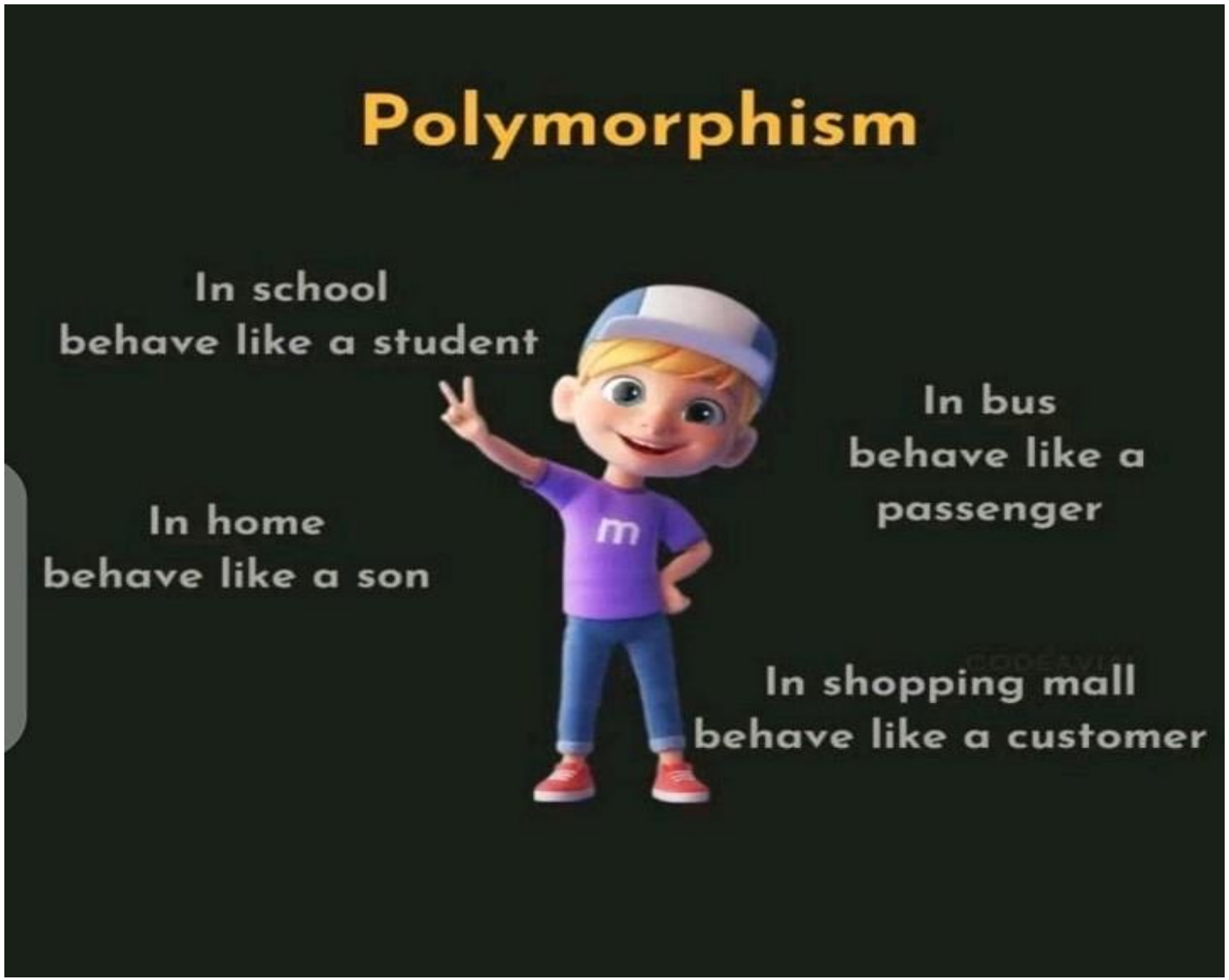
Polymorphism


Polymorphism
In school
behave like a student
In home
behave like a son
In bus
behave like a
passenger
In shopping mall
behave like a customer

Polymorphism

- Referring to the idea that an entity in code such as a variable, function or object can have more than one form.
- Polymorphism means "many forms“.

Method Overloading vs Method Overriding

- **Method Overloading**
  - Method overloading involves defining multiple methods in the same class with the same name but different parameters.
  - Overloaded methods must have a different number or type of parameters.
  - Overloading occurs within the same class.
  - Method overloading occurs in the same class and involves methods with the same name but different parameters.
- **Method Overriding**
  - Method overriding involves providing a specific implementation for a method that is already provided by its superclass (inherited method).
  - The overriding method must have the same method signature, including the same parameters and return type.
  - Overriding occurs between a superclass and its subclass.
  - Method overriding occurs between a superclass and its subclass, and it involves providing a specific implementation for an inherited method with the same signature.

Types of polymorphism

- **Compile time polymorphism.** Also known as static polymorphism, compile time polymorphism is common in OOP languages like Java. It uses method overloading to create multiple methods that have the same name in the same class, but a different number of parameters. They may also have parameters for different data types.
- **Runtime polymorphism.** Also known as dynamic polymorphism, runtime polymorphism uses method overriding to let a child class have its own definition of a method belonging to the parent class. This type of polymorphism is often associated with upcasting, which happens when a parent class points to an instance of the child's class.

```java
// // Runtime Polymorphism (Method Overriding):

class Animal {
    void makeSound() {
        System.out.println("Generic animal sound");
    }

    int add(int a, int b) {
        return a + b;
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Polymorphism {
    Run | Debug
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        Animal Rk = new Animal(); // Runtime polymorphism
        myAnimal.makeSound();
        System.out.println(myAnimal.add(25, 15));
        Rk.makeSound(); // Calls the overridden method in Dog
    }
}
```

```java
class MathOperations {
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }

    String add(String a, String b) {
        return a + b;
    }
}

public class Polymorphism {
    Run | Debug
    public static void main(String[] args) {
        MathOperations math = new MathOperations();
        int resultInt = math.add(5, 10); // Compile-time polymorphism
        double resultDouble = math.add(5.5, 10.5);
        String str = math.add("Hassan ", "Don");
        System.out.println(resultInt); // Compile-time polymorphism
        System.out.println(resultDouble);
        System.out.println(str);
    }
}
```

Task

- Compile-time Polymorphism (Method Overloading):
  - Create a class named Printer with multiple overloaded print methods that accept different types of data (int, double, String). Demonstrate the usage of these methods by printing values of different data types.
  - Design a class named MathOperations with overloaded methods for addition and multiplication. Implement separate methods for handling integers, floats, and strings. Test the class by performing operations with different data types.
- Runtime Polymorphism (Method Overriding):
  - Define a base class Animal with a virtual function makeSound. Implement two derived classes, Dog and Cat, each overriding the makeSound method to produce a different sound. Create objects of both classes and call the makeSound method.
  - Create an abstract class Shape with an abstract method calculateArea. Implement two concrete classes, Circle and Rectangle, each extending the Shape class and providing its own implementation for calculateArea. Test the classes by calculating the area for both shapes.