Introduction to Object Oriented Programming

Programming Paradigms

Programming paradigms are different ways or styles in which a given program or programming language can be organized. Each paradigm consists of certain structures, features, and opinions about how common programming problems should be tackled.

The question of why are there many different programming paradigms is similar to why are there many programming languages. Certain paradigms are better suited for certain types of problems, so it makes sense to use different paradigms for different kinds of projects. They're more like a set of ideals and guidelines that many people have agreed on, followed, and expanded upon.

Programming languages aren't always tied to a specific paradigm. There are languages that have been built with a certain paradigm in mind and have features that facilitate that kind of programming more than others (Haskel and functional programming is a good example). But there are also "multi-paradigm" languages, meaning you can adapt your code to fit a certain paradigm or another (JavaScript and Python are good examples).

Broadly speaking, the paradigms can be classified into two major types of programming paradigms:

Imperative - an imperative program consists of commands for the computer to perform to change state e.g. C, Java, Python, etc.

Declarative - focuses on what the program should accomplish without specifying all the details of how the program should achieve the result e.g. SQL, Lisp, Java etc.

Popular Programming Paradigms

Now that we have introduced what programming paradigms are and are not, let's go through the most popular ones, explain their main characteristics, and compare them.

- Imperative Programming
- Procedural Programming
- Object Oriented Programming
- Functional Programming
- Reactive Programming

1. Imperative Programming

Imperative programming consists of sets of detailed instructions that are given to the computer to execute in a given order. It's called "imperative" because as programmers we dictate exactly what the computer has to do, in a very specific way. Imperative programming focuses on describing how a program operates, step by step. Say you want to bake a cake. Your imperative program to do this might look like this

```
1- Pour flour in a bowl
2- Pour a couple eggs in the same bowl
3- Pour some milk in the same bowl
4- Mix the ingredients
5- Pour the mix in a mold
```

```
6- Cook for 35 minutes
7- Let chill
```

Using an actual code example, let's say we want to filter an array of numbers to only keep the elements bigger than 5. Our imperative code might look like this:

```
int nums[] = [1,4,3,6,7,8,9,2]
Arraylist<Integer> result = new Arraylist<>();

for (int i = 0; i < nums.length; i++) {
    if (nums[i] > 5) result.push_back(nums[i])
}
```

2. Procedural Programming

Procedural programming is a derivation of imperative programming, adding to it the feature of functions (also known as "procedures" or "subroutines").

In procedural programming, the user is encouraged to subdivide the program execution into functions, as a way of improving modularity and organization.

Following our cake example, procedural programming may look like this:

```
function pourIngredients() {
    - Pour flour in a bowl
    - Pour a couple eggs in the same bowl
    - Pour some milk in the same bowl
}

function mixAndTransferToMold() {
    - Mix the ingredients
    - Pour the mix in a mold
}

function cookAndLetChill() {
    - Cook for 35 minutes
    - Let chill
}

pourIngredients()
mixAndTransferToMold()
cookAndLetChill()
```

You can see that, thanks to the implementation of functions, we could just read the three function calls at the end of the file and get a good idea of what our program does.

That simplification and abstraction is one of the benefits of procedural programming. But within the functions, we still got same old imperative code.

3. Object-Oriented Programming

One of the most popular programming paradigms is object-oriented programming (OOP).

The core concept of OOP is to separate concerns into entities which are coded as objects. Each entity will group a given set of information (properties) and actions (methods) that can be performed by the entity.

OOP makes heavy usage of classes (which are a way of creating new objects starting out from a blueprint or boilerplate that the programmer sets). Objects that are created from a class are called instances.

Following our pseudo-code cooking example, now let's say in our bakery we have a main cook (called Frank) and an assistant cook (called Anthony) and each of them will have certain responsibilities in the baking process. If we used OOP, our program might look like this.

```
// Create the two classes corresponding to each entity
class Cook {
    String name;
    Cook (String name) {
        this.name = name
    }

    void mixAndBake() {
        - Mix the ingredients
        - Pour the mix in a mold
        - Cook for 35 minutes
    }
}

class AssistantCook {
    String name;
    AssistantCook (String name) {
        this.name = name
    }

    void pourIngredients() {
        - Pour flour in a bowl
        - Pour a couple eggs in the same bowl
        - Pour some milk in the same bowl
    }

    void chillTheCake() {
        - Let chill
    }
}

// Instantiate an object from each class
Cook Frank = new Cook('Frank')
AssistantCook Anthony = new AssistantCook('Anthony')

// Call the corresponding methods from each instance
Anthony.pourIngredients()
Frank.mixAndBake()
Anthony.chillTheCake()
```

What's nice about OOP is that it facilitates the understanding of a program, by the clear separation of concerns and responsibilities. Languages like C++, Java, Python etc. support Object Oriented Programming.

4.Declarative programming / Functional Programming

Declarative Programming is all about hiding away complexity and bringing programming languages closer to human language and thinking. It's the direct opposite of imperative programming in the sense that the programmer doesn't give instructions about how the computer should execute the task, but rather on what result is needed.

The basic objective of this style of programming is to make code more concise, less complex, more predictable, and easier to test compared to the legacy style of coding.

So far Java was supporting the imperative style of programming and object-oriented style of programming. The next big thing what java has been added is that Java has started supporting the functional style of programming with its Java 8 release.

The functional style of programming is declarative programming. In the imperative style of coding, we define what to do a task and how to do it. Whereas, in the declarative style of coding, we only specify what to do. Let’s understand this with an example. Given a list of number let’s find out the sum of double of even numbers from the list using an imperative and declarative style of coding.

```
// Java program to find the sum
// using imperative style of coding
import java.util.Arrays;
import java.util.List;
public class TestImperative {
    public static void main(String[] args)
    {
        List<Integer> numbers
            = Arrays.asList(11, 22, 33, 44,
                            55, 66, 77, 88,
                            99, 100);

        int result = 0;
        for (Integer n : numbers) {
            if (n % 2 == 0) {
                result += n * 2;
            }
        }
        System.out.println(result);
    }
}
```

The first issue with the above code is that we are mutating the variable result again and again. So mutability is one of the biggest issues in an imperative style of coding. The second issue with the imperative style is that we spend our effort telling not only what to do but also how to do the processing. Now let’s re-write above code in a declarative style.

```
// Java program to find the sum
// using declarative style of coding
```

```java
import java.util.Arrays;
import java.util.List;
public class GFG {
    public static void main(String[] args)
    {
        List<Integer> numbers
            = Arrays.asList(11, 22, 33, 44,
                            55, 66, 77, 88,
                            99, 100);

        System.out.println(
            numbers.stream()
                .filter(number -> number % 2 == 0)
                .mapToInt(e -> e * 2)
                .sum());
    }
}
```

The above example uses Java 8 Stream API. We will learn about Java 8 Streams, in later part of this LLD module. Language like Scala, Haskell, C# and Java also supports this paradigm as seen in above example.

5. Reactive Programming

Reactive programming is a programming paradigm that focuses on constructing responsive and robust software applications that can handle asynchronous data streams and change propagation, allowing developers to create scalable and more easily maintainable applications that can adapt to a dynamic environment. In a reactive system, data flows through streams, which can be thought of as sequences of events over time. Reactive programming revolves around the following principles - Data Streams, Observables, Observers and Operators. Java also supports this paradigm.

Motivation - Object Oriented Programming

Problem Statement Once upon a time in a software shop, two programmers were given the same spec and told to “build it”. The Project Manager forced the two coders - to compete. The problem statement is as follows: There will be shapes on GUI, a square, a circle and a triangle. When the user clicks the shape, it will rotate clockwise 360 degrees and play a .mp3 sound corresponding to that shape.

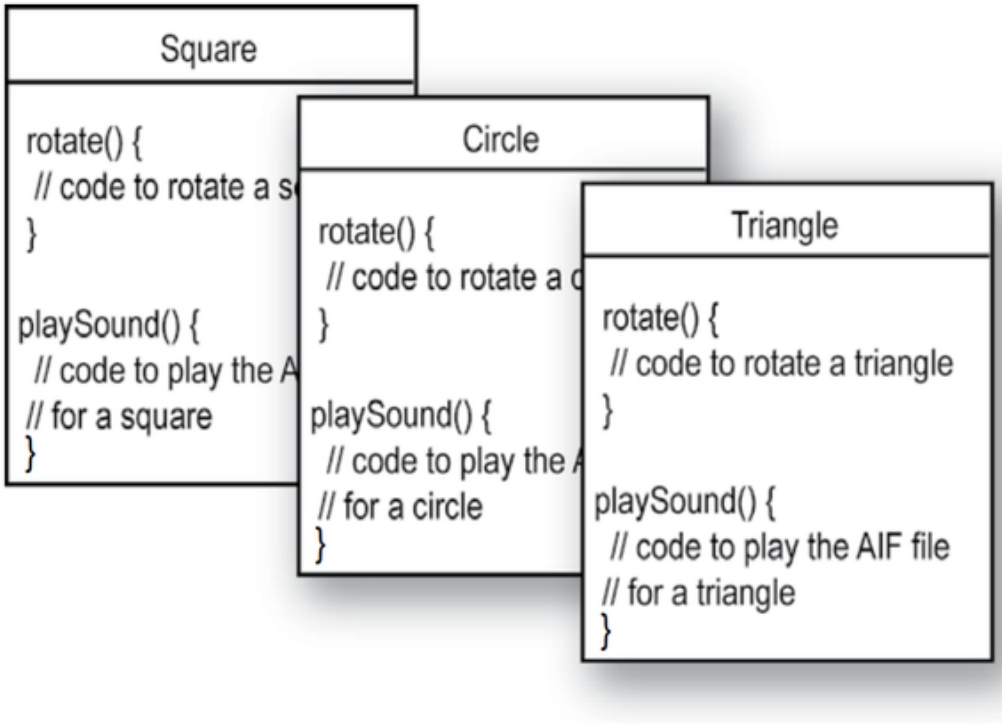
Coder 1 Focusses on writing procedures. He wrote the `rotate()` and `playSound()` procedure in no time.

```java
void rotate(int shapeNum){
    //code to rotate the shape about center
}

void playSound(int shapeNum){
        if(shapeNum==1){
             //play the square sound
        }
        else if(shapeNum==2){
            //play the circle sound
```

```
        }
        ...
}
```

Coder 2 Coder2 wrote a class for each of the three shapes.



Another Requirement comes... Now manager, added another requirement : There will be an amoeba shape on the screen, with the others. When the user clicks on the amoeba, it will rotate like the others, and play a .hif sound file.

Coder1: has to make changes in playSound method, rotate would still work!

```
void playSound(shapeNum) {
 // if the shape is not an amoeba,
        // use shapeNum to lookup which
        // AIF sound to play, and play it
 // else
        // play amoeba .hif sound
 }
```

It turned out not to be such a big deal, but it still made him queasy to touch previously-tested code. Of all people, he should know that no matter what the project manager says, the spec always changes.

Coder 2: smiled, sipped his Coffee, and wrote one new class. Sometimes the thing he loved most about OO was that he didn't have to touch code he'd already tested and delivered. "Flexibility, extensibility,..." he mused, reflecting on the benefits of Object Oriented Programming.

```
class Amoeba{
    void playSound(){
        ...
    }
    void rotate(){
        ...
    }
}
```

But even that's not the best design! We've got duplicated code! The rotate method is in all four Shape things. We didn't see the final design. Let us come back to it when we discuss inheritance in upcoming class.

What is Object Oriented Programming?

Object-oriented programming (OOP) is a programming paradigm that uses objects to model real-world things and aims to implement state and behavior using objects.

Object-Oriented Programming is based on implementing the state and behaviour concepts together. State and behaviour are combined into one new concept: an Object. An OO application can therefore produce some output by calling an Object, without needing to pass data structures.

Procedural vs OOPS World The focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines, whereas in object-oriented programming it is to break down a programming task into objects that expose behavior (methods) and data (members or attributes). The most important distinction is that while procedural programming uses procedures to operate on data structures, object-oriented programming bundles the two together, so an "object", which is an instance of a class, operates on its "own" data structure.

Classes and Objects

Object-oriented programming bundles the the data + behaviour related to one entity inside a class. Lets understand classes and objects followed by pillars of OOPS - Abstraction, Encapsulation, Inheritance and Polymorphism in more detail.

Classes

Classes are the starting point of all objects, and we may consider them as the template for creating objects. A class would typically contain member fields, member methods, and a special constructor method.

```
public class Player {
    //Data Members
    String name;
    int guess;


    //Member Methods
    void makeGuess() {
        this.guess = (int)(Math.random() * 9.0) + 1;
        System.out.println(this.name + " guessed " + this.guess);
```

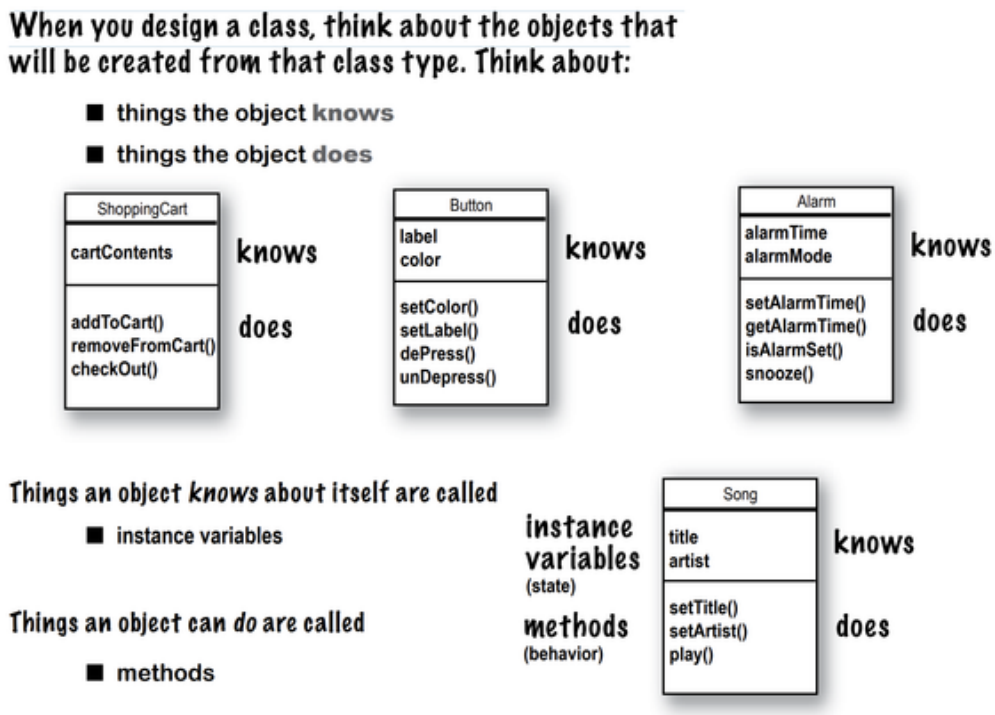
```
    }
    ...
}
```

Objects

Objects are created from classes and are called instances of the class. We create objects from classes using their constructors.

```
        Player p1 = new Player();
        Player p2 = new Player();
        p1.name = "Prateek";
        p2.name = "Naman";
        p1.makeGuess(); //Prateek guessed 9
        p2.makeGuess(); //Naman guessed 6
```

Now, we've created different Player objects, all from a single class. These objects are created from Player class at runtime. This is the point of it all, to define the blueprint in one place, and then, to reuse it many times in many places.



Pillars of Object Oriented Programming

We have 4 pillars foundational concepts in OOPS also called Pillars of Object Oriented Programming, these are Abstraction, Encapsulation, Inheritance & Polymorphism.

Abstraction

Abstraction is hiding complexities of implementation and exposing simpler interfaces. If we think about a typical computer, one can only see the external interface, which

is most essential for interacting with it, while internal chips and circuits are hidden from the user. In OOP, abstraction means hiding the complex implementation details of a program, exposing only the API required to use the implementation. In Java, we achieve abstraction by using interfaces and abstract classes.

Encapsulation

Encapsulation is hiding the state or internal representation of an object from the end-user and providing publicly accessible methods bound to the object for read-write access. This allows for hiding specific information and controlling access to internal implementation.

Encapsulation is used to hide the values or state of a structured data object inside a class, preventing direct access to them by clients in a way that could expose hidden implementation details or violate state invariance maintained by the methods.

```
public class Player {
    //Data Members
    String name;
    private int guess;

    //Provide a method to read guess
    int getGuess(){
        return guess;
    }

    //Member Methods
    void makeGuess() {
        //Randomly generated from 1-9
        this.guess = (int)(Math.random() * 9.0) + 1;
        System.out.println(this.name + " guessed " + this.guess);
    }
    ...
}
```

For example, private member fields like `guess` in the class are hidden from other classes, and they can be accessed using the member methods `getGuess()`. This provides a way to read the value of Guess but prevents write access from other classes.

Inheritance

Inheritance is the mechanism that allows one class to acquire all the properties from another class by inheriting the class. We call the inheriting class a child class and the inherited class as the superclass or parent class.

In Java, we do this by extending the parent class. Thus, the child class gets all the properties from the parent. We will talk about inheritance in detail later.

```
public class User {
    String username;
    String email;
};
```

Student inherits all properties and methods from User.

```
public class Student extends User{
    int marks;
}
```

Polymorphism

Polymorphism is the ability of an OOP language to process data differently depending on their types of inputs. In Java, this can be the same method name having different method signatures and performing different functions.

```
public class TextFile {
    //...

    public String read() {
        return this.getContent()
          .toString();
    }

    public String read(int limit) {
        return this.getContent()
          .toString()
          .substring(0, limit);
    }

    public String read(int start, int stop) {
        return this.getContent()
          .toString()
          .substring(start, stop);
    }
}
```

In this example, we can see that the method `read()` has three different forms with different functionalities. This type of polymorphism is static or compile-time polymorphism and is also called method overloading. There is also runtime or dynamic polymorphism, where the child class overrides the parent's method.We will study runtime polymorphism in later classes.

Advantages & Disadvantages of OOPS

Advantages

1. Reusability: Through classes and objects, and inheritance of common attributes and functions.

2. Security: Hiding and protecting information through encapsulation.

3. Maintenance: Easy to make changes without affecting existing objects much.

4. Inheritance: Easy to import required functionality from libraries and customize them, thanks to inheritance.

Disadvantages

1. Beforehand planning of entities is required that should be modeled as classes.

2. OOPS programs are usually larger than those of other paradigms.
3. Banana-gorilla problem - You wanted a banana but what you got was a gorilla holding the banana and the entire jungle. Read More

----- End ----