Oops:1.class>constructor>method>psvm>obj creation

Classs:blueprint

Obj:entity

To get close into real world java integrate with oops

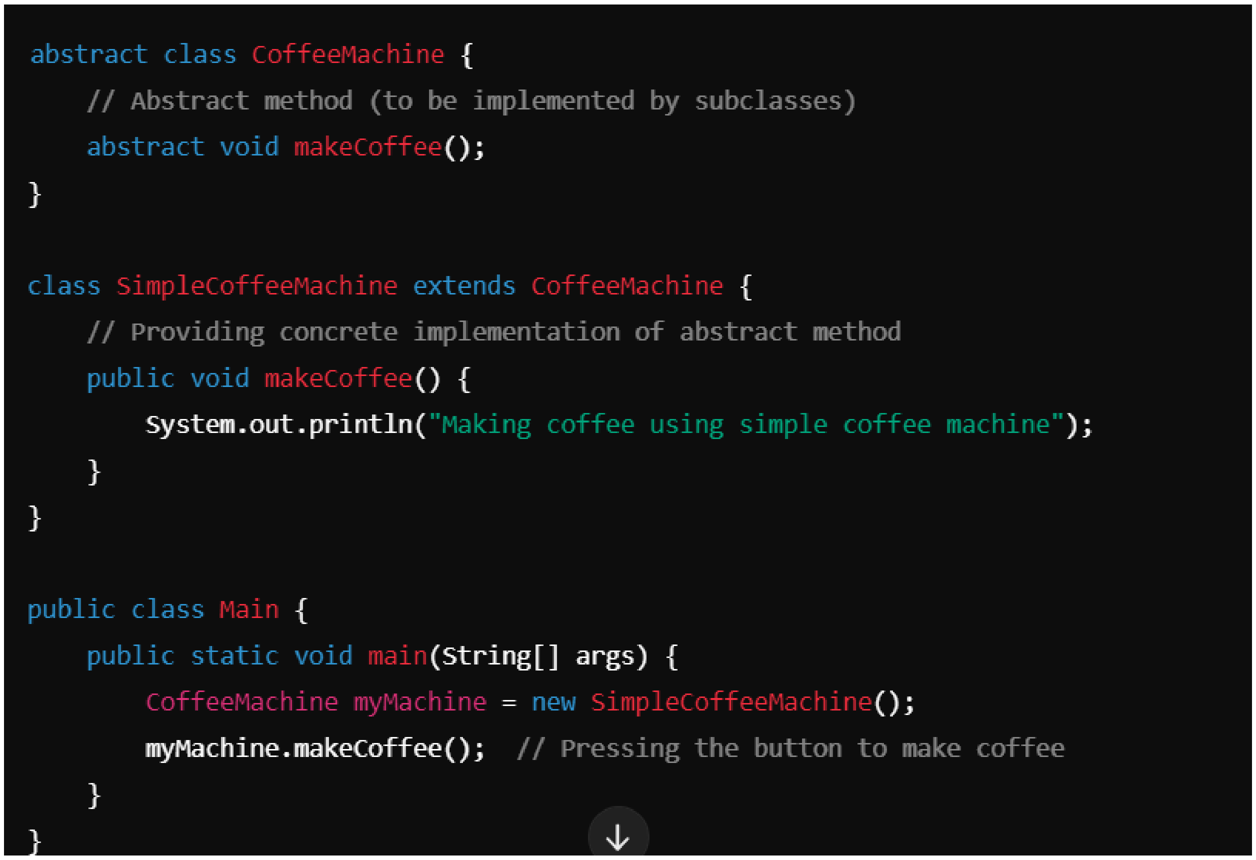
===

1.ABSTRACTION:

Abstraction will hide the unwated data from users

Can be done with abstract classes and interface

##COFFEE MACHINE EXAMPLE



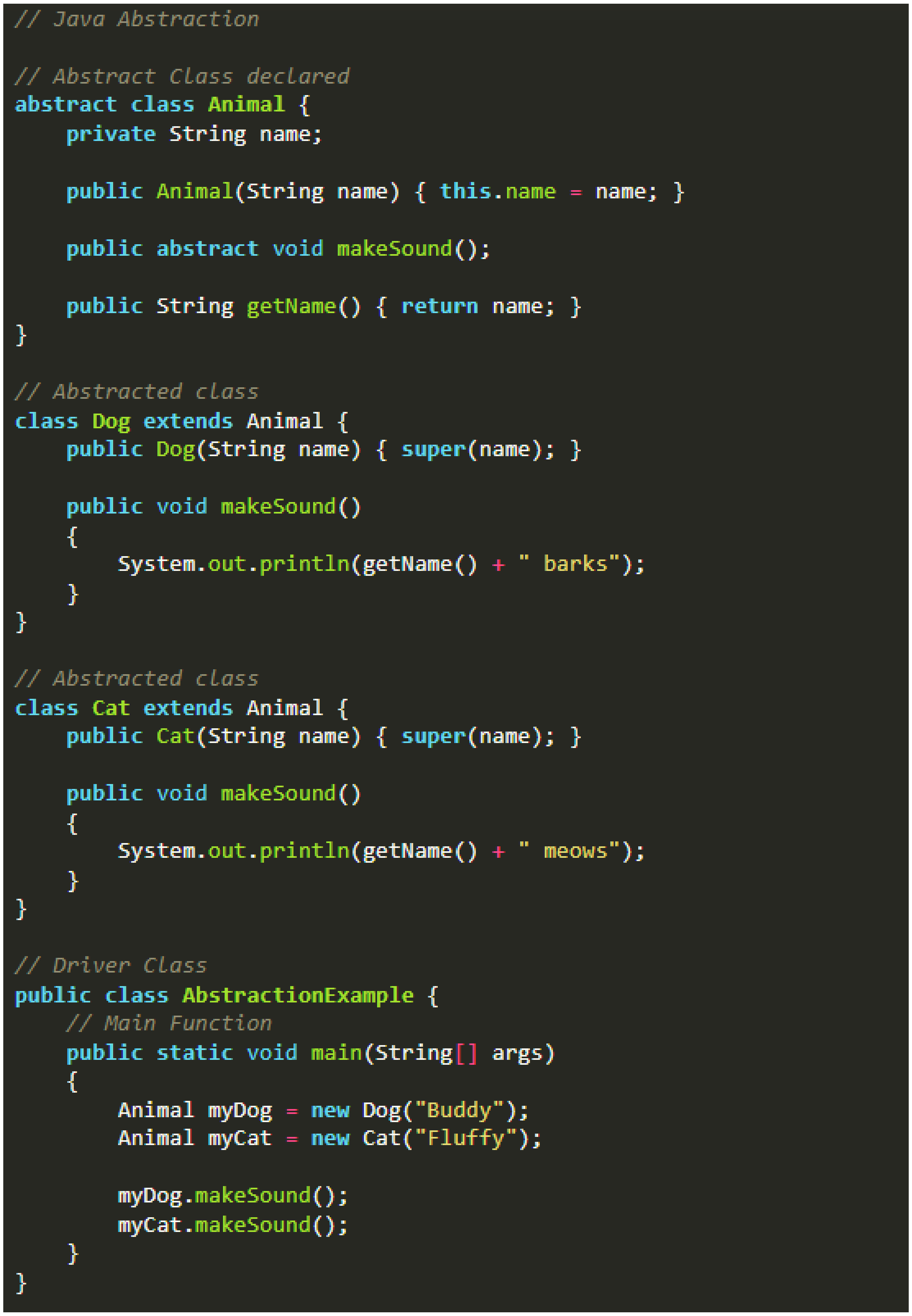
```java
abstract class CoffeeMachine {
    // Abstract method (to be implemented by subclasses)
    abstract void makeCoffee();
}

class SimpleCoffeeMachine extends CoffeeMachine {
    // Providing concrete implementation of abstract method
    public void makeCoffee() {
        System.out.println("Making coffee using simple coffee machine");
    }
}

public class Main {
    public static void main(String[] args) {
        CoffeeMachine myMachine = new SimpleCoffeeMachine();
        myMachine.makeCoffee();  // Pressing the button to make coffee
    }
}
```

1.create anstract class

2.create a class user interactie that extends abstract class

3.main class

4.obj creation to call user class



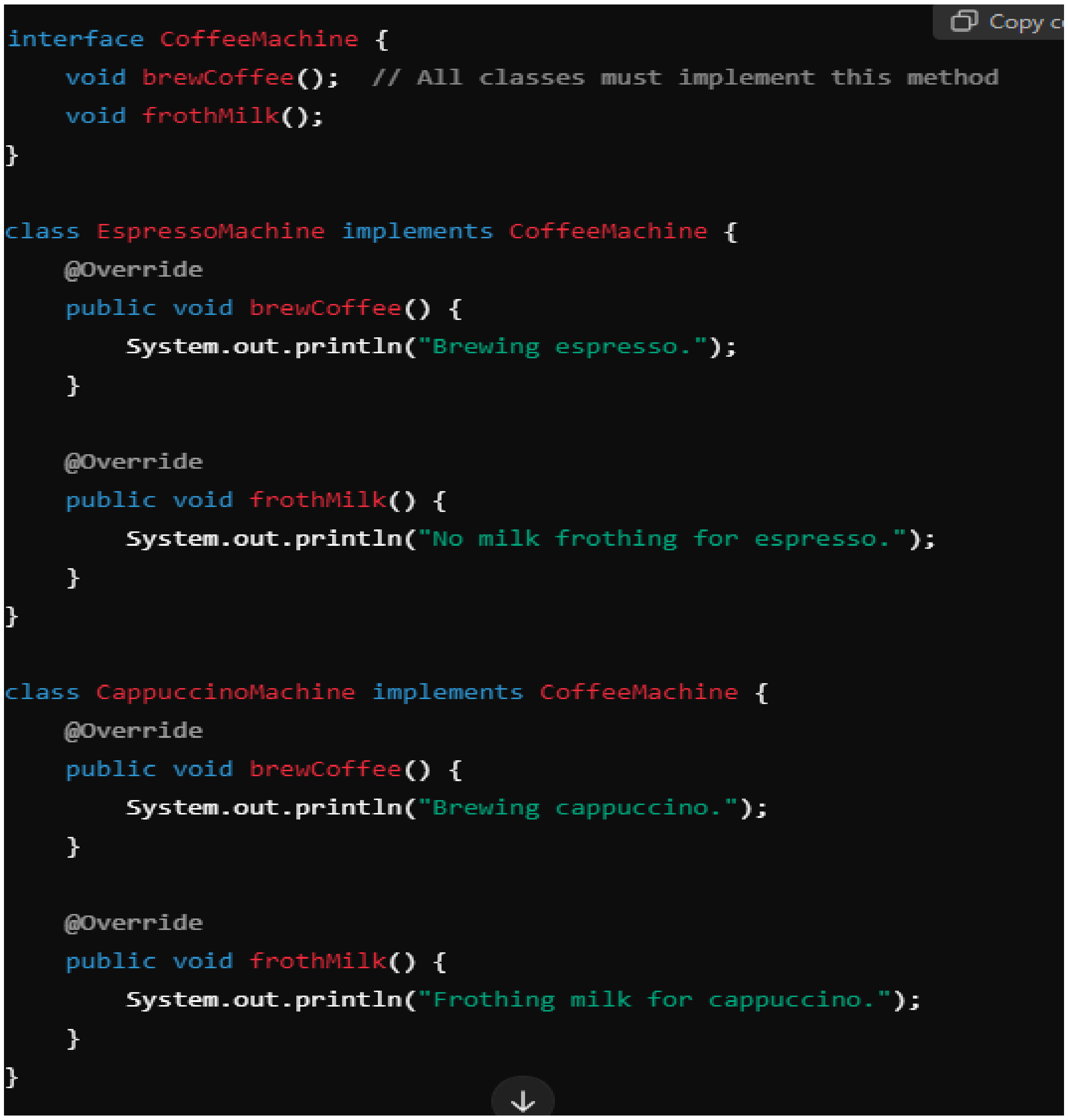
```java
// Java Abstraction

// Abstract Class declared
abstract class Animal {
    private String name;

    public Animal(String name) { this.name = name; }

    public abstract void makeSound();

    public String getName() { return name; }
}

// Abstracted class
class Dog extends Animal {
    public Dog(String name) { super(name); }

    public void makeSound()
    {
        System.out.println(getName() + " barks");
    }
}

// Abstracted class
class Cat extends Animal {
    public Cat(String name) { super(name); }

    public void makeSound()
    {
        System.out.println(getName() + " meows");
    }
}

// Driver Class
public class AbstractionExample {
    // Main Function
    public static void main(String[] args)
    {
        Animal myDog = new Dog("Buddy");
        Animal myCat = new Cat("Fluffy");

        myDog.makeSound();
        myCat.makeSound();
    }
}
```

Using interface:

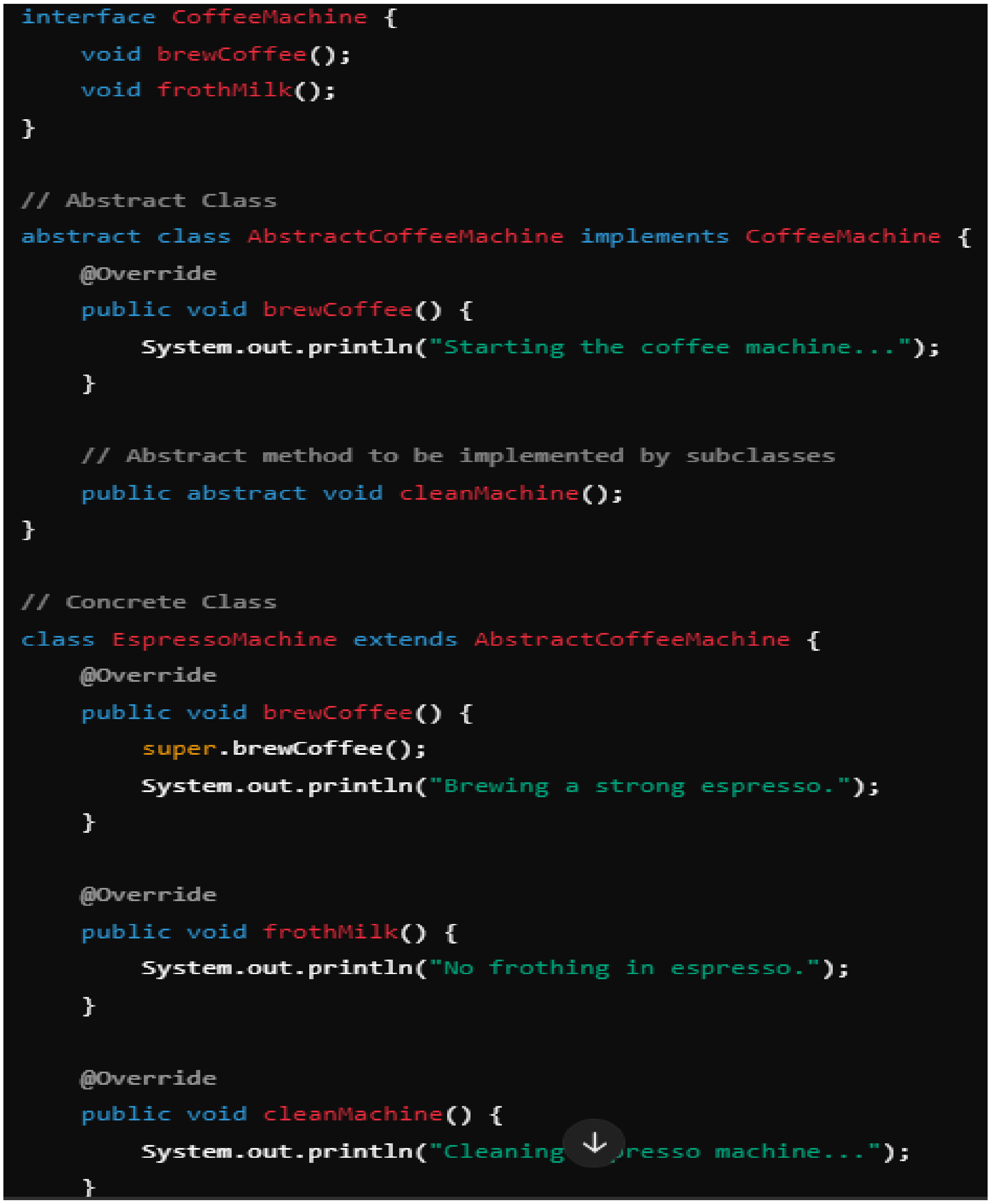
When a common spcial is shared among clases this anbstarction is used

Using interface we can do multiple inheritence

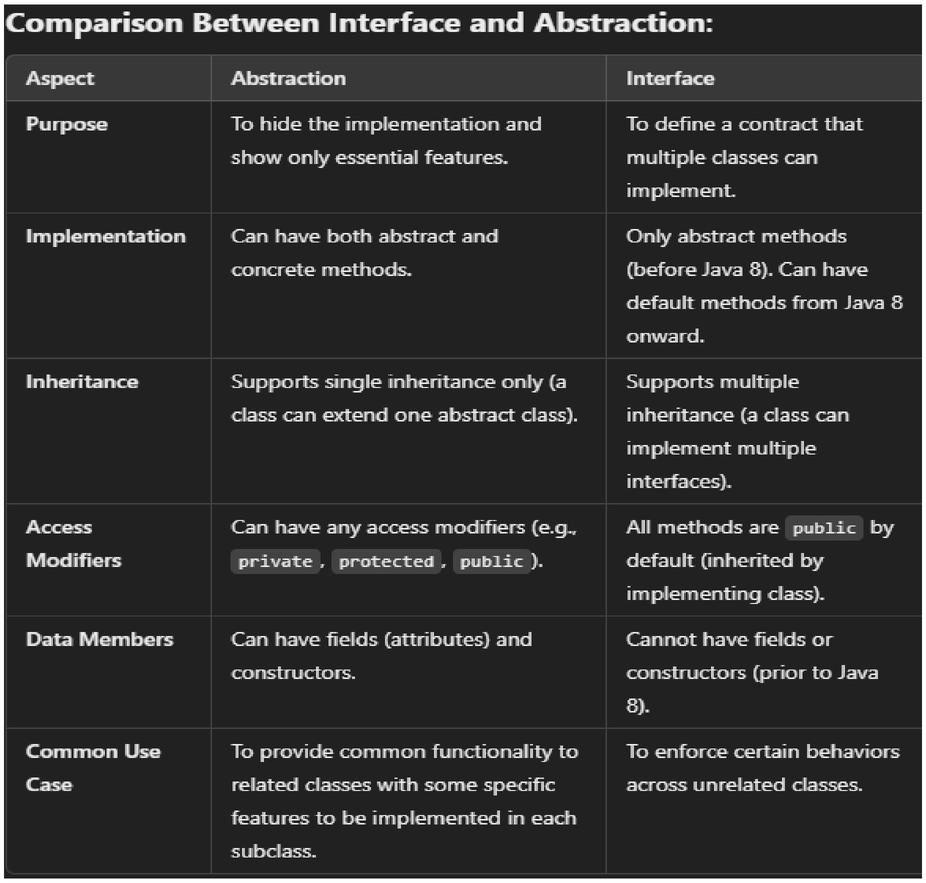
Tells what to do not how to do



```java
interface CoffeeMachine {
    void brewCoffee();  // All classes must implement this method
    void frothMilk();
}

class EspressoMachine implements CoffeeMachine {
    @Override
    public void brewCoffee() {
        System.out.println("Brewing espresso.");
    }

    @Override
    public void frothMilk() {
        System.out.println("No milk frothing for espresso.");
    }
}

class CappuccinoMachine implements CoffeeMachine {
    @Override
    public void brewCoffee() {
        System.out.println("Brewing cappuccino.");
    }

    @Override
    public void frothMilk() {
        System.out.println("Frothing milk for cappuccino.");
    }
}
```



```java
interface CoffeeMachine {
    void brewCoffee();
    void frothMilk();
}

// Abstract Class
abstract class AbstractCoffeeMachine implements CoffeeMachine {
    @Override
    public void brewCoffee() {
        System.out.println("Starting the coffee machine...");
    }

    // Abstract method to be implemented by subclasses
    public abstract void cleanMachine();
}

// Concrete Class
class EspressoMachine extends AbstractCoffeeMachine {
    @Override
    public void brewCoffee() {
        super.brewCoffee();
        System.out.println("Brewing a strong espresso.");
    }

    @Override
    public void frothMilk() {
        System.out.println("No frothing in espresso.");
    }

    @Override
    public void cleanMachine() {
        System.out.println("Cleaning [illegible]presso machine...");
    }
```



Comparison Between Interface and Abstraction:

Aspect	Abstraction	Interface
Purpose	To hide the implementation and show only essential features.	To define a contract that multiple classes can implement.
Implementation	Can have both abstract and concrete methods.	Only abstract methods (before Java 8). Can have default methods from Java 8 onward.
Inheritance	Supports single inheritance only (a class can extend one abstract class).	Supports multiple inheritance (a class can implement multiple interfaces).
Access Modifiers	Can have any access modifiers (e.g., `private`, `protected`, `public`).	All methods are `public` by default (inherited by implementing class).
Data Members	Can have fields (attributes) and constructors.	Cannot have fields or constructors (prior to Java 8).
Common Use Case	To provide common functionality to related classes with some specific features to be implemented in each subclass.	To enforce certain behaviors across unrelated classes.

##encapsulation:
giving security

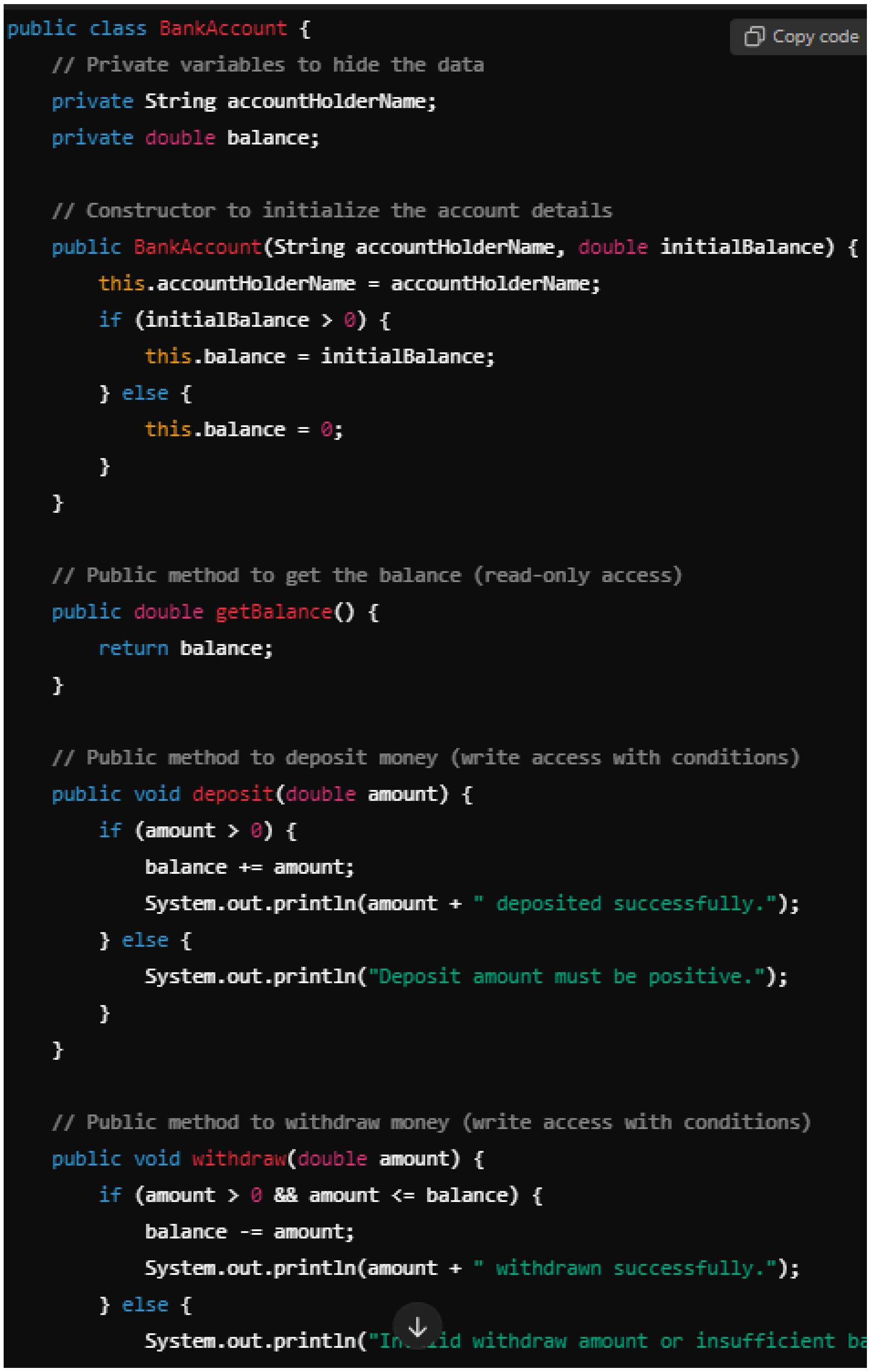
Protect data

Private keywords are used

Getter setter methods

Ex: bank recheck ad changing amount

When a account holder came to withdraw money in encapsulation first allow him to check his name if check done then next stage of processs this validation is dne at encapsulation secured data

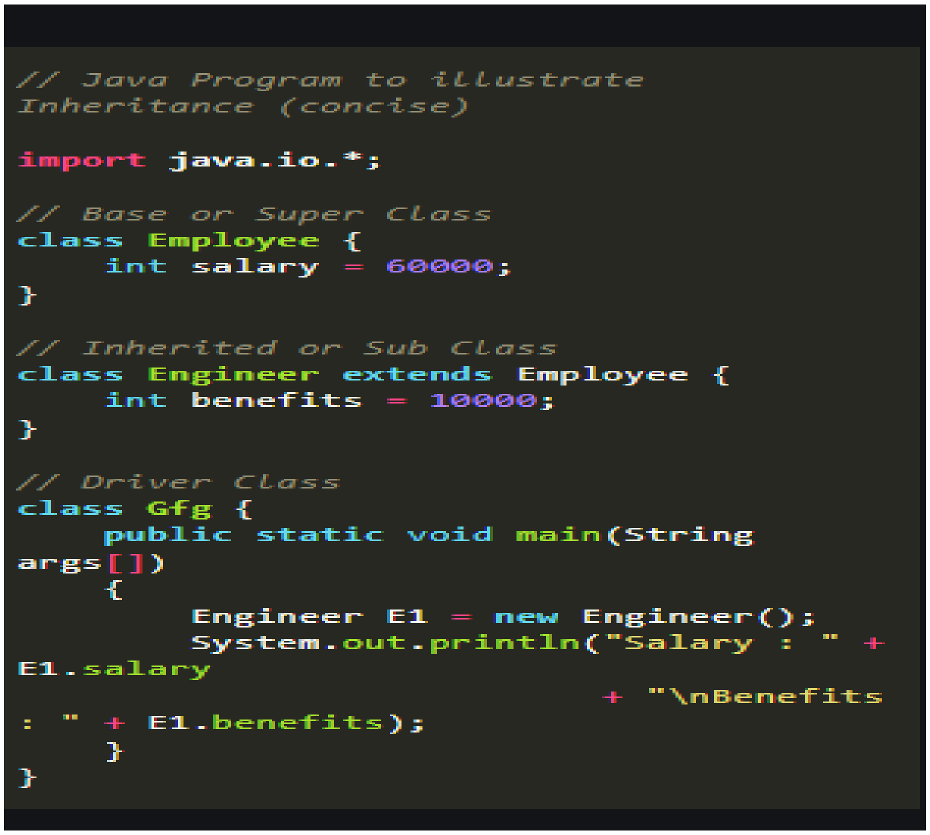


```java
public class BankAccount {
    // Private variables to hide the data
    private String accountHolderName;
    private double balance;

    // Constructor to initialize the account details
    public BankAccount(String accountHolderName, double initialBalance) {
        this.accountHolderName = accountHolderName;
        if (initialBalance > 0) {
            this.balance = initialBalance;
        } else {
            this.balance = 0;
        }
    }

    // Public method to get the balance (read-only access)
    public double getBalance() {
        return balance;
    }

    // Public method to deposit money (write access with conditions)
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println(amount + " deposited successfully.");
        } else {
            System.out.println("Deposit amount must be positive.");
        }
    }

    // Public method to withdraw money (write access with conditions)
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println(amount + " withdrawn successfully.");
        } else {
            System.out.println("Invalid withdraw amount or insufficient ba
```

##inheritence:

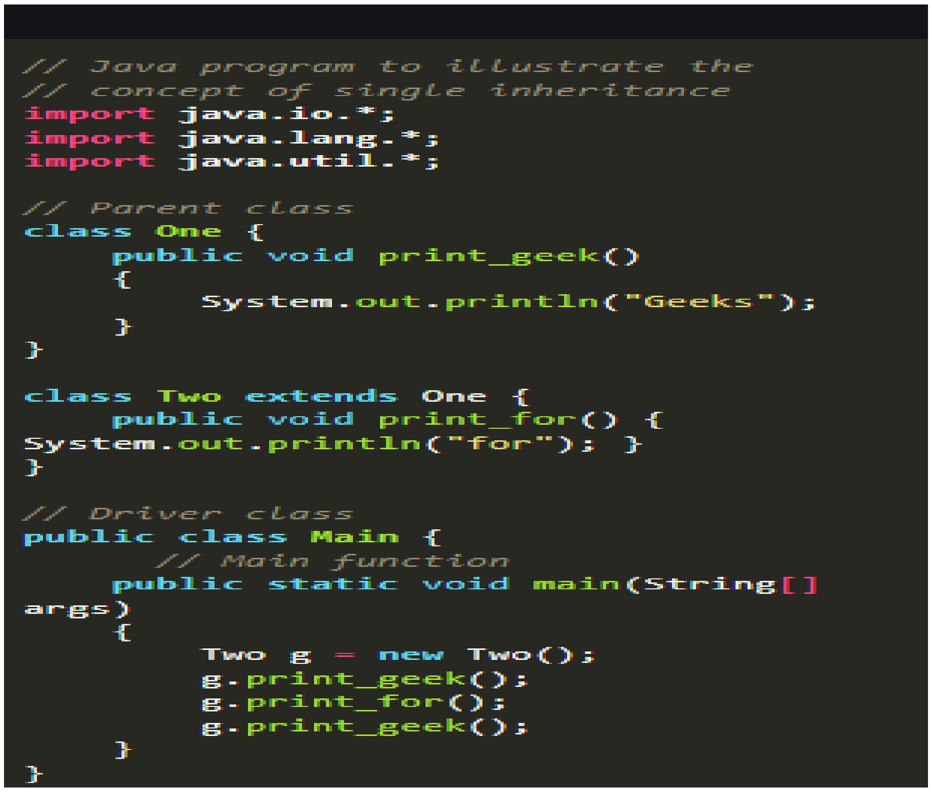
Accepting data from parent class (base) to derived class(child)

Using extend keyword



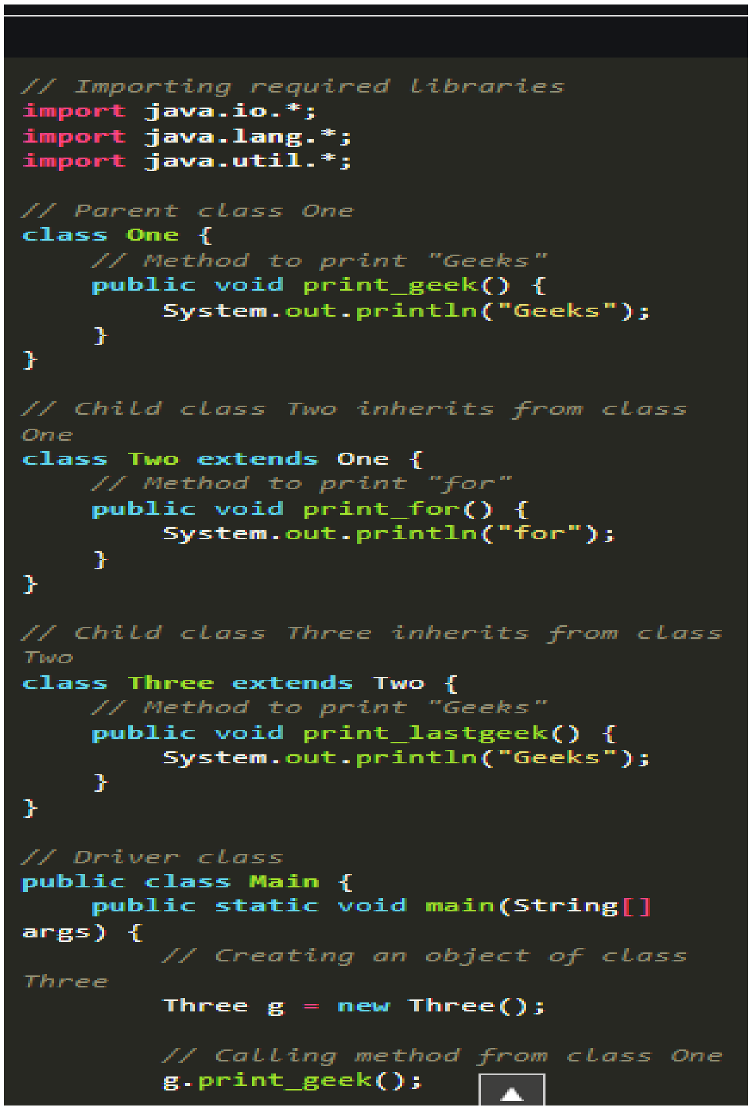
```
// Java Program to illustrate
Inheritance (concise)

import java.io.*;

// Base or Super Class
class Employee {
	int salary = 60000;
}

// Inherited or Sub Class
class Engineer extends Employee {
	int benefits = 10000;
}

// Driver Class
class Gfg {
	public static void main(String
args[])
	{
		Engineer E1 = new Engineer();
		System.out.println("Salary : " +
E1.salary
						+ "\nBenefits
: " + E1.benefits);
	}
}
```

1. Single Inheritance



```
// Java program to illustrate the
// concept of single inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

// Parent class
class One {
	public void print_geek()
	{
		System.out.println("Geeks");
	}
}

class Two extends One {
	public void print_for() {
System.out.println("for"); }
}

// Driver class
public class Main {
  	// Main function
	public static void main(String[]
args)
	{
		Two g = new Two();
		g.print_geek();
		g.print_for();
		g.print_geek();
	}
}
```

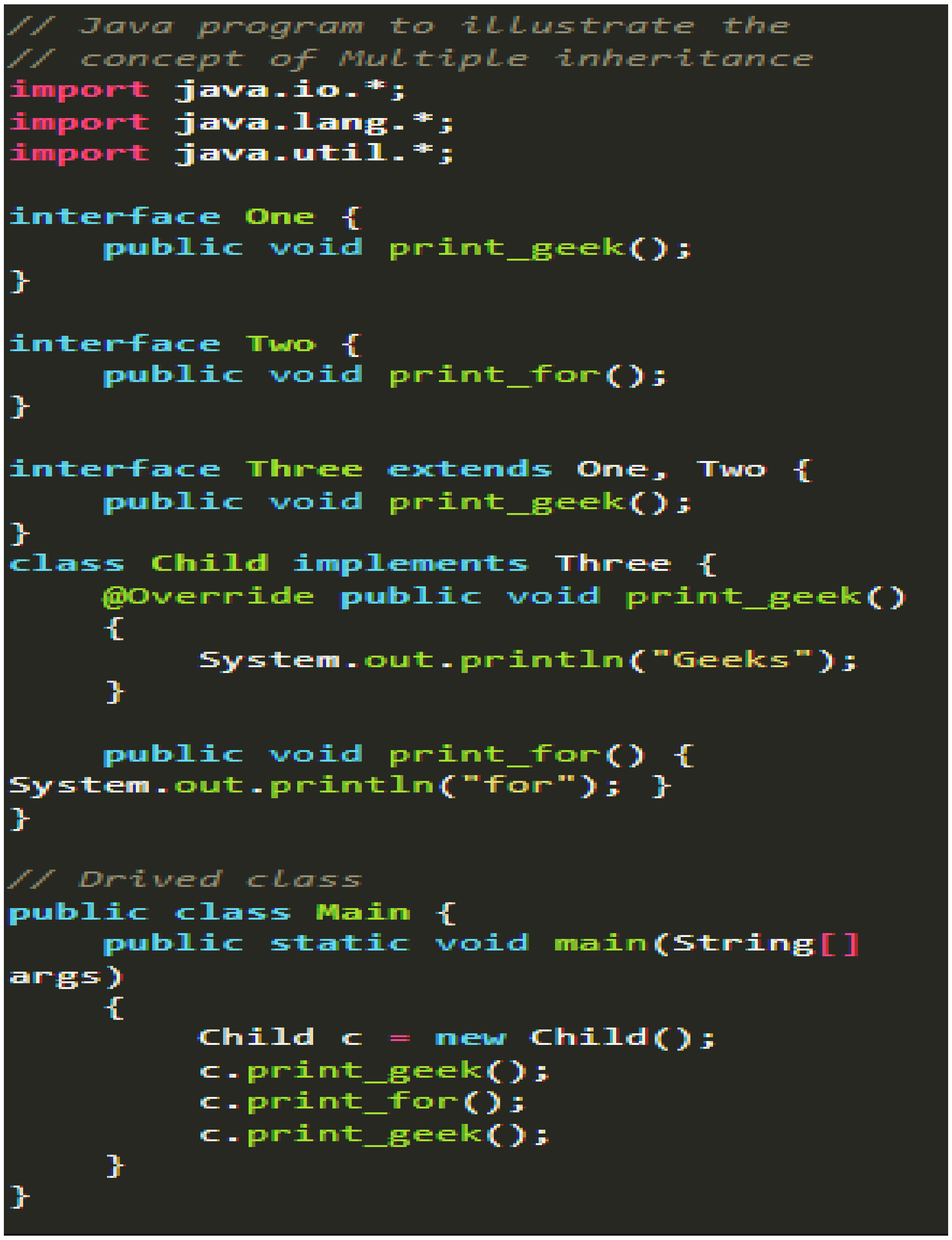
2. Multilevel Inheritance



```
// Importing required libraries
import java.io.*;
import java.lang.*;
import java.util.*;

// Parent class One
class One {
     // Method to print "Geeks"
     public void print_geek() {
          System.out.println("Geeks");
     }
}

// Child class Two inherits from class
One
class Two extends One {
     // Method to print "for"
     public void print_for() {
          System.out.println("for");
     }
}

// Child class Three inherits from class
Two
class Three extends Two {
     // Method to print "Geeks"
     public void print_lastgeek() {
          System.out.println("Geeks");
     }
}

// Driver class
public class Main {
     public static void main(String[]
args) {
          // Creating an object of class
Three
          Three g = new Three();

          // Calling method from class One
          g.print_geek();
```

3. Hierarchical Inheritance

4. Multiple Inheritance



```
// Java program to illustrate the
// concept of Multiple inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

interface One {
	public void print_geek();
}

interface Two {
	public void print_for();
}

interface Three extends One, Two {
	public void print_geek();
}
class Child implements Three {
	@Override public void print_geek()
	{
		System.out.println("Geeks");
	}

	public void print_for() {
System.out.println("for"); }
}

// Drived class
public class Main {
	public static void main(String[]
args)
	{
		Child c = new Child();
		c.print_geek();
		c.print_for();
		c.print_geek();
	}
}
```

5. Hybrid Inheritance

#####polymorphism

Having many form

Ability to ddispaly message in many forms

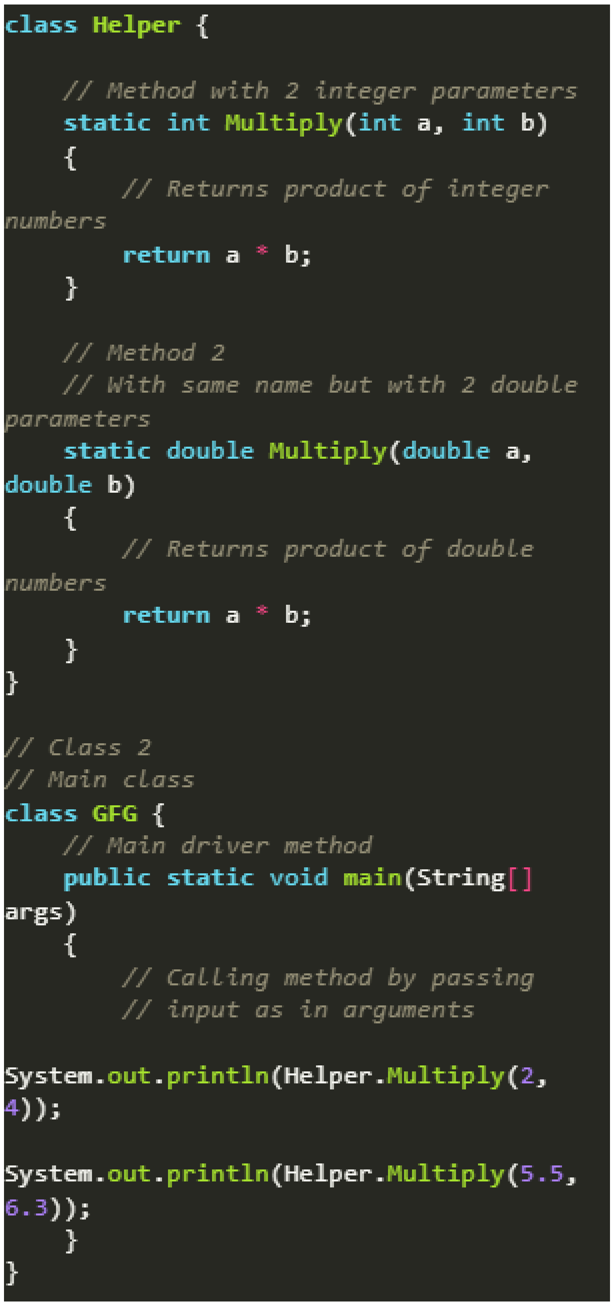
Real-life Illustration of Polymorphism in Java: A person can have different characteristics at the same time. Like a man at the same time is a father, a husband, and an employee. So the same person possesses different behaviors in different situations. This is called polymorphism.

Typr:

1,compile time

Known as static and also method overloading or operator overloading

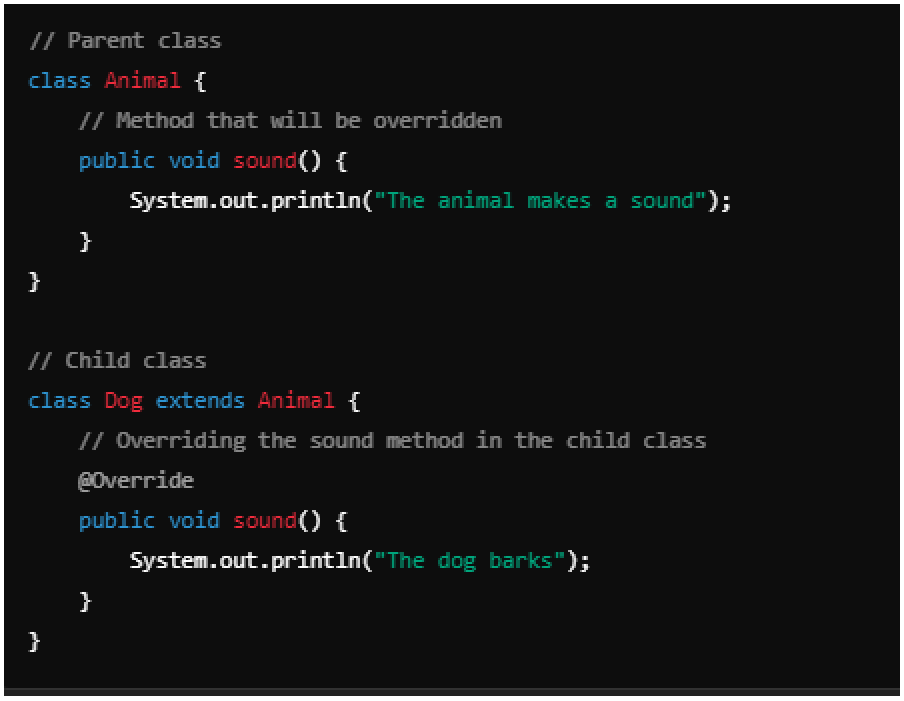
Overloading:’same functional name diff parameteds



```
class Helper {

    // Method with 2 integer parameters
    static int Multiply(int a, int b)
    {
        // Returns product of integer
numbers
        return a * b;
    }

    // Method 2
    // With same name but with 2 double
parameters
    static double Multiply(double a,
double b)
    {
        // Returns product of double
numbers
        return a * b;
    }
}

// Class 2
// Main class
class GFG {
    // Main driver method
    public static void main(String[]
args)
    {
        // Calling method by passing
        // input as in arguments

System.out.println(Helper.Multiply(2,
4));

System.out.println(Helper.Multiply(5.5,
6.3));
    }
}
```

2.runtime

A methd is again declared in child which is already in parent class



```java
// Parent class
class Animal {
    // Method that will be overridden
    public void sound() {
        System.out.println("The animal makes a sound");
    }
}

// Child class
class Dog extends Animal {
    // Overriding the sound method in the child class
    @Override
    public void sound() {
        System.out.println("The dog barks");
    }
}
```

In this animal make a sound and also dog barks will be exeited as the child overrides prent that statement also prints!!!