Mastering Abstraction

Resolving Confusion between Abstract Class and Interface in Java

Introduction

Abstraction is one of the core pillars of Object-Oriented Programming. In Java, abstraction is mainly achieved using **Abstract Classes** and **Interfaces**.
Students often get confused about when to use which, whether interfaces are slower, and what practical differences really matter in real applications.

This assignment explains:

1. When to use an Abstract Class vs an Interface using a real-life story and Java code
2. Whether interface method invocation is slower than abstract class method invocation
3. A clear comparison table for quick understanding

1. When to Use Interface vs Abstract Class (Story + Code)

Story: Online Food Delivery System

Imagine you are designing an **Online Food Delivery App**.

Different restaurants exist:

- Pizza Restaurant
- Burger Restaurant
- Chinese Restaurant

Common Rule

Every restaurant **must**:

- Prepare food
- Pack food

But:

- Some restaurants give free water
- Some give free dessert
- Some give nothing extra

Design Decision

- **Interface** → Used to define *what must be done* (rules/contract)
- **Abstract Class** → Used to define *what is common + what can vary*

Step 1: Interface (Rule Book)

```
interface Restaurant {
    void prepareFood();
    void packFood();
}
```

Why Interface here?

- Every restaurant must follow these rules
- No shared state
- Supports multiple inheritance
- Defines capability, not identity

Step 2: Abstract Class (Common Base)

```
abstract class BaseRestaurant implements Restaurant {
    String restaurantName;

    BaseRestaurant(String name) {
        this.restaurantName = name;
    }

    void printBill() {
        System.out.println("Bill printed for " + restaurantName);
```

```
    }

    abstract void giveExtra();
}
```

Why Abstract Class here?

- Common data (`restaurantName`)
- Common behavior (`printBill`)
- Partial implementation
- Logical “is-a” relationship

Step 3: Concrete Class Implementation

```
class PizzaRestaurant extends BaseRestaurant {

    PizzaRestaurant() {
        super("Pizza Hut");
    }

    @Override
    public void prepareFood() {
        System.out.println("Preparing Pizza");
    }

    @Override
    public void packFood() {
        System.out.println("Packing Pizza");
    }

    @Override
    void giveExtra() {
        System.out.println("Free garlic bread");
    }
}
```

Main Class

```
public class Main {
    public static void main(String[] args) {
        Restaurant r = new PizzaRestaurant();

        r.prepareFood();
        r.packFood();

        BaseRestaurant br = (BaseRestaurant) r;
        br.printBill();
        br.giveExtra();
    }
}
```

When to Use What (Final Rule)

Use Interface when:

- You need multiple inheritance
- You want to define a contract
- No shared state
- You care about “can do”

Use Abstract Class when:

- You need shared code or variables
- You want partial implementation
- You care about “is a type of”

2. Are Interface Method Calls Slower Than Abstract Class Calls?

Short Answer

No.
The performance difference does **not** depend on whether it is an interface or abstract class.
It depends on **the reference type used to call the method**, not where the method is declared.

Explanation

Java uses **dynamic dispatch**.
At runtime, JVM decides which method to call.

- Interface reference → JVM needs extra lookup
- Class or abstract reference → Faster resolution
- JVM JIT compiler optimizes most cases

In real-world applications, the difference is **negligible**.

Simple Demonstration Example

```
interface Shape {
    int area();
}

abstract class AbstractShape implements Shape {
    int side = 10;

    @Override
    public int area() {
        return side * side;
    }
}

class Square extends AbstractShape {
```

```
}
```

Method Invocation Test

```
public class TestPerformance {
    static int loops = 1_000_000;

    public static void main(String[] args) {
        Shape s = new Square();
        AbstractShape a = new Square();
        Square c = new Square();

        long start, end;

        start = System.nanoTime();
        for (int i = 0; i < loops; i++) {
            s.area();
        }
        end = System.nanoTime();
        System.out.println("Interface ref: " + (end - start));

        start = System.nanoTime();
        for (int i = 0; i < loops; i++) {
            a.area();
        }
        end = System.nanoTime();
        System.out.println("Abstract ref: " + (end - start));

        start = System.nanoTime();
        for (int i = 0; i < loops; i++) {
            c.area();
        }
        end = System.nanoTime();
        System.out.println("Concrete ref: " + (end - start));
    }
}
```

Observation (Matches StackOverflow Result)

- Interface reference calls are slightly slower
- Abstract and concrete references are similar
- Difference is in **nanoseconds**
- JVM optimizations often eliminate this difference

Conclusion on Performance

✓ Interface methods are **not inherently slow**
✓ Reference type matters more than method location
✓ Design clarity is far more important than micro-optimization

3. Comparison Table: Abstract Class vs Interface

Feature	Abstract Class	Interface
Multiple Inheritance	Not allowed	Allowed
Variables	Instance variables allowed	public static final only
Methods	Abstract + Concrete	Abstract, default, static
Constructors	Yes	No
State	Can hold state	No state
Access Modifiers	Any	Methods are public by default
Speed	Slightly faster via class ref	Slightly slower via interface ref
Use Case	Base class with shared code	Contract / capability

Final Conclusion

- **Interface** represents *what a class can do*
- **Abstract class** represents *what a class is*
- Performance difference is negligible
- Choose clarity, flexibility, and maintainability over micro-speed concerns