Interface in Java

An **interface** acts as a contract or a communication medium between the **user** and the **device** (or between different parts of a program).

Interface Variables

- **All variables** declared inside an interface are implicitly:
  - `public` — accessible everywhere
  - `static` — belong to the interface itself (no instance needed)
  - `final` — constant; value cannot be changed
- Therefore, interface variables **must be initialized** at the time of declaration.
- Access interface variables using the **interface name**:

```java
interface A {
    int x = 10; // public static final by default
}

interface B {
    int x = 30;
}

class C implements A, B {
    public static void main(String[] args) {
        int x = 20;
        System.out.println(x);   // Prints: 20 (local variable)
        System.out.println(A.x); // Prints: 10 (from interface A)
        System.out.println(B.x); // Prints: 30 (from interface B)
    }
}
```

Interface Methods

- All methods declared in an interface are implicitly:
  - `public`

    - `abstract` (except for `default`, `static`, and `private` methods introduced in later Java versions)
- **From Java 8 onwards**, interfaces can contain:
    - **Default methods** — methods with a body, marked by the `default` keyword. Called via child class objects.
    - **Static methods** — utility methods, called via the interface name.
    - **Example:**

```java
interface A {
    void func1();
    void func2();

    default void func() {
        System.out.println("default func");
    }

    static void staticFunc() {
        System.out.println("static func");
    }

    private void pfunc() {
        System.out.println("private func");
    }

    default void accessPfunc() {
        pfunc();
        System.out.println("default func to access pfunc");
    }

    public static void main(String[] args) {
        staticFunc();
    }
}

class B implements A {
    public void func1() { /* implementation */ }
    public void func2() { /* implementation */ }

    public static void main(String[] args) {
        B ob = new B();
        ob.func();          // calls default method
        A.staticFunc();     // calls static method via interface
        ob.accessPfunc();    // calls default method that accesses private method
    }
}
```

- **From Java 9 onwards,** interfaces can also have **private methods** to encapsulate shared code inside the interface.

Method Implementation in Multiple Interfaces

- **Case 1:** Two interfaces contain methods with the **same signature and same return type**
    - The implementing class provides a **single implementation** that serves both.

```java
interface A { void func(); }
interface B { void func(); }

class C implements A, B {
    public void func() {
        System.out.println("Single implementation for both interfaces");
    }
}
```

- **Case 2:** Two interfaces contain methods with the **same name but different signatures** (overloaded)
    - The implementing class provides **multiple method implementations**, each corresponding to one interface method.

```java
interface A { void func(); }
interface B { void func(int x); }

class C implements A, B {
    public void func() {
        System.out.println("No-arg func");
    }

    public void func(int x) {
        System.out.println("func with int argument");
    }
}
```

- **Case 3:** Two interfaces contain methods with the **same signature but different return types**
    - This situation is **not allowed** and cannot be implemented.

Marker Interface

- An interface with **no methods or fields**.
- Serves as a **tag** to provide metadata or special behavior to classes that implement it.
- Examples:
    - `Cloneable` — indicates a class's objects can be cloned.
    - `Serializable` — indicates a class's objects can be serialized (converted to a byte stream).
    - `RandomAccess` — indicates efficient random access capability for collections.
- Marker interfaces play a vital role in **runtime type checking** and special processing (e.g., serialization, cloning).
