# Abstract and Interface

## Abstract class

A abstract class is a class that you can't instantiated (can't make an object from), but can be inherited by using the keyword `extends` by other classes. A abstract class is a class that is declared with the `abstract` keyword. This is used so the sub classes can share method and variables. To instantiate a abstract class you have to create a sub class and then instantiate that sub class.

An abstract class may contain abstract methods, concrete methods, instance variables, and constructors. When you extend an abstract class, you must either override all the abstract methods in the subclass or declare the subclass itself as abstract. A abstract method can only be created inside an abstract class and when a method is abstract you don't specify a body. Because the method is abstract all the classes children have to have their own implementation of that method. The reason why you would want to do this is because you want to make sure that all the sub classes have a method with the same name but you don't want to specify what the method does. Abstract methods are declared using the `abstract` keyword. For example:

```
public abstract void abstractMethod();
```

Abstract classes can also have concrete (non-abstract) methods, which have an implementation. These methods are inherited by subclasses and can be used as-is or overridden. Subclasses are not required to provide an implementation for concrete methods.

When a subclass inherits a parent class in Java, the constructors of the parent class are not inherited by the subclass. However, the subclass does have access to the constructors of the parent class, by using the keyword `super`.

Constructors are not inherited. Each class, including the subclass, must define its own constructors.

```
public abstract class ParentClass {
    int result;
    public ParentClass(int value) {
        // Constructor code
        result = value + 10;

    }
}

public class SubClass extends ParentClass {
    public SubClass(int value) {
        super(value); // Explicitly invoking superclass constructor
        // Subclass constructor code
    }
}
```

Variables in an abstract class are **not** automatically static and final as they are in an interface. Each subclass of an abstract class has its own instance variables, including any variables it inherits from the abstract class.

When a subclass extends an abstract class, it can choose to override the value of a variable inherited from the abstract class by providing its own implementation. The subclass can have its own independent value for that variable, separate from other subclasses or the abstract class itself.

Here's an example to illustrate this:

```
abstract class AbstractClass {
    protected int variable = 10;
    public abstract void someMethod();
}

class SubClass extends AbstractClass {
    public void someMethod() {
        variable = 20;  // Changing the value of the variable in SubClass
    }
}

class AnotherSubClass extends AbstractClass {
    public void someMethod() {
        // Do something else
    }

    public void printVariable() {
        System.out.println(variable);  // Prints 10
    }
}
```

The `AnotherSubClass` also extends the `AbstractClass`, but it does not modify the value of `variable`.

Therefore, the value of the variable in each subclass can be different based on how it is implemented, and changing the value in one subclass does not affect the values in other subclasses or the abstract class itself.

---

## Interface

An interface is a contract that specifies the behavior that a class must implement but not how to implement it.

To use a interface you use the keyword `implements` and then the name of the interface. When you implement a interface you have to implement all the methods that aren't concrete in the interface. A sub class can implement multiple interfaces. Interfaces provide a way to achieve polymorphism in Java. You also can't make an object from an interface.

```
public interface Shape {
    double getArea();
    double getPerimeter();

    default void print() {
        System.out.println("I am a shape.");
    }
}
```

Interfaces by default have abstract methods, so you don't have to specify that the methods. Since java 8 you can have default methods in interfaces. A default method is a method that has a body. You can also have static methods in interfaces.

Every variable in an interface is automatically public, static and final, so you can't change the value of the variable. This variable's value is the same for every object in every sub class.

```
public interface Constants {
    int MAX_VALUE = 100;
    String DEFAULT_NAME = "John Doe";
}
```

You can also have nested interfaces. For example:

```
public interface Shape {
    double getArea();
    double getPerimeter();

    default void print() {
        System.out.println("I am a shape.");
    }

    interface Printable {
        void print();
    }
}
```

```
public class Circle implements Shape.Printable {
    private double radius;

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

    @Override
    public void print() {
        System.out.println("I am a circle.");
```

```
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Shape.Printable shape = new Circle(5);
        shape.print();
    }
}
```

Your class can implement multiple interfaces, and extend one thing. For example:

```
```java
public class Circle extends test implements Shape, Shape2 {
    private double radius;

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

    public double getArea() {
        return Math.PI * radius * radius;
    }

    public double getPerimeter() {
        return 2 * Math.PI * radius;
    }
}
```

---

**Deeper look into implementing interfaces**

In java you can create a variable of a interface type that points to a object of a class that implements that interface. For example:

```
public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle(5);
        shape.print();
    }
}
```

However if you create a variable of a interface type you can only use the methods that are in the interface. This is because the variable is of the interface type and not the class type.
```

```java
public interface Shape {
    double getArea();
    double getPerimeter();

    default void print() {
        System.out.println("I am a shape.");
    }
}
```

```java
public class Circle implements Shape {
    private double radius;

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

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

    @Override
    public void print() {
        System.out.println("I am a circle.");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle(5);
        shape.print();
        shape.getArea(); // This will give you an error
    }
}
```

Interfaces can also **extend** only **other interfaces, but not implement them**. For example:

```java
public interface Shape2 extends Shape {
    void print();
}
```

In this case `Shape` is an interface

Interface can also have static methods, since java 8. The static methods in interfaces can only be called by the interface name, and not the implementations of it. This shouldn't get confused with static variables

which can be called by the implementations of the interface.

```
public interface Shape {
    double getArea();
    double getPerimeter();

    default void print() {
        System.out.println("I am a shape.");
    }

    static void printStatic() {
        System.out.println("I am a static method in an interface.");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle(5);
        shape.print();
        Shape.printStatic();
    }
}
```

Since java 9 you can have private methods in interfaces. These methods can only be called by the interface itself. This is useful if you have a lot of methods that are similar and you want to reduce the amount of code you have to write.

```
public interface Shape {
    double getArea();
    double getPerimeter();

    default void print() {
        printArea();
        printPerimeter();
    }

    private void printArea() {
        System.out.println("Area: " + getArea());
    }

    private void printPerimeter() {
        System.out.println("Perimeter: " + getPerimeter());
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle(5);
        shape.print();
    }
}
```

---

### Why don't we want to instantiate a abstract class or an interface?

It cannot be instantiated directly because it may contain abstract methods that do not have an implementation in the abstract class itself. Using abstract classes also enforce a certain level of abstraction in your code, since abstract classes cannot be instantiated directly, they must be extended and instantiated by subclasses.

Preventing instantiation of abstract classes can also help in things like the usage of polymorphism, where objects are treated as instances of their concrete subclasses rather than the abstract class itself.

It also sometimes doesn't make sense to have a object of something, for example if you have a parent class called animal and the sub classes are cat and dog. It doesn't make sense to have a animal object because it doesn't have a specific implementation. Instantiating an abstract class would violate the conceptual correctness of the code since an abstract class is not meant to represent a specific object or state.

---

### Abstract vs Interface

Abstract class is good, if you have a lot of closely related classes that you want to have the same functionally and type of fields, but an interface is used when you have a lot of unrelated classes that you want to a certain thing.

- Abstract class can have abstract and non-abstract methods, they have non abstract methods (do have a body). Interface have abstract methods by default, but since java 8 you can have concrete methods in interface.
- Abstract uses extends keyword to inherit the class. Interface uses implements keyword to implement an interface. The subclasses can only extend one abstract class but can implement multiple interfaces.

---

### Why use abstract classes and interfaces

Use an interface when:

1. You want to define a contract or behavior that multiple unrelated classes should implement.
2. You need to achieve multiple inheritances, as Java does not support inheriting from multiple classes.
3. You want to provide a common interface for different objects to be treated interchangeably.

For example, consider a scenario where you are developing a game and want to define a common behavior for various game characters. You can create an interface called `Character`:

```
public interface Character {
    void move();
    void attack();
    void die();
}
```

Different classes representing different characters, such as `Warrior`, `Wizard`, and `Monster`, can then implement the `Character` interface and provide their own implementations for the methods.

Use an abstract class when:

1. You want to provide a partial implementation or default behavior that can be inherited by subclasses.
2. You need to define shared attributes or fields that can be used by subclasses and changed.
3. You want to enforce a structure or pattern among related classes.

For example, consider a scenario where you are building a shape hierarchy. You can create an abstract class called `Shape` that provides a default implementation for calculating the area and declares an abstract method for calculating the perimeter:

```
public abstract class Shape {
    public abstract double getPerimeter();

    public double getArea() {
        return 0; // Default implementation for area, to be overridden by
subclasses
    }
}
```

Subclasses like `Circle`, `Rectangle`, and `Triangle` can extend the `Shape` abstract class and provide their specific implementations for calculating the perimeter and area.