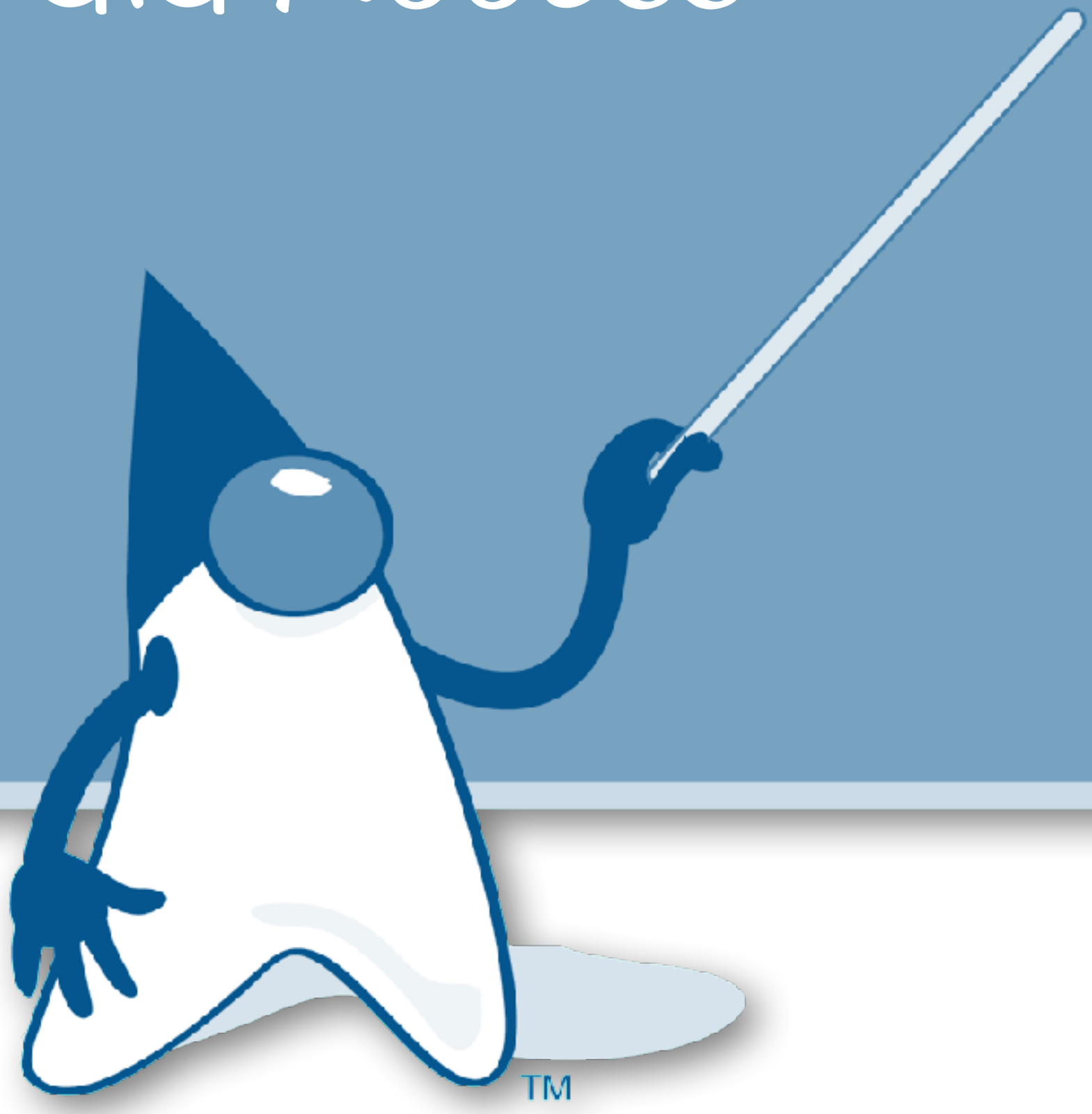
Info Support
Solid Innovator

Data Access



TM

Data Access overview

- Declarative transaction management
- Exception handling
- Simplified JDBC support
- ORM integration

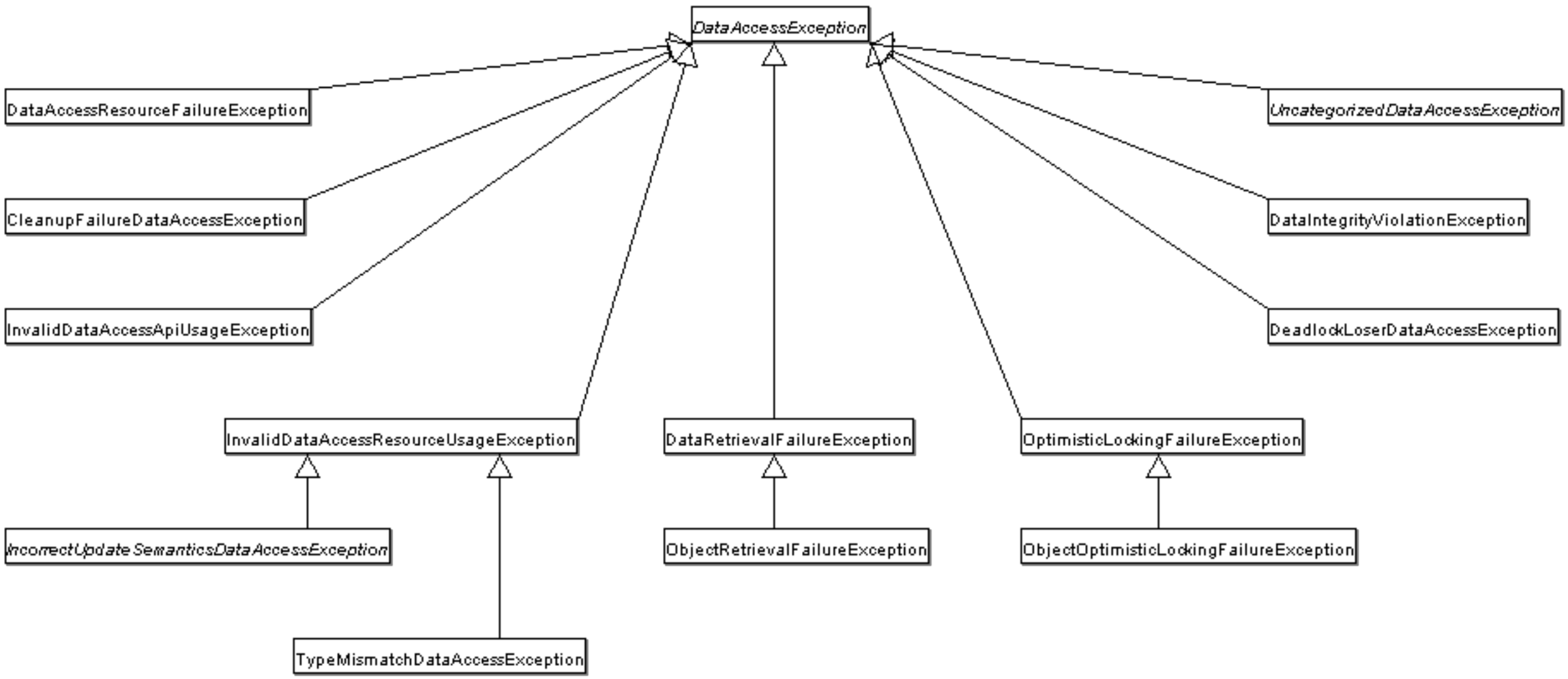
Things wrong with JDBC

- A lot of mandatory checked exception handling
  - most are not recoverable
- SQLException is very generic
  - have to parse the sql error-code yourself
- Clumsy API
  - simple things require a lot of code

Spring JDBC support

- Exception wrapping
  - all exceptions are translated to unchecked exceptions
- Consistent exceptions
  - native sql error codes are translated to consistent exception types
- Simplified APIs

Consistent Exception Hierarchy



Configuring a connection

Create a data source

```
<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=""/>
</bean>

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

Lookup a data source

```
<jee:jndi-lookup jndi-name="myJndiDS"/>
```

Simple JDBC Template

Creating a SimpleJdbcTemplate

```
private SimpleJdbcTemplate jdbcTemplate;

@Autowired
public void setDataSource(DataSource ds) {
    jdbcTemplate = new SimpleJdbcTemplate(ds);
}
```

Queries and mapping

```
public List<Movie> listMovies() {
    return jdbcTemplate.query("select * from movies",
            new RowMapper<Movie>() {

        @Override
        public Movie mapRow(ResultSet rs, int rowNum)
                throws SQLException {
            Movie movie = new Movie();
            movie.setTitle(rs.getString("title"));
            movie.setGenre(rs.getString("genre"));
            movie.setReleaseDate(rs.getDate("releaseDate"));

            return movie;
        }
    });
}
```

Called for each row

QueryForObject

```
public Movie findMovie(int id) {
    return jdbcTemplate.queryForObject("select * from movies where id = ?",
            new RowMapper<Movie>() {
                @Override
                public Movie mapRow(ResultSet rs, int i) throws SQLException {
                    Movie movie = new Movie();
                    movie.setTitle(rs.getString("title"));
                    movie.setGenre(rs.getString("genre"));
                    movie.setReleaseDate(rs.getDate("releaseDate"));

                    return movie;
                }
            }, id);
}
```

QueryFor...

- Convenience methods for several types

```
int nrOfRows = jdbcTemplate.queryForInt("select count(*) from movies");
```

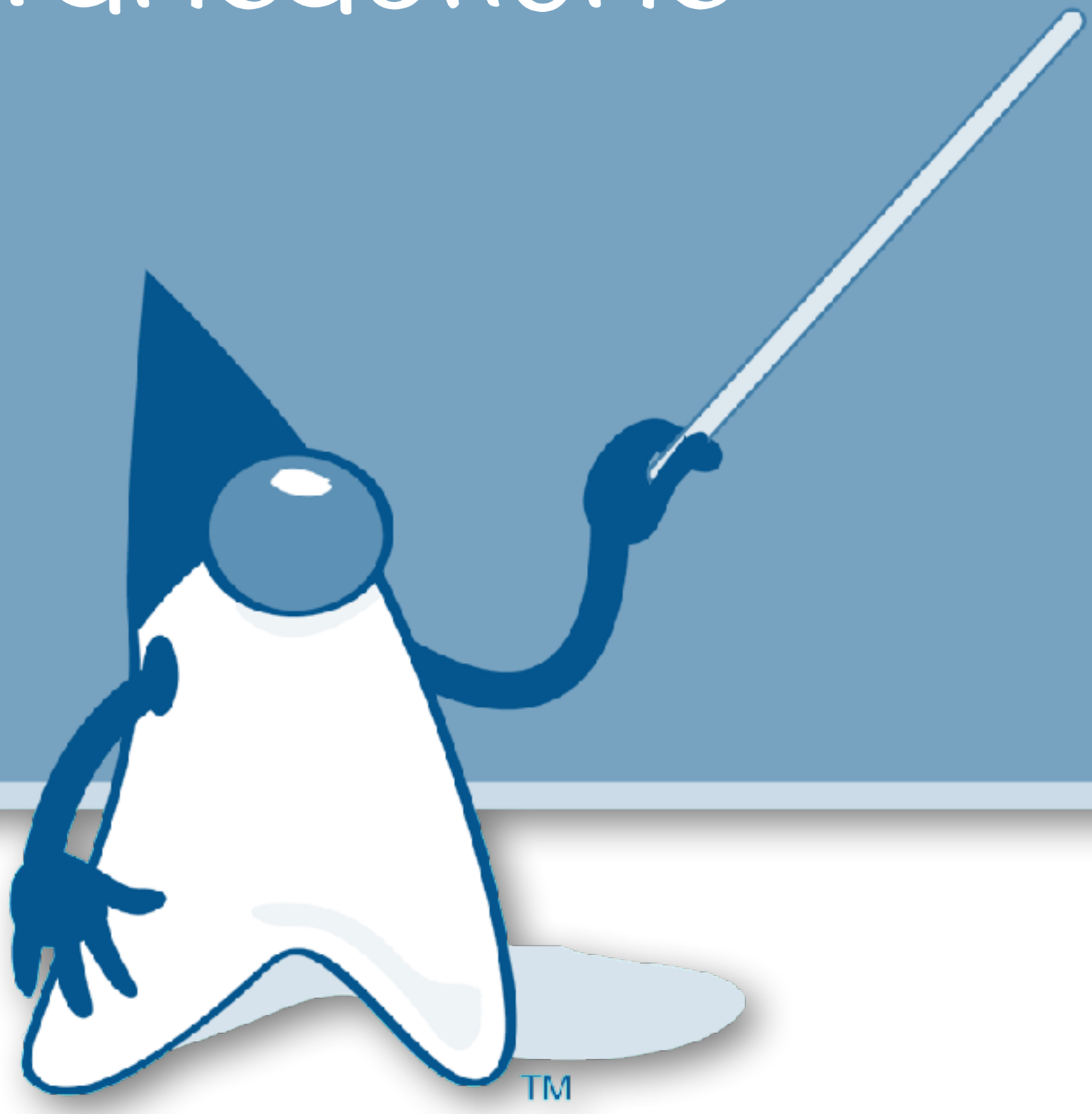
Simple JDBC Insert

Creating a SimpleJdbcInsert

```
private SimpleJdbcInsert simpleJdbcInsert;

@Autowired
public void setDataSource(DataSource ds) {
    simpleJdbcInsert = new SimpleJdbcInsert(ds).withTableName("movies")
            .usingGeneratedKeyColumns("id");
}
```

```
Map<String, Object> params = new HashMap<String, Object>();
params.put("title", movie.getTitle());
params.put("genre", movie.getGenre());
params.put("releasedate", movie.getReleaseDate());

return simpleJdbcInsert.executeAndReturnKey(params);
```

Transactions



TM

Transaction Management

- Consistent programming model across different technologies
  - JDBC, JPA, Hibernate etc.
  - Run JDBC and JPA code in the same transaction
- Declarative tx-management
- Simplified programmatic tx-management API

Creating a transaction manager

- Different transaction managers for different persistence solutions
  - DataSourceTransactionManager
  - HibernateTransactionManager
  - JpaTransactionManager
  - WebLogicJtaTransactionManager
  - WebSphereUowTransactionManager

Creating a transaction manager

JDBC transaction manager

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

JPA transaction manager

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="myEmf"/>
</bean>
```

Declarative Transaction Management

on annotation driven tx-management

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

Make each method

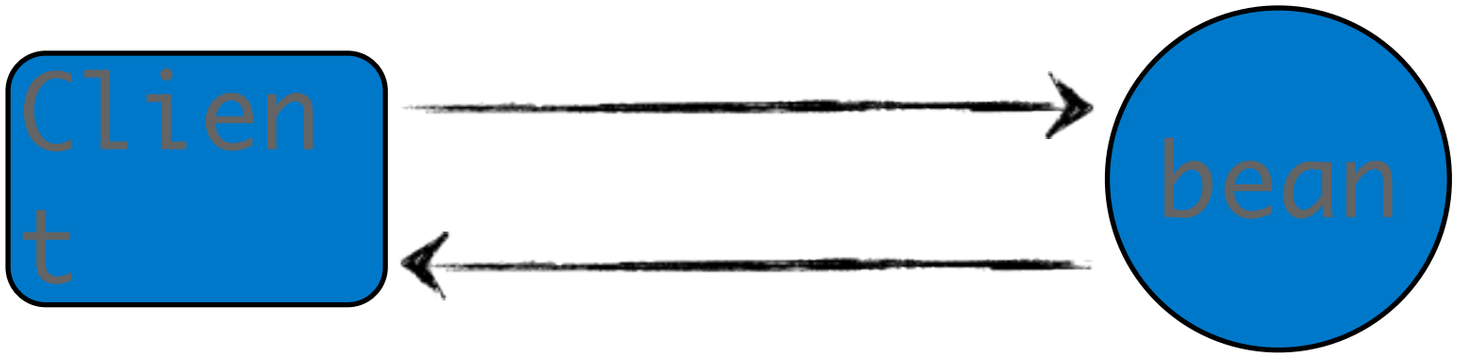
Read only transaction

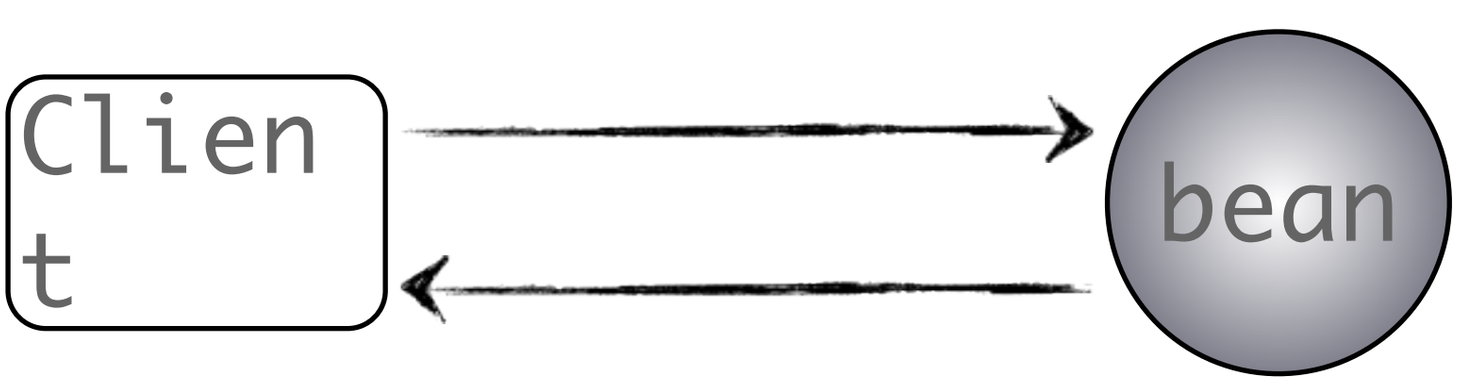
```
@Repository
@Transactional
public class ExampleDAO {
    public void saveContact(String name) {
        //Insert contact
    }

    @Transactional(readOnly = true)
    public List<String> listContacts() {
        //Query contact table
        return null;
    }
}
```

@Transactional

Attribute	Explanation
value	Optional qualifier specifying the tx-manager to use
propagation	Transaction propagation
isolation	Transaction isolation
read-only	Read/write or read-only transaction
rollbackFor	Array of Class objects that extends Throwable and should result in a rollback
noRollbackFor	Array of Class objects that extends Throwable and should not result in a rollback

Propagation REQUIRED

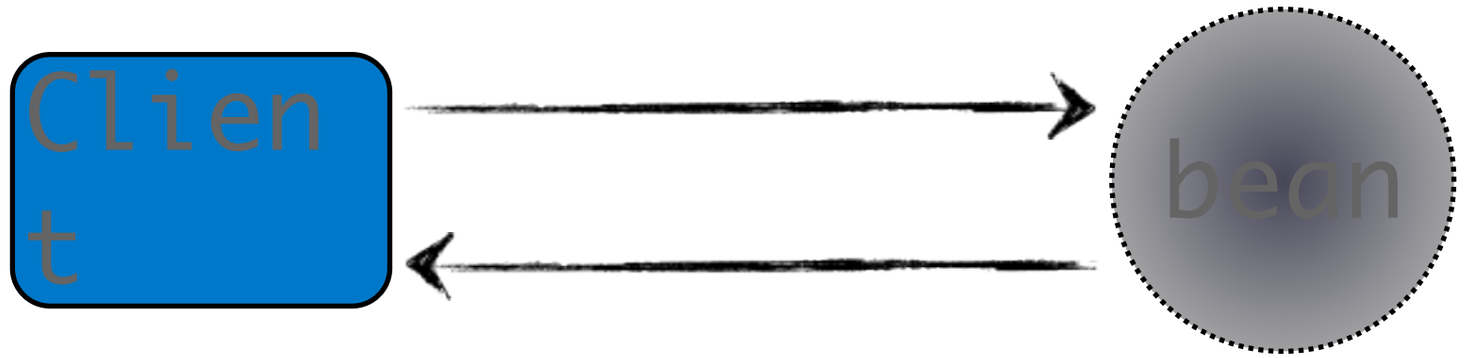


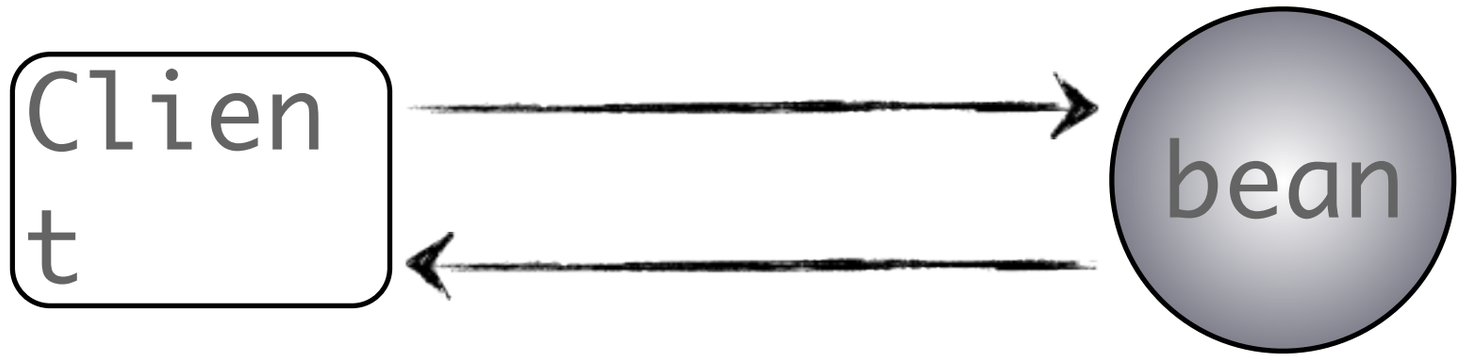


client's transactional context

bean's transactional context

Propagation NESTED





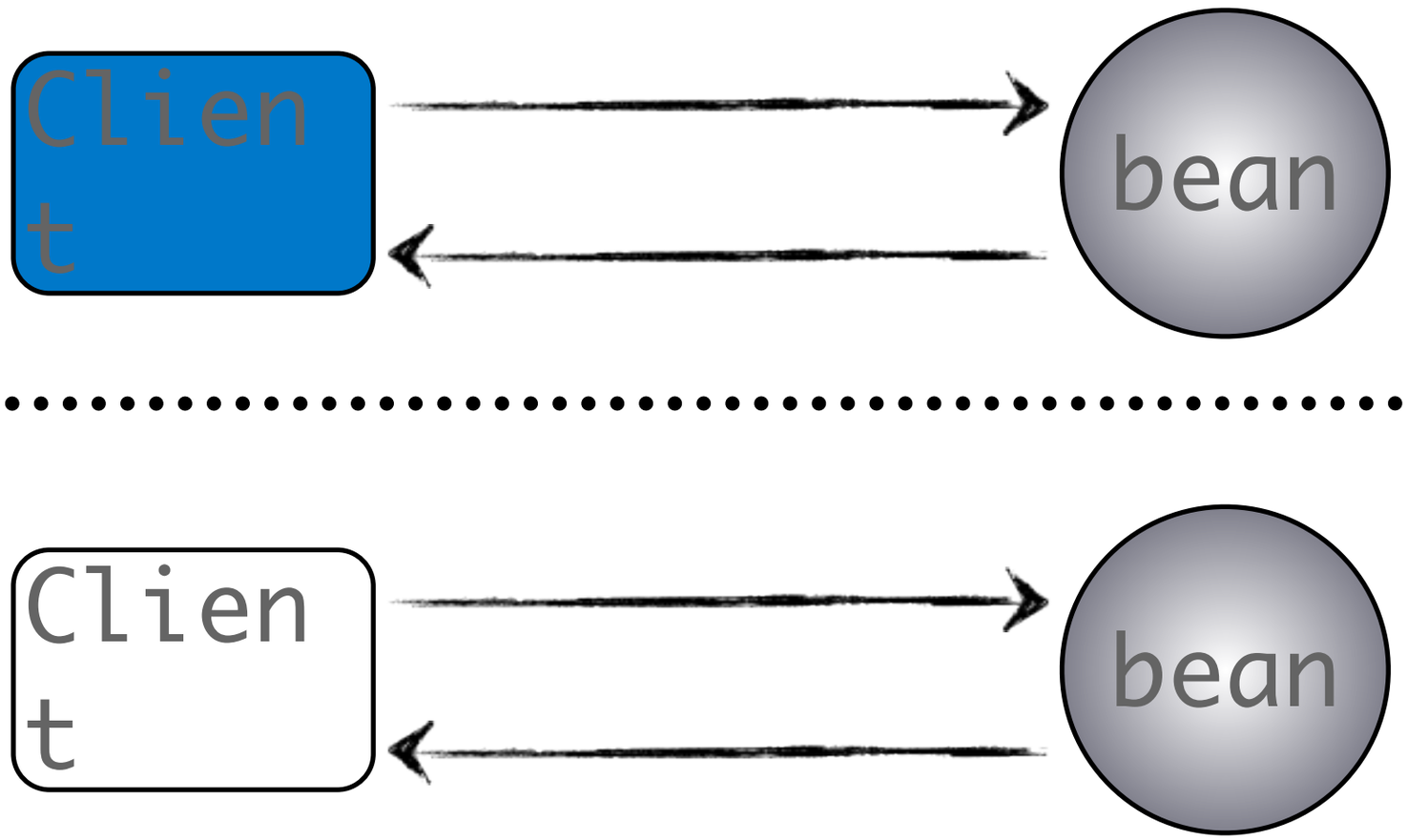
client's transactional context

bean's transactional context

no transactiona

nested transa

Propagation REQUIRES_NEW



client's transactional context

bean's transactional context

Rollback behavior

- Exceptions bubbled up the stack are handled by Spring
- Transaction is rolled back for unchecked exceptions
  - probably an unrecoverable error
- Transaction is not rolled back for check exceptions
  - probably recoverable situation

JPA integration

TM

infoSupport
Solid Innovator

Setting up JPA

- Create a META-INF/persistence.xml file
- Configure an EntityManagerFactory
- Configure a JpaTransactionManager
- Configure the JPA dialect in Spring

persistence.xml

```xml
<persistence-unit name="testPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="true" />
        <property name="hibernate.dialect"
                  value="org.hibernate.dialect.MySQL5InnoDBDialect" />
        <property name="hibernate.hbm2ddl.auto" value="update" />
    </properties>
</persistence-unit>
```

Configuring JPA

```
<bean id="myEmf"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="jpaDialect" ref="jpaDialect"/>
</bean>

<bean id="jpaDialect"
      class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"/>

<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="myEmf"/>
</bean>
```

Using JPA

Inject an EntityManager

```
@PersistenceContext
EntityManager em;
```

Plain JPA usage with declarative transactions

```
@SuppressWarnings("unchecked")
@Transactional(readOnly=true)
public List<Book> listBooks() {
    Query q = em.createQuery("select b from Book b");
    return q.getResultList();
}
```

Templates and DaoSupport

- Several Template and DaoSupport classes exists
  - similar to JdbcTemplate
- Not necessary any more because JPA API is already easy to use
- Prefer plain JPA API

Testing DAOs

- Unit testing doesn't make sense
  - queries can only be tested with a real database
- Run automated tests within a Spring container

Testing DAOs

Run within container

```
@ContextConfiguration(locations="/applicationContext.xml")
public class BookHibernateDaoTest
        extends AbstractTransactionalJUnit4SpringContextTests{

    @Autowired
    private BookCatalog dao;

    @Test
    public void testListBooks() {
        List<Book> result = dao.listBooks();
        assertEquals(5, result.size());
    }
}
```

Provides JDBC utility

Testing DAOs

- Transactions are rolled back after each test by default
  - no need to re-insert test data for each test
- Test JPA code using JDBC queries

```java
@Test
public void testSaveBook() {
    Book book = new Book();
    book.setTitle("Harry Potter");
    int books = countRowsInTable("Book");
    dao.saveBook(book);
    assertEquals(books + 1, countRowsInTable("Book"));
}
```

Disable default rollback behavior

roll back transactions for any method

```
@ContextConfiguration(locations="/applicationContext.xml")
@TransactionConfiguration(defaultRollback = false)
public class BookHibernateDaoTest
        extends AbstractTransactionalJUnit4SpringContextTests{
```

Don't roll back transactions for this method

```
@Test @Rollback(false)
public void testSaveBook() {
```