JAVA PROGRAMMING

UNIT - IV

Dr Yogish H K, Professor, Dept. of ISE, RIT, Bengaluru -54.

UNIT - IV

- **Syllabus**

Inheritance: Introduction to Inheritance, using super, creating a Multilevel Hierarchy, When Constructors are Called, Method Overriding, Dynamic Method Dispatch, Abstract Classes, final with Inheritance.

Inheritance

- □ Defined as the process where one class acquires the properties (methods and fields) of another class.
- □ The class which inherits the properties of other is known as **subclass (*derived class, child class*)**
- □ the class whose properties are inherited is known as **superclass (*base class, parent class*)**.
- □ Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

Inheritance …

- To inherit a class, use **extends** keyword.

- Declaration of subclass that inherits a superclass is:

```
class subclassName extends superclassName {
        // body of class
}
```

Example - Creats superclass called A and a subclass called B.

```
class A {                                       // Create a superclass.
           int a;
           void dispa() {
                                    System.out.println(“a:" + a );
             }
}
class B extends A {                             // Create a subclass by extending class A.
           int b;
           void dispb() {
                                    System.out.println(“b: " + b);
                      }
           void sum() {
                                    System.out.println(“a+b: " + (a+b));
                      }
     }
```

Example - creates a superclass called A and a subclass called B. ….

```
class SimpleInheritance {
    public static void main(String args []) {
        B subOb = new B( );
        subOb.a = 7;
        subOb.b = 8;
        System.out.println("Contents of a and b are: ");
        subOb.dispa();
        subOb.dispb();
        System.out.println();
        System.out.println("Sum of a and b is:");
        subOb.sum();
   }
}
```

Member Access and Inheritance

- a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.
- Ie. A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

Example : Member Access and Inheritance …

```
// Create a superclass.
class A {
        int a;                  // public by default
        private int pa;         // private to A
        void setap(int x, int y) {
                a = x;
                pa = y;
        }
 }
class B extends A {             // A’s pa is not accessible here.
                int total;
                void sum() {
                        total = a + pa; // ERROR, pa is not accessible here
                }
}
```

Creating a Multilevel Hierarchy

- We have been using simple class hierarchies that consist of only a superclass and a subclass.
- However, you can build hierarchies that contain as many layers of inheritance as you like.
- As mentioned, 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 traits found in all of its super classes. In this case, **C** inherits all aspects of B and **A**.

Example

```
class A {               int a;
                        void dispa() {
                                 System.out.println(“a " + a);
                           }
}
class B extends A {     int b;
                        void dispb() {
                                System.out.println(“b: " + b);
                            }
}
class C extends B {     int c;
                        void dispc() {
                                System.out.println(“c: " + c);
                          }
```

Example

```
  void sum() {
                    System.out.println(“a+b+c: " + (a+b+c));
          }
 }      // end of class C
class MultilevelInheritance {
        public static void main(String args[]) {
                C subOb = new C();
                subOb.a = 7;
                subOb.b = 8;
                subOb.c =9 ;
                System.out.println("Contents of a, b and c are: ");
                subOb.dispa();
                subOb. dispb();
                subOb. dispc();
                System.out.println("Sum of a ,b and c is:");
                subOb.sum();
        }
}
```

Constructors in Inheritance

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

- Example, given a **subclass** called **B** and a superclass called A, hence A's constructor called before B's.

Example …

```
class A {
        A( ) {
                System.out.println(" A's constructor.");
        }
}
class B extends A {
        B( ) {
                System.out.println("B's constructor.");
        }
}
class C extends B {
        C( ) {
                System.out.println("C's constructor.");
        }
}
```

. . .

```
class CallingCons {
        public static void main(String args[]) {
        C SubOb = new C( );
    }
}
```

Output:

```
            A’s constructor
            B’s constructor
            C’s constructor
```

Using super

- super has 2 uses.
- 1. **Used for calling the superclass' constructor.**

  super(arg-list);
- 2. **Used to access a member of the superclass that has been hidden by a member of a subclass.**

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

Example1: Used to call the superclass’ constructor

```
class A {
                int a;
                A(int i){
                                    a=i;
                          }
                 void dispa() {
                                     System.out.println(“a " + a );

                         }
}
```

Using super ….

```
class B extends A {
                    int b;
                    B(int b1,int b2 {
                          super(b1);
                          b=b2;
                        }
                   void dispb() {
                        System.out.println(“b: " + b);
                   }
  }
```

Using super ….

```
class SimpleInheritance {
 public static void main(String args[]) {
        B subOb = new B(5,10);
        System.out.println("Contents of a and b are: ");
        subOb.dispa();
        subOb.dispb();
   }
}
```

Example - to access a member of the superclass that has been hidden by a member of a subclass : **super.member;**

```
class A {                       // Using super to overcome name hiding.
          int a;
   }
class B extends A {
                          int a;       // this a hides the a in A
         B(int b1, int b2) {
                           super.a = b1;    // a in A
                           a = b2;          // a in B
          }
       void disp() {
                    System.out.println(“a in superclass: " + super.a);
                    System.out.println(“a in subclass: " + a);
            }
}
```

Continue…

```
class UseSuper {
        public static void main(String args[]) {
        B subOb = new B(14, 15);
        subOb.disp();
         }
}
```

- The instance variable **a** in **B** hides the **a** in **A**, super allows access to the **a** defined in the superclass.

Super can also be used to call methods that are hidden by a subclass.

- **// Using super to overcome method hiding.**

```
class A {
          int a;
          void disp( ) {
                System.out.println(“a in superclass: " + a);
               }
 }
```

…

```
class B extends A {
        int b ;
        B(int b1, int b2) {
                a=b1;
                b= b2;
            }
        void disp( ) {
                super.disp();
                System.out.println(“b in subclass: " + b);
            }
   }
```

• • •

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

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 its 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.

Method Overriding…EXAMPLE

```
class A {
        void disp() {
                        System.out.println(“Class A”);
                }
}
class B extends A {
        void disp() {
                        System.out.println(“Class B”);
                }
}
class Override {
                public static void main(String args[]) {
                B subOb = new B( );
                subOb.disp();       // this calls disp( ) of B
        }
}
```

Note: Method overriding occurs only when the names and the type signatures of the two methods are identical.

If they are not, then the two methods are simply overloaded.

Dynamic Method Dispatch

- Method overriding forms the basis for one of Java’s most powerful concepts: **Dynamic Method Dispatch.**
- It is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- It is used to achieve **run-time polymorphism**

Dynamic Method Dispatch …

- Superclass **reference variable** can refer to a **subclass object**.
- 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.
- It is the type of the object being referred to that determines which version of an overridden method will be executed.

Example

```
class A {
            void disp() {
                                    System.out.println(“Class A”);
                        }
}
class B extends A {                                 // override s disp( )
            void disp() {
                                    System.out.println(“Class B”);
                        }
}
 class C extends A {                               // override disp( )
            void disp() {
                                    System.out.println(“Class C”);
                        }
}
```

…

```
class Dispatch {
        public static void main(String args[]) {
                A Oa = new A( );          // object of type A
                B Ob = new B( );          // object of type B
                C Oc = new C( );         // object of type C
                A  r;               // obtain a reference of type A
                r = Oa;                          // r refers to an A object
                r. disp( );         // calls A's version of disp( )
                r = Ob;             // r refers to a B object
                r. disp( );         // calls B's version of disp( )
                r = Oc;                          // r refers to a C object
                r. disp( );         // calls C's version of disp( )
        }
}
```

Abstract Classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- A superclass that defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- Such a class determines the nature of the methods that the subclasses must implement.

Abstract Classes …

- Any class that contains one or more abstract methods must also be declared abstract.
  - **abstract class A{ }**
- There can be no objects of an abstract class.
- That is, an abstract class cannot be directly instantiated with the **new operator.**
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared **abstract** itself.

Example 1

```
abstract class A {
        abstract void disp();
}
class B extends A {
                void disp() {
                        System.out.println(“Class B”);
                }
}
 class C extends A {
        void disp() {
                        System.out.println(“Class C”);
                }
}
```

…

```
class Main{
public static void main(String[] args) {
        A Ob = new B( );   // object of type B
        A Oc = new C( );  // object of type C
        Ob.disp();
        Oc.disp( );
    }
}
```

```
A Oa;
B Ob= new disp();
Oa=Ob;
Oa.disp( );
C Oc= new disp();
Oa=Oc;
Oa.disp( );
```

Example 2

```
abstract class Shape{
         abstract void draw();
}
class Rectangle extends Shape {
               void draw(){
                      System.out.println("drawing rectangle");
               }
}
class Circle extends Shape {
             void draw(){
                      System.out.println("drawing circle");
               }
}
```

Example 2

```
class TestAbstraction{
        public static void main(String args[]){
                Shape Co=new Circle( );
                Co.draw( );

               Shape Ro=new Rectangle( );
                Ro.draw();
      }
}
```

Lab program # 17

```
abstract class Shape  {
  double b, h, r;
  abstract double Area();
}

class Triangle extends Shape {
        Triangle(double d1, double d2)   {
                        b=d1;
                        h=d2;
        }
         double Area()
        {
                 return (b*h)/2;
       }
}
```

• • •

```
class Rectangle extends Shape   {
  Rectangle(double d1, double d2)  {
          b=d1;
          h=d2;
  }
 double area()    {
         return b*h;
  }
}
class Circle extends Shape
{
          Circle(double d1)     {
                    r=d1;
          }
     double area()   {
             return 3.142*r*r;
  }
}
```

• • •

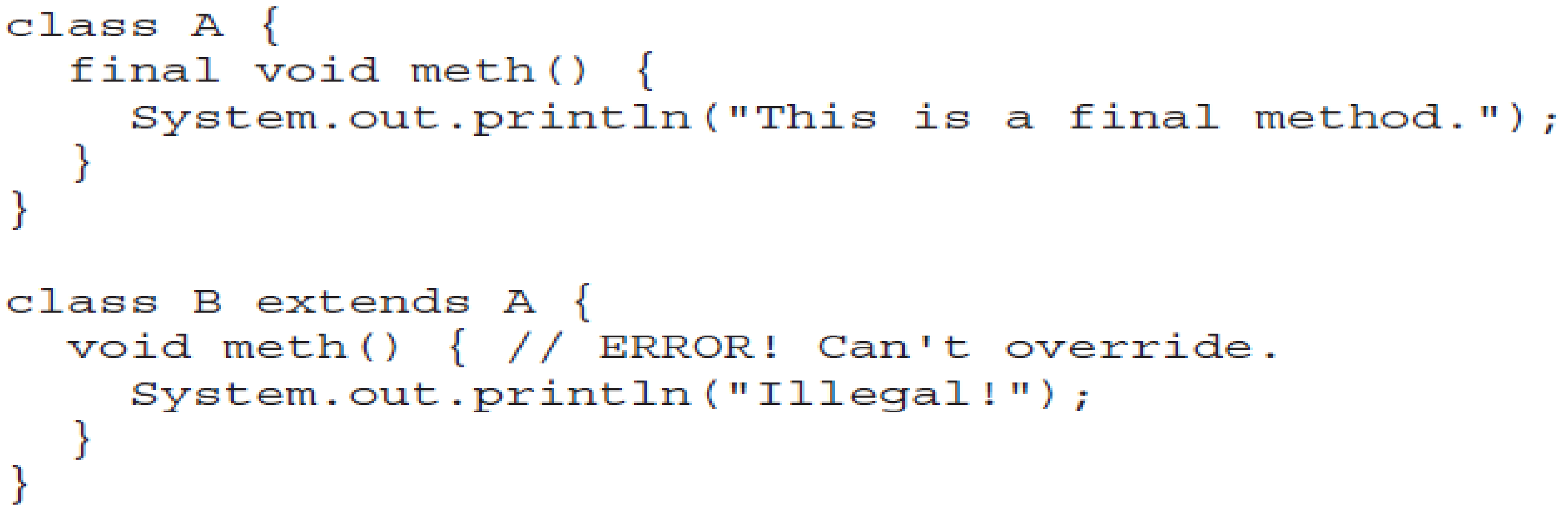
```
class AbstractClassExample
{
    public static void main(String arg[])
    {
        Shape To=new Triangle(4.3, 5.3);
        Shape Ro=new Rectangle(2.4, 4.2);
        Shape Co=new Circle(10.5);

        System.out.println("Area of Triangle is “ + To.Area());
        System.out.println("Area of Rectangle is “ + Ro.Area());
        System.out.println("Area of Circle is “ + Co.Area());

    }
}
```

Using final with Inheritance

- **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.

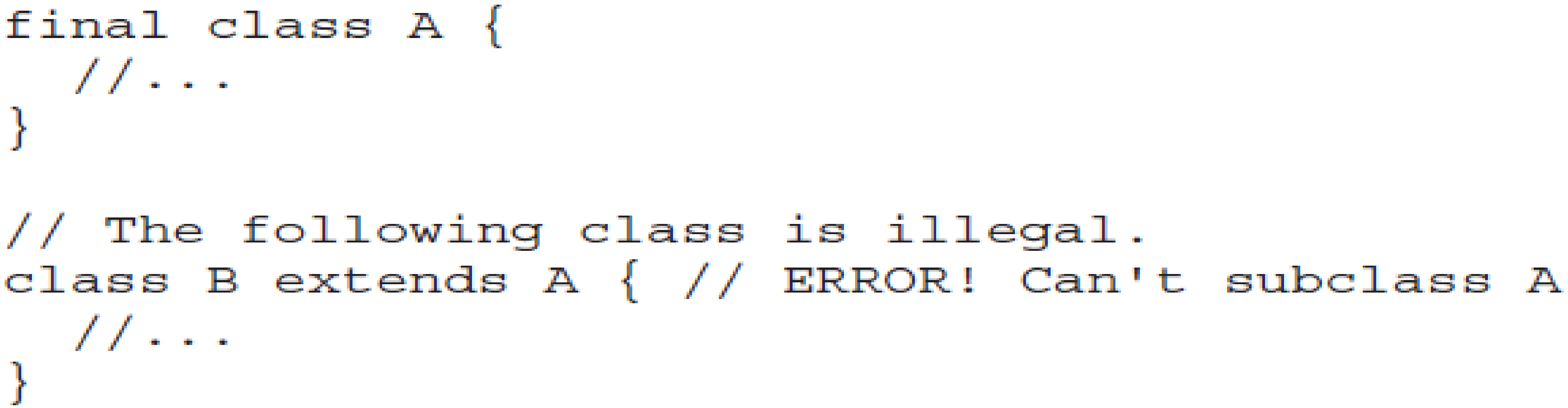


```
class A {
  final void meth() {
    System.out.println("This is a final method.");
  }
}

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

Using final with Inheritance

- Using final to Prevent Inheritance
- Sometimes you will want to prevent a class from being inherited.
- Declaring a class as final implicitly declares all of its methods as final, too.



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

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

THANK YOU