JVA-007
Development on the JavaSE7 platform
Part 1

Workbook

(edition 1.3, 30.05.2015)

JVA-007
Development on JavaSE7 platform
Part 1

Objectives:	Learn the JavaSE platform
Target audience:	Junior Java Developers
Duration:	40 hours / 5 days
The authors:	Dmitry Kalashnikov (Kalashnikov.dmitry@gmail.com) Eugene Krivosheyev (eugene.krivosheyev@gmail.com) Cătălin Tudose (ftudose@luxoft.com)
Product version:	JavaSE 7

Table of Contents

Exercise 1
Installing and configuring runtime environment

Objectives

Install and configure runtime environment.

Tasks

1. You should download and install JDK 7 from the www.oracle.com website.
2. You should download and install Eclipse from the www.eclipse.org website.

Exercise 2
Analyzing and initiating the first Java application

Objectives

Create a primitive HelloWorld Java application. Learn to compile and launch it using JDK tools. Master the skills of writing and generating JavaDoc documentation.

Tasks

1. Create a **`HelloWorld`** Java class that displays a string.
2. Compile a class using the **javac** utility.
3. Launch application using the **java** statement.
4. Modify the **`HelloWorld`** class, create JavaDoc comments for a class and the **`main()`**method. Generate JavaDoc documentation using JDK **javadoc**.
5. Add a function that receives two arguments and displays their sum. Invoke it from **`main()`**, print the result to console.

Detailed guidelines

1. Write the **`HelloWorld`** class definition in any editor (Notepad++, WordPad). Create the **`HelloWorld.java`** file. A class should be declared **`public class HelloWorld{}`**.
2. Add the **`public static void main(String[] arg)`**method. This method will be invoked at application launch, i.e. when JVM is created.
3. Add output to console in the **`main(): System.out.println("Hello, world")`**method.
4. Compile the **`HelloWorld.java`** class. Statement: **`javac HelloWorld.java`**. Make sure that PATH environment variable contains path to Java JDK (for example **`C:\Program Files\Java\jdk1.7.0_71\bin\`**). Verify that the **`HelloWorld.class`** file has been generated.
5. Launch the application: **`java HelloWorld`**. Note that you indicate class name rather than compiled file name.
6. Modify the application, add JavaDoc comments to class and the **`main()`**method

```
/**
 * This is the simplest class which prints the "Hello, world" message.
 * @author Peter Pan
*/
public class HelloWorld {
	/**
	 * Main methods which is executed by JVM and prints the message.
	 * @param args command line arguments
	 */
	public static void main(String[] args) {
		...
	}
}
```

7. Launch the javadoc utility**: `javadoc HelloWorld.java`**.
8. Make sure that the documentation is generated, launch **`index.html`**.
9. Add a function that receives two arguments and displays their sum. Invoke it from **`main()`**, print the result to console.

Exercise 3
Analysis of data domain in arbitrary notation

Objectives

Learn to distinguish key entities and activities describing certain data domain. Determine interrelations, whole-and-part relations between entities.

Tasks

1. Shape data domain describing some bank: select key entities and list correlations. The notation is arbitrary.
2. Analyze functional specifications of application.

Data domain description

A number of employees work in a bank, each works in some department. Employees take a certain place in a chain of positions. Each employee has one address and several telephone numbers. Each employee works at one or several projects. The project has a manager. The project has execution periods, delivery break points.

Detailed guidelines

1) Describe the attributes
2) Describe methods
3) Determine the inheritance hierarchy
4) Define an interface
5) Determine the behavior of polymorphic methods
6) Determine the hidden methods (encapsulation)
7) Determine the relationship part / whole

Description of functional specifications

FS01. I am an administrator and I can create an employee, indicate his or her data, and this information will be saved.
FS02. I am an administrator and I can create a project, indicate some project data and an employee in charge of this project (manager), and this information will be saved.
FS03. I am a client and I can request information about employees to look over the information about each of them.
FS04. I am a client and I can request information about projects to look over the information about each project.

Exercise 4
Designing an application with UML

Objectives

Learn to design an application kernel – implement class diagrams in UML, taking into account dependency relations between classes, as well as generalization relationship. Design shall be performed with regard to object-oriented design principles.

Tasks

1. Design the kernel of an application in UML.
2. Review the resulting design for correspondence to OOD key rules and principles.

Detailed guidelines

Choose one of the following tasks for UML design:

1) **University**: create a system for accounting students, attendance, and academic achievements. Optional: it must contain reviews of teachers.
2) **Shop**: should keep information about the goods and sales of the goods. Optional: The system should provide discounts for regular customers.
3) **Clinic**: to create a system for the appointment to the doctors, to the time of the doctors. Optional: keeping patient cards by physicians.
4) **Airport**: you must manage the landing of the aircrafts, with their distribution in time and runways. Optional: notifications about flight delays to passengers, via SMS.
5) **Restaurant**: to create a system of accounting for the employment of tables, selection of dishes from the menu, bill payment. Optional: allow the reservation of tables.

Exercise 5
Class diagram in Java

Objectives

Translate the UML class diagram implemented at exercise 4 into a Java program.

Tasks

1. Translate your class diagram to Java.
2. Implement method **`toString()`** in each class.
3. Initialize all classes in **`main`** method.
4. Print data.

Exercise 6
Primitives

Objectives

Understand Java primitive data types, casting, precision loss and side effects.

Tasks

Execute the Test program that works with primitives. Make analysis for the results that are displayed. Change the program and follow the new behavior.

Exercise 7
Arrays

Objectives

Understand the work with arrays in Java.

Tasks

Starting from the Arrays program:

1. Allocate a primitive array consisting of 5 elements.
2. Initialize the elements of the array with the numbers 1, 2, 3, 4, 5.
3. Print the length of the array.
4. Refer to the 6th element of the array, and explain the cause of the error.

Exercise 8
Object references

Objectives

Understand Java references, logical and physical equality.

Tasks

Execute the StringWorker program that processes **`String`**s. Make analysis for the results that are displayed. Change the program and follow the new behavior.

Exercise 9
Developing Java application from two classes

Objectives

Create a Java application comprising two classes. Learn to create objects, invoke methods, and execute the application.

Tasks

Use Eclipse to complete the task.

1. Create the **`HelloWorld`** Java class, define the **`main()`** method.
2. Create the **`MessagePrinter`** class and the **`String sayHello()`** method within it.
3. Create a **`MessagePrinter`** class instance within the **`main()`** method and invoke the **`sayHello()`** method.
4. Compile and launch the application.

Detailed guidelines

1. Create the **`HelloWorld`** Java class, define the **`main()`** method.
2. Create the **`MessagePrinter`** class and the **`String sayHello()`** method within it. The method must return the **`"Hello, world"`** message.
3. Create a **`MessagePrinter`** class instance in the **`main()`** method and invoke the **`sayHello()`** method.
4. Compile and launch the application. Make sure that the compiler automatically looks up and compiles dependent file **`MessagePrinter.java`**.

Exercise 10
Calculator

Objectives

Write a Java application to emulate a calculator having the 4 basic arithmetical operations.

Tasks

1. Create the Calculator application. The application must receive two operands and an operation as command-line arguments, for example: java Calculator 5 + 7.
2. The program displays an error message if the number of arguments is other than 3, if the operands are not numbers, if the operation is not +, -, *, /
3. After checking the necessary number of elements in the array **`String args[]`** you must convert arguments 1 and 3 strings to numbers, and use argument 2 as the operation symbol. To do this, use **`Double.parseDouble(String)`** and **`string.charAt(int)`**.

Exercise 11
Enums

Objectives

Understand Java enums, how they work and how to use them.

Tasks

Execute the Enums program that works with Java enums. Make analysis for the results that are displayed and follow the behavior.

Exercise 12
Working with packages and JAR files

Objectives

Learn to create java packages, archive a package to a JAR file, and specify **`classpath`**.

Tasks

Use Eclipse to complete the task.

1. Create **`com.luxoft.jva007`** and **`com.luxoft.jva007.presentation`** packages. Place **`HelloWorld.java`** into **`com.luxoft.jva007`** and **`MessagePrinter.java`** into **`com.luxoft.jva007.presentation`**.
2. Compile the classes.
3. Archive the **`com.luxoft.jva007.presentation`** package to a JAR file.
4. Delete **`MessagePrinter.class`**.
5. Launch the application using the **java** statement by specifying the created JAR file in **`classpath`**.

Detailed guidelines

1. Create the **`com`** package directory, it will contain the **`luxoft`** directory, this one will contain the **`jva007`** and this one in its turn will contain the **`presentation`** directory. Place the **`HelloWorld.java`** file to **`com\luxoft\jva007`**, and **`MessagePrinter.java`** file to **`com\luxoft\jva007\presentation`**.
2. With the help of the **`package`** directive declare packages in classes. Make required conversions in order to avoid compile-time errors.
3. Compile the classes.
4. Archive the **`com.luxoft.presentation`** package to a JAR file. To do this, execute the **`jar cf presentation.jar –C`** statement.
5. Delete the **`MessagePrinter.class`**.
6. Launch the application using the **`java`** command, specifying the created JAR file in **`classpath`**: **`java –classpath presentation.jar com.luxoft.jva007.HelloWorld`**.

Exercise 13
Inner and anonymous classes

Objectives

Understand Java inner and anonymous classes, how they work and how to use them.

Tasks

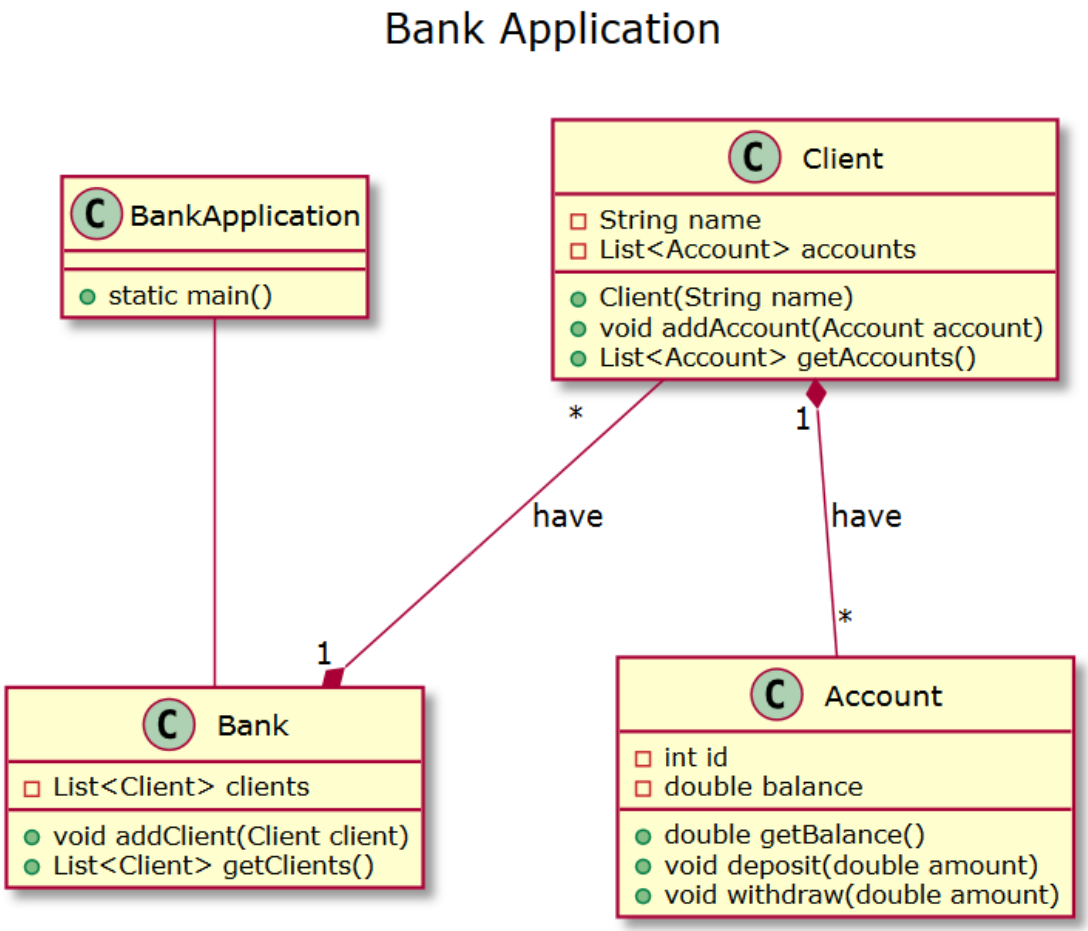
Execute the Dog program that works with Java inner and anonymous classes. Make analysis for the results that are displayed. Change the program and follow the new behavior.

Exercise 14
Bank Application: Creating Bank Application

Objectives

1. Learn principles of class creation and constructor definition.
2. Learn how to work with Java **`enum`**.
3. Understand the principles of polymorphism, overloading, and method overriding.
4. Learn to create abstract classes and interfaces.
5. Learn package-wise class division and the usage of access modifiers, as well as the **`final`** modifiers.

Model



Tasks

Task 1

1. Create the classes defined by the diagram. For the sake of simplicity, on this step, bank has one ten-client-array where one client can have one account.
2. The **`Account`** class has the constructor that accepts the **`id`** and the starting **`balance`** value, and also has **`deposit(x)`** method adding the **`x`** value to the balance and **`withdraw(x),`** that decreases the balance by **`x`**, in case **`balance >= x`**. Implement the **`getBalance()`** method that returns the current account balance value.

3. Create the **BankApplication** class. **BankApplication** creates a **Bank** class object and adds several new clients to the bank. Each new client must have one account with some starting value.
4. Implement the **BankApplication.modifyBank()** method that changes balance values (using the **deposit()** and **withdraw()**methods) for some bank clients' accounts.
5. Implement the **BankApplication.printBalance()** method that iterates through the bank clients and prints the balance value of their accounts.

Task 2

1. Add the feature of specifying client's gender and name. Define corresponding fields of the **Client** class and constructor.
2. The field value shall be specified with the help of the **enum Gender**. Create a corresponding **enum** with instances **MALE, FEMALE**.
3. Create the **getClientGreeting()** method of the **Client** class that displays hello message in a way: Mr. NAME or Ms. NAME depending on client's gender.
4. If the previous step has been performed with the help of conditional operator (for example **if (gender == Gender.MALE) {}**), change the conditional statement. To do this, create a constructor and access methods for accessing gender string representation ("Mr." and "Ms.") in **Gender** enumeration.

Task 3 (improving the application, "layering", refactoring)

1. Create the **BankService** class that will add clients to the bank and will perform other services. The **BankService.addClient(bank, client)** static method adds a client to a bank. The **BankService** class methods are invoked from **BankApplication**.
2. Execute code refactoring. Place the **Bank**, **Client**, **Account**, **Gender** classes into the **com.luxoft.bankapp.domain** package. Place the **BankService** class to **com.luxoft.bankapp.service**.
3. Review domain objects, limit visibility scope of methods that are not part of client's API (i.e. inner and utility methods).

Task 4 (polymorphism)

1. New business logic requirement: there are two kinds of accounts. One kind is a **SavingAccount**. Second kind is a **CheckingAccount**. The diagram of this operation is different in a way that when creating this account the positive **overdraft** value is indicated, i.e. a loan that a Bank issues to a Client. In case if the **x** value of the **withdraw(x)** method is larger than the current **balance** value, an additional amount of money is allowed to be withdrawn within the limit of the **overdraft** value.
2. Implement the **Account** interface (that defines the methods **deposit(double x)** and **withdraw(double x))** and the abstract class **AbstractAccount** that implements the **Account** interface and has two subclasses: **SavingAccount** and **CheckingAccount**.
3. Change the code that used the **Account** class to use the **Account** interface.
4. Define into the **Account** interface the **maximumAmountToWithdraw()** method that returns the amount of money that can be paid to a client (taking into account the **overdraft** for **CheckingAccount**), and implement it in **SavingAccount** and **CheckingAccount**.

5. **`BankApplication`** must create accounts of both types. Implement the **`BankService.printMaximumAmountToWithdraw(bank)`** method that prints corresponding value for each account. Learn the work of virtual methods and polymorphism principles.

Exercise 15
Bank Application: Working with nested classes

Objectives

Learn to implement listeners by using nested classes.

Description

The event-action paradigm is considered to be rather useful when there are a lot of possible system responses to certain system or user action. In other words, one event requires several actions. In the example of the **`Bank Application`** an event "adding a new client to a bank" can have the following responses: "print added client", "send e-mail notification", etc. It makes no sense to hardcode these actions to the **`Bank`** class, as long as the bank will execute unusual functions and it will be difficult to expand it.

Tasks

Task 1

1. Create the **`ClientRegistrationListener`** interface, define the **`void onClientAdded(Client c)`** method.
2. In the **`Bank`** class create two interface implementations as nested classes: the **`PrintClientListener`**, where the implementation of the **`onClientAdded(Client c)`** method prints a client to console and the **`EmailNotificationListener`**, where the implementation of the **`onClientAdded(Client c)`** method displays to console **`Notification email for client … to be sent`**.
3. Place these two implementations to an array of the **`Bank`** class: **`ClientRegistrationListener [] listeners`**.
4. Modify the **`bank.addClient(Client c)`** method. When new client is added the method iterates the **`listeners`** array and invokes the **`onClientAdded()`** listener's method.
5. Add the **`DebugListener`** listener to the **`Bank`** class constructor. **`DebugListener`** displays the client and current time to console. Implement the **`DebugListener`** as nested class.
6. For each of the listeners defined above, introduce into the **`Bank`** class integer variables (**`printedClients`**, **`emailedClients`**, **`debuggedClients`**) to keep track of the number of clients that have been addressed. These ones will be used for testing purposes, to make sure the **`onClientAdded()`** listener's method has been called.

Exercise 16
Bank Application: Exceptions

Objectives

Handle application errors using an exception hierarchy.

Description

All exceptions need to be created into the **`com.luxoft.bankapp.exceptions`** package. Create exception hierarchy and implement their throwing and handling for the following situations:

1. A negative overdraft value is given when creating a checking account – throw **`IllegalArgumentException`**.
2. The **`withdraw`** or **`deposit`** methods try to work with a negative amount of money - throw **`IllegalArgumentException`**.
3. The **`withdraw`** method requests the amount of money that exceeds the amount that can be given to the client (taking into account the overdraft for **`CheckingAccount`**) - throw checked exception **`NotEnoughFundsException`**.
4. A client with the given name already exists in the bank - throw checked exception **`ClientExistsException`**.

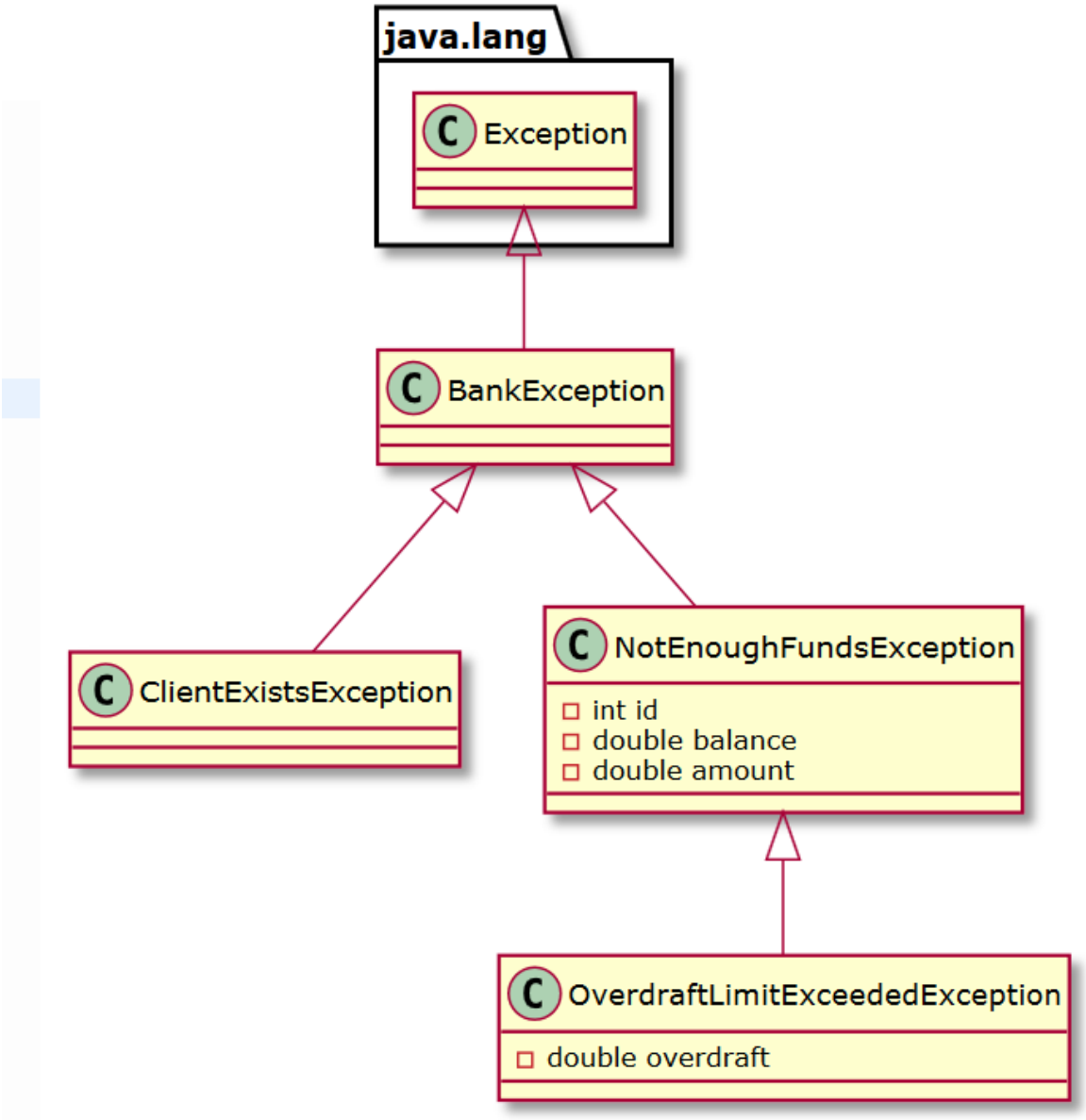
Tasks

Task 1

1. Implement the exception classes' hierarchy. Modify the application taking into account the requirements specified in the description. The **`BankService`** class methods do not catch and handle exceptions. Quite the opposite, they just declare throwing of relevant exceptions. Catching of **`checked`** exceptions is performed by **`BankApplication`**.

Task 2

1. Implement the **`OverdraftLimitExceededException`** subclass of the **`NotEnoughFundsException`** class. This exception is thrown by the **`CheckingAccount`** class in case there are not enough credit funds for issuing the requested amount.
2. Declare the **`OverdraftLimitExceededException`** thrown by an overriden **`withdraw()`** method of the **`CheckingAccount`** class, catch the exception in the **`BankApplication`**.
3. Modify the **`NotEnoughFundsException`** class. Add a field and a corresponding constructor encapsulating the client's account that threw the exception, as well as the maximum amount of money that can be given to the client. Implement relevant access methods; call them when handling exception in **`BankApplication`**.
4. Modify the **`OverdraftLimitExceededException`** class. Add a field and a corresponding constructor encapsulating the **`overdraft`**. Implement the access method; call it when handling exception in **`BankApplication`**.


java.lang
Exception
BankException
ClientExistsException
NotEnoughFundsException
int id
double balance
double amount
OverdraftLimitExceededException
double overdraft