CoreJava\interfaces.md

Abstract Classes and Interfaces

Abstract methods must be in abstract classes.

Interfaces

Differences in Abstract Classes and Interfaces:

1. Abstract classes are able to allow both concrete methods and abstract methods. Interfaces are able to allow only abstract methods.
2. To declare abstract classes, we need the `class` keyword and the `abstract` keyword. To declare interfaces, we need the `interface` keyword.
3. For abstract classes and interfaces, we are able to create reference variables, but making objects of these is not allowed.
4. In an interface, all variables are **`public, static, and final`**. In an abstract class, this is not the case and we need to declare them explicitly.
5. In an interface, all methods are `public and abstract`, so they don't need to be explicitly marked as such. In the case of an abstract class, there's no default case, and we need to declare methods explicitly.
6. In an interface, `all inner classes are static in nature`, but this is not the case in abstract classes. There's no default case in abstract classes.
7. Abstract classes allow constructors, `but interfaces do not allow constructors`.
8. Abstract classes can have static blocks and instance blocks, but these are not allowed in interfaces.
9. Abstract classes have medium shareability, whereas interfaces have high shareability.
10. interface can extend as many intefaces inside it, but abstract class can extend only one class.

Can interfaces have static methods?

- No. Interfaces can’t have static methods.

Can we override an interface method with visibility other than public?

- No. While overriding any interface methods, we should use public only.
- Because, all interface methods are public by default and you should not reduce the visibility while overriding them.

Can a Method be private in an abstract class?

- No the method cannot be private

Can an abstract method declaration include throws?

- Yes, it is possible

Can a Method be static and abstract in an abstract class?

- No as static methods belong to class and resolved at compile time there is no implementation available at compile time. So they are not allowed.

Can a method be synchronized in an inteface?

- No it is not possible

Can a method be native or strictfp in interface?

- No it is not allowed.

Can we write an inner class in interface?

- Yes, we can and they are static by default.

Example:

```
class A implements I{
    public void m1() {
        // Always make it public as the access modifier cannot be narrower than
        // All methods of interfaces are public and abstract in nature.
    }

    public static class Test{
        public static void main(String[] args) {
            I i = new A();
            i.m1();
            i.m2();
            // We are using the reference variable of the parent and not the ch
        }
    }
}

interface I1 {
    void m1();
    void m2();
}

interface I3 {
    void m3();
}

class A implements I1, I2 {
    public void m1() {
        // sout
    }

    public void m2() {
        // sout
    }

    public void m3() {
        // sout
    }
}
```

Valid Syntax Questions

1. class extend class- valid
2. class extend class, class- invalid
3. class extend abstract class- valid
4. class extend abstract class, abstract class- invalid
5. class extend interface- valid
6. class extend class implement interface- valid
7. class extend class implement interface, interface- valid
8. class extend abstract class implement inteface- valid
9. class extend abstract class implement interface, interface -valid
10. class implement interface extend class - invalid

class should come first and then interface.

MARKER INTERFACES

1. It is a java interface, it does not include abstract methods and it will be implemented to get some abilities to the respective object at runtime.
2. These are tag interfaces which act as the set of rules and tells the compiler that this object follows these set of rules.

Examples of Marker Interfaces

1. Cloneable Interface
2. Serializable Interface
3. Remote Interface
4. Random Interface

What is the need of marker interfaces

1. provide information at runtime to the libraries
2. Helps in behaviour indication.Eg: Serializable Interface
3. Helps in intgeration with the frameworks like Remote interface
4. Helps in figuring out the intent, they can undestand the expected behaviour without implementation details.

Can we Write our own marker interface?

1. Yes, we can write our own by reflection API.

What is Serializable Interface

1. Serializable interface is present in java.io package. It is used to make an object eligible for saving its state into a file. This is called **Serialization**.
2. Java will throw a **`java.io.NotSerializableException`**

```
import java.io.*;

// By implementing Serializable interface
```

```
// we make sure that state of instances of class A
// can be saved in a file.
class A implements Serializable
{
    int i;
    String s;

    // A class constructor
    public A(int i,String s)
    {
        this.i = i;
        this.s = s;
    }
}

public class Test
{
    public static void main(String[] args)
      throws IOException, ClassNotFoundException
    {
        A a = new A(20,"GeeksForGeeks");

        // Serializing 'a'
        FileOutputStream fos = new FileOutputStream("xyz.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a);

        // De-serializing 'a'
        FileInputStream fis = new FileInputStream("xyz.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        A b = (A)ois.readObject();//down-casting object

        System.out.println(b.i+" "+b.s);

        // closing streams
        oos.close();
        ois.close();
    }
}
```

Cloneable Interface

1. java.lang package
2. This is a set of rules that shows that yes it can be cloned.
3. To clone an object it must implement the cloneable interface. Second step is to override the clone() method in the Object class.
4. the Java runtime will throw a **`CloneNotSupportedException`** exception. This exception indicates that the object cannot be cloned because it doesn't follow the required rules by not implementing the **`Cloneable`** interface.

```
public class NonCloneableClass {
    public static void main(String[] args) {
        NonCloneableClass original = new NonCloneableClass();
```

```
        try {
            NonCloneableClass copy = (NonCloneableClass) original.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```