OBJECT ORIENTED PROGRAMMING - LAB 6

3rd October, 2023

Learning Objectives:

1. Abstracting real-life concepts with Object-oriented ideas
2. Making and using Constructors
3. Understanding the importance of getters and setters
4. Understanding and using access modifiers
5. Basic UML diagram designs

In the previous labs, you have learned about abstraction using the **`Dictionary`** classes. In this lab, we will apply the same concepts to the **`Person`** and **`BankAccount`** classes. We will first go through the example of the **`Person`** class that has already been implemented for you. But you have to carefully understand it to apply similar paradigms to the **`BankAccount`** class. While going through the **`Person`** class, we will learn about constructors, getters/setters, and access modifiers.

Constructors

Constructors are special methods that are called when we create an instance of the class. The have the same name as that of their class. When we make a new object using the **`new`** command, the constructor of that class is called. The primary purpose of a constructor is to initialize the object's attributes to specific values (that can be passed when creating the object). This ensures that the object starts in a well-defined state.

Constructors can also be overloaded. Overloading occurs when we create 2 methods by the same name but different signatures. Consider this example:

```
public class MyNumber
{
    public int myint;
    public double mydouble;
    public MyNumber(int num)
    {
        this.myint = num;
    }
    public MyNumber(double num)
   {
       this.mydouble = num;
   }
}
```

```
public class Driver
{
    public static void main(String[]
args)
    {
         MyNumber num1 = new
MyNumber(1);
         MyNumber num2 = new
MyNumber(1.2);
    }
}
```

In this example, we see that for **`num1`** the first constructor is used as **`1`** matches to an **`int`**, but for **`num2`**, the second constructor is used as **`1.2`** matches to a **`double`**. Similar to constructors, we can also overload other methods. You will make use of overloaded constructors in **`BankAccount`**.

Access Modifiers

Access modifiers are keywords that define the visibility and accessibility of variables, methods, or classes outside the class. Today, our main focus is on:

1. **`public`**: These are accessible outside the class. If a variable is declared public, then it can be viewed and even changed from outside the code. If a method is made, then it can used by another class as well. This is the most widely useable access.
2. **`private`**: These are not accessible outside the class. A private variable can only be read and written into by the class in which it is declared. private methods cannot be accessed/ used outside the class.
3. **`default`**: when no access modifier is provided, then the variable/ method takes on the default access i.e. available to all classes inside the package. For now, you can consider the package to be a folder. We will get into it in more detail later on. For our exercise, all classes are in the same package, so the default access modifier will give access to all classes.

In the Person class, we choose to keep the name **`public`** and the age **`private`**. This is a *design choice* we make to explain their use. Try accessing **`name`** and **`age`** of **`p1`** from the **`Driver`** class directly. What happens for both? In which one do you get an error?

Note: There is also an access modifier called **`protected`**. We will talk about it in detail when we come to packages. You need not worry about it for now.

In addition to these access modifiers, there are also keywords like **`final`** and **`static`**. **`final`** makes the variable immutable i.e. it cannot change its value even inside the class. **`static`** variables are the ones that belong to the class and not to the object we create.

Take a look at **`num_of_people`** variable in the **`Person`** class. It is declared **`static`**, indicating that its value belongs to the **`Person`** class. we update it whenever a new person is made. Also, look at how we access this variable in the **`Driver`** class. Even though we could have also accessed it by an object (p1 or p2), we chose to access it by the **`Person`** class itself to show that its value belongs to the class rather than any object of it.

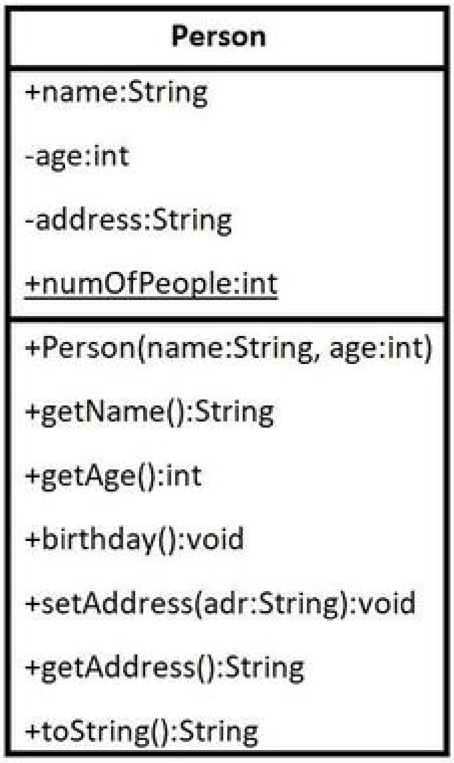
Did you notice that even the **`main`** method in the **`Driver`** class is **`static`**? This is because the **`main`** method is the first one to be executed, and the compiler accesses it from the **`Driver`** class and not from any object of the **`Driver`** class.

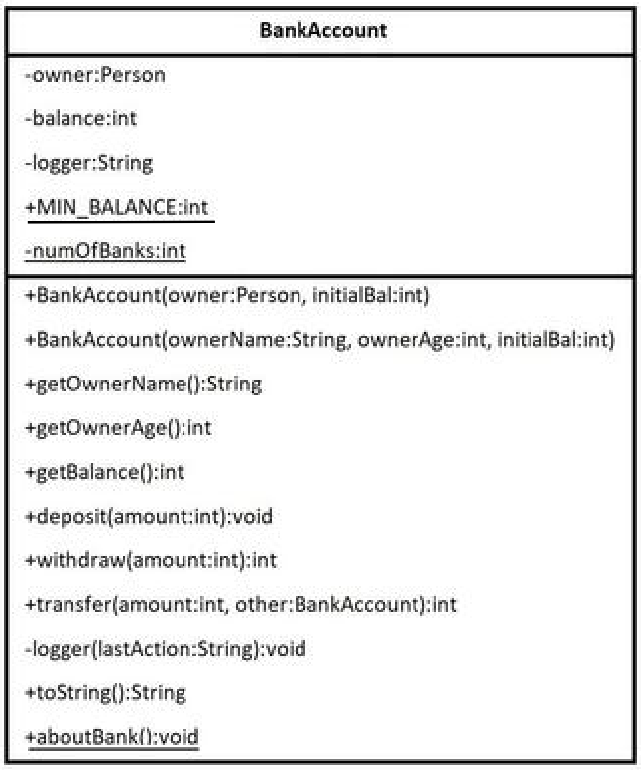
Getters and Setters

Look into the implementation of the **`Person`** class. You will observe that for getting and setting the different variables, we use getters and setters. This is primarily because we cannot access a variable like **`age`** of a person from the **`Driver`** class (since it is **`private`**). So we created a getter to get the value for us. Since the getter is inside the **`Person`** class, it can access the **`private`** variables as well and fetch the value for us.

Even if variables are **`public`**, it is always better to make the getters and setters for each variable, as this gives you (as a programmer) more control over the variables of your class and also prevents unexpected changes from happening from outside the class. Take a close look at all the getters and setters of the Person class. You will have to make similar methods for **`BankAcoount`**!!

UML




BankAccount
-owner:Person
-balance:int
-logger:String
+MIN_BALANCE:int
-numOfBanks:int
+BankAccount(owner:Person, initialBal:int)
+BankAccount(ownerName:String, ownerAge:int, initialBal:int)
+getOwnerName():String
+getOwnerAge():int
+getBalance():int
+deposit(amount:int):void
+withdraw(amount:int):int
+transfer(amount:int, other:BankAccount):int
-logger(lastAction:String):void
+toString():String
+aboutBank():void

UML diagrams provide an easy visual representation of the complete Design. Observe the UML diagrams provided to you. The variables and methods are grouped together in separate sections of the *Class box*. We use (+) to indicate **`public`** access and (-) to indicate **`private`** access. Additionally, the **`static`** variables and methods are underlined.

For example:

1. **`+numOfPeople:int`** indicates that the variable is **`static`**, **`public`**, and of type **`int`**

2. **`+setAddress(adr:String):void`** indicates that the method is **`public`**, takes a **`String`**, and does return anything

Question

Now we get to the main question for this lab. Similar to the **`Person`** class, you have to implement the **`BankAccount`** class. Do not take this exercise lite! The quiz will be based on this. You have to implement the following methods:

1. **`public BankAccount(Person owner, int initialBal)`**
2. **`public BankAccount(String ownerName, int ownerAge, int initialBal)`**
3. **`public String getOwnerName()`**
4. **`public int getOwnerAge()`**
5. **`public int getBalance()`**
6. **`public void deposit(int amount)`**
7. **`public int withdraw(int amount)`**
8. **`public int transfer(int amount, BankAccount other)`**
9. **`private void logger(String lastAction)`**
10. **`public String toString()`**

Keep the following points in mind while completing the methods:

1. In all transaction methods (withdraw, deposit, and transfer), you need to use the **`logger()`** to set the last transaction. You can store strings like "Transfer Rs 100", "Deposit Rs 1000", or even "Withdraw for Rs 1000 failed"!

2. It is OK to make a bank account with an initial deposit less than **`MIN_BALANCE`**. However, for any withdrawal or transfer, you need to make sure that the bank account has at least **`MIN_BALANCE`** after the operation. In this case, the Person needs to add money using the deposit method.

3. There is no fixed answer for **`toString()`**. You can make it as informative as possible.