Deep dive into Constructors and the `this` keyword

Objectives

- Recap previous lesson
- Introduction to encapsulation
- The `this` keyword
- Deep dive into constructors
- Homework exercises
- Guidelines

Recap previous lesson

- What do we mean by OOP?
- What is a class?
- What is an object?
- What is the difference between a class and an object?
- What is a constructor?
- What is the default constructor?
- What are the differences between a constructor and a method?
- What are the similarities between a constructor and a method?
- What is the difference between a variable of a primitive type and an object?
- What is the state of an object?
- What is the behaviour of an object?
- What is a getter?
- What is a setter?

Introduction to encapsulation

Encapsulation is one of the four fundamental OOP concepts. The other three are `inheritance`, `polymorphism`, and `abstraction`.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

To achieve encapsulation in Java we have to:

1. Declare the variables of a class as private.
2. Provide public setter and getter methods to modify and view the variables values.

- Following is an example that demonstrates how to achieve Encapsulation in Java:

```
public class Person {
   private String name;
   private String cnp;
   private int age;
```

```
   public int getAge() {
      return this.age;
   }

   public String getName() {
      return this.name;
   }

   public String getCnp() {
      return this.cnp;
   }

   public void setAge( int newAge) {
      this.age = newAge;
   }

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

   public void setCnp( String newCnp) {
      this.cnp = newCnp;
   }
}
```

- The public `setXXX()` and `getXXX()` methods are the access points of the instance variables of the Person class.
- Normally, these methods are referred as `getters` and `setters`.
- Therefore, any class that wants to access the variables should access them through these getters and setters.
- Below you can see an example of how the internal state of the Person can and should be accessed:

```
public class Application {

   public static void main(String args[]) {
      Person person = new Person();
      encap.setName("James");
      encap.setAge(20);
      encap.setCnp("1931114171093");

      System.out.print("Name : " + person.getName() + " Age : " +
person.getAge());
   }
}
```

- Now you might wonder, why all this hassle for accessing and mutating (updating) the fields when we can set them directly if we drop the `private` access modifier
- The problem, of letting the fields `public` is the fact that in this manner, everyone can give whatever values they want, thus, interfering with our logic.
    - For example, let's suppose the previous `Person` class. If, for the `CNP` field, we allow everyone to directly access it, wrong values can be entered (like an incomplete CNP, or wrong birthdate, wrong county code, etc).
    - But by letting the external clients accessing our code only via the predefined setters, we can have a logic which checks if the value that will be passed to our setter matches the desired format and only then, the state of our object will be updated.
    - See the example below where we perform a small check to see if the new CNP has 13 digits (of course, you can expand this logic further but we will use only this check just to prove a point):

```
public void setCnp(String newCnp) {
    if(newCnp.length == 13) {
        this.cnp = newCnp;
    } else {
        //Someone tried to use an invalid value
        //If we want, we can warn it here
    }
}
```

- In summary, we can extract two main benefits of using `Encapsulation`:

    1. The fields of a class can be made read-only or write-only.

    2. A class can have total control over what is stored in its fields.

Deep dive into constructors

- A `constructor` is a block of code that initializes the newly created object. The constructor resembles an instance method in JAVA but it’s not quite a method as it doesn’t have a return type. In short constructor and method are different
- People often refer constructor as special type of method in Java.

Constructor has same name as the class. Have a look at the snippet below:

```
public class MyClass{
//This is the constructor
    MyClass(){
    }
}
```

NOTE: The `constructor` name matches with the `class` name and it doesn’t have a return type.

- To understand the working of constructor, let's take an example in which we have a class `MyClass`.

- When we create the object of MyClass like this: `MyClass obj = new MyClass()` The `new` keyword here creates the object of class `MyClass` and invokes the constructor to initialize this newly created object.

Below we have created an object obj of class `Person` and then we displayed the instance variable `name` of the object. As you can see that the output is what we have passed to the name during initialization in constructor. This shows that when we created the object obj the constructor got invoked. In this example we have used `this` keyword, which refers to the current object, namely `obj`.

```
public class Person {
   String name;
   Hello(){
      this.name = "Bogdan, Niculeasa";
   }
}

public class Application{
   public static void main(String[] args) {
      Person person = new Person();
      System.out.println(person.name);
   }
}
```

Types of constructors

There are three types of constructors:

1. Default
2. No-arg constructor
3. Parameterized constructor

Default Constructor

- If you do not implement any constructor in your class, Java compiler inserts a default constructor into your code on your behalf.
- This constructor is known as `default` constructor. You would not find it in your source code(the java file) as it would be inserted into the code during compilation and exists in .class file.
- NOTE: If you implement any constructor then you no longer receive a default constructor from Java compiler.

No-arg constructor

- The `constructor` with no arguments is known as `no-arg constructor`. The signature is the same as for the `default constructor`, however the body can have any code unlike `default constructor` where the body of the constructor is empty.

Although you may see some people claim that that default and no-arg constructor is same but in fact they are not, even if you have a `no-arg constructor` it cannot be called `default constructor` since you have

written the code of it.

- Below you can see a snippet with a `no-arg` constructor:

```
class Demo
{
    public Demo()
    {
        System.out.println("This is a no argument constructor");
    }
    public static void main(String args[]) {
        new Demo();
    }
}
```

Parameterized constructor

- The constructor with arguments(or you can say parameters) is known as the `Parameterized constructor`.
- If you remember from the previous lesson, we have created the `Car` class with a constructor where we have initialized all the fields from that class. The constructor looked like the one in the snippet below:

```
    public Car(String brand, String model, int doors, int
engineCapacity, String color) {
        this.brand = brand;
        this.model = model;
        this.doors = doors;
        this.engineCapacity = engineCapacity;
        this.color = color;
      }
```

- When you call the constructor to initialize the object, you should specify values for each of the parameters as in the snippet below:

```
Car bogdanCar = new Car("Volkswagen", "Golf", 5, 1500, "Blue");
```

- How this constructor works is very similar with how the methods with parameters works. Even if you have parameters of same type, the orders in which you pass them it matters! For example, we have the same type for the the `brand` and `model`, namely `String`, and if we swap them, the program will work but we just created something which is called a logical error. This refers that we have initialized our state with wrong values as there is no Car maker in the world called `Golf`.

Constructor usage

- As with methods where you can have multiple methods with same name but different number of parameters, the same applies to `Constructors`. Also, again, this is called `overloading` but this time is `constructor overloading`.
- In the example below, we design the state of the class `Sandwich`. And due to the fact that we can create a sandwich in multiple ways, we define multiple constructors to initialize the fields:

```
    public class Sandwich {
        private String bread;
        private String meat;
        private int slicesOfTomatoes;
        private boolean mustard;
        private boolean ketchup;


        public Sandwich() {
            //This is the no-arg constructor
        }

        //This is the parameterized constructor
        public Sandwich(String bread, String meat, int
slicesOfTomatoes, boolean mustard, boolean ketchup) {
            this.bread = bread;
            this.meat = meat;
            this.slicesOfTomatoes = slicesOfTomatoes;
            this.mustard = mustard;
            this.ketchup = ketchup;
        }


        public String getBread() {
            return bread;
        }

        public void setBread(String bread) {
            this.bread = bread;
        }

        public String getMeat() {
            return meat;
        }

        public void setMeat(String meat) {
            this.meat = meat;
        }

        public int getSlicesOfTomatoes() {
            return slicesOfTomatoes;
        }

        public void setSlicesOfTomatoes(int slicesOfTomatoes) {
            this.slicesOfTomatoes = slicesOfTomatoes;
```

```
    }

    public boolean isMustard() {
        return mustard;
    }

    public void setMustard(boolean mustard) {
        this.mustard = mustard;
    }

    public boolean isKetchup() {
        return ketchup;
    }

    public void setKetchup(boolean ketchup) {
        this.ketchup = ketchup;
    }

}
```

- Now let's explain a bit what happened while creating this class
- We have the fields that we envisioned to use while creating instances of the `Sandwich` class, nothing new so far.
- Next, we have two constructors, the no-arg constructor and the parameterized constructor.
    - It might be confusing why do we need the no-arg constructor. Well, think that you might use the no-arg constructor to create a `Sandwich` object and then, you will add whatever you want to it, maybe skipping some ingredients. It is just like you order a Schawrma and thell to the guy who is selling it that you don't want tomatoes, for example. See the snippet below for an example of how this no-arg constructor might be useful:

```
public class Application {

    public static void main(String[] args) {
        Sandwich customSandwich = new Sandwich();
        customSandwich.setBread("Whole grain");
        customSandwich.setMeat("Turkey");
        customSandwich.setKetchup(true);

        System.out.println(customSandwich.getBread());
        System.out.println(customSandwich.getMeat());
        System.out.println(customSandwich.isKetchup());

    }

}
```

    - As we said earlier, our newly created object makes use of the `no-arg` constructor and then manually set the ingredients that we want.

- Now, the usage of the next constructor is similar to what we have seen earlier this lesson and also in the previous lesson:

```
    public class Application {

        public static void main(String[] args) {

            Sandwich sandwich = new Sandwich("Multigrain", "Chicken",
3, true, false);
            System.out.println(sandwich.getBread());
            System.out.println(sandwich.getMeat());
            System.out.println(sandwich.getSlicesOfTomatoes());
            System.out.println(sandwich.isMustard());
            System.out.println(sandwich.isKetchup());
        }

    }
```

- There is still, another scenario where we have multiple defined parameterized constructors with a variable number of parameters. If we want, each constructor can call, inside other constructor from the same class using `this()`. Here, this can be seen as the constructor that we want to call and the difference will be made by what we pass to it between the paranthesis.

- See the below snippet where we have multiple defined constructors for the `Sandwich` class:

```
    public Sandwich() {
        //This is the no-arg constructor
    }

    //This is the parameterized constructor
    public Sandwich(String bread, String meat, int slicesOfTomatoes,
boolean mustard, boolean ketchup) {
        this.bread = bread;
        this.meat = meat;
        this.slicesOfTomatoes = slicesOfTomatoes;
        this.mustard = mustard;
        this.ketchup = ketchup;
    }


    public Sandwich(String bread, String meat) {
        this(bread, meat, 3, true, true);
    }

    public Sandwich(String bread) {
        this(bread, "turkey", 3, true, true);
    }
```

- Now you might wonder, why would I need to call another constructor from a constructor?
- The idea is that you never want to have uninitialized fields inside a class
- Thus, you can say that you give default values for the parameters that are missing
    - In our case, if we are a Sandwich factory, we can say that we have some predefined Sandwiches where we only let customer choose the bread and meat, or even only the bread and then we choose the ingredients.
- Just to make this long story short, constructors are used for initializing the fields from a class.
- Most of the time, you want your constructor to initialize all of your fields, even if you don't have parameters for all of them (now you know how to call another constructor from the current constructor).

The `this` keyword

- The `this` keyword can be used in a few scenarios when using the JAVA programming language:
    1. `this` can be used to refer current class instance variable
    2. `this` can be used to invoke current class method (implicitly)
    3. `this()` can be used to invoke current class constructor.

`this` can be used to refer current class instance variable

- If you remember from our previous sandwich class where we had the parameterized constructor, we initialized the fields by firstly referencing them with the `this` keyword and then setting its value to the associated parameter like in the example below:

```
    public class Sandwich {
        private String bread;
        private String meat;
        private int slicesOfTomatoes;
        private boolean mustard;
        private boolean ketchup;

        public Sandwich(String bread, String meat, int
slicesOfTomatoes, boolean mustard, boolean ketchup) {
            this.bread = bread;
            this.meat = meat;
            this.slicesOfTomatoes = slicesOfTomatoes;
            this.mustard = mustard;
            this.ketchup = ketchup;
        }
```

- In the previous scenario, if we don't use the `this` keyword, our fields will not be initialized. More, eclipse will give you an warning saying that the initialization has no effect. Just like in the image below:

```
public class Sandwich {
    private String bread;
    private String meat;
    private int slicesOfTomatoes;
    private boolean mustard;
    private boolean ketchup;


    public Sandwich() {
        //This is the no-arg constructor
    }

    //This is the parameterized constructor
    public Sandwich(String bread, String meat, int slicesOfTomatoes, boolean mustard, boolean ketchup) {
        bread = bread;
        meat = meat;
        slicesOfTomatoes = slicesOfTomatoes;
        mustard = mustard;
        ketchup = ketchup;
    }
```

- Now if we run the code from the image above, we will get the following output:

```
    public static void main(String[] args) {

        Sandwich sandwich = new Sandwich("Multigrain", "Chicken", 3,
true, false);
        System.out.println(sandwich.getBread());
        System.out.println(sandwich.getMeat());
        System.out.println(sandwich.getSlicesOfTomatoes());
        System.out.println(sandwich.isMustard());
        System.out.println(sandwich.isKetchup());
}
```

```
    null
    null
    0
    false
    false
```

  - and that's because, even if we use the same name as the fields of our class, we are not pointing to them! This can be done using the `this` keyword.

`this` can be used to invoke current class method (implicitly)

- We haven't touched this part so far, explicitly, but you can also have methods defined in a `class`.
- Now let's a imagine a very simple class related to the engine of a car. The class will have only two fields for the capacity and to indicate if the engine is started or not.
- Then, we create the getters and setters for this class and another method called 'startEngine'. The method, will call the setter of the field that indicates if the engine is started or not and set its value to `true`:

```
    public class CarEngine {
        private int capacity;
        private boolean isEngineStarted;
```

```
        public CarEngine(int capacity, boolean isEngineStarted) {
            this.capacity = capacity;
            this.isEngineStarted = false;
        }


        public void startEngine() {
            this.setEngineStarted(true);
        }


        public int getCapacity() {
            return capacity;
        }

        public void setCapacity(int capacity) {
            this.capacity = capacity;
        }

        public boolean isEngineStarted() {
            return isEngineStarted;
        }

        public void setEngineStarted(boolean isEngineStarted) {
            this.isEngineStarted = isEngineStarted;
        }

    }
```

- You can see that the setter is called using the `this` keyword. Is like saying go to the instance of the current object and check if there is a method called like this.
- There is one catch here, also. You can call the method without using the `this` keyword but doing it like that, will make the code less readable.

Homework exercises

1. Again, try to give a real word example of `Encapsulation`.
2. Design the class for a `Radio`. Follow the guidelines related to `Encapsulation` when creating the state and behavior of it.
3. A class called circle is designed as it follows:

- Two private instance variables: radius (of the type double) and color (of the type String)
- Two overloaded constructors
    - a default constructor with no argument
    - a constructor which takes a double argument for radius.
- Two public methods: getRadius() and getArea(), which return the radius and area of this instance, respectively.
- Now, perform the following requirements:

    - First, design the class as per the statement of the exercises
    - Constructor: Modify the class Circle to include a third constructor for constructing a Circle instance with two arguments - a double for radius and a String for color
    - Getter: Add a getter for variable color for retrieving the color of this instance.
    - Setter: Is there a need to change the values of radius and color of a Circle instance after it is constructed? If so, add two public methods called setters for changing the radius and color of a Circle instance

Note: Try to create multiple objects of this class and then play with them in order to observe their behaviour.

Guidelines

- Try to give an answer for each of the questions in the recap section
- All of them are questions with big chance of meeting them in an interview and apart from this, they are very important in the `Object Oriented Programming` paradigm.