



Pivotal®

Introduction to Spring JDBC

Using JdbcTemplate



Objectives

After completing this lesson, you should be able to

- Explain the problems with traditional JDBC
- Use and configure Spring's **JdbcTemplate**
- Execute queries using callbacks to handle result sets
- Handle Exceptions

Agenda

- **Problems with Traditional JDBC**
- Spring's **JdbcTemplate**
- Lab
- Optional Slides



Redundant, Error Prone Code in Traditional JDBC code

```
public List<Person> findByLastName(String lastName) {  
    List<Person> personList = new ArrayList<>();  
    String sql = "select first_name, age from PERSON where last_name=?";  
  
    try (Connection conn = dataSource.getConnection();  
         PreparedStatement ps = conn.prepareStatement(sql)) {  
        ps.setString(1, lastName);  
  
        try (ResultSet rs = ps.executeQuery()) {  
            while (rs.next()) {  
                personList.add(new Person(rs.getString("first_name"), ...));  
            }  
        }  
    } catch (SQLException e) {  
        /* ??? */  
    }  
  
    return personList;  
}
```

Redundant, Error Prone Code in Traditional JDBC code

```
public List<Person> findByLastName(String lastName) {  
    List<Person> personList = new ArrayList<>();  
    String sql = "select first_name, age from PERSON where last_name=?";  
  
    try (Connection conn = dataSource.getConnection();  
        PreparedStatement ps = conn.prepareStatement(sql)) {  
        ps.setString(1, lastName);  
  
        try (ResultSet rs = ps.executeQuery()) {  
            while (rs.next()) {  
                personList.add(new Person(rs.getString("first_name"), ...));  
            }  
        }  
    } catch (SQLException e) {  
        /* ??? */  
    }  
  
    return personList;  
}
```

The bold matters - the
rest is *boilerplate*

Redundant, Error Prone Code in Traditional JDBC code

```
public List<Person> findByLastName(String lastName) {  
    List<Person> personList = new ArrayList<>();  
    String sql = "select first_name, age from PERSON where last_name=?";  
  
    try (Connection conn = dataSource.getConnection();  
        PreparedStatement ps = conn.prepareStatement(sql)) {  
        ps.setString(1, lastName);  
  
        try (ResultSet rs = ps.executeQuery()) {  
            while (rs.next()) {  
                personList.add(new Person(rs.getString("first_name"), ...));  
            }  
        }  
    } catch (SQLException e) {  
        /* ??? */  
    }  
  
    return personList;  
}
```

How do you handle low-level
SQLException?

Agenda

- Problems with Traditional JDBC
- Spring's `JdbcTemplate`
 - Introduction
 - Basic Usage
 - Working with result sets
 - Exception handling
- Lab
- Optional Slides



Template Design Pattern

- Widely used and useful pattern
 - http://en.wikipedia.org/wiki/Template_method_pattern
- Define the outline or skeleton of an algorithm
 - Leave the details to specific implementations later
 - Hides away large amounts of *boilerplate* code
- Spring provides many template classes
 - `JdbcTemplate`, `JmsTemplate`
 - `RestTemplate`, `WebServiceTemplate` ...
 - Most hide low-level resource management

Spring's JdbcTemplate

- Greatly simplifies use of the JDBC API
 - Eliminates repetitive boilerplate code
 - Alleviates common causes of bugs
 - Handles **SQLExceptions** properly
- Without sacrificing power
 - Provides full access to the standard JDBC constructs



“Life is too short to write JDBC!”

- Rod Johnson co-founder of Spring

JdbcTemplate in a Nutshell

```
int count = jdbcTemplate.queryForObject(  
    "SELECT COUNT(*) FROM CUSTOMER", Integer.class);
```

- Acquisition of the connection
- Participation in the transaction
- Execution of the statement
- Processing of the result set
- Handling exceptions
- Release of the connection

**All handled
by Spring**

Using Callbacks

```
List<Customer> results = jdbcTemplate.query(someSql,  
    new RowMapper<Customer>() {  
        public Customer mapRow(ResultSet rs, int row) throws SQLException {  
            // map the current row to a Customer object  
        }  
    });
```

```
class JdbcTemplate {  
    public List<Customer> query(String sql, RowMapper rowMapper) {  
        try {  
            // acquire connection  
            // prepare statement  
            // execute statement  
            // for each row in the result set  
            results.add(rowMapper.mapRow(rs, rowNumber));  
            return results;  
        } catch (SQLException e) {  
            // convert to root cause exception  
        } finally { /* release connection */ }  
    }  
}
```

Callback
method

Creating a JdbcTemplate

- Requires a **DataSource**

```
JdbcTemplate template = new JdbcTemplate(dataSource);
```

- Create a template once and re-use it
 - Do not create one for each thread
 - Thread safe after construction
- Uses
 - *Anytime* JDBC is needed
 - In utility or test code
 - To clean up messy legacy code

Agenda

- Problems with Traditional JDBC
- Spring's **JdbcTemplate**
 - Introduction
 - **Basic Usage**
 - Working with Result Sets
 - Exception Handling
- Lab
- Optional Slides



Implementing a JDBC-based Repository

```
public class JdbcCustomerRepository implements CustomerRepository {  
  
    private JdbcTemplate jdbcTemplate;  
  
    public JdbcCustomerRepository(DataSource dataSource) {  
        this.jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
  
    public int getCustomerCount() {  
        String sql = "select count(*) from customer";  
        return jdbcTemplate.queryForObject(sql, Integer.class);  
    }  
}
```

No try / catch needed
(unchecked exceptions)

Querying with JdbcTemplate

- **JdbcTemplate** can query for
 - Simple types (int, long, String, Date, ...)
 - Generic Maps
 - Domain Objects

Query for Simple Java Types

- Query with no bind variables

```
public Date getOldest() {  
    String sql = "select min(dob) from PERSON";  
    return jdbcTemplate.queryForObject(sql, Date.class);  
}  
  
public long getPersonCount() {  
    String sql = "select count(*) from PERSON";  
    return jdbcTemplate.queryForObject(sql, Long.class);  
}
```



Older alternatives, `queryForInt()`, `queryForLong()`, deprecated and removed since Spring 4.2

Query With Bind Variables

- Can query using bind variables – ?
 - Note the use of a variable argument list

```
private JdbcTemplate jdbcTemplate;  
  
public int getCountOfNationalsOver(Nationality nationality, int age) {  
    String sql = "select count(*) from PERSON " +  
                "where age > ? and nationality = ?";  
  
    return jdbcTemplate.queryForObject  
        (sql, Integer.class, age, nationality.toString());  
}
```

Bind to first ?

Bind to second ?

Database Writes (1)

- Inserting a new row
 - Returns number of rows modified

```
public int insertPerson(Person person) {  
    return jdbcTemplate.update(  
        "insert into PERSON (first_name, last_name, age)" +  
        "values (?, ?, ?)",  
        person.getFirstName(),  
        person.getLastName(),  
        person.getAge());  
}
```

Database Writes (2)

- Updating an existing row

```
public int updateAge(Person person) {  
    return jdbcTemplate.update(  
        "update PERSON set age=? where id=?",  
        person.getAge(),  
        person.getId());  
}
```

Any non-SELECT SQL is run using update()

Agenda

- Problems with Traditional JDBC
- Spring's **JdbcTemplate**
 - Introduction
 - Basic Usage
 - **Working with Result Sets**
 - Exception Handling
- Lab
- Optional Slides



Generic Queries

- **JdbcTemplate** can return each row of a **ResultSet** as a **Map**
 - When expecting a single row
 - Use `queryForMap(..)`
 - When expecting multiple rows
 - Use `queryForList(..)`
- Useful for *ad hoc* reporting, testing use cases
 - The data fetched *does not need* mapping to a Java object



ad hoc – created or done for a particular purpose as necessary
– sometimes called “window-on-data” queries

Querying for Generic Maps (1)

- Query for a single row

```
public Map<String,Object> getPersonInfo(int id) {  
    String sql = "select * from PERSON where id=?";  
    return jdbcTemplate.queryForMap(sql, id);  
}
```

- Returns

Map { ID=1, FIRST_NAME="John", LAST_NAME="Doe" }

A Map of [Column Name | Field Value] pairs

Querying for Generic Maps (2)

- Query for multiple rows

```
public List<Map<String,Object>> getAllPersonInfo() {  
    String sql = "select * from PERSON";  
    return jdbcTemplate.queryForList(sql);  
}
```

- Returns

List {

A List of Maps of [Column Name | Field Value] pairs

0 - Map { ID=1, FIRST_NAME="John", LAST_NAME="Doe" }

1 - Map { ID=2, FIRST_NAME="Jane", LAST_NAME="Doe" }

2 - Map { ID=3, FIRST_NAME="Junior", LAST_NAME="Doe" }

}

Domain Object Queries

- Often it is useful to map relational data into domain objects
 - e.g. a `ResultSet` to an `Account`
- Spring's `JdbcTemplate` supports this using a callback approach
- You may prefer to use ORM for this
 - Need to decide between `JdbcTemplate` queries and JPA (or similar) mappings
 - Some tables may be too hard to map with JPA

RowMapper for mapping a row

- Spring provides a **RowMapper** interface for mapping a single row of a **ResultSet** to an object
 - Can be used for both single and multiple row queries
 - Parameterized to define its return-type

```
public interface RowMapper<T> {  
    T mapRow(ResultSet rs, int rowNum) throws SQLException;  
}
```

Querying for Domain Objects (1)

- Query for single row with `JdbcTemplate`

Returns a Domain object

```
public Person getPerson(int id) {  
    return jdbcTemplate.queryForObject(  
        "select first_name, last_name from PERSON where id=?",  
        (rs, rowNum) -> new Person(rs.getString("first_name"),  
                                     rs.getString("last_name")),  
        id);  
}
```

Define RowMapper
using *Lambda*



Alternative implementation using an explicit `RowMapper` subclass is shown at the end of this section.

Querying for Domain Objects (2)

- Query for multiple rows

Returns a List of Domain objects

```
public List<Person> getAllPersons() {  
    return jdbcTemplate.query(  
        "select first_name, last_name from PERSON",  
        (rs, rowNum) -> new Person(rs.getString("first_name"),  
                                     rs.getString("last_name"))  
    );  
}
```

Define RowMapper
using *Lambda*

ResultSetExtractor

- Spring provides a **ResultSetExtractor** interface for processing an entire **ResultSet** at once
 - *You* are responsible for iterating the **ResultSet**
 - *Example:* mapping entire **ResultSet** to a *single* object

```
public interface ResultSetExtractor<T> {  
    T extractData(ResultSet rs) throws SQLException,  
        DataAccessException;  
}
```



You may need this for the lab!

Using a ResultSetExtractor

Using a *lambda*

```
public class JdbcOrderRepository {  
    public Order findByConfirmationNumber(String number) {  
        // Execute an outer join between order and item tables  
        return jdbcTemplate.query(  
            "select...from order o, item i...conf_id = ?",  
            (ResultSetExtractor<Order>)(rs) -> {  
                Order order = null;  
                while (rs.next()) {  
                    if (order == null)  
                        order = new Order(rs.getLong("ID"), rs.getString("NAME"), ...);  
                    order.addItem(mapItem(rs));  
                }  
                return order;  
            },  
            number);  
    }  
}
```

Cast
needed

Summary of Callback Interfaces

- **RowMapper**
 - Best choice when *each* row of a **ResultSet** maps to a domain object
- **ResultSetExtractor**
 - Best choice when *multiple* rows of a **ResultSet** map to a *single* object
- **RowCallbackHandler**
 - Yet another handler that writes to alternative destinations

Agenda

- Problems with Traditional JDBC
- Spring's **JdbcTemplate**
 - Introduction
 - Basic Usage
 - Working with Result Sets
 - **Exception Handling**
- Lab
- Optional Slides



Exception Handling and Spring

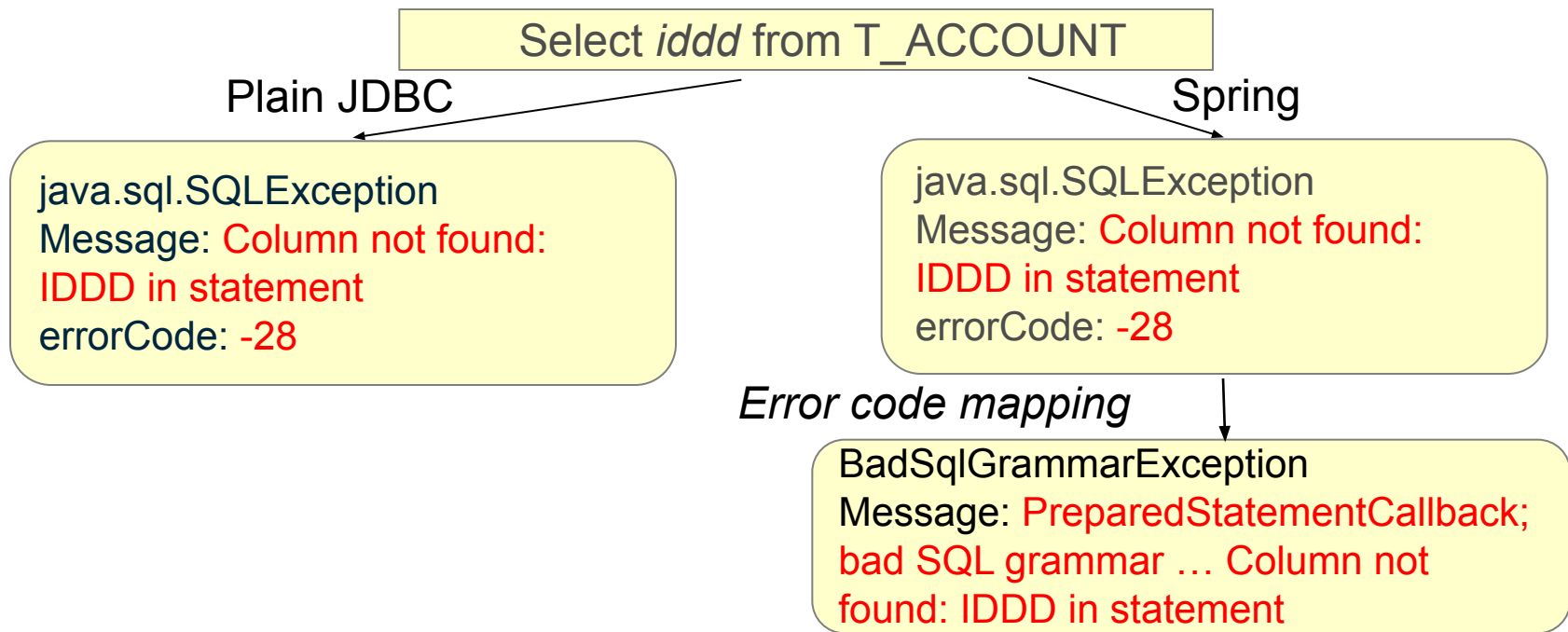
- Checked Exceptions
 - Force developers to handle errors
 - But if you can't handle it, must declare it
 - **Bad:** intermediate methods must declare exception(s) from *all* methods below
 - A form of tight-coupling
- Unchecked Exceptions
 - Can throw up the call hierarchy to the best place to handle it
 - **Good:** Methods in between don't know about it
 - Better in an Enterprise Application
 - **Spring always throws Runtime (unchecked) Exceptions**



Data Access Exceptions

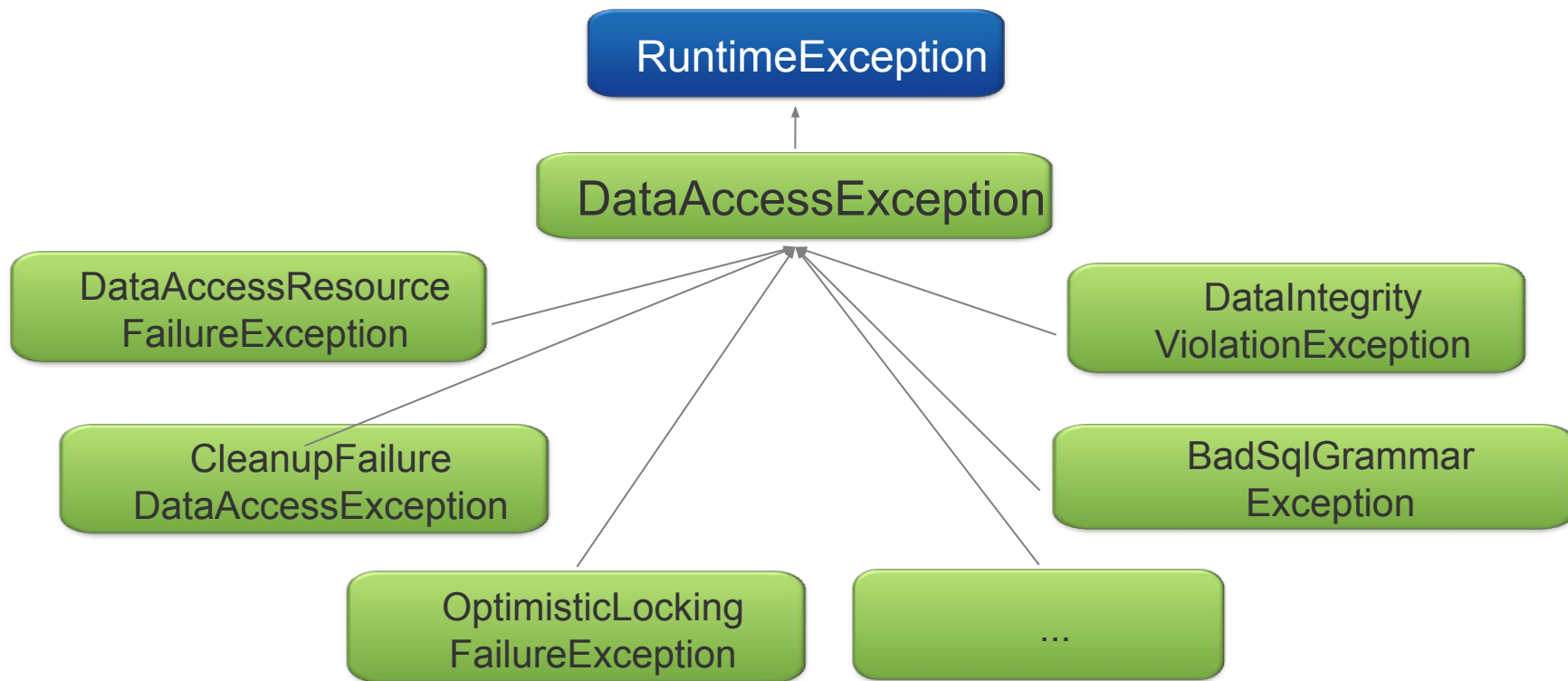
- **SQLException**
 - Too general – one exception for every database error
 - Calling class 'knows' you are using JDBC
 - Tight coupling
- Spring provides **DataAccessException** hierarchy
 - Hides whether you are using JPA, Hibernate, JDBC ...
 - Actually a hierarchy of sub-exceptions
 - Not just one exception for everything
 - Consistent across all supported Data Access technologies
 - Unchecked

Example: *BadSqlGrammarException*



<https://github.com/spring-projects/spring-framework/blob/master/spring-jdbc/src/main/resources/org/springframework/jdbc/support/sql-error-codes.xml>

Spring Data Access Exceptions



Summary

- JDBC is useful
 - But using JDBC API directly is tedious and error-prone
- **JdbcTemplate** simplifies data access and enforces consistency
 - DRY principle hides most of the JDBC
 - Many options for reading data
- **SQLExceptions** typically cannot be handled where thrown
 - Should not be *checked* Exceptions
 - Spring provides ***DataAccessException*** instead

A man with a beard and a woman are sitting at a desk, looking at a computer monitor. The man is pointing at the screen. The background is a blurred office environment.

Lab: JDBC Simplification with JdbcTemplate

**Lab project:
26-jdbc**

**Anticipated Lab time:
45 Minutes**

Optional Topics: Callbacks without Lambdas

Querying for Domain Objects (1)

- Query for single row with JdbcTemplate

```
public Person getPerson(int id) {  
    return jdbcTemplate.queryForObject(  
        "select first_name, last_name from PERSON where id=?",  
        new PersonMapper(), id);  
}
```

No need to cast

Maps rows to Person objects

Parameterizes return type

```
class PersonMapper implements RowMapper<Person> {  
    public Person mapRow(ResultSet rs, int rowNum) throws SQLException {  
        return new Person(rs.getString("first_name"),  
            rs.getString("last_name"));  
    }  
}
```

Querying for Domain Objects (2)

- Query for multiple rows

No need to cast


```
public List<Person> getAllPersons() {  
    return jdbcTemplate.query(  
        "select first_name, last_name from PERSON",  
        new PersonMapper());
```

Same row mapper can be used

```
class PersonMapper implements RowMapper<Person> {  
    public Person mapRow(ResultSet rs, int rowNum) throws SQLException {  
        return new Person(rs.getString("first_name"),  
            rs.getString("last_name"));  
    }  
}
```

ResultSetExtractor *without* a Lambda

```
public class JdbcOrderRepository {  
    public Order findByConfirmationNumber(String number) {  
        // execute an outer join between order and item tables  
        return jdbcTemplate.query(  
            "select...from order o, item i...conf_id = ?",  
            new OrderExtractor(), number);  
    }  
}
```



```
class OrderExtractor implements ResultSetExtractor<Order> {  
    public Order extractData(ResultSet rs) throws SQLException {  
        Order order = null;  
        while (rs.next()) {  
            if (order == null) {  
                order = new Order(rs.getLong("ID"), rs.getString("NAME"), ...);  
            }  
            order.addItem(mapItem(rs));  
        }  
        return order;  
    }  
}
```