List of Concepts Involved:

- What is an interface? : almost same as abstract class
- Need of Interface
- Important key points of Interface
- Abstract vs interface
- Additional features of Interface
- Functional Interface

What is an interface?

- Interface is a Java Feature, it will allow only abstract methods.
- In Java applications, for interfaces, we are able to create only reference variables, we are unable to create objects.
- In the case of interfaces, by default, all the variables are "public static final".
- In the case of interfaces, by default, all the methods are "public and abstract".
- In Java applications, constructors are possible in classes and abstract classes but constructors are not possible in interfaces.
- Interfaces will provide more shareability in Java applications when compared with classes and abstract classes.

Need of Interface

- Any service requirement specification (srs) is called an interface.

Example 1:
Sun people responsible to define JDBC API and database vendor will provide implementation for that.

Example 2:
Sun people define Servlet API to develop web applications web server vendor is responsible to provide implementation.

- From the client point of view an interface define the set of services what is expecting. From the service provider point of view an interface defines the set of services it is offering. Hence an interface is considered as a contract between client and service provider.

Example 3:
ATM GUI screen describes the set of services that bank people offer, at the same time the same GUI screens the set of services that customer is expecting hence this GUI screen acts as a contract between bank and customer.

- Inside interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.

Note:
Any service requirement specification (SRS) or any contract between client and service provider or 100% pure abstract classes is considered as an interface.

Important key points of Interface

- Whenever we are implementing an interface compulsory for every method of that interface we should provide implementation otherwise we have to declare class as abstract in that case child class is responsible to provide implementation for remaining methods.
- Whenever we are implementing an interface method, it should be declared as public, otherwise we will get compile time error.
- In Java applications, it is not possible to extend more than one class to a single class but it is possible to extend more than one interface to a single interface. – multiple inheritance possible
- In Java applications, it is possible to implement more than one interface into a single implementation class.
- A class can extend a class and can implement any no. Of interfaces simultaneously.
- An interface can extend any no. Of interfaces at a time.

Note:
Which of the following is true?

1. A class can extend any no.Of classes at a time.
2. An interface can extend only one interface at a time.
3. A class can implement only one interface at a time.
4. A class can extend a class and can implement an interface but not both simultaneously.
5. An interface can implement any no.Of interfaces at a time.
6. None of the above.

Ans: 6

Consider the expression X extends Y for which of the possibilities of X and Y this expression is true?

1. Both x and y should be classes.
2. Both x and y should be interfaces.
3. Both x and y can be classes or can be interfaces.
4. No restriction.

Ans: 3

Interface methods:
Every method present inside the interface is always public and abstract whether we are declaring or not. Hence inside the interface the following method declarations are equal.

```
void methodOne();
public Void methodOne();
abstract Void methodOne(); Equal
public abstract Void methodOne();
```

public:
To make this method available for every implementation class.
abstract: Implementation class is responsible to provide implementation .
As every interface method is always public and abstract we can't use the following modifiers for interface methods.

private, protected, final, static, synchronized, native, strictfp.

Inside the interface which method declarations are valid?

1. public void methodOne(){}
2. private void methodOne();
3. public final void methodOne();
4. public static void methodOne();
5. public abstract void methodOne();

Ans: 5

Interface variables:

- An interface can contain variables
- The main purpose of interface variables is to define requirement level constants.
- Every interface variable is always public static and final whether we are declaring or not.

Example:

```
interface interf
{
int x=10;
}
```

public:
To make it available for every implementation class.

static: Without an existing object also we have to access this variable.
final: Implementation class can access this value but cannot modify.

Hence inside the interface the following declarations are equal.
int x=10;
public int x=10;
static int x=10;
final int x=10;
public static int x=10;
public final int x=10;
static final int x=10;
public static final int x=10;

As every interface variable by default public static final we can't declare with the following modifiers.

o Private
o Protected
o Transient
o Volatile

For the interface variables compulsory we should perform initialization at the time of declaration only otherwise we will get a compile time error.

Example:

```
interface Interf
{
int x;
}
```

Output:
Compile time error.
D:\Java>javac Interf.java Interf.java:3: = expected
int x;

Which of the following declarations are valid inside the interface ?
1. int x;
2. private int x=10;
3. public volatile int x=10;
4. public transient int x=10;
5. public static final int x=10;

Ans: 5

Note:
Interface variables can be accessed from the implementation class but cannot be modified

```
interface Interf
{
 int x= 10;
}
class Demo implements Interf
{
 public static void main(String[] args)
 {
  x  = 20;
  System.out.println(x);
 }
}
```

Output: Compile time error.

Interface naming conflicts

1. Method naming conflicts

Case 1:
If two interfaces contain a method with same signature and same return type in the implementation class only one method implementation is enough.

```
interface Left
  {
     public void methodOne();
  }

interface Right
  {
     public void methodOne();
  }

class Test implements Left,Right
{
     public void methodOne()
     {
     }
}
```

Output:
D:\Java>javac Left.java
D:\Java>javac Right.java
D:\Java>javac Test.java

Case 2:
if two interfaces contain a method with same name but different arguments in the implementation class we have to provide implementation for both methods and these methods act as an overloaded method.

```
interface Left
  {
     public void methodOne();
  }
interface Right
  {
     public void methodOne(int i);
  }

class Test implements Left,Right
  {
     public void methodOne()
     {
     }
     public void methodOne(int i)
     {
     }
}
```

overloaded methods

Output:
D:\Java>javac Left.java
D:\Java>javac Right.java
D:\Java>javac Test.java

Case 3:
If two interfaces contain a method with the same signature but different return types then it is not possible to implement both interfaces simultaneously.

```
interface Left
  {
    public void methodOne();
  }

interface Right
  {
    public int methodOne(int i);
  }
```

We can't write any java class that implements both interfaces simultaneously.

Note:
Is a java class can implement any no. Of interfaces simultaneously ?
Yes, except if two interfaces contains a method with the same signature but different return types.

Variable naming conflicts
Two interfaces can contain a variable with the same name and there may be a chance of variable naming conflicts but we can resolve variable naming conflicts by using interface names.

```
interface Left
  {
    int x=888;
  }
interface Right
  {
    int x=999;
  }
class Test implements Left,Right
{
   public static void main(String args[]){
       //System.out.println(x);
       System.out.println(Left.x);
       System.out.println(Right.x);
   }
}
```

Output:
D:\Java>javac Left.java
D:\Java>javac Right.java
D:\Java>javac Test.java
D:\Java>java Test
888
999

Marker interface:

- If an interface doesn't contain any methods and by implementing that interface if our objects will get some ability such type of interfaces are called Marker interface (or) Tag interface (or) Ability interface.
- Serializable,Cloneable,RandomAccess,SingleThreadModel.
- By implementing a **Serializable** interface we can send that object across the network and we can save the state of an object into a file.
- By implementing **SingleThreadModel** interface Servlet can process only one client request at a time so that we can get "Thread Safety".
- By implementing a **Cloneable** interface our object is in a position to provide exactly duplicate cloned object.

Note:
Without having any methods in the marker interface how objects will get ability ?
Internally JVM is responsible to provide required ability.

When should we go for interface, abstract class and concrete class?

- If we don't know anything about implementation just we have requirement specification then we should go for an interface.
- If we are talking about implementation but not completely (partial implementation) then we should go for an abstract class.
- If we are talking about implementation completely and ready to provide service then we should go for a concrete class.

What is the Difference between interface and abstract class ?

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.
- For the interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.
- Inside the interface we can't take static and instance blocks.
- Inside the interface we can't take constructor.

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.
- There are no restrictions on abstract class variable modifiers.
- It is not required to perform initialization for abstract class variables at the time of declaration.
- Inside abstract class we can take both static and instance blocks.
- Inside the abstract class we can take constructor.

Note:
Every method present inside interface is abstract but in abstract class also we can take only abstract methods then what is the need of interface concept ?
We can replace interface concepts with abstract class. But it is not a good programming practice. We are misusing the role of abstract class. It may create performance problems also.

JAVA8 Features over Interfaces:

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

1. Default Methods in Interfaces:

- In general, if we declare abstract methods in an interface then we have to implement all that interface methods in more no.of classes with variable implementation part.
- In the above context, if we require any method implementation common to every implementation class with fixed implementation then we have to implement that method in the interface as default method.
- To declare default methods in interfaces we have to use the "default" keyword in method syntax like access modifier.

```
interface Interf{
       default void m1(){
              System.out.println("m1-A");
       }
}
class A implements Interf{

}
class Test{
      public static void main(String args[]){
             Interf i=new A();
             i.m1();
      }
}
```

NOTE:
It is possible to provide more than one default method within a single interface.

Example:

```
interface Interf{
        default void m1(){
                        ---------
        }
        default void m2(){
                        --------
        }
}
```

In JAVA8,it is possible to override default methods in the implementation classes.

```
interface Interf{
        default void m1(){
               System.out.println("m1-A");
        }
}
class A implements Interf{
        public void m1(){
              System.out.println("m1-A");
        }
}
class Test{
        public static void main(String args[]){
              Interf i=new A();
                    i.m1();
        }
}
```

2. Static Methods in Interfaces:

- Upto JAVA7 version, static methods are not possible in interfaces but from JAVA8 version static methods are possible in interfaces in order to improve sharability.
- If we declare static methods in the interfaces then it is not required to declare any implementation class to access that static method,we can use directly interface name to access static method.

NOTE:
If we declare static methods in an interface then they will not be available to the respective implementation classes,we have to access static methods by using only interface names not even by using interface reference variable

Example:

```
interface I{
       static void m1(){
             System.out.println("m1-1");
       }
}
class Test{
       public static void main(String args[]){
             I.m1();
       }
}
```

Note:
In JAVA8 version, interfaces will allow concrete methods along with either "static" keyword or "default" keyword.

3. 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

* this method of function interface is useful in Lambda Expression

EX:
java.lang.Runnable
java.lang.Comparable

NOTE:
In Functional Interfaces we have to provide only one abstract method but we can provide any no.of default methods and any no.of static methods.

```
@Functional Interface
interface Interf{
	void m1();

	default void m3(){
		System.out.println("m3-I");
	}
	static void m4(){
		System.out.println("m4-I");
	}
}
class A implements Interf{
	public void m1(){
		System.out.println("m1-A");
	}
}
public class Test{
	public static void main(String args[]){
		Interf i=new A();
		i.m1();
		i.m3();
		Interf.m4();
	}
}
```

Output:
m1-A
m3-I
m4-I

Inner Classes :- (Class inside a Class)

- ① Member Inner Class :- it is non static
- ② Static Inner Class : static class # Static keyword is always used by inner class only.
- ③ Anonymous Class :-

① Member Inner Class :-

List of Concepts Involved:

Why use Lambda Expression
① uses function interface
② less coding

- What is Lambda Expression
- Different ways to create Lambda Expression
- Lambda Expression exercises

Lambda Expression

- Lambda calculus is a big change in the mathematical world which was introduced in 1930.
- Because of the benefits of Lambda calculus, slowly these concepts started being used in the programming world.
- "LISP" is the first programming which uses Lambda Expression.
- The other languages which uses lambda expressions are:
  C#.Net
  C Objective
  C
  C++
  Python
  Ruby etc.
  and finally in java also.
- The Main Objective of λLambda Expression is to bring benefits of functional programming into java.

What is Lambda Expression (λ):

- Lambda Expression is just an anonymous(nameless) function. That means the function which doesn't have the name, return type and access modifiers.
- Lambda Expression also known as anonymous functions or closures.

Ex: 1

```
public void m1() {
        System.out.println("hello");
}
```

Equivalent lambda expressions

```
() →System.out.println("hello");
```

Ex:2

```
public void add(int a, int b) {
        System.out.println(a+b);
}
```

Equivalent lambda expressions

```
(a,b) →System.out.println(a+b);
```

- If the type of the parameter can be decided by the compiler automatically based on the context then we can remove types also.
- The above Lambda expression we can rewrite as (a,b)->System.out.println (a+b);

Ex: 3

```
public String str(String str) {
        return str
}
```

Equivalent lambda expressions

```
(str) →str
```

Conclusions:

- A lambda expression can have zero or more parameters(arguments).
- Usually we can specify the type of parameter.If the compiler expects the type based on the context then we can remove type. i.e., a programmer is not required.
- If multiple parameters are present then these parameters should be separated with comma(,).
- If there are zero number of parameters available then we have to use empty parameter [like ()].
- If only one parameter is available and if the compiler can expect the type then we can remove the type and parentheses also.
- Similar to method body lambda expression body also can contain multiple statements.if more than one statements present then we have to enclose inside within curly braces.
- if one statement is present then curly braces are optional.
- Once we write a lambda expression we can call that expression just like a method, for this functional interfaces are required.

Functional Interfaces:

If an interface contains only one abstract method, such types of interfaces are called functional interfaces and the method is called functional method or single abstract method(SAM).

- **Runnable** It contains only run() method
- **Comparable** It contains only compareTo() method
- **ActionListener** It contains only actionPerformed()
- **Callable** It contains only call()method

Inside the functional interface in addition to the single Abstract method(SAM) we write any number of default and static methods.

Ex:

```
interface Interf {
       public abstract void m1();
       default void m2() {
              System.out.println (“hello”);
       }
}
```

In Java 8 ,SunMicroSystem introduced @FunctionalInterface annotation to specify that the interface is FunctionalInterface.

Ex:

```
@FunctionalInterface
Interface Interf {
	public void m1();
}
```

InsideFunctionalInterface we can take only one abstract method,if we take more than one abstract method then the compiler raise an error message that is called we will get compilation error.

Ex:

```
@FunctionalInterface {
	public void m1(); //this code gives compilation error.
	public void m2();
}
```