# Exercise 1

# Inner and anonymous classes

## Objectives

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

## Task

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 2 Bank Application: Working with nested classes

## Objectives

Learn to implement listeners by using nested classes.

## Description

The event-action paradigm is 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, if the bank will execute unusual functions and it will be difficult to expand it.

## Tasks:

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 3 (Project m3-jva007-bank-app)

# 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**.