Packages and Interfaces

Acknowledgment
The contents of the slides are taken from the followings:
(a) Java- One Step Ahead by A. Seth and B.L. Juneja
(b) The Complete Reference by H. Schildt

Packages

- *Packages* are containers for classes.
- They are used to keep the class name space compartmentalized.
- You can define classes inside a package that are not accessible by code outside that package or class members that are only exposed to other members of the same package.
- This allows the classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

- Simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines name space in which classes are stored. If the package statement is omitted, the class names are in the default package, which has no name.

Define a package

- Simply include a package command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- The package statement defines name space in which classes are stored.
- If the package statement is omitted, the class names are in the default package, which has no name.
- Java uses file system directories to store packages. Any classes declared to be in a particular package have to be saved in the a directory named after the package.
- For example, classes that belong to the package MyPackage have to be in the directory called MyPackage.

package MyPackage;

- You can create a hierarchy of packages.
- The general form is package.pkg1.pkg2.pkg3
- A package hierarchy must also be reflected in the file system of your computer.
- For example, java.awt.image needs to be stored in java.awt.image directory.

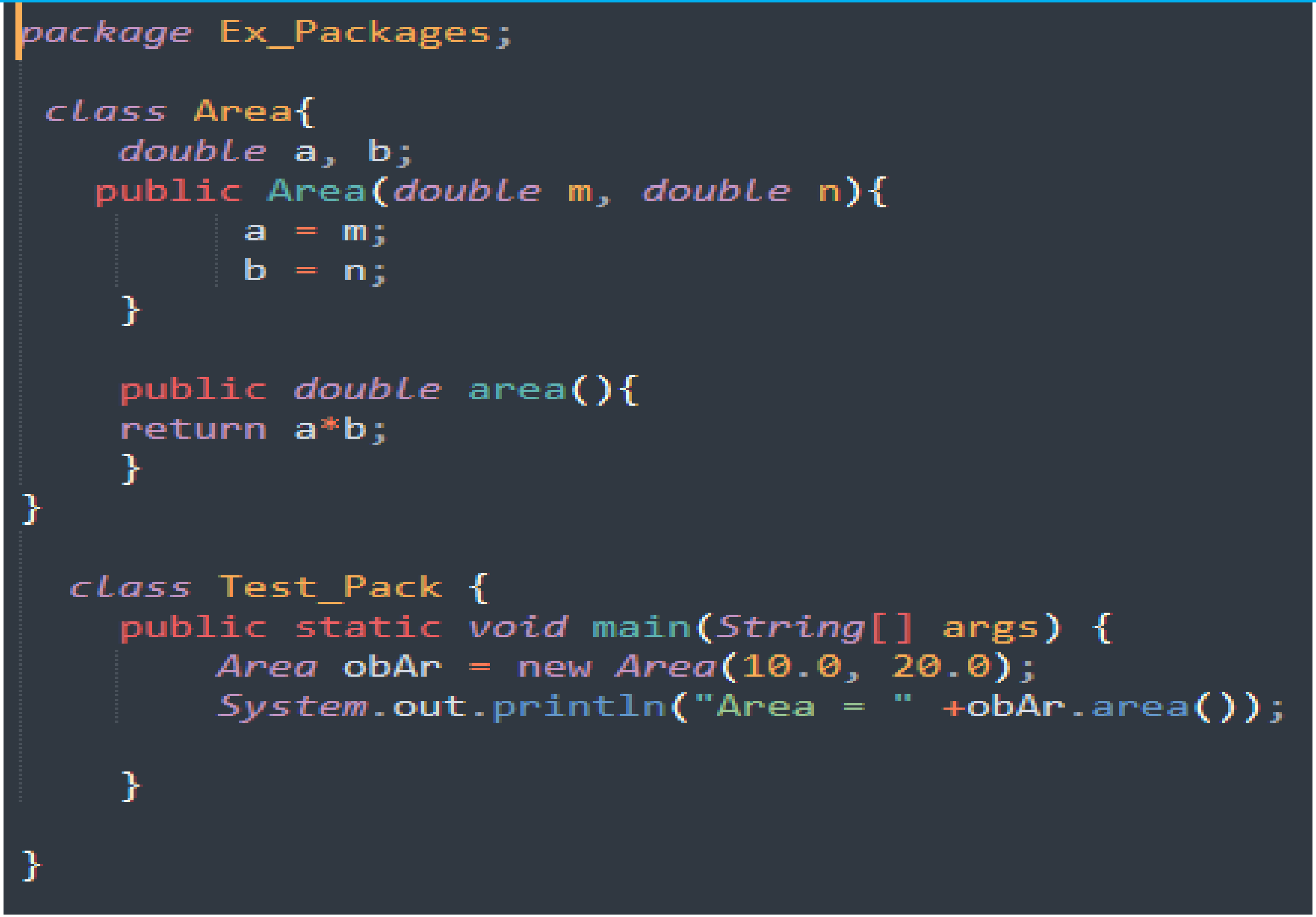
Finding Packages and CLASSPATH

- As packages are mirrored by directories, now an important question is that :

"How does the Java run-time system know where to look for packages that you create? "

- The answer has three parts.
  - First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.
  - Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.
  - Third, you can use the -classpath option with java and javac to specify the path to your classes.

Package: An Example

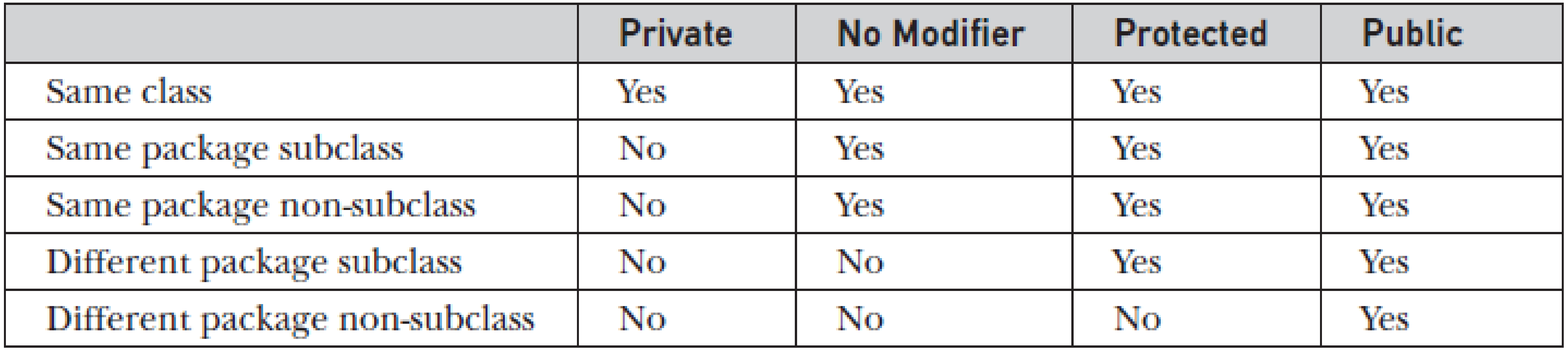


```
package Ex_Packages;

 class Area{
     double a, b;
    public Area(double m, double n){
              a = m;
              b = n;
     }

     public double area(){
     return a*b;
     }
}

  class Test_Pack {
     public static void main(String[] args) {
         Area obAr = new Area(10.0, 20.0);
         System.out.println("Area = " +obAr.area());

     }

}
```

Access Protection

- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code.
- Anything declared public can be accessed from anywhere.
- Anything declared private cannot be seen outside of its class.
- By default, when no access modifier is specified, a class member can be accessed by subclasses and other classes in the same package.
- If you want a member to be seen outside the current package, but only to classes that subclass your class directly, declare that member protected.

Access Protection: Class Member Access



	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Importing Packages

- Java provides import statement to bring certain classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly.

import pkg.*

Import pkg.classname

import pkg1.pkg2.classname

- The may increase compilation time.

Interface

- An interface also introduces a new reference type.
- An interface represents an encapsulation of constants, classes, interfaces, and one or more abstract methods that are implemented by a class.
- An interface does not contain instance variables.
- An interface cannot implement itself; it has to be implemented by a class.
- The methods in an interface have no body.
- Only headers are declared with the parameter list that is followed by a semicolon.
- The class that implements the interface has to have full definitions of all the abstract methods in the interface.
- An interface can be implemented by any number of unrelated classes with their own definitions of the methods of the interface.
- Different classes can have different definitions of the same methods but the parameter list must be identical to that in the interface.
- Thus, interfaces provide another way of dynamic polymorphic implementation of methods.

Interface

- According to modifications in Java SE8, an interface can now have default methods and static methods with full definitions and these methods are inherited by the classes that implement the interface.
- At the same time, the class may override the methods if necessary.
- Any number of interfaces can be implemented by a class.
- This fulfils the need for multiple inheritance.
- The multiple inheritances of classes are not allowed in Java, and therefore, interfaces provide a stopgap arrangement.

Similarities between Interface and Class

- Declaring an interface is similar to that of class; the keyword *class* is replaced by keyword *interface*.
- Its accessibility can be controlled just like a class.
- An interface declared public is accessible to any class in any package, whereas the ones without an access specifier is accessible to classes in the same package only.
- One can create variables as object references of interface that can use the interface.
- It can contain inner classes (nested classes) and inner interfaces.
- Since Java 8, an interface can have full definitions of methods with default or static modifiers.

Dissimilarities between Class and Interface

- Interface cannot implement itself; it must be implemented by a class.
- An interface can contain only method headers followed by a semicolon.
- It cannot have the full definition of a method. The full definition is given in the class that implements it.
- Java 8 modification allows the default and static method declarations in interfaces.
- The methods declared in the interface are implicitly public.
- An interface does not contain instance variables.
- The variables declared in an interface are implicitly public, static, and final,
- Interfaces cannot have a constructor like a class.
- An interface cannot extend a class nor can it have a subclass. It can only extend other interfaces.
- A class can extend (inherit) only one class but an interface can extend any number of interfaces.

Rules for Classes that Implement Interface

- A non-abstract class that implements an interface must have concrete implementation of all the abstract methods of the interface.
- The @Override annotation should be used on the definitions of interface methods in the class.
- The methods declared static and default with full definitions in an interface are inherited by the implementing class since Java 8.

Types of Interfaces

Top level interfaces

- It is an interface that is not nested in any class or interface.
- It comprises a collection of abstract methods.
- It can contain any number of methods that are needed to be defined in the class.

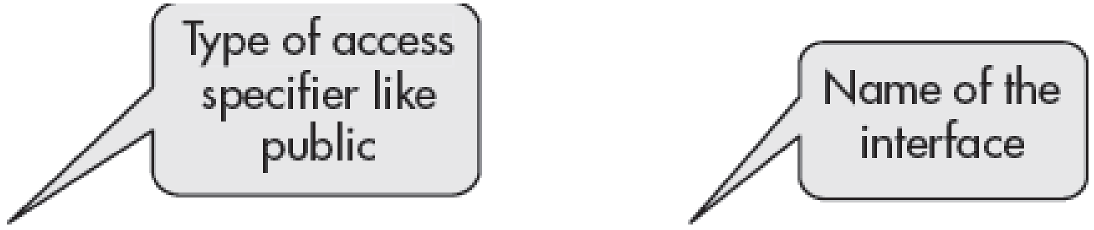
Nested interface

- It is an interface that is defined in the body of a class or interface.
- In nested interfaces, one or more interfaces are grouped, so that it becomes easy to maintain.
- It is referred to by the outer interface or class and cannot be accessed directly.

Declaration of Interface

- Declaration of an interface starts with the access modifier followed by keyword interface.
- It is in then followed by its name or identifier that is followed by a block of statements;
- These statements contain declarations of variables and abstract methods.
- The variables defined in interfaces are implicitly public, static, and final.
- They are initialized at the time of declaration. The methods declared in an interface are public by default.

Declaration of Interface



```
access_Specifier interface Identifier
```

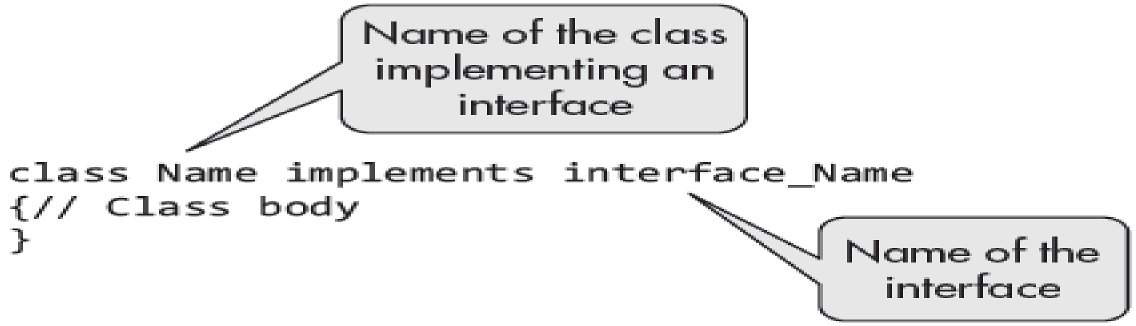
```
{ //body
type variable1_name= value1;
 -----------------------------
type Method1_name (parameter list);

// Since Java SE8 an interface may have default methods.
default void display1 ()
{System.out.println("It is default method.");

//Since Java SE8 an interface may have static methods.
static void display2(){System.out.println("cosine 60 =" +
Math.cos(60*3.141/180));}
----------------------------- }
```

Members of Interface

- The members declared in the body of the interface.
- The members inherited from any super interface that it extends.
- The methods declared in the interface are implicitly public abstract member methods.
- The field variables defined in interfaces are implicitly public, static, and final.
- However, the specification of these modifiers does not create a compile-type error.
- The field variables declared in an interface must be initialized; otherwise, compile-type error occurs.
- Since Java SE8, static and default methods with full definition can also be members of interface.

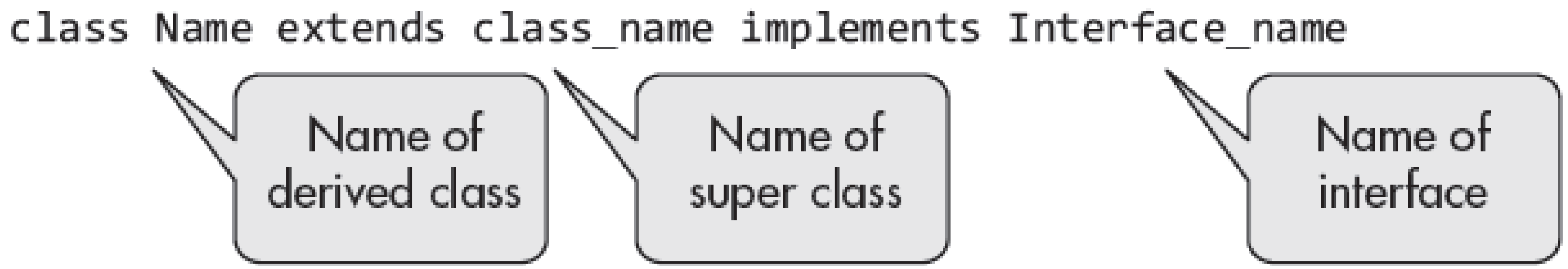
Implementation of Interface

Declaration of class that implements an interface



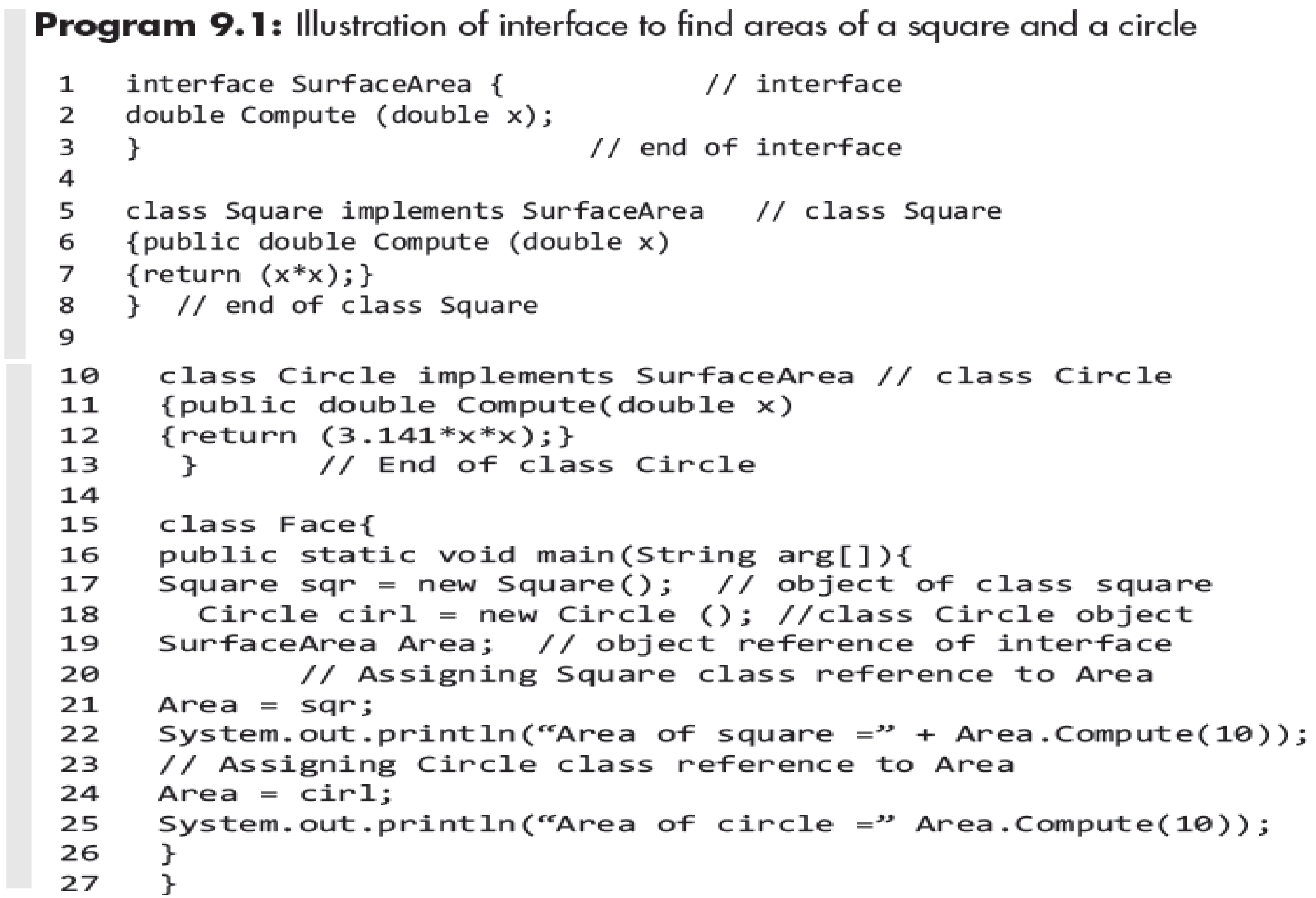
```
class Name implements interface_Name
{// Class body
}
```

If a class extends another class as well as implements interfaces, it is declared as



```
class Name extends class_name implements Interface_name
```

Programming Example



Program 9.1: Illustration of interface to find areas of a square and a circle

```
interface SurfaceArea {              // interface
double Compute (double x);
}                                // end of interface

class Square implements SurfaceArea   // class Square
{public double Compute (double x)
{return (x*x);}
}  // end of class Square

class Circle implements SurfaceArea // class Circle
{public double Compute(double x)
{return (3.141*x*x);}
 }       // End of class Circle

class Face{
public static void main(String arg[]){
Square sqr = new Square();  // object of class square
  Circle cirl = new Circle (); //class Circle object
SurfaceArea Area;  // object reference of interface
       // Assigning Square class reference to Area
Area = sqr;
System.out.println("Area of square =" + Area.Compute(10));
// Assigning Circle class reference to Area
Area = cirl;
System.out.println("Area of circle =" Area.Compute(10));
}
}
```

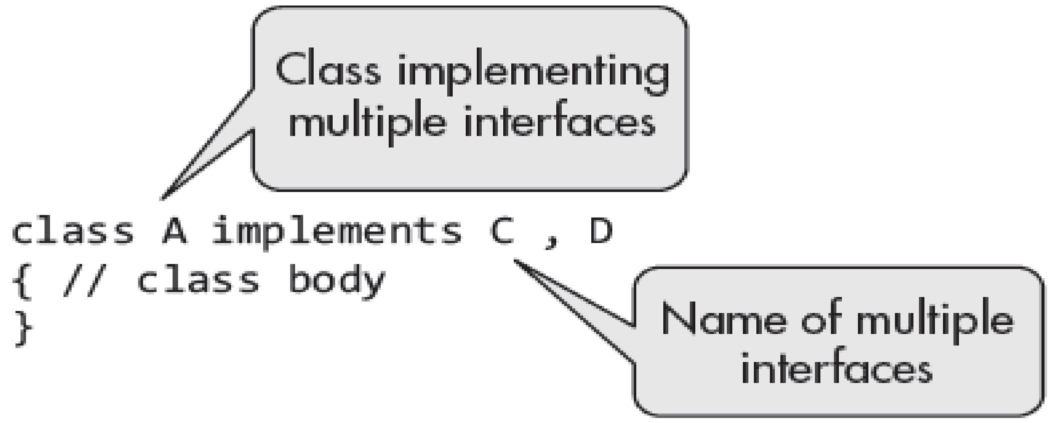
Constants in Interfaces

Program 9.2: Illustration of getting constant values from interface

```
interface Dimensions{
int x = 30;   // implicitly public and final
int y = 20;     //implicitly public and final
}// End of interface Dimensions

class Room implements Dimensions
{
public int area(){
int m = x;
int n = y;
return (m*n);}
     }     // end of class Room

class Inface
{public static void main(String arg[]){
Room rm = new Room();
     Dimensions d;
     d = rm;       // assigning Room reference to D
System.out.println("Area of room =" + rm.area());
     }
}
```

Multiple Interfaces

- Multiple interfaces can also be implemented in Java.
- For this, the class implements all the methods declared in all the interfaces.
- When the class is declared, names of all interfaces are listed after the keyword *implements* and separated by comma.
- As for example, if class A implements interfaces C and D, it is defined as



```
class A implements C , D
{ // class body
}
```

Class implementing multiple interfaces
Name of multiple interfaces

Interface References

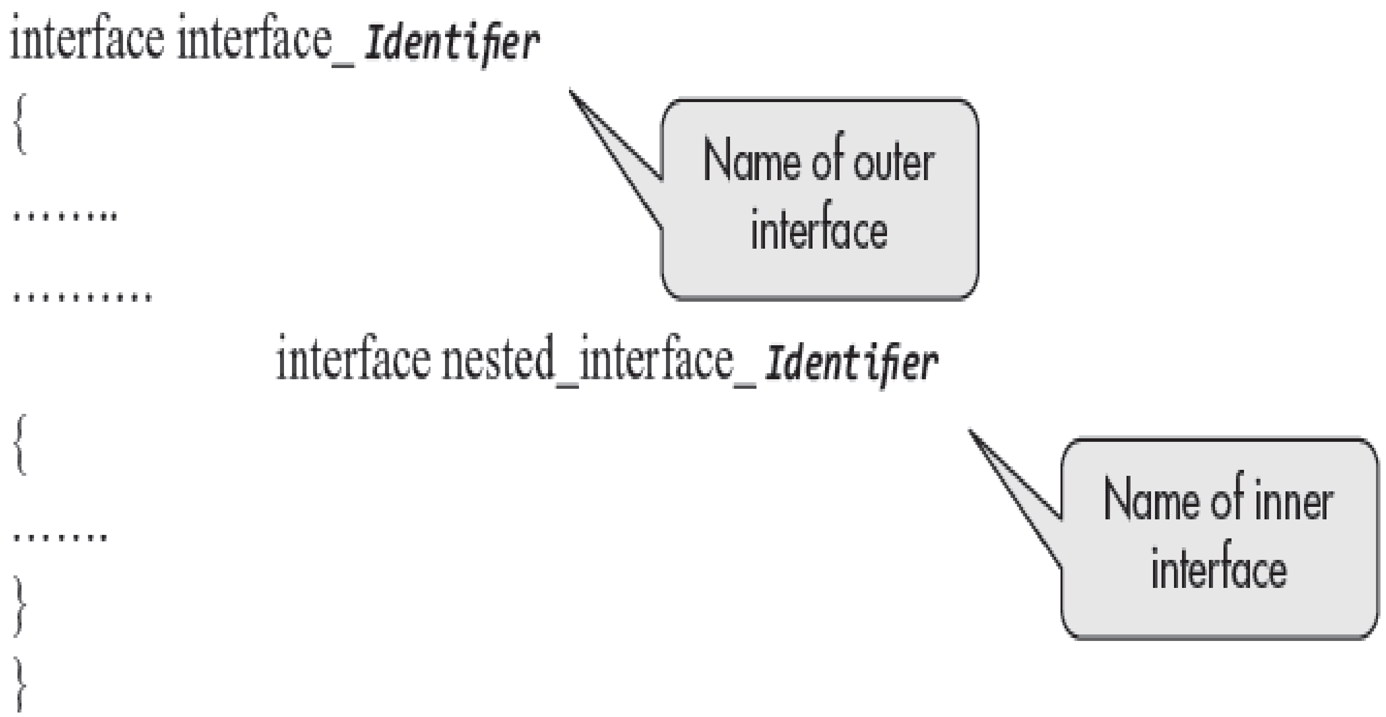
- For interface references, variables can be declared as object references.
- In this case, the object reference would use interface as the type instead of class.
- The appropriate method is called on the basis of actual instance of the interface that is being referred to.

Nested Interfaces

- An interface may be declared as a member of a class or in another
- interface.
- In the capacity of a class member, it can have the attributes that are applicable to other class members.
- In other cases, an interface can only be declared as public or with

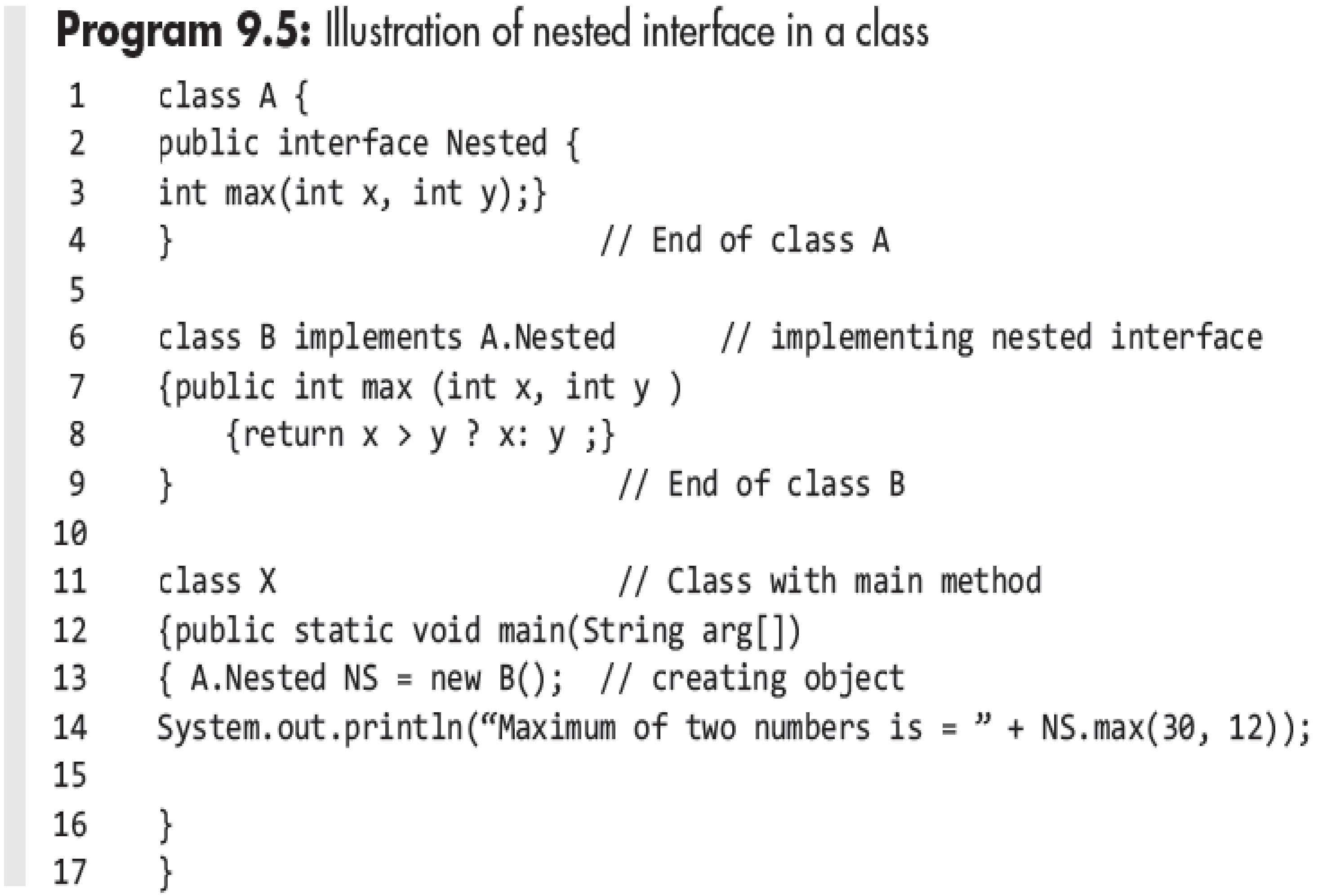
  default (no-access modifier) access.
- Syntax of nested interface in another interface is given as

Nested Interfaces



```
interface interface_ Identifier
{
……..
…………
                 interface nested_interface_ Identifier
{
…….
}
}
```

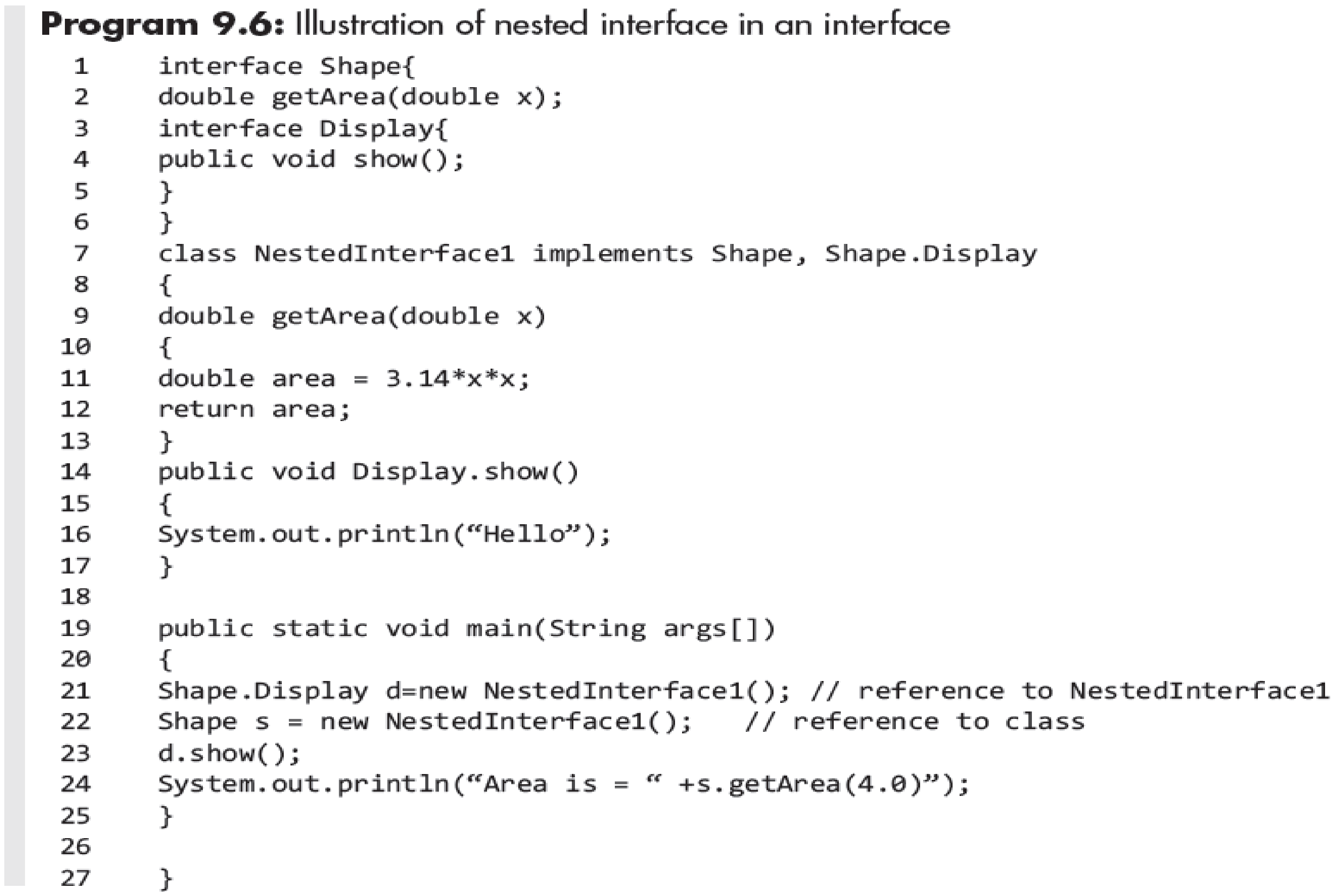
Programming Example



Program 9.5: Illustration of nested interface in a class

```
class A {
public interface Nested {
int max(int x, int y);}
}                         // End of class A

class B implements A.Nested      // implementing nested interface
{public int max (int x, int y )
    {return x > y ? x: y ;}
}                          // End of class B

class X                    // Class with main method
{public static void main(String arg[])
{ A.Nested NS = new B();  // creating object
System.out.println("Maximum of two numbers is = " + NS.max(30, 12));

}
}
```

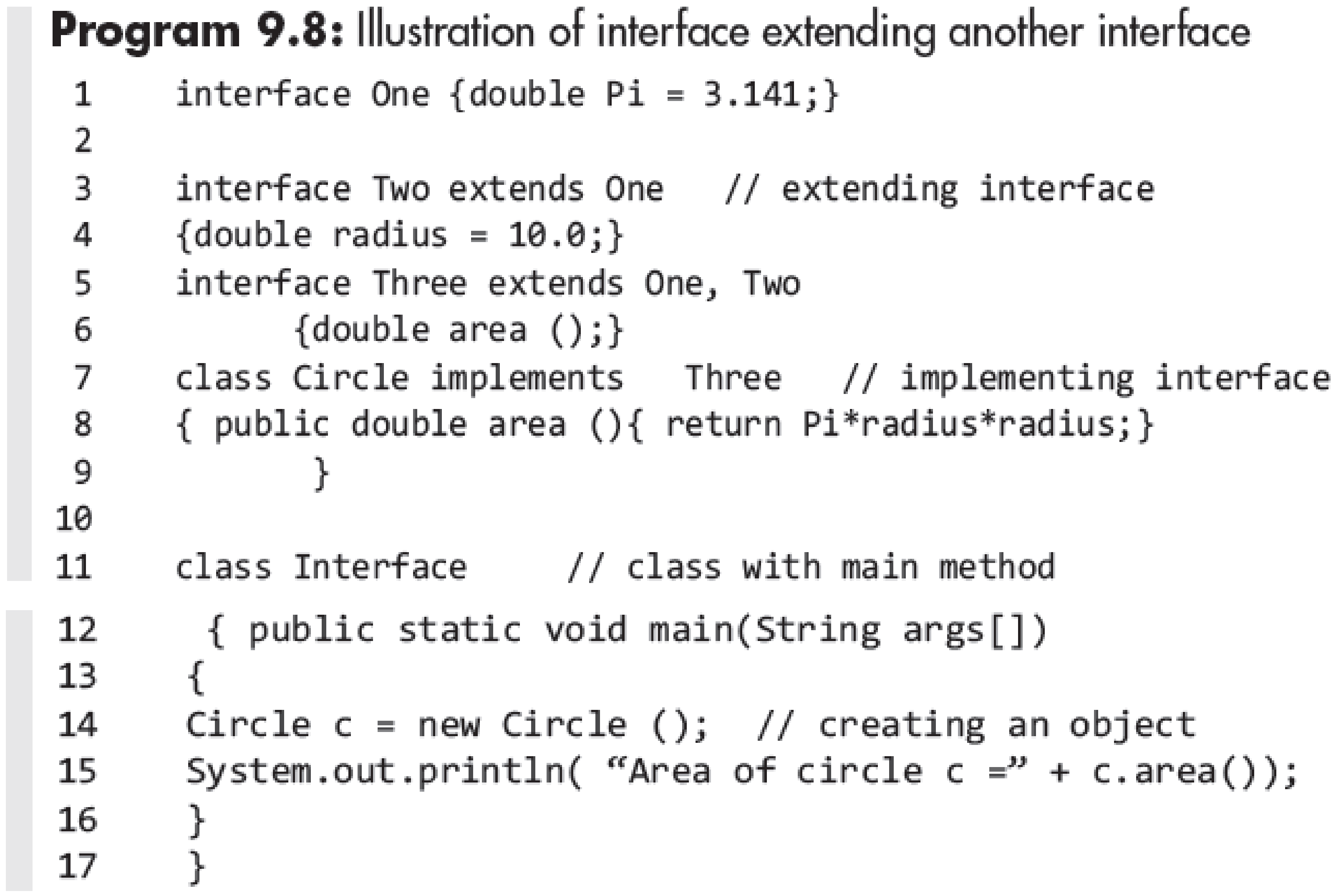
Programming Example



Program 9.6: Illustration of nested interface in an interface

```
interface Shape{
double getArea(double x);
interface Display{
public void show();
}
}
class NestedInterface1 implements Shape, Shape.Display
{
double getArea(double x)
{
double area = 3.14*x*x;
return area;
}
public void Display.show()
{
System.out.println("Hello");
}

public static void main(String args[])
{
Shape.Display d=new NestedInterface1(); // reference to NestedInterface1
Shape s = new NestedInterface1();    // reference to class
d.show();
System.out.println("Area is = " +s.getArea(4.0)");
}

}
```

Inheritance of Interfaces



Program 9.8: Illustration of interface extending another interface

```
interface One {double Pi = 3.141;}

interface Two extends One   // extending interface
{double radius = 10.0;}
interface Three extends One, Two
      {double area ();}
class Circle implements   Three   // implementing interface
{ public double area (){ return Pi*radius*radius;}
       }

class Interface     // class with main method
 { public static void main(String args[])
 {
 Circle c = new Circle ();  // creating an object
 System.out.println( “Area of circle c =” + c.area());
 }
 }
```

Default Methods in Interfaces

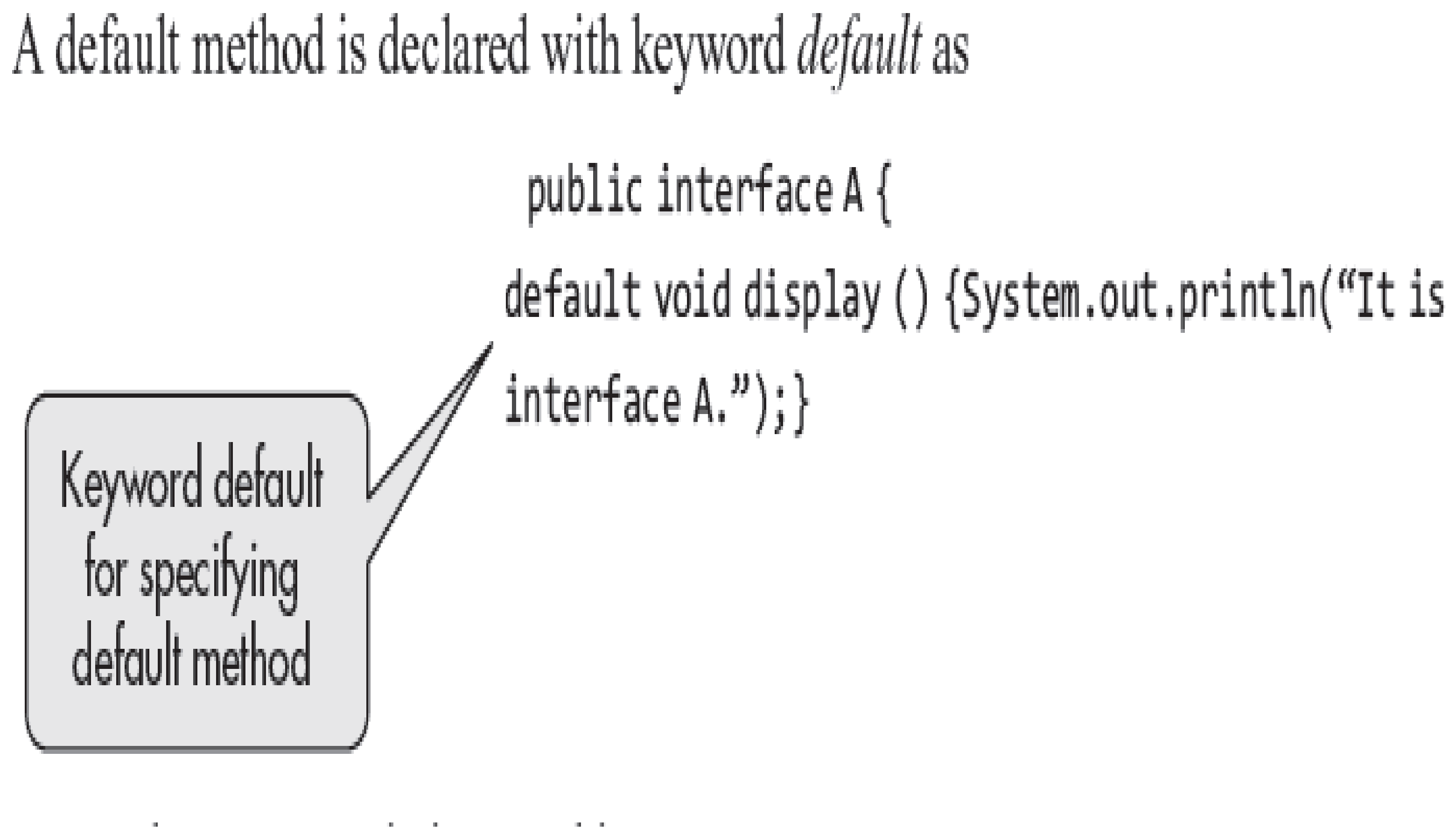
- The *Java SE8* enhancement of interfaces allows the interfaces to have full definitions of default and static methods.
- A class that implements an interface must define all the abstract methods of the interface.
- Now, if at a later stage, it is required to introduce another abstract method to the interface, then all the classes implementing the interface must be modified.
- To overcome this problem, the following two options are available in the existing (before Java SE8) arrangement.

  - Declare another interface that inherits the present interface and define the new method in it.
  - Declare an abstract class with all the methods of the existing interface and the full definitions of new methods.

Default Methods in Interfaces

- The enhancement in Java 8 spares the programmer of doing any change in the existing classes that implement the interface by allowing the interface to have full definition of default methods and static methods that are implicitly inherited by the class implementing the interface.
- The benefit of using the interface over having abstract super-class is that it allows the class to have another super class because in Java, there can be only one super class.
- By this enhancement, the existing framework is totally undisturbed, and the new functionality can be added to the interface, which is inherited by classes implementing the interface.
- The inherited methods are also members of the class, and therefore, these maybe called other methods of class.

Default Methods in Interfaces



A default method is declared with keyword *default* as

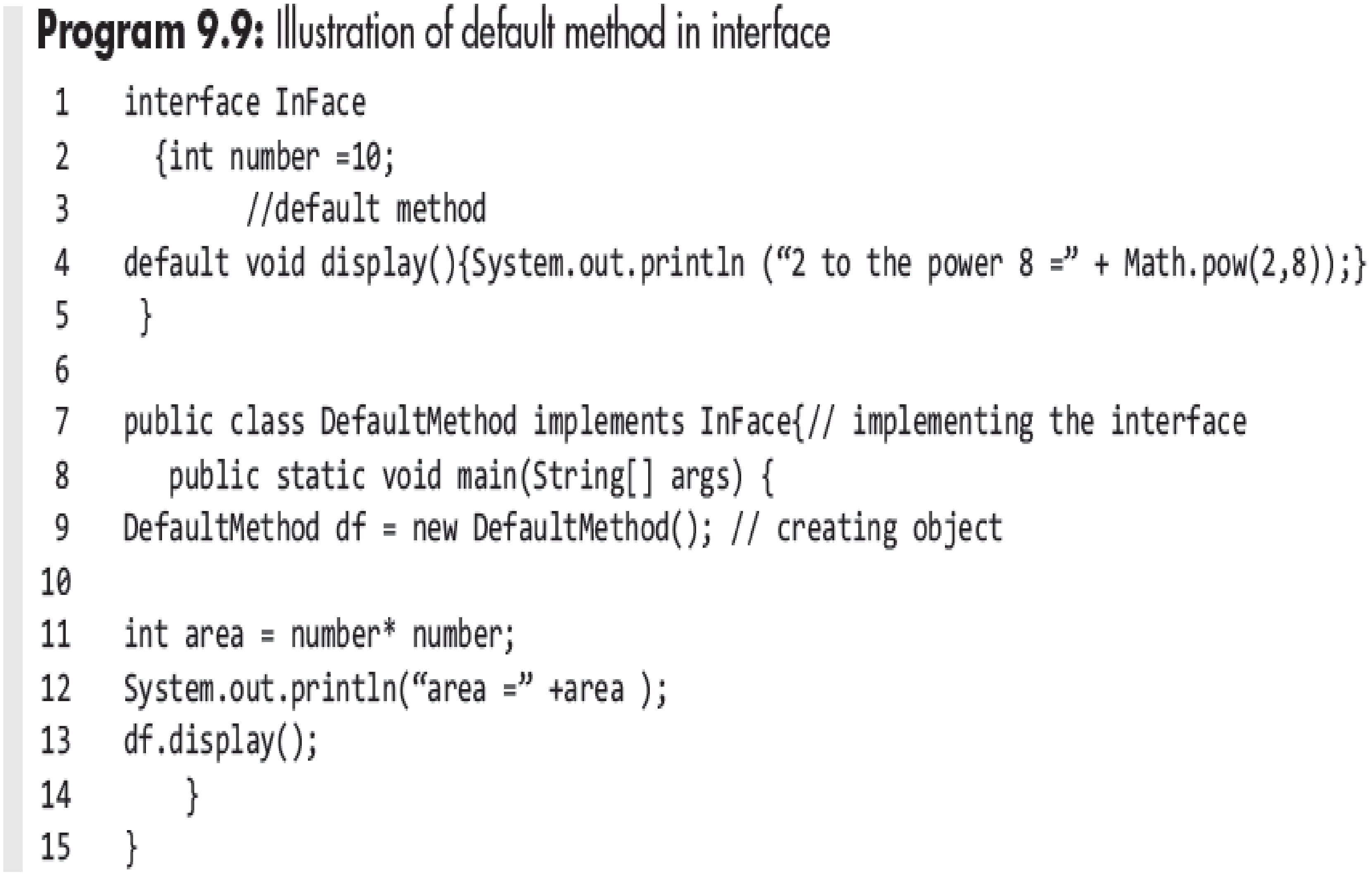
```
public interface A {
default void display () {System.out.println("It is
interface A.");}
```

Keyword default for specifying default method

Default Methods in Interfaces

- If a class implements two or more interfaces, the interfaces cannot have default methods with the same signature because this would cause ambiguity as to which one to execute.
- In the case of methods with the same name, the compiler would choose by matching parameters.
- If the class that is extending the interfaces also defines the method with the same name, then class definition has priority over other definitions.
- A default method cannot be declared final.
- A default method cannot be synchronized; however, blocks of statements in the default method may be synchronized.
- The object class is inherited by all classes. Therefore, a default method should not override any non- final method of object class.

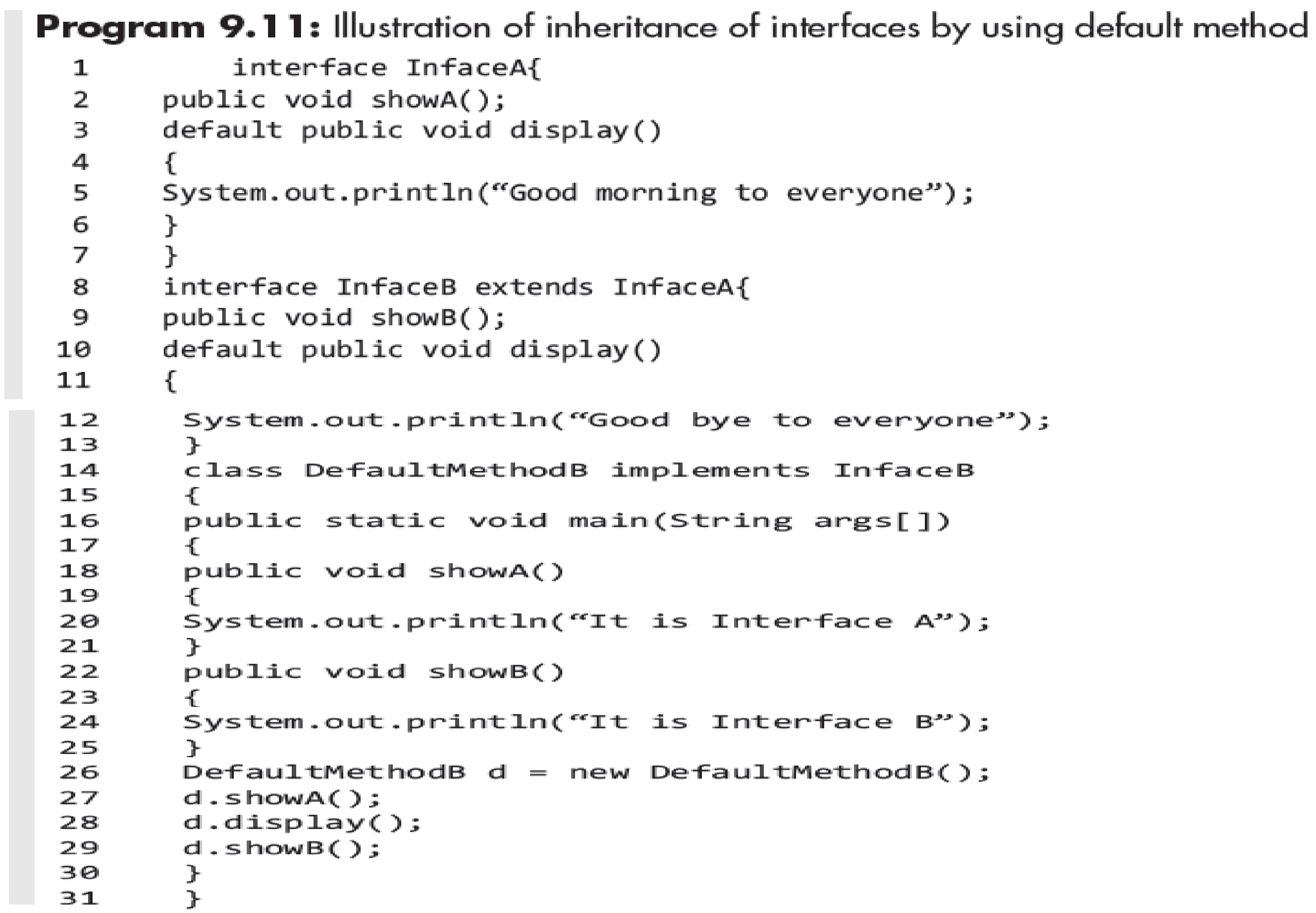
Programming Example



Program 9.9: Illustration of default method in interface

```
interface InFace
  {int number =10;
        //default method
default void display(){System.out.println (“2 to the power 8 =” + Math.pow(2,8));}
 }

public class DefaultMethod implements InFace{// implementing the interface
   public static void main(String[] args) {
DefaultMethod df = new DefaultMethod(); // creating object

int area = number* number;
System.out.println(“area =” +area );
df.display();
    }
}
```

Programming Example



Program 9.11: Illustration of inheritance of interfaces by using default method

```
    interface InfaceA{
public void showA();
default public void display()
{
System.out.println("Good morning to everyone");
}
}
interface InfaceB extends InfaceA{
public void showB();
default public void display()
{
System.out.println("Good bye to everyone");
}
class DefaultMethodB implements InfaceB
{
public static void main(String args[])
{
public void showA()
{
System.out.println("It is Interface A");
}
public void showB()
{
System.out.println("It is Interface B");
}
DefaultMethodB d = new DefaultMethodB();
d.showA();
d.display();
d.showB();
}
}
```

Programming Example

Program 9.12: Default method declared as final

```
    interface A{
public void show();
final default public void display()        // error due to final
{
System.out.println("Good morning to everyone");
}
}
class DemoC implements A
{
public void show()
{    System.out.println("It is Interface");
}

    public static void main(String args[])
{
    DemoC d = new DemoC();
d.show();
d.display();
}
}

```

Static Methods in Interface

- The Java version 8 allows full definition of static methods in interfaces.
- A static method is a class method.
- For calling a static method, one does not need an object of class.
- It can simply be called with class name as

  class_name.method_name()

Dynamic Method Look up

- Dynamic method lookup is the process of determining which method definition a method signature denotes during runtime, based on the type of the object. However, a call to a private instance method is not polymorphic.
- This process is similar to using a superclass reference to access a subclass object, as described in Inheritance.

Annotations

- Annotation framework in Java language was first introduced in Java 5 through a provisional interface Apt;
- It is a type of metadata that can be integrated with the source code without affecting the running of the program.
- Annotations may be retained up to runtime and may be used to instruct the compiler and runtime system to do or not to do certain things.
- Since Java SE 8, the annotations may be applied to classes, fields, interfaces, methods, and type declarations like throw clauses.
- The annotations are no longer simply for metadata inclusion in the program but have become a method for user's communication with compiler or runtime system.

Annotations

- An annotation like @Override consists of two distinct words @ and Override.
- It may as well be written as @ Override;
- The name Override is the name of the interface that defines the annotation.
- There are a number of annotations that are predefined and are part of the package java.lang.annotation.
- However, a programmer may also define an annotation.
- Annotations are defined by interfaces that are preceded by the tag character @.
- Thus, annotations can be easily recognized by symbol @ in the code.
- Java language library has several predefined annotations in its library.
- Java language has provisions for user defined annotations as well as plug-in third party developments.

Programming Example

Program 9.19: Illustration of application of annotation `@override`

```
class XX {
public void display(){System.out.println(“This is class XX.”);
  }}      // end of class XX

class YY extends XX
{@Override public void display(){System.out.println(“This is class YY.”);
  }}             // end of class YY

class ZZ extends XX

{ @Override
public void display(){System.out.println(“This is class ZZ.”);
   }}     // end of class ZZ
                // below is class with main method
public class OverrideSuper {
public static void main (String Str[])
 {
     XX objX = new XX();
     YY objY = new YY();
     ZZ objZ = new ZZ();

objX.display();
 objY.display();
 objZ.display();
 }
}
```

Benefits of Using Annotations

- It provides useful information to the compiler for detecting errors.
- The information may also be used for suppressing warnings.
- Annotations may be used for generating code in xml.
- Annotation may be retained by another annotation for processing at runtime.
- It can carry metadata up to runtime and the information may be obtained at runtime.

Difference between abstract class and interface

Abstract class	interface
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.
Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.
An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.