Java SE & Spring

Module 1: Java SE

06.Object Oriented Programming (OOP)

What is OOP?

Java is an object-oriented programming language. The core concept of the object-oriented approach is to break complex problems into smaller objects. OOP is about creating objects that contain both data and methods. At first, in order to make clear the context, we should understand what the **class** and the **object** is.

Class: A class is a blueprint for the object. Before we create an object, we first need to define the class. Here are a few examples of classes:

class Vegetable	**class** Animal	**class** Mammal	**class** Tree	**class** Bicycle

What is OOP?

- 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 description, we can create many objects from a class.
- Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight, color and model name etc. , and methods, such as drive and brake.

What is OOP?

Creating Class

Syntax:

```
class ClassName {
  // fields (states)
  // methods (behaviours)
}
```

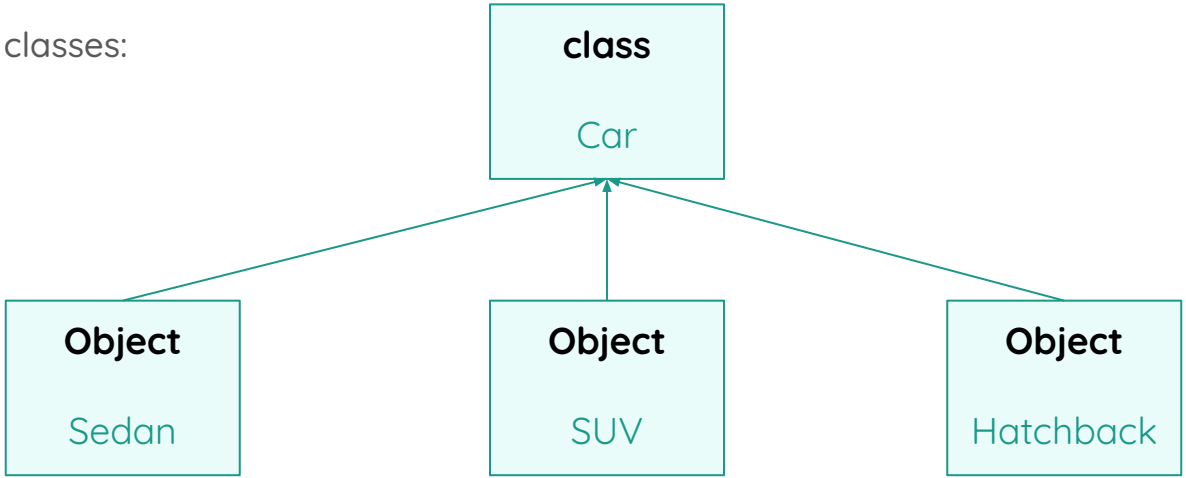
Example:

```
class Car {

  // state or field
  private String color = “Aqua Green”;
  private double weight = 1350.0;

  // behavior or method
  public void braking() {
    System.out.println("Slowing!");
  }
}
```

What is OOP?

Object: Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive and brake.

A class is a blueprint for the object. Before we create an object, we first need to define the class. Here are a few examples of classes:



Creating Objects

In Java, here is how we can create an object:

Syntax:

```
className object = new className();
```

Example:

```
1  // creating objects for Car class
2  Car sedan = new Car();
3  Car suv = new Car();
4  Car hatchback = new Car();
```

The `new Car()` statement is called object creation statement or object instantiation statement, so when we say creating an instance, this means creating an object.

Methods

A method is a block of code that performs a specific task. Dividing a complex problem into smaller chunks makes your program easy to understand and reusable. In Java, there are two types of methods:

- **User-defined Methods:** We can create our own method based on our requirements.
- **Standard Library Methods:** These are built-in methods in Java that are available to use.

Syntax:

```
modifier static returnType nameOfMethod (parameters ...) {
  // method body
}
```

- **modifier** defines access types whether the method is **`public`**, **`private`**, **`protected`** or **`default`**.
- **`static`** specifies that method whether can be accessed without creating objects or not. We will discuss static keyword on next slides.

Methods

Syntax:

```
1  modifier static returnType nameOfMethod (parameters ...) {
2    // method body
3  }
```

- **nameOfMethod** is an identifier that is used to refer to the particular method in a program. For example, the sqrt() method of standard Math class is static. Hence, we can directly call Math.sqrt() without creating an instance of Math class.
- **parameters** are values passed to a method. We can pass any number of arguments to a method.

Further reading: https://www.programiz.com/java-programming/methods

Calling Methods

Syntax:

```
// calls the method
addNumbers();
```

Example:

```
class Pet {

  // state or field
  private String name = "Snowy";
  private String species = "Dog";
  private int birthYear = 2018;

  // behavior or method
  public int calculateAge(int currentYear){
    return 2021 - this.birthYear;
  }
}
```

```
// calls the calculateAge method
calculateAge(2021);
```

Constructors

Code sample below that we discussed in previous slides, We have used the **new** keyword along with the constructor of the class to create an object.

```
1  // creating objects for Car class
2  Car sedan = new Car();
3  Car suv = new Car();
4  Car hatchback = new Car();
```

`Car()` is the constructor of the `Car` class. A constructor in Java is similar to a method that is invoked when an object of the class is created. Unlike Java methods, a constructor has the same name as that of the class and does not have any return type.

Properties of Constructors

- Constructors are invoked implicitly when you instantiate objects.
- The two rules for creating a constructor are:
  - The name of the constructor should be the same as the class.
  - A Java constructor must not have a return type.
- If a class doesn't have a constructor, the Java compiler automatically creates a default constructor during run-time. The default constructor initializes instance variables with default values. For example, the int variable will be initialized to 0.
- A constructor cannot be **`abstract`** or **`static`** or **`final`**.
- A constructor can be overloaded but can not be overridden.

Constructor Types

In Java, constructors can be divided into 3 types:

- **No-Arg Constructor:** Similar to methods, a Java constructor may or may not have any parameters (arguments). If a constructor does not accept any parameters, it is known as a no-argument constructor.

Syntax:

```
accessModifier Constructor() {
   // body of the constructor
}
```

Example:

```
// creating objects with no-arg constructor for Car class
Car sedan = new Car();
Car suv = new Car();
Car hatchback = new Car();
```

Constructor Types

- **Parameterized Constructor:** A Java constructor can also accept one or more parameters. Such constructors are known as parameterized constructors (constructor with parameters).

Syntax:

```
accessModifier Constructor(dataType param1, dataType param2,...) {
   // body of the constructor
}
```

Example:

```
public Pet(String species, String name,  int birthYear){
  this.species = species;
  this.name = name;
  this.birthYear = birthYear;
}
```

Constructor Types

- **Default Constructor:** If we do not create any constructor, the Java compiler automatically create a no-arg constructor during the execution of the program. This constructor is called default constructor. The default constructor (no-args constructor too) initializes any uninitialized instance variables with default values.

Access Modifiers

Modifier	Description
`public`	accessible everywhere
`protected`	accessible in the same package and in sub-classes
default	accessible only in the same package
`private`	accessible only in the same class

- **public:** When methods, variables, classes, and so on are declared public, then we can access them from anywhere. The public access modifier has no scope restriction.
- **protected:** When methods and data members are declared protected, we can access them within the same package as well as from subclasses.
- **default:** The code is only accessible in the same package. This is used when you don't specify a modifier.
- **private:** When variables and methods are declared private, they cannot be accessed outside of the class.

➔ this keyword

this keyword is used to refer to the current object inside a method or a constructor. We already used this keyword in our examples in parameterized constructor below.

```
1  public Pet(String species, String name,  int birthYear){
2    this.species = species;
3    this.name = name;
4    this.birthYear = birthYear;
5  }
```

We can use this keyword in these cases: for Ambiguity Variable Names, with Getter and Setter methods, in Constructor Overloading and using as an argument for methods. We will cover these cases in code samples in Intellij IDEA.

`this`, `final` and `static` keywords

➔ **`final` keyword**

Once any entity (variable, method or class) is declared final, it can be assigned only once. Those are:

1. the **`final`** variable cannot be reinitialized with another value
2. the **`final`** method cannot be overridden
3. the **`final`** class cannot be extended

1. **`final`** variable

If we declare a variable as **`final`**, then we should assign a value to it while declaring to it or in a constructor.

2nd and 3rd use cases will be discussed in next chapter in inheritance section.

➔ `static` keyword

If we want to access class members, we must first create an instance of the class. But there will be situations where we want to access class members without creating any variables. In those situations, we can use the **`static`** keyword in Java. If we want to access class members without creating an instance of the class, we need to declare the class members static.

Syntax:

```
1  //static method
2  modifier static returnType nameOfMethod (parameters ...) {
3    // method body
4  }
5  //static variable
6  modifier static dataType nameOfVariable;
```

Nested and Inner Classes

In Java, you can define a class within another class. Such class is known as nested class. The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.

Syntax:

```
class OuterClass {
    // ...
    class NestedClass {
        // ...
    }
}
```

There are two types of nested classes you can create in Java:

- Non-static nested class (inner class)
- Static nested class

Nested and Inner Classes

Example:

```
1  class OuterClass {
2    int x = 10;
3
4    class InnerClass {
5      int y = 5;
6    }
7  }
8
9  public class Main {
10   public static void main(String[] args) {
11     OuterClass myOuter =  new OuterClass();
12     OuterClass.InnerClass myInner = myOuter. new InnerClass();
13     System.out.println(myInner.y + myOuter.x);
14   }
15 }
```

`private` Inner Classes

Unlike a "regular" class, an inner class can be **`private`** or **`protected`**. If you don't want outside objects to access the inner class, declare the class as **`private`**:

Example:

```
1  class OuterClass {
2    int x = 10;
3
4    private class InnerClass {
5      int y = 5;
6    }
7  }
8
9  public class Main {
10   public static void main(String[] args) {
11     OuterClass myOuter =  new OuterClass();
12     OuterClass.InnerClass myInner = myOuter. new InnerClass();
13     System.out.println(myInner.y + myOuter.x);
14   }
15 }
```

If you try to access a private inner class from an outside class (Main etc.), an error occurs.

`static` Inner Classes

An inner class can also be **`static`**, which means that you can access it without creating an object of the outer class.

Example:

```
1  class OuterClass {
2    int x = 10;
3
4    static class InnerClass {
5      int y = 5;
6    }
7  }
8
9  public class Main {
10   public static void main(String[] args) {
11     OuterClass myOuter =  new OuterClass();
12     OuterClass.InnerClass myInner = new OuterClass.InnerClass();
13     System.out.println(myInner.y + myOuter.x);
14   }
15 }
```

As like static attributes and methods, a static inner class does not have access to members of the outer class.

Practice Time: Let’s Code Together!

Let’s code together!🚀

📌**Exercise:**

Create a class named Employee which has following states (properties) and behaviours(methods):

1. `id` in type of integer , `name` and `surname` (separately) as String type, age and birthYear in type of integer.
2. `phoneNumber, address, departmentName, officeLocation`in type of String
3. Each field should have getter and setter methods.
4. No-Args and Parameterized constructor should be exist.
5. Each fields should be instantiated through constructors.
6. name and surname field maybe final, officeLocation maybe static.
7. The class should has a calculateAge method which it takes currentYear as parameter and calculates the employee’s age and assign it to age field.

Questions ?

Next: OOP in Depth