Unit 2: Inheritance

Inheritance

- It allows the creation of hierarchical classification.
- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
  - That is, the child class inherits the methods and data defined for the parent class
- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- *Software reuse* is at the heart of inheritance
- **Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent**

General Form

class *subclass-name extends superclass-name {*

    // body of class

}

- In Java, we use the reserved word `extends` to establish an inheritance relationship
- For Example

```
Class Vehicle {
    //some variables & methods
}
Class Car extends Vehicle {
    //class content
}
```

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with `public` visibility are inherited;
  - But those with `private` visibility are not
- But `public` variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

```
class A {
    int i; // public by default
    private int j; // private to A
      void setij(int x, int y) {
            i = x;
            j = y;
      }
    }
    class B extends A {
      int total;
      void sum() {
            total = i + j; // ERROR, j is not accessible here
      }
    }
    class Access {
      public static void main(String args[]) {
            B subOb = new B();
            subOb.setij(10, 12);
            subOb.sum();
            System.out.println("Total is " + subOb.total);
      }
    }
```

NOTE:

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.

```
// This program uses inheritance to extend Box.
class Box {
    double width;
    double height;
    double depth;
    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }
    // constructor used when all dimensions specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
```

```
    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }
    // constructor used when cube is created
    Box(double len) {
      width = height = depth = len;
    }
    // compute and return volume
    double volume() {
      return width * height * depth;
    }
}
```

```
// Here, Box is extended to include weight.
class BoxWeight  extends Box {
    double weight; // weight of box

    // constructor for BoxWeight
    BoxWeight(double w, double h, double d, double m) {
        width = w;
        height = h;
        depth = d;
        weight = m;
    }
}
```

OUTPUT:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

```
class DemoBoxWeight {
    public static void main(String args[]) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        double vol;
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
    }
}
```

A Superclass Variable Can Reference a Subclass Object

```
class RefDemo {
      public static void main(String args[]) {
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
        Box plainbox = new Box();
        double vol;
        vol = weightbox.volume();
        System.out.println("Volume of weightbox is " + vol);
        System.out.println("Weight of weightbox is " +weightbox.weight);
        System.out.println();
        // assign BoxWeight reference to Box reference
        plainbox = weightbox;
        vol = plainbox.volume(); // OK, volume() defined in Box
        System.out.println("Volume of plainbox is " + vol);
        /* The following statement is invalid because plainbox does not define a weight
              member. */
        // System.out.println("Weight of plainbox is " + plainbox.weight);
      }
}
```

Using super

- Previous example (Eg. Box, BoxWeight etc)was not efficient or robust as it could be.
  - For example, the constructor for **BoxWeight explicitly** initializes the **width,height, and depth fields of class Box.**
- However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private).
  - In this case, there would be no way for a subclass to directly access or initialize these variables on its own.

- Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem.
  - Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super.**
- **super has two general forms.**
  - **The first calls the superclass' constructor.**
  - **The second is** used to access a member of the superclass that has been hidden by a member of a subclass.

- Syntax
  1. super(*arg-list*);
  2. super.*member*

NOTE: **super()** must always be the first statement executed inside a subclass' constructor.

```
class Box {
        private double width;
        private double height;
        private double depth;
        // construct clone of an object
        Box(Box ob) { // pass object to constructor
                width = ob.width;
                height = ob.height;
                depth = ob.depth;
        }
        // constructor used when all dimensions specified
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
        // constructor used when no dimensions specified
        Box() {
                width = -1; // use -1 to indicate
                height = -1; // an uninitialized
                depth = -1; // box
        }
```

```
        // constructor used when cube is created
        Box(double len) {
                width = height = depth = len;
        }
        // compute and return volume
        double volume() {
                return width * height * depth;
        }
}

class BoxWeight extends Box {
        double weight; // weight of box
        // construct clone of an object
        BoxWeight(BoxWeight ob) { // pass object to constructor
                super(ob);
                weight = ob.weight;
        }
        // constructor when all parameters are specified
        BoxWeight(double w, double h, double d, double m) {
                super(w, h, d); // call superclass constructor
                weight = m;
        }
        // default constructor
        BoxWeight() {
                super();
                weight = -1;
        }
```

```
        // constructor used when cube is created
        BoxWeight(double len, double m) {
            super(len);
            weight = m;
        }
    }
class DemoSuper {
        public static void main(String args[]) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        BoxWeight mybox3 = new BoxWeight(); // default
        BoxWeight mycube = new BoxWeight(3, 2);
        BoxWeight myclone = new BoxWeight(mybox1);
        double vol;
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();
```

```
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
        System.out.println();
        vol = mybox3.volume();
        System.out.println("Volume of mybox3 is " + vol);
        System.out.println("Weight of mybox3 is " + mybox3.weight);
        System.out.println();
        vol = myclone.volume();
        System.out.println("Volume of myclone is " + vol);
        System.out.println("Weight of myclone is " + myclone.weight);
        System.out.println();
        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
        System.out.println("Weight of mycube is " + mycube.weight);
        System.out.println();
    }
}
```

OUTPUT:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3
Volume of mycube is 27.0
Weight of mycube is 2.0
```

```
// 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();
	}
}
```

Creating a Multi-level Hierarchy

- CAN YOU WRITE A PROGRAM ON THIS?

When Constructors Are Called

- WRITE A PROGRAM & EXPLAIN

```
class A {
	A() {
		System.out.println("Inside A's constructor.");
	}
}
// Create a subclass by extending class A.
class B extends A {
	B() {
		System.out.println("Inside B's constructor.");
	}
}
class C extends B {
	C() {
		System.out.println("Inside C's constructor.");
	}
}
class CallingCons {
	public static void main(String args[]) {
		C c = new C();
	}
}
```

OUTPUT:
?

OUTPUT:
Inside A's constructor.
Inside B's constructor.
Inside C's constructor.

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.

```
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}
```

```
class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    // display k – this overrides show() in A
    void show() {
        System.out.println("k: " + k);
    }
}
```

```
class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show(); // this calls show() in B
    }
}
```

OUTPUT:
k: 3

```
class A {
        int i, j;
        A(int a, int b) {
                i = a;
                j = b;
        }
        // display i and j
        void show() {
                System.out.println("i and j: " + i + " " + j);
        }
}
```

```
class B extends A {
        int k;
        B(int a, int b, int c) {
                super(a, b);
                k = c;
        }
        //Overridden Method
        void show() {
                // this calls A's show()
                super.show();
                System.out.println("k: " + k);
        }

}
```

```
class Override {
        public static void main(String args[]) {
                B subOb = new B(1, 2, 3);
                // this calls show() in B
                subOb.show();
        }
}
```

```
class A {
        int i, j;
        A(int a, int b) {
                i = a;
                j = b;
        }
        // display i and j
        void show() {
                System.out.println("i and j: " + i + " " + j);
        }
}
class B extends A {
        int k;
        B(int a, int b, int c) {
                super(a, b);
                k = c;
        }
        void show(String s) {
                System.out.println(" DISPLAY STRING: " + s);
        }

}
```

```
class Override {
        public static void main(String args[]) {
                B subOb = new B(1, 2, 3);
                // this calls show() in B
                subOb.show();
                // this calls show(String s) in B
                subOb.show(" WELCOME TO BMSCE ");
        }
}
```

Dynamic Method Dispatch

- Dynamic method dispatch 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.
- ***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**
  - 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.

```
class A {
    void callme() {
        System.out.println("Inside A's callme method");
    }
}
class B extends A {
    void callme() {
        System.out.println("Inside B's callme method");
    }
}
class C extends A {
    void callme() {
        System.out.println("Inside C's callme method");
    }
}
```

```
class Dispatch {
    public static void main(String args[]) {
        A a = new A();
        B b = new B();
        C c = new C();
        A r;
        //super ref variable is referring subclass object
        r = a;
        r.callme();


        r = b;
        r.callme();

        r = c;
        r.callme();
    }
}
```

OUTPUT:
Inside A's callme method
Inside B's callme method
Inside C's callme method

```
// Using run-time polymorphism.
class Figure {
    double dim1;
    double dim2;
    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }
    double area() {
        System.out.println("Area for Figure is undefined.");
        return 0;
    }
}
class Rectangle extends Figure {
    Rectangle(double a, double b) {  super(a, b);  }
    // override area for rectangle
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
```

```
class Triangle extends Figure {
    Triangle(double a, double b) {      super(a, b);    }
    // override area for right triangle
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}
```

```
class FindAreas {
    public static void main(String args[]) {
        Figure f = new Figure(10, 10);
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref;
        figref = r;
        System.out.println("Area is " + figref.area());
        figref = t;
        System.out.println("Area is " + figref.area());
        figref = f;
        System.out.println("Area is " + figref.area());
    }
}
```

OUTPUT:

Inside Area for Rectangle.

Area is 45

Inside Area for Triangle.

Area is 40

Area for Figure is undefined.

Area is 0

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Product
{
    // contents
}
```

Abstract Classes

- An abstract class often contains abstract methods with no definitions (like an interface)
- Unlike an interface, the `abstract` modifier must be applied to each abstract method
- Also, an abstract class typically contains non-abstract methods with full definitions
- A class declared as abstract does not have to contain abstract methods -- simply declaring it as abstract makes it so

Abstract Classes

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered **abstract**
- An abstract method **cannot** be defined as **`final`** or **`static`**
- The use of abstract classes is an important element of software design
  - it allows us to establish common elements in a hierarchy that are too generic to instantiate

```
abstract class A {
    abstract void callme();
    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();
    }
}
```

B's implementation of callme.
This is a concrete method.

```
abstract class Figure {
    double dim1;
    double dim2;
    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }
    // area is now an abstract method
    abstract double area();
}
class Rectangle extends Figure {
    Rectangle(double a, double b) {  super(a, b);}
    double area() {
        System.out.println("Inside Area for Rectangle.");
        return dim1 * dim2;
    }
}
```

```
class Triangle extends Figure {
    Triangle(double a, double b) {   super(a, b);   }
    double area() {
        System.out.println("Inside Area for Triangle.");
        return dim1 * dim2 / 2;
    }
}
class AbstractAreas {
    public static void main(String args[]) {
        // Figure f = new Figure(10, 10); // illegal now
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref; // this is OK, no object is created
        figref = r;
        System.out.println("Area is " + figref.area());
        figref = t;
        System.out.println("Area is " + figref.area());
    }
}
```

```
Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0
```

Another Example of Abstract

```
//abstract class
abstract class Person {

	private String name;
	private String gender;

	public Person(String nm, String gen){
		this.name=nm;
		this.gender=gen;
	}

	//abstract method
	public abstract void work();

	@Override
	public String toString(){
		return "Name="+this.name+"::Gender="+this.gender;
	}

	public void changeName(String newName) {
		this.name = newName;
	}
}
```

```
class Employee extends Person {

	private int empId;

	public Employee(String nm, String gen, int id) {
		super(nm, gen);
		this.empId=id;
	}

	@Override
	public void work() {
		if(empId == 0){
			System.out.println("Not working");
		}else{
			System.out.println("Working as employee!!");
		}
	}
}
```

```
class AbstractEmpDemo {
    public static void main(String args[]){
        //coding in terms of abstract classes
        Person student = new Employee("DurgaBhavani","Female",0);
        Person employee = new Employee("Syed","Male",123);
        student.work();
        employee.work();
        //using method implemented in abstract class - inheritance
        employee.changeName("Pankaj Kumar");
        System.out.println(employee);
    }
}
```

```
Not working
Working as employee!!
Name=Pankaj Kumar::Gender=Male
```

Using final with Inheritance

```
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 to Prevent Inheritance

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

```
// Circle.java:  Contains both Circle class and its user class
//Add Circle class code here
class DemoRadCir
{
	public static void main(String args[])
	{
		Circle aCircle;  // creating reference
		aCircle = new Circle(); // creating object
		aCircle.x = 10;  // assigning value to data field
		aCircle.y = 20;
		aCircle.r = 5;
		double area = aCircle.area(); // invoking method
		double circumf = aCircle.circumference();
		System.out.println("Radius="+aCircle.r+" Area="+area);
		System.out.println("Radius="+aCircle.r+" Circumference ="+circumf);
	}
}
```

REFERENCES

- COMPLETE REFERENCE JAVA HANDBOOK BY HERBERT SCHILDT