Spring JDBC

OSCON 2005

JDBC Overview

- JDBC is standard Java API for operating on relational databases or other tabular data stores.
- JDBC tries to provide database independent SQL access to the data.
- JDBC uses:
  - Driver managed by DriverManager
  - Connection
  - Statement, PreparedStatement, CallableStatement
  - ResultSet to represent a table or set of rows
  - Types to map SQL data to Java classes or primitives

Spring JDBC Overview

- Spring manages all JDBC resources so you can concentrate on the data access logic instead.
- Spring translates SQLExceptions into a technology independent hierarchy of meaningful exceptions.
- Spring JDBC uses:
  - DataSource managed by Spring "Dependency Injection" container
  - JdbcTemplate provides a template based programming style with callbacks for low level database operations like row mapping etc.
  - RDMBS Objects for a more object oriented programming style for SqlQuery, SqlUpdate and StoredProcedure.

Spring JDBC
Division of Responsibilites

Task	Spring	You
Connection Management	√	
SQL		√
Statement Management	√	
ResultSet Management	√	
Row Data Retrieval		√
Parameter Declaration		√
Parameter Setting	√	
Transaction Management	√	

Spring JDBC Outline

- Working with a DAO (Data Access Object)
- Database connection provided via a DataSource
- Example of basic queries using JdbcTemplate
  - queryForInt / queryForObject
  - queryForList
  - queryForRowSet
- Exploring the MappingSqlQuery
- JdbcTemplate update and SqlUpdate solutions
- How to use the StoredProcedure class

Data Access Object Pattern

- Core J2EE Pattern - "*Use a Data Access Object (DAO) to abstract and encapsulate all access to the data source. The DAO manages the connection with the data source to obtain and store data.*"
- DAO interface defines methods used to retrieve and save data in the persistent data store.
- Allows for easy testing by providing mock DAOs
- Provides a clear and strongly typed persistence API
- Allows you to encapsulate custom query languages

The central player JdbcTemplate

- It simplifies the use of JDBC since it handles the creation and release of resources.
- Executes the core JDBC workflow like statement creation and execution, leaving application code to provide SQL and extract results.
- Catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy.

The central player JdbcTemplate

- Many convenience methods like queryForInt, queryForList, execute etc.
- Code using this class only need to implement certain callback interfaces if needed:
  - PreparedStatementCreator callback interface creates a prepared statement
  - CallableStatementCreator callback interface creates a callable statement
  - RowCallbackHandler interface extracts values from each row of a ResultSet
  - RowMapper maps values for each individual row into an object.

Very simple example comparing JDBC vs. Spring

- We want to know the number of beer brands we currently have in our database.
- The query:

  “`select count(*) from beers`”

```
public class JdbcDao {
    private Logger logger = Logger.getLogger("SpringOne");

    public int getBeerCount() {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;
        int count = 0;
        Properties properties = new Properties();
        try {
            properties.load(new FileInputStream("jdbc.properties"));
        } catch (IOException e) {
            throw new MyDataAccessException("I/O Error", e);
        }
        try {
            Class.forName(properties.getProperty("driverClassName"));
            conn = DriverManager.getConnection(properties.getProperty("url"), properties);
            stmt = conn.createStatement();
            rs = stmt.executeQuery("select count(*) from beers");
            if (rs.next()) {
                count = rs.getInt(1);
            }
        }
        catch (ClassNotFoundException e) {
            throw new MyDataAccessException("JDBC Error", e);
        }
        catch (SQLException se) {
            throw new MyDataAccessException("JDBC Error", se);
        }
        finally {
            if (rs != null) {
                try {
                    rs.close();
                }
                catch (SQLException ignore) {
                }
            }
            if (stmt != null) {
                try {
                    stmt.close();
                }
                catch (SQLException ignore) {
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                }
                catch (SQLException ignore) {
                }
            }
        }
        return count;
    }
}
```

Using straight JDBC code

Using Spring

```
public class SpringDao {
    private Logger logger = Logger.getLogger("SpringOne");

    public int getBeerCount() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        int count = 0;
        Properties properties = new Properties();
        try {
            properties.load(new FileInputStream("jdbc.properties"));
        } catch (IOException e) {
            logger.severe(e.toString());
        }
        dataSource.setConnectionProperties(properties);
        dataSource.setDriverClassName(properties.getProperty("driverClassName"));
        dataSource.setUrl(properties.getProperty("url"));

        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        count = jdbcTemplate.queryForInt("select count(*) from beers");
        return count;
    }
}
```

Using Spring with Dependency Injection

```
public class SpringDao {
    private DataSource dataSource;
    private JdbcTemplate jdbcTemplate;

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

    public int getBeerCount() {
        int count = jdbcTemplate.queryForInt("select count(*) from beers");
        return count;
    }
}
```

Using Spring with JdbcDaoSupport

```
public class SpringDao extends JdbcDaoSupport {

    public int getBeerCount() {
        int count = getJdbcTemplate().queryForInt("select count(*) from beers");
        return count;
    }
}
```

Example
Retreiving a specific object

How to retrieve a specific instance of a Beer:

```
package com.springcheers.model;

import java.math.BigDecimal;

public class Beer {
    private Long id;
    private String brand;
    private BigDecimal price;

...

// getters and setters

...

}
```

Example -- Step-by-step

1. DataAccessObject -- extend JdbcDaoSupport which provides a preconfigured JdbcTemplate
2. Test class extending *AbstractTransactionalDataSourceSpringContextTests*
3. applicationContext/dataAccessContext with TransactionManager and DataSource to be used
4. Test data loaded with DBUnit
5. All jar files needed on the Classpath
6. Run the tests

Example -- Step-by-step

✓ Step 1 -- DAO

```
public class BeerDaoImpl extends JdbcDaoSupport implements BeerDao {

    public Beer getBeer(Long id) {
        Beer beer =
          (Beer)getJdbcTemplate().queryForObject(
                  "select id, brand, price from beers where id = ?",
                    new Object[] {id},
                    new BeerRowMapper());
        return beer;
    }
  ...
    private static class BeerRowMapper implements RowMapper {
        public Object mapRow(ResultSet resultSet,int i)
                throws SQLException {
            Beer b = new DomesticBeer();
            b.setId(new Long(resultSet.getLong("id")));
            b.setBrand(resultSet.getString("brand"));
            b.setPrice(resultSet.getBigDecimal("price"));
            return b;
        }
    }
}
```

SQL

Parameter

RowMapper

Example -- Step-by-step

✓ Step 2a -- Tests -- AbstractDaoTests

```
public abstract class AbstractDaoTests extends
                          AbstractTransactionalDataSourceSpringContextTests {

    protected  String[] getConfigLocations() {
        return new String[] {"WEB-INF/applicationContext.xml",
                             "WEB-INF/dataAccessContext.xml"};
    }

    public void onSetUpInTransaction() throws Exception {
        // Load test data using DBUnit
        DataSource ds = jdbcTemplate.getDataSource();
        Connection con = DataSourceUtils.getConnection(ds);
        IDatabaseConnection dbUnitCon = new DatabaseConnection(con);
        IDataSet dataSet = new FlatXmlDataSet(new FileInputStream("db/springcheers.xml"));
        try {
            DatabaseOperation.CLEAN_INSERT.execute(dbUnitCon, dataSet);
        }
        finally {
            DataSourceUtils.releaseConnection(con, ds);
        }
    }
}
```

Example -- Step-by-step

✓ Step 2b -- Tests -- BeerDaoTests

```
public class BeerDaoTests extends AbstractDaoTests {
    private BeerDao beerDao = null;

    public void setBeerDao(BeerDao beerDao) {
        this.beerDao = beerDao;
    }
...
    public void testGetBeer() {
        Beer b1 = beerDao.getBeer(new Long(4));
        assertEquals("correct beer id", 4L, b1.getId().longValue());
        assertEquals("correct beer brand", "Bass", b1.getBrand());
        assertEquals("correct beer price", new BigDecimal("24.55"), b1.getPrice());

    }
...
}
```

Example -- Step-by-step

✓ Step 3a -- applicationContext.xml

```
<beans>
    <bean id="propertyConfigurer"
       class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
         <property name="location" value="WEB-INF/jdbc.properties"/>
    </bean>

    <bean id="beerDao" class="com.springcheers.dao.jdbc.BeerDaoImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>
</beans>
```

Example -- Step-by-step

- ✓ Step 3b -- dataAccessContext.xml

```
<beans>
    <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>
    <!-- Transaction manager for a single JDBC DataSource -->
    <!-- (see dataAccessContext-jta.xml for an alternative) -->
    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource"><ref local="dataSource"/></property>
    </bean>
</beans>
```

Example -- Step-by-step

✓ Step 4 -- Test Data (DBUnit dataset)

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <beers id="1" brand="Budweiser" price="19.50"/>
  <beers id="2" brand="Heineken" price="23.55"/>
  <beers id="3" brand="Spring Ale" price="25.95"/>
  <beers id="4" brand="Bass" price="24.55"/>
  <customers id="1" name="Rod&apos;s Tavern"/>
  <customers id="2" name="Juergen Brau"/>
  <customers id="3" name="Colin&apos;s Corner"/>
  <customers id="4" name="Beer Garden"/>
  <orders id="1" status="shipped" ref_customer_id="2"/>
</dataset>
```

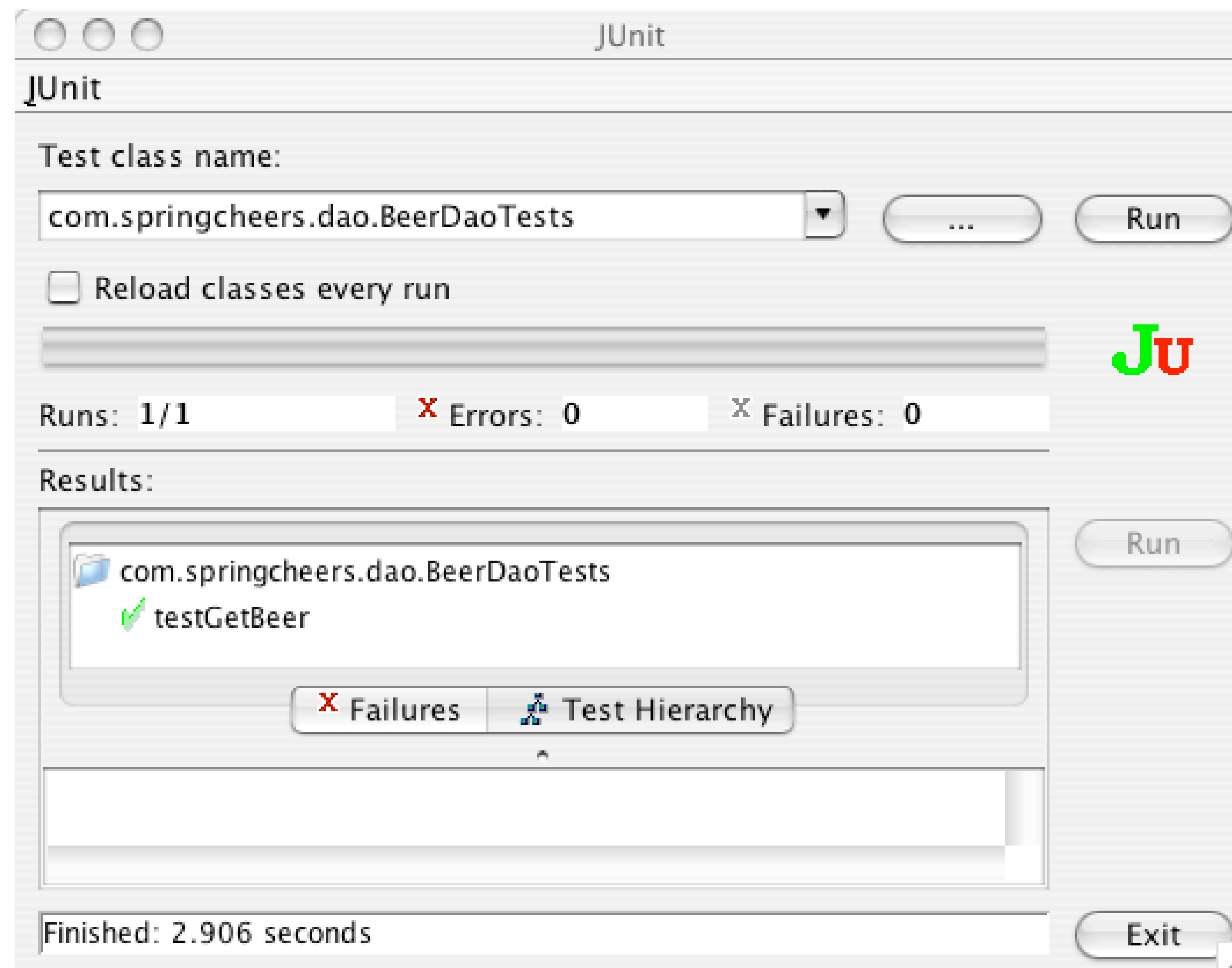
Example -- Step-by-step

✓ Step 5 -- jar files/libraries

```
Required for Spring
spring.jar
commons-logging.jar
“jdbc-driver”

Required for Commons DBCP:
commons-collections.jar
commons-dbcp.jar
commons-pool.jar

Required for Testing:
dbunit-2.1.jar
junit.jar
spring-mock.jar
```

Example -- Step-by-step

✓ Step 6 -- Run Test!



MappingSqlQuery

```
public class BeerDaoImpl extends JdbcDaoSupport implements BeerDao {
    private BeerQuery beerQuery;

    public void initDao() {
        beerQuery = new BeerQuery(getDataSource());
    }
    public Beer getBeer(Long id) {
        return beerQuery.find(id);
    }
    public class BeerQuery extends MappingSqlQuery {
        private static final String sql = "select id, brand, price from beers where id = ?";

        public BeerQuery(DataSource dataSource) {
            super(dataSource, sql);
            declareParameter(new SqlParameter("id", Types.INTEGER));
            compile();
        }
        public Beer find(Long id) {
            return (Beer) findObject(new Object[] {id});
        }
        protected Object mapRow(ResultSet resultSet,int i) throws SQLException {
            Beer b = new DomesticBeer();
            b.setId(new Long(resultSet.getLong("id")));
            b.setBrand(resultSet.getString("brand"));
            b.setPrice(resultSet.getBigDecimal("price"));
            return b;
        }
    }
}
```

JdbcTemplate update

```
public class BeerDaoImpl extends JdbcDaoSupport implements BeerDao {

  ...

    public void updateBeer(Beer beer) {
        getJdbcTemplate().update(
            "update beers set brand = ?, price = ? where id = ?",
            new Object[3] {beer.getBrand(),
                           beer.getPrice(),
                           beer.getId()});
    }

  ...

}
```

SQL

Parameters

SqlUpdate

```
public class BeerDaoImpl extends JdbcDaoSupport implements BeerDao {
    private BeerUpdate beerUpdate;

    public void initDao() {
        beerUpdate = new BeerUpdate(getDataSource());
    }
    public void updateBeer(Beer beer) {
        beerUpdate.update(beer);
    }

    public class BeerUpdate extends SqlUpdate {
        private static final String sql =
            "update beers set brand = ?, price = ? where id = ?";

        public BeerUpdate(DataSource dataSource) {
            super(dataSource, sql);
            declareParameter(new SqlParameter("brand", Types.VARCHAR));
            declareParameter(new SqlParameter("price", Types.DECIMAL));
            declareParameter(new SqlParameter("id", Types.INTEGER));
            compile();
        }
        public void update(Beer beer) {
            Object[] params = new Object[3];
            params[0] = beer.getBrand();
            params[1] = beer.getPrice();
            params[2] = beer.getId();
            update(params);
        }
    }
}
```

StoredProcedure

```
create or replace function delete_order(integer)
  returns integer
as '
  delete from orders where id = $1;
  select 0;
' language 'sql';
```

StoredProcedure

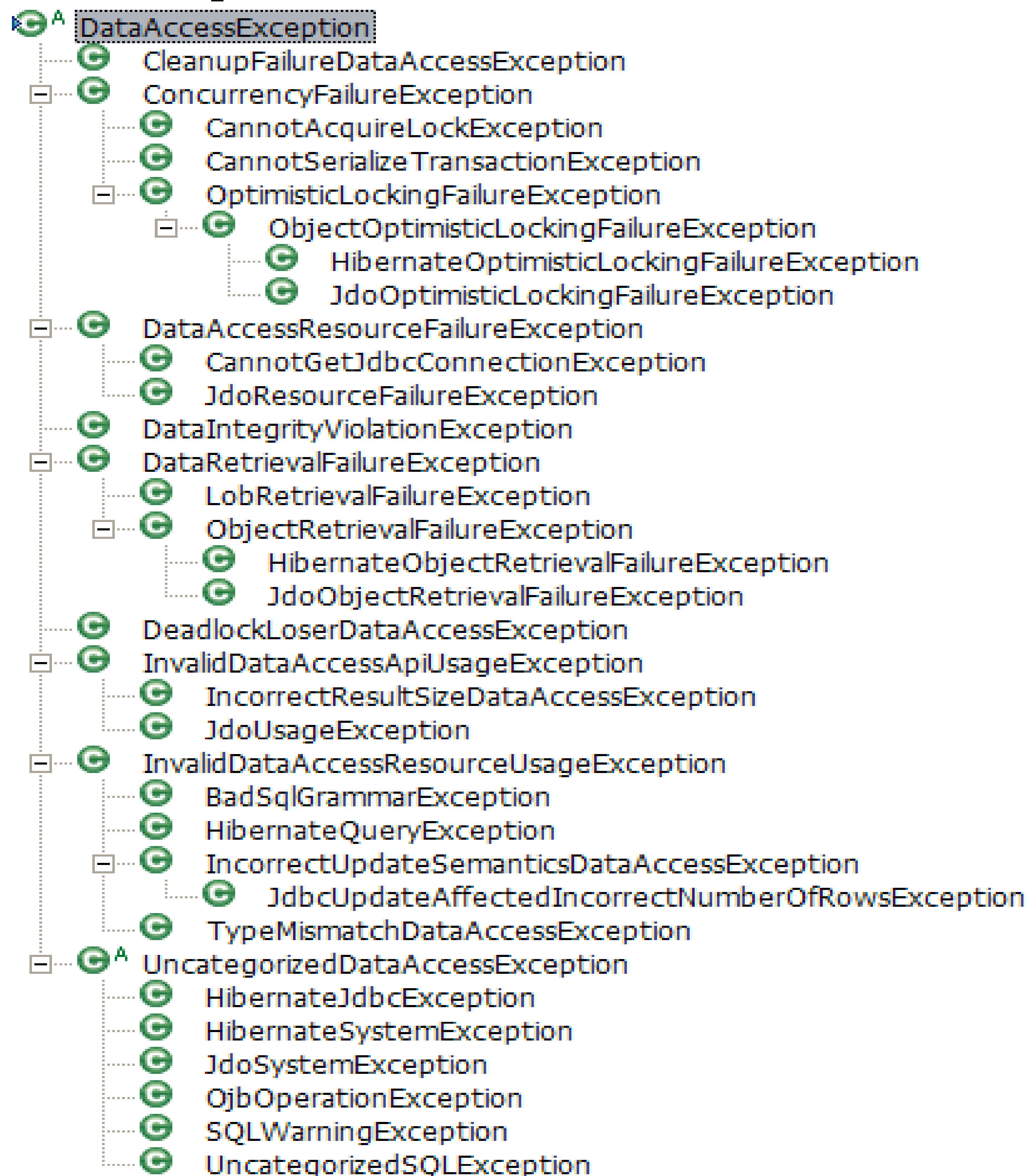
```
public class OrderDaoImpl extends JdbcDaoSupport implements OrderDao {
    private DeleteOrderProc delete;

    public void initDao() {
        delete = new DeleteOrderProc(getDataSource());
    }
    public void deleteOrder(Long id) {
        delete.execute(id);
    }

    public class DeleteOrderProc extends StoredProcedure {
        private static final String sql = "delete_order";

        public DeleteOrderProc(DataSource dataSource) {
            super(dataSource, sql);
            declareParameter(new SqlParameter("id", Types.INTEGER));
            compile();
        }

        public void execute(Long id) {
            Map params = new HashMap(1);
            params.put("id", id);
            execute(params);
        }
    }
}
```

Spring

Additional Features

- Spring's exception translations. Default is stored in sql-error-codes.xml. Can be customized with additional codes and custom exceptions.
- BLOB/CLOB support makes it easy to work in similar manner across databases.
- Extraction of native jdbc connection for vendor specific features.
- RDMS specific object types supported as well.

Exception Translation



Exception Translation

```xml
<bean id="MyDatabase" class="org.springframework.jdbc.support.SQLErrorCodes">
  <property name="badSqlGrammarCodes">
    <value>11,24,33</value>
  </property>
  <property name="dataIntegrityViolationCodes">
     <value>1,12,17,22</value>
  </property>
  <property name="customTranslations">
    <list>
      <bean
        class="org.springframework.jdbc.support.CustomSQLErrorCodesTranslation">
        <property name="errorCodes">
          <value>942</value></property>
        <property name="exceptionClass">
          <value>com.mycompany.test.MyCustomException</value>
        </property>
      </bean>
    </list>
  </property>
</bean>
```

Batch Processing

```
import java.sql.Types;
import java.util.Iterator;
import java.util.List;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.core.support.JdbcDaoSupport;
import org.springframework.jdbc.object.BatchSqlUpdate;
import org.springframework.jdbc.object.SqlUpdate;

public class DataStuff extends JdbcDaoSupport {
    private PersonQuery personQuery;
    private static final String insertSql = "insert into mytest (id, name) values(?, ?)";

    public List getNames() {
        return personQuery.execute(0);
    }

    public void insertNames(List names) {
        BatchSqlUpdate nameInsert = new BatchSqlUpdate(getDataSource(), insertSql);
        nameInsert.declareParameter(new SqlParameter("id", Types.INTEGER));
        nameInsert.declareParameter(new SqlParameter("newName", Types.VARCHAR));
        nameInsert.compile();
        Iterator nameIter = names.iterator();
        while (nameIter.hasNext()) {
            nameInsert.update((Object[]) nameIter.next());
        }
        nameInsert.flush();
    }

    public void initDao() {
        this.personQuery = new PersonQuery(getDataSource());
    }

}
```

Batch Processing

```
import java.util.ArrayList;
import java.util.List;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class JustStuff {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx;

        ctx = new ClassPathXmlApplicationContext("dataAccessContext.xml");
        DataStuff test = (DataStuff)ctx.getBean("dataStuff");

        System.out.println("Before: " + test.getNames());

        List names = new ArrayList();
        names.add(new Object[] {new Long(4), "Foo"});
        names.add(new Object[] {new Long(5), "Bar"});
        test.insertNames(names);

        System.out.println("After: " + test.getNames());

        ctx.close();
        System.out.println("OK!");
    }
}
```

Batch Processing

- Create a new instance of BatchSqlUpdate and declare the parameters
  - NOTE: this class is NOT thread safe.
- Pass in a List of arrays containing the parameters
- Call flush() when the batch is done