Designing Classes - deep dive

Objectives

- Recap previous sessions
- Choosing classes
- Accessors, Mutators, and Immutable Classes
- Common Errors
- Static methods
- Homework exercises
- Guidelines
- Next steps

Recap previous sessions

- What is a constructor?
- How many types of constructors do you know?
- What is the `this` keyword?
- Give examples where `this` is helpful
- What is encapsulation?
- What is the difference between private and public variables?
- What level of visibility should we choose for helper methods?

Choosing classes

- Designing a class can be a challenge. it is not always easy to tell how to start or whether the result is of good quality
- As a rule of thumb, `Class names should be nouns, and method names should be verbs`
  - `A class should represent a single concept from the problem domain, such as business, science, mathematics, etc`
- Now let's see some good examples of classes from:
  - Mathematics:
    - Point
    - Rectangle
    - Ellipse
  - Real life entities:
    - BankAccount
    - Car
    - Animal
  - etc
- For the previous classes, the properties of a typical object are easy to understand. A `Rectangle` object has a width and height. A `BankAccount` object can give you the possibility to withdraw money (an action, thus a very good name for a method)
- Even if you encounter weird class names when you are using a certain framework, try to stick as much as you can to use nouns when choosing names for your newly created classes

- You know what makes a good class when you know exactly what an object from that class is supposed to do. If you don't, there is a high probability that you're not on the right track and the class should be redesigned.
- One common mistake which is done by beginner programmers is to turn an action into a class. For example, if you have a homework assignment which requires you to compute a paycheck, you may consider writing a class `ComputePaycheck`. But can you visualize a `ComputePaycheck` object? The fact that `ComputePaycheck` isn't a noun, indicates from the start that there is something going wrong there. But, on the other hand, if we use a `Paycheck` class, everything makes a lot more sense. The word `paycheck` is a noun and you can easily visualize a paycheck object. And then, you can also think about useful methods of the `Paycheck` class, such as `computeTaxes, etc`.
- Just to briefly summarise what we spoke before, in order to find classes for the business logic that you are implementing, first, try to spot all the nouns.

Accessors, Mutators, and Immutable Classes

- If you recall from the previous lesson, we talked about `setters` which are methods that modifies the object on which it is invoked. They are also called `mutators`.

- Also, we spoke about `getters` which merely access information without making any modifications, they are also called `accessors`.

- Before moving forward, I want to say the following:

    - Any method which modifies the state is also called a `mutator`
    - Any method which simply reads information is called an `accessor`
    - Thus, not only getters and setters because we can have more methods in a class and for sure, we will!

- Let's design a class which will abstract the concept of a bank account. This will make it easier for us to understand the concepts that we just talked about.

    - Regarding the name, if we simply choose `account` it does not feel complete. A user might have a bank account, a social network account (like facebook), etc. Thus, we need to be more specific and due to the fact that we will be using this concept of an account to perform banking related logic, `BankAccount` seems just right.
    - Next, we will have actions which will enable us to deposit, withdraw and also to check the available balance for our account.

- In the snippet below, you will find the class that we just discussed. Also, you can observe that the `deposit` and `withdraw` methods are mutator methods, but the `getBalance` method does not modify the state of the `account` object

```
public class BankAccount {
    double amount;

    public BankAccount(double amount) {
        this.amount = amount;
    }
```

```
    public void deposit(double amountToDeposit) {
        this.amount = this.amount + amountToDeposit;
    }

    public void withdraw(double amountToWithdraw) {
        this.amount = this.amount-amountToWithdraw;
    }

    public double getBalance() {
        return this.amount;
    }
}

  public class Application {

      public static void main(String[] args) {
          BankAccount account = new BankAccount(2555.32);
          account.deposit(1000);
          account.withdraw(250);
          System.out.println(account.getBalance());
      }

  }
```

- Note that you can call the `accessor` methods as many times as you want! The internal state of the object will not be changed. This is not the case with `mutator` methods. They will modify the state at each call so pay attention whenever you call them.
- When a method changes the internal state of an object, you can say that the method has a `side effect`.
- In the situation where we have only accessor methods, thus no mutators, we say that the class is `Immutable`. This means, that once an object is created, you cannot change its internal state.
- A good example of an Immutable class is the `String` class. Whatever method you will call from this class, it will return a new String. It will not touch the existing one.
    - Pay attention as this question is often asked during interviews.
- Let's take a look at some examples with `String` objects where we can observe that the object is not changed:

```
public class Application {

  public static void main(String[] args) {
    String name = "Bogdan Niculeasa";
    String uppercaseName = name.toUpperCase();
    System.out.println(name);
    String replaced = name.replace("a", "i");
    System.out.println("Replaced = " +  replaced);
    System.out.println("Name: "+ name);
```

```
    }
  }
```

- An immutable class has a major advantage: `It is safe to give out references to its objects freely`.
    - If no method can change the object's value, then no code can modify the object at an unexpected time.
    - In contrast, if you give out a `BankAccount` reference to any other method, you have to be aware that the state of your object may change—the other method can call the deposit and withdraw methods on the reference that you gave it.

Common Errors

- Methods cannot update parameters of primitive type(numbers, `char` and `boolean`)
- In order to illustrate this, let's add a new method to our `BankAccount` class which knows to transfer money from one banck account into another:

```
public void transfer(double amount, BankAccount destinationAccount) {
    destinationAccount.deposit(amount);
    // This will have no effect after the method exits
    amount = 250;
  }
```

    - Next, let's call this method and afterwards, we will print the value of the `amount` variable:

```
  public class Application {

    public static void main(String[] args) {
      BankAccount account1 = new BankAccount(1000);
      BankAccount account2 = new BankAccount(2000);

      double amountToBeTransfered = 500;
      account1.transfer(amountToBeTransfered, account2);
      System.out.println("Balance for account 1: " +
account1.getBalance());
      System.out.println("Balance for account 2: " +
account2.getBalance());
      System.out.println("Amount = " +  amountToBeTransfered);
    }
  }
```

        - The output of this snippet will be the following:

```

```

```
Balance for account 1: 500.0
Balance for account 2: 2500.0
Amount = 500.0
```

  - This is because as the method starts, JAVA will make a copy for every parameter that a method receives. You can play with it inside but as soon as the method exits, this copy dies.
  - This method of creating a copy of the parameters is called `Pass by value`! And that's how JAVA works!
  - NOTE: `Parameters are passed by value` is another fundamental question which can be seen during interviews. The opposite of this technique is called `Pass by reference` but it is not supported by JAVA

- Regarding the `pass by value` mechanism, you saw that you can do whatever you want to that variable inside the method but as soon as the methods exits, any change that you performed will die. This also applies to objects!
  - It might look confusing that inside a method you can change the state of the object and it will persist after you exit the method but if you reassign the object variable to a new object, the changes will have no effect as soon as you exit the method. See the snippet below where inside of our `transfer` method, we try to reassign the BankAccount object to a whole new object:

```
        public void transfer(double amount, BankAccount
destinationAccount) {
        destinationAccount.deposit(amount);
        this.withdraw(amount);
        // This will have no effect after the method exits
        amount = 250;
        destinationAccount = new BankAccount(10000);
      }
```

    - Now if we go and run the same code as before, the account2 will not have 10000 as amount:

```
    public class Application {

      public static void main(String[] args) {
        BankAccount account1 = new BankAccount(1000);
        BankAccount account2 = new BankAccount(2000);

        double amountToBeTransfered = 500;
        account1.transfer(amountToBeTransfered, account2);
        System.out.println("Balance for account 1: " +
  account1.getBalance());
        System.out.println("Balance for account 2: " +
  account2.getBalance());
```

```
        System.out.println("Amount = " +  amountToBeTransfered);
    }
}
```

- As we said, it will have the same output as before:

```
Balance for account 1: 500.0
Balance for account 2: 2500.0
Amount = 500.0
```

- This concludes the following:
  - `In JAVA, both primitive types and object references are copied by value (or passed by value)`
- Note: do not be confused about the fact that you can change the internal state of an object! Pass by value means that you cannot make the object point to something else after you exit the method.

Static methods

- There is a chance that sometimes you need a method that is not invoked on an object. Such a method is called a `static method` or `class method`
- In contrast, all the methods that we have written for our previous classes are called `instance methods` because they operate on a particul instance of an object
- NOTE: `instance method vs class method` is also another topic that you might encounter during interviews.
- A typical example of static methods are the ones from the Math class like:
  - min
  - max
  - sqrt
  - abs
  - etc.
  - You don't need an object in order to call them. Simply specify the name of the class followed by a dot and then directly by the name of the method
- If you try to reason why would you ever want to have a static method, well, you can have multiple reasons:
  - if you perform certain mathematical operations
  - if you want to have some logic which is independent from the objects that will be created by that class.
- Note that you can also have fields that are static! They obey the same rule as the static methods, meaning that you don't need an instance in order to access them

    - As static methods belong to a class, so does the `static fields`

- There is also a catch if you want to access a normal field in a static method, `JAVA does not allow this!` and it will throw the following error:

```
Cannot make a static reference to the non-static field <field_name>
```

    - In order to prove the concepts above, let's take a look at the snippet below where we want to count how many instances are created from a class:

```
public class Car {
  private static int numberOfInstances = 0;
  public Car() {
    numberOfInstances++;
  }

  public static int getNumberOfInstances() {
    return numberOfInstances;
  }

}
```

    - And nowm when we call the static method, we should call it using the class name. I said `should` because we can also call it via an object reference but your IDE will give an warning saying that the method should be accessed in a static way:

```
public class Application {

  public static void main(String[] args) {
    Car car = new Car();
    Car car2 = new Car();

    System.out.println(Car.getNumberOfInstances());
  }
}
```

    - This exercise is a very helpful one! Try to run and understand each piece of it! If you understand the logic behind it, a huge number of doors will be opened in the future.

Homework exercises

1. What is the rule of thumb for finding classes?
2. If you will have to write a program that plays chess, will `ChessBoard` be an appropriate class? What about `MovePiece`
3. Is the substring method of the String class an accessor or a mutator?

4. Suppose Java had no static methods. Then all methods of the Math class would be instance methods. How would you compute the square root of x?
5. Is a Single Class with Many Static Methods not Object-Oriented
6. Name two static fields of the `System` class.

Guidelines

- We have repeated content from the previous session but we have also added a lot of new concepts
- Try to understand each concept what does it mean and where it can help us. If you have difficulties in understanding this, please ask me in the following lectures.
- Pay more attention to the topics that I say you might encounter them in interviews

Next steps

- In the following lessons we will take a look at the following topicS:
    - Alternative forms of field initialization
    - Scope of Class members
    - Overlapping Scope and shadowing
    - Organizing code in Packages
    - Free topics (feel free to propose anything)