JAVA THEORY QUESTIONS

Q1. Explain the difference between primitive and reference data types with examples.

ANS.

Primitive Data Types:

1. Primitive types are stored in stack memory, which allows quick access and automatic memory management.

2. They store actual values directly, unlike reference types that store memory addresses.

3. Each primitive type has a fixed size in memory, such as int (4 bytes) and double (8 bytes).

4. Primitive types are immutable, meaning if you modify a value, a new copy is created rather than changing the original. 5.They are used for basic operations, including numbers (int, double), characters (char), and Boolean values (Boolean).

Example: int a = 12;

Reference Data Types:

1. Reference types are stored in heap memory, with their references (addresses) stored in stack memory.

2. They hold memory addresses that point to objects, rather than storing actual values directly.

3 Unlike primitive types, reference types have a dynamic size, meaning they can expand or shrink as needed.

4. Reference types are mutable, so changes made to an object are reflected across all references to it.

5. They are used for complex data structures, such as arrays, strings, and objects created from classes.

Example: String name = "Sarthak";

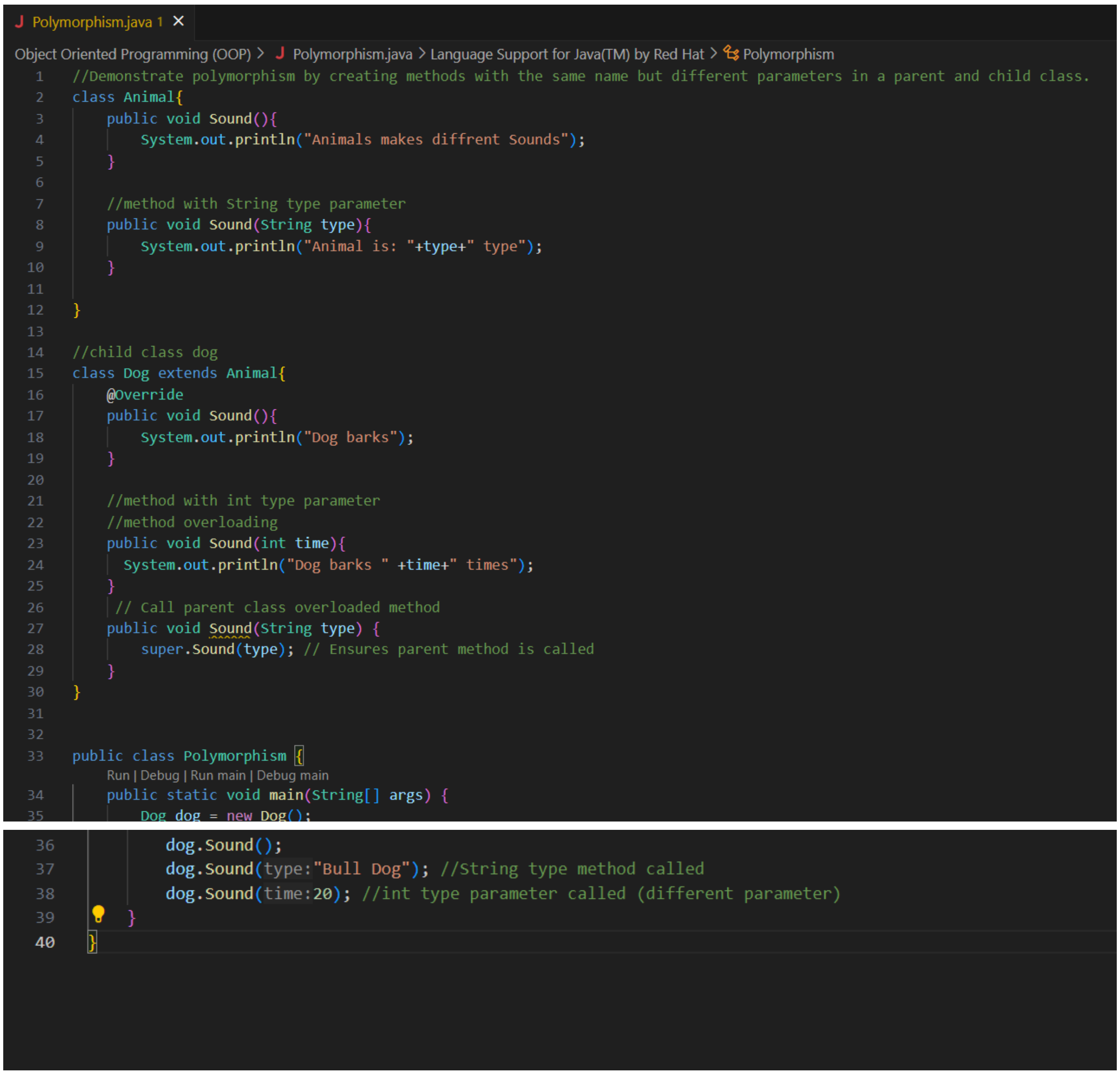
Q2. Explain the concept of encapsulation with a suitable example.

ANS. Encapsulation is a fundamental concept in object-oriented programming that helps protect data and ensure controlled access. It is similar to how we see a medicine capsule which have various medicine all together just like variables and methods in our code.

In programming, encapsulation means restricting direct access to certain variables and allowing controlled modifications through methods (like getters and setters). This prevents accidental or unauthorized changes to important data.

Encapsulation helps in maintaining security, data integrity, and better control over how data is accessed and modified.

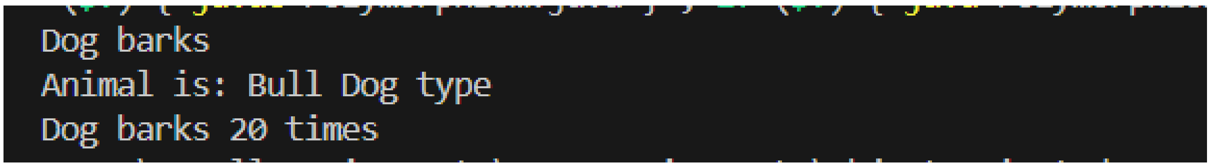
EXAMPLE:



Polymorphism.java 1
Object Oriented Programming (OOP) > Polymorphism.java > Language Support for Java(TM) by Red Hat > Polymorphism
Run | Debug | Run main | Debug main

```java
//Demonstrate polymorphism by creating methods with the same name but different parameters in a parent and child class.
class Animal{
    public void Sound(){
        System.out.println("Animals makes diffrent Sounds");
    }

    //method with String type parameter
    public void Sound(String type){
        System.out.println("Animal is: "+type+" type");
    }

}

//child class dog
class Dog extends Animal{
    @Override
    public void Sound(){
        System.out.println("Dog barks");
    }

    //method with int type parameter
    //method overloading
    public void Sound(int time){
      System.out.println("Dog barks " +time+" times");
    }
     // Call parent class overloaded method
    public void Sound(String type) {
        super.Sound(type); // Ensures parent method is called
    }
}


public class Polymorphism {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.Sound();
        dog.Sound(type:"Bull Dog"); //String type method called
        dog.Sound(time:20); //int type parameter called (different parameter)
    }
}
```

OUTPUT:



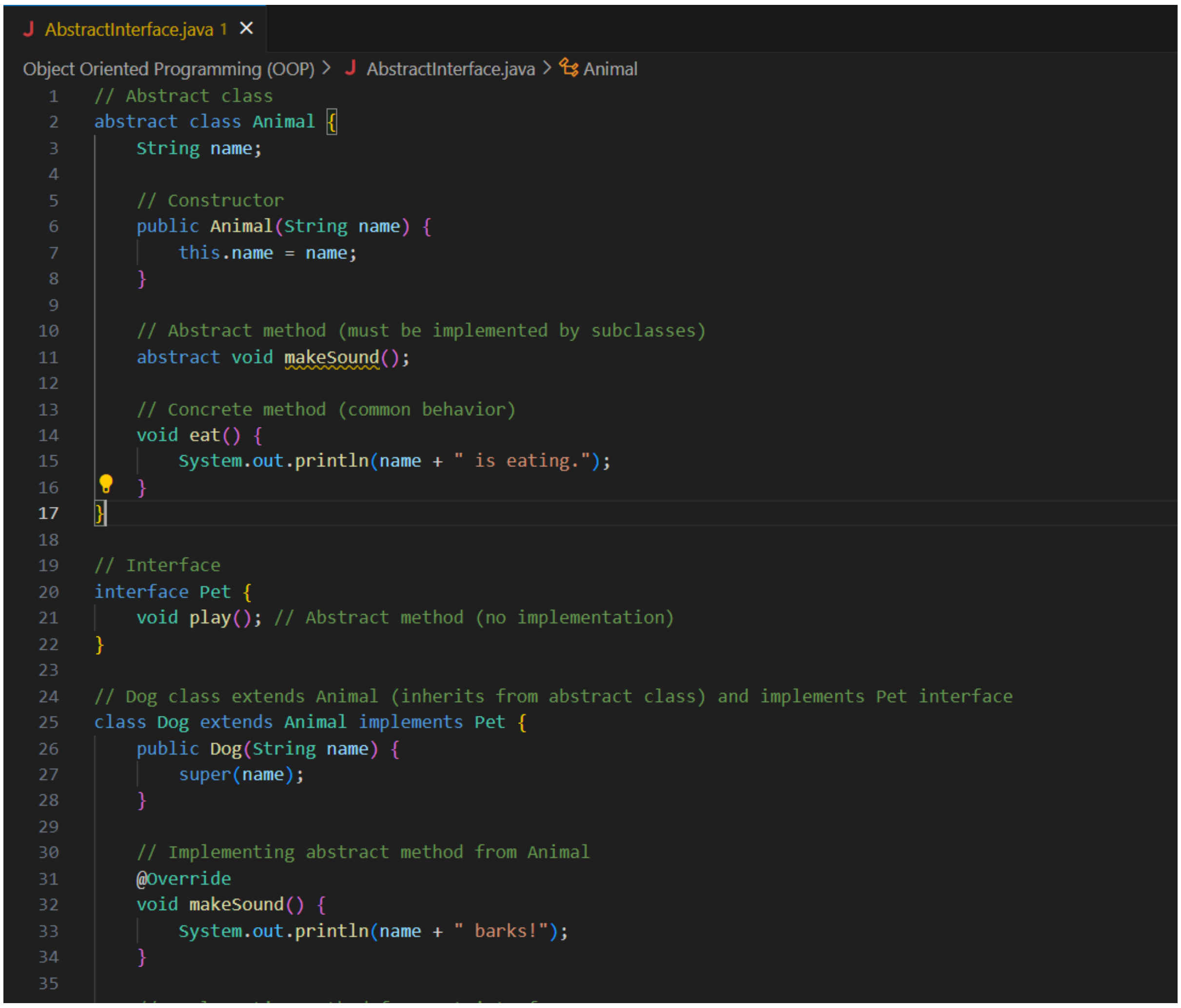
```
Dog barks
Animal is: Bull Dog type
Dog barks 20 times
```

Q. Explain the concept of interfaces and abstract classes with examples.

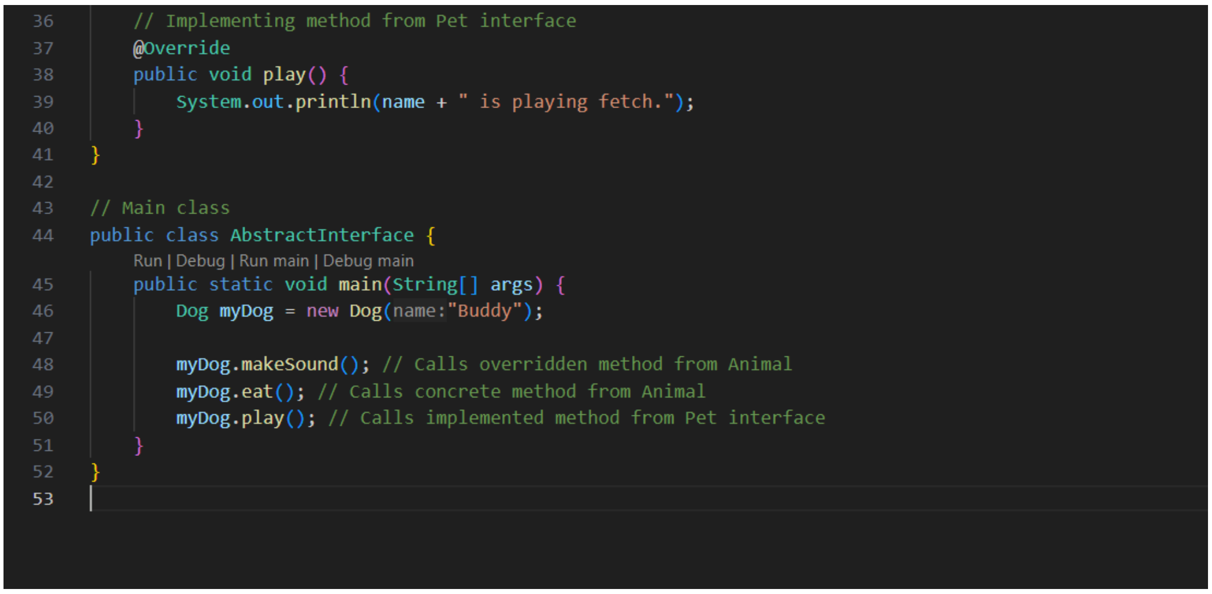
ANS: An **abstract class** in Java is a class that cannot be instantiated on its own and may contain both abstract and concrete methods. It is useful when we want to share code among related classes but still we want some of the methods to be defined by subclasses.

An **interface** is like a blueprint that only contains method declarations (without implementations). It is used to enforce a contract on classes that implement it. A class can implement multiple interfaces, unlike abstract classes, which can only be extended by one subclass.

EXAMPLE:

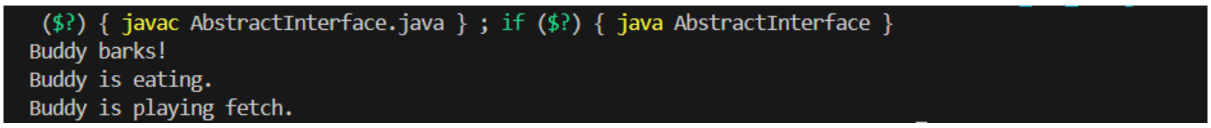

AbstractInterface.java 1 ×
Object Oriented Programming (OOP) > AbstractInterface.java > Animal

```java
// Abstract class
abstract class Animal {
    String name;

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

    // Abstract method (must be implemented by subclasses)
    abstract void makeSound();

    // Concrete method (common behavior)
    void eat() {
        System.out.println(name + " is eating.");
    }
}

// Interface
interface Pet {
    void play(); // Abstract method (no implementation)
}

// Dog class extends Animal (inherits from abstract class) and implements Pet interface
class Dog extends Animal implements Pet {
    public Dog(String name) {
        super(name);
    }

    // Implementing abstract method from Animal
    @Override
    void makeSound() {
        System.out.println(name + " barks!");
    }

```



```java
    // Implementing method from Pet interface
    @Override
    public void play() {
        System.out.println(name + " is playing fetch.");
    }
}

// Main class
public class AbstractInterface {
    public static void main(String[] args) {
        Dog myDog = new Dog(name:"Buddy");

        myDog.makeSound(); // Calls overridden method from Animal
        myDog.eat(); // Calls concrete method from Animal
        myDog.play(); // Calls implemented method from Pet interface
    }
}
```

OUTPUT:



```
  ($?) { javac AbstractInterface.java } ; if ($?) { java AbstractInterface }
 Buddy barks!
 Buddy is eating.
 Buddy is playing fetch.
```

Q. Explore multithreading in Java to perform multiple tasks concurrently.

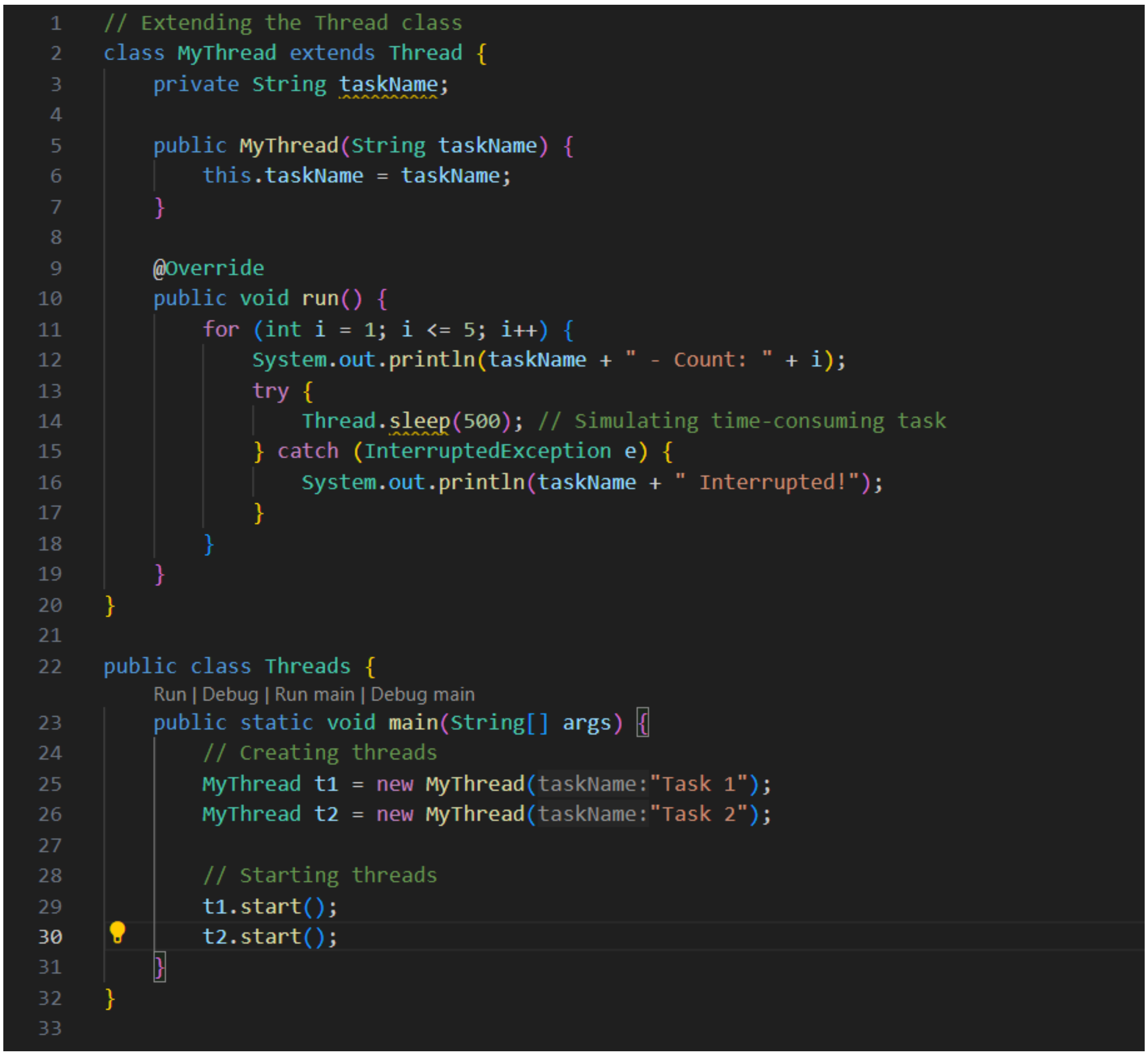
ANS: Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms:

1. Extending the Thread class
2. Implementing the Runnable Interface

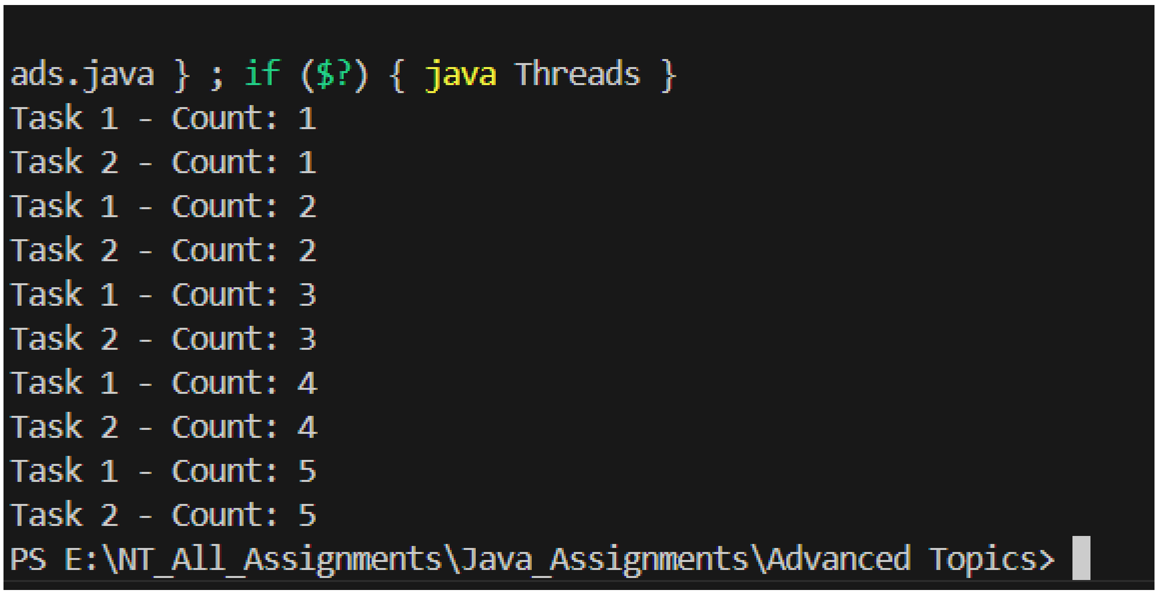
Java provides the Thread class and Runnable interface to create and manage threads. Multithreading is useful in scenarios like handling multiple user requests, performing background tasks, and parallel computing.

EXAMPLE:



```java
// Extending the Thread class
class MyThread extends Thread {
    private String taskName;

    public MyThread(String taskName) {
        this.taskName = taskName;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(taskName + " - Count: " + i);
            try {
                Thread.sleep(500); // Simulating time-consuming task
            } catch (InterruptedException e) {
                System.out.println(taskName + " Interrupted!");
            }
        }
    }
}

public class Threads {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        // Creating threads
        MyThread t1 = new MyThread(taskName:"Task 1");
        MyThread t2 = new MyThread(taskName:"Task 2");

        // Starting threads
        t1.start();
        t2.start();
    }
}
```

OUTPUT:



```
ads.java } ; if ($?) { java Threads }
Task 1 - Count: 1
Task 2 - Count: 1
Task 1 - Count: 2
Task 2 - Count: 2
Task 1 - Count: 3
Task 2 - Count: 3
Task 1 - Count: 4
Task 2 - Count: 4
Task 1 - Count: 5
Task 2 - Count: 5
PS E:\NT_All_Assignments\Java_Assignments\Advanced Topics>
```