Abstraction in Java

What is Abstraction?

Abstraction in Java is a way of hiding the complex details of how things work and showing only the essential features. It simplifies interaction by focusing on what an object does rather than how it does it.

Real-Life Example:

Think of AC remote control. You can change the temperature without knowing how the remote sends signals to the AC. The remote hides the complex internal workings and presents a simple interface.

Abstraction in Java:

In Java, abstraction is implemented through **abstract classes** and **interfaces**:

- **Abstract Classes:** These classes cannot be instantiated directly. They can contain both abstract methods (without implementation) and concrete methods (with implementation). Abstract methods must be implemented by subclasses.
- **Interfaces:** Interfaces can achieve 100% abstraction. They only define method signatures without any implementation. Classes that implement an interface must provide the method implementations.

When to Use Abstraction:

Abstraction is useful when you want to:

- Define a base class that outlines common features or behaviors without specifying exact details.
- Allow subclasses to provide specific implementations for some methods.

For instance, if you have a base class called Shape, it might define method like area(), but the actual implementations for these method is provided by subclasses like Circle or Rectangle.

Advantages of Abstraction:

- Reduces complexity by hiding implementation details.
- Increases code reusability and security.
- Makes code easier to maintain and modify.
- Provides a clear and simple interface to the user.

Disadvantages of Abstraction:

- Can make understanding and debugging more difficult.
- May introduce additional complexity and performance overhead.
- Overuse can lead to unnecessary complexity and reduced flexibility.

abstract Keyword in Java

The abstract keyword in Java helps us achieve abstraction, a core concept in Object-Oriented Programming (OOP). It allows us to define classes and methods that are not fully implemented, leaving some parts to be defined by subclasses. Here's a simplified overview of how it works:

Characteristics:

1. **Abstract Classes**:
    - **Cannot Be Instantiated**: You can't create an object directly from an abstract class. Instead, you extend it to create concrete (fully implemented) subclasses.
    - **Can Contain Both Abstract and Concrete Methods**: An abstract class can have methods with implementations (concrete methods) as well as methods without implementations (abstract methods).
    - **Can Have Constructors**: Abstract classes can have constructors to initialize variables. However, these constructors are called through subclasses, not directly.
    - **Can Have Instance Variables**: Abstract classes can have variables that are used by both the class itself and its subclasses.
    - **Can Implement Interfaces**: Abstract classes can implement interfaces, requiring them to provide concrete implementations of all interface methods.
2. **Abstract Methods**:
    - **No Implementation**: Abstract methods are declared without a body. They end with a semicolon, not curly braces.
    - **Must Be Implemented**: Any class that extends an abstract class must provide implementations for all its abstract methods, unless it too is abstract.
3. **Important Rules for Abstract Methods**:
    - A class containing one or more abstract methods must also be declared abstract.
    - Abstract methods cannot be combined with certain modifiers like final, static, private, etc.

Example:

1. **Defining an Abstract Class**:

```
abstract class Animal {
    // Abstract method (does not have a body)
    abstract void makeSound();
    // Regular method (does have a body)
    void sleep() {
        System.out.println("Zzz...");
    }
}
```

2. **Extending an Abstract Class**:

```
class Dog extends Animal {
    // Providing implementation for the abstract method
    void makeSound() {
        System.out.println("Woof");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.makeSound();  // Outputs: Woof
        myDog.sleep();      // Outputs: Zzz...
    }
}
```

Advantages of Using abstract Keyword

1. **Common Interface**: Defines a standard set of methods that all subclasses must implement, ensuring consistency.
2. **Polymorphism**: Allows treating different subclass objects as instances of the abstract class, making your code more flexible and easier to extend.
3. **Code Reuse**: Abstract classes can provide common methods and variables that all subclasses can use, reducing code duplication.
4. **Enforces Implementation**: Guarantees that certain methods are implemented in all subclasses, improving reliability.
5. **Late Binding**: Determines the actual method to call at runtime, allowing for more dynamic and adaptable code.

Abstract vs. Final

- **Abstract Class**: Used to provide a base for other classes to extend. Cannot be instantiated on its own.
- **Final Class**: Cannot be extended or subclassed. Combining final and abstract does not make sense, as they serve opposing purposes.

Abstract Classes in Java

What is an Abstract Class?

An abstract class in Java is a class that cannot be instantiated on its own. Instead, it must be extended by another class to use its properties and methods. It is declared using the abstract keyword.

Characteristics:

1. **Cannot Be Instantiated:**
    - You cannot create an object directly from an abstract class. For example, Shape s = new Shape(); will cause an error.
2. **Can Have Constructors:**
    - Abstract classes can have constructors, which are called when a subclass instance is created.
3. **May Have Abstract and Non-Abstract Methods:**
    - Abstract methods are declared without a body (e.g., abstract void draw();), and must be implemented by subclasses.
    - Non-abstract methods can have a body and can be used as is or overridden by subclasses.
4. **Can Have Final Methods:**
    - Abstract classes can have final methods, which means they cannot be overridden by subclasses.
5. **Can Include Static Methods:**
    - Static methods can be defined in abstract classes and called without creating an object.
6. **Abstract Class Without Abstract Methods:**
    - An abstract class can have no abstract methods but still cannot be instantiated.
7. **Abstract Inner Classes:**
    - Abstract classes can include abstract inner classes, which also need to be implemented by further subclasses.
8. **Abstract Methods in Child Classes:**
    - If a subclass of an abstract class does not implement all abstract methods, it must also be declared abstract.

Examples:

1. **Abstract Class with Abstract Method:**

```
abstract class Shape {
    abstract void draw(); // Abstract method
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a Circle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle();
        shape.draw();
    }
}
```

2. **Abstract Class with Constructor and Methods:**

```
abstract class Subject {
    Subject() {
        System.out.println("Learning Subject");
    }

    abstract void syllabus(); // Abstract method

    void learn() {
        System.out.println("Preparing Right Now!");
    }
}

class IT extends Subject {
    void syllabus() {
        System.out.println("C, Java, C++");
    }
}

public class Main {
    public static void main(String[] args) {
        Subject subject = new IT();
        subject.syllabus();
        subject.learn();
    }
}
```

3. **Abstract Class with Static Method:**

```
abstract class Helper {
    static void demo() {
        System.out.println("Static method in abstract class");
    }
}

public class Main {
    public static void main(String[] args) {
        Helper.demo(); // Call static method directly
    }
}
```

4. **Abstract Class with Final Method:**

```
abstract class Base {
    final void show() {
        System.out.println("Final method in abstract class");
    }
}

class Derived extends Base {
}

public class Main {
    public static void main(String[] args) {
        Base base = new Derived();
        base.show();
    }
}
```

Abstract Classes in Java:

An abstract class:

- Is declared using the abstract keyword.
- Can have both abstract methods and concrete methods.
- Cannot be instantiated directly.

Example:

```
abstract class Shape {
    String color;

    abstract double area();
    public abstract String toString();

    public Shape(String color) {
        this.color = color;
    }

    public String getColor() {
        return color;
    }
}

class Circle extends Shape {
    double radius;

    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * radius * radius;
    }

    @Override
    public String toString() {
        return "Circle color is " + getColor() + " and area is: " + area();
    }
}
```

Interfaces in Java: A Simple Guide

What is an Interface?

- An interface in Java is like a special type of class that only contains abstract methods. Abstract methods don't have a body (i.e., they don't do anything on their own).
- To create an interface, we use the interface keyword. Unlike regular classes, you can't create objects directly from an interface.

How to use?

- To use an interface, other classes must implement it. We use the implements keyword to do this.
- Interfaces can also extend other interfaces using the extends keyword.

Why we Use Interfaces?

- **Abstraction**: Interfaces help us achieve abstraction by defining methods that must be implemented by classes. For example, an interface Polygon might have a method getArea() that different polygons will implement differently.
- **Specifications**: They set rules that classes must follow. Any class implementing the Polygon interface must provide its own version of getArea().
- **Multiple Inheritance**: Interfaces allow a form of multiple inheritance, which means a class can implement multiple interfaces.

Default Methods in Interfaces (Java 8 and Later)

- Java 8 introduced default methods, which allow interfaces to have methods with actual implementations. You use the default keyword to create these methods.

Why Default Methods?

If you add a new method to an interface, all classes implementing it need to update their code to include the new method. Default methods avoid this problem by providing a default implementation.

Static Methods in Interfaces (Java 8 and Later)

Interfaces can now have static methods. You access these methods using the interface name, similar to static methods in classes.

Private Methods in Interfaces (Java 9 and Later)

Java 9 added private methods to interfaces. These methods help other methods within the interface but can't be accessed from outside the interface.

Example:

```
interface Shape {
    double calculateArea();
}

class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```