Interfaces

```
abstract class Test1
{
    abstract public void meth1();
    abstract public void meth2();
}

class Test2 extends Test1
{
    public void meth1()
    {
    =
    }
    public void meth2()
    {
    =
    }
}
```

```
Test1 t=new Test2();
```

```
interface Test1
{
    void meth1();
    void meth2();
}

class Test2 implements Test1
{
    public void meth1()
    {
    =
    }
    public void meth2()
    {
    =
    }
}
```

--> An interface is a fully abstract class which includes a group of abstract methods (i.e. methods without a body).

--> Interfernce ek trah sai abstract class hi hai bas ise mai "abstract class" likhna ki jga ham "interface" likhta hai. Also jab "inheritance" karna hota hai to "extend" - keyword ka place par "implements" - keyword ka use karta hai.

--> Jis trah "abstract - class" ka object create nahi hota ha waisa hi "interface" ka bhi nahi hota hai. Bas ham uska reference create kar sakta hai.

--> Aur, interfaces super class kai reference ka use kar ka subclass ka help object create ho sakta hai.

explain interface class with context of java.

Interface in Java

An **interface** in Java is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. It is a way to achieve abstraction and multiple inheritance in Java. Interfaces are used to specify a set of methods that a class must implement, providing a way to enforce certain behaviors across different classes.

Key Features of Interfaces

1. **Abstract Methods**:
   - Interfaces primarily define abstract methods, which are methods without a body. Any class that implements the interface must provide concrete implementations for all of its abstract methods.
   - In Java 8 and later, interfaces can also include **default methods** with an implementation and **static methods**.

   Example:

```java
interface Animal {
    // Abstract method (no body)
    void sound();

    // Default method (with body)
    default void sleep() {
        System.out.println("Sleeping...");
    }

    // Static method (with body)
    static void eat() {
        System.out.println("Eating...");
    }
}
```

2. **Implementation**:
   - A class implements an interface using the `implements` keyword and must provide implementations for all abstract methods defined in the interface.

   Example:

```java
class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}
```

3. **Multiple Inheritance**:

- Java supports multiple inheritance through interfaces. A class can implement multiple interfaces, allowing it to inherit behaviors from more than one source.

Example:

```java
interface CanFly {
    void fly();
}

interface CanSwim {
    void swim();
}

class Duck implements CanFly, CanSwim {
    @Override
    public void fly() {
        System.out.println("Duck flies");
    }

    @Override
    public void swim() {
        System.out.println("Duck swims");
    }
}
```

4. **No Constructor**:
   - Interfaces cannot have constructors. They are not meant to be instantiated; instead, they are implemented by classes.
5. **Fields**:
   - Fields in an interface are implicitly `**public**`, `**static**`, and `**final**`. They must be initialized when declared.

Example:

```java
interface Constants {
    int MAX_VALUE = 100;  // public, static, and final by default
}
```

6. **Inheritance of Interfaces**:
   - Interfaces can extend other interfaces. A subclass of an interface inherits all the abstract methods of the parent interface.

Example:

```java
interface Animal {
    void sound();
}

interface Pet extends Animal {
    void play();
}
```

Example of Using Interfaces

java

```java
// Define an interface
interface Animal {
    void sound();  // Abstract method
    default void sleep() {  // Default method
        System.out.println("Sleeping...");
    }
}

// Implement the interface
class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

// Implement another interface
class Bird implements Animal {
    @Override
    public void sound() {
        System.out.println("Bird chirps");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        myDog.sound();  // Outputs: Dog barks
        myDog.sleep();  // Outputs: Sleeping...

        Animal myBird = new Bird();
        myBird.sound();  // Outputs: Bird chirps
    }
}
```

In this example, the `Animal` interface defines an abstract method `sound()` and a default method `sleep()`. The `Dog` and `Bird` classes implement the `Animal` interface and provide specific implementations for the `sound()` method. Both classes can use the default `sleep()` method provided by the interface.

When to Use Interfaces

- **To Define a Contract**: Use interfaces to specify a set of methods that implementing classes must provide, ensuring a common behavior or capability across different classes.
- **To Achieve Multiple Inheritance**: Use interfaces when you need to implement multiple inheritance of behavior, as Java does not support multiple inheritance with classes.
- **To Define Common Methods Across Classes**: Use interfaces to provide a common interface for different classes that need to share a set of methods.

Differences Between Abstract Classes and Interfaces

Feature	Abstract Class	Interface
Methods	Can have abstract and concrete methods	Can have abstract, default, and static methods
Constructors	Can have constructors	Cannot have constructors
Fields	Can have instance fields and static fields	Can only have static final fields (constants)

Feature	Abstract Class	Interface
Multiple Inheritance	A class can extend only one abstract class	A class can implement multiple interfaces
Access Modifiers	Methods can have any access modifier	Methods are implicitly public and abstract unless they are default or static
Use Case	Use when you have a common base class with some default behavior	Use when you want to define a contract for different classes

Summary

- **Interfaces** in Java define a contract that implementing classes must adhere to.
- They allow for the definition of abstract methods that classes must implement.
- Java interfaces support **multiple inheritance**, allowing a class to implement multiple interfaces.
- Interfaces can have default and static methods (from Java 8 onwards), enabling some level of method implementation directly within the interface.
- They provide a way to achieve abstraction and define common behavior across unrelated classes.

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Abstract vs interface

Abstract class

- If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
- Every method present inside abstract class need not be public and abstract.
- There are no restrictions on abstract class method modifiers.
- Every abstract class variable need not be a public static final.
- Inside the abstract class we can take constructor.

interface

- If we don't' know anything about implementation just we have requirement specification then we should go for an interface.
- Every method present inside the interface is always public and abstract whether we are declaring or not.
- We can't declare interface methods with the modifiers private, protected, final, static, synchronized, native, strictfp.
- Every interface variable is always a public static final whether we are declaring or not following modifiers. private, protected, transient, volatile.
- Inside the interface we can't take constructor.

Main.java 2, U Java8Fe.java U

Java8Fe.java > A > config()

```java
interface A
{
    void show();
    default void config()
    {

    }
    static void abc()
    {
        System.out.println("in abc");
    }
}

class B implements A
{
    public void show()
    {
        System.out.println("in show");
    }
}

public class Java8Fe {
    Run | Debug
    public static void main(String[] args) {
        A.abc();
        A obj = new B();
        obj.show();
        obj.config();
    }
}
```

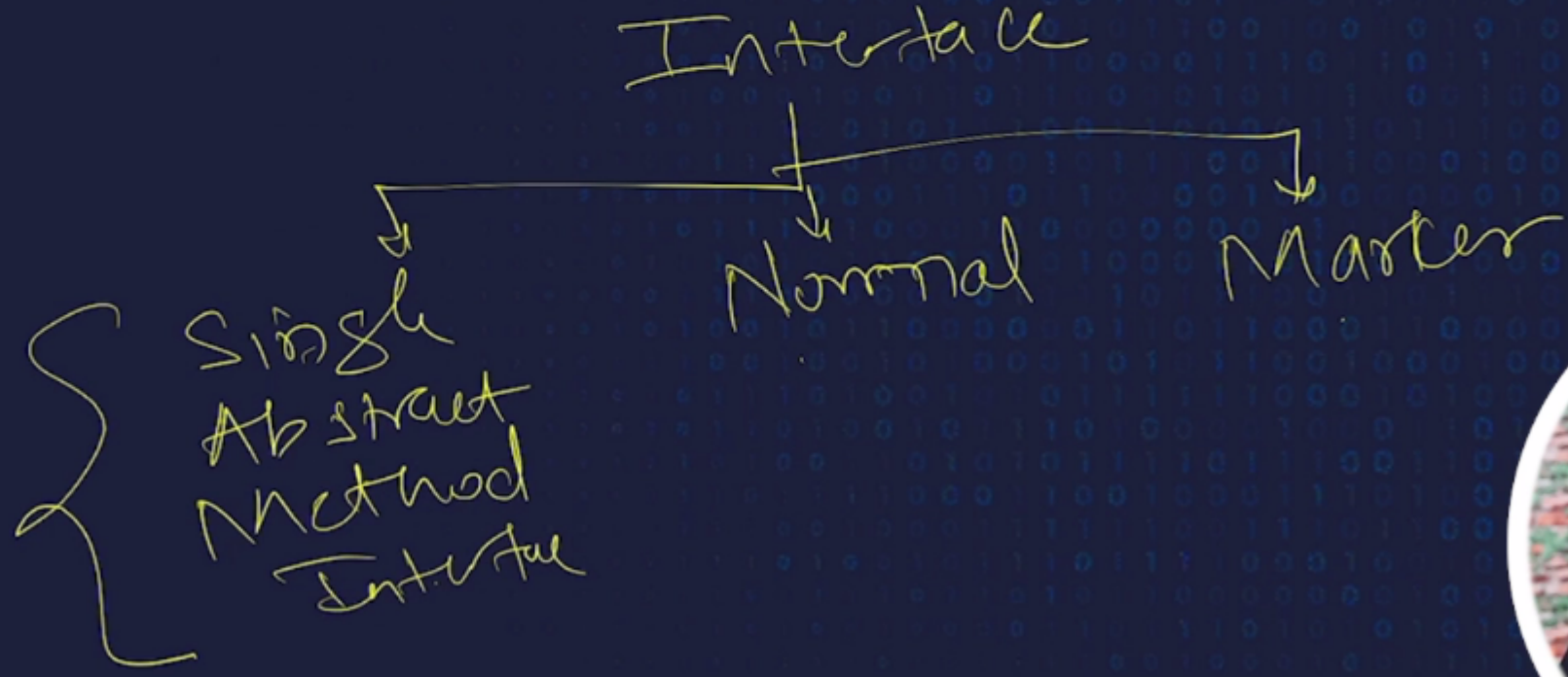
--> Above version of java 8, ham interface mai "method" ko define kar sakta hai but sirf hma uska aaga "default - keyboard" ka use karna hota hai.

--> Also, interface ka andar ham static method bhi define kar sakta hai.

--> Static mth. ko ham direct access kar sakta hai but non static method ko use karna ka liya hma uska class ka object create karna hota, then uska use karta hai.

JAVA 8 Features over Interfaces:

1. Default Methods in Interfaces
2. Static Methods in Interfaces
3. Functional Interfaces



Functional Interface

- If any Java interface allows only one abstract method then it is called a "Functional Interface".

- To make any interface as Functional Interface then we have to use the following annotation just above of the interface. @FunctionalInterface
  Ex: java.lang.Runnable
  java.lang.Comparable

Java

what does mean by functional inteface

Functional Interface in Java

A **functional interface** in Java is an interface that has exactly one abstract method. These interfaces can contain multiple default or static methods, but they must have only one abstract method. Functional interfaces are used primarily to represent a single abstract method (SAM) type, making them a core component in Java's support for functional programming, particularly with the introduction of lambda expressions in Java 8.

Key Characteristics of a Functional Interface

1. **Single Abstract Method**:
   - A functional interface must have exactly one abstract method. This method defines the functional behavior of the interface.

   Example:

```java
@FunctionalInterface
interface MyFunction {
    void apply(int a);  // Single abstract method
}
```

2. **Default and Static Methods**:
   - A functional interface can have multiple default methods and static methods. These methods provide additional functionality without affecting the functional nature of the interface.

   Example:

```java
@FunctionalInterface
interface MyFunction {
    void apply(int a);  // Single abstract method

    default void printMessage() {
        System.out.println("Hello from default method");
    }

    static void printStaticMessage() {
        System.out.println("Hello from static method");
```

```
    }
}
```

3. **@FunctionalInterface Annotation**:
   - The `@FunctionalInterface` annotation is optional but recommended. It helps to ensure that the interface adheres to the functional interface contract by generating a compile-time error if the interface has more than one abstract method.

   Example:

```java
@FunctionalInterface
interface MyFunction {
    void apply(int a);  // Single abstract method

    // @FunctionalInterface annotation ensures only one abstract method
}
```

4. **Lambda Expressions**:
   - Functional interfaces are commonly used with lambda expressions, which provide a concise way to implement the abstract method of the interface.

   Example:

```java
@FunctionalInterface
interface MyFunction {
    void apply(int a);
}

public class Main {
    public static void main(String[] args) {
        // Lambda expression implementing MyFunction
        MyFunction myFunction = (a) -> System.out.println("Value: " + a);
        myFunction.apply(10);  // Outputs: Value: 10
    }
}
```

Common Functional Interfaces in Java

Java provides several built-in functional interfaces in the `java.util.function` package. Some of the commonly used ones include:

1. `Function<T, R>`:
   - Represents a function that takes an argument of type `T` and returns a result of type `R`.

```java
Function<Integer, String> intToString = (i) -> Integer.toString(i);
```

2. `Predicate<T>`:
   - Represents a predicate (boolean-valued function) of one argument.

```java
Predicate<String> isNotEmpty = (s) -> !s.isEmpty();
```

3. `Consumer<T>`:
   - Represents an operation that takes a single input argument and returns no result.

```java
Consumer<String> print = (s) -> System.out.println(s);
```

4. `Supplier<T>`:
   - Represents a supplier of results. It takes no arguments and returns a result of type `T`.

```java
Supplier<Double> randomSupplier = () -> Math.random();
```

5. `UnaryOperator<T>`:
   - Represents an operation on a single operand that produces a result of the same type as its operand.

```java
UnaryOperator<Integer> square = (x) -> x * x;
```

6. `BinaryOperator<T>`:
   - Represents an operation on two operands of the same type that produces a result of the same type.

```java
BinaryOperator<Integer> add = (a, b) -> a + b;
```

Use Cases of Functional Interfaces

- **Lambda Expressions**: Functional interfaces are the target types for lambda expressions, which provide a concise syntax for implementing the abstract method.
- **Stream API**: Functional interfaces are used extensively in Java's Stream API for operations like filtering, mapping, and reducing.
- **Event Handling**: They are used in event-driven programming for handling events with concise code.

Summary

- A **functional interface** has exactly one abstract method.

- It can have multiple default and static methods.
- The `@FunctionalInterface` annotation is optional but helps to enforce the functional interface contract.
- Functional interfaces are used with lambda expressions to provide a clear and concise way to represent single-method interfaces.
- Java provides several built-in functional interfaces in the `java.util.function` package for various common programming tasks.

Functional interfaces are fundamental to functional programming in Java, allowing for more expressive and maintainable code through lambda expressions and method references.

```java
package interfacepractice;

interface Test
{
    final static int X=10;
    public abstract void meth1();
    public abstract void meth2();

    private void meth3()
    {
        System.out.println("Meth3 of Test");
    }
    default void meth5()
    {
        meth3();
    }

}

interface Test2 extends Test
{
    void meth4();
}

class My implements Test2
{
    public void meth1(){}
    public void meth2(){}
    public void meth4(){}
}
public class InterfacePractice
{
    public static void main(String[] args)
    {
        System.out.println(Test.X);
        My m=new My();
        m.meth3();

    }
}
```

Do or Don't for interfaces

--> By default, methods are Public and Abstract.

--> As methods are to be implemented by the classes, they can't be made private.

--> Identifiers can be used in interfaces but the Identifiers must be given in Uppercases.

--> Identifiers are by default final and static.

--> Method inside an interface cannot have body but the method can have body if the method is static.

--> Static members can be accessed in main method by using interface name and dot operator.

--> An interface can be extended from another interfaces.