4 Pillars of OOP

There are 4 pillars of oop:

1. Encapsulation
2. Inheritance
3. Abstraction
4. Polymorphism

Let's discuss each of them with a short explanation and a real-life code example

1. Encapsulation

We all have studied encapsulation as the hiding data members and enabling the users to access data using public methods that we call getters and setters. But why? Let's forget that and reiterate with a simpler definition.

Encapsulation is a technique of restricting a user from directly modifying the data members or variables of a class in order to maintain the integrity of the data. How do we do that? We restrict the access of the variables by **switching the access-modifier to private** and exposing public methods that we can use to access the data. Let's have a look at the precise examples below. It will help us understand how we can leverage encapsulation to maintain the integrity of data.

No encapsulation:

```
/**
 * @author thegeekyasian.com
 */
public class Account {

  public double balance;

  public static void main(String[] args) {

        Account theGeekyAsianAccount = new Account();

        theGeekyAsianAccount.balance = -54;
```

```
  }
}
```

In the above code snippet, the `main()` method accesses the `balance` variable directly. It allows a user to set any double value to the balance variable of the `Account` class. We can lose the integrity of the data by allowing anyone to set balance value as anything invalid number e.g. -54 in this case.

With Encapsulation:

```
/**
 * @author thegeekyasian.com
 */
public class Account {

  private double balance;

  public void setBalance(double balance) {

    if(balance < 0) { // Validating input data in order to maintain data integrity
          throw new IllegalArgumentException("Balance cannot be less than zero
(0)");
    }
    this.balance = balance;
  }

  public static void main(String[] args) {

        Account theGeekyAsianAccount = new Account();

        theGeekyAsianAccount.setBalance(1); // Valid input - Allowed
        theGeekyAsianAccount.setBalance(-55); // Stops user and throws exception
  }
}
```

In this code, we have restricted access to the balance variable and added a `setBalance()` method to allow users to set the value of the Account balance. The setter validates the value provided before assigning it to the balance variable. If the value is any number lesser than zero, it throws an exception. This makes sure that the integrity data is not compromised.

With the above examples explained, I hope the idea of encapsulation as one of the 4 pillars of oop proves its value to you.

2. Inheritance

Inheritance is a technique of acquiring the properties of another class having features in common. It allows us to increase the reusability and reduce the duplication of code. It is also known as a child-parent relationship, where a child inherits the properties of its parent. This is the reason it is called 'is-a relationship' where the child is-a type of parent.

Let's dig deep into two short examples and see how inheritance makes code simpler and reusable.

No Inheritance:

```
/**
 * @author thegeekyasian
 */
public class Rectangle {

  private int width;
  private int height;

  public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
  }

  public int getArea() {
        return width * height;
  }
}

public class Square {

  private int width; // Duplicate property, also used in class Rectangle

  public Square(int width) {
        this.width = width;
  }

  public int getArea() { // Duplicate method, similar to the class Rectangle
        return this.width * this.width;
  }
}
```

The two classes are similar have the `width` property and the `getArea()` method in common. We can increase the reusability of the code by doing a small refactor where class `Square` ends up inheriting the class `Rectangle`.

With Inheritance:

```
/**
 * @author thegeekyasian
 */
public class Rectangle {

  private int width;
  private int height;

  public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
  }

  public int getArea() {
        return width * height;
  }
}

public class Square extends Rectangle {

  public Square(int width) {
        super(width, width); // A rectangle with the same height as width is a
square
  }
}
```

Just by extending the class Rectangle, the class Square now is-a type of Rectangle. **This means it has inherited all the properties that are common between the Square and Rectangle**.

In the examples above, we can see how inheritance plays a vital role in making the code reusable. It also enables a class to inherit the behavior of the parent class.

3. Abstraction

Abstraction is a technique of providing only the essential details to the user by hiding the unnecessary or irrelevant details of an entity. This helps in reducing the operational complexity at the user-end.

Abstraction enables us to provide a simple interface to a user without asking for complex details to perform an action. In simpler words, **giving the user the ability to drive the car without requiring to understand tiny details of 'how does the engine work'**.

Let's have a look at an example first and then discuss how abstraction helps us.

```
/**
* @author thegeekyasian.com
*/
public class Car {

  public void lock() {}
  public void unlock() {}

  public void startCar() {

        checkFuel();
        checkBattery();
        whatHappensWhenTheCarStarts();
  }

  private void checkFuel() {
        // Check fuel level
  }

  private void checkBattery() {
        // Check car battery
  }

  private void whatHappensWhenTheCarStarts() {
        // Magic happens here
  }
}
```

In the code above, the `lock()`, `unlock()` and `startCar()` methods are public, while the rest are private to the class. We have simplified the access of using the car to the user by handling the complex details internally. If a user was asked to `checkFuel()` and `checkBattery()` manually before startCar() that would increase the complexity at the user's end. **With the above code, all the user has to do is use `startCar()` and the rest will be taken care of by the class. This is what we call 'abstraction'.**

4. Polymorphism

The last and the most important of all 4 pillars of OOP is Polymorphism. Polymorphism means "many forms". By its name, it is a feature that allows you to perform an action in multiple or different ways. When we talk about polymorphism, there isn't a lot to discuss unless we talk about its types.

There are two types of polymorphism:

1. Method Overloading – Static Polymorphism (Static Binding)
2. Method Overriding – Dynamic Polymorphism (Dynamic Binding)

Let's discuss each of these types and see what's the difference between the two.

Method Overloading – Static Polymorphism:

The method overloading or static polymorphism, also known as Static Binding, also known as compile-time binding is a type where method calls are defined at the time of compilation. Method overloading allows us to have multiple methods with the same name having different datatypes of parameter, or a different number of parameters, or both.

But the question is, how is method overloading (or static polymorphism) useful? Let's see the below examples to understand method overloading even better.

No Method Overloading:

```
/**
* @author thegeekyasian.com
*/
public class Number {

  public void sumInt(int a, int b) {
        System.out.println("Sum: " + (a + b));
  }

  public void sumDouble(double a, double b) {
        System.out.println("Sum: " + (a + b));
  }

  public static void main(String[] args) {

        Number number = new Number();

        number.sumInt(1, 2);
        number.sumDouble(1.8, 2.5);
  }
}
```

In the above example, we have created two methods with different names just to add two different kinds of numbers. If we continue with a similar implementation, we will end up having multiple methods with different names. This will decrease the code quality and accessibility. To improve this, we can use method overloading

to use the same name for different methods. Which will allow the user to have a single option as an entry-point to perform sum on different kinds of numbers.

Method overloading works when two or more have the same name but different parameters. The return type can be the same as well as different. If two methods have the same name, same parameters **but different return types**, NO! That is not valid example of overloading, and will throw a compilation error.

With Method Overloading:

```
/**
* @author thegeekyasian.com
*/
public class Number {

  public void sum(int a, int b) {
        System.out.println("Sum: " + (a + b));
  }

  public void sum(double a, double b) {
        System.out.println("Sum: " + (a + b));
  }

  public static void main(String[] args) {

        Number number = new Number();

        number.sum(1, 2);
        number.sum(1.8, 2.5);
  }
}
```

In the same code, with a smaller tweak, we were able to overload both methods by making the name similar for both. The user can now provide their specific data types as parameters in the method. It would then perform the action based on their provided data type. **This binding of methods is done at the time of compilation as the compiler knows which method will be called with the provided type of parameter. This is also why we call it compile-time binding.**

Method Overriding- Dynamic Polymorphism:

In contrast to method overloading, method overriding allows you have to exactly the same signature as multiple methods, but they should be in multiple different classes. The question is how is this special? These classes have an IS-A relationship i.e. should have inheritance between them. In other words, in method overriding or

dynamic polymorphism, methods are resolved dynamically at the runtime when the method is called. This is done based on the reference of the object it is initialized with.

Here is a small example of method overriding:

```
/**
* @author thegeekyasian.com
*/
public class Animal {

  public void walk() {
        System.out.println("Animal walks");
  }
}

public class Cat extends Animal {

  @Override
  public void walk() {
        System.out.println("Cat walks");
  }
}

public class Dog extends Animal {

  @Override
  public void walk() {
        System.out.println("Dog walks");
  }
}

public class Main {

  public static void main(String[] args) {

        Animal animal = new Animal();
        animal.walk(); // Animal walks

        Cat cat = new Cat();
        cat.walk(); // Cat walks

        Dog dog = new Dog();
        dog.walk(); // Dog walks

        Animal animalCat = new Cat(); // Dynamic Polymorphism
        animalCat.walk(); // Cat walks

        Animal animalDog = new Dog(); // Dynamic Polymorphism
        animalDog.walk(); //Dog walks
  }
}
```

In this example of overriding, we have dynamically assigned the ‘Dog’ and ‘Cat’ type objects to the Animal type. This enables us to call the `walk()` method of the referenced instances dynamically at the runtime. We can do this with the help of method overriding (or dynamic polymorphism).

With this, we have concluded a short discussion of the 4 pillars of OOP and I hope it turns out to be helpful. For any questions or suggestions or improvements to this article, feel free to drop your comments below.