JDBC

Overview

- JDBC – Java Database Connectivity
- Main API for working with RDBMS in Java
- All major DB vendors support JDBC (Oracle, DB2, MS SQL, MySQL, PostgreSQL)
- Latest version is JDBC 4.3

JDBC API and Driver

- JDBC consists of JDBC API and JDBC Driver
- JDBC API contains all abstractions for DB access
- Application developer uses these abstractions
- JDBC API is a part of JDK
  - JDBC 4.1 is included in Java 7
  - JDBC 4.2 is included in Java 8
- JDBC Driver provides vendor specific implementation of JDBC API
- Separation between API and Driver gives ability to switch RDBMS vendor without changing application code (in theory)

JDBC API – main interfaces

- DataSource
- Connection
- Statement
- PreparedStatement
- CallableStatement
- ResultSet

DataSource

- Factory for connections to RBDMS (getConnection methods)
- Implementation of this interface is provided by JDBC driver
- Configuration of a connection by URL
  (e.g., jdbc:mysql://localhost/mydb) or set of properties
- Creates new connection for each getConnection method call or returns a connection from a connection pool if connection pool is used

Connection pooling

- Creating a connection is expensive (network roundtrip, resource allocation)
- Connection pooling: creating several connections in advance and reusing them
- All connection pools implement DataSource interface

Connection

- A connection (session) with a specific DB
- All operations are performed in the context of a connection
- Provides API for getting DB metadata
- Provides API for configuration of Isolation level and transaction boundaries
- Creates statement objects for query execution
  - Statement
  - PreparedStatement
  - CallableStatement
- Don't forget to close a connection

Statement

- The object used for executing a static SQL statement and returning the results it produces
- Can be used for execution of SQL statements (DML, DDL and queries) as a single query or as a batch
- Don't forget to close

PreparedStatement

- Extends Statement interface
- An object that represents a precompiled SQL statement
- This object can then be used to efficiently execute this statement multiple times
- Provides API for setting query parameters of different types
- Placeholder for a parameter value '?'

CallableStatement

- Extends PreparedStatement interface
- The interface used to execute SQL stored procedures
- In addition to input parameters provides API for getting OUT parameters of DB stored procedures

ResultSet

- Object for getting results of a query
- Like 'Iterator' pattern
- You can iterate through it only once and only from the first row to the last row
- Don't forget to close

Example

Spring JDBC support

- JDBC was the only way to interact with RDBMS
- Now JDBC API is very low level and other libraries built on-top of JDBC are used
  - Every JDBC operation throws an exception
  - Don't forget to close
- Spring provides a thin façade on top of JDBC API
  - Implementation of Façade design pattern
  - Common operation can be performed by a single line of code
  - You always have access to underlying JDBC API

Spring JDBC support

- Takes DataSource instance as a parameter
- JdbcOperations
  - Implemented by JdbcTemplate
  - ? - placeholder for a parameter
- NamedParameterJdbcOperations
  - Implemented by NamedParameterJdbcTemplate
  - Allows using of named parameters rather than the traditional '?' placeholders

Spring JDBC support

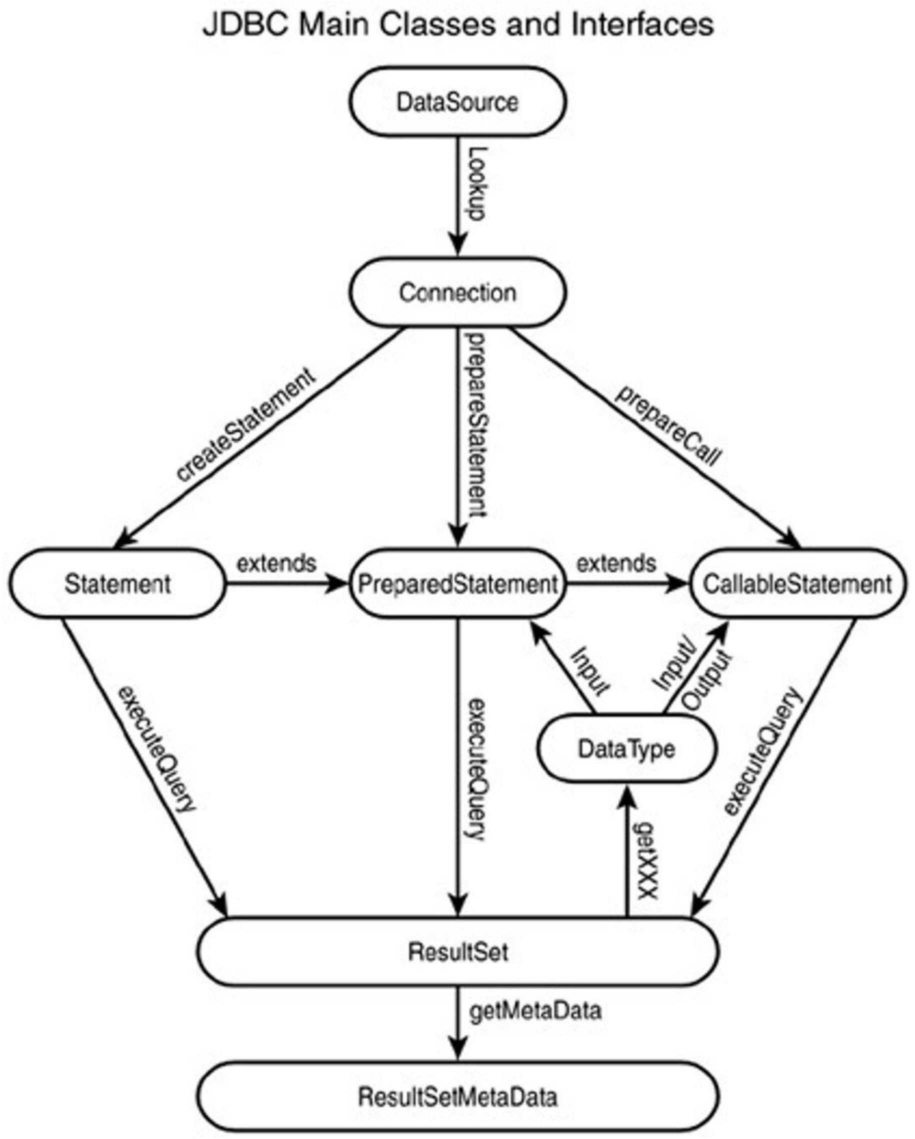
```
public void insert(Book book) {
    String sql = "INSERT INTO BOOK (TITLE, DATE_RELEASE) VALUES (?, ?)";
    PreparedStatement statement;
    try {
        Connection connection = openConnection();
        statement = connection.prepareStatement(sql);
        statement.setString(1, book.getTitle());
        statement.setDate(2, new java.sql.Date(book.getDateRelease().getTime()));
        statement.executeUpdate();
        statement.close();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        closeConnection();
    }
}
```

Spring JDBC support

```
public void insert(Book book) {
   String sql = "INSERT INTO BOOK (TITLE, DATE_RELEASE) VALUES (?, ?)";
   PreparedStatement statement;
   try {
      Connection connection = openConnection();
      statement = connection.prepareStatement(sql);
      statement.setString(1, book.getTitle());
      statement.setDate(2, new java.sql.Date(book.getDateRelease().getTime()));
      statement.executeUpdate();
      statement.close();
   } catch (SQLException e) {
      e.printStackTrace();
   } finally {
      closeConnection();
   }
}
```

Spring JDBC support

```
public void insert(Book book) {
   String sql = "INSERT INTO BOOK (TITLE, DATE_RELEASE) VALUES (?, ?)";
   PreparedStatement statement;
   try {
      Connection connection = openConnection();
      statement = connection.prepareStatement(sql);
      statement.setString(1, book.getTitle());
      statement.setDate(2, new java.sql.Date(book.getDateRelease().getTime()));
      statement.executeUpdate();
      statement.close();
   } catch (SQLException e) {
      e.printStackTrace();
   } finally {
      closeConnection();
   }
}
```



Spring JDBC support

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

@Override
public void insert(Book book) {
   String sql = "INSERT INTO BOOK (TITLE, DATE_RELEASE) VALUES (?, ?)";
   jdbcTemplate.update(sql, new Object[] { book.getTitle(),
                new java.sql.Date(book.getDateRelease().getTime()) });
}
```

Spring JDBC support

Without Spring:

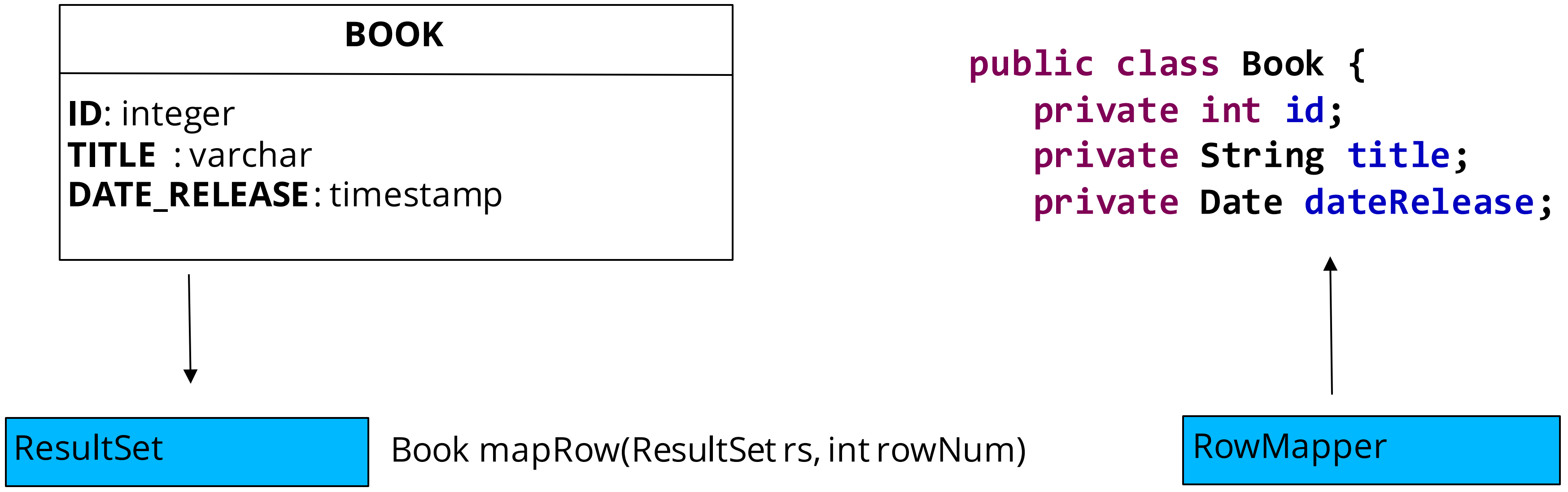
- Define connection parameters;
- Open the connection;
- Specify the statement;
- Prepare and execute the statement;
- Iteration through the results;
- Do the work for each iteration;
- Process any exception;
- Handle transactions;
- Close the connection;

With Spring support:

- Define connection parameters by creating a bean
- Specify the statement;
- Do the work for each iteration;

RowMapper

Mapping data from DB to the object model



BOOK

ID: integer
TITLE : varchar
DATE_RELEASE: timestamp

```
public class Book {
    private int id;
    private String title;
    private Date dateRelease;
```

Book mapRow(ResultSet rs, int rowNum)

RowMapper is doing mapping of **ResultSet** to the certain objects

RowMapper

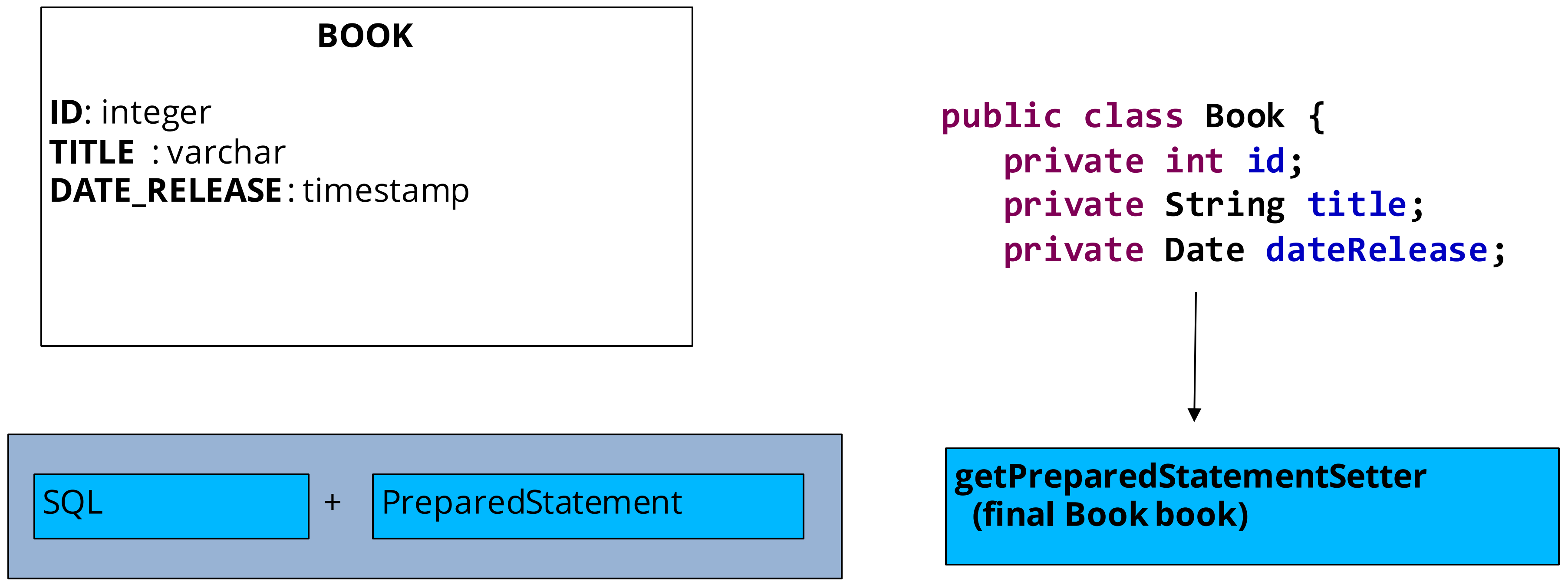
```
private RowMapper<Book> rowMapper = new RowMapper<Book>() {
    public Book mapRow(ResultSet resultSet, int rowNum) throws SQLException {
        Book book = new Book();
        book.setId(resultSet.getInt("id"));
        book.setTitle(resultSet.getString("title"));
        book.setDateRelease(resultSet.getDate("date_release"));
        return book;
    }
};

@Override
public Book getById(int id) {
    String sql = "SELECT * FROM BOOK WHERE ID = ?";
    return jdbcTemplate.queryForObject(sql, rowMapper, id);
}

@Override
public List<Book> getAll() {
    return jdbcTemplate.query("SELECT * FROM BOOK", rowMapper);
}
```

PreparedStatementSetter

Mapping data from object model to SQL



BOOK

ID: integer
TITLE : varchar
DATE_RELEASE : timestamp

```
public class Book {
   private int id;
   private String title;
   private Date dateRelease;
```

PreparedStatementSetter is doing mapping of object to SQL request

PreparedStatementSetter

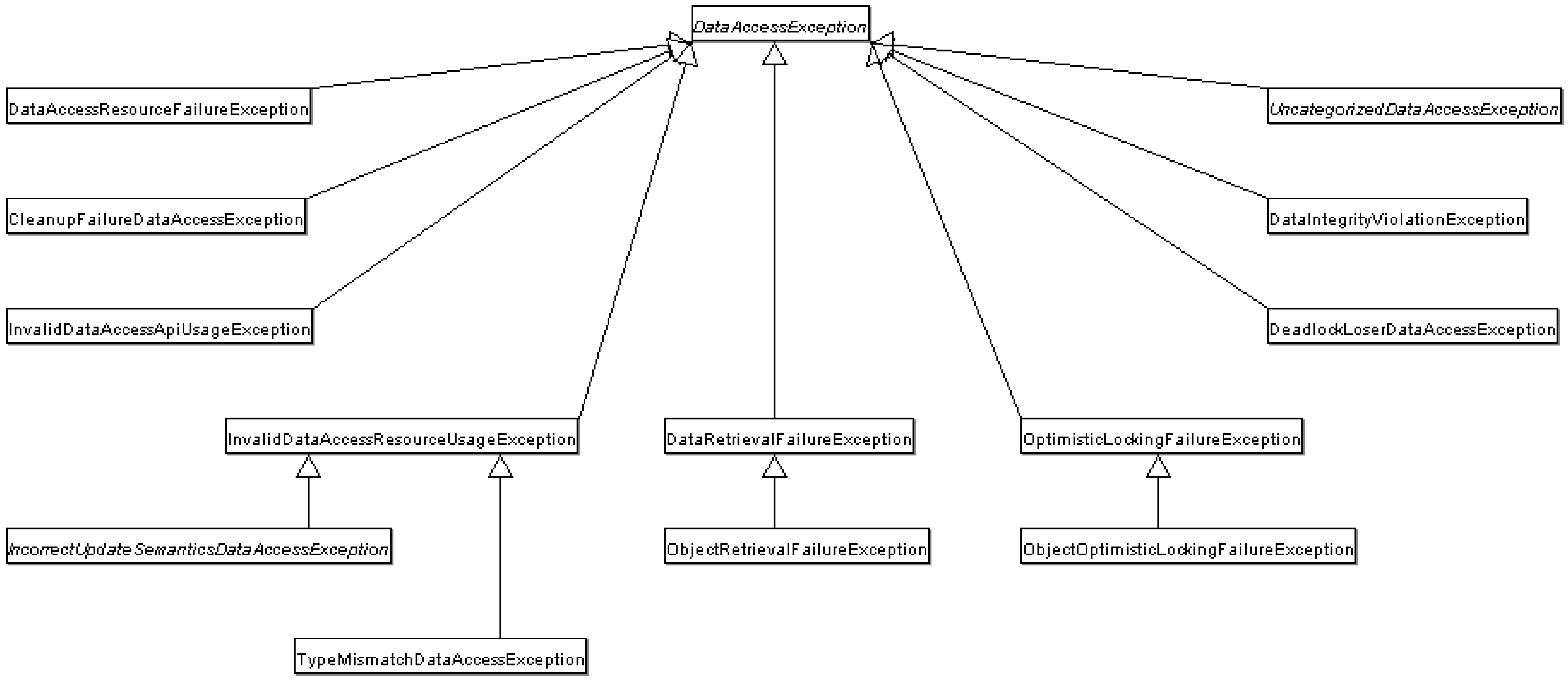
```
private PreparedStatementSetter getPreparedStatementSetter(final Book book) {
    return new PreparedStatementSetter() {
        public void setValues(PreparedStatement preparedStatement) throws SQLException {
            int i = 0;
            preparedStatement.setString(++i, book.getTitle());
            preparedStatement.setDate(++i,
                    new java.sql.Date(book.getDateRelease().getTime()));
        }
    };
}

public void insert(Book book) {
    String sql = "INSERT INTO BOOK (TITLE, DATE_RELEASE) VALUES (?, ?)";
    jdbcTemplate.update(sql, getPreparedStatementSetter(book));
}
```

Spring JDBC support

- Spring provides a convenient translation from technology-specific exceptions like SQLException to its own exception class hierarchy with the DataAccessException as the root exception.
- These exceptions wrap the original exception so there is never any risk that one might lose any information as to what might have gone wrong
- DataAccessException is a runtime exception

Spring JDBC – exceptions


DataAccessException
DataAccessResourceFailureException
CleanupFailureDataAccessException
InvalidDataAccessApiUsageException
InvalidDataAccessResourceUsageException
DataRetrievalFailureException
OptimisticLockingFailureException
UncategorizedDataAccessException
DataIntegrityViolationException
DeadlockLoserDataAccessException
IncorrectUpdateSemanticsDataAccessException
TypeMismatchDataAccessException
ObjectRetrievalFailureException
ObjectOptimisticLockingFailureException