Interface: Interface in Java is similar to abstract class, but it isn't a class, it is a reference type. It only contains constants (no instance variables), method signatures, default methods and static methods. It is more like a contract between the class and the client code that the compiler enforces. By declaring its using an interface, your class must implement all the abstract methods on the interface. However, each class is free to determine the details of its own implementation.

```
// Interface definition
interface Vehicle {
    void start();
    void stop();
}
```

By providing interface keyword, Java allows you to fully utilize one interface, multiple methods aspect of polymorphism.

```
public interface FlightEnabled {}
```

Interface Declaration: We declare an interface using interface keyword instead of using class keyword.

- Usually many interfaces will end in 'able' like Comparable, Iterable, Flight enabled. They are named according to the set of behavior's it describe.
- While defining an interface, if no access modifier is included, then default access is given to the interface. That means it is only available to other members of the package in which it is declared. Whereas if it is declared as public, the interface can be used by code outside its package.

Using an interface (implements):

- A class is associated to an interface by using **implements** clause in the class declaration.
- Because of the above declaration, we can now use FlightEnabled as the reference type and assign it to the instance of a bird. See below example.

```
public class Bird implements FlightEnabled {

}
```

```
FlightEnabled flier = new Bird();
```

Note: A class can only extend single class. This is why Java is called single inheritance whereas a class can implement many interfaces. This gives us plug and play functionality. A class can both extend to another class and implement one or more interfaces.

- All interfaces are implicitly declared as abstract, you don't have to specify it explicitly.
- Similarly, all methods declared without a body of an interface are implicitly declared as both public and abstract.
- If we omit the access modifier on interface member, it implicitly is public, Whereas if we omit an access modifier on a class member, it implicitly is package private. A field/variable declared on an interface is implicitly public, static and final (This means the values of these variables cannot be changed by the class that is implementing the interface).
- If we try to change access modifier of our interface method to protected, it gives a compilation error. Only concrete methods can have private access.
- Any class that implements an interface, it has to implement all the abstract methods of those interfaces. Additionally, this class can also have methods of its own.

Final keyword: When you declare a field on interface, it actually means final static field (A field which cannot be reassigned or given a different value after its initialization.)

- A field declared as final static means the objects field cannot be reassigned or given different value after its initialization.
- A final variable in a block of code. This means once it is assigned a value, any remaining code in the block can't change it.
- A final variable in the method: It also means final method parameter, which means we can't assign a different value to a parameter in the method Code block.
- A final method means it cannot be overridden by a subclass.
- A final class can't be overridden meaning no class can use it in the extents class.

Variables in Interfaces: Constants in Java?

Constant in Java is a variable that can't be changed. Constant in Java are usually named with uppercase letters and with underscores between words. A constant variable is a final variable of either primitive type or type String that is initialized with a constant expression.

E.g. INTEGER.MAX_VALUE, and the INTEGER.MIN_VALUE fields.

Accessing implementations through interface reference:

Here we can see that the class implementing interface is implementing all the abstract methods of the interface. Additionally, it is also having its own methods.

We can also declare variables as object references that uses an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. We can see that, variable calls the callback() via an interface reference variable.

We can see that variable C is declared to be of the interface type Callback, yet it was assigned to an instance of class Client.

Interface reference variables e.g. **C** do not have access to other members of the class. They can only access methods declared by interface declaration.

```
interface Callback {
  void callback(int param);
}

class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("callback called with " + p);
  }

  void nonIfaceMeth() {
    System.out.println("Classes that implement interfaces " +
                       "may also define other members, too.");
  }
}

class TestIface {
  public static void main(String[] args) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

Output:

```
callback called with 42
```

Partial implementation: Any class that implements interface must implement all the abstract methods of interface. Otherwise the class must declare itself as abstract.

Nested interface: An interface can be declared as a member of a class or another interface. These interfaces. Are called member interfaces or nested interface. And this nested interface can be declared as public, private or protected. This is different from top level interface which either must be public or use default access level.

Note: when a nested interface is used outside office and closing scope, It must be qualified by the name of the class or interface of which it is a member.

Notice that A defines a member interface called NestedIF and that it is declared public. Next, B implements the nested interface by specifying implements **A.NestedIF**. Here the name is fully qualified by using enclosed class name.

```
// A nested interface example.

// This class contains a member interfac
class A {
  // this is a nested interface
  public interface NestedIF {
    boolean isNotNegative(int x);
  }
}

// B implements the nested interface.
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false: true;
  }
}
class NestedIFDemo {
  public static void main(String[] args) {

    // use a nested interface reference
    A.NestedIF nif = new B();

    if(nif.isNotNegative(10))
      System.out.println("10 is not negative");
    if(nif.isNotNegative(-12))
      System.out.println("this won't be displayed");
  }
}
```

Interfaces can be extended:

- For a class to implement interface we use implements keyword.
- For an interface to implement another interface we use extends keyword instead of implements keyword.
- A class cannot inherit multiple super classes using extends keyword. However an interface can use the extends expression with multiple interfaces.
- In this scenario, if a class implements an interface that inherits another interface, it must provide implementations for all the abstract methods required by interface inheritance chain. If not implemented, it causes compilation error.

java

```java
interface Swim {
    void swim();
}

interface Fly {
    void fly();
}

interface Duck extends Swim, Fly {
    // Duck interface inherits behavior
}
```

java

```java
class MallardDuck implements Duck {
    @Override
    public void swim() {
        System.out.println("Mallard duck is swimming");
    }

    @Override
    public void fly() {
        System.out.println("Mallard duck is flying");
    }
}
```

Default interface methods: In JDK 8 version, a new capability is added to interface called default method. A default method lets you define a default implementation for interface methods, means it allows you to provide a body rather than being abstract. By providing the default, the interface makes the implementation of default methods by class optional.

- It is important to note that default methods do not change the key aspect of interface. The interface still cannot have instance variables. This is still the major difference between the interface and a class. It is still not possible to create an instance for an interface by itself. An interface must be implemented by a class if an instance is to be created. The default method just gives you added flexibility.
- The default method in an interface is created by using default keyword before the method declaration.

Multiple Inheritance Issue: Classes can implement multiple interfaces, allowing them to inherit behavior from multiple sources without introducing conflict, because interfaces define contracts without providing implementation. However. Interfaces does not solve multiple inheritance issue because the difference between a class and interface is a class can maintain a state information especially through the use of instant variables whereas interface cannot.

For example, let's consider a class that implements 2 interfaces and both the interfaces has same default method. In this case, which interface default method is used? What happens if a class has its own implementation of the method?

- A class implementation takes priority over interface default implementation. In this case both the interface default methods are overridden by class implementation.

- In case if one interface inherits other, both defining a common default method, the inheriting interface version of the method will take precedence.
- In case if one interface does not inherit other interface but both define a common default method And if Glass also does not override the default method, then an error will occur.

Static interface methods: Starting JDK 8, we can include one or more static methods in an interface.

Note: A static method defined by an interface can be called independently of any object. No implementation of interface is necessary. And no instance of interface is required.

Additionally, static interface methods are not inherited by either an implementing class or an sub interface.

Private interface methods: Starting JDK 9, Private methods can also be included in an interface. However, it cannot be used by code outside the interface in which it is defined. The restrictions also include some interfaces, because private interface methods are also not inherited by sub interface.

Starting Java 14, records and enums can implement interfaces. This allows them to provide implementation for the methods declared in those interfaces, providing additional flexibility in designing the code.

Abstract Class

- With abstract classes, you can declare fields that aren't static and final, instance fields in other words.
- Also with abstract classes, we can use any of the four access modifiers for its concrete methods. We can also use all but the private access modifier, for its abstract methods.
- An abstract class can extend only one parent class, but it can implement multiple interfaces.
- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it doesn't, then the subclass must also be declared abstract.

Interface

- In an interface, we define what kind of operation an object can perform. These operations are defined by the classes that implement the interface.
- Interfaces form a contract between the class, and the outside world, and this contract is enforced at build time, by the Java compiler.
- You can't instantiate interfaces, but they may contain a mix of methods declared with, or without an implementation.
- All methods on interfaces, declared without a method body, are automatically public and abstract.
- An interface can extend another interface.

	Abstract Class	**Interface**
An instance can be created from it	No	No
Has a constructor	Yes	No
Implemented as part of the Class Hierarchy. Uses Inheritance	Yes (in extends clause)	No (in implements clause)
records and enums can extend or implement?	No	Yes
Inherits from java.lang.Object	Yes	No
Can have both abstract methods and concrete methods	Yes	Yes (as of JDK 8)
Abstract methods must include abstract modifier	Yes	No (Implicit)
Supports default modifier for it's methods	No	Yes (as of JDK 8)
Can have instance fields (non-static instance fields)	Yes	No
Can have static fields (class fields)	Yes	Yes - (implicitly **`public static final`**)