CORE JAVA

Day-6

INTERFACES

- In Java, an interface is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields.
- The methods in interfaces are abstract by default, which means they do not specify a body and instead only define method signatures.
- Interfaces are used to specify a set of methods that a class must implement, thereby providing a way to enforce certain functionalities across multiple classes.

KEY CHARACTERISTICS OF INTERFACES

- Abstract Methods: Every method defined in an interface is implicitly abstract, meaning it must be implemented by the class(es) that agree to the contract defined by the interface, except in cases where the method is a default or static method.
- Implementing Interfaces: A class uses the implements keyword to incorporate an interface. A class can implement multiple interfaces, which helps to overcome the limitation of single inheritance in Java by allowing a class to inherit from more than one abstract type.
- Default Methods: Since Java 8, interfaces can contain default methods. These are methods that can have a body. Default methods were introduced to provide additional functionality to interfaces without breaking existing implementations.
- Static Methods: Interfaces can also have static methods which can be called independently of an object instance, similar to static methods in classes.
- Multiple Inheritance of Type: A class can implement several interfaces, which allows it to inherit from multiple abstract "types". This provides a way to circumvent Java's restriction against multiple inheritance of classes.

NEW INTERFACE FEATURES (JAVA 8 & 11)

- With Java 8, interfaces were significantly enhanced with the introduction of default and static methods.
- Default Methods: Allow the interfaces to have methods with implementation without breaking existing implementations of these interfaces.
- Static Methods: Static methods in interfaces are similar to static methods in classes. They belong to the interface, not the object instance of the interface. Java 11 did not introduce any specific changes to interfaces; most interface-related enhancements were part of Java 8.

DEFAULT METHODS IN INTERFACE

- In Java, default methods were introduced in Java 8 to enable the developers to add new functionalities to interfaces without breaking the existing implementations of these interfaces.
- The introduction of default methods has made it possible to add new functionality to interfaces while ensuring that the classes that implement these interfaces do not have to change unless they want to override these new methods.

```
public interface Vehicle
{
    default void print() {
        System.out.println("I am a vehicle!");
    }
}
```

STATIC METHODS IN INTERFACE

- In Java, static methods in interfaces were introduced in Java 8 along with default methods.
- Class-Level Methods: they belong to the interface, not to any instance of it. They can be called without creating an instance of any implementing class.
- Invocation: They must be invoked using the interface name, not an instance of the interface or an implementing class.
- Utility Functions: They are useful for providing utility methods related to the interface.
- Not Inheritable: Unlike other methods in interfaces, static methods are not inherited by the classes that implement the interface. This means they cannot be overridden and must be called on the interface itself.

```
public interface Loggable {
    static void log(String message) {
        System.out.println("Log: " + message);
    }
}
```

FUNCTIONAL INTERFACE

- A functional interface in Java is an interface that has exactly one abstract method, although it can have multiple default and static methods. These interfaces are used as the basis for lambda expressions and method references.
- Example:

```
@FunctionalInterface
public interface SimpleFunctionalInterface {
    void execute();
}
```

- The @FunctionalInterface annotation is optional but helps in providing a clearer intent and enforcing the rule that exactly one abstract method must be present.

NESTED CLASSES

- An inner class in Java is a class defined within another class. Java supports several types of inner classes: non-static nested classes (often simply called inner classes), static nested classes, local classes (classes defined within a block of code), and anonymous classes (a local class without a name).
- Each type of inner class has its own peculiarities and use cases.

NON-STATIC NESTED CLASSES (INNER CLASSES)

- An inner class is associated with an instance of its enclosing class and can access both static and non-static members of the enclosing class. The objects of the inner class cannot exist without an instance of the outer class.

```
class Outer {
    private int outerField = 30;
    class Inner {
        void display() {
            // Can access private members of outer class
            System.out.println(outerField);
        }
    }
    public void createInner() {
        Inner inner = new Inner();
        inner.display();
    }
}
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.createInner();
    }
}
```

STATIC NESTED CLASSES

- Unlike inner classes, static nested classes are associated with the outer class rather than an instance of the outer class. This means they can only access the static members of the outer class.

```
class Outer {
    private static int outerStaticField = 10;
    static class StaticNested {
        void display() {
            // Can access static members of outer class
            System.out.println(outerStaticField);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Outer.StaticNested nested = new
Outer.StaticNested();
        nested.display();
    }
}
```

LOCAL CLASSES (CLASSES INSIDE A BLOCK)

- Local classes are defined within a block, usually a method, and are invisible outside of that block. Like inner classes, they can access local variables and parameters of the enclosing block that are effectively final.

```
class Outer {
    void outerMethod() {
        int x = 100;
        class Local {
            void display() {
                System.out.println(x);
            }
        }
        Local local = new Local();
        local.display();
    }
}
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.outerMethod();
    }
```

ANONYMOUS CLASSES USING ABSTRACT CLASS

- An anonymous class is a local class without a name. It is both declared and instantiated all at once and is useful for instantiating classes that are used only once.

```
abstract class AbstractDemo {
    abstract void show();
}
public class Main {
    public static void main(String[] args) {
        AbstractDemo obj = new AbstractDemo() {
            void show() {
                System.out.println("Anonymous Class Example");
            }
        };
        obj.show();
    }
}
```

ANONYMOUS CLASS USING INTERFACE

- This is a common use case for anonymous classes, especially in situations where you need to implement an interface with only one or a few methods for a specific purpose without having to actually implement the interface in a named class.

```
interface IntegerMath {
    int operation(int a, int b);
}
public class Main {
    public static void main(String[] args) {
        // Create an anonymous class that implements IntegerMath to perform addition
        IntegerMath add = new IntegerMath() {
            @Override
            public int operation(int a, int b) {
                return a + b;
            }
        };
        // Use the implementation
        System.out.println("Result of addition: " + add.operation(5, 3));
    }
}
```

LAMBDA EXPRESSIONS

- Lambda expressions in Java, introduced in Java 8, are a way to provide a clear and concise way to implement functional interfaces. A functional interface is an interface with a single abstract method (SAM), and lambda expressions allow these interfaces to be implemented in a succinct manner, often with a single line of code.

  - `() -> System.out.println("Hello, world!");`
  - `(String name) -> System.out.println("Hello, " + name);`
  - `name -> System.out.println("Hello, " + name);`
  - `(int a, int b) -> a + b;`
  -
```
(String s1, String s2) -> {
        if (s1.length() > s2.length())
           return s1;
        else
           return s2;
    };
```

MARKER INTERFACE

- A marker interface is an interface with no method declarations. It is used to convey metadata about a class that implements it.
- Essentially, it "marks" a class as being capable or possessing certain traits simply by the fact of its implementing the interface.
- Marker interfaces are used for signaling, type declaration, or applying special behavior in conditional structures such as instanceof checks.
- Examples of marker interfaces in Java include Serializable and Cloneable.
- These interfaces do not contain any methods but indicate to the Java runtime and compiler that classes implementing these interfaces should be treated in a special way (e.g., they can be serialized or cloned).

INTERFACE VARIABLES

- In Java, interfaces can define constants, which are implicitly public, static, and final.
- Characteristics
  - Public: Interface variables are accessible from anywhere other variables would normally be accessed.
  - Static: They belong to the interface itself rather than to any instance of classes that implement the interface.
  - Final: Their values cannot be changed once they are set. They must be initialized when declared.