Multiple inheritance through interface

To resolve the ambiguity between interfaces we are using super keyword to call parent interface methods.

```java
package com.pack1;

public interface InterfaceA
{
    default void msg()
    {
        System.out.println("Java is awesome");
    }
}
```

```java
package com.pack1;

public interface InterfaceB
{
    default void msg()
    {
        System.out.println("Java is amazing!!!");
    }
}
```

```java
package com.pack1;

public class ClassA implements InterfaceA,InterfaceB
{
    public void msg()
    {
        System.out.println("We are overridding the msg()");
        InterfaceA.super.msg();
    }
    public static void main(String[] args)
    {
        ClassA aobj=new ClassA();
        aobj.msg();
    }
}
```

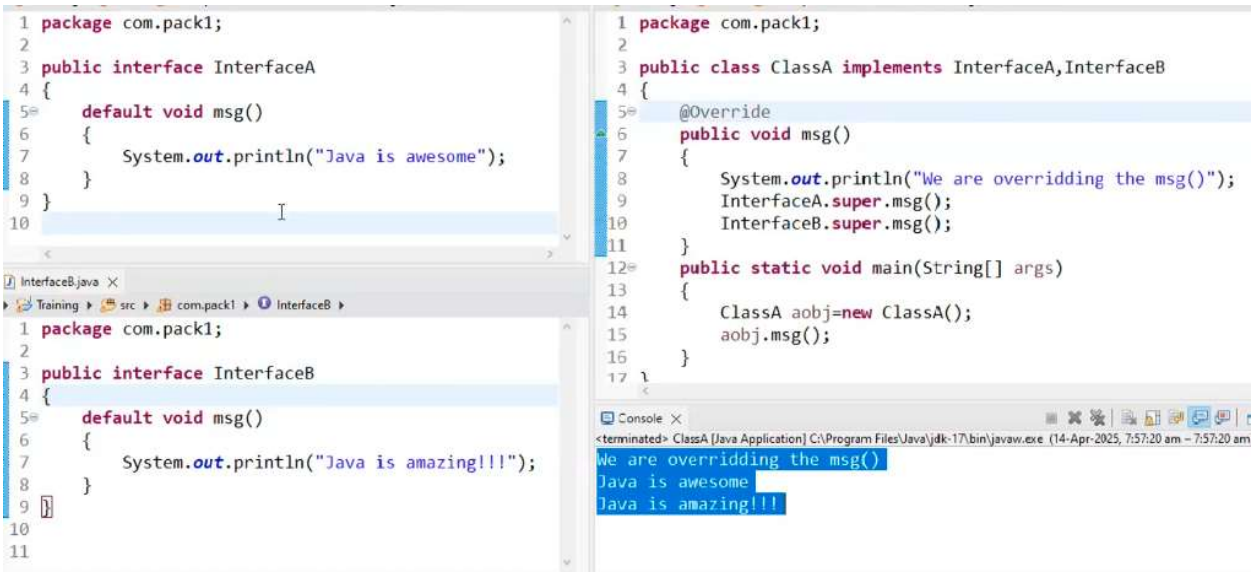
```
<terminated> ClassA [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (14-Apr-2025, 7:56:44 am – 7:56:45 am) [pid: 15264]
We are overridding the msg()
Java is awesome
```

```java
package com.pack1;

public interface InterfaceA
{
    default void msg()
    {
        System.out.println("Java is awesome");
    }
}
```

```java
package com.pack1;

public interface InterfaceB
{
    default void msg()
    {
        System.out.println("Java is amazing!!!");
    }
}
```

```java
package com.pack1;

public class ClassA implements InterfaceA,InterfaceB
{
    public void msg()
    {
        System.out.println("We are overridding the msg()");
        InterfaceA.super.msg();
        InterfaceB.super.msg();
    }
    public static void main(String[] args)
    {
        ClassA aobj=new ClassA();
        aobj.msg();
    }
}
```

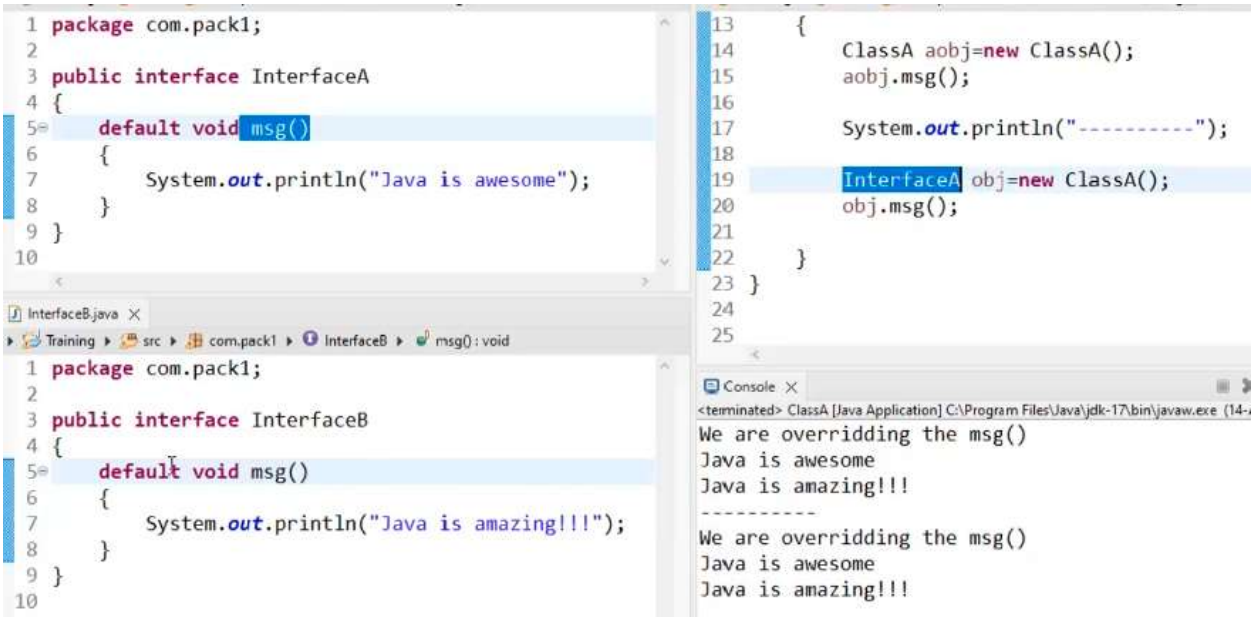
```
<terminated> ClassA [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (14-Apr-2025, 7:57:20 am – 7:57:20 a
We are overridding the msg()
Java is awesome
Java is amazing!!!
```



Note

Multiple inheritance in java is not supported thought classes, but we can achieve this by using interfaces.

Wherever we are inheriting multiple interfaces into a class, if both the interfaces are having same method name, we can resolve that ambiguity by overriding the ambiguous method. We can call our parent interface methods by using super keyword.



```java
package com.pack1;

public interface InterfaceA
{
    default void msg()
    {
        System.out.println("Java is awesome");
    }
}
```

InterfaceB.java

Training ▸ src ▸ com.pack1 ▸ InterfaceB ▸ msg() : void

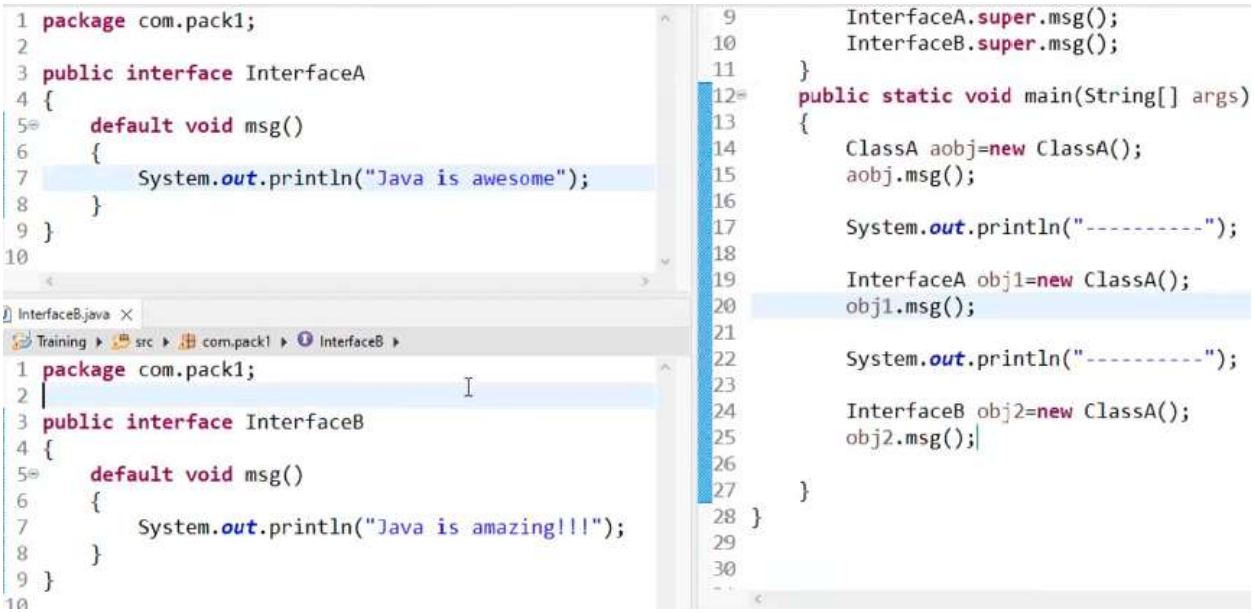
```java
package com.pack1;

public interface InterfaceB
{
    default void msg()
    {
        System.out.println("Java is amazing!!!");
    }
}
```

```java
    {
        ClassA aobj=new ClassA();
        aobj.msg();

        System.out.println("----------");

        InterfaceA obj=new ClassA();
        obj.msg();

    }
}
```

Console

<terminated> ClassA [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe (14-

```
We are overridding the msg()
Java is awesome
Java is amazing!!!
----------
We are overridding the msg()
Java is awesome
Java is amazing!!!
```



```java
package com.pack1;

public interface InterfaceA
{
    default void msg()
    {
        System.out.println("Java is awesome");
    }
}
```

InterfaceB.java

Training ▸ src ▸ com.pack1 ▸ InterfaceB ▸

```java
package com.pack1;

public interface InterfaceB
{
    default void msg()
    {
        System.out.println("Java is amazing!!!");
    }
}
```

```java
        InterfaceA.super.msg();
        InterfaceB.super.msg();
    }
    public static void main(String[] args)
    {
        ClassA aobj=new ClassA();
        aobj.msg();

        System.out.println("----------");

        InterfaceA obj1=new ClassA();
        obj1.msg();

        System.out.println("----------");

        InterfaceB obj2=new ClassA();
        obj2.msg();

    }
}
```

Functional interface

An interface which has only one abstract method is known as a functional interface.

Inside a functional interface we can write any number of default methods, static methods, private methods including main method but there should be only one abstract method.

Functional interfaces concept is being introduced in java-1.8 version

Examples

1. Runnable interface

2. Consumer interface

3. Supplier interface

4.Pridicate interface

5. Comparable interface

6. Comparator interface...etc.

```
1  package com.pack1;
2
3  @FunctionalInterface
4  public interface InterfaceA
5  {
6      void meth1();
7
8      void meth2();
9  }
10
```

```java
package com.pack1;

@FunctionalInterface
public interface InterfaceA
{
    void meth1();
}
```

Wait for implementation of functional interface in coming classes

Marker interface

An interface without any methods or variables is known as a marker interface.

It is an empty interface.

Whenever a class is inheriting this marker interface that class will achieve some special properties.

examples

1. Serializable interface
2. Cloneable interface

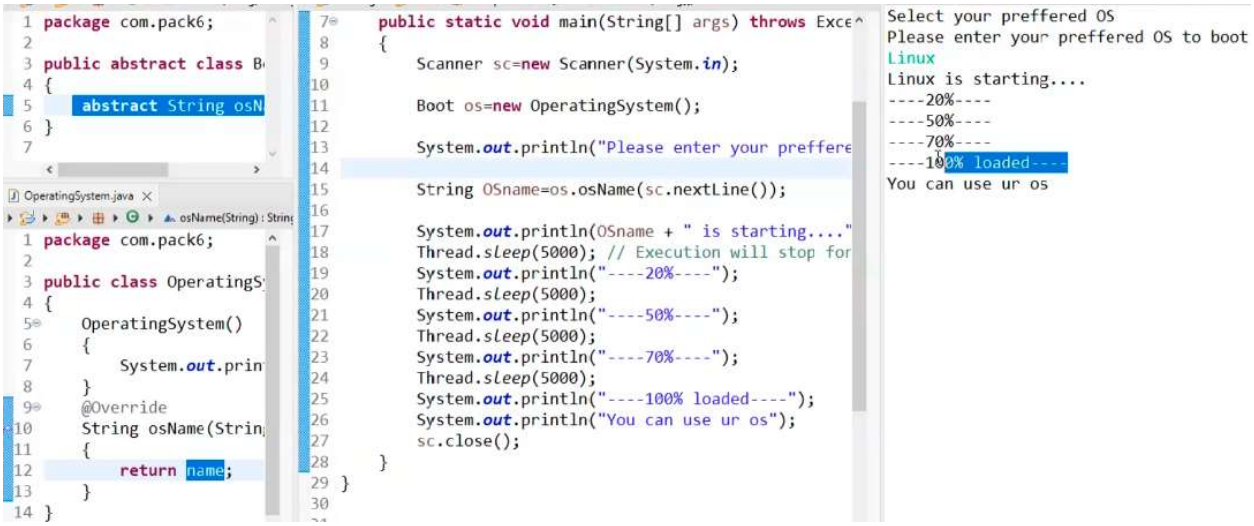
Overall oops small example

```java
package com.pack6;

public abstract class Boot
{
    abstract String osName(String name);
}

```

```java
package com.pack6;

public abstract class Boot
{
    abstract String osName(String name);
}

```

```java
package com.pack6;

public class OperatingSystem extends Boot
{
    OperatingSystem()
    {
        System.out.println("Select your preffered OS");
    }
    @Override
    String osName(String name)
    {
        return name;
    }
}

```

```java
package com.pack6;

import java.util.Scanner;

public class Start extends OperatingSystem
{
    public static void main(String[] args) throws Exception
    {
        Scanner sc=new Scanner(System.in);

        Boot os=new OperatingSystem();

        System.out.println("Please enter your preffered OS to boot");

        String OSname=os.osName(sc.nextLine());

        System.out.println(OSname + " is starting....");
        Thread.sleep(5000); // Execution will stop for 5 sec
        System.out.println("----20%----");
        Thread.sleep(5000);
        System.out.println("----50%----");
        Thread.sleep(5000);
        System.out.println("----70%----");
        Thread.sleep(5000);

        System.out.println("----100% loaded----");
        System.out.println("You can use ur os");
        sc.close();
    }
}
```



1. Encapsulation

Definition: Wrapping data (variables) and methods (functions) into a single unit (class) and restricting direct access to some of the object's components.

Uses:

- Protects internal object state.
- Promotes modularity — changes to one part don't affect others.
- Allows data hiding using access modifiers like private, protected, public.
- Example:

```
class Student {
   private int age;  // cannot be accessed directly outside

   public void setAge(int age) {
      this.age = age;
   }

   public int getAge() {
```

```
    return age;
  }
}
```

---

2. Inheritance

Definition: The process by which one class (child/subclass) inherits properties and behaviors from another class (parent/superclass).

Uses:

- Reusability of code.
- Hierarchical classification (e.g., Dog is an Animal).
- Reduces redundancy.
- Example:

```
class Animal {
  void sound() {
    System.out.println("Animal makes sound");
  }
}
```

```
class Dog extends Animal {
   void bark() {
      System.out.println("Dog barks");
   }
}
```

---

3. Polymorphism

Definition: The ability of one function or object to behave in different ways based on context.

Types:

- **Compile-time (method overloading)**
- **Run-time (method overriding)**

Uses:

- Code flexibility and maintainability.
- Improves readability and reusability.
- Example (overriding):

```
class Animal {
   void sound() {
      System.out.println("Animal sound");
   }
}

class Cat extends Animal {
   void sound() {
      System.out.println("Cat meows");
   }
}
```

---

4. Abstraction

Definition: Hiding complex implementation details and showing only the necessary features of an object.

Uses:

- Reduces complexity for users.
- Focuses on essential aspects.
- Achieved via abstract classes or interfaces.

- Example:

```
abstract class Shape {
   abstract void draw();  // only declares, doesn't define
}

class Circle extends Shape {
   void draw() {
      System.out.println("Drawing Circle");
   }
}
```

Summary Table:

Concept	Purpose	Benefit
Encapsulation	Hide internal state	Protects data, improves security
Inheritance	Reuse code from parent class	Reduces redundancy, enables hierarchy
Polymorphism	One interface, many implementations	Flexibility, cleaner code
Abstraction	Show only relevant features, hide details	Reduces complexity, improves clarity

Assignment

Assignment: Create a Library Management System

Write a Java program that models a simple library management system.
Your system should have the following classes:

1. **Book Class**: Represents a book with attributes such as title, author, ISBN, and availability status.

2. **Library Class**: Represents the library itself, which contains an array of books.

3. **Member Class**: Represents a library member with attributes like name, ID, and borrowed books.

4. **LibraryApp Class**: Main class to run the program.

Your program should allow the following functionalities:

1. Add a book to the library.
2. Display all available books.
3. Display all borrowed books.
4. Allow a member to borrow a book (if available).
5. Allow a member to return a book.
6. Display all members and the books they've borrowed.