List of Concepts Involved:

- What is an interface?
- 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.
- 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)
     {
     }
}
```

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