



## **D-JAVA-201-001**

**Developing Object-Oriented  
Programs in Java  
Hands-on exercises**

<b><i>Lab 01: Object-Oriented Concepts, Banking .....</i></b>	<b><i>3</i></b>
<b><i>Lab 02: Building Java classes .....</i></b>	<b><i>5</i></b>
<b><i>Lab 03: Static Methods .....</i></b>	<b><i>8</i></b>
<b><i>Lab 04: Junit Testing.....</i></b>	<b><i>9</i></b>
<b><i>Lab 05: Using Junit Annotations .....</i></b>	<b><i>10</i></b>
<b><i>Lab 06: Inheritance .....</i></b>	<b><i>15</i></b>
<b><i>Lab 07: Abstract Classes.....</i></b>	<b><i>19</i></b>
<b><i>Lab 08: Interfaces .....</i></b>	<b><i>23</i></b>
<b><i>Lab 09: Collections .....</i></b>	<b><i>24</i></b>
<b><i>Lab 10: Exception Handling .....</i></b>	<b><i>26</i></b>
<b><i>Lab 11: Creating Database .....</i></b>	<b><i>26</i></b>
<b><i>Lab 12: Working with JDBC .....</i></b>	<b><i>30</i></b>
<b><i>Lab 13: Annotations.....</i></b>	<b><i>32</i></b>
<b><i>Lab 14: Mocking .....</i></b>	<b><i>36</i></b>

## Lab 01: Object-Oriented Concepts, Banking

### Objective:

Object-oriented analysis makes extensive use of use-cases. A use-case is a description of an interaction with a system. In this session, you will be presented with some use cases and asked to determine what the candidate objects are. A candidate object is something from the use-case that can potentially be an object in the implementation. Typically, during the analysis stage, too many objects are discovered. That is, you will likely find too many candidate objects — this is normal and expected.

### Lab:

#### Part 1: Identify objects

Determine the candidate objects from the following use-case:

“Joe Smith walks up to the automated teller machine. He inserts his card and is prompted for his password, and transaction information. Joe enters his password via the keypad and indicates that he would like to make a withdrawal of \$50. The password is validated by the automated teller machine, the transaction is processed and \$50 is delivered to Joe along with a transaction receipt. The machine asks if Joe would like to make a further transaction. Joe presses the ‘No’ button and the machine returns Joe’s card.”

What are the objects?

#### Part 2: Identify basic classes

Based on the results of previous step what classes make up the banking system? What are the class names, data and behavior?

The basic classes in the system may be ATM, Bank, Customer, Account, Card etc.

Customer has accounts and set of transactions. A Card may have accounts associated with etc.

Draw an informal diagram of the message passing between the objects in the use case. Use any symbols, pictures, arrows, text etc. that you like. Indicate which objects are likely to exist in the real world, which in the software, and which in both.

#### Part 3: Review of Concepts

In this part, we explore the world of objects. There are few wrong answers for the following questions. However, you must be able to justify your answers.

1. You might expect to find the following kinds of objects at in a banking application: bank, customer, account, and transaction. What relationships exist between these objects (i.e. a Bank manages several Accounts)
2. For each of the following objects, list any fields and methods that the object might have.

Class	Fields	Methods
Bank		
Customer		
Account		
Transaction		

3. Decide whether each of the following is a class or an instance:

	Class?	Instance?
a) Bob's bank account		
b) Checkbook		
c) Bank		
d) My bank statement for September		
e) the Federal Reserve Bank of New York		
f) Central Bank		
g) Bank Account		
h) Corporate Loan		
i) Anisa's savings statement		
j) my branch of the Chase Manhattan Bank		
k) Mortgage		

## Lab 02: Building Java classes

### Objective:

In this hands-on session you will build the BankAccount class. This class is part of a larger collection of classes that model the inner workings of a bank. Instances of the BankAccount class provide a simple model of how bank accounts might work in an overly simplified world.

For this session, a bank account is exclusively interested in maintaining the name of the account owner, the number of the account and the account's balance. Behavior will be limited to methods that provide a means of crediting and debiting the account.

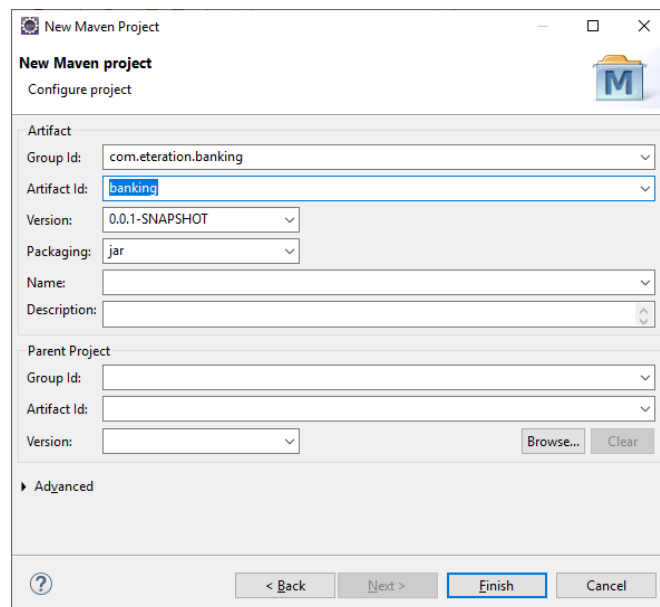
The following code demonstrates how BankAccounts might be used:

```
BankAccount account;  
account = new BankAccount("Jim", 12345);  
account.credit(1000.0);  
account.debit(50.0);  
System.out.println(account.getBalance());
```

### Lab:

#### Step 1: Create the project

Create new Simple Maven Project called banking.



Add java compiler plugin in the pom.xml file

```
<build>  
  <plugins>  
    <plugin>  
      <groupId>org.apache.maven.plugins</groupId>  
      <artifactId>maven-compiler-plugin</artifactId>
```

```
<configuration>
  <source>1.8</source>
  <target>1.8</target>
  <encoding>UTF-8</encoding>
</configuration>
</plugin>
</plugins>
</build>
```

### Step 2: Create package

In the BankingApplication project, create a package called "banking.models". All of the classes we build this week will be placed in this package. Usually classes that represent a model are in the same package, classes used for testing are in a different package, etc. To create new package, select the project and choose New -> Package from the context menu. Use banking.models as a package name and click Finish.

### Step 3: Create BankAccount class

Create a class BankAccount that will have private fields owner where the field type is java.lang.String. Also create fields to hold the accountNumber (int) and balance (double). These fields should be declared private.

Generate getter and setter for the fields. Add a constructor to the BankAccount class that takes the owner name and account number as parameters (as shown in the example above). This constructor should set the appropriate fields in the new instance.

Will the following code segments work? Make sure that you understand why or why not.

```
BankAccount aBankAccount = new BankAccount();
aBankAccount.setOwner("Your Name");
aBankAccount.getOwner();
BankAccount aBankAccount = new BankAccount("Jim", 12345);
aBankAccount.owner = "Wayne";
BankAccount.setOwner("Your Name");
```

Build the credit() method to be used as specified above. This method adds the supplied amount to the receiving BankAccounts balance.

```
public void credit(double amount) {
  ...
}
```

Build the debit() method. This method subtracts the supplied amount from the receiving BankAccounts balance.

```
public void debit(double amount) {
  ...
}
```

Override the `toString()` method in the `BankAccount` class to simply return string `"Jim's $40 BankAccount"`. Later in the labs you will add more behavior to this method.

#### **Step 4: Creating BankAccount instances**

Create a class named `TestBankAccount` with main method. Add following code snippet. Run class and inspect the result

```
BankAccount account;  
account = new BankAccount("Jim", 12345);  
account.credit(1000.0);  
account.debit(50.0);  
System.out.println(account.getBalance());
```

## Lab 03: Static Methods

### Static Methods:

1. Modify the BankAccount to include a static field of type double named **interestRate**. Make the field private. Set the initial value to 3.5.

Test your change using the following code sample.

```
BankAccount account1 = new BankAccount("Jim", 12345);  
BankAccount account2 = new BankAccount("Alice", 567);  
System.out.println(account1.interestRate);  
System.out.println(account2.interestRate);
```

What values were printed out? When were these values set?

2. Modify your test code so that the **interestRate** field is modified. Use this code:

```
BankAccount account1 = new BankAccount("Jim", 12345);  
BankAccount account2 = new BankAccount("Alice", 567);  
account1.interestRate = 5;  
System.out.println(account2.interestRate);
```

What is the value of account2's **interestRate**? Why?



## Lab 04: Junit Testing

In this lab you will be using Junit for testing BankAccount class.

1. First of all, we need to add dependencies for Junit5 into our pom.xml file:

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.5.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.5.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

2. Create a java package named banking.models.test into the src/test/java folder
3. Create a new Junit5 Test Case and name it TestingBankAccount in the previously created folder
4. Add first test method for testing BankAccount's credit and debit methods:

```
@Test
void testAccount() {
    BankAccount account;
    account = new BankAccount("Jim", 12345);
    assertTrue(account.getBalance()==0);
    account.credit(1000.0);
    assertTrue(account.getBalance()==1000);
    account.debit(50.0);
    assertTrue(account.getBalance()==950);
}
```

5. Add another test method for test static field and method:

```
@Test
void testInterestRate() {
    BankAccount account1 = new BankAccount("Jim", 12345);
    BankAccount account2 = new BankAccount("Alice", 567);
    assertEquals(account1.getInterestRate(),account2.getInterestRate());

    account1.setInterestRate(5);
    assertEquals(account1.getInterestRate(),account2.getInterestRate());

    BankAccount.setInterestRate(4.5);
    assertEquals(account1.getInterestRate(),account2.getInterestRate());
}
```

6. Delete TestBankAccount class.
7. Run test class. Your tests should be complete successfully

## Lab 05: Using Junit Annotations

- Add other junit dependencies into your pom.xml

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.5.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-suite-api</artifactId>
    <version>1.5.2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>1.5.2</version>
    <scope>test</scope>
</dependency>
```

- Create a new package named banking.junit.test in the src/test/java

### @ParameterizedTest

- Parameterized tests make it possible to run a test multiple times with different arguments. They are declared just like regular @Test methods but use the @ParameterizedTest annotation instead.



```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;

class TestJUnit {

    @ParameterizedTest
    @ValueSource(strings= {"ali", "ahmet", "arda"})
    void endWith(String str) {
        assertTrue(str.startsWith("a"));
    }
}
```

Expected Output:

Finished after 0.292 seconds



Runs: 3/3     Errors: 0     Failures: 0

TestJUnit [Runner: JUnit 5] (0.066 s)  
 endWith(String) (0.066 s)  
   [1] ali (0.066 s)  
   [2] ahmet (0.004 s)  
   [3] arda (0.006 s)

## @RepeatedTest

JUnit 5 has the ability to repeat a test a specified number of times simply by annotating a method with `@RepeatedTest` and specifying the total number of repetitions desired. Each invocation of a repeated test behaves like the execution of a regular `@Test` method. This is useful in UI testing with

```
@RepeatedTest(value = 4, name = "{displayName}"
{currentRepetition}/{totalRepetitions}")
@DisplayName("RepeatingTest")
void customDisplayName(RepetitionInfo repInfo, TestInfo testInfo) {
    int i = 2;
    System.out.println(testInfo.getDisplayName() + "-->" +
repInfo.getCurrentRepetition());
    assertEquals(repInfo.getCurrentRepetition(), i);
}
```

Runs: 7/7     Errors: 0     Failures: 3

TestJUnit [Runner: JUnit 5] (0.071 s)  
 > endWith(String) (0.032 s)  
 RepeatingTest (0.028 s)  
   RepeatingTest 1/4 (0.028 s)  
   RepeatingTest 2/4 (0.003 s)  
   RepeatingTest 3/4 (0.003 s)  
   RepeatingTest 4/4 (0.003 s)

- As you can see from the result of the test, when the test only `i==2` pass, otherwise the test is fails.

## @DisplayName

Test classes and test methods can declare custom display names that will be displayed by test runners and test reports.

```
@DisplayName("JUnit5 Display Header")
class TestJUnit {

    @Test
    @DisplayName("JUnit5 Display Name Example")
    void testWithDisplayName() {
    }
```

```
}
```

## @BeforeEach - @AfterEach

The @BeforeEach annotation denotes that the annotated method should be executed before each test method.

Create a new TestCase class and add following methods:

```
class TestJUnit2 {  
  
    @BeforeEach  
    void init(TestInfo testInfo) {  
        String callingTest = testInfo.getTestMethod().get().getName();  
        System.out.println("before method:"+callingTest);  
    }  
  
    @Test  
    void firstStep() {  
        System.out.println(1);  
    }  
  
    @Test  
    void secondStep() {  
        System.out.println(2);  
    }  
  
    @AfterEach  
    void after(TestInfo testInfo) {  
        String callingTest = testInfo.getTestMethod().get().getName();  
        System.out.println("after method:"+callingTest);  
    }  
}
```

Expected Output:

```
before method:firstStep  
1  
after method:firstStep  
before method:secondStep  
2  
after method:secondStep
```

## @BeforeAll - @AfterAll

@BeforeAll annotation executes a method before all tests.

The @AfterAll annotation is used to execute the annotated method, only after all tests have been executed.

Create a new TestCase class and add following methods:

```
class TestJUnit3 {  
  
    @BeforeAll  
    static void beforeAll() {
```

```
        System.out.println("only once before all test");
    }

    @Test
    void firstTest() {
        System.out.println("first");
    }

    @Test
    void secondTest() {
        System.out.println("second");
    }

    @AfterAll
    static void afterAll() {
        System.out.println("only once after all test");
    }
}
```

#### Expected Output:

```
only once before all test
first
second
only once after all test
```

### @Disabled

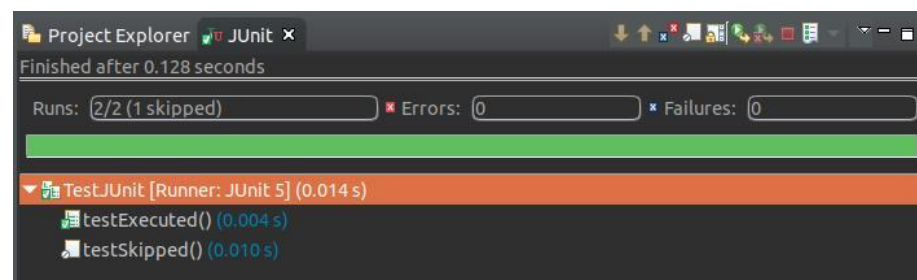
The **@Disabled** annotation is used to disable or skip tests at class or method level. When declared at class level, all @test methods are skipped.

```
class TestJUnit4 {

    @Disabled
    @Test
    void testSkipped() {
    }

    @Test
    void testExecuted() {
    }
}
```

#### Expected Output:



## @Tag

We can use this annotation to declare tags for filtering tests, either at the class or method level. The **@Tag** annotation is useful when we want to create a test pack with selected tests.

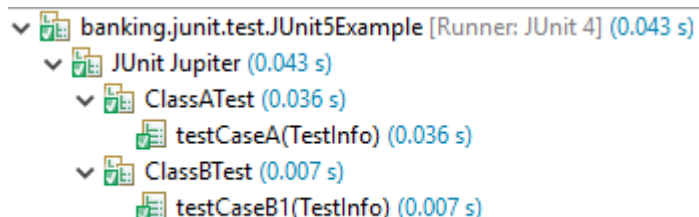
```
@RunWith(JUnitPlatform.class)
@SelectPackages("banking.junit.test")
@IncludeTags("development")
public class JUnit5Example
{
}
```

```
@Tag("development")
public class ClassATest
{
    @Test
    @Tag("userManagement")
    void testCaseA(TestInfo testInfo) {
    }
}
```

```
public class ClassBTest
{
    @Test
    @Tag("development")
    @Tag("production")
    void testCaseB1(TestInfo testInfo) {
    }

    @Test
    @Tag("production")
    void testCaseB2(TestInfo testInfo) {
    }
}
```

Expected Result:



```
banking.junit.test.JUnit5Example [Runner: JUnit 4] (0.043 s)
├── JUnit Jupiter (0.043 s)
│   ├── ClassATest (0.036 s)
│   │   └── testCaseA(TestInfo) (0.036 s)
│   └── ClassBTest (0.007 s)
│       └── testCaseB1(TestInfo) (0.007 s)
```

## assertAll

The new assertion introduced in JUnit 5 is `assertAll`.

This assertion allows the creation of grouped assertions, where all the assertions are executed and their failures are reported together. In details, this assertion accepts a heading, that will be included in the message string for the `MultipleFailureError`, and a `Stream of Executable`.

```
@Test
void groupAssertions() { int[] numbers = { 0, 1, 2, 3, 4 };
    assertAll("numbers",
        () -> assertEquals(numbers[0], 1),
        () -> assertEquals(numbers[3], 3),
        () -> assertEquals(numbers[4], 1));
}
```

Expected Output:

```
org.opentest4j.MultipleFailuresError: numbers (2 failures)
    java.lang.AssertionError: expected:<0> but was:<1>
    java.lang.AssertionError: expected:<4> but was:<1>
```

If we change the code like this, Our test will be pass.

```
@Test
void groupAssertions() { int[] numbers = { 0, 1, 2, 3, 4 };
    assertAll("numbers",
        () -> assertEquals(numbers[0], 0),
        () -> assertEquals(numbers[3], 3),
        () -> assertEquals(numbers[4], 4));
}
```

Expected Output:

```
JUnit5Example2 [Runner: JUnit 5] (0.031 s)
  groupAssertions() (0.031 s)
```

## Lab 06: Inheritance

### Objective:

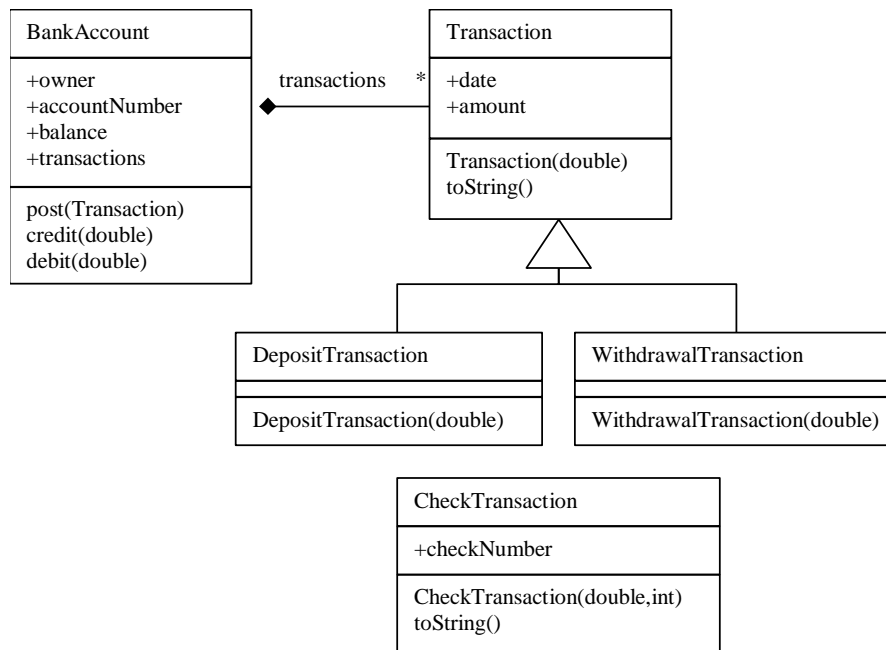
In this lab you will be using inheritance for better design, increased flexibility and maintainability of the system.

Inheritance helps with refactoring the code, as well as with appropriate representation of the objects (i.e. it makes objects closer to the real life). Quite often some commonalities between similar classes are abstracted in an abstract super class and reused in subclasses. An abstract class is a class that cannot have instances. Interfaces are used in Java for further abstraction as they define only object's protocol. Interfaces are also commonly used for cross-hierarchy polymorphism

The object model for our banking system will be extended to include transaction objects. A transaction object keeps track of the kind of transaction (deposit, withdrawal or check) as well as the date and amount of the transaction. Check transactions further require the check number. Note that checks are written against the account (they cause money to be removed from the account).

The following diagram shows how BankAccounts and Transactions are related. Note that, for this session, Transactions will be implemented as a hierarchy of classes. An instance of DepositTransaction represents a deposit; a WithdrawalTransaction represents a withdrawal

(the triangle on the diagram indicates inheritance). Inheritance for the CheckTransaction class is not shown - you must decide where to put this class.



These transaction objects will be used both to make financial requests of a **BankAccount** and to keep a record of those requests. The following code segment indicates how transactions will be used.

```

BankAccount account = new BankAccount("Jim", 12345);
account.post(new DepositTransaction(1000));
account.post(new WithdrawalTransaction(200));
account.post(new CheckTransaction(50, 101));

```

In this hands-on you are limited to implementing the Transaction hierarchy. This involves the creation of the **Transaction**, **DepositTransaction**, **WithdrawalTransaction** and **CheckTransaction** classes. You will implement the required fields and their access methods. You will implement any required constructors.

1. Create a class called "Transaction" in the `banking.models` package. Which class should Transaction inherit from? Is Transaction a kind of Date, a kind of BankAccount, or a kind of Object?
2. Transaction should inherit from Object. Basically, a Transaction is not very much like any other kind of object. While a Transaction may contain a date, it is certainly not a Date.



3. Transactions have the fields **date** and **amount**. The date field should contain an instance of class `java.util.Date`. The amount field should be a double. Use the Java Editor to add these fields (make them private) and use eclipse tools provided in the Java editor to automatically create their getters (public) and setters (private). Why make the setter private? Because once a Transaction has been created, we do not want it to be modifiable. That is, our Transactions cannot be changed once they have been created.
4. Build a constructor for Transaction class that takes a double value for its parameter (subclasses can use this method). Have the constructor set the transaction's amount field to the provided value. Set the date of the Transaction to the current date (an instance of class `Date` that represents the current date).
5. Create subclasses of Transaction called "DepositTransaction" and "WithdrawalTransaction". What should the new hierarchy look like? Are both classes direct subclasses of Transaction? Should we have any fields in these classes?  
Use eclipse tools provided in the Java editor to automatically create any required constructors for you.
6. Create the "CheckTransaction" class. What should the new hierarchy look like? Should we directly subclass Transaction? Perhaps another class is more appropriate?  
A CheckTransaction needs a `checkNumber` field (int). Create this as a private field along with a public getter and a private setter (CheckTransactions should not allow their check number to change once they have been created).

Use eclipse tools provided in the Java editor to automatically create any required constructors for you. It has this signature:

```
public CheckTransaction(double amount)
```

Modify this constructor so that it has the signature below and implement the constructor body. Don't forget to assign the `checkNumber`. *Hint: chain constructors.*

```
public CheckTransaction(double amount, int checkNumber)
```

7. Create a `toString()` method in the Transaction class. Has it display the type and amount of the transaction (do not hard code the class name, discover the class name dynamically)? This method will be inherited by all Transaction subclasses. Override `toString()` in the CheckTransaction class. Reuse the Transaction version of `toString()` and append the check number. For example:

```
new DepositTransaction(50) ⇒ "a $50  
banking.models.DepositTransaction"  
new WithdrawalTransaction(27) ⇒ "a $27  
banking.models.WithdrawalTransaction"  
new CheckTransaction(50, 101) ⇒ "a $50  
banking.models.CheckTransaction #101"
```

Note that at this point, the full package name is included with the class name. With this in mind, try to reuse as much code as possible. How much code can be reused? Test your methods.

*Hint: You can ask an object for its class and that a class can be asked for its name.*

8. Create a new Junit test case named `TestingTransactions` in the `banking.models.test` package. Test your newly created classes. You can use following code block:

```
@Test
```

```
void testTransactions() {  
    Transaction deposit = new DepositTransaction(500.0);  
    System.out.println(deposit);  
  
    Transaction withdrawal = new WithdrawalTransaction(50.0);  
    System.out.println(withdrawal);  
  
    Transaction check = new CheckTransaction(500, 123);  
    System.out.println(check);  
  
    CheckTransaction check2 = (CheckTransaction) check;  
    assertTrue(check2.getCheckNumber()==123);  
}
```

We will improve this test method later

### Extra Work

1. Extend your toString() method for Transactions so that it prints the class name without the full package name. In particular, transactions should print as follows:

```
new DepositTransaction(50) ⇒ "a $50 DepositTransaction"  
new WithdrawalTransaction(27) ⇒ "a $27 WithdrawalTransaction"  
new CheckTransaction(50, 101) ⇒ "a $50 CheckTransaction #101"
```

Make sure that any future Transaction subclasses will automatically print correctly.

2. The BankAccount class has two constructors with the following signatures:

```
public BankAccount() {...}  
public BankAccount(String owner, int accountNumber) {...}
```

Create and Inspect a new BankAccount using each of the constructors. What are the values for owner and accountNumber when you use the no-arguments constructor? (They should be null and 0 respectively).

Modify the no-argument constructor so that default values for owner and accountNumber are hardcoded as parameters to a chained constructor call i.e. call the BankAccount(String, int) constructor from the BankAccount() constructor.

3. Re-run your test method.

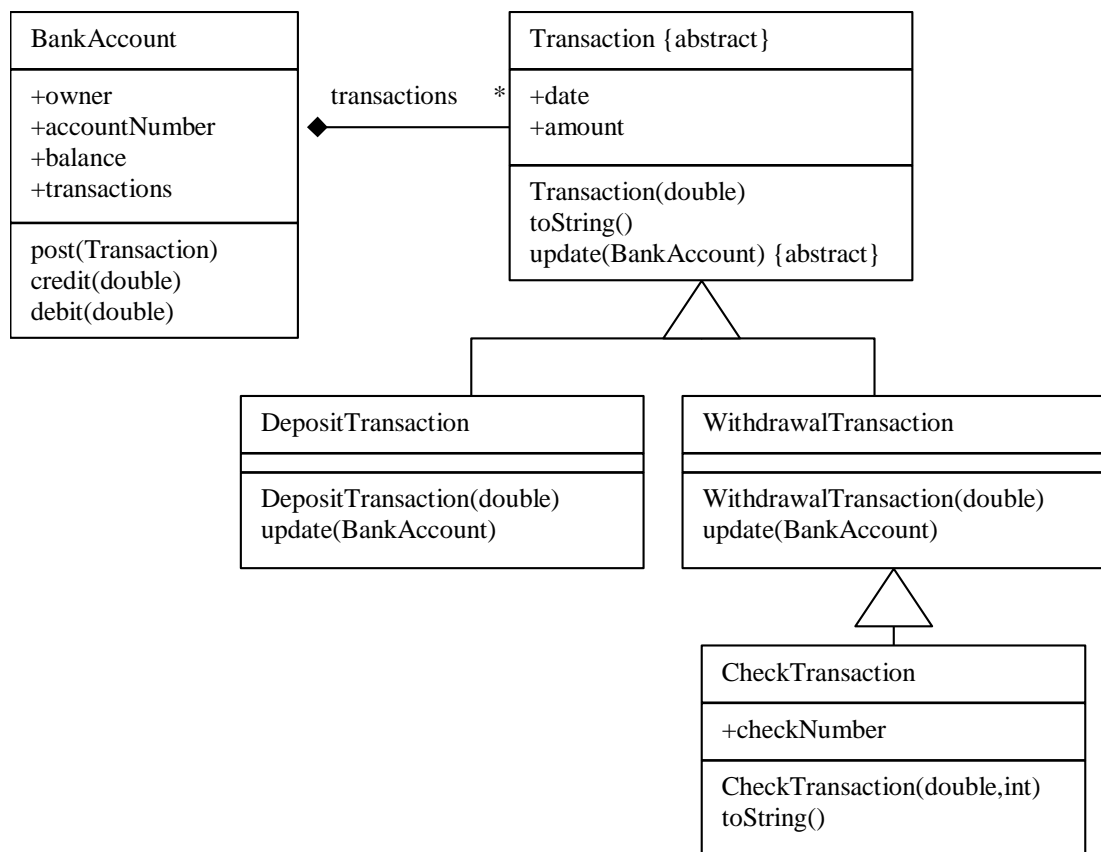
## Lab 07: Abstract Classes

### Objective:

The object model for our banking system has been extended to include Transaction objects. A transaction object keeps track of the kind of transaction (deposit, withdrawal or check) as well as the date and amount of the transaction.

In this exercise you will implement the posting of Transactions to the BankAccount.

BankAccount and Transactions *collaborate*. This collaboration is achieved by having the BankAccount and Transaction exchange messages.



These transaction objects will be used both to make financial requests of a BankAccount and to keep a record of those requests. The following code segment indicates how transactions are used.

```
BankAccount account = new BankAccount("Jim", 12345);
account.post(new DepositTransaction(1000));
account.post(new WithdrawalTransaction(200));
account.post(new CheckTransaction(50, 101));
```

1. The end user should not be able to create Transactions, they should be creating DepositTransaction, WithdrawalTransaction and CheckTransaction. To enforce this, Transaction has been marked as an abstract class. Make Transaction an abstract class.
2. Create a new Junit test case named TestingTransactions.

Test your change in the testcase using the following code sample. What happens?

```
new Transaction(50.0)
```

2. In the TestCase, execute the following code. Note that you can store an instance of the subclass (e.g. DepositTransaction) in a variable of the superclass type (i.e. Transaction).

```
Transaction t1 = new DepositTransaction(1000);
System.out.println(t1.toString());
Transaction t2 = new CheckTransaction(50, 101);
System.out.println(t2.toString());
```

What appears in the Console?

With the CheckTransaction, which toString() is being called? Transaction::toString() or CheckTransaction::toString()?

3. In the TestCase, execute the following code:

```
Transaction t2 = new CheckTransaction(50, 101);
System.out.println(t2.getCheckNumber());
```

What happens when you execute this code?

Why?

4. Modify the test method in your TestingTransactions class so that it is transaction-oriented. This sample method below will not compile initially, you have not implemented the post() method yet.

```
@Test
void testTransactions() throws Exception {
    Transaction depositTrx = new DepositTransaction(500.0);
    System.out.println(depositTrx);

    Transaction withdrawalTrx = new WithdrawalTransaction(50.0);
    System.out.println(withdrawalTrx);

    Transaction checkTrx = new CheckTransaction(100, 123);
    System.out.println(checkTrx);

    BankAccount account = new BankAccount("Jim", 12345);
    account.post(depositTrx);
    account.post(withdrawalTrx);
    account.post(checkTrx);
    assertTrue(account.getBalance()==350);
}
```

Implement a stubbed version of the post() method as shown below. This belongs in the BankAccount class.

```
public void post(Transaction transaction) {
    // to be implemented
}
```

How should the post method be implemented? Its implementation depends on the type of the *transaction* parameter. If *transaction* is a DepositTransaction we need to credit the account, if its a WithdrawalTransaction we need to debit the account. There are three ways we can implement this behavior.

One solution would be to overload the post method, creating different post methods for each Transaction type. e.g. post (DepositTransaction) and post(WithdrawalTransaction). This solution will work but creating families of overloaded methods is discouraged as it causes problems with maintenance. Consider, if we added more Transaction subclasses we would need to keep changing the BankAccount class, overloading even more post methods.

Another solution is shown below:

```
public void post(Transaction transaction) {  
    if(transaction instanceof DepositTransaction)  
        this.credit(transaction.getAmount());  
    else if(transaction instanceof WithdrawalTransaction)  
        this.debit(transaction.getAmount());  
    else if ... // handle other subclasses  
}
```

This solution will work; however, this style of programming is not recommended. It is considered bad form in OO languages to write case statements based on the type of objects. It also has the same maintenance problems as the first solution. Adding more Transaction subclasses would require changes

The third option is to delegate the operation using polymorphism. The parameter *transaction* knows its type. The object in the *transaction* parameter knows what it needs to do to the BankAccount e.g. DepositTransactions will credit the BankAccount, WithdrawalTransactions will debit the BankAccount. So, the solution is to send a message to *transaction* and let polymorphism resolve the typing problem. Remember that regardless of the variable type, method lookup always begins in the class of the receiver. What we need is a common message that DepositTransaction and WithdrawalTransaction both respond to. Let's call this method update (), and pass it the BankAccount to credit/debit.

5. Implement the update method in the DepositTransaction class. It should have the following signature and credit the transaction amount to the BankAccount.

```
public void update(BankAccount account)
```

6. Implement the update method in the WithdrawalTransaction class. It should have the following signature and debit the transaction amount from the BankAccount.

```
public void update(BankAccount account)
```

7. Implement the body of the post method so that it sends the update method to the transaction.

```
public void post(Transaction transaction) {  
    transaction.update(this);  
}
```

Unfortunately, as it stands, this code will not compile correctly. Why? Save the method anyway.

8. The post method will not compile because the Transaction class does not declare an update(BankAccount) method. This is the same problem as Q3 above where you discovered that storing an object in a variable of its superclass type (Transaction) prevented access to methods not declared in the variables type (e.g. getCheckNumber()).

To solve this problem we declare the `update(BankAccount)` method in the `Transaction` class. You make this method abstract to force subclasses to implement it.

```
public abstract void update(BankAccount account);
```

The `post()` method now compiles correctly as `Transaction` now has an `update` method declared.

9. Execute the `TestCase`. What is the balance of the `BankAccount` it returns? Is this the value you expect?

## Lab 08: Interfaces

### Objective:

One popular standard for interfaces has interfaces fully specify the public methods for a corresponding implementation class. If we were to follow this standard with the banking system, we would have an interface named `BankAccount` with a corresponding implementation class called `BankAccountImpl`. Similar standards use different names but are conceptually similar.

This standard does offer a lot of flexibility for future development (distributed computing being one example) but does have require that you maintain two types. When you change the method signature, you must change it in two places: the interface and the implementation class.

The standard does also involve usage conventions. By convention, the implementation class should be used sparingly — typically for creation of objects only. All variables and parameter types should be set to interface types. The `Employee` interface, for example, specifies the public methods for an employee object. Any variable that might contain objects that implement the interface should be declared to be of the interface type. The following code, for example, demonstrates how to use employees:

```
Store store = StoreTester.exampleStore();
Iterator iter = store.getItems().iterator();
...
```

### Bank Account Updater

In this session you will modify your transactions to make use of interfaces. As they are currently implemented, a `Transaction` is an object that will update a `BankAccount` (via the `update(...)` method). It is possible that other kinds of objects may be required to make (and record) changes to `BankAccounts`. In particular, a `BankAccount` should likely keep track of changes to the owner name.

1. Create an interface called `BankAccountUpdater`. Add the following method definitions to your interface:

```
public void update(BankAccount account);
public Date getDate();
```

2. Revisit your `Transaction` class and modify it so that it implements the `BankAccountUpdater` interface.
3. Revisit `BankAccount`'s `post(...)` method. Change the parameter type so that this method accepts an object that implements the `BankAccountUpdater` interface.
4. Test your changes to confirm that everything still works.

### Bonus

Instances of the `OwnerChange` class effect and record owner name changes for accounts.

1. Create a new class called `OwnerChange` that implements the `BankAccountUpdater` interface. This class should have three fields, `oldName`, `newName`, and `date`. This class essentially represents the change of the owner name in a specified `BankAccount`. Provide methods to get and set the values of these fields.
2. Provide a constructor that takes the new owner name as a single parameter. In your constructor, set the date for the instance to the current date.

3. Implement the `update(...)` method as specified by the interface. This method should save the `BankAccount`'s old owner in the `oldName` field and set its new value to the contents of the `newName` field.
4. Confirm that `BankAccount`'s `post(...)` method works by posting an `OwnerChange`. The following code demonstrates how this should work. You add these codes into your `TestCase`:

```
...
BankAccount account = new BankAccount("Jennifer", 12345);
account.post(new OwnerChange("Wayne"));
assertTrue(account.getOwner().equals("Wayne"));...
```

## Lab 09: Collections

### Objective:

In this lab you will examine the creation, conversion, and sequencing of collections and gain experience using arrays, Lists, HashMaps, Iterators and Strings

1. Build a method that, given a List of `BankAccount` objects, will provide an equivalent List containing just the names of the owners. What class should implement this method?  
Create a List of `BankAccounts` and use it to test your code.
3. Now you are ready to store transactions in your bank account. To do this, you will have to add a list typed field called "transactions" to the `BankAccount` class. This field will contain only transactions. (you can use **instanceof** keyword). Be sure to add accessing and modifying methods and ensure that new instances of `BankAccount` have an empty List. Modify the `post(...)` so that all posted transactions are stored by the account.
4. Implement the following methods for `BankAccount`:

```
public Transaction largestTransaction( );
public double averageTransactionAmount( );
public List allTransactionsLargerThan(double amount);
```

Create a test method that creates an account with a large number of transactions. Use this method to test the above.

### Bonus

1. Revisit your `BankAccountUpdater` interface and add the following method definition:

```
public boolean isTransaction();
```

Provide implementations of this method in your `Transaction` and `OwnerChange` classes that answer true and false, respectively.

2. Change the name and type of the transactions field to "updates" and `BankAccountUpdater`. Create a get method for the updates List. Update your constructors to make use of the updates field.

```
private ArrayList<BankAccountUpdater> updates=new
ArrayList<BankAccountUpdater>();
```



```
public BankAccount(String owner, int accNumber) {  
  
    super();  
    this.owner = owner;  
    this.accountNumber = accNumber;  
    this.updates = new ArrayList<BankAccountUpdater>();  
}
```

3. Revisit your post method and modify it so that updates filed should keep the track of every kind of Updater objects

```
public void post(BankAccountUpdater updater) throws Exception {  
  
    updates.add(updater);  
    updater.update(this);  
}
```

4. Revisit your getTransactions( ) method and modify it so that it answers a List that contains only those objects in the updates field that are instances of Transaction (or one of its subclasses). With this modification, any methods that use the getTransactions( ) method should work again.

Have your getTransactions( ) method enumerate the contents of the updates List and collect those entries that are instances of Transaction (or one of its subclasses) into a new List. Hint: use the isTransaction( ) method to determine if an object in your List is an instance of Transaction or not.

```
public List<Transaction> getTransactions(){  
    List<Transaction> transactions=new ArrayList<Transaction>();  
    for (BankAccountUpdater updater : updates) {  
        if(updater.isTransaction()) {  
            transactions.add((Transaction)updater);  
        }  
    }  
    return transactions;  
}
```

## Lab 10: Exception Handling

### Objective:

The objective of this lab is to work with exceptions. In this lab you will learn how to throw and catch exceptions.

### Lab:

In the banking application, we might want to throw an exception when someone tries to withdraw more money than is held in an account. While there are several Exception classes in the library, none of them are suitable for this situation.

1. To solve the problem, create an exception class called "InsufficientFundsException". Implement a zero-argument constructor and a constructor that takes a String parameter. The latter constructor allows the developer to store a message inside an instance of InsufficientFundsException.
2. Rewrite the debit(...) method in the BankAccount class so that it throws the InsufficientFundsException if the debit amount exceeds the account's balance.

In our application, a debit(...) method is invoked as part of a BankAccount object's response to a post(...) method. What other methods have to be rewritten, and how must they be rewritten, if we want any method that invokes the post(...) method to ultimately end up handling the InsufficientFundsException exception? Specifically, the following code should handle the exception:

```
BankAccount account = new BankAccount();
System.out.println("Attempting withdrawal...");
try {
    account.post(new WithdrawalTransaction(500.0));
    System.out.println("Transaction successful!");
} catch (InsufficientFundsException exception) {
    System.out.println("Transaction Failed: Insufficient Funds!");
}
```

Note that changing the exceptions thrown by a method can have a significant impact on other methods that use it.

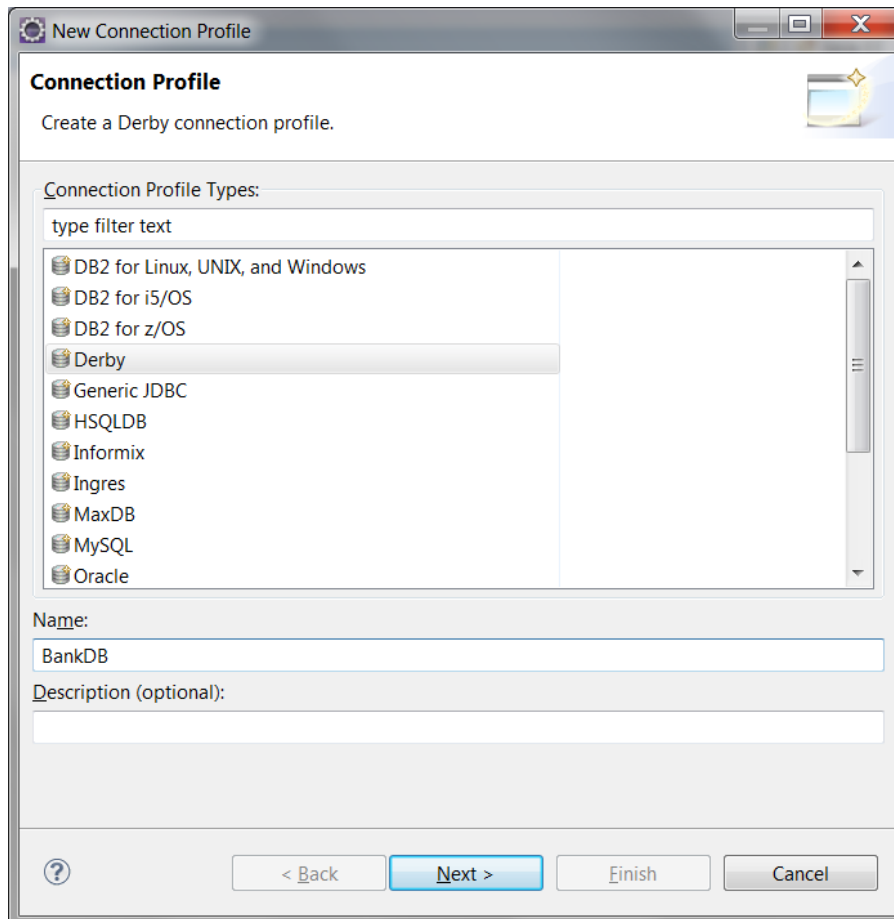
## Lab 11: Creating Database

We will use the Derby database in this session to practice with database access. Derby is a client/server database system implemented in Java.

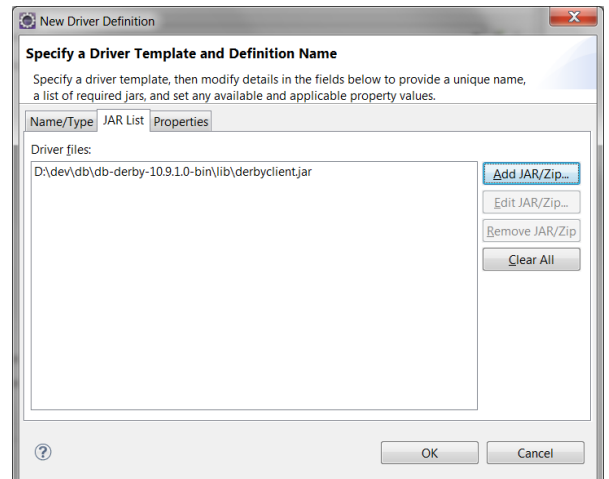
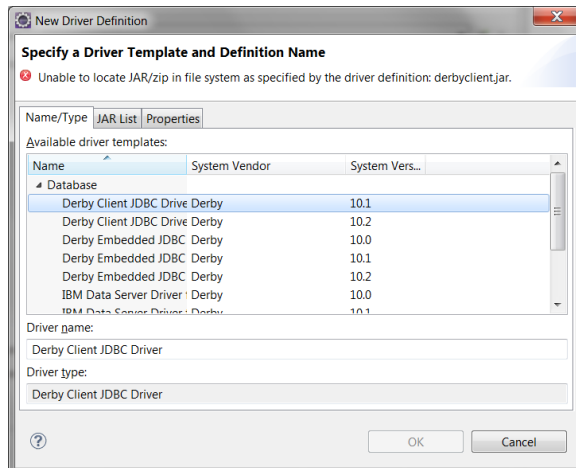
1. Start your Derby server. The Derby server can be started via startServer.cmd in the derby installation bin directory.

**Note: If you get Security error while starting derby server use "noSecurityManager" option:**  
**>startNetworkServer -noSecurityManager**

1. In eclipse, switch to “Database Development” perspective.
2. Right Click over the “Database Connections” and select **New..**
3. Select **Derby** as Connection Profile Type and use BankDB as name:

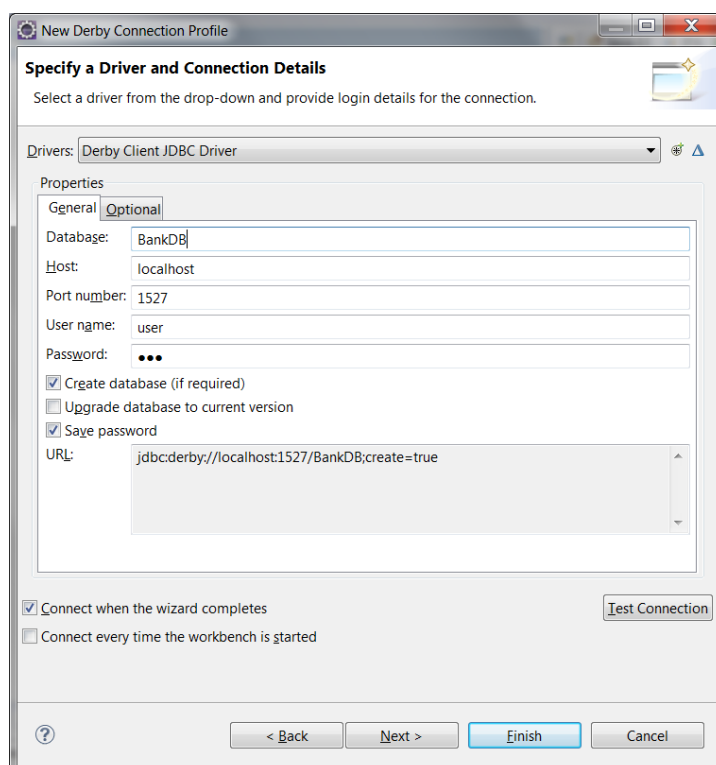


4. Click Next button. Next screen is used for specifying the derby connection driver. Click on the “New Driver Definition” button and select Derby 10.1 as database vendor. After that switch to Jar List tab and add derbyclient.jar from derby installation lib folder.

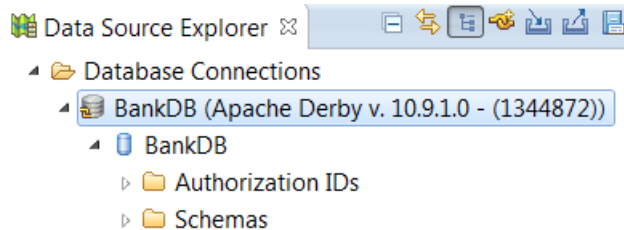


5. Click OK button.
6. Change database name as BankDB . Use (user, 123) as (username-password). Check “save password”. Click Test Connection button. You should get “ping succeeded” message. Copy URL link. We will use this link later

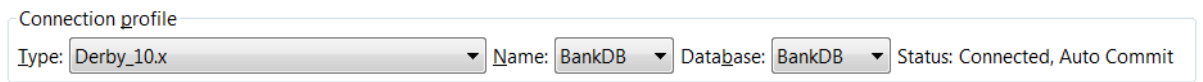
`jdbc:derby://localhost:1527/BankDB;create=true`



- Click Finish button. You should see your connection in the DataSource Explorer window



- In your banking application project create a new file named script.sql. This file will open with the **SQL File Editor**. Select connection type, profile name and database name.



- Write a sql script for creating table

```
CREATE TABLE BANK_ACCOUNT (  
  OWNER CHAR(40),  
  ACCOUNT_NUMBER INTEGER,  
  BALANCE DOUBLE  
)
```

Select sql lines and click over the **Execute Selected Text** from popup menu. Your table will be created.

- Switch to Database Development perspective and open BankDB connection. Right click over the BANK\_ACCOUNT table under the USER schema and click Edit. Enter sample data for Bank\_Account table and click Save button.

OWNER [CHAR(40)]	ACCOUNT_NUMBER [INTEGER]	BALANCE [INTEGER]
CAN	123	100
CEM	12981	881
CANAN	818	771
<new row>		

11. Now your database is ready.

## Lab 12: Working with JDBC

### Goals

- To learn how to use database connection pools and datasources and
- Use JDBC API to access databases

### What you should know

Before you complete this set of exercises, you should understand the following:

- JDBC fundamentals
- How to write JDBC statements for database operations.

1. Open your pom.xml and add dependency for derbyclient:

```
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derbyclient</artifactId>
  <version>10.14.2.0</version>
</dependency>
```

2. Create new package for your database access classes. Name it as `banking.db`.
3. Create a `BankAccountDAO` class. This class will be responsible for creating-managing connections and getting - saving data from database. Add two methods `getConnection()`, `closeConnection(Connection)` into this class.

```
private Connection createConnection() throws SQLException {
    Connection connection = DriverManager.getConnection(
        "jdbc:derby://localhost:1527/BankDB", "user",
        "123");
    return connection;
}
```

```
private void closeConnection(Connection conn) throws SQLException {
    if (conn != null) {
        conn.close();
    }
}
```

```
}
```

4. Add readAllAccounts method for reading all accounts from database. This method should return list of accounts.

```
public List<BankAccount> readAllAccounts() throws SQLException {  
    List<BankAccount> accounts = new ArrayList<BankAccount>();  
    Connection conn = createConnection();  
    Statement statement = conn.createStatement();  
    ResultSet result = statement  
        .executeQuery("SELECT OWNER, ACCOUNT_NUMBER,  
BALANCE FROM BANK_ACCOUNT");  
    while (result.next()) {  
        BankAccount acc = new BankAccount();  
        acc.setOwner(result.getString("owner"));  
        acc.setAccountNumber(result.getInt("ACCOUNT_NUMBER"));  
        acc.setBalance(result.getDouble("BALANCE"));  
        accounts.add(acc);  
    }  
    closeConnection(conn);  
    return accounts;  
}
```

5. Add a new method for saving account to database

```
public void saveAccount(BankAccount acc) throws SQLException{  
    Connection conn = createConnection();  
    PreparedStatement statement = conn.prepareStatement("INSERT  
INTO BANK_ACCOUNT(OWNER, ACCOUNT_NUMBER,BALANCE) VALUES(?,?,?)");  
    statement.setString(1, acc.getOwner());  
    statement.setInt(2, acc.getAccountNumber());  
    statement.setDouble(3, acc.getBalance());  
    statement.executeUpdate();  
    closeConnection(conn);  
}
```

6. Write a test class and test your methods

```
class TestDB {  
  
    @Test  
    void test() throws SQLException {  
        BankAccount account1 = new BankAccount("Jim", 12345);  
        BankAccount account2 = new BankAccount("Alice", 238);  
  
        BankAccountDAO dao=new BankAccountDAO();  
        List<BankAccount> listBeforeInsert=dao.readAllAccounts();  
        dao.saveAccount(account1);  
        dao.saveAccount(account2);  
        List<BankAccount> listAfterInsert=dao.readAllAccounts();
```

```
        assertTrue(listAfterInsert.size()==listBeforeInsert.size()+2);  
    }  
}
```

## Lab 13: Annotations

We're going to create three custom annotations with the goal of serializing an object into a JSON string.

- We'll use the first one on the class level, to indicate to the compiler that our object can be serialized.
- we'll apply the second one to the fields that we want to include in the JSON string.
- we'll use the third annotation on the method level, to specify the method that we'll use to initialize our object.

1. In the `banking.custom.annotations` package create a new annotation as follows:

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface JsonSerializable {  
  
}
```

This annotation has runtime visibility, and we can apply it to types (classes). Moreover, it has no methods, and thus serves as a simple marker to mark classes that can be serialized into JSON.

2. create our second annotation, to mark the fields that we are going to include in the generated JSON:

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.FIELD)  
public @interface JsonElement {  
    public String key() default "";  
}
```

The annotation declares one String parameter with the name “key” and an empty string as the default value.

When creating custom annotations with methods, we should be aware that these methods must have no parameters and cannot throw an exception. Also, the return types are restricted to primitives, String, Class, enums, annotations, and arrays of these types, and the default value cannot be null.

3. Before serializing an object to a JSON string, we want to execute some method to initialize an object. For that reason, we're going to create an annotation to mark this method:



```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Init {
}
```

We declared a public annotation with runtime visibility that we can apply to our classes' methods.

- Now, let's see how we can use our custom annotations. We have an object of type `BankAccount` that we want to serialize into a JSON string. This type has a method that capitalizes the first letter of the first and last names. We'll want to call this method before serializing the object:

```
@JsonSerializable
public class BankAccount implements Comparable<BankAccount> {
    @JsonElement
    private String owner;
    @JsonElement
    private int accountNumber;
    @JsonElement(key = "accountBalance")
    private double balance;
    ...

    @Init
    private void initOwner() {
        this.owner = this.owner.substring(0, 1).toUpperCase()
            + this.owner.substring(1);
    }
    ...
}
```

- By using our custom annotations, we're indicating that we can serialize a `BankAccount` object to a JSON string. In addition, the output should contain only the owner, accountNumber, and balance fields of that object. Moreover, we want the `initOwner()` method to be called before serialization. By setting the key parameter of the `@JsonElement` annotation to "accountBalance", we are indicating that we'll use this name as the identifier for the field in the JSON output.
- Now, we're going to see how to take advantage of them by using Java's Reflection API. Create a new class named `ObjectToJsonConverter`. The first step will be to check whether our object is null or not, as well as whether its type has the `@JsonSerializable` annotation or not:

```
private void checkIfSerializable(Object object) {
    if (Objects.isNull(object)) {
        throw new JsonSerializationException("Can't serialize a null
```

```
object");
    }

    Class<?> clazz = object.getClass();
    if (!clazz.isAnnotationPresent(JsonSerializable.class)) {
        throw new JsonSerializationException("The class " +
            clazz.getSimpleName() + " is not annotated with JsonSerializable");
    }
}
```

Add you need to add a custom exception class:

```
public class JsonSerializationException extends RuntimeException {

    private static final long serialVersionUID = 1L;

    public JsonSerializationException(String message) {
        super(message);
    }
}
```

7. Then, we look for any method with @Init annotation, and we execute it to initialize our object's fields:

```
private void initializeObject(Object object) throws IllegalAccessException,
    IllegalArgumentException, InvocationTargetException {
    Class<?> clazz = object.getClass();
    for (Method method : clazz.getDeclaredMethods()) {
        if (method.isAnnotationPresent(Init.class)) {
            method.setAccessible(true);
            method.invoke(object);
        }
    }
}
```

The call of `method.setAccessible(true)` allows us to execute the private `initOwner()` method. After the initialization, we iterate over our object's fields, retrieve the key and value of JSON elements, and put them in a map. Then, we create the JSON string from the map:

```
private String getJsonString(Object object) throws IllegalArgumentException,
    IllegalAccessException {
    Class<?> clazz = object.getClass();
    Map<String, String> jsonElementsMap = new HashMap<>();
    for (Field field : clazz.getDeclaredFields()) {
        field.setAccessible(true);
        if (field.isAnnotationPresent(JsonElement.class)) {
            jsonElementsMap.put(getKey(field), field.get(object)+"");
        }
    }

    String jsonString = jsonElementsMap.entrySet()
        .stream()
```

```

        .map(entry -> "\"" + entry.getKey() + "":"\" + entry.getValue() +
        "\"" )
        .collect(Collectors.joining(","));
    return "{" + jsonString + "}";
}

```

Again, we used `field.setAccessible(true)` because the `Person` object's fields are private.

8. Our JSON serializer class combines all the above steps:

```

public class ObjectToJsonConverter {
    public String convertToJson(Object object) throws
    JsonSerializationException {
        try {

            checkIfSerializable(object);
            initializeObject(object);
            return getJsonString(object);

        } catch (Exception e) {
            throw new JsonSerializationException(e.getMessage());
        }
    }
}

```

```

private String getKey(Field field) {
    String value = field.getAnnotation(JsonElement.class)
        .key();
    return value.isEmpty() ? field.getName() : value;
}

```

9. Finally, we run a unit test to validate that our object was serialized as defined by our custom annotations:

```

class TestAnnotations {
    @Test
    public void givenObjectNotSerializedThenExceptionThrown() throws
    JsonSerializationException {
        Object object = new Object();
        ObjectToJsonConverter serializer = new ObjectToJsonConverter();
        assertThrows(JsonSerializationException.class, () -> {
            serializer.convertToJson(object);
        });
    }

    @Test
    public void givenObjectSerializedThenTrueReturned() throws
    JsonSerializationException {
        BankAccount account = new BankAccount("jim", 12345);
        account.credit(1000.0);
        account.debit(50.0);

        ObjectToJsonConverter serializer = new ObjectToJsonConverter();
        String jsonString = serializer.convertToJson(account);
    }
}

```

```
System.out.println(jsonString);

assertEquals("{\"owner\":\"Jim\",\"accountNumber\":\"12345\",\"accountBalance\":\"950.0\"}", jsonString);
    }
}
```

## Lab 14: Mocking

### Stub Testing

Stub objects are objects that are injected into your object and are used to replace real objects in test situations. Stub should only be scanned for testing. Stub objects are easy to implement and reusable. However, any change to the interface requires to stub to be updated and that consumes time.

---

### Mock Testing

The main idea of mock is splitting test method into three parts: setting, executing and verifying. First step is creating the mock. Second step is determining the mock behavior, setting result for particular parameter. Final step is verify step which is the step that verifies if the functionality getting met for the given parameters. Mock objects are easy to create and maintained thanks to frameworks like Mockito.

---

Following example shows that how to use Mock Testing.

### Exercise: Mock Testing

1. To use Mockito framework, open `pom.xml` file and add following code inside the dependencies.

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>2.23.4</version>
    <scope>test</scope>
</dependency>
```

4. The balance should be changed to other currency units. Because of that, we will need class of DAO. Let's create `CurrencyDAO` interface inside `banking.dao` package.

```
public interface CurrencyDAO {
    public double getTLRate(String currencyCode);
}
```

5. Add the "CurrencyDAO" dependency to the "BankAccount" class.

```
private CurrencyDAO currencyDAO;

public CurrencyDAO getCurrencyDAO() {
```

```
        return currencyDAO;
    }

    public void setCurrencyDAO(CurrencyDAO currencyDAO) {
        this.currencyDAO = currencyDAO;
    }
}
```

**6. Add the method to convert the balance amount to other currencies in the "BankAccount" class.**

```
public double convertBalance(String currencyCode){
    double rate=getCurrencyDAO().getTLRate(currencyCode);
    return (int)(getBalance()/rate*100)/100.0;
}
```

**7. Add static import to TestBankAccount.java.**

```
import static org.mockito.Mockito.*;
```

**8. Add a test method for testing currency converter**

```
@Test
void testCurrencyConverter() {
}
}
```

**9. Create an account with has a balance equals of 1000.**

```
BankAccount account = new BankAccount("Canan",1234);
CurrencyDAO mockedCurrencyDAO = mock(CurrencyDAO.class);
account.post(new DepositTransaction(1000));
```

**9. Let's record Mock Object.**

```
when(mockedCurrencyDAO.getTLRate("USD")).thenReturn(5.34);
when(mockedCurrencyDAO.getTLRate("EUR")).thenReturn(6.06);
account.setCurrencyDAO(mockedCurrencyDAO);
```

**10. Add the assertions and be sure about the CurrencyDAO is called twice.**

```
assertEquals(187.26,account.convertBalance("USD"));
assertEquals(165.01,account.convertBalance("EUR"));
verify(mockedCurrencyDAO).getTLRate("EUR");
verify(mockedCurrencyDAO).getTLRate("USD");
```

**11. After these steps, the method should like below**

```
@Test
public void testCurrencyConverter() {
    Account account = new Account("Canan","1234");
    CurrencyDAO mockedCurrencyDAO = mock(CurrencyDAO.class);
    account.post(new DepositTransaction(1000));
    when(mockedCurrencyDAO.getTLRate("USD")).thenReturn(5.34);
    when(mockedCurrencyDAO.getTLRate("EUR")).thenReturn(6.06);
    account.setCurrencyDAO(mockedCurrencyDAO);
}
```

```
assertEquals(187.26,account.convertBalance("USD"),0);;  
assertEquals(165.01,account.convertBalance("EUR"),0);  
verify(mockedCurrencyDAO).getTLRate("EUR");  
verify(mockedCurrencyDAO).getTLRate("USD");  
}
```

12. If we run the test now, test will pass successfully