## Interface in Java

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction.

There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

In an interface, you can’t instantiate variables and create an object.

The interface cannot contain concrete (with implementation) methods

In Interface only one specifier is used- Public.

Prior to JDK 8, the interface could not define the implementation. We can now add default implementation for interface methods. This default implementation has a special use and does not affect the intention behind interfaces.

## Important Points about Interface or Summary of the Article:

* We can’t create an instance (interface can’t be instantiated) of the interface but we can make the reference of it that refers to the Object of its implementing class.
* A class can implement more than one interface.
* An interface can extend to another interface or interface (more than one interface).
* A class that implements the interface must implement all the methods in the interface.
* All the methods are public and abstract. And all the fields are public, static, and final.
* It is used to achieve multiple inheritances.
* It is used to achieve loose coupling.

## New Features Added in Interfaces in JDK 9

From Java 9 onwards, interfaces can contain the following also:

1. Static methods
2. Private methods
3. Private Static methods

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

The default methods were introduced to provide backward compatibility so that existing interfaces can use the lambda expressions without implementing the methods in the implementation class. Default methods are also known as **defender methods**or **virtual extension methods**.

**Important Points:**

1. Interfaces can have default methods with implementation in Java 8 on later.
2. Interfaces can have static methods as well, similar to static methods in classes.
3. Default methods were introduced to provide backward compatibility for old interfaces so that they can have new methods without affecting existing code.

Syntax

Public interface <interface Name> {

Public default <return type> <method name> {

Logic code

}

}

Static Method in Java

The static keyword is used to create methods that will exist independently of any instances created for the class.

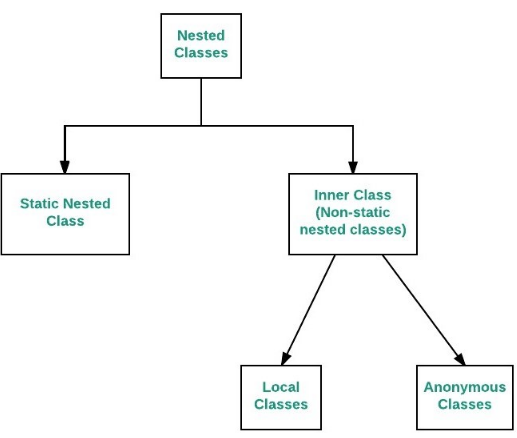
Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.

 Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

Nested Class in Java

In Java, it is possible to define a class within another class, such classes are known as nested classes. They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation, and creates more readable and maintainable code.

* The scope of a nested class is bounded by the scope of its enclosing class. Thus in below example, class Nested Class does not exist independently of class Outer Class.
* A nested class has access to the members, including private members, of the class in which it is nested. However, the reverse is not true i.e., the enclosing class does not have access to the members of the nested class.
* A nested class is also a member of its enclosing class.
* As a member of its enclosing class, a nested class can be declared private, public, protected, or package private (default).
* Nested classes are divided into two categories:
  1. **static nested class :** Nested classes that are declared static are called static nested classes.
  2. **inner class:**An inner class is a non-static nested class.



# Anonymous Inner Class in Java

Nested Classes in Java is prerequisite required before adhering forward to grasp about anonymous Inner class. It is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain “extras” such as overriding methods of a class or interface, without having to actually subclass a class.

The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

**Syntax:**

// Test can be interface, abstract/concrete class

Test t = new Test ()

{

// data members and methods

Public void test\_method ()

{

........

........

}

};

Now let us do discuss the difference between regular class(normal classes) and Anonymous Inner class

* A normal class can implement any number of interfaces but the anonymous inner class can implement only one interface at a time.
* A regular class can extend a class and implement any number of interfaces simultaneously. But anonymous Inner class can extend a class or can implement an interface but not both at a time.
* For regular/normal class, we can write any number of constructors but we can’t write any constructor for anonymous Inner class because the anonymous class does not have any name and while defining constructor class name and constructor name must be same.

**Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class**

Like local classes, anonymous classes can capture variables; they have the same access to local variables of the enclosing scope:

* An anonymous class has access to the members of its enclosing class.
* An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.
* Like a nested class, a declaration of a type (such as a variable) in anonymous class shadows any other declarations in the enclosing scope that have the same name.

Anonymous classes also have the same restrictions as local classes with respect to their members:

* We cannot declare static initializers or member interfaces in an anonymous class.
* An anonymous class can have static members provided that they are constant variables.

**Note:**We can declare the following in anonymous classes as follows:

* Fields
* Extra methods (even if they do not implement any methods of the supertype)
* Instance initializers
* Local classes

**Ways:**

Anonymous inner classes are generic created via below listed two ways as follows:

1. Class (may be abstract or concrete)
2. Interface

Functional Interface

Java has forever remained an Object-Oriented Programming language. By object-oriented programming language, we can declare that everything present in the Java programming language rotates throughout the Objects, except for some of the primitive data types and primitive methods for integrity and simplicity. There are no solely functions present in a programming language called Java. Functions in the Java programming language are part of a class, and if someone wants to use them, they have to use the class or object of the class to call any function.

A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. **Runnable**,

Functional Interface is additionally recognized as **Single Abstract Method Interfaces**. In short, they are also known as **SAM interfaces**. Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.

Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward. Functional interfaces are interfaces that ensure that they include precisely only one abstract method. Functional interfaces are used and executed by representing the interface with an **annotation called @FunctionalInterface**. As described earlier, functional interfaces can contain only one abstract method. However, they can include any quantity of default and static methods.

In Functional interfaces, there is no need to use the abstract keyword as it is optional to use the abstract keyword because, by default, the method defined inside the interface is abstract only. We can also call Lambda expressions as the instance of functional interface.

Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

Method Reference

[Functional Interfaces in Java](https://www.geeksforgeeks.org/functional-interfaces-java/) and [Lambda Function](https://www.geeksforgeeks.org/lambda-expressions-java-8/) are prerequisites required in order to grasp grip over method references in Java. As we all know that a [method](https://www.geeksforgeeks.org/methods-in-java/) is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping the code. In this article, we will see how to use methods as value. In Java 8 we can use the method as if they were objects or primitive values, and we can treat them as a variable. The example shows the function as a variable in java: