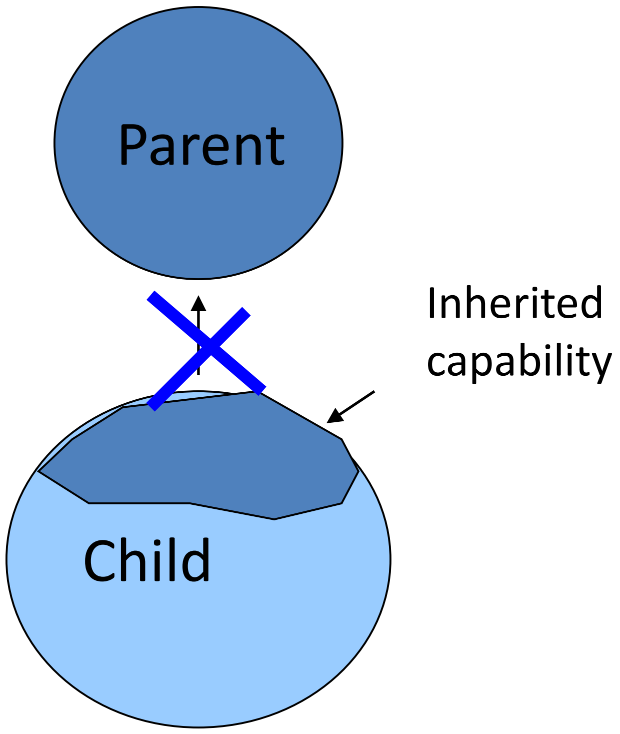
Final ,Abstract classes and Interfaces

Restricting Inheritance



Final Members: A way for Preventing Overriding of Members in Subclasses

- All methods and variables can be overridden by default in subclasses.
- This can be prevented by declaring them as final using the keyword “final” as a modifier. For example:
  - final int marks = 100;
  - final void display();
- This ensures that functionality defined in this method cannot be altered any. Similarly, the value of a final variable cannot be altered.

Final Classes: A way for Preventing Classes being extended

- We can prevent an inheritance of classes by other classes by declaring them as final classes.
- This is achieved in Java by using the keyword final as follows:

```
final class Marks
{ // members
}
final class Student extends Person
{ // members
}
```

- Any attempt to inherit these classes will cause an error.

Abstract Method and Classes :

We can indicate that a method must always be redefined in a subclass, thus making overriding compulsory.

This is done using the modifier keyword abstract in the method definition .

When a class contains atleast one abstract method then that class should also be defined as abstract.

We cannot use abstract classes to instantiate objects directly.

The abstract methods of an abstract class must be redefined in its subclass.

We cannot declare abstract constructors or abstract static methods.

Abstract Class Syntax

```
abstract class ClassName
{
            …
            …
            abstract Type MethodName1();
            …
            …
            Type Method2()
            {
               // method body
            }
}
```

```
abstract class PgCourse
{
    float intpassmin=12.5;
    float extpassmin=37.5;
    abstract void Test();
    abstract void report();
    void cut-off()
     {
        int cut_mark=40;
      System.out.println("The cut-off value is"+cut_mark);
     }
  }

PgCourse mca=new PgCourse(); //error
```

```
class Mca extends Pgcourse
 {
   void Test()
   {
      //Here, Method body should be fully defined,
      //Otherwise declare this method also as  abstract
   }
   void report()
    {
     //Define the method body of this method too
    }
      //we can also override cut-off() even though it is already defined
}
```

Abstract Classes Properties

- A class with one or more abstract methods is automatically abstract and it cannot be instantiated.
- A class declared abstract, even with no abstract methods can not be instantiated.
- A subclass of an abstract class can be instantiated if it overrides all abstract methods by implementation them.
- A subclass that does not implement all of the superclass abstract methods is itself abstract; and it cannot be instantiated.
- Abstract classes can also be extended

Interfaces

Design Abstraction and a way for realizing Multiple Inheritance

interface

- An **interface** in java is a blueprint of a class.
- It has static constants and abstract methods.
- It cannot be instantiated just like abstract class.
- It is used to achieve abstraction and multiple inheritance in Java.
- Since Java 8, Why use Java interface? default and static methods.
    - It is used to achieve abstraction.
    - By interface, we can support the functionality of multiple inheritance.

Interfaces

- *Interface* is a conceptual entity similar to a Abstract class.
- Can contain only constants (final variables) and abstract method (no implementation) - Different from Abstract classes.
- Use when a number of classes share a common interface.
- Each class should implement the interface.
- You cannot instantiate an interface.
- An interface does not contain any constructors.

Interfaces: An informal way of realising multiple inheritance

- An interface is basically a kind of class—it contains methods and variables, but they have to be only abstract classes and final fields/variables.
- Therefore, it is the responsibility of the class that implements an interface to supply the code for methods.
- A class can implement any number of interfaces, but cannot extend more than one class at a time.
- Therefore, interfaces are considered as an informal way of realizing multiple inheritance in Java.

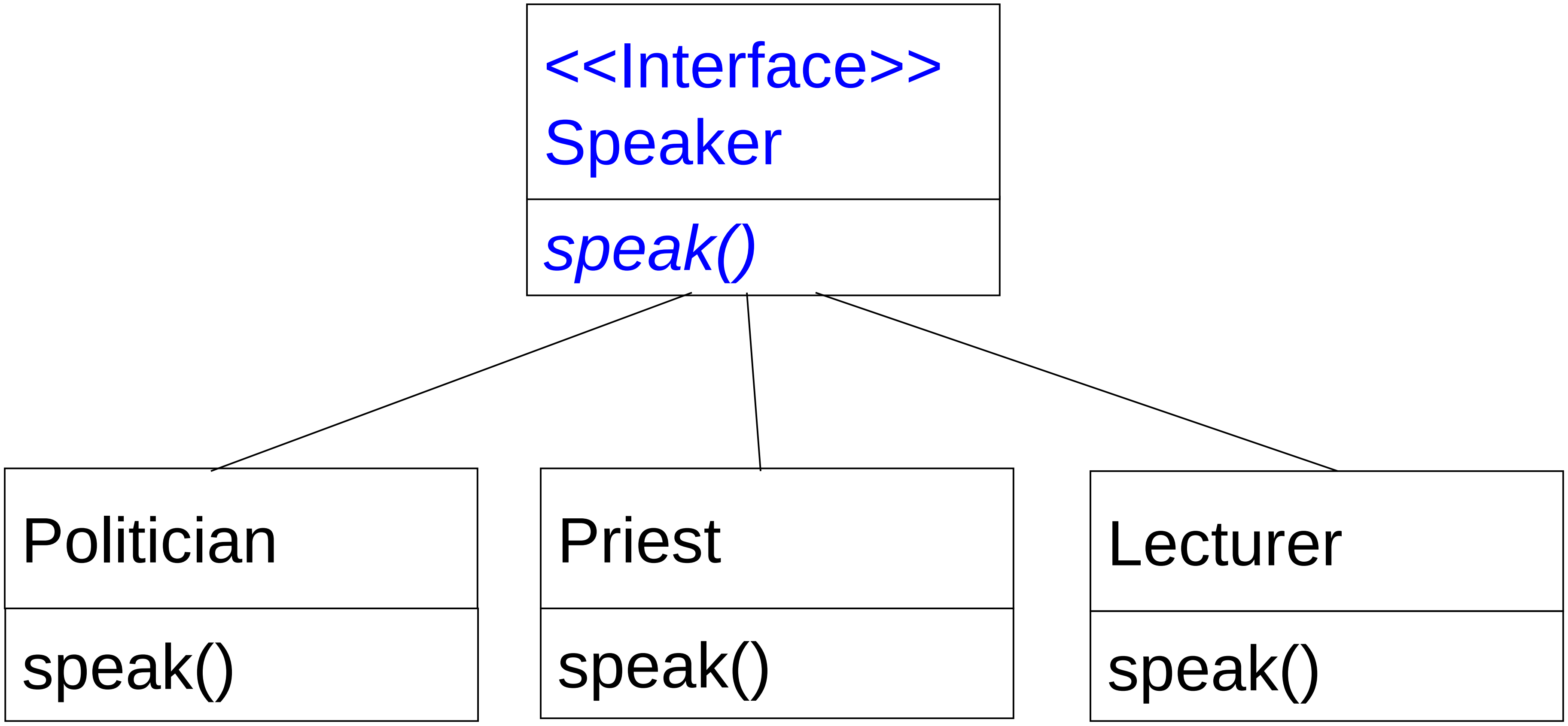
Built-in Interfaces

- The Java standard library includes lots more built-in interfaces
  - they are listed in the API with the classes
- Examples:
  - `Clonable` – implements a `clone()` method
  - `Formattable` – can be formatted with `printf`

Properties of **interfaces**

- The **interface** keyword is used to declare an interface.
- Interfaces have the following properties:
  - An interface is implicitly abstract. We do not need to use the **abstract** keyword when declaring an interface.
  - Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
  - **Methods in an interface are implicitly public**.
  - **Variable in an interface are implicitly public, static & final** (and have to be assigned values there).

Interface - Example



Interfaces Definition

- Syntax (appears like abstract class):

```
interface InterfaceName {
    // Constant/Final Variable Declaration
    // Methods Declaration – only method body
}
```

- Example:

```
interface Speaker {
    public void speak( );
}
```

Implementing Interfaces

- Interfaces are used like super-classes who properties are inherited by classes. This is achieved by creating a class that implements the given interface as follows:

```
class ClassName implements InterfaceName [, InterfaceName2, …]
{
    // Body of Class
}
```

Implementing Interfaces Example

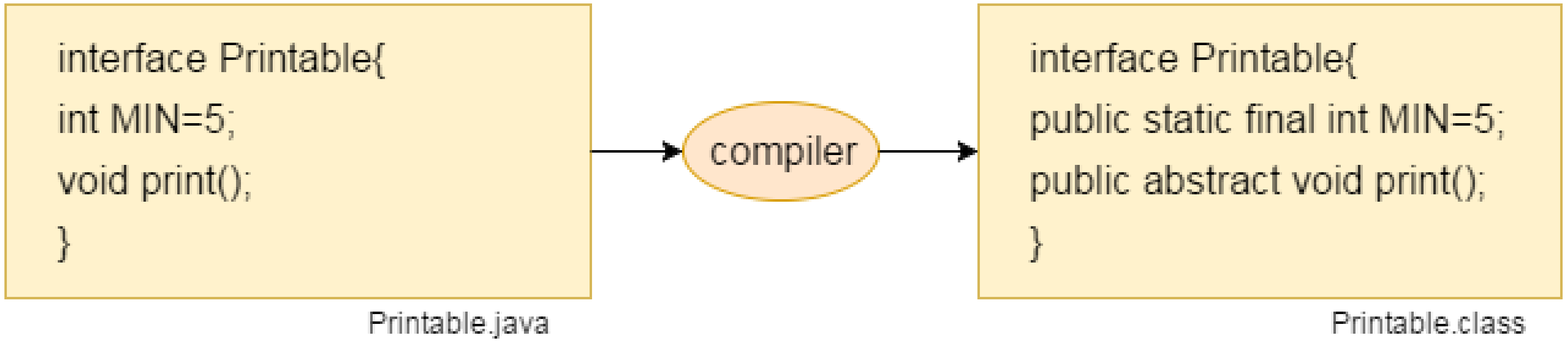
```
class  Politician implements Speaker {
        public void speak(){
            System.out.println(“Talk politics”);
        }
}
```

```
class  Priest implements Speaker {
        public void speak(){
            System.out.println(“Religious Talks”);
        }
}
```

```
class  Lecturer implements Speaker {
        public void speak(){
            System.out.println(“Talks Object Oriented Design and Programming!”);
        }
}
```

interface

- **interface** fields are public, static and final by default, and methods are public and



```
interface Printable{
int MIN=5;
void print();
}
```

Printable.java

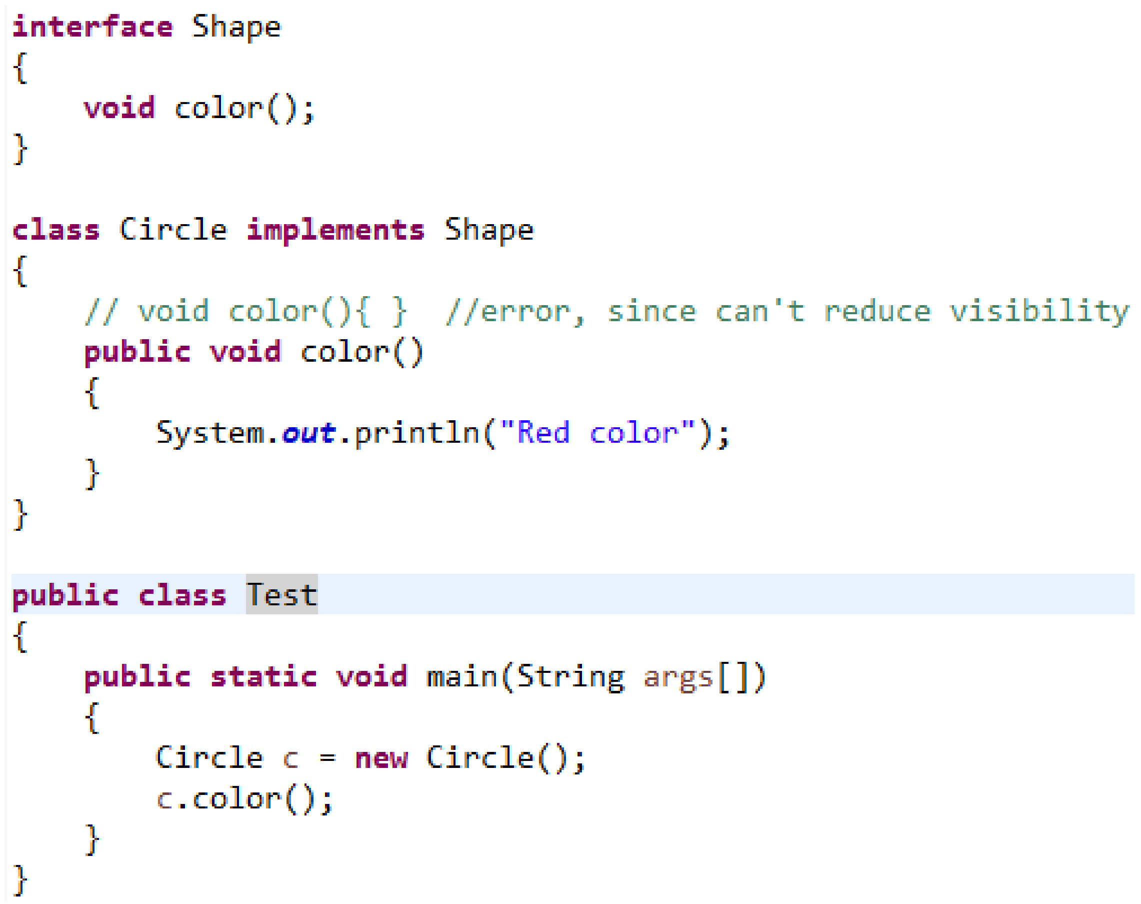
```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

Printable.class

Implementing interfaces

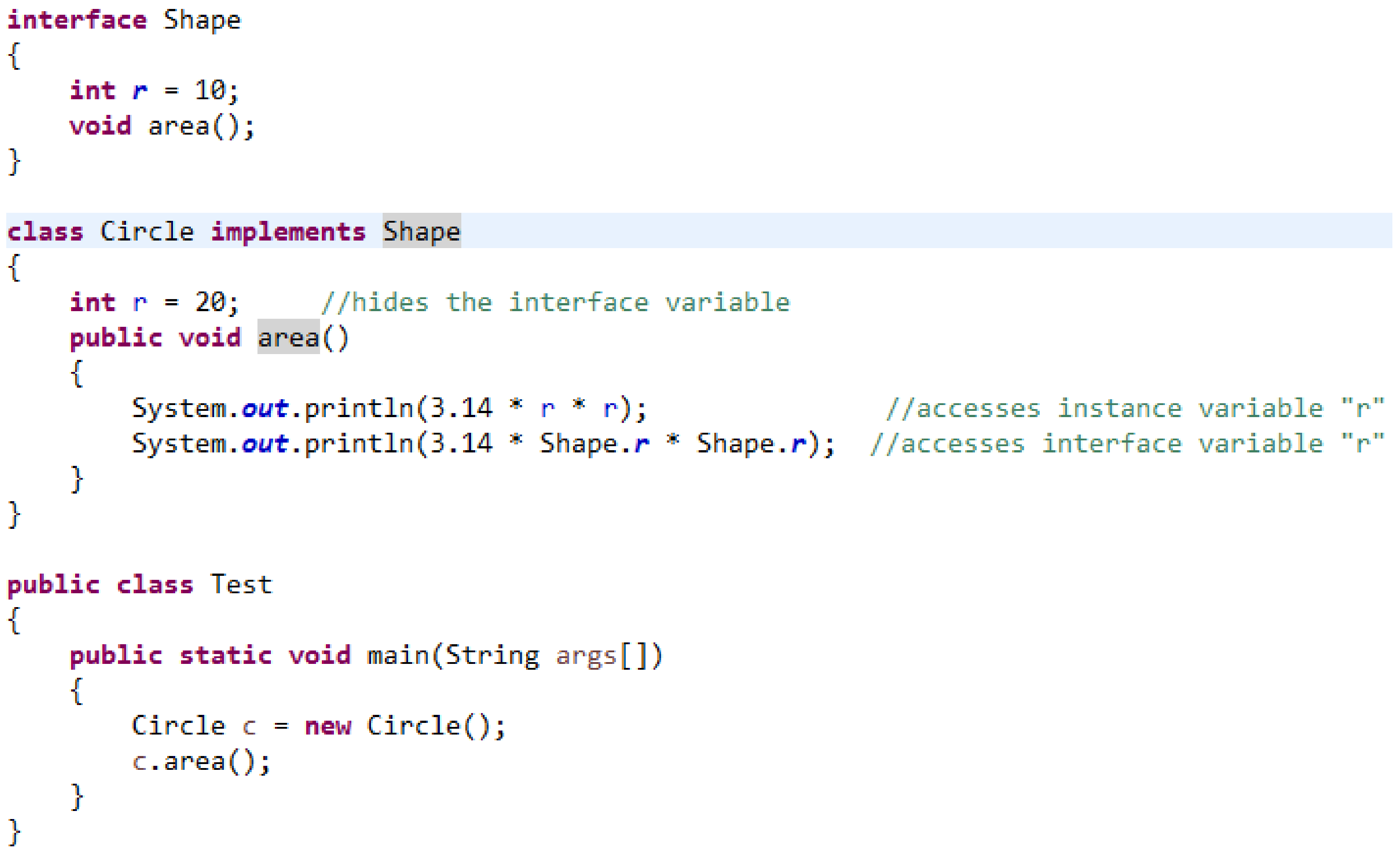
- When a class implements an interface, then it has to perform the specific behaviors of the interface.
- If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.
- A class uses the **implements** keyword to implement an interface.
- The implements keyword appears in the class declaration following the extends portion of the declaration.

Example -1 (implementing interface)



```java
interface Shape
{
    void color();
}

class Circle implements Shape
{
    // void color(){ }  //error, since can't reduce visibility
    public void color()
    {
        System.out.println("Red color");
    }
}

public class Test
{
    public static void main(String args[])
    {
        Circle c = new Circle();
        c.color();
    }
}
```

Example (members accessibility)

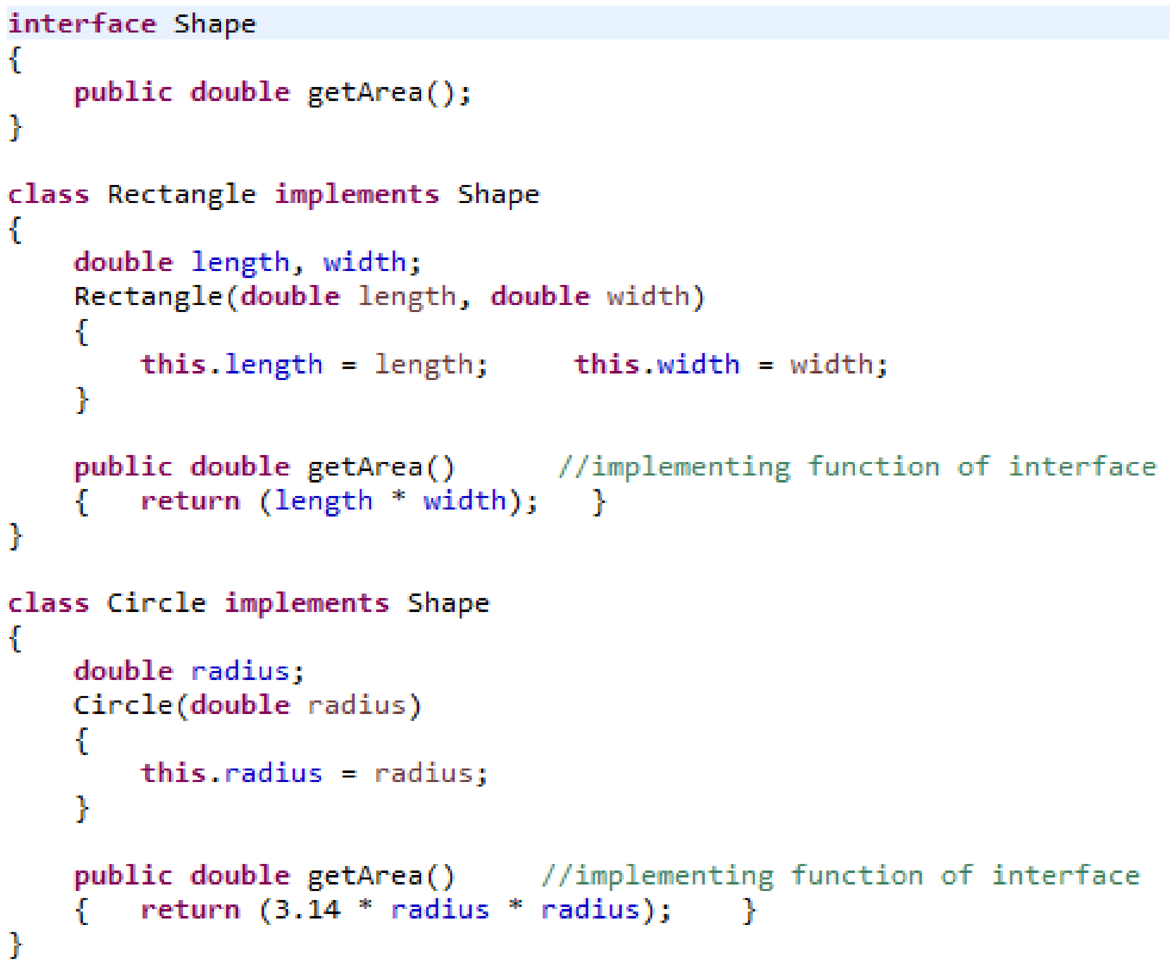


```
interface Shape
{
    int r = 10;
    void area();
}

class Circle implements Shape
{
    int r = 20;     //hides the interface variable
    public void area()
    {
        System.out.println(3.14 * r * r);                //accesses instance variable "r"
        System.out.println(3.14 * Shape.r * Shape.r);   //accesses interface variable "r"
    }
}

public class Test
{
    public static void main(String args[])
    {
        Circle c = new Circle();
        c.area();
    }
}
```

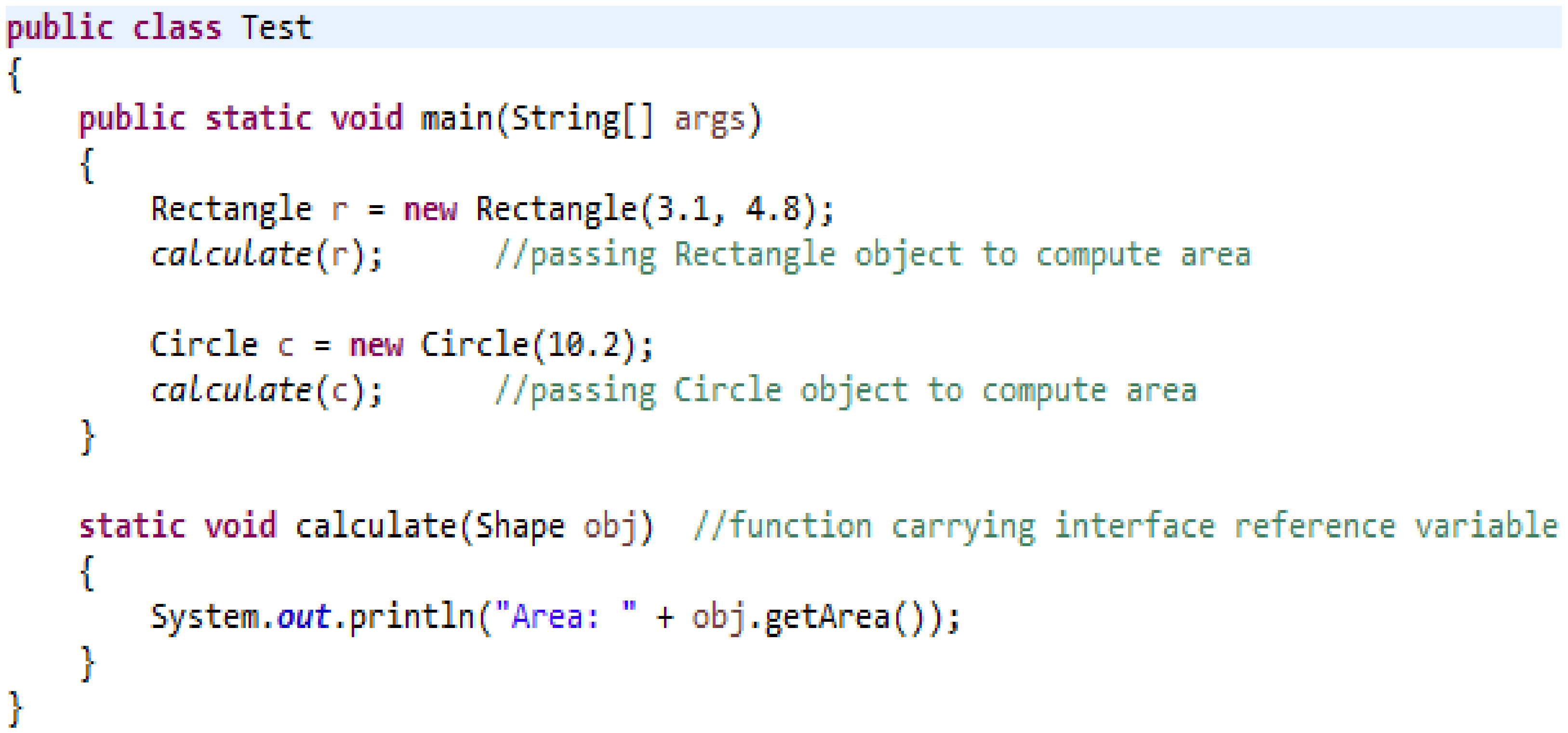
Example (Bank)

```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

Example (Shape)



```
interface Shape
{
    public double getArea();
}

class Rectangle implements Shape
{
    double length, width;
    Rectangle(double length, double width)
    {
        this.length = length;     this.width = width;
    }

    public double getArea()      //implementing function of interface
    {   return (length * width);    }
}

class Circle implements Shape
{
    double radius;
    Circle(double radius)
    {
        this.radius = radius;
    }

    public double getArea()     //implementing function of interface
    {   return (3.14 * radius * radius);    }
}
```



```java
public class Test
{
    public static void main(String[] args)
    {
        Rectangle r = new Rectangle(3.1, 4.8);
        calculate(r);      //passing Rectangle object to compute area

        Circle c = new Circle(10.2);
        calculate(c);      //passing Circle object to compute area
    }

    static void calculate(Shape obj)  //function carrying interface reference variable
    {
        System.out.println("Area: " + obj.getArea());
    }
}
```

Example-2

- interface IAccount
```
{
   //public static final
   float rate=6.0;
  //public abstract
   double getBalance();

   void deposit(double amount);

   void withdraw(double amount);
}
```

```
class HDFCAccount implements IAccount
    {
        double deposits;
        double withdrawals;


        public double getBalance()
        {
            return deposits - withdrawals;
        }

        public void deposit(double amount)
        {
            deposits += amount;
        }

        public void withdraw(double amount)
        {
            withdrawals += amount;
        }
    }
```

```
class StateBankAccount implements IAccount
    {
        double balance;

        public double getBalance()
        {
            return balance;
        }

        public void deposit(double amount)
        {
            balance += amount;
        }

        public void withdraw(double amount)
        {
            balance -= amount;
        }
    }
```

Object Creation

```
interface Mail
{
      void register();
      void validation();
}
```

✗

new Mail();

```
abstract class Car
{
      void door();
      void glass();
}
```

✗

new Car();

Object Creation

```
interface Mail
{
        void register();
        void validation();
}
```

✓

new Yahoo();

```
abstract class Car
{
        void door();
        void glass();
}
```

✓

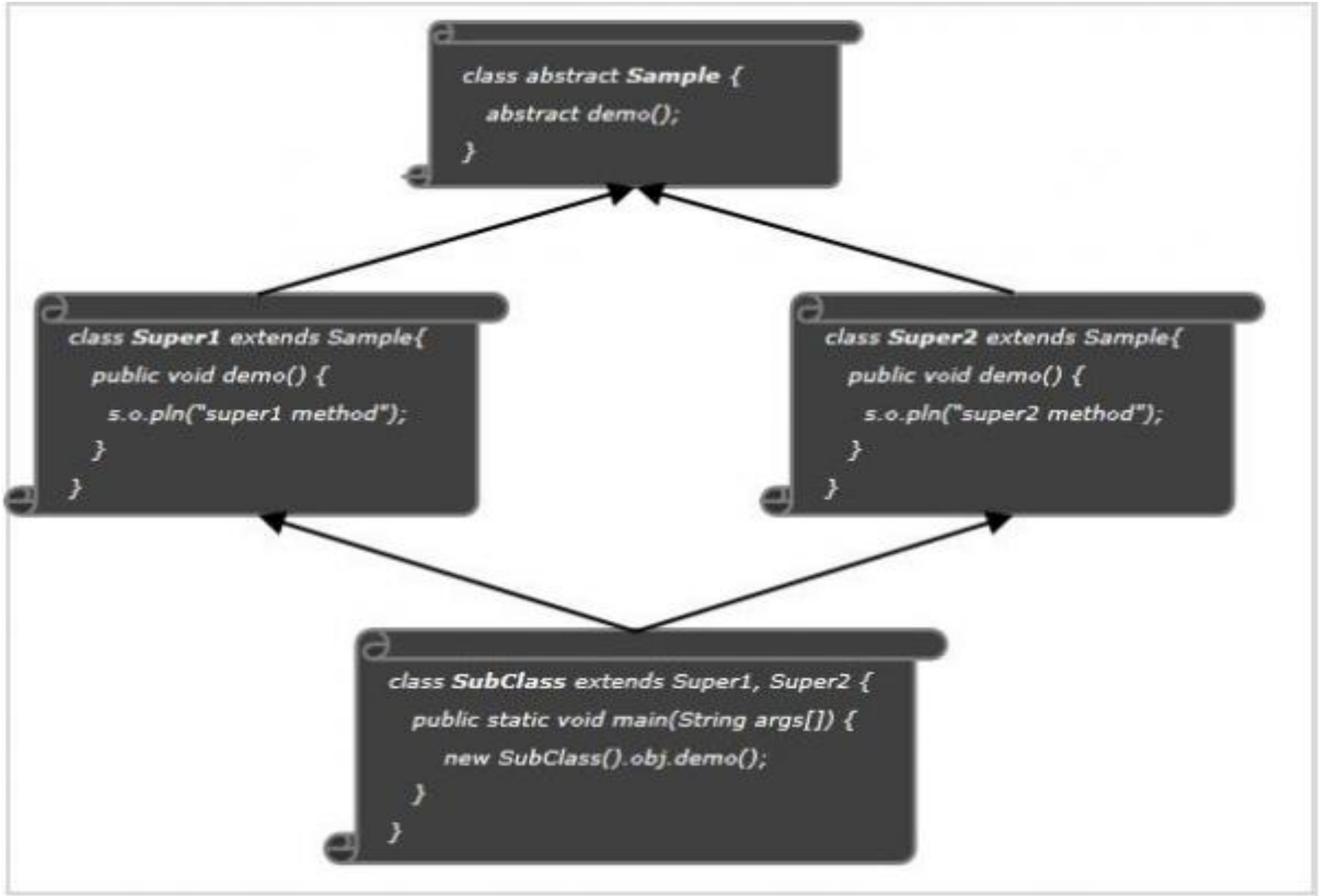
new Benz();

Object Creation

```
 ✓               ✓
Mail ob1 = new Yahoo();
```

```
 ✓         ✓
Car c1 = new Benz();
```

Diamond Problem in Java



```
class abstract Sample {
  abstract demo();
}
```

```
class Super1 extends Sample{
  public void demo() {
    s.o.pln("super1 method");
  }
}
```

```
class Super2 extends Sample{
  public void demo() {
    s.o.pln("super2 method");
  }
}
```

```
class SubClass extends Super1, Super2 {
  public static void main(String args[]) {
     new SubClass().obj.demo();
  }
}
```

interface vs. **class**

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte-code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding byte-code file must be in a directory structure that matches the package name.

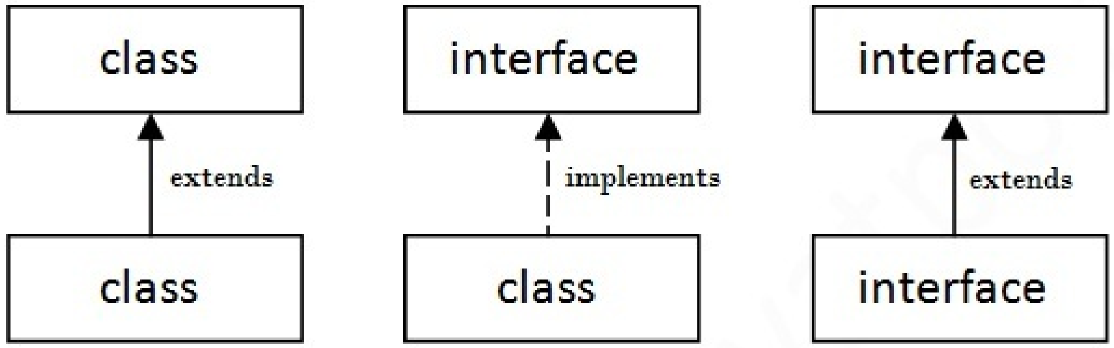
interface vs. **class**

An interface is different from a class in several ways, including:

- We cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

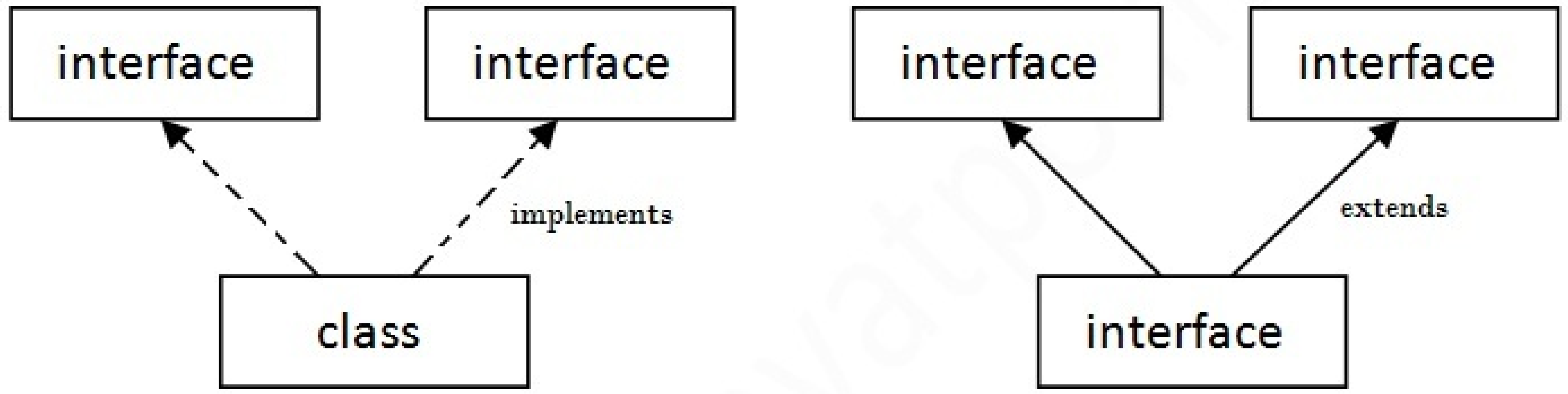
Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

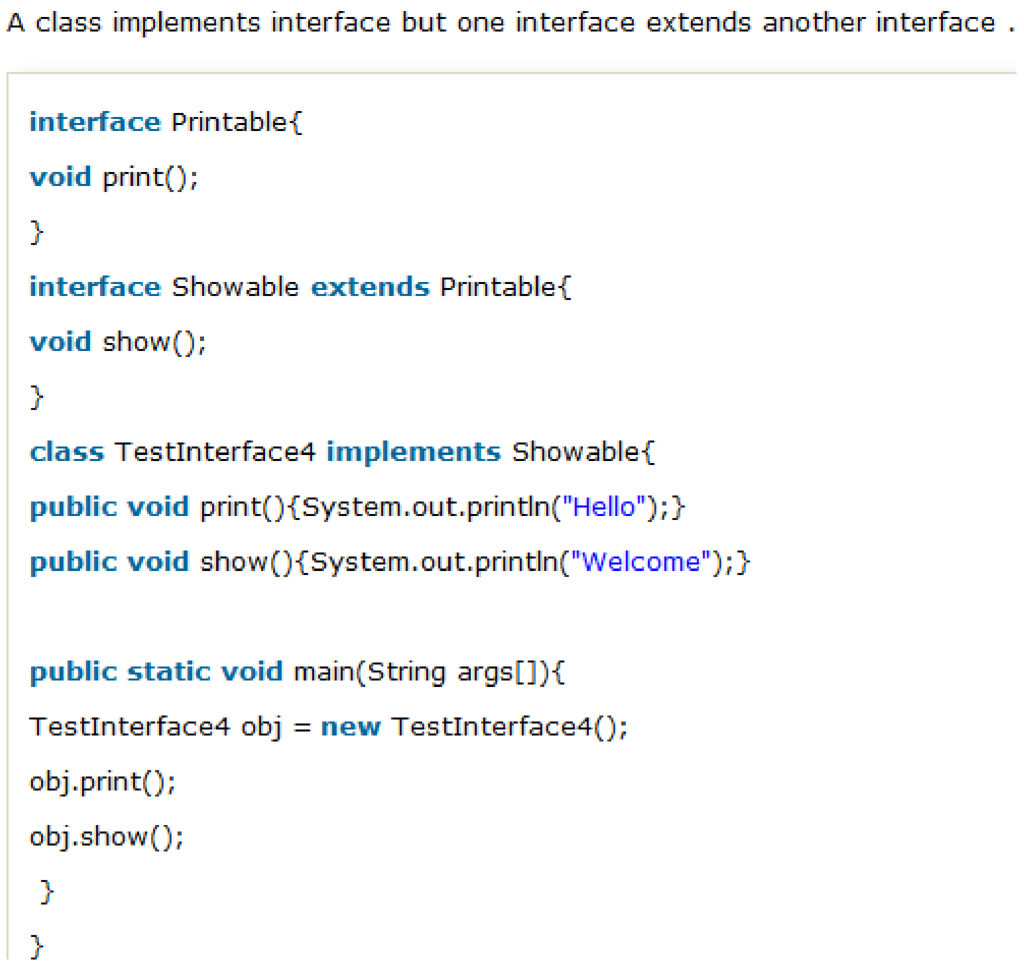


Multiple Inheritance in Java

Example (multiple inheritance using interfaces)

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Example (inheritance in interfaces)



A class implements interface but one interface extends another interface .

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated. But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

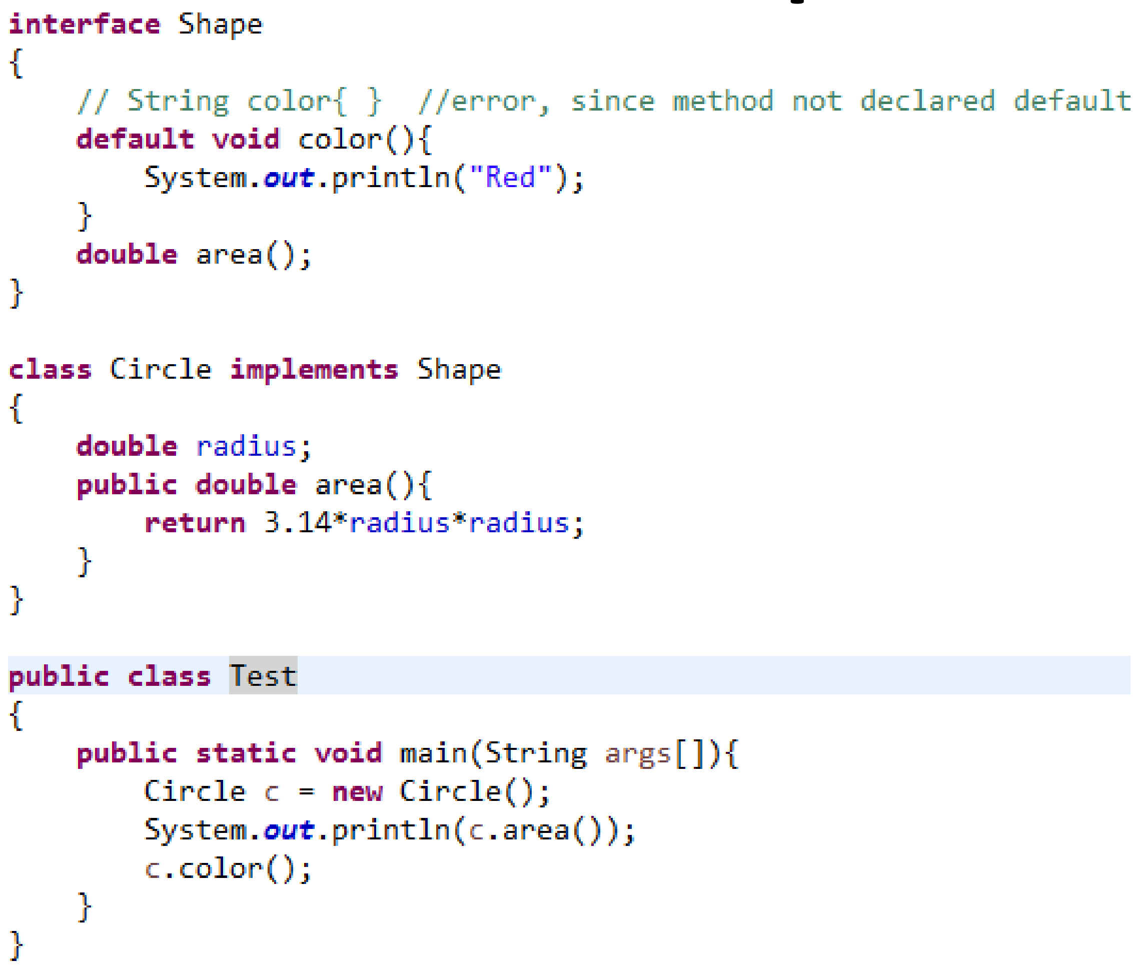
6) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Default method in interface

Default methods

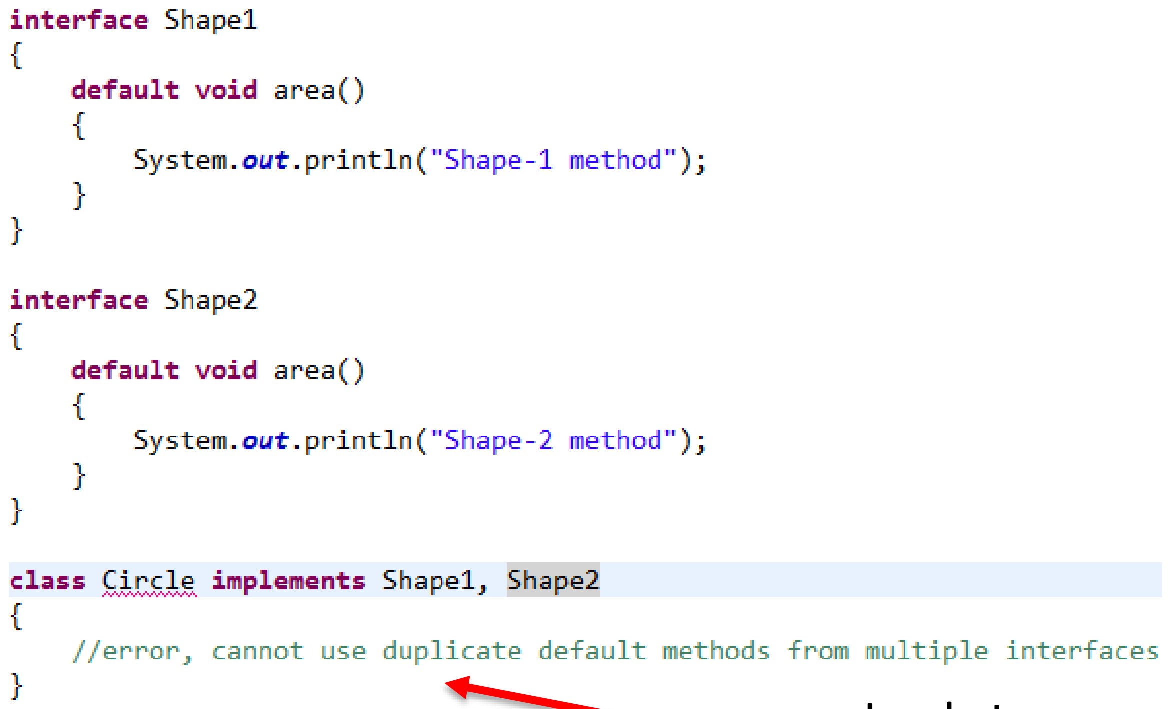
- Default methods are defined inside the interface and tagged with **default** are known as default methods.
- These methods are non-abstract methods.
- Since Java 8, we can have method body in interface. But we need to make it **default**.

Example



```
interface Shape
{
    // String color{ }  //error, since method not declared default
    default void color(){
        System.out.println("Red");
    }
    double area();
}

class Circle implements Shape
{
    double radius;
    public double area(){
        return 3.14*radius*radius;
    }
}

public class Test
{
    public static void main(String args[]){
        Circle c = new Circle();
        System.out.println(c.area());
        c.color();
    }
}
```

Example (multiple inheritance using interfaces with default methods)



```
interface Shape1
{
    default void area()
    {
        System.out.println("Shape-1 method");
    }
}

interface Shape2
{
    default void area()
    {
        System.out.println("Shape-2 method");
    }
}

class Circle implements Shape1, Shape2
{
    //error, cannot use duplicate default methods from multiple interfaces
}
```

In order to remove ambiguity, override the method in the class as:

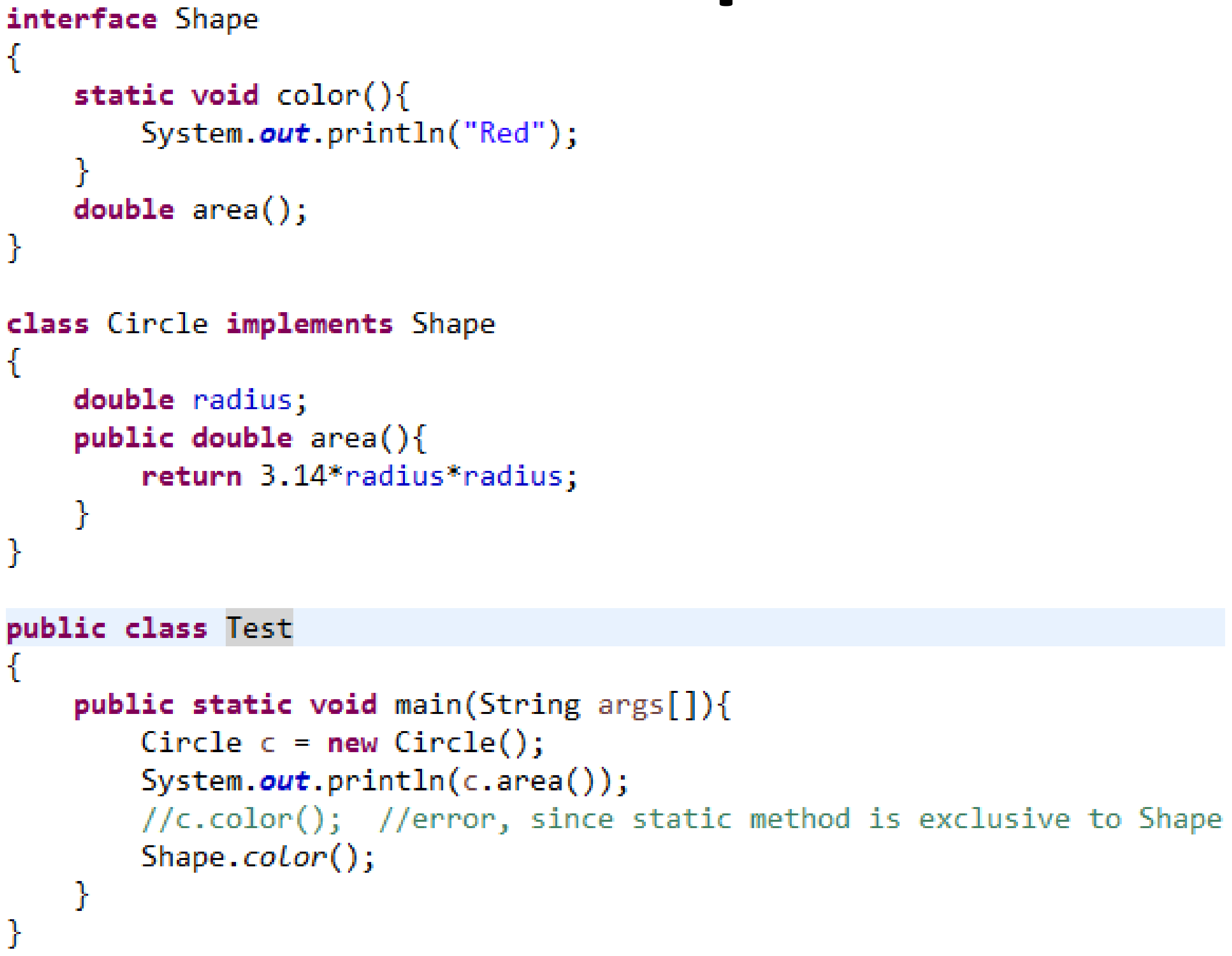
```
public void area()
{
    Shape1.super.area();//to call method of Shape1
}
```

static method in interface

static method

- Since Java 8, you can define static methods in interfaces.
- Java interface static method is part of interface, we can’t use/access it using class objects.
- Java interface static method helps us in providing security by not allowing implementation classes to override them.

Example



```
interface Shape
{
    static void color(){
        System.out.println("Red");
    }
    double area();
}

class Circle implements Shape
{
    double radius;
    public double area(){
        return 3.14*radius*radius;
    }
}

public class Test
{
    public static void main(String args[]){
        Circle c = new Circle();
        System.out.println(c.area());
        //c.color();  //error, since static method is exclusive to Shape
        Shape.color();
    }
}
```

Example (multiple inheritance using interfaces with static methods)

```java
interface Shape1
{
    static void area()
    {
        System.out.println("Shape-1 method");
    }
}

interface Shape2
{
    static void area()
    {
        System.out.println("Shape-2 method");
    }
}
```

```java
class Circle implements Shape1, Shape2
{
    void fun()
    {
        //area();      //error, method not defined for Circle class
        Shape1.area();
        Shape2.area();
    }
}

public class Test
{
    public static void main(String args[])
    {
        Circle c = new Circle();
        c.fun();
    }
}
```

Extending Interfaces

- Like classes, interfaces can also be extended. The new sub-interface will inherit all the members of the superinterface in the manner similar to classes. This is achieved by using the keyword **extends** as follows:

```
interface InterfaceName2 extends InterfaceName1
{
    // Body of InterfaceName2
}
```

Inheritance and Interface Implementation

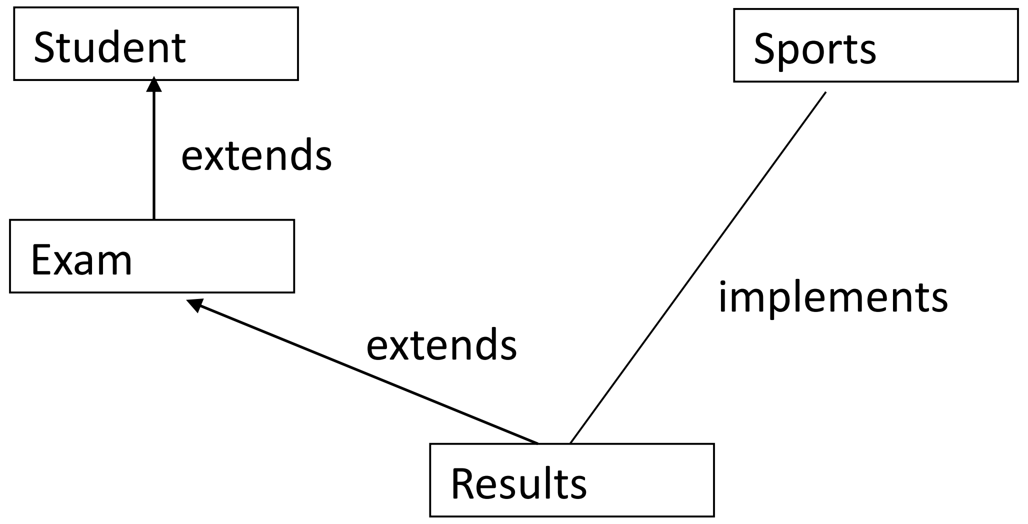
- A general form of interface implementation:

```
class ClassName extends SuperClass implements InterfaceName [,
InterfaceName2, ...]
{
    // Body of Class
}
```

- This shows a class can extended another class while implementing one or more interfaces. It appears like a multiple inheritance (if we consider interfaces as special kind of classes with certain restrictions or special features).

Student Assessment Example

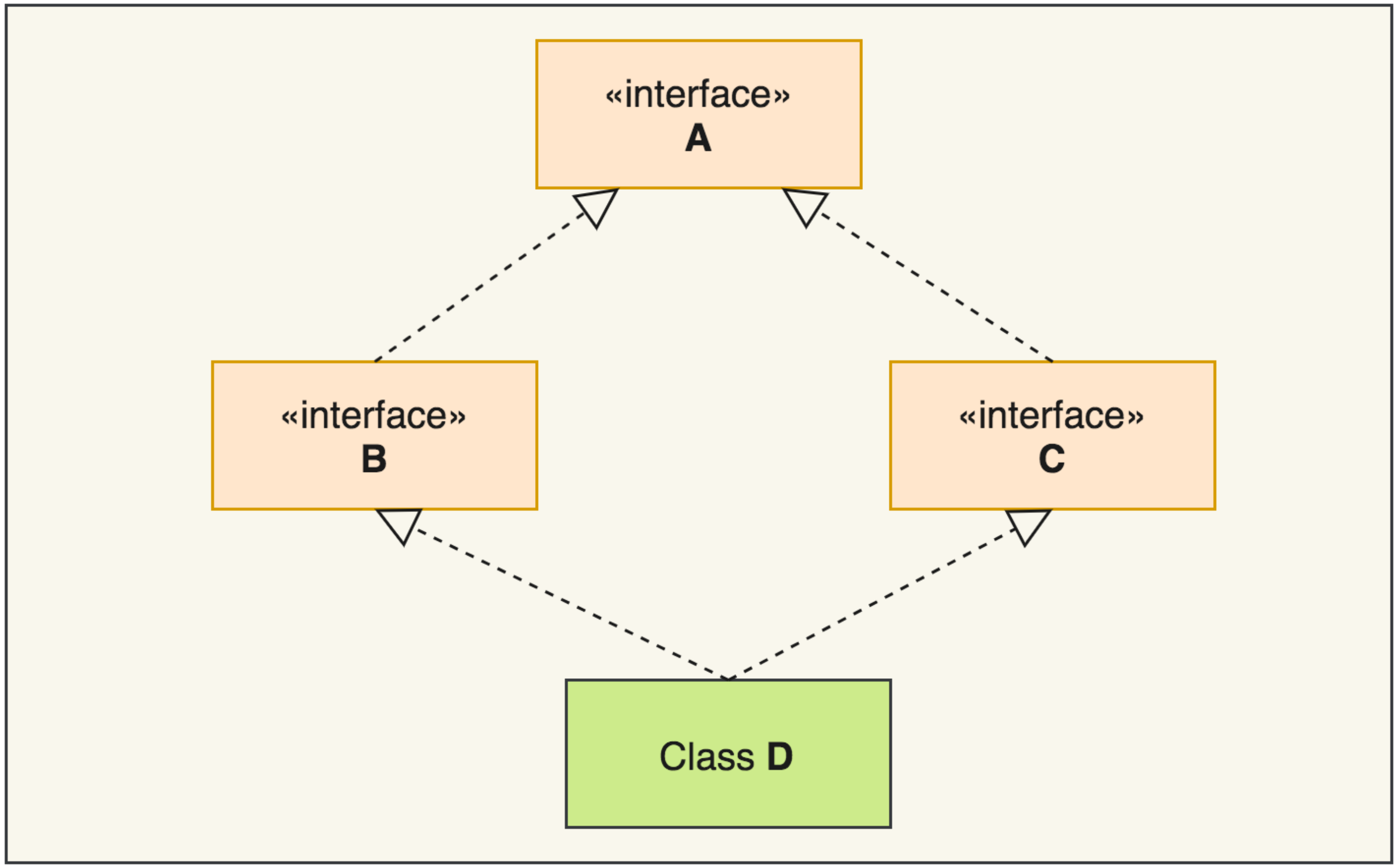
- Consider a university where students who participate in the national games or Olympics are given some grace marks. Therefore, the final marks awarded = Exam_Marks + Sports_Grace_Marks. A class diagram representing this scenario is as follow:



Software Implementation

```
class Student
{
      // student no and access methods
}
interface Sport
{
      // sports grace marks (say 5 marks) and abstract methods
}
class Exam extends Student
{
      // example marks (test1 and test 2 marks) and access methods
}
class Results extends Exam implements Sport
{
      // implementation of abstract methods of Sport interface
      // other methods to compute total marks = test1+test2+sports_grace_marks;
      // other display or final results access methods
}
```

Classic Diamond problem



Diamond Problem

- Before java 8 there was no diamond problem
- Ever since Java 8 introduced default and static methods in JDK 8, it's become possible to **define non-abstract methods in interfaces.**
- Since Java, one class can implement multiple interfaces and because there can be concrete methods in interfaces, the diamond problem has surfaced again.
- What will happen if two interfaces have methods o the same name and a Java class inherit from it?

Example-Diamond problem

```
interface MyInterface1{
  public static int num = 100;
  public default void display() {
    System.out.println("display method of MyInterface1");
  }
}
interface MyInterface2{
  public static int num = 1000;
  public default void display() {
    System.out.println("display method of MyInterface2");
  }
}
public class InterfaceExample implements MyInterface1, MyInterface2{
  public void display() {  //Implementing class must override
    MyInterface1.super.display();
    //or,
    MyInterface2.super.display();
  }
  public static void main(String args[]) {
    InterfaceExample obj = new InterfaceExample();
    obj.display();
  }
}
```

Output

```
display method of MyInterface1
display method of MyInterface2
```

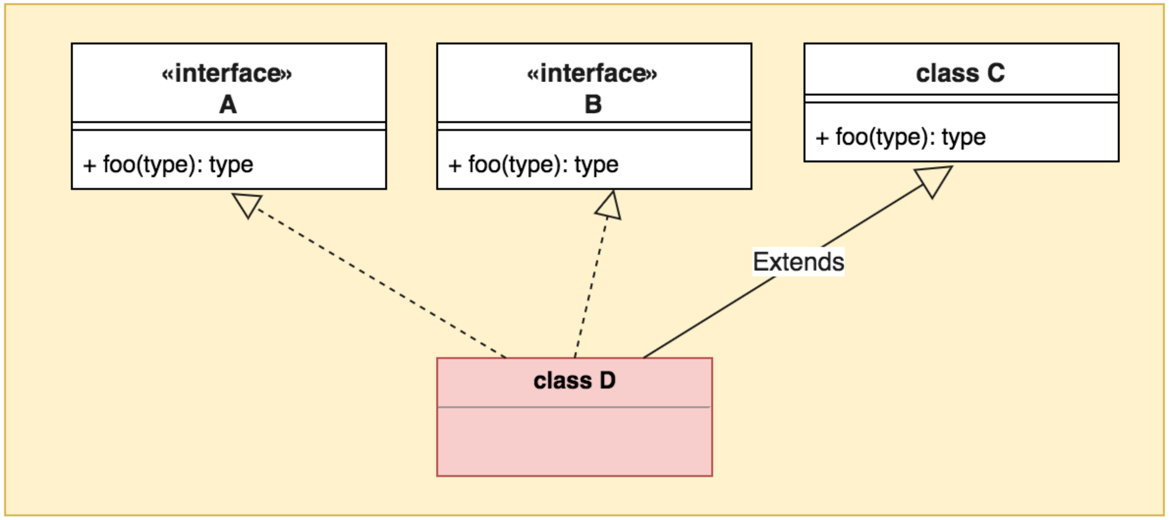
How does Java 8 tackles diamond problem of multiple inheritance?

- Java 8 brought a major change where interfaces can provide default implementation for its methods.
- Java designers kept in mind the diamond problem of inheritance while making this big change.
- There are clearly defined conflict resolution rules while inheriting default methods from interfaces using Java 8.

Rule 1

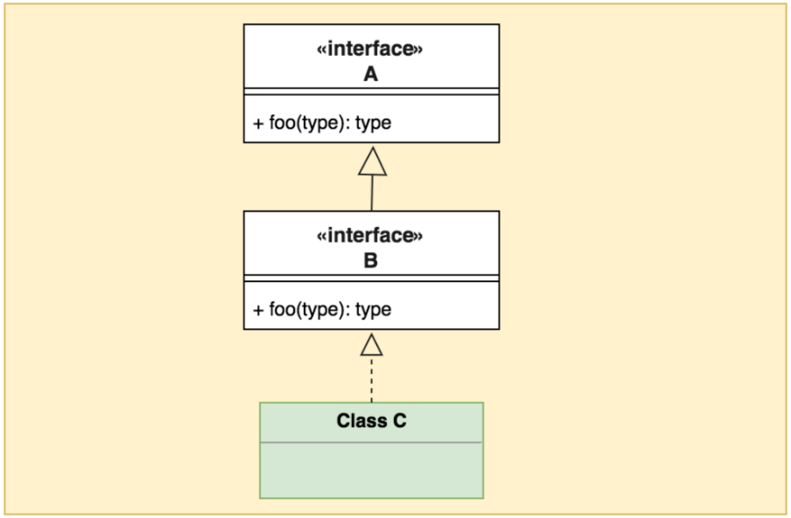
- Any method inherited from a class or a superclass is given higher priority over any default method inherited from an interface.

- class has higher precedence than interface default methods.
- In the diagram above, foo() method of class D will inherit from class C.



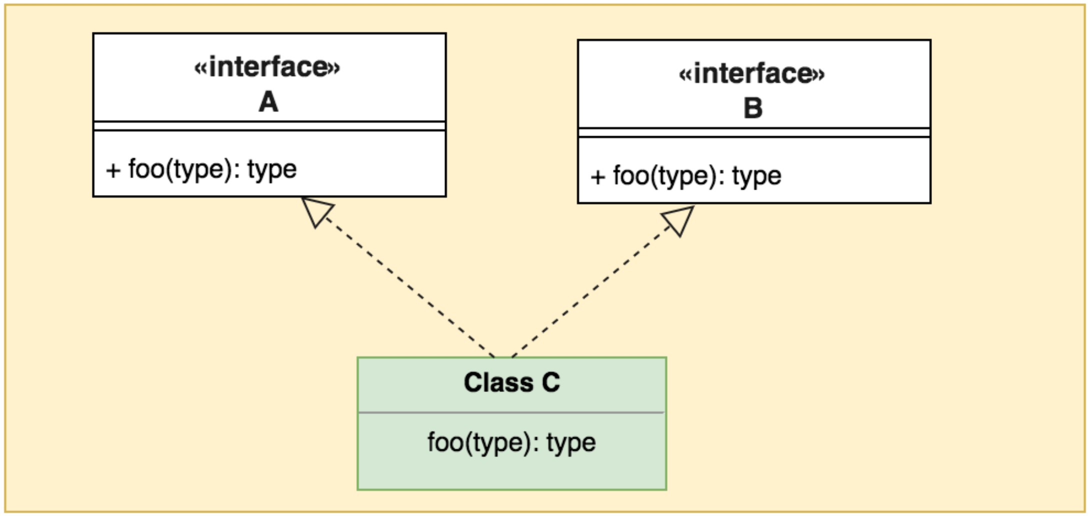
Rule 2

- Derived interfaces or sub-interfaces take higher precedence than the interfaces higher-up in the inheritance hierarchy.
- sub-interface has higher priority
- In the class diagram, foo() of class C will inherit from default method of interface B.



Rule 3

- In case Rule 1 and Rule 2 are not able to resolve the conflict then the implementing class has to specifically override and provide a method with the same method definition.
- class C must override foo method
- In class diagram, since interface A & B are at same level, to resolve conflict, class C must provide its own implementation by overriding method foo().



Interfaces and Software Engineering

- *Interfaces*, like abstract classes and methods, provide templates of behaviour that other classes are expected to implement.
- Separates out a design hierarchy from implementation hierarchy. This allows software designers to enforce/pass common/standard syntax for programmers implementing different classes.
- Pass method descriptions, not implementation
- Java allows for inheritance from only a single superclass. *Interfaces* allow for *class mixing*.
- Classes *implement* interfaces.

A Summary of OOP and Java Concepts Learned So Far

Summary

- *Class* is a collection of data and methods that operate on that data
- An *object* is a particular instance of a *class*
- *Object members* accessed with the ‘dot’ (Class.v)
- *Instance variables* occur in each instance of a class
- *Class variables* associated with a class
- Objects created with the *new* keyword

Summary

- Objects are not explicitly ‘freed’ or destroyed. Java automatically reclaims unused objects.
- Java provides a default constructor if none defined.
- A class may inherit the non-private methods and variables of another class by *subclassing*, declaring that class in its *extends* clause.
- *java.lang.Object* is the default *superclass* for a class. It is the root of the Java *hierarchy*.

Summary

- *Method overloading* is the practice of defining multiple methods which have the same name, but different argument lists
- *Method overriding* occurs when a class redefines a method inherited from its superclass
- *static, private*, and *final* methods cannot be overridden
- From a *subclass*, you can explicitly invoke an overridden method of the *superclass* with the *super* keyword.

Summary

- Data and methods may be hidden or encapsulated within a class by specifying the *private* or *protected* visibility modifiers.
- An abstract method has no *method body*. An abstract class contains abstract methods.
- An *interface* is a collection of *abstract methods* and constants. A class *implements* an interface by declaring it in its *implements* clause, and providing a method body for each *abstract method.*

Summary

- Inheritance promotes reusability by supporting the creation of new classes from existing classes.
- Various forms of inheritance can be realised in Java.
- Child class constructor can be directed to invoke selected constructor from parent using super keyword.
- Variables and Methods from parent classes can be overridden by redefining them in derived classes.
- New Keywords: extends, super, final