>() {  
                public void setValues(PreparedStatement ps, Actor argument) throws SQLException {  
                    ps.setString(1, argument.getFirstName());  
                    ps.setString(2, argument.getLastName());  
                    ps.setLong(3, argument.getId().longValue());  
                }  
            });  
        return updateCounts;  
    }  
  
    // ... additional methods  
}
```

The batch update methods for this call returns an array of int arrays containing an array entry for each batch with an array of the number of affected rows for each update. The top level array's length indicates the number of batches executed and the second level array's length indicates the number of updates in that batch. The number of updates in each batch should be the batch size provided for all batches except for the last one that might be less, depending on the total number of update objects provided. The update count for each update statement is the one reported by the JDBC driver. If the count is not available, the JDBC driver returns a -2 value.

19.5 Simplifying JDBC operations with the SimpleJdbc classes

The `SimpleJdbcInsert` and `SimpleJdbcCall` classes provide a simplified configuration by taking advantage of database metadata that can be retrieved through the JDBC driver. This means there is less to configure up front, although you can override or turn off the metadata processing if you prefer to provide all the details in your code.

19.5.1 Inserting data using SimpleJdbcInsert

Let's start by looking at the `SimpleJdbcInsert` class with the minimal amount of configuration options. You should instantiate the `SimpleJdbcInsert` in the data access layer's initialization method. For this example, the initializing method is the `setDataSource` method. You do not need to subclass the `SimpleJdbcInsert` class; simply create a new instance and set the table name using the `withTableName` method. Configuration methods for this class follow the "fluid" style that returns the instance of the `SimpleJdbcInsert`, which allows you to chain all configuration methods. This example uses only one configuration method; you will see examples of multiple ones later.

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor = new SimpleJdbcInsert(dataSource).withTableName("t_actor");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Object>(3);
        parameters.put("id", actor.getId());
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        insertActor.execute(parameters);
    }

    // ... additional methods
}
```

The execute method used here takes a plain `java.util.Map` as its only parameter. The important thing to note here is that the keys used for the Map must match the column names of the table as defined in the database. This is because we read the metadata in order to construct the actual insert statement.

19.5.2 Retrieving auto-generated keys using SimpleJdbcInsert

This example uses the same insert as the preceding, but instead of passing in the id it retrieves the auto-generated key and sets it on the new Actor object. When you create the `SimpleJdbcInsert`, in addition to specifying the table name, you specify the name of the generated key column with the `usingGeneratedKeyColumns` method.

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor = new SimpleJdbcInsert(dataSource)
            .withTableName("t_actor")
            .usingGeneratedKeyColumns("id");
    }
}
```

```
        .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Object>(2);
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

The main difference when executing the insert by this second approach is that you do not add the id to the Map and you call the `executeAndReturnKey` method. This returns a `java.lang.Number` object with which you can create an instance of the numerical type that is used in our domain class. You cannot rely on all databases to return a specific Java class here; `java.lang.Number` is the base class that you can rely on. If you have multiple auto-generated columns, or the generated values are non-numeric, then you can use a `KeyHolder` that is returned from the `executeAndReturnKeyHolder` method.

19.5.3 Specifying columns for a SimpleJdbcInsert

You can limit the columns for an insert by specifying a list of column names with the `usingColumns` method:

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor = new SimpleJdbcInsert(dataSource)
            .withTableName("t_actor")
            .usingColumns("first_name", "last_name")
            .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Object>(2);
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

The execution of the insert is the same as if you had relied on the metadata to determine which columns to use.

19.5.4 Using SqlParameterSource to provide parameter values

Using a `Map` to provide parameter values works fine, but it's not the most convenient class to use. Spring provides a couple of

implementations of the `SqlParameterSource` interface that can be used instead. The first one is `BeanPropertySqlParameterSource`, which is a very convenient class if you have a JavaBean-compliant class that contains your values. It will use the corresponding getter method to extract the parameter values. Here is an example:

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor = new SimpleJdbcInsert(dataSource)
            .withTableName("t_actor")
            .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        SqlParameterSource parameters = new BeanPropertySqlParameterSource(actor);
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

Another option is the `MapSqlParameterSource` that resembles a Map but provides a more convenient `addValue` method that can be chained.

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.insertActor = new SimpleJdbcInsert(dataSource)
            .withTableName("t_actor")
            .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        SqlParameterSource parameters = new MapSqlParameterSource()
            .addValue("first_name", actor.getFirstName())
            .addValue("last_name", actor.getLastName());
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

As you can see, the configuration is the same; only the executing code has to change to use these alternative input classes.

19.5.5 Calling a stored procedure with SimpleJdbcCall

The `SimpleJdbcCall` class leverages metadata in the database to look up names of `in` and `out` parameters, so that you do not have to declare them explicitly. You can declare parameters if you prefer to do that, or if you have parameters such as `ARRAY` or `STRUCT` that do not have an automatic mapping to a Java class. The first example shows a simple procedure that returns only scalar values in `VARCHAR` and `DATE` format from a MySQL database. The example procedure reads a specified actor entry and returns `first_name`, `last_name`, and `birth_date` columns in the form of `out` parameters.

```
CREATE PROCEDURE read_actor (  
    IN in_id INTEGER,  
    OUT out_first_name VARCHAR(100),  
    OUT out_last_name VARCHAR(100),  
    OUT out_birth_date DATE)  
BEGIN  
    SELECT first_name, last_name, birth_date  
    INTO out_first_name, out_last_name, out_birth_date  
    FROM t_actor where id = in_id;  
END;
```

The `in_id` parameter contains the `id` of the actor you are looking up. The `out` parameters return the data read from the table.

The `SimpleJdbcCall` is declared in a similar manner to the `SimpleJdbcInsert`. You should instantiate and configure the class in the initialization method of your data access layer. Compared to the `StoredProcedure` class, you don't have to create a subclass and you don't have to declare parameters that can be looked up in the database metadata. Following is an example of a `SimpleJdbcCall` configuration using the above stored procedure. The only configuration option, in addition to the `DataSource`, is the name of the stored procedure.

```
public class JdbcActorDao implements ActorDao {  
  
    private JdbcTemplate jdbcTemplate;  
    private SimpleJdbcCall procReadActor;  
  
    public void setDataSource(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
        this.procReadActor = new SimpleJdbcCall(dataSource)  
            .withProcedureName("read_actor");  
    }  
  
    public Actor readActor(Long id) {  
        SqlParameterSource in = new MapSqlParameterSource()  
            .addValue("in_id", id);  
        Map out = procReadActor.execute(in);  
        Actor actor = new Actor();  
        actor.setId(id);  
        actor.setFirstName((String) out.get("out_first_name"));  
        actor.setLastName((String) out.get("out_last_name"));  
        actor.setBirthDate((Date) out.get("out_birth_date"));  
        return actor;  
    }  
  
    // ... additional methods  
}
```

The code you write for the execution of the call involves creating an `SqlParameterSource` containing the IN parameter. It's important to match the name provided for the input value with that of the parameter name declared in the stored procedure. The case does not have to match because you use metadata to determine how database objects should be referred to in a

stored procedure. What is specified in the source for the stored procedure is not necessarily the way it is stored in the database. Some databases transform names to all upper case while others use lower case or use the case as specified.

The `execute` method takes the IN parameters and returns a Map containing any `out` parameters keyed by the name as specified in the stored procedure. In this case they are `out_first_name`, `out_last_name` and `out_birth_date`.

The last part of the `execute` method creates an Actor instance to use to return the data retrieved. Again, it is important to use the names of the `out` parameters as they are declared in the stored procedure. Also, the case in the names of the `out` parameters stored in the results map matches that of the `out` parameter names in the database, which could vary between databases. To make your code more portable you should do a case-insensitive lookup or instruct Spring to use a `LinkedCaseInsensitiveMap`. To do the latter, you create your own `JdbcTemplate` and set the `setResultsMapCaseInsensitive` property to `true`. Then you pass this customized `JdbcTemplate` instance into the constructor of your `SimpleJdbcCall`. Here is an example of this configuration:

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcCall procReadActor;

    public void setDataSource(DataSource dataSource) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.procReadActor = new SimpleJdbcCall(jdbcTemplate)
            .withProcedureName("read_actor");
    }

    // ... additional methods

}
```

By taking this action, you avoid conflicts in the case used for the names of your returned `out` parameters.

19.5.6 Explicitly declaring parameters to use for a SimpleJdbcCall

You have seen how the parameters are deduced based on metadata, but you can declare them explicitly if you wish. You do this by creating and configuring `SimpleJdbcCall` with the `declareParameters` method, which takes a variable number of `SqlParameter` objects as input. See the next section for details on how to define an `SqlParameter`.



Explicit declarations are necessary if the database you use is not a Spring-supported database. Currently Spring supports metadata lookup of stored procedure calls for the following databases: Apache Derby, DB2, MySQL, Microsoft SQL Server, Oracle, and Sybase. We also support metadata lookup of stored functions for MySQL, Microsoft SQL Server, and Oracle.

You can opt to declare one, some, or all the parameters explicitly. The parameter metadata is still used where you do not declare parameters explicitly. To bypass all processing of metadata lookups for potential parameters and only use the declared parameters, you call the method `withoutProcedureColumnMetaDataAccess` as part of the declaration. Suppose that you have two or more different call signatures declared for a database function. In this case you call the `useInParameterNames` to specify the list of IN parameter names to include for a given signature.

The following example shows a fully declared procedure call, using the information from the preceding example.

```
public class JdbcActorDao implements ActorDao {
```

```
private SimpleJdbcCall procReadActor;

public void setDataSource(DataSource dataSource) {
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
    jdbcTemplate.setResultsMapCaseInsensitive(true);
    this.procReadActor = new SimpleJdbcCall(jdbcTemplate)
        .withProcedureName("read_actor")
        .withoutProcedureColumnMetaDataAccess()
        .useInParameterNames("in_id")
        .declareParameters(
            new SqlParameter("in_id", Types.NUMERIC),
            new SqlOutParameter("out_first_name", Types.VARCHAR),
            new SqlParameter("out_last_name", Types.VARCHAR),
            new SqlOutParameter("out_birth_date", Types.DATE)
        );
}

// ... additional methods
}
```

The execution and end results of the two examples are the same; this one specifies all details explicitly rather than relying on metadata.

19.5.7 How to define SqlParameter

To define a parameter for the SimpleJdbc classes and also for the RDBMS operations classes, covered in [Section 19.6](#), “Modeling JDBC operations as Java objects”, you use an `SqlParameter` or one of its subclasses. You typically specify the parameter name and SQL type in the constructor. The SQL type is specified using the `java.sql.Types` constants. We have already seen declarations like:

```
new SqlParameter("in_id", Types.NUMERIC),
new SqlOutParameter("out_first_name", Types.VARCHAR),
```

The first line with the `SqlParameter` declares an IN parameter. IN parameters can be used for both stored procedure calls and for queries using the `SqlQuery` and its subclasses covered in the following section.

The second line with the `SqlOutParameter` declares an `out` parameter to be used in a stored procedure call. There is also an `SqlInOutParameter` for `InOut` parameters, parameters that provide an `IN` value to the procedure and that also return a value.



Only parameters declared as `SqlParameter` and `SqlInOutParameter` will be used to provide input values. This is different from the `StoredProcedure` class, which for backwards compatibility reasons allows input values to be provided for parameters declared as `SqlOutParameter`.

For IN parameters, in addition to the name and the SQL type, you can specify a scale for numeric data or a type name for custom database types. For `out` parameters, you can provide a `RowMapper` to handle mapping of rows returned from a `REF` cursor. Another option is to specify an `SqlReturnType` that provides an opportunity to define customized handling of the return values.

19.5.8 Calling a stored function using SimpleJdbcCall

You call a stored function in almost the same way as you call a stored procedure, except that you provide a function name rather than a procedure name. You use the `withFunctionName` method as part of the configuration to indicate that we want to make a call to a function, and the corresponding string for a function call is generated. A specialized execute call, `executeFunction`, is used to execute the function and it returns the function return value as an object of a specified type, which means you do not have to retrieve the return value from the results map. A similar convenience method named `executeObject` is also available for stored procedures that only have one `out` parameter. The following example is based on a stored function named `get_actor_name` that returns an actor's full name. Here is the MySQL source for this function:

```
CREATE FUNCTION get_actor_name (in_id INTEGER)
RETURNS VARCHAR(200) READS SQL DATA
BEGIN
    DECLARE out_name VARCHAR(200);
    SELECT concat(first_name, ' ', last_name)
        INTO out_name
        FROM t_actor where id = in_id;
    RETURN out_name;
END;
```

To call this function we again create a `SimpleJdbcCall` in the initialization method.

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;
    private SimpleJdbcCall funcGetActorName;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.funcGetActorName = new SimpleJdbcCall(jdbcTemplate)
            .withFunctionName("get_actor_name");
    }

    public String getActorName(Long id) {
        SqlParameterSource in = new MapSqlParameterSource()
            .addValue("in_id", id);
        String name = funcGetActorName.executeFunction(String.class, in);
        return name;
    }

    // ... additional methods

}
```

The execute method used returns a `String` containing the return value from the function call.

19.5.9 Returning ResultSet/REF Cursor from a SimpleJdbcCall

Calling a stored procedure or function that returns a result set is a bit tricky. Some databases return result sets during the JDBC results processing while others require an explicitly registered `out` parameter of a specific type. Both approaches need additional processing to loop over the result set and process the returned rows. With the `SimpleJdbcCall` you use the `returningResultSet` method and declare a `RowMapper` implementation to be used for a specific parameter. In the case where the result set is returned during the results processing, there are no names defined, so the returned results will have to match the order in which you declare the `RowMapper` implementations. The name specified is still used to store the

processed list of results in the results map that is returned from the execute statement.

The next example uses a stored procedure that takes no IN parameters and returns all rows from the `t_actor` table. Here is the MySQL source for this procedure:

```
CREATE PROCEDURE read_all_actors()  
BEGIN  
    SELECT a.id, a.first_name, a.last_name, a.birth_date FROM t_actor a;  
END;
```

To call this procedure you declare the `RowMapper`. Because the class you want to map to follows the JavaBean rules, you can use a `BeanPropertyRowMapper` that is created by passing in the required class to map to in the `newInstance` method.

```
public class JdbcActorDao implements ActorDao {  
  
    private SimpleJdbcCall procReadAllActors;  
  
    public void setDataSource(DataSource dataSource) {  
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
        jdbcTemplate.setResultsMapCaseInsensitive(true);  
        this.procReadAllActors = new SimpleJdbcCall(jdbcTemplate)  
            .withProcedureName("read_all_actors")  
            .returningResultSet("actors",  
                BeanPropertyRowMapper.newInstance(Actor.class));  
    }  
  
    public List getActorsList() {  
        Map m = procReadAllActors.execute(new HashMap<String, Object>(0));  
        return (List) m.get("actors");  
    }  
  
    // ... additional methods  
}
```

The execute call passes in an empty Map because this call does not take any parameters. The list of Actors is then retrieved from the results map and returned to the caller.

19.6 Modeling JDBC operations as Java objects

The `org.springframework.jdbc.object` package contains classes that allow you to access the database in a more object-oriented manner. As an example, you can execute queries and get the results back as a list containing business objects with the relational column data mapped to the properties of the business object. You can also execute stored procedures and run update, delete, and insert statements.



Many Spring developers believe that the various RDBMS operation classes described below (with the exception of the `StoredProcedure` class) can often be replaced with straight `JdbcTemplate` calls. Often it is simpler to write a DAO method that simply calls a method on a `JdbcTemplate` directly (as opposed to encapsulating a query as a full-blown class).

However, if you are getting measurable value from using the RDBMS operation classes, continue using these classes.

19.6.1 SqlQuery

`SqlQuery` is a reusable, threadsafe class that encapsulates an SQL query. Subclasses must implement the `newRowMapper(...)` method to provide a `RowMapper` instance that can create one object per row obtained from iterating over the `ResultSet` that is created during the execution of the query. The `SqlQuery` class is rarely used directly because the `MappingSqlQuery` subclass provides a much more convenient implementation for mapping rows to Java classes. Other implementations that extend `SqlQuery` are `MappingSqlQueryWithParameters` and `UpdatableSqlQuery`.

19.6.2 MappingSqlQuery

`MappingSqlQuery` is a reusable query in which concrete subclasses must implement the abstract `mapRow(...)` method to convert each row of the supplied `ResultSet` into an object of the type specified. The following example shows a custom query that maps the data from the `t_actor` relation to an instance of the `Actor` class.

```
public class ActorMappingQuery extends MappingSqlQuery<Actor> {

    public ActorMappingQuery(DataSource ds) {
        super(ds, "select id, first_name, last_name from t_actor where id = ?");
        super.declareParameter(new SqlParameter("id", Types.INTEGER));
        compile();
    }

    @Override
    protected Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
        Actor actor = new Actor();
        actor.setId(rs.getLong("id"));
        actor.setFirstName(rs.getString("first_name"));
        actor.setLastName(rs.getString("last_name"));
        return actor;
    }

}
```

The class extends `MappingSqlQuery` parameterized with the `Actor` type. The constructor for this customer query takes the `DataSource` as the only parameter. In this constructor you call the constructor on the superclass with the `DataSource` and the SQL that should be executed to retrieve the rows for this query. This SQL will be used to create a `PreparedStatement` so it may contain place holders for any parameters to be passed in during execution. You must declare each parameter using the `declareParameter` method passing in an `SqlParameter`. The `SqlParameter` takes a name and the JDBC type as defined in `java.sql.Types`. After you define all parameters, you call the `compile()` method so the statement can be prepared and later executed. This class is thread-safe after it is compiled, so as long as these instances are created when the DAO is initialized they can be kept as instance variables and be reused.

```
private ActorMappingQuery actorMappingQuery;

@Autowired
public void setDataSource(DataSource dataSource) {
    this.actorMappingQuery = new ActorMappingQuery(dataSource);
}

public Customer getCustomer(Long id) {
    return actorMappingQuery.findObject(id);
}
```

The method in this example retrieves the customer with the id that is passed in as the only parameter. Since we only want one

object returned we simply call the convenience method `findObject` with the id as parameter. If we had instead a query that returned a list of objects and took additional parameters then we would use one of the execute methods that takes an array of parameter values passed in as varargs.

```
public List<Actor> searchForActors(int age, String namePattern) {
    List<Actor> actors = actorSearchMappingQuery.execute(age, namePattern);
    return actors;
}
```

19.6.3 SqlUpdate

The `SqlUpdate` class encapsulates an SQL update. Like a query, an update object is reusable, and like all `RdbmsOperation` classes, an update can have parameters and is defined in SQL. This class provides a number of `update(..)` methods analogous to the `execute(..)` methods of query objects. The `SqlUpdate` class is concrete. It can be subclassed, for example, to add a custom update method, as in the following snippet where it's simply called `execute`. However, you don't have to subclass the `SqlUpdate` class since it can easily be parameterized by setting SQL and declaring parameters.

```
import java.sql.Types;

import javax.sql.DataSource;

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

public class UpdateCreditRating extends SqlUpdate {

    public UpdateCreditRating(DataSource ds) {
        setDataSource(ds);
        setSql("update customer set credit_rating = ? where id = ?");
        declareParameter(new SqlParameter("creditRating", Types.NUMERIC));
        declareParameter(new SqlParameter("id", Types.NUMERIC));
        compile();
    }

    /**
     * @param id for the Customer to be updated
     * @param rating the new value for credit rating
     * @return number of rows updated
     */
    public int execute(int id, int rating) {
        return update(rating, id);
    }
}
```

19.6.4 StoredProcedure

The `StoredProcedure` class is a superclass for object abstractions of RDBMS stored procedures. This class is `abstract`, and its various `execute(..)` methods have `protected` access, preventing use other than through a subclass that offers tighter typing.

The inherited `sql` property will be the name of the stored procedure in the RDBMS.

To define a parameter for the `StoredProcedure` class, you use an `SqlParameter` or one of its subclasses. You must

specify the parameter name and SQL type in the constructor like in the following code snippet. The SQL type is specified using the `java.sql.Types` constants.

```
new SqlParameter("in_id", Types.NUMERIC),
new SqlOutParameter("out_first_name", Types.VARCHAR),
```

The first line with the `SqlParameter` declares an IN parameter. IN parameters can be used for both stored procedure calls and for queries using the `SqlQuery` and its subclasses covered in the following section.

The second line with the `SqlOutParameter` declares an `out` parameter to be used in the stored procedure call. There is also an `SqlInOutParameter` for `INOUT` parameters, parameters that provide an `in` value to the procedure and that also return a value.

For `IN` parameters, in addition to the name and the SQL type, you can specify a scale for numeric data or a type name for custom database types. For `out` parameters you can provide a `RowMapper` to handle mapping of rows returned from a REF cursor. Another option is to specify an `SqlReturnType` that enables you to define customized handling of the return values.

Here is an example of a simple DAO that uses a `StoredProcedure` to call a function, `sysdate()`, which comes with any Oracle database. To use the stored procedure functionality you have to create a class that extends `StoredProcedure`. In this example, the `StoredProcedure` class is an inner class, but if you need to reuse the `StoredProcedure` you declare it as a top-level class. This example has no input parameters, but an output parameter is declared as a date type using the class `SqlOutParameter`. The `execute()` method executes the procedure and extracts the returned date from the results `Map`. The results `Map` has an entry for each declared output parameter, in this case only one, using the parameter name as the key.

```
import java.sql.Types;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

public class StoredProcedureDao {

    private GetSysdateProcedure getSysdate;

    @Autowired
    public void init(DataSource dataSource) {
        this.getSysdate = new GetSysdateProcedure(dataSource);
    }

    public Date getSysdate() {
        return getSysdate.execute();
    }

    private class GetSysdateProcedure extends StoredProcedure {

        private static final String SQL = "sysdate";

        public GetSysdateProcedure(DataSource dataSource) {
            setDataSource(dataSource);
            setFunction(true);
            setSql(SQL);
        }
    }
}
```

```

        declareParameter(new SqlOutParameter("date", Types.DATE));
        compile();
    }

    public Date execute() {
        // the 'sysdate' sproc has no input parameters, so an empty Map is supplied...
        Map<String, Object> results = execute(new HashMap<String, Object>());
        Date sysdate = (Date) results.get("date");
        return sysdate;
    }
}

```

The following example of a `StoredProcedure` has two output parameters (in this case, Oracle REF cursors).

```

import oracle.jdbc.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;

public class TitlesAndGenresStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "AllTitlesAndGenres";

    public TitlesAndGenresStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new TitleMapper()));
        declareParameter(new SqlOutParameter("genres", OracleTypes.CURSOR, new GenreMapper()));
        compile();
    }

    public Map<String, Object> execute() {
        // again, this sproc has no input parameters, so an empty Map is supplied
        return super.execute(new HashMap<String, Object>());
    }
}

```

Notice how the overloaded variants of the `declareParameter(...)` method that have been used in the `TitlesAndGenresStoredProcedure` constructor are passed `RowMapper` implementation instances; this is a very convenient and powerful way to reuse existing functionality. The code for the two `RowMapper` implementations is provided below.

The `TitleMapper` class maps a `ResultSet` to a `Title` domain object for each row in the supplied `ResultSet`:

```

import org.springframework.jdbc.core.RowMapper;

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

import com.foo.domain.Title;

public final class TitleMapper implements RowMapper<Title> {

    public Title mapRow(ResultSet rs, int rowNum) throws SQLException {

```

```
Title title = new Title();
title.setId(rs.getLong("id"));
title.setName(rs.getString("name"));
return title;
}
}
```

The `GenreMapper` class maps a `ResultSet` to a `Genre` domain object for each row in the supplied `ResultSet`.

```
import org.springframework.jdbc.core.RowMapper;

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

import com.foo.domain.Genre;

public final class GenreMapper implements RowMapper<Genre> {

    public Genre mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Genre(rs.getString("name"));
    }
}
```

To pass parameters to a stored procedure that has one or more input parameters in its definition in the RDBMS, you can code a strongly typed `execute(..)` method that would delegate to the superclass' untyped `execute(Map parameters)` method (which has `protected` access); for example:

```
import oracle.jdbc.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;

import java.sql.Types;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

public class TitlesAfterDateStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "TitlesAfterDate";
    private static final String CUTOFF_DATE_PARAM = "cutoffDate";

    public TitlesAfterDateStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlParameter(CUTOFF_DATE_PARAM, Types.DATE));
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new TitleMapper()));
        compile();
    }

    public Map<String, Object> execute(Date cutoffDate) {
        Map<String, Object> inputs = new HashMap<String, Object>();
        inputs.put(CUTOFF_DATE_PARAM, cutoffDate);
        return super.execute(inputs);
    }
}
```

19.7 Common problems with parameter and data value handling

Common problems with parameters and data values exist in the different approaches provided by the Spring Framework JDBC.

19.7.1 Providing SQL type information for parameters

Usually Spring determines the SQL type of the parameters based on the type of parameter passed in. It is possible to explicitly provide the SQL type to be used when setting parameter values. This is sometimes necessary to correctly set NULL values.

You can provide SQL type information in several ways:

- Many update and query methods of the `JdbcTemplate` take an additional parameter in the form of an `int` array. This array is used to indicate the SQL type of the corresponding parameter using constant values from the `java.sql.Types` class. Provide one entry for each parameter.
- You can use the `SqlParameterValue` class to wrap the parameter value that needs this additional information. Create a new instance for each value and pass in the SQL type and parameter value in the constructor. You can also provide an optional scale parameter for numeric values.
- For methods working with named parameters, use the `SqlParameterSource` classes `BeanPropertySqlParameterSource` or `MapSqlParameterSource`. They both have methods for registering the SQL type for any of the named parameter values.

19.7.2 Handling BLOB and CLOB objects

You can store images, other binary data, and large chunks of text in the database. These large objects are called BLOBs (Binary Large Object) for binary data and CLOBs (Character Large Object) for character data. In Spring you can handle these large objects by using the `JdbcTemplate` directly and also when using the higher abstractions provided by RDBMS Objects and the `SimpleJdbc` classes. All of these approaches use an implementation of the `LobHandler` interface for the actual management of the LOB (Large Object) data. The `LobHandler` provides access to a `LobCreator` class, through the `getLobCreator` method, used for creating new LOB objects to be inserted.

The `LobCreator/LobHandler` provides the following support for LOB input and output:

- BLOB
 - `byte[]` — `getBlobAsBytes` and `setBlobAsBytes`
 - `InputStream` — `getBlobAsBinaryStream` and `setBlobAsBinaryStream`
- CLOB
 - `String` — `getClobAsString` and `setClobAsString`
 - `InputStream` — `getClobAsAsciiStream` and `setClobAsAsciiStream`
 - `Reader` — `getClobAsCharacterStream` and `setClobAsCharacterStream`

The next example shows how to create and insert a BLOB. Later you will see how to read it back from the database.

This example uses a `JdbcTemplate` and an implementation of the `AbstractLobCreatingPreparedStatementCallback`. It implements one method, `setValues`. This method provides a `LobCreator` that you use to set the values for the LOB columns in your SQL insert statement.

For this example we assume that there is a variable, `lobHandler`, that already is set to an instance of a `DefaultLobHandler`. You typically set this value through dependency injection.


```

final File blobIn = new File("spring2004.jpg");
final InputStream blobIs = new FileInputStream(blobIn);
final File clobIn = new File("large.txt");
final InputStream clobIs = new FileInputStream(clobIn);
final InputStreamReader clobReader = new InputStreamReader(clobIs);
jdbcTemplate.execute(
    "INSERT INTO lob_table (id, a_clob, a_blob) VALUES (?, ?, ?)",
    new AbstractLobCreatingPreparedStatementCallback(lobHandler) { ❶
        protected void setValues(PreparedStatement ps, LobCreator lobCreator) throws SQLException {
            ps.setLong(1, 1L);
            lobCreator.setClobAsCharacterStream(ps, 2, clobReader, (int)clobIn.length()); ❷
            lobCreator.setBlobAsBinaryStream(ps, 3, blobIs, (int)blobIn.length()); ❸
        }
    }
);
blobIs.close();
clobReader.close();

```

- ❶ Pass in the `lobHandler` that in this example is a plain `DefaultLobHandler`.
- ❷ Using the method `setClobAsCharacterStream`, pass in the contents of the CLOB.
- ❸ Using the method `setBlobAsBinaryStream`, pass in the contents of the BLOB.



If you invoke the `setBlobAsBinaryStream`, `setClobAsAsciiStream`, or `setClobAsCharacterStream` method on the `LobCreator` returned from `DefaultLobHandler.getLobCreator()`, you can optionally specify a negative value for the `contentLength` argument. If the specified content length is negative, the `DefaultLobHandler` will use the JDBC 4.0 variants of the set-stream methods without a length parameter; otherwise, it will pass the specified length on to the driver.

Consult the documentation for the JDBC driver in use to verify support for streaming a LOB without providing the content length.

Now it's time to read the LOB data from the database. Again, you use a `JdbcTemplate` with the same instance variable `lobHandler` and a reference to a `DefaultLobHandler`.

```

List<Map<String, Object>> l = jdbcTemplate.query("select id, a_clob, a_blob from lob_table",
    new RowMapper<Map<String, Object>>() {
        public Map<String, Object> mapRow(ResultSet rs, int i) throws SQLException {
            Map<String, Object> results = new HashMap<String, Object>();
            String clobText = lobHandler.getClobAsString(rs, "a_clob"); ❶
            results.put("CLOB", clobText); byte[] blobBytes = lobHandler.getBlobAsBytes(rs, "a_blob"); ❷
            results.put("BLOB", blobBytes); return results; } });

```

- ❶ Using the method `getClobAsString`, retrieve the contents of the CLOB.
- ❷ Using the method `getBlobAsBytes`, retrieve the contents of the BLOB.

19.7.3 Passing in lists of values for IN clause

The SQL standard allows for selecting rows based on an expression that includes a variable list of values. A typical example would be `select * from T_ACTOR where id in (1, 2, 3)`. This variable list is not directly supported for prepared statements by the JDBC standard; you cannot declare a variable number of placeholders. You need a number of variations with the desired number of placeholders prepared, or you need to generate the SQL string dynamically once you know how

many placeholders are required. The named parameter support provided in the `NamedParameterJdbcTemplate` and `JdbcTemplate` takes the latter approach. Pass in the values as a `java.util.List` of primitive objects. This list will be used to insert the required placeholders and pass in the values during the statement execution.



Be careful when passing in many values. The JDBC standard does not guarantee that you can use more than 100 values for an `in` expression list. Various databases exceed this number, but they usually have a hard limit for how many values are allowed. Oracle's limit is 1000.

In addition to the primitive values in the value list, you can create a `java.util.List` of object arrays. This list would support multiple expressions defined for the `in` clause such as

`select * from T_ACTOR where (id, last_name) in 1, 'Johnson'), (2, 'Harrop')`. This of course requires that your database supports this syntax.

19.7.4 Handling complex types for stored procedure calls

When you call stored procedures you can sometimes use complex types specific to the database. To accommodate these types, Spring provides a `SqlReturnType` for handling them when they are returned from the stored procedure call and `SqlTypeValue` when they are passed in as a parameter to the stored procedure.

Here is an example of returning the value of an Oracle `STRUCT` object of the user declared type `ITEM_TYPE`. The `SqlReturnType` interface has a single method named `getTypeValue` that must be implemented. This interface is used as part of the declaration of an `SqlOutParameter`.

```
final TestItem = new TestItem(123L, "A test item",
    new SimpleDateFormat("yyyy-M-d").parse("2010-12-31"));

declareParameter(new SqlOutParameter("item", OracleTypes.STRUCT, "ITEM_TYPE",
    new SqlReturnType() {
        public Object getTypeValue(CallableStatement cs, int colIdx, int sqlType, String typeName)
            STRUCT struct = (STRUCT) cs.getObject(colIdx);
            Object[] attr = struct.getAttributes();
            TestItem item = new TestItem();
            item.setId(((Number) attr[0]).longValue());
            item.setDescription((String) attr[1]);
            item.setExpirationDate((java.util.Date) attr[2]);
            return item;
    }
));
```

You use the `SqlTypeValue` to pass in the value of a Java object like `TestItem` into a stored procedure. The `SqlTypeValue` interface has a single method named `createTypeValue` that you must implement. The active connection is passed in, and you can use it to create database-specific objects such as `StructDescriptor`s, as shown in the following example, or `ArrayDescriptor`s.

```
final TestItem = new TestItem(123L, "A test item",
    new SimpleDateFormat("yyyy-M-d").parse("2010-12-31"));

SqlTypeValue value = new AbstractSqlTypeValue() {
    protected Object createTypeValue(Connection conn, int sqlType, String typeName) throws SQLException
        StructDescriptor itemDescriptor = new StructDescriptor(typeName, conn);
        Struct item = new STRUCT(itemDescriptor, conn,
            new Object[] {
```

```

        testItem.getId(),
        testItem.getDescription(),
        new java.sql.Date(testItem.getExpirationDate().getTime())
    });
    return item;
}
};

```

This `SqlTypeValue` can now be added to the Map containing the input parameters for the execute call of the stored procedure.

Another use for the `SqlTypeValue` is passing in an array of values to an Oracle stored procedure. Oracle has its own internal `ARRAY` class that must be used in this case, and you can use the `SqlTypeValue` to create an instance of the Oracle `ARRAY` and populate it with values from the Java `ARRAY`.

```

final Long[] ids = new Long[] {1L, 2L};

SqlTypeValue value = new AbstractSqlTypeValue() {
    protected Object createTypeValue(Connection conn, int sqlType, String typeName) throws SQLException {
        ArrayDescriptor arrayDescriptor = new ArrayDescriptor(typeName, conn);
        ARRAY idArray = new ARRAY(arrayDescriptor, conn, ids);
        return idArray;
    }
};

```

19.8 Embedded database support

The `org.springframework.jdbc.datasource.embedded` package provides support for embedded Java database engines. Support for `HSQL`, `H2`, and `Derby` is provided natively. You can also use an extensible API to plug in new embedded database types and `DataSource` implementations.

19.8.1 Why use an embedded database?

An embedded database is useful during the development phase of a project because of its lightweight nature. Benefits include ease of configuration, quick startup time, testability, and the ability to rapidly evolve SQL during development.

19.8.2 Creating an embedded database using Spring XML

If you want to expose an embedded database instance as a bean in a Spring `ApplicationContext`, use the `embedded-database` tag in the `spring-jdbc` namespace:

```

<jdbc:embedded-database id="dataSource" generate-name="true">
  <jdbc:script location="classpath:schema.sql"/>
  <jdbc:script location="classpath:test-data.sql"/>
</jdbc:embedded-database>

```

The preceding configuration creates an embedded HSQL database populated with SQL from `schema.sql` and `test-data.sql` resources in the root of the classpath. In addition, as a best practice, the embedded database will be assigned a uniquely generated name. The embedded database is made available to the Spring container as a bean of type `javax.sql.DataSource` which can then be injected into data access objects as needed.

19.8.3 Creating an embedded database programmatically

The `EmbeddedDatabaseBuilder` class provides a fluent API for constructing an embedded database programmatically. Use this when you need to create an embedded database in a standalone environment or in a standalone integration test like in the following example.

```
EmbeddedDatabase db = new EmbeddedDatabaseBuilder()
    .generateUniqueName(true)
    .setType(H2)
    .setScriptEncoding("UTF-8")
    .ignoreFailedDrops(true)
    .addScript("schema.sql")
    .addScripts("user_data.sql", "country_data.sql")
    .build();

// perform actions against the db (EmbeddedDatabase extends javax.sql.DataSource)

db.shutdown();
```

Consult the Javadoc for `EmbeddedDatabaseBuilder` for further details on all supported options.

The `EmbeddedDatabaseBuilder` can also be used to create an embedded database using Java Config like in the following example.

```
@Configuration
public class DataSourceConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .generateUniqueName(true)
            .setType(H2)
            .setScriptEncoding("UTF-8")
            .ignoreFailedDrops(true)
            .addScript("schema.sql")
            .addScripts("user_data.sql", "country_data.sql")
            .build();
    }
}
```

19.8.4 Selecting the embedded database type

Using HSQL

Spring supports HSQL 1.8.0 and above. HSQL is the default embedded database if no type is specified explicitly. To specify HSQL explicitly, set the `type` attribute of the `embedded-database` tag to `HSQL`. If you are using the builder API, call the `setType(EmbeddedDatabaseType)` method with `EmbeddedDatabaseType.HSQL`.

Using H2

Spring supports the H2 database as well. To enable H2, set the `type` attribute of the `embedded-database` tag to `H2`. If you are using the builder API, call the `setType(EmbeddedDatabaseType)` method with `EmbeddedDatabaseType.H2`.

Using Derby

Spring also supports Apache Derby 10.5 and above. To enable Derby, set the `type` attribute of the `embedded-database` tag to `DERBY`. If you are using the builder API, call the `setType(EmbeddedDatabaseType)` method with `EmbeddedDatabaseType.DERBY`.

19.8.5 Testing data access logic with an embedded database

Embedded databases provide a lightweight way to test data access code. The following is a data access integration test template that uses an embedded database. Using a template like this can be useful for *one-offs* when the embedded database does not need to be reused across test classes. However, if you wish to create an embedded database that is shared within a test suite, consider using the [Spring TestContext Framework](#) and configuring the embedded database as a bean in the Spring `ApplicationContext` as described in [Section 19.8.2, “Creating an embedded database using Spring XML”](#) and [Section 19.8.3, “Creating an embedded database programmatically”](#).

```
public class DataAccessIntegrationTestTemplate {

    private EmbeddedDatabase db;

    @Before
    public void setUp() {
        // creates an HSQL in-memory database populated from default scripts
        // classpath:schema.sql and classpath:data.sql
        db = new EmbeddedDatabaseBuilder()
            .generateUniqueName(true)
            .addDefaultScripts()
            .build();
    }

    @Test
    public void testDataAccess() {
        JdbcTemplate template = new JdbcTemplate(db);
        template.query( /* ... */ );
    }

    @After
    public void tearDown() {
        db.shutdown();
    }

}
```

19.8.6 Generating unique names for embedded databases

Development teams often encounter errors with embedded databases if their test suite inadvertently attempts to recreate additional instances of the same database. This can happen quite easily if an XML configuration file or `@Configuration` class is responsible for creating an embedded database and the corresponding configuration is then reused across multiple testing scenarios within the same test suite (i.e., within the same JVM process) — for example, integration tests against embedded databases whose `ApplicationContext` configuration only differs with regard to which bean definition profiles are active.

The root cause of such errors is the fact that Spring's `EmbeddedDatabaseFactory` (used internally by both the `<jdbc:embedded-database>` XML namespace element and the `EmbeddedDatabaseBuilder` for Java Config) will set the name of the embedded database to `"testdb"` if not otherwise specified. For the case of `<jdbc:embedded-database>`, the embedded database is typically assigned a name equal to the bean's `id` (i.e., often

something like `"dataSource")`). Thus, subsequent attempts to create an embedded database will not result in a new database. Instead, the same JDBC connection URL will be reused, and attempts to create a new embedded database will actually point to an existing embedded database created from the same configuration.

To address this common issue Spring Framework 4.2 provides support for generating *unique* names for embedded databases. To enable the use of generated names, use one of the following options.

- `EmbeddedDatabaseFactory.setGenerateUniqueDatabaseName()`
- `EmbeddedDatabaseBuilder.generateUniqueName()`
- `<jdbc:embedded-database generate-name="true" ... >`

19.8.7 Extending the embedded database support

Spring JDBC embedded database support can be extended in two ways:

- Implement `EmbeddedDatabaseConfigurer` to support a new embedded database type.
- Implement `DataSourceFactory` to support a new `DataSource` implementation, such as a connection pool to manage embedded database connections.

You are encouraged to contribute back extensions to the Spring community at jira.spring.io.

19.9 Initializing a DataSource

The `org.springframework.jdbc.datasource.init` package provides support for initializing an existing `DataSource`. The embedded database support provides one option for creating and initializing a `DataSource` for an application, but sometimes you need to initialize an instance running on a server somewhere.

19.9.1 Initializing a database using Spring XML

If you want to initialize a database and you can provide a reference to a `DataSource` bean, use the `initialize-database` tag in the `spring-jdbc` namespace:

```
<jdbc:initialize-database data-source="dataSource">
  <jdbc:script location="classpath:com/foo/sql/db-schema.sql"/>
  <jdbc:script location="classpath:com/foo/sql/db-test-data.sql"/>
</jdbc:initialize-database>
```

The example above executes the two scripts specified against the database: the first script creates a schema, and the second populates tables with a test data set. The script locations can also be patterns with wildcards in the usual ant style used for resources in Spring (e.g. `classpath*:com/foo/**/sql/*-data.sql`). If a pattern is used, the scripts are executed in lexical order of their URL or filename.

The default behavior of the database initializer is to unconditionally execute the scripts provided. This will not always be what you want, for instance, if you are executing the scripts against a database that already has test data in it. The likelihood of accidentally deleting data is reduced by following the common pattern (as shown above) of creating the tables first and then inserting the data—the first step will fail if the tables already exist.

However, to gain more control over the creation and deletion of existing data, the XML namespace provides a few additional options. The first is a flag to switch the initialization on and off. This can be set according to the environment (e.g. to pull a boolean value from system properties or an environment bean), for example:

```
<jdbc:initialize-database data-source="dataSource"
    enabled="#{systemProperties.INITIALIZE_DATABASE}">
    <jdbc:script location="..." />
</jdbc:initialize-database>
```

The second option to control what happens with existing data is to be more tolerant of failures. To this end you can control the ability of the initializer to ignore certain errors in the SQL it executes from the scripts, for example:

```
<jdbc:initialize-database data-source="dataSource" ignore-failures="DROPS">
    <jdbc:script location="..." />
</jdbc:initialize-database>
```

In this example we are saying we expect that sometimes the scripts will be executed against an empty database, and there are some `DROP` statements in the scripts which would therefore fail. So failed SQL `DROP` statements will be ignored, but other failures will cause an exception. This is useful if your SQL dialect doesn't support `DROP ... IF EXISTS` (or similar) but you want to unconditionally remove all test data before re-creating it. In that case the first script is usually a set of `DROP` statements, followed by a set of `CREATE` statements.

The `ignore-failures` option can be set to `NONE` (the default), `DROPS` (ignore failed drops), or `ALL` (ignore all failures).

Each statement should be separated by `;` or a new line if the `;` character is not present at all in the script. You can control that globally or script by script, for example:

```
<jdbc:initialize-database data-source="dataSource" separator="@">
    <jdbc:script location="classpath:com/foo/sql/db-schema.sql" separator=";" />
    <jdbc:script location="classpath:com/foo/sql/db-test-data-1.sql" />
    <jdbc:script location="classpath:com/foo/sql/db-test-data-2.sql" />
</jdbc:initialize-database>
```

In this example, the two `test-data` scripts use `@` as statement separator and only the `db-schema.sql` uses `;`. This configuration specifies that the default separator is `@` and override that default for the `db-schema` script.

If you need more control than you get from the XML namespace, you can simply use the `DataSourceInitializer` directly and define it as a component in your application.

Initialization of other components that depend on the database

A large class of applications can just use the database initializer with no further complications: those that do not use the database until after the Spring context has started. If your application is *not* one of those then you might need to read the rest of this section.

The database initializer depends on a `DataSource` instance and executes the scripts provided in its initialization callback (analogous to an `init-method` in an XML bean definition, a `@PostConstruct` method in a component, or the `afterPropertiesSet()` method in a component that implements `InitializingBean`). If other beans depend on the same data source and also use the data source in an initialization callback, then there might be a problem because the data has not yet been initialized. A common example of this is a cache that initializes eagerly and loads data from the database on application startup.

To get around this issue you have two options: change your cache initialization strategy to a later phase, or ensure that the database initializer is initialized first.

The first option might be easy if the application is in your control, and not otherwise. Some suggestions for how to implement this include:

- Make the cache initialize lazily on first usage, which improves application startup time.
- Have your cache or a separate component that initializes the cache implement `Lifecycle` or `SmartLifecycle`. When the application context starts up a `SmartLifecycle` can be automatically started if its `autoStartup` flag is set, and a `Lifecycle` can be started manually by calling `ConfigurableApplicationContext.start()` on the enclosing context.
- Use a Spring `ApplicationEvent` or similar custom observer mechanism to trigger the cache initialization. `ContextRefreshedEvent` is always published by the context when it is ready for use (after all beans have been initialized), so that is often a useful hook (this is how the `SmartLifecycle` works by default).

The second option can also be easy. Some suggestions on how to implement this include:

- Rely on the default behavior of the Spring `BeanFactory`, which is that beans are initialized in registration order. You can easily arrange that by adopting the common practice of a set of `<import/>` elements in XML configuration that order your application modules, and ensure that the database and database initialization are listed first.
- Separate the `DataSource` and the business components that use it, and control their startup order by putting them in separate `ApplicationContext` instances (e.g. the parent context contains the `DataSource`, and child context contains the business components). This structure is common in Spring web applications but can be more generally applied.

20. Object Relational Mapping (ORM) Data Access

20.1 Introduction to ORM with Spring

The Spring Framework supports integration with Hibernate, Java Persistence API (JPA) and Java Data Objects (JDO) for resource management, data access object (DAO) implementations, and transaction strategies. For example, for Hibernate there is first-class support with several convenient IoC features that address many typical Hibernate integration issues. You can configure all of the supported features for O/R (object relational) mapping tools through Dependency Injection. They can participate in Spring's resource and transaction management, and they comply with Spring's generic transaction and DAO exception hierarchies. The recommended integration style is to code DAOs against plain Hibernate, JPA, and JDO APIs. The older style of using Spring's DAO templates is no longer recommended; however, coverage of this style can be found in the [Section 39.1, "Classic ORM usage"](#) in the appendices.

Spring adds significant enhancements to the ORM layer of your choice when you create data access applications. You can leverage as much of the integration support as you wish, and you should compare this integration effort with the cost and risk of building a similar infrastructure in-house. You can use much of the ORM support as you would a library, regardless of technology, because everything is designed as a set of reusable JavaBeans. ORM in a Spring IoC container facilitates configuration and deployment. Thus most examples in this section show configuration inside a Spring container.

Benefits of using the Spring Framework to create your ORM DAOs include:

- *Easier testing.* Spring's IoC approach makes it easy to swap the implementations and configuration locations of Hibernate `SessionFactory` instances, JDBC `DataSource` instances, transaction managers, and mapped object implementations (if needed). This in turn makes it much easier to test each piece of persistence-related code in isolation.
- *Common data access exceptions.* Spring can wrap exceptions from your ORM tool, converting them from proprietary (potentially checked) exceptions to a common runtime `DataAccessException` hierarchy. This feature allows you to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches, throws, and exception declarations. You can still trap and handle exceptions as necessary. Remember that JDBC exceptions (including DB-specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.