10.6-Need of Interface

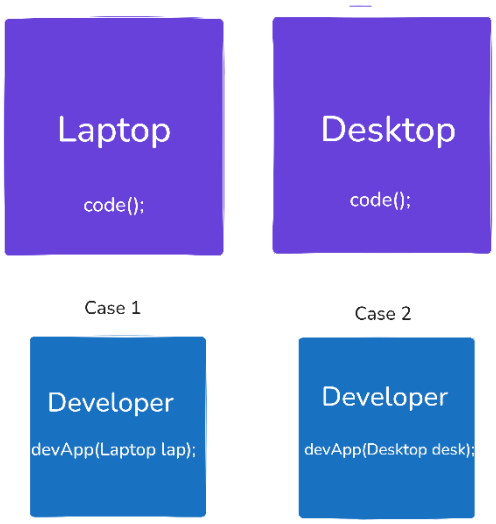
In this section, we'll explore the importance of interfaces in Java and understand their usage with examples. Interfaces play a crucial role in achieving abstraction, flexibility, and adhering to design principles like **"programming to an interface"**.

Initial Scenario: Without Interface

Consider the following example:

```
class Laptop {
    public void code() {
        System.out.println("Code, compile, and run");
    }
}

class Desktop {
    public void code() {
        System.out.println("Code, compile, and run... faster");
    }
}

class Developer {
    public void devApp(Laptop laptop) {
        laptop.code();
    }
}


public class MyClass {
  public static void main(String args[]) {
    Laptop laptop = new Laptop();
        Desktop desktop = new Desktop();
        Developer developer = new Developer();
        developer.devApp(laptop); //Output : Code, compile, and run

  }
}
```

Explanation:



In this example, there are three main classes:

1. **Laptop** – Represents a laptop that can run code.
2. **Desktop** – Represents a desktop that runs code faster than a laptop.
3. **Developer** – The developer class that uses the device (either Laptop or Desktop) to write and run code.

The Developer class has a method devApp(Laptop laptop), which takes a Laptop object as a parameter and calls its code() method. In the Demo class, we instantiate the Laptop, Desktop, and Developer classes, and pass the Laptop object to the developer.

Problem:

Initially, the developer works with a laptop. But later, if the company decides to provide a desktop instead, changes must be made to the Developer class to accommodate the desktop. This violates the design principle of flexibility. Each time the working environment changes, we would have to modify the Developer class.

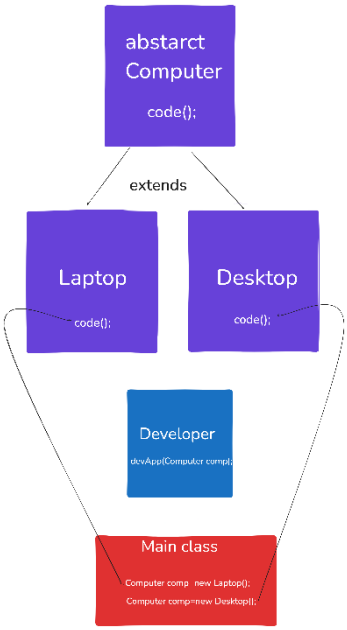
This is where **interfaces** come into play.

Solution: Using Abstraction

One possible improvement is to use **abstraction** to generalize both the Laptop and Desktop classes. Both are essentially computers, so we can introduce an abstract class called Computer:

```java
abstract class Computer {
    public abstract void code();
}

class Laptop extends Computer {
    public void code() {
        System.out.println("Code, compile, and run");
    }
}

class Desktop extends Computer {
    public void code() {
        System.out.println("Code, compile, and run... faster");
    }
}

class Developer {
    public void devApp(Computer comp) {
        comp.code();
    }
}

public class MyClass {
  public static void main(String args[]) {
            Computer laptop = new Laptop();
        Computer desktop = new Desktop();
        Developer developer = new Developer();
        developer.devApp(laptop); // Output: "Code, compile, and run"
        developer.devApp(desktop); // Output: "Code, compile, and run... faster"

  }
}
```

Explanation:



- We created an abstract class Computer with an abstract method code().
- Both Laptop and Desktop classes extend the Computer class and provide their specific implementation of the code() method.
- The Developer class now works with the Computer type instead of a specific Laptop or Desktop. This makes the developer's code flexible and allows the developer to work with any type of computer without modifying the Developer class.

In this case:

- If a laptop is passed to devApp(), it outputs: **"Code, compile, and run"**.
- If a desktop is passed, it outputs: **"Code, compile, and run... faster"**.

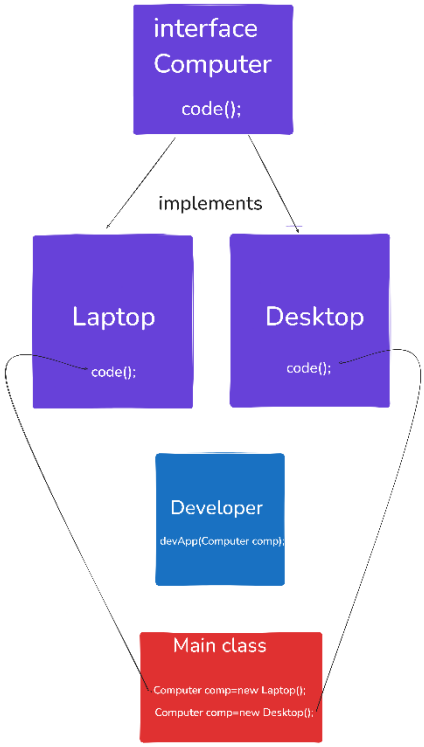
Issue with Abstract Classes:

While this solution is better, **abstract classes** only provide partial abstraction. We still need to define common code (if any) in the abstract class. If the scenario requires **full abstraction** (where only method declarations are needed without any method bodies), we should use **interfaces**.

Better Solution: Using Interface

Java interfaces provide a higher level of abstraction than abstract classes. They allow us to declare methods without providing any implementation. Let's update the code to use an interface instead of an abstract class:

```
interface Computer {
    void code();
}

class Laptop implements Computer {
    public void code() {
        System.out.println("Code, compile, and run");
    }
}

class Desktop implements Computer {
    public void code() {
        System.out.println("Code, compile, and run... faster");
    }
}

class Developer {
    public void devApp(Computer comp) {
        comp.code();
    }
}

public class MyClass {
  public static void main(String args[]) {
        Computer laptop = new Laptop();
        Computer desktop = new Desktop();
        Developer developer = new Developer();
        developer.devApp(laptop); // Output: "Code, compile, and run"
        developer.devApp(desktop); // Output: "Code, compile, and run... faster"


  }
}
```



Why Use Interfaces?

- **Full Abstraction**: All methods in an interface are abstract by default (in older versions of Java), meaning the implementing classes must provide the method body. This ensures flexibility.
- **Decoupling**: The Developer class no longer depends on the concrete implementations (Laptop or Desktop) but only on the abstraction (Computer interface). This makes the code more modular and extensible.

Conclusion:

By using an interface, we provide a flexible design where:

- We don't need to change the Developer class if a new type of computer (e.g., Tablet) is introduced.
- The code becomes more maintainable and adheres to good design principles, such as **programming to an interface**.