Java Course

Object-Oriented Programming

Table of Content

Object Oriented Programming

- Java is an object-oriented programming language
- The core concept of OOP is to break complex problems into smaller pieces
- Core concepts:
  - Objects
  - Classes
  - Abstraction
  - Encapsulation
  - Polymorphism
  - Inheritance

Java Class

- A class is a blueprint for creating objects
- With classes, we define fields (state) and methods (behaviour) of objects
- We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.
- Since many houses can be made from the same sketch, we can create many objects from a class.

Java Class

- Class contains fields (state) and methods (behaviour) of the object.

```
class ClassName {
  // fields
  // methods
}
```

```java
public class Car {

    // fields (state)
    String manufacturer;
    String model;
    int year;
    String color;


    // methods (behavior)
    void start() {
        System.out.println("Starting the engine");
    }

    void stop() {
        System.out.println("Stopping the car");
    }
}

```

Java Objects

- Objects are created from classes
- Object is an **instance** of the class
- Objects are created using `new` keyword

```
public static void main(String[] args) {
    Car volvo = new Car();

}
```

Java Object Constructor

- Constructor is a special method that is implicitly invoked when an object is being created
- Constructor has the same name as the name of the class, and does not have any return type
- Constructor **does not have** a return type!

Constructors

- 3 types:
  - No-Arg constructor
  - Parameterized constructor
  - Default constructor

No-Arg Constructor

```java
public class Car {

    // fields (state)
    String manufacturer;
    String model;
    int year;
    String color;

    // No-Arg constructor
    Car() {
        // body of constructor method
        year = 2022;
    }


    // methods (behavior)
    void start() {
        System.out.println("Starting the engine");
    }
```

Parametrized Constructor

- Constructor method can accept one or more parameters

```
2
3  public class Car {
4
5      // fields (state)
6      String manufacturer;
7      String model;
8      int year;
9      String color;
10
11     // Parametrized constructor
12     Car(String manufacturer, String model, int year, String color) {
13         this.manufacturer = manufacturer;
14         this.model = model;
15         this.year = year;
16         this.color = color;
17     }
18
```

Default Constructor

- If we don’t create any constructor, Java compiler automatically creates a no-arg constructor during the execution of the program - default constructor
- Default constructor initializes any uninitialized instance variables with default values

Type	Default Value
`boolean`	false
`byte`	0
`short`	0
`int`	0
`long`	0L
`char`	\u0000
`float`	0.0f
`double`	0.0d
`object`	Reference null

Method Overloading

- In Java, two or more methods may have the same name if they differ in parameters:
  - different number of parameters
  - different types of parameters
  - or both
- This reflects on **constructor** methods as well

Method Overloading

```
    // we can create two methods with different name
    // with the same behaviour
    int plusMethodInt(int x, int y) {
        return x + y;
    }

    double plusMethodDouble(double x, double y) {
        return x + y;
    }

    // or we can OVERLOAD method and have two methods
    // with the same name doing the same thing
    int plusMethod(int x, int y) {
        return x + y;
    }

    double plusMethod(double x, double y) {
        return x + y;
    }
```

Access Modifiers

- Access modifiers are used to set the accessibility (visibility) of classes, variables, methods, constructors, interfaces,..
- 4 types of access modifiers:
  - Default - visible within the package
  - Private - visible within the class
  - Protected - visible within the package or all subclasses
  - Public - visible everywhere

Default Access Modifier

Visible within the package

- If we don’t explicitly specify any access modifier for classes, methods, variables, etc, then by default - the default access modifier is considered
- Visibility scope: within the package

Public Access Modifier

Visible everywhere

- When methods, variables, classes, etc, are declared `public`, then we can access them from anywhere
- Visibility scope: everywhere

Private Access Modifier

Visible within the class

- When methods, variables, etc, are declared `private`, they cannot be accessed outside of the class
- Visibility scope: within the class

Protected Access Modifier

Visible within the package or all subclasses

- When methods and variables, are declared `protected`, we can access them within the same package as well as from subclasses
- Visibility scope: within the package or all subclasses
- Note: We cannot declare classes or interfaces `protected` in Java

`this` keyword

- In Java, this refers the current object inside a method or a constructor
- It's often used for variable name ambiguity

```
2
3  public class Person {
4
5      private String firstName;
6
7      // constructor which takes firstName as parameter
8      // and sets object's property value
9      public Person(String firstName) {
10         // we have firstName parameter, and
11         // firstName as class property
12         firstName = firstName;       // this is a problem
13
14         this.firstName = firstName; // BUT, this is OK!
15     }
16
17
18 }
```

Getters & Setters

- Used for accessing and manipulating values of class fields
- Getter (*accessor*) method - returns class field value
- Setter (*mutator*) method - updates class field value
- By convention, getter method name starts with get, and setter method name starts with set

```java
public class Person {

    private String firstName;

    // constructor
    public Person(String firstName) {
        this.firstName = firstName;
    }

    // getter
    public String getFirstName() {
        return this.firstName;
    }

    // setter
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

```

Why should we use getters & setters?

- Getters & setters allows us to *encapsulate* sensitive data and hide it from users
- Provides better control of class property values
- Class properties (attributes) can be made *read-only* or *write-only*
- Setter method can be used for validating data before updating property value

```java
public class Person {

    private int age;

    // default constructor
    public Person() {

    }
    // constructor
    public Person(int age) {
        this.age = age;
    }

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

    // setter
    public void setAge(int age) {
        if (age < 0) {
            System.out.println("Age cannot be a negative number");
            // throw some exception
            return;
        }
        this.age = age;
    }
}
```

```java
package day5;

public class Main {

    public static void main(String[] args) {
        Person person = new Person();

        // we can't access the age property of a person
        // because it's a private property
        // but we can use getter method to access it
        System.out.println(person.getAge());        // here we have a default value of 0

        // we can't update it directly because it's private
        // but we can update it using setter method
        person.setAge(25);
        System.out.println(person.getAge());

        // if we try to set age to negative number, we'll
        // get the message that the operation is not possible
        // and person will still be 25 years old from before
        person.setAge(-20);
        System.out.println(person.getAge());



    }

}
```

Exercise 1

- Create a class called Circle
- Circle has 2 properties: radius and colour
- Create a no-arg constructor which should set default radius and colour of Circle (default circle should be red with any radius you want for default)
- Create a parameterized constructor which takes both radius and colour as parameters and sets Circle properties to the provided values
- Create getters & setters for both fields
- Make sure radius can’t be set to a negative value (hint: setter method)
- Create a method for calculating circle area (r^2 * 3.14) - method should return calculated value
- What should we do if we want the colour to be read-only?
- Create Main class and main method to test it (instantiate a Circle, test getters & setters, test implemented method for calculating area,…)

Exercise 2

- Create a class Car
- Car has only one boolean property holding the information if the engine is running
- Create an empty constructor (no-arg) with the default value of the property
- Create two methods: one for starting the car, and one for stopping the car
- Create a getter method for property
- Create a method called power that takes no arguments and toggles the state of boolean property
- Create a Main class and main method to test everything (instantiate one Car, call methods for starting and stopping the car, check state of Car property after starting/stopping the car,..)

Exercise 3

- Create a class Student
- Student has first name, last name, JMBG, college, department
- Create a default (empty - no-arg) constructor
- Create a parametrised constructor
- Create getters & setters
- Create a method for JMBG validation that receives one parameter - JMBG string, and checks if JMBG has exactly 13 characters (think about return type of this method)
- Call this method inside of parametrised constructor and JMBG setter method (if provided JMBG is not valid, we don't want to set it)
- Create a method that prints all the information about student, something like:
  Student: Pera Peric (1308993222111)
  Faculty: FTN, Software Engineering
- In Main, try different approaches for creating students (create a student with empty constructor and set properties with setters; create a student with parametrised constructor)
- *Let's change the method for printing to be overridden `toString` method. We will se what this means