BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

CS F213 – Object Oriented Programming

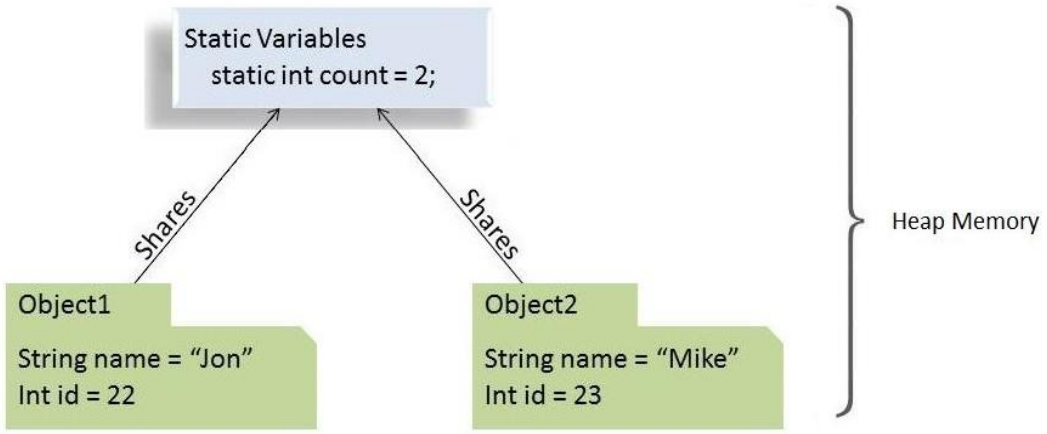
Lab 4: Static variables, methods, and blocks, Pass By Reference, Varargs

Time:	2 hours
Objective:	The objective of this lab is to deepen students' understanding of static variables, methods, and blocks, the concept of pass-by reference, variable-length argument lists (`varargs`), and wrapper classes for primitive types in Java.
Concepts Covered:	Static Members: Class-level variables, methods, and initialization blocks. Pass By Reference: Object references in method calls. Varargs: Flexible methods with variable argument lengths. Wrapper Classes: Use cases and utility methods for primitive types.
Prerequisites:	Some Java IDE (Eclipse, IntelliJ IDEA, Visual Studio Code), Basic Programming Concepts

1. Static variables, methods and blocks

Static variables, methods, and blocks in Java belong to the class rather than any specific instance, making them shared among all objects of the class. Static variables store class-level data, static methods can be called without creating an instance, and static blocks initialize static resources when the class is loaded into memory. They are crucial for optimizing memory usage and implementing utility functionality.

Fun Fact - Main method is static, since it must be accessible for an application to run, before any instantiation takes place.



1.1. Static Variables

Static variables are special variables in Java that are associated with the **class** itself rather than any specific **object (instance)** of the class. This means that **all objects of the class share a single copy of the static variable.**

Key Characteristics of Static Variables:

1. Initialization : Static variables are initialized **only once**, when the class is loaded into memory (at the start of program execution). They are initialized before any instance variables or methods.

2. Shared Across All Objects: A single copy of the static variable exists, and it is shared by all instances of the class. If one instance modifies the static variable, the change is reflected for all other instances.

3. Access Without an Object: Static variables can be accessed directly using the **class name** without the need to create an object of the class.

```
System.out.println(Math.PI);  // Math is the class, and PI is a static variable.
```

Static variables are commonly used for constants (e.g., `Math.PI`) or to maintain shared data that should be consistent across all objects (e.g., keeping a count of how many objects of a class have been created).

Syntax: *<class-name>.<variable-name>;*

```
public class Static_Variable {
      class Example {
          static int count = 0;  // Static variable
          public Example() {
              count++;
          }
      }
      public static void main(String[] args) {
            System.out.println(Example.count); // Accessing static variable
      }
}
```

1.2. Static Methods

Static methods are methods that belong to the **class** and not to any specific **instance (object)** of the class. They are particularly useful for operations that do not depend on object-specific data.

Key Characteristics of Static Methods:

1. **Restricted Access**: Static methods can only work with **static variables** and can call **other static methods.** They **cannot access instance variables or methods** because static methods operate at the class level and do not have any reference to a specific object.

2. **Cannot Use `this` or `super`:** Since static methods do not belong to any instance, they **cannot use `this` or `super` keywords**, which are tied to object context.

3. **Utility Functions** : Static methods are ideal for utility functions like mathematical operations or helper methods.

```
int result = Math.max(10, 20);  // Math is the class, max() is a static method.
```

Syntax: *<class-name>.<method-name>();*

```
public class Static_Method {
      class Calculator {
          static int add(int a, int b) {
              return a + b;
          }
      }
      public static void main(String[] args) {
            System.out.println(Calculator.add(5, 3));  // Accessing static method
      }
}
```

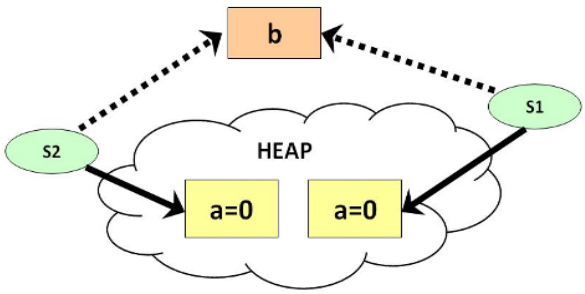
Example:

```
class MyStatic {
   int a; // Initialized to zero
   static int b; // Initialized to zero only when the class is loaded, not for each object
created.
   // Constructor incrementing static variable b
   MyStatic() {
       b++;
   }
   public void showData() {
       System.out.println("Value of a = " + a);
       System.out.println("Value of b = " + b);
   }
   // Uncommented method to demonstrate static behavior (optional)
   public static void increment() {
       // Cannot modify instance variable 'a' here because 'a' is not static
       // Uncomment the below line to observe the compilation error:
       // a++;
       b++; // Valid, as 'b' is a static variable
   }
}
public class StaticDemo {

      static {
          // Code for initialization goes here
```

```
        System.out.println("Static block executed!");
    }

   public static void main(String[] args) {
       MyStatic s1 = new MyStatic();
       s1.showData();
       MyStatic s2 = new MyStatic();
       s2.showData();
       MyStatic.increment();
       // Directly modifying static variable using class name
       MyStatic.b++;
       s1.showData();
   }
}
```

Run the above program and observe the output.

Following diagram shows, how reference variables & objects are created and static variables are accessed by the different instances.



It is possible to access a static variable from outside the class: `ClassName.VariableName`.

Uncommenting the **a++** in the static increment method results in a compilation error as the static method does not have access to the instance variable *a*.

1.3. Static Blocks

Static blocks, also known as static initialization blocks, are used to initialize static variables or execute any logic that needs to run when the class is loaded into the JVM (Java Virtual Machine). They are written as normal blocks of code enclosed in curly braces { } and preceded by the static keyword.

Syntax:

```
static {
      // Code for initialization goes here
      System.out.println("Static block executed!");
}
```

Key Features:

1. Execution During Class Loading:
    - Static blocks are executed **automatically** when the JVM loads the class, even before any objects are created or any static methods are called.
    - This makes them ideal for performing setup tasks like initializing static variables or logging configuration details.
2. Order of Execution:
    - A class can contain **multiple static blocks**, and they can appear anywhere in the class body.

- The JVM guarantees that the static blocks are executed **in the order in which they appear in the source code**.

3. **Unified Execution:**
   - During compilation, all static blocks in a class are combined into a single static block by the JVM and executed as one.

4. **Access to Static Members:**
   - Static blocks can directly reference static variables and methods.
   - If a static variable or method is used inside a static block, the variable/method is executed as part of the same initialization process.

Example:

```
public class StaticExample{
   static {
       System.out.println("This is first static block");
   }
   public StaticExample(){
       System.out.println("This is constructor");
   }
   public static String staticString = "Static Variable";
   static {
       System.out.println("This is second static block and "
                                                + staticString);
   }
   public static void main(String[] args){
       StaticExample statEx = new StaticExample();
       StaticExample.staticMethod2();
   }
   static {
       staticMethod();
       System.out.println("This is third static block");
   }
public static void staticMethod() {
       System.out.println("This is static method");
   }
   public static void staticMethod2() {
       System.out.println("This is static method2");
   }
}
```

First, all static blocks are positioned in the code, and they are executed when the class is loaded into JVM. Since the static method `staticMethod()` is called inside the third static block, it is executed before calling the main method. However, the `staticMethod2()` static method is executed after the class is instantiated because it is being called after the instantiation of the class.

Again, if you miss to precede the block with the static keyword, the block is called a "constructor block" and will be executed when the class is instantiated. The constructor block will be copied into each constructor of the class. For example, you have four parameterized constructors, and then four copies of

constructor blocks will be placed inside the constructor, one for each. Let's execute the below example and see the output.

```
public class ConstructorBlockExample {
	{
		System.out.println("This is first constructor block");
	}
	public ConstructorBlockExample() {
		System.out.println("This is no parameter constructor");
	}
	public ConstructorBlockExample(String param1) {
		System.out.println("This is single parameter constructor");
	}
	{
		System.out.println("This is second constructor block");
	}
	public static void main(String[] args) {
		ConstructorBlockExample constrBlockEx = new ConstructorBlockExample();
		ConstructorBlockExample constrBlockEx1 = new ConstructorBlockExample("param1");
	}
}
```

1.4. Private Static Methods as an Alternative to Static Blocks

Static blocks are widely used for initializing static variables, but there's an alternative: *private static methods*. These methods offer greater flexibility because they can be reused later if the static variables need to be re-initialized.

Example:

```
class PrivateStaticMethodExample {
   public static int myVar = initializeClassVariable();
   private static int initializeClassVariable() {
       // Initialization logic goes here
       return 42; // Example value
   }
}
```

Advantages of Private Static Methods:

1. **Reusability:** Unlike static blocks, private static methods can be called multiple times to reinitialize static variables if needed.
2. **Flexibility:** They provide a structured way to handle complex initialization logic without bloating the static block.

Why Not Use Public Static Methods? While public static methods can perform the same function, they are typically not used for initializing class variables. The reason is simple: initialization logic is internal to the class and should not be exposed to other classes, hence the use of `private`.

Advantages and Disadvantages of Static Blocks

Advantages:

1. **Complex Initialization:** Static blocks allow computation or logic to initialize static variables, ensuring that this code runs only once when the class is loaded.
2. **Guaranteed Execution Order:** When multiple static blocks exist, they are executed in the order they appear in the code.

Disadvantages:

1. **Size Limitation:** A static block cannot exceed **64 KB** due to JVM constraints.
2. **No Checked Exceptions:** Static blocks cannot throw checked exceptions.
3. **No `this` or `super`:** Since static blocks are executed during class loading, they cannot use `this` or `super` keywords because no instance or parent class exists at that point.
4. **Testing Challenges:** Code within static blocks is executed automatically, making unit testing more difficult.
5. **No Return Values:** Static blocks cannot return any value.

Example: Static Block and Private Static Method Combined

This example demonstrates initializing an array using both a static block and a private static method.

```
public class StaticBlockExample {
   static int[] values = initializeArray(10); // Initialize array using private static
method
   private static int[] initializeArray(int size) {
       int[] arr = new int[size];
       for (int i = 0; i < arr.length; i++) {
           arr[i] = i; // Populate array with sequential values
       }
       return arr;
   }
   void listValues() {
       for (int value : values) {
           System.out.println(value);
       }
   }
   public static void main(String[] args) {
       StaticBlockExample example = new StaticBlockExample();
       System.out.println("\nFirst object:");
       example.listValues();
       example = new StaticBlockExample();
       System.out.println("\nSecond object:");
       example.listValues();
   }
}
```

Example: Using Static Variables and Methods in a Class

This example demonstrates the use of static variables and methods in a Circle class.

```java
class Circle {
   static double PI; // Class variable shared across all instances
   private double radius;
   // Overloaded constructor
   Circle(double radius) {
       this.radius = radius;
       Circle.PI = 3.141; // Assign value to static variable
   }
   // Accessor method
   public double getRadius() {
       return radius;
   }
   // Mutator method
   public void setRadius(double radius) {
       this.radius = radius;
   }
   // Method to calculate area
   public double area() {
       return (PI * radius * radius);
   }
   // Static method to calculate circumference
   public static void getCircumference(double radius) {
       System.out.println("Circumference = " + (2 * PI * radius));
   }
}
public class TestCircle {
   public static void main(String[] args) {
       Circle c1 = new Circle(2.3);
       System.out.println("Area of c1: " + c1.area());
       // Access static method using the class name
       Circle.getCircumference(2.3);
       Circle c2 = new Circle(3.45);
       System.out.println("Area of c2: " + c2.area());
       // Accessing static method using an object reference (discouraged)
       c2.getCircumference(3.45);
       // Experiment with static method and observe behavior
       /*
        1. Make the area() function static and observe the behavior.
        2. Remove the formal argument from getCircumference() and observe the results.
        3. Replace the static keyword with final in the getCircumference() method and fix
any errors.
       */
   }
}
```

2. Pass by reference or value

In Java, when objects are passed as parameters to methods, their references and current states are passed. The behaviour depends on whether you're modifying the object's fields (state) or changing the reference itself.

Pass-by-Reference (Object State Changes):

- When you modify the fields of an object passed to a method, the changes will be reflected in the calling method.
- This happens because the method works with the same object reference.

Pass-by-Value (Object Reference Changes):

- The object reference itself is passed by value i.e. a copy of the value being passed to the method will be used.
- If you assign the reference to a new object inside the method, the original reference outside the method remains unchanged.

Example 1: Points in a 2D Plane

This example demonstrates both behaviors using a `Point` class.

```
class Point {
   private double x; // x-coordinate
   private double y; // y-coordinate
   // Constructor
   Point(double x, double y) {
       this.x = x;
       this.y = y;
   }
   // Accessor and Mutator methods
   public double getX() { return x; }
   public double getY() { return y; }
   public void setX(double x) { this.x = x; }
   public void setY(double y) { this.y = y; }
   // String representation
   public String toString() {
       return "X=" + x + ", Y=" + y;
   }
   // Method to modify the state of the object
   public static void changeState(Point other) {
       other.setX(-20);
       other.setY(-20);
   }
   // Method to change the reference of the object
   public static void changeReference(Point other) {
       other = new Point(-20, -20); // Reference change won't affect the caller
   }
}
public class PointTest {
   public static void main(String[] args) {
       // Pass-by-Reference: Modifying object state
       Point p1 = new Point(10, 20);
       System.out.println("Before changeState: " + p1);
       Point.changeState(p1); // Modifies the same object
       System.out.println("After changeState: " + p1);
       // Pass-by-Value: Attempting to change the reference
```

```
        Point p2 = new Point(100, 200);
        System.out.println("Before changeReference: " + p2);
        Point.changeReference(p2); // Reference change has no effect
        System.out.println("After changeReference: " + p2);
    }
}
```

Expected Output:

```
Before changeState: X=10, Y=20
After changeState: X=-20, Y=-20
Before changeReference: X=100, Y=200
After changeReference: X=100, Y=200
```

Example 2: Box Class and Swapping Behavior

The BOX class demonstrates how object references behave when attempting to swap two objects inside a method.

```
public class BOX {
    private double length, width, height;
    // Constructor
    BOX(double l, double w, double h) {
        length = l;
        width = w;
        height = h;
    }
    // Mutator and Accessor Methods
    public void setLength(double l) { length = l; }
    public void setWidth(double w) { width = w; }
    public void setHeight(double h) { height = h; }
    public double getLength() { return length; }
    public double getWidth() { return width; }
    public double getHeight() { return height; }
    public String toString() {
        return "Length=" + length + ", Width=" + width + ", Height=" + height;
    }
    public static void swapBoxes(BOX b1, BOX b2) {
        BOX temp = b1;
        b1 = b2;
        b2 = temp;
    }
}
```

There are two driver classes being given for the Box class. Observe the output in each case.

Driver Class 1: Using `swapBoxes` Method

```
public class BoxTest {
    public static void main(String[] args) {
        BOX b1 = new BOX(10, 40, 60);
        BOX b2 = new BOX(20, 30, 80);
        System.out.println("Before swapping:");
```

```
        System.out.println("Box 1: " + b1);
        System.out.println("Box 2: " + b2);
        BOX.swapBoxes(b1, b2); // This won't swap the objects
        System.out.println("After swapping:");
        System.out.println("Box 1: " + b1);
        System.out.println("Box 2: " + b2);
    }
}
```

Driver Class 2: Swapping Using Local References

```
public class BoxTest {
    public static void main(String[] args) {
        BOX b1 = new BOX(10, 40, 60);
        BOX b2 = new BOX(20, 30, 80);
        System.out.println("Before swapping:");
        System.out.println("Box 1: " + b1);
        System.out.println("Box 2: " + b2);
        BOX temp = b1; // Swapping locally
        b1 = b2;
        b2 = temp;
        System.out.println("After swapping:");
        System.out.println("Box 1: " + b1);
        System.out.println("Box 2: " + b2);
    }
}
```

In Java, objects are always passed to methods by reference, but the reference itself is passed by value. This means that while the state of the object (its fields) can be modified inside the method, the reference pointing to the object cannot be permanently changed in the calling method. This distinction explains why attempts to swap objects inside a method fail.

In the `swapBoxes` method, when the references `b1` and `b2` are passed, the method receives copies of these references. Any changes to these references, such as swapping them, remain local to the method and do not affect the original references in the calling code. As a result, after the method finishes, the original references in the calling method still point to their respective objects, and no swapping is observed.

On the other hand, swapping works in the `main` method directly because the references being swapped are the actual ones declared in the calling scope. By using a temporary variable to hold one reference and reassigning the others, the original references in the calling method are modified, resulting in successful swapping.

---

3. Wrapper Classes

Java has **8 primitive types**: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `char`. These primitive types are **not objects**, which means they don't have methods like objects do. However, there are situations where Java needs to treat primitive values as objects. For example:

```
System.out.println(10);
```

Here, the number **10** is a primitive type (**int**). Since primitive types don't have methods, it cannot call methods like **toString()** directly. Java automatically converts this **int** into an object of the wrapper class **Integer** so it can call **toString()** to print the value. Internally, the compiler rewrites the statement as:

```
System.out.println(new Integer(10).toString());
```

This conversion of a primitive type to its corresponding wrapper object is called **autoboxing** and converting it back to a primitive is called **auto unboxing**. Modern versions of Java (1.5 and later) handle these conversions automatically, but earlier versions required manual conversion.

Each primitive type in Java has a corresponding wrapper class:

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Example Programs

CASE 1: Manual Boxing and Unboxing

Here's an example to observe how wrapper classes were used before autoboxing and auto unboxing:

```
public class Test1 {
   public static void main(String[] args) {
       Integer I = new Integer(10); // Create an Integer object
       Integer J = new Integer(20); // Create another Integer object

       // Explicitly get the primitive int value
       System.out.println(I.intValue()); // Output: 10
       System.out.println(I); // Output: 10 (implicitly calls toString())

       System.out.println(J.intValue()); // Output: 20
       System.out.println(J); // Output: 20
       // Add primitive values explicitly
```

```
        Integer K1 = new Integer(I.intValue() + J.intValue());
        System.out.println(K1); // Output: 30

        // Automatic unboxing (Java rewrites this to use intValue() internally)
        Integer K2 = I + J + K1;
        System.out.println(K2); // Output: 60
    }
}
```

CASE 2: Using Autoboxing and Autounboxing

Modern versions of Java simplify this process with **autoboxing** and **auto unboxing**. You don't need to explicitly create wrapper objects or call methods like `intValue()`.

```
public class Test2 {
    public static void main(String[] args) {
        Integer I = 10; // Autoboxing: primitive int to Integer
        Integer J = 20; // Autoboxing

        // Autounboxing: Automatically converts Integer to int
        System.out.println(I.intValue()); // Output: 10
        System.out.println(I); // Output: 10

        System.out.println(J.intValue()); // Output: 20
        System.out.println(J); // Output: 20

        // Autounboxing and arithmetic
        Integer K1 = I + J; // Autounboxing for addition, then autoboxing for result
        Integer K2 = I + J + K1;
        System.out.println(K2); // Output: 60
    }
}
```

4. Variable Arguments (varargs)

Understanding Varargs (Variable Arguments) in Java

In Java, there are situations where you may want to write a method that can accept a variable number of arguments. For example, you might want to create a method that calculates the sum of any number of integers. Instead of overloading the method for different numbers of parameters, Java provides a feature called **varargs** (short for "variable arguments").

What Are Varargs?

Varargs allow you to pass **zero or more arguments** to a method. You define a varargs parameter by using an **ellipsis** (`...`) after the type of the parameter in the method declaration.

Syntax

```
returnType methodName(returnType... variableName) {
            // method body where variableName is considered an array of type 'dataType'
}
```

- The `...` indicates that the method can accept **any number of arguments** of the specified type.
- Inside the method, the variable is treated as an **array**.

Key Points

1. **Only one varargs parameter is allowed per method**.
2. The varargs parameter must always be the **last parameter** in the method signature.
3. Varargs are optional, so you can call the method with no arguments.

Example: Using Varargs

Let's create a method to calculate the sum of numbers using varargs:

```
public class VarargsExample {
   // Method with varargs
   public static int sum(int... numbers) {
       int total = 0;
       for (int num : numbers) { // 'numbers' is treated as an array
           total += num;
       }
       return total;
   }
   public static void main(String[] args) {
       System.out.println(sum());            // No arguments, output: 0
       System.out.println(sum(10));         // One argument, output: 10
       System.out.println(sum(10, 20, 30)); // Multiple arguments, output: 60
   }
}
```

1. **Method Declaration**:
   The method `sum(int... numbers)` can take zero or more integers as arguments.

2. **Inside the Method**:
   The `numbers` parameter is treated as an **array of integers** (`int[]`). You can iterate through this array using a `for` loop or any other array-processing technique.

3. **Calling the Method**:
   - `sum()` → No arguments are passed, so the `numbers` array is empty, and the total remains 0.
   - `sum(10)` → A single argument is passed, and the total is `10`.
   - `sum(10, 20, 30)` → Multiple arguments are passed, and the total is `60.`

Advanced Example: Varargs with Other Parameters

You can combine varargs with regular parameters, but **the varargs must always come last**. If you put them as the first parameter, then it will result in a **compile-time error.**

```
public class VarargsExample2 {
   public static String joinStrings(String separator, String... strings) {
       StringBuilder result = new StringBuilder();
       for (int i = 0; i < strings.length; i++) {
           result.append(strings[i]);
           if (i < strings.length - 1) {
               result.append(separator);
           }
       }
       return result.toString();
   }
   public static void main(String[] args) {
       System.out.println(joinStrings(", ", "apple", "banana", "cherry"));
       // Output: apple, banana, cherry

       System.out.println(joinStrings(" - ", "Java", "Python", "C++"));
       // Output: Java - Python - C++
   }
}
```

Why Use Varargs?

- **Flexibility**: You can write a single method to handle a variable number of arguments.
- **Simplifies Code**: No need to overload methods for different numbers of parameters.
- **Readable**: Makes your code cleaner and easier to understand.

When Not to Use Varargs?

- If the method needs to process a **fixed number of arguments**, use regular parameters.
- Avoid using varargs with large data sets, as it creates an array internally and can impact performance.

---

5. Exercises

Q1. Design a `BankAccount` class with:

- **Instance fields:** `accountNumber`, `balance`
- **Static fields:** `BankName`, `interestRate`, `totalBankReserve`
- **Methods:** `deposit()`, `withdraw()`, `calculateInterest()`, static getters for static fields

Implement:

- `deposit()` increases `balance`, `totalBankReserve`, and increases `interestRate` by 0.1% every 1000 units increase in `totalBankReserve`.
- `withdraw()` decreases `balance` and `totalBankReserve` if sufficient funds.

Create objects, perform transactions, and display relevant information (bank name, interest rate, balances, etc.).

Q2. Problem Statement:

1. **Create a `Product` class** with instance fields: **`productName`** (String), **`price`** (Double), **`quantity`** (Integer). Use the wrapper classes instead of primitive classes.
2. **Create a `calculateTotalPrice` method:**
    - This method should be **static** and use **varargs** to accept a variable number of **`Product`** objects.
    - The method should calculate and return the total price of all the products passed as arguments.

Partial Boilerplate Code:

```
public class Product {
   private final String productName;
   private final Double price;
   private final Integer quantity;
   public Product(String productName, Double price, Integer quantity) {
       this.productName = productName;
       this.price = price;
       this.quantity = quantity;
   }
   public String getProductName() {
       return productName;
   }
   public Double getPrice() {
       return price;
   }
   public Integer getQuantity() {
       return quantity;
   }
//    Implement this static function
public static Double calculateTotalPrice() {
}
   public static void main(String[] args) {
       Product product1 = new Product("Laptop", 1500.00, 1);
       Product product2 = new Product("Mouse", 50.00, 2);
       Product product3 = new Product("Keyboard", 150.00, 1);
       Product product4 = new Product("Phone", 800.00, 1);
       // Attempt to modify (this should not be easily possible)
       // product1.price = 1000.00; // This might not compile or have no effect
       // Double total = calculateTotalPrice(product1, product2);
       // System.out.println("Total Price: " + total);
   }
}
```