

## Lab 1a: Java basics

# 1 Introduction

The first part of today's lab will help you grasp the main idea of the Java programming language. Java is a class-based Object-Oriented Programming (OOP) language. As the name suggests, OOP is based on the concept of *objects*, which are the encapsulation of data and/or its behaviour into a single container.

## 2 Objects and Classes

### 2.1 Objects

There are many examples of real-world objects: a car, a dog, an apartment, etc. Objects share two characteristics: state and behavior. For each object that we can think of, we can ask ourselves:

- What possible states can this object be in?
- What possible behavior can this object have?

Let's take an example of an object, for instance a bicycle. Bicycles have a state, represented by the current gear, pedal cadence and speed, and a behavior, such as changing gear, changing pedal cadence, speed up, applying brakes, and so on.

Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming. Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables) and describes its behavior through methods (functions).

### 2.2 Classes

As mentioned, java is also a class-based programming language. Very often, the terms **class** and **object** are used interchangeably. However, there is a clear difference between them. Namely, in the real world, there are many objects of the same kind. With respect to our previous example, each bicycle is built from the same set of blueprints and therefore contains the same main components that describe them as bicycles, but each bicycle is physically different from each other. In object-oriented terms, we say that **aBicycle** and **anotherBicycle** are two instances of the class of objects known as **Bicycle**. Therefore, a class is the blueprint from which individual objects are created (instantiated).

## 2.3 Simple code

Let us consider an example of the class **Bicycle**. The fields **cadence**, **speed**, and **gear** represent the class' state (main components), and the methods (**changeCadence(...)**, **changeGear(...)**, **speedUp(...)**, **applyBrakes(...)**, **printStates()**) define some of the functions that use the the class' main components.

Notice that the **Bicycle** class does not contain a **main(...)** method. This is due to the fact that this class is not a complete application, it just represents the **category** of bicycles from which different objects (specific bicycles) can be instantiated.

```
class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    void printStates() {
        System.out.println("cadence:" + cadence);
        System.out.println("speed:" + speed);
        System.out.println("gear:" + gear);
    }
}
```

## 3 Today's lab

In this lab we will build the account class. This class will contain the *name* of the account holder, the *unique identification number*, and the *available amount of money* in the bank account. The behaviors will consist of *balance inquiry*, *deposit*, and *withdrawal*.

### 3.1 First class

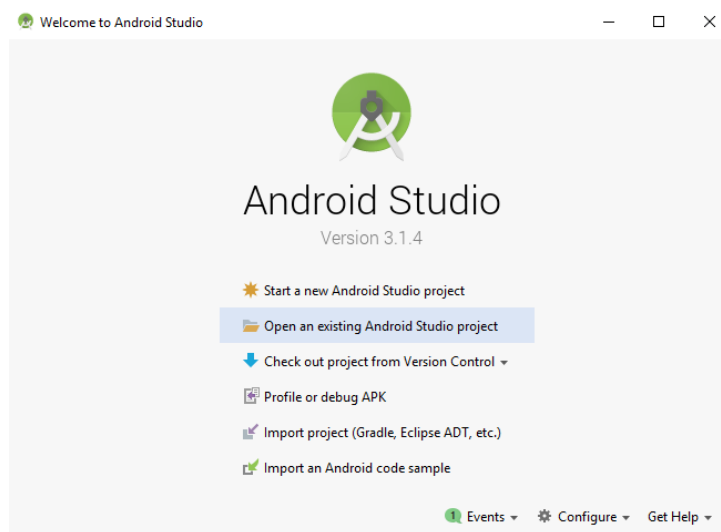



Figure 1: Open an existing Android Studio project

For this lab, download the project from moodle<sup>1</sup>, called **MyApplication**. Open **Android Studio** and click on **Open an existing Android Studio project**, as shown in Figure 1, and browse to the folder **MyApplication** that you just downloaded. The application should have this icon  to show that it is an *Android Studio* project. We will explain more about the files in the project, as we go along. Open the **Account.java** file, that is the class file. The class file looks like this:

```
class Account {
    String name;
    int number;
    double balance;

    public void deposit(double amount) {
        balance += amount;
    }
}
```

<sup>1</sup><https://moodle.epfl.ch/course/view.php?id=15420>

```

    }

    public void withdrawal(double amount) {
        if (balance >= amount) {
            balance -= amount;
        } else {
            System.out.println("Insufficient Funds");
        }
    }

    public void displayBalance() {
        System.out.println("The balance on account "
            + number + " is " + balance);
    }
}

```

As you can see in the class file, there are three **fields**:

- a **String** name
- an **int** number
- a **double** balance

Notice that these variables are not created inside any particular method. This class will not execute on its own as there is no entry point, that is to say no **main(...)** method. It will just contain the information necessary to create account objects in other classes that may wish to use them.

In Java, the way we make classes is by writing the key word **class** followed by the arbitrary name of the class. Following the usual Java coding style, we write the **fields** in the beginning, and then we write the methods of the class.


Let's now have a look at the methods defined within the **class Account**. As mentioned, all methods defined in one class can use the fields of that class. For instance, the method **displayBalance()** displays the account number along with the available balance.

## 3.2 Using a class

As previously mentioned, we cannot run this file. We need a main file to do that. Let's now have a look at the account main class (**TestClass.java** in this lab). The main class will instantiate several **Account** objects, giving the names **acct1** and **acct2**). The way we instantiate different objects is by using the key word **new**. After this, we assign different values to the fields of **each instance** of the class **Account**.

The main class looks like this:

```
public class TestClass {  
    public static void main(String[] args) {  
        Account acct1 = new Account();  
        Account acct2 = new Account();  
  
        acct1.name = "Bill";  
        acct1.number = 738924;  
        acct1.balance = 231.48;  
  
        acct2.name = "Sue";  
        acct2.number = 894730;  
        acct2.balance = 0;  
  
        acct1.displayBalance();  
        acct1.deposit(89.00);  
        acct1.displayBalance();  
  
        acct2.displayBalance();  
        acct2.withdrawal(300);  
  
        System.out.println(acct1.toString());  
    }  
}
```

If we run the main file (right click on the **TestClass.java** in the Android Studio tree and select  **Run 'TestClass.main()'**), we should get the following result:

```
The balance on account 738924 is 231.48  
The balance on account 738924 is 320.48  
The balance on account 894730 is 0.0  
Insufficient Funds
```

### 3.3 Visibility and 'access rights'

Now that we have both the class file and the main file, there are still some modifications to be made. Namely, now other classes have direct access to the fields of our class. Usually, class fields should be labeled as **private** and only accessible to the object itself. Contrary to what you may think, you do not have direct access to your own balance at your bank. You

are allowed to change your name, address, etc. through your bank's office or website. For instance, knowing someone else's bank account number does not enable you to change their balance: you have to go through the bank system which will control the access rights. By making class fields **private**, you are protecting your data. By not allowing class fields to be accessed directly, we control how data will be modified and accessed through well-designed methods in our class. This is done by adding the key word **private** in front of the name of the class fields.

Declare all of the account variable as private and then run the account client program again.

```
private String name;  
private int number;  
private double balance;
```

If we have a look at the **TestClass.java** file, this program will not work anymore, as we removed access to **Account** class fields. Therefore, we also need to add some methods in **Account** in order to accommodate the changes.

We have two different types of methods in a class, accessors, also known as **getters**, and mutators, or **setters**. Accessor methods are used to access data from outside the object, and are therefore declared as **public**. They are simple methods that return the value of a specific class field. On the other side, mutator methods are methods that change the value of a class fields. In order to access data from **TestClass** add the following three **getters**:

```
public String getName() {  
    return name;  
}  
  
public int getNumber() {  
    return number;  
}  
  
public double getBalance() {  
    return balance;  
}
```

Furthermore, add the following two mutator methods for setting the account name and balance. Do not add any mutator method for setting the balance, as we still do not want users overwriting their own balances directly.

```
public void setName(String name) {
```

```
        this.name = name;
    }

    public void setNumber(int number) {
        this.number = number;
    }
}
```

Your main program should now look like this:

```
public class TestClass {
    public static void main(String[] args) {
        Account acct1 = new Account();
        Account acct2 = new Account();

        acct1.setName("Bill");
        acct1.setNumber(738924);
        acct1.deposit(231.48);

        acct2.setName("Sue");
        acct2.setNumber(894730);
        acct2.deposit(0);

        acct1.displayBalance();
        acct1.deposit(89.00);
        acct1.displayBalance();

        acct2.displayBalance();
        acct2.withdrawal(300);

        System.out.println(acct1.toString());
    }
}
```

### 3.4 Constructors

Notice how we are creating an object then filling in the fields. We can combine these two operations into a third type of method used in a class file, the constructor. Constructors are used to create new objects from class files, and we have already used one, the default constructor:

```
Account acct1 = new Account();
```

We create an object in one line beginning with the name of the class for which we are creating an object, then the object name, followed by the keyword `new`, and a call to the constructor method. The name of the constructor method is always the same as the name of the class itself. The default constructor is automatically created when a class file is written. It creates an empty instance of an object.

We can also build our own constructors, with or without input parameters. These parameters can then be used to initialize some or all of the fields as the object is being created. In the case of writing our own constructors, the default constructor goes away.

Here is an example of two different constructors.

```
public Account(String newName, int newNumber) {  
    this.name = newName;  
    this.number = newNumber;  
}  
  
public Account(String newName, int newNumber, double initialBalance) {  
    this.name = newName;  
    this.number = newNumber;  
    this.balance = initialBalance;  
}
```

The first one would be used to create an account and set the name and number. Since the balance was not initialized, it will be set to the default value of zero. The second constructor will initialize all three instance variables. All constructors must have the same name as the class itself, they are public, and they do not have a return value.

Notice that both constructors have the same name, but different parameter lists. This is an example of **overloading** a method. The compiler will recognize which constructor you are calling by checking the given arguments. As it can be seen in the main file, after adding the previous two explicit constructors in the class file, if we try to run the main program, it will fail. This is due to the fact that we are still using the default constructor, but recall that constructor goes away once we start writing constructors on our own. The default constructors came with the class, as long as no other constructors are written. However, if we want to use the default constructor, we must write it within the class file. Add the following two newly-defined constructors calls in the main file (**TestClass.java**), and run the program again.

```
Account acct1 = new Account("Bill", 738924)  
Account acct2 = new Account("Sue", 894730, 150)
```



Notice that assigning values to name and number is done in both constructors. Let's take advantage of code that has already been written. Namely, in the second constructor we can use the first constructor and just add on the code to initialize the balance. To refer to a constructor in our own class we use the keyword **this**. Let's rewrite the second constructor as follows:

```
public Account(String newName, int newNumber, double initialBalance) {  
    this(newName, newNumber);  
    this.balance = initialBalance;  
}
```

In this example, the keyword **this** is used to refer to the class itself i.e. to use one of the constructors of the class. However, notice that we have already used this keyword in the implementation of the **setName(...)** function. Let's have a look at this function:

```
public void setName(String name) {  
    this.name = name;  
}
```

The account class has a field variable called **name** but we may also use this variable's name as a parameters or local variables in a method. To distinguish between them, we use the keyword **this**. We call this shadowing, using the same variable name from the field variable as a parameter or local variable. It makes program more readable and allows to use meaningful variable names.

Another use for the keyword, this is when calling another method within the same object. Let's suppose that we have functions **updateBalance()** and **calcInterest()** within the class Account:

```
public class Account {  
    ...  
    public void updateBalance(){  
        int increase = this.calculateInterest();  
        balance = balance + increase;  
    }  
    ...  
    private int calculateInterest() {  
        //It calculates monthly interest on current balance  
    }  
}
```

When we want to call a method within the same class, it may be helpful to use the keyword, `this` to emphasize that we want to use one of the class methods. Notice also that we made the method `calcInterest()` private. This is due to the fact that this method will not be used by other clients, only by the objects themselves, i.e. it does not need to be accessible outside of the object.

To sum up: The word `this` refers to the implicit parameter i.e. the object on which a method is called and it is used in the following cases:

- To call one constructor from another constructor `this(parameters)`
- To refer to a class field `this.field`
- To call another class method `this.method()`

### 3.5 Inheritance

In the Java language, classes can be derived from other classes. A class that is derived from another class is called a subclass (child class). The class from which the subclass is derived from is called a superclass (parent class).

**Object** is the only class in Java that has no superclass. Every class is implicitly a subclass of **Object**. There is no limit in inheritance, and ultimately everything is derived from the topmost class, **Object**.

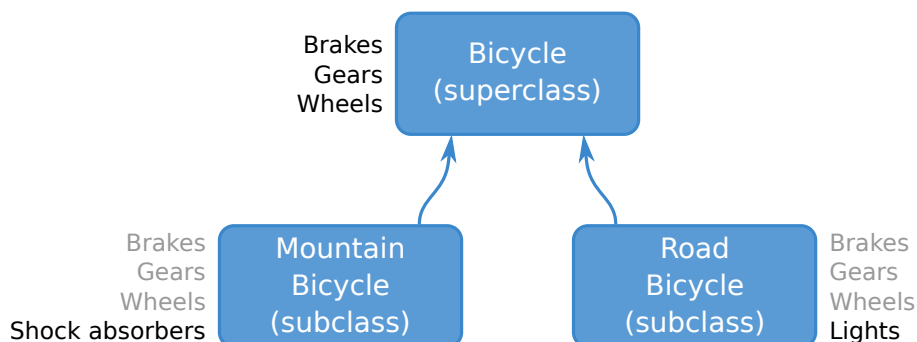


Figure 2: Two subclasses of the superclass **Bicycle** inheriting same features and adding more specific ones

The idea of inheritance is simple: creating a new class with the same features as one already existing and, also, adding more specific features. Coming back to the **Bicycle** example, this class describes a generic bicycle with features, such as gears, breaks, etc. If we want to go on a road trip, we might need a road bike that has lights for when it becomes dark and to be seen. Although, in our road trip we might need a mountain bike, since we want to go off track. As shown in Figure 2, we can create a class named **MountainBicycle**, that would have the same features as a generic bicycle, with some additional

ones, such as shock absorbers. In both cases, the bicycles have wheels gears and brakes, but they can be tuned for a specific use case. **MountainBicycle** and **RoadBicycle** are a subclasses of **Bicycle**. By doing this, we can reuse the fields and methods of the existing superclass without having to write them ourselves. A subclass inherits all the members (fields, methods, and nested classes) from its superclass.

Considering today's lab, we declare a class with the name **PrivateAccount** as a subclass of the superclass **Account** (pay attention to the key word **extends** that we use for the class inheritance):

```
public class PrivateAccount extends Account {

    private boolean blockCard; // we cannot cash out more

    public PrivateAccount(String newName, int newNumber,
                           double initialBalance, boolean limit) {
        super(newName, newNumber, initialBalance);
        this.blockCard = limit;
    }

    private boolean maxReached(double amount) {
        double currentBalance = getBalance();
        if (amount >= currentBalance/2) {
            blockCard = true;
            return blockCard;
        }
        else {
            blockCard = false;
            return blockCard;
        }
    }

    @Override
    public void withdrawal(double amount) {
        boolean isOk = maxReached(amount);
        double currentBalance = getBalance();
        if (isOk)
            System.out.println("The limit has been reached");
        else
```

```
        super.withdrawal(amount);  
    }  
}
```

**PrivateAccount** inherits all the fields and methods of the superclass **Account** and adds the new field **blockCard** and the new method **maxReached()** to check if the card should be blocked or not. When it comes to inheritance, constructors are not inherited, but the constructor of the superclass can be invoked from a subclass. This is done by using the keyword **super**:

```
super(newName, newNumber, initialBalance);
```

In a subclass, the inherited methods can be used directly as they are. Importantly, new methods in the subclass that have the same signature (method name and number, type and order of its parameters) as the one in the superclass can be overwritten, or, to use the Java terminology, overridden. This is the case of the **withdrawal(...)** method. In this case, we use the annotation **@Override** to override the method from the superclass, as shown in the code snippet.

In the main program add the following three lines in order to use the subclass **PrivateAccount**, and run the program again:

```
PrivateAccount acct3 = new PrivateAccount("Milan", 234567, 500, false);  
acct3.withdrawal(50);  
acct3.displayBalance();  
acct3.withdrawal(350);
```

Now it's time to use these Java basics and build your first Android application!