DAY 2

Object Oriented Concepts in JAVA

- Object-Oriented Programming Concepts are very important for programming. Without having an idea about OOPS concepts, you will not be able to design systems in the object-oriented programming model.
- The object-oriented programming model revolves around the concept of Objects

1. WHAT IS AN OBJECT???????

An object is an **instance of a Class**. It contains properties and functions. They are like real-world objects. For example, your car, house, laptop, etc. are all objects. They have some specific properties and methods to perform some action.

2. WHAT IS A CLASS?????

The Class defines the **blueprint of Objects**. They define the **properties and functionalities of the objects**. For example, Laptop is a class and your laptop is an instance of it.

Core OOPS concepts are:

- **Abstraction**
- **Encapsulation**
- **Polymorphism**
- **Inheritance**
- **Association**
- **Aggregation**
- **Composition**

1.ABSTRACTION:-

Objects are the building blocks of Object-Oriented Programming. An object contains some properties and methods. We can hide them from the outer world through access modifiers. We can provide access only for required functions and properties to the other programs. This is the general procedure to implement abstraction in OOPS.

Abstraction in real life:-Your car is a great example of abstraction. You can start a car by turning the key or pressing the start button. You don’t need to know how

the engine is getting started, what all components your car has. The car internal implementation and complex logic is completely hidden from the user.

In the example below, you can see an abstract class called `Animal` with two abstract and one concrete method.

```
abstract class Animal {
	// abstract methods
	abstract void move();
	abstract void eat();

	// concrete method
	void label() {
		System.out.println("Animal's data:");
	}
}
```

Extend the `Animal` abstract class with two child classes: `Bird` and `Fish`. Both of them set up their own functionality for the `move()` and `eat()` abstract methods.

```
class Bird extends Animal {

void move() {
		System.out.println("Moves by flying.");
     }
	void eat() {
		System.out.println("Eats birdfood.");
}
}

class Fish extends Animal {
		void move() {
			System.out.println("Moves by swimming.");
	    }
		void eat() {
			System.out.println("Eats seafood.");
		}
}
```

Now, test it with the `TestBird` and `TestFish` classes. Both call the one concrete (`label()`) and the two abstract (`move()` and `eat()`) methods.

```
class TestBird {
	public static void main(String[] args) {
		Animal myBird = new Bird();

		myBird.label();
		myBird.move();
```

```
                myBird.eat();
        }
}

class TestFish {
        public static void main(String[] args) {
                Animal myFish = new Fish();

                myFish.label();
                myFish.move();
                myFish.eat();
        }
}
```

In the console, the concrete method has been called from the `Animal` abstract class, while the two abstract methods have been called from `Bird()` and `Fish()`, respectively.

```
[output of TestBird]
Animal's data:
Moves by flying.
Eats birdfood.

[output of TestFish]
Animal's data:
Moves by swimming.
Eats seafood.
```

2. Encapsulation

Encapsulation is the technique used to implement abstraction in object-oriented programming. Encapsulation is used for access restriction to class members and methods.

Access modifier keywords are used for encapsulation in object oriented programming. For example, encapsulation in java is achieved using private, protected and public keywords.

The `Animal` class below is fully encapsulated. It has three private fields and each of them has its own set of getter and setter methods.

```
class Animal {
	private String name;
	private double averageWeight;
	private int numberOfLegs;

	// Getter methods
	public String getName() {
		return name;
	}
	public double getAverageWeight() {
		return averageWeight;
	}
	public int getNumberOfLegs() {
		return numberOfLegs;
	}

	// Setter methods
	public void setName(String name) {
		this.name = name;
	}
	public void setAverageWeight(double averageWeight) {
		this.averageWeight = averageWeight;
	}
	public void setNumberOfLegs(int numberOfLegs) {
		this.numberOfLegs = numberOfLegs;
	}
}
```

The `TestAnimal` class first sets a value for each field with the setter methods, then prints out the values using the getter methods.

```
public class TestAnimal {
	public static void main(String[] args) {
		Animal myAnimal = new Animal();

		myAnimal.setName("Eagle");
		myAnimal.setAverageWeight(1.5);
		myAnimal.setNumberOfLegs(2);

		System.out.println("Name: " + myAnimal.getName());
		System.out.println("Average weight: " +
myAnimal.getAverageWeight() + "kg");
		System.out.println("Number of legs: " +
myAnimal.getNumberOfLegs());
	}
}
```

As you can see below, the Java console returns properly all the values you set with the setter methods:

```
Name: Eagle
Average weight: 1.5kg
Number of legs: 2
```

3. Polymorphism

Polymorphism is the concept where an object behaves differently in different situations. There are two types of polymorphism – compile time polymorphism and runtime polymorphism.

- Compile-time polymorphism is achieved by method overloading. For example, we can have a class as below.

```
public class Circle {

    public void draw(){

        System.out.println("Drwaing circle with default color Black and diameter 1 cm.");

                    }

    public void draw(int diameter){

            System.out.println("Drwaing circle with default color Black and diameter"+diameter+" cm.");

                                }

    public void draw(int diameter, String color){

            System.out.println("Drwaing circle with color"+color+" and diameter"+diameter+" cm.");

            }

            }
```

Here we have multiple draw methods but they have different behavior. This is a case of method overloading because all the methods name is same and arguments are different. Here compiler will be able to identify the method to invoke at compile-time, hence it's called compile-time polymorphism.

Runtime polymorphism is implemented when we have an "IS-A" relationship between objects. This is also called a method overriding because the subclass has to override the superclass method for runtime polymorphism.

4.Inheritance

Inheritance is the object-oriented programming concept where an object is based on another object. Inheritance is the mechanism of code reuse. The object that is getting inherited is called the superclass and the object that inherits the superclass is called a subclass.

We use extends keyword in java to implement inheritance. Below is a simple example of inheritance in java.

```
package java.examples1;

class SuperClassA {

	public void foo(){
		System.out.println("SuperClassA");
	}

}

class SubClassB extends SuperClassA{

	public void bar(){
		System.out.println("SubClassB");
	}

}

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

```
                SubClassB a = new SubClassB();

                a.foo();
                a.bar();
        }
}
```

In the example, the `Eagle` class extends the `Bird` parent class. It inherits all of its fields and methods, plus defines two extra fields that belong only to `Eagle`.

```
class Bird {
        public String birth= "egg";
        public String outerCovering = "feather";

        public void flyUp() {
                System.out.println("Flying up...");
        }
        public void flyDown() {
                System.out.println("Flying down...");
        }
}

class Eagle extends Bird {
        public String name = "eagle";
        public int lifespan = 15;
}
```

The `TestEagle` class instantiates a new `Eagle` object and prints out all the information (both the inherited fields and methods and the two extra fields defined in the `Eagle` class).

```
class TestEagle {
        public static void main(String[] args) {
                Eagle myEagle = new Eagle();

                System.out.println("Name: " + myEagle.name);
                System.out.println("Reproduction: " +
myEagle.birth);
                System.out.println("Outer covering: " +
myEagle.outerCovering);
                System.out.println("Lifespan: " + myEagle.lifespan);

                myEagle.flyUp();
                myEagle.flyDown();
        }
```

```
}
[output of TestEagle]
birth: another egg
Outer covering: feather
Lifespan: 15
Flying up...
Flying down...
```

5.Association

Association is the OOPS concept to define the relationship between objects. The association defines the multiplicity between objects. For example Teacher and Student objects. There is a one-to-many relationship between a teacher and students. Similarly, a student can have a one-to-many relationship with teacher objects. However, both student and teacher objects are independent of each other.

6.Aggregation

Aggregation is a special type of association. In aggregation, objects have their own life cycle but there is ownership. Whenever we have “HAS-A” relationship between objects and ownership then it’s a case of aggregation.

7. Composition

The composition is a special case of aggregation. The composition is a more restrictive form of aggregation. When the contained object in “HAS-A” relationship can’t exist on its own, then it’s a case of composition. For example, House has-a Room. Here the room can’t exist without the house. Composition is said to be better than inheritance