Inheritance

Inheritance Basics

- In Java, a class that is inherited is called a **superclass** and the class that does the inheriting is called a **subclass**
- A **subclass** is a specialized version of a superclass which inherits all of the variables and methods defined by the **superclass** and adds its own elements
- To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.
- The general form of a class declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name {
 // body of class
}
```

- Ex: The following program creates a superclass called A and a subclass called B

```
class A{
    int a;
    void show(){
        a=10;
    }
}
class B extends A{
    int b;
    B(int b){
        this.b=b;
    }
    void sum(){
        System.out.println("The sum of a and b is "+(a+b));
    }
}
public class Demo1{
    public static void main(String[] args) {
        B obj = new B(10);
        obj.show();
        System.out.println("Value of a is "+obj.a);
        System.out.println("Value of b is "+obj.b);
        obj.sum();
    }
}
```

- The subclass **B** includes all of the members of its superclass, **A**. **B** inherits variable **a** and method **show().** This is why **obj** can access **a** and invoke **show( )** directly as if they were part of **B**. Also, it adds its own variable **b** and method **sum( )**. **sum()** can access variable a directly as if it was a part of class **B**

Output:

Value of a is 10

Value of b is 10

The sum of a and b is 20

- However, class **A** cannot access variable **b** and method **sum( )** defined in its **subclass B** because the superclass has no knowledge about its subclasses**.** Attempting to do so will result in compile time error.

```
class A{
    int a;
    void show(){
        a=10;
    }
}
class B extends A{
    int b;
    B(int b){
        this.b=b;
    }
    void sum(){
        System.out.println("The sum of a and b is "+(a+b));
    }
}
public class Demo1{
    public static void main(String[] args) {
        A obj = new A();
        obj.b=20; //cannot access b. Compile time error
        obj.sum(); // can't access sum(). Compile time error
    }
}
```

Member Access and Inheritance

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.
- Example:

```
class A{
    int a;
    private int b;
    void show(){
        a=10;
        b=20;
    }
}
class B extends A{
    void sum(){
        //Compile time error as variable b is declared as private in class A
        //This won't compile. b is not accessible here
        System.out.println("The sum of a and b is "+(a+b));
    }
}
```

- This program will not compile because the reference to **b** inside the **sum( )** method of **B** causes an access violation. Since **b** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

Constructors and Inheritance

- Constructors cannot be inherited by the subclasses.
- It is possible for both superclass and subclasses to have their own constructors.
- The constructor in the **superclass** constructs the **superclass portion of the object** and the constructor in the **subclass** constructs the **subclass portion of the object**
- When only the subclass defines a constructor, simply construct subclass object. The superclass portion of the object is constructed automatically using its **default constructor.**
- For example,

```
class Fruit{
    Fruit(){ //This constructor will be called automatically
        System.out.println("We are in Fruit class");
    }
}
class Apple extends Fruit{
    Apple(){
        System.out.println("Its an apple class");
    }
}
public class Demo{
    public static void main(String[] args) {
        Apple a=new Apple(); //calls cosntructor of subclass Apple
    }
}
```

Output:

We are in Fruit class

Its an apple class

- In the above program, **subclass constructor** is invoked through **subclass object** and the **superclass constructor** is called **automatically**

Using super

- **super** has two general forms.
  - The first calls the superclass's constructor.
  - The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

- A subclass can call a constructor defined by its superclass by use of the following form of **super**:

super(*arg-list*);

- Here, *arg-list* specifies any arguments needed by the constructor in the superclass.
- **super( )** must always be the **first statement** executed inside a **subclass's constructor**.

Example:

```
class A{
    A(int a){
      System.out.println("Its the A class constructor");
    }
}
class B extends A{
    B(){
        super(10); //calls its superclass constructor
        System.out.println("Its the B class constructor");
    }
}
public class Demo1_1 {
    public static void main(String[] args) {
       B obj = new B();
    }
}
```

Output:

Its the A class constructor

Its the B class constructor

- In the above program, if the **subclass constructor** does not invoke the **superclass constructor** explicitly using **super( )** then a compile time error occurs. Because, subclass constructor automatically calls the **default constructor** of its superclass. Since the **superclass** defines a parameterized constructor, its **default constructor** will no longer be available which results in **compile time error**

```
class A{
    A(int a){
      System.out.println("Its the A class constructor");
    }
}
class B extends A{
    B(){
       //does not call superclass constructor explicitly
        System.out.println("Its the B class constructor");
    }
}
public class Demo1_1 {
    public static void main(String[] args) {
       B obj = new B();
    }
}
```

Output:

```
COMPILATION ERROR :
-------------------------------------------------------------
Demo1_1.java:[8,8] constructor A in class A cannot be applied to given types;
  required: int
  found: no arguments
  reason: actual and formal argument lists differ in length
1 error
```

```
class Shape{
    int length,width;
    Shape(){
        length=0;
        width=0;
    }
    Shape(int l,int w){
       length = l;
       width =w;
    }
}
class Rectangle extends Shape{
    String style;
    Rectangle(){
       super(); //Shape() constructor will be called
       style = "filled";
    }
   Rectangle(int l,int w, String s){
      super(l,w); //Shape(int l,int w) constructor will be called
      style = s;
   }
   void area(){
       System.out.println("The area is "+(length*width));
   }
}
public class Demo1_1 {
    public static void main(String[] args) {
       Rectangle r = new Rectangle(2,3,"solid");
       r.area();
    }
}
```

- In this program, Rectangle() constructor calls super() with the parameters l and w. This causes Shape() constructor to be called, which then initializes length and width using these values.
- Rectangle does not initialize these values on its own. It calls the its superclass constructor which does the job. Rectangle need to initialize only the value which is unique to it i.e. “style”
- Any form of constructor defined by the superclass can be called by super(). The constructor executed will be the one that matches the arguments

Using super to Access Superclass Members

- The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

  super.member

- Here, member can be either a method or an instance variable.

```
// Using super to overcome name hiding.
class A {
  int i;
}

// Create a subclass by extending class A.
class B extends A {
  int i; // this i hides the i in A

  B(int a, int b) {
    super.i = a; // i in A
    i = b; // i in B
  }

  void show() {
    System.out.println("i in superclass: " + super.i);
    System.out.println("i in subclass: " + i);
  }
}

class UseSuper {
  public static void main(String args[]) {
    B subOb = new B(1, 2);

    subOb.show();
  }
}
```

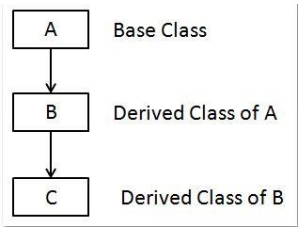
This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

- Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass

Creating a Multilevel Hierarchy

- It is perfectly acceptable to use a subclass as a superclass of another.
- For example, given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A. When this type of situation occurs, each subclass inherits all of the aspects found in all of its superclasses. In this case, C inherits all aspects of B and A.



```
class A {
    int a;
    A()
    {
        a=10;
    }
   void funcA() {
      System.out.println("This is class A");
   }
}
class B extends A {
    int b;
    B(){
        super(); //calls its superclass constructor(class A)
        b=20;
    }
   void funcB() {
      System.out.println("This is class B");
   }
}
class C extends B {
    int c;
    C(){
        super(); //calls its superclass constructor(class B)
        c=30;
    }
   void funcC() {
      System.out.println("This is class C");
   }
}
public class Demo1 {
   public static void main(String args[]) {
      C obj = new C();
      obj.funcA();
      obj.funcB();
      obj.funcC();
      System.out.println("Values of a, b and c are: "+obj.a+" "+obj.b+" "+obj.c);
   }
}
```

OUTPUT:

```
This is class A
This is class B
This is class C
Values of a, b and c are: 10 20 30
```

- In the above program, class B inherits A, C inherits B. So, C includes all members of classes B and A.
- A class C object can be used to call methods and access variables defined by itself and its superclasses
- super() always refers to the constructor in the closest superclass.
- The **super( )** in **C** calls the constructor in **B**. The **super( )** in **B** calls the constructor in **A**.

When Constructors Are Called?

- In a class hierarchy, constructors are called in order of derivation, from superclass to subclass

- For example,

```
class A{
    A(){
        System.out.println("Its the superclass constructor A");
    }
}
class B extends A{
    B(){
        //implicit call to its superclass constructor A by implicit super()
        System.out.println("Its the subclass constructor B");
    }
}
class C extends B{
    C(){
        //implicit call to its superclass constructor B by implicit super()
        System.out.println("Its the subclass constructor C");
    }
}
public class Demo1{
    public static void main(String[] args) {
        C obj = new C();
    }
}
```

OUTPUT:

```
Its the superclass constructor A
Its the subclass constructor B
Its the subclass constructor C
```

- Always superclass constructors will be called before their subclass constructors. This is because each subclass constructor will **implicitly** call its superclass default constructor
- Hence, the constructors are called in order of derivation

A Superclass Variable Can Reference a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. i.e. a superclass reference can refer to a subclass object

Example:

```
class X{
    int a;
    X(int i){
        a=i;
    }
}
class Y extends X{
    int b;
    Y(int i, int j){
        super(i);
        b=j;
    }
}

public class SupSubRef {
    public static void main(String[] args) {
        X x=new X(10);
        Y y=new Y(5,6);
        X x2;
        x2 = x;
        System.out.println("value of a "+ x2.a);
        x2=y; //assign Y reference to X reference
        System.out.println("value of a"+ x2.a);
       //System.out.println("value of b "+ x2.b);//Error, X doesn't
                                                  //have a b member
    }
}
```

- It is the type of the **reference variable—not the type of the object** that it refers to—that determines what members can be accessed.
- That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have **access only to those parts of the object** defined by the superclass
- This is why **x2** can't access **b** even when it refers to a **Y** object.

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.
- The version of the method defined by the superclass will be hidden.

Example:

```
class Shape{
    int length,width;
    Shape(){
        length=1;
        width=1;
    }
    void area(){
        System.out.println("This will be overridden by subclasses of Shape");
    }
}
class Rectangle extends Shape{
    Rectangle(){
        super();
    }
    @Override
    void area(){
        System.out.println("This overrides the method area in Shape class");
        System.out.println("Area of rectangle is "+(length*width));
    }
}
public class Method_Override {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        r.area(); //area() method in Rectangle class will be invoked

    }
}
```

OUTPUT:

```
This overrides the method area in Shape class
Area of rectangle is 1
```

- When **area( )** is invoked on an object of type **Rectangle**, the version of **area( )** defined within **Rectangle** is used. That is, the version of **area( )** inside **Rectangle** overrides the version declared in **Shape**.
- To access the superclass version of an overridden method, you can do so by using **super** keyword.

```
class Rectangle1 extends Shape{
    Rectangle1(){
        super();
    }
    @Override
    void area(){
        super.area(); //calls area() defined in superclass Shape
        System.out.println("This overrides the method area in Shape class");
        System.out.println("Area of rectangle is "+(length*width));
    }
}
public class Method_Override {
    public static void main(String[] args) {
        Rectangle1 r = new Rectangle1();
        r.area(); //area() method in Rectangle class will be invoked

    }
}
```

OUTPUT:

```
This will be overridden by subclasses of Shape
This overrides the method area in Shape class
Area of rectangle is 1
```

```
class Shape{
    int length,width;
    Shape(){
        length=1;
        width=1;
    }
    void area(){
        System.out.println("Default area in Shape class");
    }
}
class Rectangle extends Shape{
    void area(int l,int w){ //this does not override area() in Shape,
                            //it simply overloads
        System.out.println("Area of rectangle is "+(l*w));
    }
}
public class Method_Override {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        r.area(2,4); //area() in Rectangle class will be invoked
        r.area(); //area() method in Shape class will be invoked
    }
}
```

OUTPUT:

```
Area of rectangle is 8
Default area in Shape class
```

- The version of **area( )** in **Rectangle** takes two integer parameters. This makes its type signature different from the one in **Shape**, which takes no parameters. Therefore, no overriding (or name hiding) takes place. Instead, the version of **area( )** in **Rectangle** simply overloads the version of **area( )** in **Shape**.

Overridden Methods Support Polymorphism or Dynamic Method Dispatch or Run-time Polymorphism

- Dynamic method dispatch (**late binding**) is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- **Dynamic method dispatch** is important because this is how Java **implements run-time polymorphism**
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.
- In other words, it is the type of the object being referred to (**not the type of the reference variable**) that determines which version of an overridden method will be executed.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.
  Example,

```
class X1{
    void display(){
        System.out.println("Its the Class X");
    }
}
class Y1 extends X1{
    @Override
    void display(){
        System.out.println("Its the Class Y");
    }
}
class Z1 extends X1{
    @Override
    void display(){
        System.out.println("Its the Class Z");
    }
}
```

```
public class Runtime_Polymorphism {
    public static void main(String[] args) {
        X1 x = new X1();
        Y1 y = new Y1();
        System.out.println("Output of Compile time polymorphism");
        x.display(); // resolved at compile time,
                     //display() is called through the object of X1 directly
        X1 x1;
        x1=y;
        System.out.println("Output of Run time polymorphism");
        x1.display(); // resolved at run time,
                      //display() is called through the reference variable of the
                      //superclass X1 which is referencing to object of Y1
        Z1 z=new Z1();
        x1=z;
        x1.display();
    }
}
```

OUTPUT:

```
Output of Compile time polymorphism
Its the Class X
Output of Run time polymorphism
Its the Class Y
Its the Class Z
```

- This program creates one superclass called **X1** and two subclasses of it, called **Y1** and **Z1**.
- Subclasses **Y1** and **Z1** override **display( )** declared in **X1**. Inside the **main( )** method, objects of type **X1, Y1**, and **Z1** are declared. Also, a reference of type **X1**, called **x1**, is declared.
- The program then in turn assigns a reference to each type of object to **x1** and uses that reference to invoke **display( )**.
- As the output shows, the version of **display( )** executed is determined by the type of object being referred to at the time of the call.

Why Overridden Methods?

- Overridden methods allow Java to support run-time polymorphism
- It allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.
- Dynamic, run-time polymorphism is one of the most powerful mechanisms that object oriented design brings to bear on code reuse and robustness.

Using Abstract Classes

- Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
-

- You can have methods that must be overridden by the subclass in order for the subclass to have any meaning. If we consider the class Triangle, it has no meaning if area( ) is not defined.
- In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the **abstract method.**
- Methods must be overridden by subclasses by specifying the abstract type modifier.
- These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass
- To declare an abstract method, use this **general form**:

  abstract type name(parameter-list);
- No method body is present
- Any class that contains one or more abstract methods must also be declared abstract.
- To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator
- You cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must implement all of the abstract methods in the superclass

```
// A Simple demonstration of abstract.
abstract class A {
  abstract void callme();

  // concrete methods are still allowed in abstract classes
  void callmetoo() {
    System.out.println("This is a concrete method.");
  }
}

class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  }
}

class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

- No objects of class **A** are declared in the program. It is not possible to instantiate an abstract class.
- One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable.
- Although abstract classes cannot be used to instantiate objects, they can be used to create object references
- It must be possible to create a reference to an abstract class so that it can be used to point to a subclass object

Using final

- The keyword **final** has three uses.

1. **Using final to Prevent Overriding**
   - To disallow a method from being overridden, specify final as a modifier at the start of its declaration.
   - Methods declared as final cannot be overridden.
   - The following fragment illustrates final

```
class A {
  final void meth() {
    System.out.println("This is a final method.");
  }
}

class B extends A {
  void meth() { // ERROR! Can't override.
    System.out.println("Illegal!");
  }
}
```

   - Because **meth( )** is declared as **final**, it cannot be overridden in **B**.
   - If you attempt to do so, a compile-time error will result
2. **Using final to Prevent Inheritance**
   - You can prevent a class from being inherited by preceding the class declaration with **final**.
   - Declaring a class as **final** implicitly declares all of its methods as **final**, too
   - It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a final class:

```
final class A {
  // ...
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
  // ...
}
```

As the comments imply, it is illegal for B to inherit A since A is declared as final.

3. **Using final with Data Members**
   - If a class variable's name is preceded with **final**, its value cannot be changed throughout the lifetime of the program.
   - Initial value can be assigned.

```
class finalKey{
    final int i=10;
    void change(){
        i=30; //Error, as i is declared as final,
              //value of i cannot be changed.
    }
}
class name {
    public static void main(String[] args) {
        finalKey k=new finalKey();
        k.change();
    }

}
```

The Object Class (FYI)

- There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**.
- That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class.
- Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.

Object defines the following methods, which means that they are available in every object

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

- The methods **getClass( ), notify( ), notifyAll( ), and wait( )** are declared as **final**. You may override the others.
- The **equals( )** method compares the contents of two objects. It returns true if the objects are equivalent, and false otherwise.
- The **toString( )** method returns a string that contains a description of the object on which it is called

Interfaces

Interface Fundamentals

- An interface defines a set of methods that will be implemented by a class
- Interfaces are syntactically similar to classes, except that no method can include a body
- This means that an interface provides no implementation whatsoever of the methods it defines
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.
- To implement an interface, a class must provide bodies (implementations) for the methods declared by the interface.
- Interfaces are used to achieve **total abstraction** which was not achieved through abstract classes
- By providing the interface keyword, Java allows you to fully utilize the **"one interface, multiple methods"** aspect of polymorphism.

Here are mainly three reasons to use interface. They are given below.

1. It is used to achieve abstraction.
2. By interface, we can support the functionality of multiple inheritance which was not achieved through classes
3. It can be used to achieve loose coupling.

Creating an Interface

- An interface is defined much like a class. This is the general form of an interface:

```
access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    type final-varname1 = value;
    type final-varname2 = value;
    // ...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}
```

- When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.
- When it is declared as **public**, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface

```
interface Callback {
  void callback(int param);
}
```

- It declares a simple interface that contains one method called **callback( )** that takes a single integer parameter

Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.

  - To implement an interface, include the **implements** clause in a class definition, and
  - Then create the methods defined by the interface
- The general form of a class that includes the **implements** clause looks like this:

```
class classname extends superclass implements interface{
    // class-body
  }
```

- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition

```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {

    System.out.println("callback called with " + p);
  }
}
```

Notice that **callback( )** is declared using the **public** access specifier.

Using Interface References

- An interface declaration creates a new reference type.
- When a class implements an interface, it is adding that interface's type to its type
- As a result, an instance of a class that implements an interface is also an instance of that interface type.
- Because an interface defines a type, you can declare a reference variable of an interface type.
- In other words, you can create an interface reference variable but not its object just like abstract classes.
- However, it can refer to any object that implements the interface.
- When you call a method on an object through an interface reference, it is the version of the method implemented by the object that is executed.

The following example calls the **display( ) and show( )** method via an interface reference variable:

```
interface In {
    // implicitly public and abstract
    void display();
    void show();
}
class TestClass implements In { //Needs to prvide implementation for both
                                //display and show methods
    // Implementing the capabilities of interface.
    public void display(){
      System.out.println("This is overriding display in interface In");
    }
    public void show(){
      System.out.println("This is the implementation for show in interface In");
    }
}
public class Interface_ex{
    public static void main(String[] args){
        In i; //reference variable if interface In.
              //object cannot be created as interface is not a complete entity
        i= new TestClass(); //interface reference variable is
                            //referencing to TestClass object
        i.display(); //display in TestClass is called through
                     //interface reference variable
        i.show();
    }
}
```

Implementing Multiple Interfaces

- A class can implement more than one interface.
- The class must implement all of the methods specified by each interface.
- For example

```
interface IfA{
    void doSomething();
}
interface IfB{
    void doSomethingElse();
}

//Implement both IfA and IfB
class MyClass implements IfA, IfB{
    public void doSomething(){
        System.out.println("Doing something");
    }
    public void doSomethingElse(){
        System.out.println("Doing something else");
    }
}
```

- In this example, **MyClass** specifies both **IfA** and **IfB** in its implements clause.

Constants in Interfaces

- Interface can also include variables. Such variables are not instance variables. Instead, they are implicitly **public, final and static** and must be initialised.
- Hence, they are constants

```
interface IConstant {
    // implicitly public final and static
    int MAX=999; //initial value is required
}
class TestClass implements IConstant{
    void show(){
        //won't compile as final variable cannot be changed!!
        System.out.println("The value of max is "+(MAX=MAX+1));
    }
}
```

- IConstant inteface defines **MAX** constant. As required, they are given initial value.
- The above program won't compile because once the final variable in initialized, its value cannot be altered.

Interfaces can be Extended (FYI)

- One interface can inherit another by use of the keyword **extends**
- When class implements an interface that inherits another interface. It must provide implementations for all methods defined within the interface inheritance chain.

```
// One interface can extend another.
interface A {
  void meth1();
  void meth2();
}

// B now includes meth1() and meth2() - it adds meth3().
interface B extends A {
  void meth3();
}

// This class must implement all of A and B
class MyClass implements B {
  public void meth1() {
    System.out.println("Implement meth1().");
  }
  public void meth2() {
    System.out.println("Implement meth2().");
  }

  public void meth3() {
    System.out.println("Implement meth3().");
  }
}

class IFExtend {
  public static void main(String args[]) {
    MyClass ob = new MyClass();

    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

- In this example, interface A is extended by interface B
- MyClass implements B. This means that MyClass must implement all of the methods defined by both interfaces A and B.

Nested Interfaces

- An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*.
- A nested interface can be declared as **public**, **private**, or **protected**
- An interface nested in another interface is implicitly **public**
- When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.

```
// A nested interface example.

// This class contains a member interface.
class A {
```

```
  // this is a nested interface
  public interface NestedIF {
    boolean isNotNegative(int x);
  }
}

// B implements the nested interface.
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false : true;
  }
}

class NestedIFDemo {
  public static void main(String args[]) {

    // use a nested interface reference
    A.NestedIF nif = new B();

    if(nif.isNotNegative(10))
      System.out.println("10 is not negative");
    if(nif.isNotNegative(-12))
      System.out.println("this won't be displayed");
  }
}
```

- **A** defines a member interface called **NestedIF** and that it is declared **public**.
- Next, **B** implements the nested interface by specifying
  implements A.NestedIF
- The name is fully qualified by the enclosing class name. Inside the **main( )** method, an **A.NestedIF** reference called **nif** is created, and it is assigned a reference to a **B** object. Because **B** implements **A.NestedIF**
- Since **B** implements only the nested interface NestedIF, it does not need to implement **doSomething( )** method.