Object-Oriented Programming, the possible advantages and disadvantages of using various OOP features in the design and implementation of software.

By- Mohammad Huzaifa Shaikh

Overview-

This aim of this article is to highlight, in-depth, the possible benefits and drawbacks of using OOP and its features in designing and implementing software. This article is also designed to be used as a report by companies to make decisions on whether to adopt and use various object-oriented features in the design and implementation of that company's software.

For the ease of the readers understanding, while elaborating key points in this article, high-level (surface level) information is highlighted first followed by "***moreover***," after which the information goes into more detail and more towards the lower-level (in-depth) aspects of the point of discussion.

---

table of contents

1- Brief description of OOP in creation of software

A programming paradigm known as object-oriented programming (OOP) is based on the ideas of classes and objects. It is used to organise software into straightforward, reusable classes of code blueprints, which are then utilised to build distinct instances of things also known as objects.

2- Advantages of OOP

Some of the biggest advantages of OOP are actually the features/concepts of itself. OOP has four fundamental features: - abstraction, encapsulation, inheritance and polymorphism. These features are what combine to make OOP such an appealing programming paradigm and a no-brainer for when it comes to developers to choose their method/strategy for creating major software. Let's see what makes these certain features of OOP into considerable advantages. [M]

2.1 Abstraction.

Many would agree that it is far more straightforward to reason and design a program when you can separate the interface of a class from its implementation and focus on the interface. This is like treating a system as a "Black Box" where it is not important to understand the gory inner workings in order to reap the benefits of using it. This Process is called abstraction in OOP as we are abstracting away the implementation details of a class and only presenting a clean, easy-to-use interface via the class's functions/methods [1].

Example: A real world example-

Imagine you have to go to your friend's birthday party. You order an UBER and once the app finalizes a driver nearby for you, it gives you the details of that driver and his/her car such as the drivers name, the plate number, the make of the car and the color of the car. These are the details you will need for things such as, but not limited to, identifying the car, making sure you are getting the right car designated to you by Uber or sending a screenshot to one of your family members in case of an emergency. However, the rest of the data about the driver and their car is pretty much irrelevant to you as a passenger. Like the Social Insurance number of the driver, the birth date of the driver, the mileage on the car or the drivers' mothers name, such information is hidden from you as it is pretty much irrelevant to you but can still be critical information be used elsewhere or later in time for other purposes. [M]

Moreover, Abstract classes like the one in the example above can also have concrete methods as well as, obviously, abstract methods. Unlike a normal class which cannot have an abstract method, all methods must be predefined [M].

Abstract methods are mostly declared where two or more subclasses are also doing the same thing in different ways through different implementations. It also extends the same Abstract class and offers different implementations of the abstract methods. Abstract classes help to describe generic types of behaviors and object-oriented programming class hierarchy. It also describes subclasses to offer implementation details of the abstract class. [2]

Example:

Let's say we are working on the backend of a banking software that compares different banks. We would create an abstract Bank class that would store the data/methods that would then be used by all instances of the class (different banks) [M].

```
4   //Written by Mohammad Huzaifa Shaikh
5   abstract class Bank {
6       abstract void getName();
7       abstract void getAddress();
8       abstract void getTransit();
9       abstract void getInterest();
10
11  }
12
13
```

fig 1[M]

```
13
14  class TD extends Bank {
15      private int transit = 67390471;
16      private int rate = 5;
17
18      void getName(){System.out.println("TD Canada Trust");}
19      void getAddress(){System.out.println("3252 Imaginary Street, Toronto, Ontario");}
20      void getTransit(){System.out.println("Transit Number = " + this.transit);}
21      void getInterest(){System.out.println("Interest Rate = " + this.rate + "%");}
22  }
23
24  class BMO extends Bank {
25      private int transit = 7396381;
26      private int rate = 4;
27
28      void getName(){System.out.println("Bank of Montreal");}
29      void getAddress(){System.out.println("1234 Random Street, Toronto, Ontario");}
30      void getTransit(){System.out.println("Transit Number = " + this.transit);}
31      void getInterest(){System.out.println("Interest Rate = " + this.rate + "%");}
32  }
33
34  class MHS extends Bank {
35      private int transit = 6942069;
36      private int rate = 9;
37
38      void getName(){System.out.println("Bank of Huzaifa");}
39      void getAddress(){System.out.println("6969 Huzaifa Street, Toronto, Ontario");}
40      void getTransit(){System.out.println("Transit Number = " + this.transit);}
41      void getInterest(){System.out.println("Interest Rate = " + this.rate + "%");}
42  }
43
```

fig 2[M]

```
45
46
47  class Main {
48      public static void main(String[] args) {
49          System.out.println ("\n\n");
50
51          Bank A = new TD();
52          A.getName();
53          A.getAddress();
54          A.getTransit();
55          A.getInterest();
56
57          System.out.println ("\n\n");
58
59          Bank B = new BMO();
60          B.getName();
61          B.getAddress();
62          B.getTransit();
63          B.getInterest();
64
65          System.out.println ("\n\n");
66
67          Bank C = new MHS();
68          C.getName();
69          C.getAddress();
70          C.getTransit();
71          C.getInterest();
72
73      }
74  }
```

fig 3[M]

Above, we can see that when a regular class extends the abstract bank class, it can now use all the methods that were initially declared in the abstract class and override them to better fit the specific class. Here we create a default abstract Bank class that contains all the methods and then use they keyword "extends" after we declare the class which enables each separate bank class to override the abstract methods and add its own data. [M]

2.2 Encapsulation.

Encapsulation describes the idea of bundling data and methods that work on that data within one unit, like a class in Java. This concept is also often used to hide the internal representation, or state of an object from the outside. This is called information hiding. [3]

The general idea of this mechanism is simple. For example, you have an attribute that is not visible from the outside of an object. You bundle it with methods that provide read or write access. Encapsulation allows you to hide specific information and control access to the internal state of the object [3].

Many programming languages use encapsulation frequently in the form of classes. A class is a program-code-template that allows developers to create an object that has both variables (data) and behaviors

(functions or methods). A class is an example of encapsulation in computer science in that it consists of data and methods that have been bundled into a single unit. [4]

Encapsulation is extremely advantageous for when it comes to things like hiding data, reusability and flexibility. It gives you the ability to decide whether the variables of an object will be read/write or read only/write only. [M]

2.2.1 Hiding data

Better known as data hiding/information hiding is a technique of hiding internal object details. Data hiding ensures that only members of the class have access to the data. The data integrity is preserved. While data hiding limits the use of the data to ensure data security, encapsulation wraps up complex data to give the user a simplified perspective. [M]

The main goal of data hiding is to prevent needless intrusion from outside the class and hide data within it from unauthorised access. To preserve object integrity and stop unintentional or intended modifications to the application in question, data hiding ensures controlled data access. [M]

Data hiding is the process of concealing certain sections of programme code from object members. The program will produce an error if an object member tries to access concealed data. This is a precautionary measure designed to prevent the programmer from connecting to false data that has been concealed. Frequently, the concealed parts are internal ones that the user is unlikely to ever require. [M]

2.2.2 Reusability

The high cost of building software systems can be attributed to the fact that most software development efforts are done from scratch. [M]

In an ideal software development environment, new systems are built by "ordering components from [catalogs of software modules] and combining them, rather than reinventing the wheel every time" [5]

You could be able to employ classes of objects that are created using object-oriented design in a variety of applications since they may have broad universal applicability. This process of constructing several systems fast and effectively is known as code reuse. OOP's use of the inheritance principle enables designers to build base classes with capabilities that may be used by several applications and then derive classes with characteristics tailored to a particular "variant on a theme." [M]

Example:

```java
class Name {

    private int age; // Private is using to hide the data

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

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

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

        Name n1 = new Name();

        n1.setAge(19);

        System.out.println("The age of the person is: "
                           + n1.getAge());
    }
}
```

fig 4[6]

In the program above the class Name encapsulates the variable age as it is set to private. And then declares the getter and setter functions and sets them as public, these can be used to access variables and/or set values.

Example:

```java
// fields to calculate area
class Area {

  int length;
  int breadth;

  // constructor to initialize values
  Area(int length, int breadth) {
    this.length = length;
    this.breadth = breadth;
  }

  // method to calculate area
  public void getArea() {
    int area = length * breadth;
    System.out.println("Area: " + area);
  }
}

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

    Area rectangle = new Area(2, 16);
    rectangle.getArea();
  }
}
```

fig 5 [6]

The program above is a great example that illustrates, on a basic level, the reusability in encapsulation. Class area is defined and it is encapsulated with methods that use the given width and breadth to calculate area. Then all that we need to do when declaring new shapes is state the width and breadth, declare it as an instance of that class and then it calls function/methods and does the rest. Unlike having to do the calculation every time we declare an object.

Moreover,

A module is encapsulated if clients are restricted by the definition of the programming language to access the module only via its defined external interface. Encapsulation thus assures designers that compatible changes can be made safely, which facilitates software evolution and maintenance. These benefits are especially important for large software and long-lived data. To maximize the advantages of encapsulation, one should minimize the exposure of implementation details in external interfaces. A programming language supports encapsulation to the degree that it allows minimal external interfaces to be defined and enforced) This support can be characterized by the kinds of changes that can safely be made to the implementation of a module. For example, one characteristic of an object-oriented language is whether it permits a designer to define a class such that its instance variables can be renamed without affecting clients. [7]

2.3 Inheritance.

Inheritance is a characteristic of majority of object-oriented programming languages that sets them apart from other regular languages. Each class has a superclass from which it derives its internal structure and operations. A class has the option of expanding or changing the operations it inherits. Classes, however, are unable to remove inherited operations. The process through which one class acquires the characteristics and methods of another class is known as inheritance. The Parent class is the one from which the inherited properties and methods are derived. The Child class is the one that receives the parent class's attributes through inheritance. [M]

You can create a subclass that extends a superclass using the potent idea of inheritance. By doing so, the subclass acquires the superclass's types as well as all of its protected and public attributes and methods. After that, you can utilise the superclass's inherited properties, call its inherited methods, or override them, and cast the subclass to any superclass type. [M]

Example:

Lion, Dog, Deer and Elephant. All these fall under the category, Animals. Here in a programming sense, Animals is the parent class and then each of Lion, Dog, Deer and Elephant would be the child classes since they are all animals.

However, they can each have additional and unique characteristics of their own. All of them will have 4 legs, this is a characteristic of the parent class Animals itself. But the Lion will have the ability to Roar which the rest of them wont and the elephant will have a trunk etc. This goes just to show that each child class may have additional or overridden characteristics(methods). [M]

Example:

Another good example of inheritance in terms of software would be something like a premium subscription. Let's say we are developing a movie streaming software. We can offer the customer a basic

subscription to the software which can have something like 30 movies a month, 5 ads per movie and 1080p quality.

However, we can use this basic class as a super/parent class and create a sub/child class that inherits these properties but is the premium subscription and overrides most of these properties such as maximum movies can be 65 instead of 30 and there can be only 1 add per movie instead of 5 and the maximum quality could be 4k instead of 1080p. [M]

Example:

```java
class Calculator {
    int c;
    public void add(int a, int b) {
        c = a + b;
        System.out.println("Sum:" + c);
    }

    public void subtract(int a, int b) {
        c = a - b;
        System.out.println("Subtraction:" + c);
    }
}

public class AdvancedCalculator extends Calculator {
    public void multiplication(int a, int b) {
        c = a * b;
        System.out.println("Multiplication:" + c);
    }

    public void division(int a , int b){
        c = a / b;
        System.out.println("division:" + c);
    }
}

public class CalculatorDemo {
    public static void main(String args[]) {
        int a = 5, b = 4;
        AdvancedCalculator Cal = new AdvancedCalculator();
        Cal.add(a, b);
        Cal.subtract(a, b);
        Cal.multiplication(a, b);
        Cal.division(a, b);
    }
}
```

fig 6[8]

In the above program, when an object of AdvanceCalculator class is created, a copy of all methods and fields of the superclass Calculator acquire memory in this object. So by using an object of a subclass we can also access the members of a superclass. [8]

Moreover, by sharing common code among numerous subclasses, inheritance helps to reduce the amount of duplicate code. We can move shared code up to a mutual superclass when it appears in two classes that are related to one another.

The nicest part is that the derived class will immediately receive any future updates to the superclass's characteristics and functions. To put it another way, a superclass serves as the declaration point for the common properties and methods of all classes in the hierarchy. If changes are necessary, we simply need to make them in the superclass; subclasses will automatically inherit them. The subclass may also add new methods and attributes as needed.

If inheritance were not possible, we would have to modify every source code file that already exists and uses the same logic. Code that is cut and pasted from one class to another could cause mistakes to spread across numerous source code files. Thus, inheritance also aids in preventing such mistakes.

2.4 Polymorphism

Every object-oriented programming language must include the fundamental concept of polymorphism, which is inseparable from OOP. In essence, an object or reference can have distinct forms in various contexts. Polymorphism as a whole would be defined as "a property of having numerous forms" since, as the name suggests, "poly" denotes "many" and "morph" refers to "forms." [M]

One of the important OOP ideas is polymorphism. You can have different or numerous kinds of objects, variables, or methods using polymorphism. Polymorphism allows for different implementations of the same method depending on the requirements of the class. [M]

The idea of polymorphism enhances scalability and readability of the code. You can provide polymorphism so that each instance of the Parent class's method has a different implementation. [M]

Overloading is used to implement polymorphism. Through the use of overloading, functions can accept several types of arguments and/or varied numbers of parameters while yet sharing the same name. numerous variants of a single function. [M]

Example:
Imagine a function that adds two numbers. A sum function might accept two integers as parameters and return an integer, whereas the same sum function might take two floats as arguments and return a float. The compiler determines which function needs to be called when we call sum while passing two numbers, based on the parameters we have supplied. In either scenario, depending on the parameters we supply when calling methods with the same name, we can get a different output. [M]

Example:

```
class Person {
   public void teach(){
      System.out.println("Person can teach");
   }
}

class Teacher extends Person {
   public void teach() {
      System.out.println("Teacher can teach in a school");
   }
}

public class TestTeacher {
   public static void main(String args[]) {
      Person person = new Person(); //Person reference and object
      Person another_person = new Teacher(); //Person reference,
Teacher object
      Teacher teacher = new Teacher(); //Teacher reference and obj.


      person.teach();//output: Person can teach


      // Here you can see Teacher object's method is executed even-
      // -though the Person reference was used
      another_person.teach();//output: Teacher can teach in a school


      teacher.teach();//output: Teacher can teach in a school
   }
}
```

fig7 [9]

The above code snippet shows polymorphism in effect. A class Person is created with a void method prints an output. Followed by a class Teacher which extends the Person class and overrides the same method. When the objects in the main method are created, we can see that another_person is of type Teacher, but references Person. Therefore, when person.teach() is issued it outputs the method form the original Person class and when another_person.teach() is issued, the overridden method from the subclass is output.[M]

Moreover, there are two types of polymorphism. Compile-time polymorphism and run-time polymorphism.

Compile-time, also known as static polymorphism, is achieved by method/function overloading. As we've seen in examples above, when we use the same name of the function, but in different subclasses we use different number of parameters for these same functions, we can say we overload the function. The same can be said for examples above where we change the type of arguments, again we call this method overloading. [M]

Run-time, also known as Dynamic Polymorphism, is achieved by method/function overriding. When a subclass has its own definition of a function which was originally inherited from its parent class, the parent class's function is now overridden. When an object of the subclass is created, the function that is inside the subclass will be called as it now has a higher precedence. [M]

2.4 Troubleshooting

One of the best things that go Hand in hand with Object-Oriented Programming is the easy of troubleshooting. Are you having problems with the Dog object you just created? The error will most probably be in Dog class! You save yourself the trouble of having to go through the entire code before finding where the problem is occurring. [M]

If there is a flaw in the code, let's say the user has no notion where it is. In addition, the user is unable to locate the issue in the code. For common programming languages, this is fairly challenging. But if Object-Oriented Programming is used, the user will always know exactly where to look in the code if an error occurs. Other code parts don't need to be checked because the error will point out the problem. [M]

That's what makes OOP so wonderful. Each component of functionality in an object operates independently, leaving the others alone. Additionally, its flexibility reduces the possibility that one person would replicate the functionality of another and enables a software team to work on several items at once. [M]

Example:
Let’s say we create a class, FlightTicket. This class contains the following variables:-

Name(String)
AirFare (int)
Duration (Float)
BaggageAllowance (int)

Now let’s say we create multiple objects from this class. We would assign all kinds of different values to each object and can create as many as we like with a readymade template (the original class). However, we assign a float value to the AirFare value of every object but later realize that in the original class, AirFare is set to int type. All we would have to do now is change the type of AirFare from int to float and that would resolve the problem in every single instance of the class (every object). As opposed to a no OOP approach. We would have created every flight ticket individually, declaring new variables every single time. Now imagine we again realize that AirFare had to be of type Float and not int, this time we would have to individually change the variable type of AirFare in every flight ticket that we have created. That just sounds quite tedious! But not with an Object-Oriented approach. [M]

3- Disadvantages of OOP

References.

[M] This is my own personal work. 100% my own ideas and thinking about the topic. Examples labeled with 'M' are also created entirely by me.

[1] Kyle Herrity: What Is Object-Oriented Programming? 4 Basic Concepts of OOP. available at https://www.indeed.com/career-advice/career-development/what-is-object-oriented-programming.

[2] James Hartman: What is Abstraction in OOPs? Java Abstract Class & Method. available at https://www.guru99.com/java-data-abstraction.html

[3] Thorben Janssen. OOP Concept for Beginners: What is Encapsulation. available at:- https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/

[4] Encapsulation - Definition & Overview. available at https://www.sumologic.com/glossary/encapsulation/#:~:text=What%20does%20encapsulation%20mean%3A%20In,in%20the%20form%20of%20classes.

[5] Josephine Micallef, Reusability and Extensibility. In *Encapsulation, reusability and extensibility in OOP languages*. Page 42, August 1987

[6] Encapsulation in Java. Available at https://www.geeksforgeeks.org/encapsulation-in-java/

[7] Alan Snyder, Encapsulation in *Encapsulation and Inheritance in Object-Oriented Programming. Page 39.* **I have slightly modified the wordings in this paragraph to better suit the use of software companies using this report**.

[8] Inheritance in Object Oriented Programming (Java). Available at https://www.enjoyalgorithms.com/blog/inheritance-in-java

[9] Shanika Ediriweera in *Polymorphism explained simply!.* Available at https://medium.com/@shanikae/polymorphism-explained-simply-7294c8deeef7