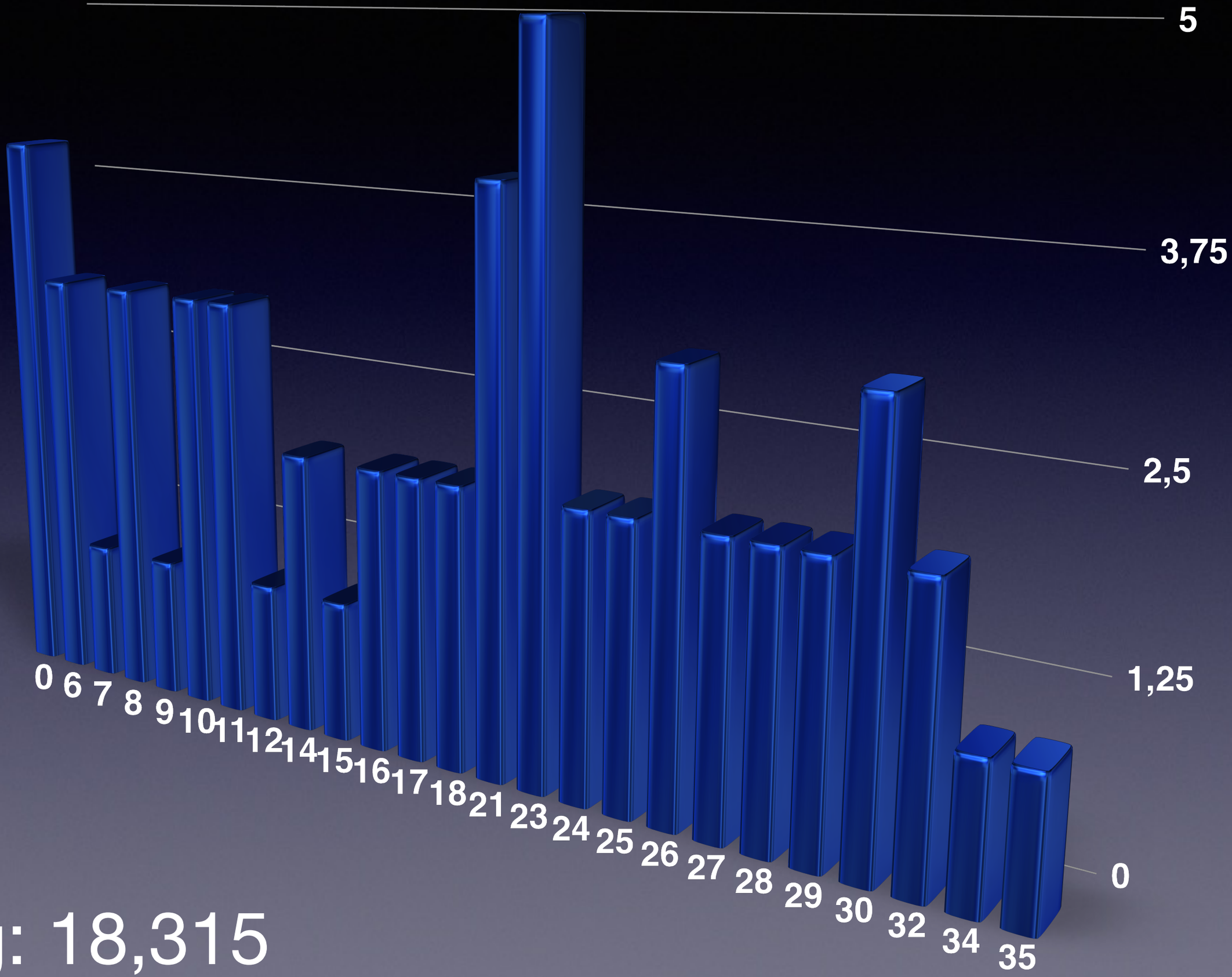
BBM490

ENTERPRISE WEB ARCHITECTURE

Mert ÇALIŞKAN

Hacettepe University Spring Semester ’15

Term Notes



Avg: 18,315

Week 6

JDBC

We'll cover Java Database Connectivity and give samples to connect different databases. We'll also cover what Spring offers as an infra for JDBC and we'll detail the transaction management system.

Relational Database

- It's a database that has a collection of tables of data items.
- Each table should identify a column or group of columns as *primary key*, to uniquely identify each row.
- A *relationship* can then be established between each row in the table and a row in another table by creating a *foreign key*, a column or group of columns in one table that points to the primary key of another table.

What's JDBC?

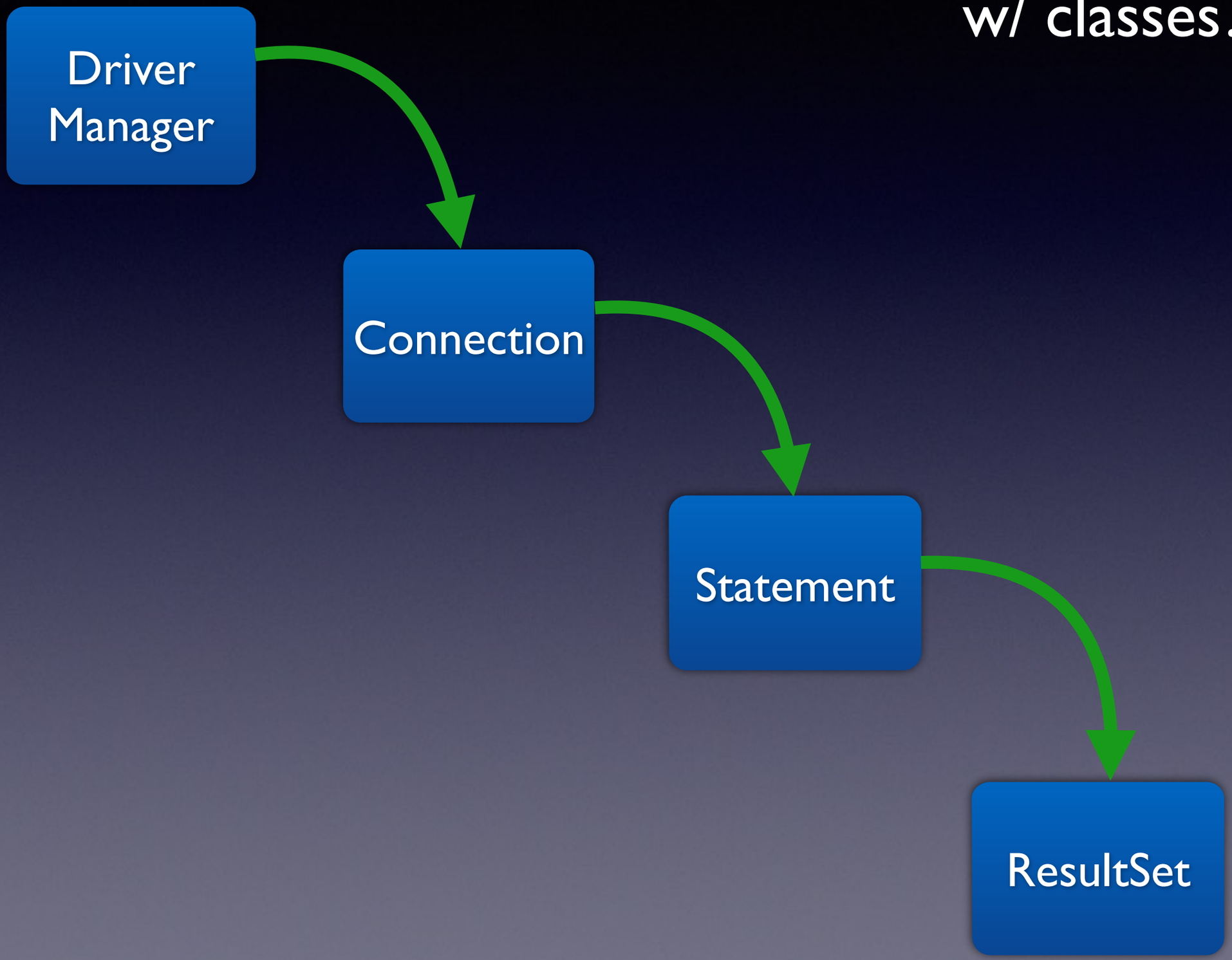
- It's an API for Java that defines how a client may access a relational database.
- It provides an interface to interact with different databases.
- You can also call it as a standard which allows individual providers to implement and extend it with their own JDBC drivers.
- A Driver is an implementation of the JDBC interface and it communicates with a particular database.

Basic Flow of JDBC Usage

- Establishes a connection with a database
- Create JDBC Statements
- Execute SQL Statements
- Processes the results
- Close connections

Steps of JDBC

w/ classes…



Establish a Connection

1. Load the vendor specific driver.

```
Class.forName("com.oracle.jdbc.Driver");
```

The code is forcing the class representing the MySQL driver to load and initialise. There is also another way to achieve this.

```
Driver myDriver = new oracle.jdbc.driver.OracleDriver();
DriverManager.registerDriver(myDriver);
```

- Which one is error prone?

Establish a Connection

2. After you've loaded the driver, you can establish a connection.

```
Connection con =
DriverManager.getConnection( “jdbc:oracle:thin:@oracle-
prod:1521/OPROD", username, password);
```

- The first parameter here is a JDBC url, which defines the way to connect the database. It differs from driver to driver, we'll detail it.
- URL is a must. Other overloaded versions are:

```
getConnection(String url)
getConnection(String url, Properties prop)
getConnection(String url, String user, String password)
```

JDBC URLs

- A database URL is an address that points to your database. Format of the URL is stated below:

```
jdbc:subprotocol:source
```

- Each driver has its own subprotocol.
- Each subprotocol has its own syntax for the source.

JDBC URLs

- Formulation of the URL for different databases is stated below.

DB	JDBC Driver Name	URL Format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname:portNumber/databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:portNumber:databaseName
DB2	oracle.jdbc.driver.OracleDriver	jdbc:db2:hostname:portNumber/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname:portNumber/databaseName

Query the Database

3. Create a Statement object for sending SQL statements to the database. It's the object that is used for executing a static SQL statement and returning the results it produces.

```
Statement stmt = con.createStatement();
```

- Statement provides 2 methods for executing SQLs.
  - `executeQuery()` for SELECT statements
  - `executeUpdate()` for INSERT, UPDATE, DELETE, statements

Process Results

- Result of a SELECT statement (rows/columns) returned as a ResultSet object like,

```
ResultSet rs =  stmt.executeQuery("SELECT * FROM users”);
```

- Step through each row in the result

```
rs.next();
```

- Now we can iterate over the result set and print out the fetched result.

Print the Users table

```
ResultSet rs = stmt.executeQuery("SELECT * FROM users");

while (rs.next()) {
     String userid = rs.getString(1);
     String firstname = rs.getString(“firstname”);
     String lastname = rs.getString(“lastname”);
     String password = rs.getString(4);
     int type = rs.getInt(“type”);
     System.out.println(userid + ” ” + firstname + ” ” +
     lastname + ” ” + password + ” ” + type);
}
```

userid	firstname	lastname	password	type
mkennedy	Mike	Kennedy	cat	1
lshepperd	Laura	Shepperd	dog	0

Closing connection

- At the end of your JDBC program, it is required explicitly close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.
- Relying on garbage collection, especially in database programming, is very poor programming practice. You should make a habit of always closing the connection with the close() method associated with connection object.

- Close the ResultSet object

```
rs.close();
```

- Close the Statement object

```
stmt.close();
```

- Close the connection

```
conn.close();
```

Let's create a Project

- We'll create a Maven application to connect to Oracle database.
- We need to add Oracle driver as maven dependency to the project.

```
<dependency>
  <groupId>com.oracle</groupId>
  <artifactId>ojdbc14</artifactId>
  <version>10.2.0.4.0</version>
</dependency>
```

src:jdbc-simple

Creating Table

- For creating the Users Table on Oracle, execute:

```
CREATE TABLE Users (
    "USERID" VARCHAR2(250 CHAR) NOT NULL,
    "FIRSTNAME" VARCHAR2(255 CHAR) NOT NULL,
    "LASTNAME" VARCHAR2(255 CHAR) DEFAULT NULL,
    "PASSWORD" VARCHAR2(255 CHAR) NOT NULL,
    "TYPE" NUMBER(1,0) DEFAULT NULL
);
```

- The table will have userid, firstname and password columns as not null and lastname and type could be empty within the table.

If we have used PostgreSQL

- For PostgreSQL JDBC Driver add maven dependency as,

```
<dependency>
    <groupId>postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.1-901.jdbc4</version>
</dependency>
```

- To use driver we need to register it as,

```
Driver myDriver = new org.postgresql.Driver();
DriverManager.registerDriver(myDriver);

String url = "jdbc:postgresql://localhost:5432/bbm490";
Connection conn = DriverManager.getConnection(url, "user", "pass");
```

Conditional Cases

- What about if we want to add a conditional case to the SQL?

Interfaces	Recommended Use
Statement	Use for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use when you want to access database stored procedures. The CallableStatement interface can also accept runtime input parameters.

PreparedStatement

- The *PreparedStatement* interface extends the Statement interface which gives you added functionality.
- All parameters in JDBC are represented by the ? symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.
- The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter.

PreparedStatement

- Now we can easily select users by providing a userid. Here we are using PreparedStatement.

```
String sql = "SELECT * FROM users where userid = ?";
PreparedStatement stmt = conn.prepareStatement(sql);
stmt.setString(1, "001");
ResultSet rs = stmt.executeQuery();
```

- Of course we will need a way to update the data also.

Data Types

SQL	JDBC/Java
VARCHAR	java.lang.String
CHAR	java.lang.String
LONGVARCHAR	java.lang.String
BIT	boolean
NUMERIC	java.math.BigDecimal
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
FLOAT	float
DOUBLE	double
VARBINARY	byte[ ]
BINARY	byte[ ]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
CLOB	java.sql.Clob
BLOB	java.sql.Blob
ARRAY	java.sql.Array
REF	java.sql.Ref
STRUCT	java.sql.Struct

What we can use for the setXXX methods…

INSERTING DATA

- We only did query on the DB. How about inserting some?
- We'll be using executeUpdate for executing an insert statement.

```
Connection conn = DriverManager.getConnection(url, "root", "");
Statement stmt = conn.createStatement();
String SQL = "INSERT INTO Users VALUES ('003', 'Mike', 'Johnson', 'bird', 1)";
stmt.executeUpdate(SQL);
```

Transactions

- A transaction is a unit of work performed within a database management system. It's a way of representing a state change.
- Said as a unit of work, a transaction contains a sequence of one or more SQL operations that are treated as a unit.
- Either all of these SQLs succeed or fail.
- Transactions should be *atomic*, *consistent*, *isolated* and *durable*, abbreviated as ACID.

Transactions

- Commit / Rollback concept,
- Commit:
    - Execute all statements as one unit
    - Finalise updates
- Rollback:
    - Abort transaction
    - All un-commited statements are discarded
    - Revert database to its previously original state

ACID

- Atomic: There might be several operations performed over data in any transaction. Those operations must all succeed or commit, or if something goes wrong none of them should be persisted, in other words they all must be rolled back. Atomicity is also known as unit of work.
- Consistency: It defines that during the course of an active transaction underlying database will never be in an inconsistent state. For example, if order items cannot exist without an order, system won't let you add order items without first adding an order.

ACID

- Isolated:It defines how protected your uncommitted data to other concurrent transactions. There are several isolation levels ranging from least protective which offers access to uncommitted data, to the most at which no two transactions appear work at the same time. Isolation is closely related with concurrency and consistency. If you increase the level of isolation you get more consistency but loose from concurrency, i.e. performance. On the other hand if you decrease the level, your transactions performance will increase but you will risk of consistency.
- Durability: It means that when we receive successful commit message from system, we can be sure that our changes are reflected to the system and they will survive any system failure that might occur after that time. Basically, when we commit, our changes get permanent and won't be lost.

Transactions

- JDBC Connection is set as auto-commit to true by default. You can disable it as,

```
conn.setAutoCommit(false);
```

- When you want to commit your changes you can call,

```
conn.commit();
```

- And to rollback,

```
conn.rollback();
```

Transactions

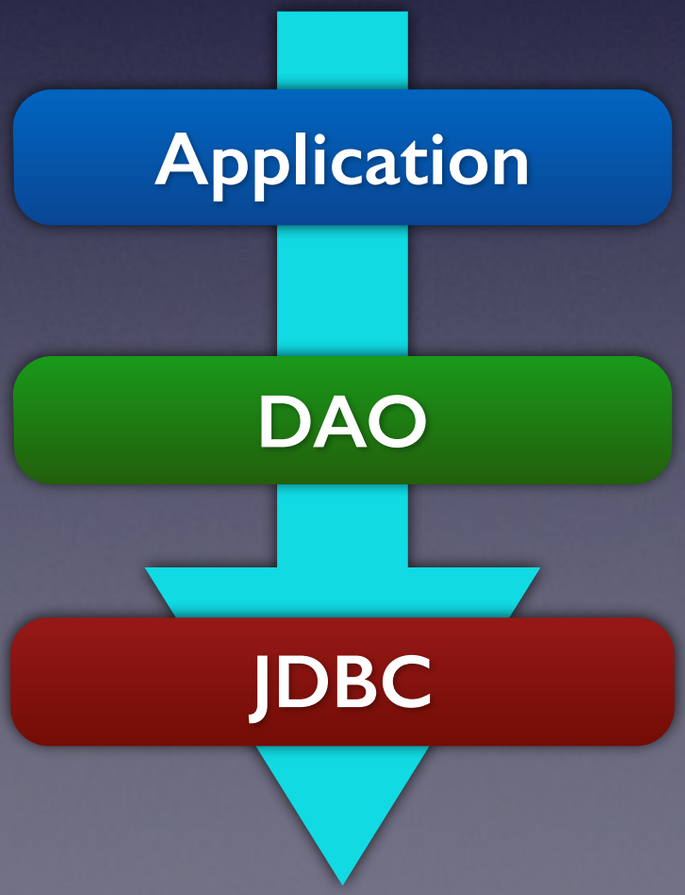
```
Connection conn = DriverManager.getConnection(url, "root", "");
try {
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();
   String SQL = "INSERT INTO Users VALUES ('003', 'Mike', 'Johnson', 'bird',
1)";
   stmt.executeUpdate(SQL);

   SQL = "INSERT IN Users VALUES ('004', 'John', 'Nash', 'duck', 0)";
   stmt.executeUpdate(SQL);

   conn.commit();
} catch (SQLException se) {
   conn.rollback();
}
```

DAO Pattern

- DAO stands for Data Access Object
- The pattern is a widely accepted approach to make an abstraction on top of the persistency layer/code.



- You can easily replace the DAO layer and its attached JDBC code since the persistency layer is detached from the application code.

Spring JDBC

- Well, plain vanilla JDBC code is difficult.
- It does not support some good stuff, like named parameters, row mapping & etc, out of the box.
- Spring JDBC takes care of all the low-level details like registering driver, opening connection closing it & etc. It also supports enhanced templating features.
- Spring is the saviour again!

Spring JDBC

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	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, statement and resultset.	X	

JDBC Template

- JbdcTemplate : This is the central class in the JDBC core package. It simplifies the use of JDBC and helps to avoid common errors. It executes core JDBC workflow that we defined in previous slides and leaves the application code just to provide SQL and extract results.

Let's create Spring Project

- We'll use Spring Jdbc Template and show the features of JdbcDaoSupport for our base Dao class.
- You will see that it will ease the development and we don't need the vanilla Jdbc code anymore.

src:spring-jdbc-simple

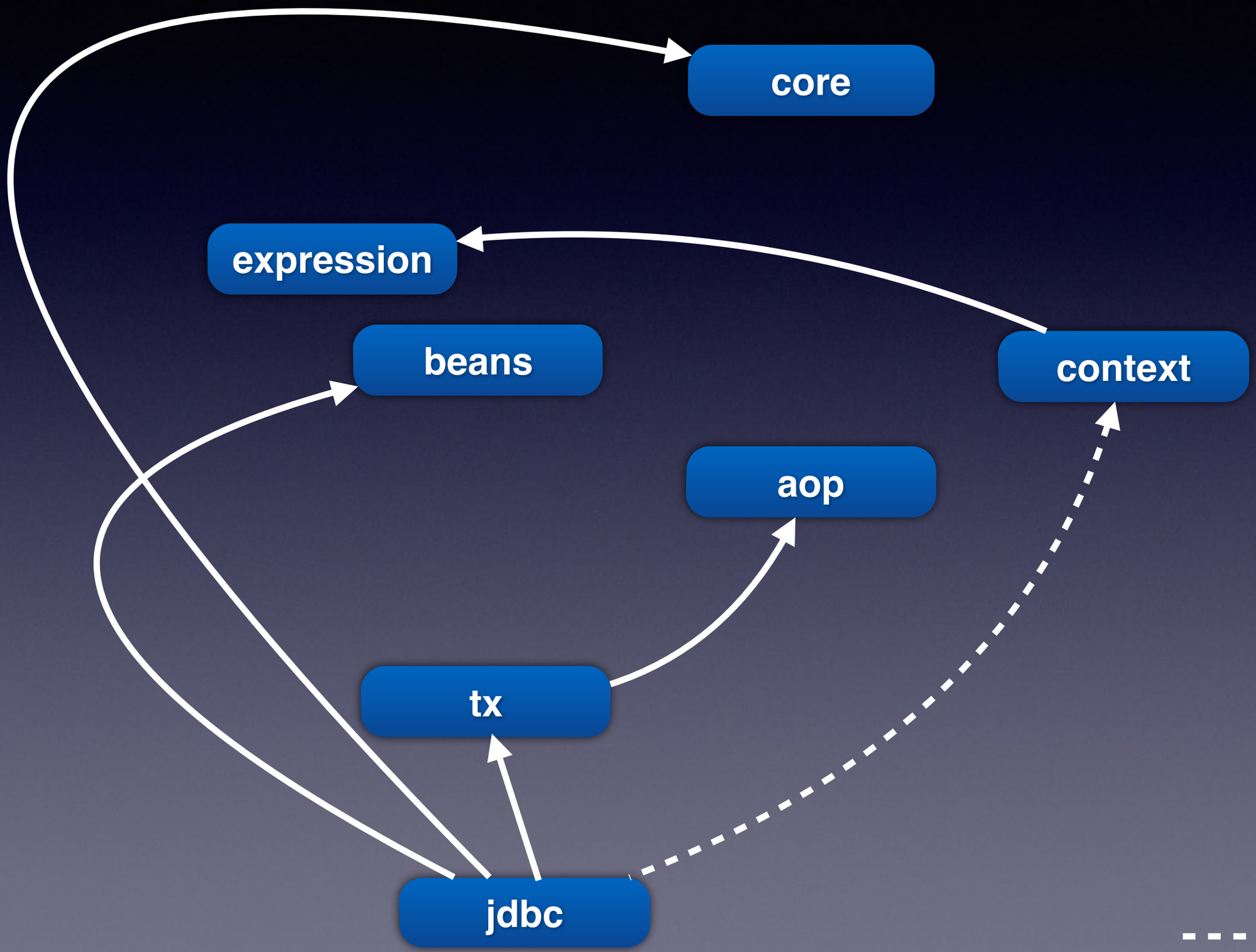
Project Configurations

- We have new Spring subprojects' dependencies.
- -jdbc: JDBC Data Access Library

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-jdbc</artifactId>
   <version>4.1.5.RELEASE</version>
</dependency>
```

- Offers a good deal of transitive dependencies.
- Remember the graph? (comin' up next…)

Spring Sub-Projects' Inter-Dependencies



The Dao Class

```
public class UserDaoImpl extends JdbcDaoSupport implements UserDao {

   public void create(String userid, String firstname, String lastname,
         String password, int type) {

      String sql = "insert into Users "
            + "(userid, firstname, lastname, password, type) values
(?, ?, ?, ?, ?)";

      getJdbcTemplate().update(sql,
            new Object[] { userid, firstname, lastname, password, type });
   }

   public User find(String userid) {
      String sql = "select * from Users where userid = ?";

      User user = getJdbcTemplate().queryForObject(sql,
            ParameterizedBeanPropertyRowMapper.newInstance(User.class),
userid);

         return user;
   }
}
```

ApplicationContext

- Now we can define the datasource via Spring and inject it to our Dao class.

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@localhost:1521/ORCL"/>
    <property name="username" value="BUWORKSHOP"/>
    <property name="password" value="BUWORKSHOP"/>
</bean>


<bean id="userDao"
      class="tr.edu.hacettepe.bbm490.UserDaoImpl">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

RowMapper

- RowMapper: It's interface used by JdbcTemplate for mapping rows of a ResultSet on a per-row basis. Implementations of this interface perform the actual work of mapping each row to a result object.
- ParameterizedRowMapper It's an implementation of the *RowMapper* interface that converts a row into a new instance of the specified mapped target class.

```
String sql = "select * from Users where userid = ?";
User user = getJdbcTemplate().queryForObject(sql,
                ParameterizedBeanPropertyRowMapper.newInstance(User.class), userid);
```

Creating your own RowMapper

```
public class UserMapper implements RowMapper<User> {

  public User mapRow(ResultSet rs, int rowNum) throws
SQLException {
    User user = new User();
    user.setUserid(rs.getString("userid"));
    user.setFirstname(rs.getString("firstname"));
    user.setLastname(rs.getString("lastname"));
    user.setPassword(rs.getString(4));
    user.setType(rs.getInt("type"));

    return user;
  }
}
```

Transaction Management w/ Spring

- We'll create an application with Spring based on Maven project structure.
- We will demonstrate the Spring Transaction Management infrastructure.
- @Transactional annotation will be used to handle transactions.

Maven Dependencies

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-context</artifactId>
   <version>4.1.5.RELEASE</version>
</dependency>
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-jdbc</artifactId>
   <version>4.1.5.RELEASE</version>
</dependency>

<dependency>
   <groupId>com.oracle</groupId>
   <artifactId>ojdbc6</artifactId>
   <version>11.2.0.4</version>
</dependency>
```

applicationContext

- TransactionManager definition with annotation based configuration.

```
<?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"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="tr.edu.hacettepe.bbm490" />
    <tx:annotation-driven />

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

applicationContext

- dataSource definition with Dao bean definition

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@localhost:1521/ORCL"/>
    <property name="username" value="BUWORKSHOP"/>
    <property name="password" value="BUWORKSHOP"/>
</bean>

<bean id="userDao"
      class="tr.edu.hacettepe.bbm490.UserDaoImpl">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

DAO

- What would be the behaviour?

```
public void create(String userid, String firstname, String lastname,
      String password, int type) {

   String sql = "insert into Users "
         + "(userid, firstname, lastname, password, type) values
(?, ?, ?, ?, ?)";

   getJdbcTemplate().update(sql,
         new Object[] { userid, firstname, lastname, password, type });

   throw new RuntimeException();
}
```

- We are throwing a RuntimeException :O

DAO

- And what now?

```
@Transactional
public void create(String userid, String firstname, String lastname,
      String password, int type) {

   String sql = "insert into Users "
         + "(userid, firstname, lastname, password, type) values
(?, ?, ?, ?, ?)";

   getJdbcTemplate().update(sql,
         new Object[] { userid, firstname, lastname, password, type });

   throw new RuntimeException();
}
```

- Should I see an insert on the DB?

src:spring-jdbc-transaction

@Transactional

- Describes transaction attributes on a method or class.
- It has different propagation types.
- REQUIRED: Support a current transaction, create a new one if none exists.
- SUPPORTS: Support a current transaction, execute non-transactionally if none exists.
- MANDATORY: Support a current transaction, throw an exception if none exists.
- REQUIRES_NEW: Create a new transaction, suspend the current transaction if one exists.

@Transactional

- NOT_SUPPORTED: Execute non-transactionally, suspend the current transaction if one exists.
- NEVER: Execute non-transactionally, throw an exception if a transaction exists.
- NESTED: Execute within a nested transaction if a current transaction exists, behave like PROPAGATION_REQUIRED
- Propagation type can be easily set like:

```
@Transactional(propagation=Propagation.REQUIRED)
```

rollbackFor / noRollbackFor

- rollbackFor: Defines zero (0) or more exception classes, indicating which exception types must cause a transaction rollback.
- noRollbackFor: Defines zero (0) or more exception classes, indicating which exception types must **not** cause a transaction rollback.

```
@Transactional(rollbackFor=RuntimeException.class)
```