OOPS Concept;

Data Abstraction and Abstract Class:

- Abstraction is a design concept on which we only declare the Functionality but we do not define it because we don't know about them at design point.
- **Abstract Keyword:**

This is a special keyword which is used as a non- access modifier with classes and Methods.

- **Abstract Keyword with Class:**

If an abstract keyword is used with the class, then no one can instantiate(no one can create an object of that class) that class and these classes are known as Abstract Class.

When the `abstract` keyword is used with a class in Java, it signifies that the class is an abstract class. An abstract class is a class that cannot be instantiated directly, meaning you cannot create objects (instances) of that class using the `new` keyword. Abstract classes are designed to be extended by other classes, and they often serve as a blueprint for other classes to inherit from.

Key points about abstract classes:

> **Cannot be Instantiated: Abstract classes cannot be instantiated on their own. You cannot create objects of an abstract class using the `new` keyword.**
> **May Contain Abstract Methods: Abstract classes can have abstract methods. Abstract methods are methods declared in the abstract class without providing an implementation. Subclasses that extend the abstract class must provide concrete implementations for these abstract methods.**

Here's an example:

```
// Abstract class with an abstract method
abstract class Shape {
    // Abstract method (no implementation)
    abstract void draw();
```

```
    // Concrete method
    void display() {
        System.out.println("Displaying shape");
    }
}
// Concrete subclass extending the abstract class
class Circle extends Shape {
    // Providing concrete implementation for the abstract method
    @Override
    void draw() {
        System.out.println("Drawing a circle");
    }
}
// Concrete subclass extending the abstract class
class Rectangle extends Shape {
    // Providing concrete implementation for the abstract method
    @Override
    void draw() {
        System.out.println("Drawing a rectangle");
    }
}
public class Main {
    public static void main(String[] args) {
        // Cannot instantiate the abstract class directly
        // Shape shape = new Shape();  // Compilation error
        // Can create objects of concrete subclasses
        Circle circle = new Circle();
        circle.draw();    // Calls the overridden draw method in Circle
        circle.display(); // Calls the inherited display method from Shape
        Rectangle rectangle = new Rectangle();
        rectangle.draw();    // Calls the overridden draw method in Rectangle
        rectangle.display(); // Calls the inherited display method from Shape
    }
}
```

In this example, `Shape` is an abstract class with an abstract method `draw()`. It cannot be instantiated directly, but concrete subclasses like `Circle` and `Rectangle` can be created, and they must provide concrete implementations for the abstract method `draw()`.

- **Abstract Keyword with Method:**

If an abstract keyword is used with the Methods, then it must be overriden in the first concrete class.

May Contain Abstract Methods: Abstract classes can have abstract methods. Abstract methods are methods declared in the abstract class without providing an implementation. Subclasses that extend the abstract class must provide concrete implementations for these abstract methods.

Abstract class vs Concrete Class:

- **The classes which can not be instantiated(Will not be able to create an object of that class) are known as Abstract classes.**
- **Concrete classes are those classes which can be instantiated(able to create an object of that class).**

>> The first concrete class is the very first child class of that Abstract Class.

- **Abstract Class**: A class declared with the `abstract` keyword in Java. An abstract class may contain abstract methods (methods without a body) and concrete methods (methods with an implementation). An abstract class cannot be instantiated directly; it serves as a blueprint for other classes.
- Concrete Class: A class that is not declared as abstract and provides concrete implementations for all the abstract methods inherited from its abstract superclass.

Now, when we say "The first concrete class is the very first child class of that Abstract Class," it means that among all the subclasses that extend a particular abstract class, the first one that provides concrete implementations for all abstract methods is considered the first concrete class.

```
//Abstract class with an abstract method
abstract class Animal {
 abstract void makeSound();  // Abstract method
}
//The first concrete subclass of Animal
class Dog extends Animal {
 // Providing a concrete implementation for the abstract method
 @Override
 void makeSound() {
    System.out.println("Dog barks");
 }
 // Additional methods for Dog
 void fetch() {
    System.out.println("Dog fetches the ball");
 }
}
//Another concrete subclass of Animal
class Cat extends Animal {
```

```
    // Providing a concrete implementation for the abstract method

    @Override
    void makeSound() {
        System.out.println("Cat meows");
    }
    // Additional methods for Cat
    void scratch() {
        System.out.println("Cat scratches");
    }
}
public class Main {
    public static void main(String[] args) {
        // Animal animal = new Animal();  // Compilation error, cannot
instantiate abstract class
        // Creating objects of concrete subclasses
        Dog dog = new Dog();
        dog.makeSound();  // Calls the overridden makeSound method in
Dog
        dog.fetch();      // Calls the method specific to Dog
        Cat cat = new Cat();
        cat.makeSound();  // Calls the overridden makeSound method in Cat
        cat.scratch();    // Calls the method specific to Cat
    }
}
```

Abstract methods in Java are methods that are declared without an implementation in an abstract class. The implementation of these methods is deferred to the subclasses that extend the abstract class. Abstract methods play a crucial role in creating a blueprint for a class hierarchy in real-life projects, such as an e-commerce system.

Let's consider an example of an abstract class `Product` in an e-commerce application. The `Product` class might have common properties for all products, but

the specifics of how each product is represented can vary. Therefore, we can define abstract methods for these varying behaviours.

```
// Abstract class representing a product in an e-commerce system
public abstract class Product {
    private String name;
    private double price;
    // Constructor
    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }
    // Abstract method to get the product type
    public abstract String getProductType();
    // Abstract method to calculate the discounted price
    public abstract double calculateDiscount();
    // Getter methods
    public String getName() {
        return name;
    }
    public double getPrice() {
        return price;
    }
}
```

In this example:

- The `Product` class is declared as abstract.
- It contains abstract methods `getProductType()` and `calculateDiscount()`.
- The `getProductType()` method is meant to be implemented by subclasses to provide the specific type of product (e.g., Electronics, Clothing).
- The `calculateDiscount()` method is meant to be implemented by subclasses to calculate the discounted price based on specific rules for each product type.

Now, let's create concrete subclasses that extend the `Product` class and provide implementations for the abstract methods:

```
//Concrete subclass representing an Electronics product
public class ElectronicsProduct extends Product {
 private int warrantyPeriod;
 // Constructor
 public ElectronicsProduct(String name, double price, int warrantyPeriod) {
    super(name, price);
    this.warrantyPeriod = warrantyPeriod;
 }
 // Implementation of the abstract method to get the product type
 @Override
 public String getProductType() {
    return "Electronics";
 }
 // Implementation of the abstract method to calculate the discounted price
 @Override
 public double calculateDiscount() {
    // Specific logic for calculating discount for electronics products
    return getPrice() * 0.1; // 10% discount
 }
 // Additional methods specific to ElectronicsProduct
 public int getWarrantyPeriod() {
    return warrantyPeriod;
 }
}
```

n Java, the `super` keyword is used to invoke the constructor of the parent class. In the context of the `ClothingProduct` class, `super(name, price);` is calling the constructor of the parent class `Product`. This is necessary because the `Product` class is an abstract class, and abstract classes cannot be instantiated directly. When you create an instance of a concrete subclass (like `ClothingProduct`), you need to ensure that the constructor of the abstract parent class is called to initialize its attributes.

Let's break down the constructor in `ClothingProduct`:

```
public ClothingProduct(String name, double price, String size) {
    super(name, price);
    this.size = size;
}
```

> **`super(name, price);`: This line is invoking the constructor of the `Product` class (the abstract parent class) with the provided `name` and `price` parameters. It ensures that the common attributes of the product (name and price) are initialized by calling the constructor of the abstract class.**
> **`this.size = size;`: This line initializes the specific attribute `size` of the `ClothingProduct` class.**

By using `super(name, price);`, you are leveraging the constructor of the abstract class to handle the initialization of common attributes, following the principles of inheritance. This helps in code reuse and ensures that the initialization logic in the abstract class is executed.

In summary, the `super` keyword is essential in the constructor of a subclass to ensure proper initialization of attributes inherited from the abstract parent class.

Note 👍

The use of `super(name, price);` in the constructor of a subclass is specifically designed to invoke the constructor of the parent class (in this case, the abstract class `Product`). The `this` keyword, on the other hand, is used to refer to the current instance of the class.

In the context of a constructor, `super` is used to call the constructor of the immediate parent class, while `this` is used to refer to the instance variables or methods of the current class.

If you attempt to use `this(name, price);` instead of `super(name, price);`, it would result in a compilation error because `this` can only be used to call other constructors within the same class, not in the context of calling the constructor of the parent class.

So, to properly initialize the attributes of the parent class (`Product`), you must use `super(name, price);` in the constructor of the subclass (`ClothingProduct`).

 # Abstract Method and its Properties:

An abstract method is a method declared in an abstract class or interface that does not have an implementation (body) in the class or interface where it is declared. It is meant to be overridden by concrete (non-abstract) subclasses or implemented by classes that implement the interface.

Here are the key points about abstract methods:

1. **No Implementation in the Abstract Class/Interface:** Abstract methods are declared without providing any implementation in the abstract class or interface.
2. **Must be Overridden or Implemented:** Any concrete subclass that extends an abstract class or any class that implements an interface containing abstract methods must provide a concrete implementation for those abstract methods.
3. **Purpose of Enforcing Implementation:** The purpose of abstract methods is to enforce that concrete subclasses provide specific behaviour. It allows the abstract class or interface to declare a method signature that must be present in all its concrete subclasses.

```
abstract class AbstractClass {
    // Abstract method with no implementation
    abstract void abstractMethod();
}
// Concrete subclass extending the abstract class
class ConcreteClass extends AbstractClass {
    // Providing an implementation for the abstract method
    @Override
    void abstractMethod() {
        System.out.println("Implementation of abstractMethod in ConcreteClass.");
    }
}
public class Main {
    public static void main(String[] args) {
        ConcreteClass concreteObject = new ConcreteClass();
        concreteObject.abstractMethod(); // Calls the implementation in ConcreteClass
    }
}
```

Question : How we can call concrete Method from an Abstract class:

.In Java, you can call a concrete method from an abstract class in a few different ways:

1. Inside Abstract Class: If the concrete method is defined in the same abstract class, you can directly call it using **this** keyword.

```
abstract class AbstractClass {
    void concreteMethod() {
        System.out.println("Concrete method in AbstractClass.");
    }
    abstract void abstractMethod();
    void callConcreteMethod() {
        // Calling concrete method from within the abstract class
        this.concreteMethod();
    }
}
```

In this example, `callConcreteMethod` within the abstract class directly calls the `concreteMethod` using `this`.

2.Using Super in Subclass: If the concrete method is defined in the abstract class and you are in a subclass, you can use the `super` keyword to call the concrete method.

```
abstract class AbstractClass {
    void concreteMethod() {
        System.out.println("Concrete method in AbstractClass.");
    }
    abstract void abstractMethod();
}
class SubClass extends AbstractClass {
    @Override
    void abstractMethod() {
        // Calling concrete method from within a subclass using super
        super.concreteMethod();
```

```
}
```

In this example, `SubClass` overrides the abstract method `abstractMethod` and calls the concrete method using `super`.

Question : **is java Allowing to create an Object of Abstract class?**

Ans:

No, you cannot create an object of an abstract class in Java. Abstract classes are meant to be subclassed, and they can have abstract methods without providing a complete implementation. Since abstract classes may not have a complete implementation, creating an object of an abstract class wouldn't make sense.

Here's an example to illustrate this:

```
abstract class AbstractClass {
    abstract void abstractMethod();
}
public class Main {
    public static void main(String[] args) {
        // Compilation error: Cannot instantiate the type AbstractClass
        AbstractClass abstractObject = new AbstractClass();
    }
}
```

In the example above, attempting to create an instance of `AbstractClass` (`new AbstractClass()`) results in a compilation error. Abstract classes can only be instantiated through their concrete subclasses, where the abstract methods are implemented.

If you want to use the functionality of an abstract class, you would need to create a concrete subclass that extends the abstract class, provide implementations for all abstract methods, and then instantiate the subclass.

```java
class ConcreteClass extends AbstractClass {
    @Override
    void abstractMethod() {
        // Provide implementation for the abstract method
        System.out.println("Implementation of abstractMethod in ConcreteClass.");
    }
}
public class Main {
    public static void main(String[] args) {
        // Instantiate the concrete subclass
        ConcreteClass concreteObject = new ConcreteClass();
        // Call the abstract method through the concrete object
        concreteObject.abstractMethod();
    }
}
```

 # InterFace And its Declaration 👍 : Interface In Java OOPS Concept:

1. **Interface is nothing but a pure Abstract Class , which contains only Abstract Method. We can not create concrete Method inside the Interface , if we do not write abstract inside the Interface method by default it will be considered as Abstract.**
2. **We can not create an object of interface .**
3. **Main difference between abstract and Interface is in Abstract class we can create abstract as well as concrete methods but it is not possible in Interface.**

but with the introduction of Java 8 and the concept of default methods in interfaces, it's now possible to have concrete methods in interfaces as well. Let's clarify the differences:

1. Abstract Class:

- In an abstract class, you can indeed have both abstract methods (methods without a body) and concrete methods (methods with a body).
- Abstract classes can have instance variables (fields), constructors, and methods with a body.
- You can use the `abstract` **keyword for abstract methods, but it's optional for methods without a body (concrete methods).**

```
abstract class AbstractClass {
   // Abstract method
   abstract void abstractMethod();
   // Concrete method
   void concreteMethod() {
      System.out.println("Concrete method in AbstractClass");
   }
}
```

2. **Interface:**

- Traditionally, interfaces could only have abstract methods, and all methods were implicitly public and abstract.
- With Java 8, interfaces can have default methods (methods with a default implementation) and static methods. These are concrete methods.
- Interfaces can declare constants (public static final fields).

```
interface MyInterface {
   // Abstract method (implicitly public and abstract)
   void abstractMethod();
   // Default method (concrete method with default implementation)
   default void defaultMethod() {
      System.out.println("Default method in MyInterface");
   }
   // Static method (concrete method)
   static void staticMethod() {
```

```
        System.out.println("Static method in MyInterface");
    }
}
```

So, while the traditional distinction was that abstract classes could have both abstract and concrete methods, and interfaces could only have abstract methods, this distinction has blurred with the introduction of default methods in interfaces. Now, both abstract classes and interfaces can have a mix of abstract and concrete methods. The choice between them depends on the specific requirements of your design.

Question : Can we create an abstract and Interface as final ?

Ans: Ideally No.

In Java, the `final` keyword is used to indicate that a class or method cannot be further extended or overridden. However, its application to abstract classes and interfaces has some differences:

1. **Abstract Classes:**

You can declare an abstract class as `final`, **but there is a restriction. If a class is marked as `abstract`, it is expected to be subclassed, providing concrete implementations for its abstract methods. Marking it as `final` contradicts the purpose of being abstract.**

The combination of `abstract` and `final` is not allowed in the same class declaration.

```
//This is not allowed
final abstract class MyAbstractClass {
 // ...
}
```

If a class is marked as `abstract`, it's implicitly meant for extension. If you want to prevent further subclassing, you typically don't mark it as `abstract`.

2. Interfaces:

- Interfaces in Java are implicitly **`abstract`.** **You can declare**
- **an interface as `final`, but it also has some implications.**
- **If an interface is marked as `final`, it means that other interfaces cannot extend it, and classes cannot implement it. This is allowed but somewhat unusual, as interfaces are generally designed to be extended or implemented.**

```
//Allowed but unusual
final interface MyInterface {
 // ...
}
```

In summary, while it's technically possible to use the `final` keyword with abstract classes and interfaces, doing so may go against their intended use cases. Abstract classes are meant to be extended, and interfaces are designed to be implemented or extended. The decision to mark them as `final` should be made with a clear understanding of the design implications. In most cases, it's not a common practice to use `final` with abstract classes or interfaces.

Note : In abstract class it is mandatory for child class to oveeride the abstract Methods.

Note: any class is implementing an Interface it is actually providing the definition to all methods which are inside it.

Note: If class is Inheriting to the interface we are using the keyword implements.

If there any interface is inheriting another interface we are using the keywords extends

Note:

Class Implementing an Interface:

- When a class implements an interface in Java, the `implements` **keyword is used.**
- **This indicates that the class is providing concrete implementations for the abstract methods declared in the interface.**

```
interface MyInterface {
    void myMethod();
}
class MyClass implements MyInterface {
    @Override
    public void myMethod() {
        // Concrete implementation
    }
}
```

2. Interface Extending Another Interface:

- When one interface extends another interface, the `extends` **keyword is used.**
- **This indicates that the sub-interface inherits the abstract methods (and constants) from the parent interface.**

```
interface ParentInterface {
    void parentMethod();
}
interface ChildInterface extends ParentInterface {
    void childMethod();
}
```

In this example, `ChildInterface` **extends** `ParentInterface`**, meaning that** `ChildInterface` **includes the methods from both interfaces (**`parentMethod` **from** `ParentInterface` **and** `childMethod` **from** `ChildInterface`**).**

So, to summarise:

- `implements` **is used when a class is implementing an interface.**
- `extends` **is used when an interface is extending another interface.**

Multiple Inheritance in java we can Achieve using Interface;

```
//Class implementing both interfaces, achieving multiple inheritance
class MultipleInheritanceClass implements Functionality1, Functionality2 {
 @Override
 public void method1() {
    System.out.println("Method 1 from Functionality1");
 }
 @Override
 public void method2() {
    System.out.println("Method 2 from Functionality1");
 }
 @Override
 public void method3() {
    System.out.println("Method 3 from Functionality2");
 }
 @Override
 public void method4() {
    System.out.println("Method 4 from Functionality2");
 }
}
public class MultipleInheritanceExample {
 public static void main(String[] args) {
    // Creating an instance of the class that implements multiple interfaces
    MultipleInheritanceClass instance = new MultipleInheritanceClass();
    // Calling methods from both interfaces
    instance.method1();
    instance.method2();
    instance.method3();
    instance.method4();
 }
}
```

In this example, `MultipleInheritanceClass` implements both `Functionality1` and `Functionality2` interfaces. It inherits the methods `method1` and `method2` from `Functionality1` and `method3` and `method4` from `Functionality2`. The `main` method demonstrates calling methods from both interfaces through an instance of `MultipleInheritanceClass`.