Chapter: 4

Inheritance in Java/C#

Inheritance fundamentals, multilevel class hierarchy, superclass references to subclass objects, Methods overriding, abstract classes, final, Object class

4.1 Inheritance fundamentals

The main theme is similar to C++, but ***multiple base*** class is not allowed (Which is allowed in C++). ***Superclass*** act as ***base*** class and ***subclass*** act as ***derived*** class.

- ❖ In the language of Java, a class that is *inherited* is called a ***superclass***.
- ❖ The class that does the *inheriting* is called a ***subclass***.

☞ A *subclass* is a specialized version of a *superclass*. It inherits all of the variables and methods defined by the *superclass* and adds its own, unique elements.

☞ Java supports inheritance by allowing one class to ***incorporate another class*** into its ***declaration***. This is done by using the ***extends*** keyword. Thus, the *subclass adds to (extends) the superclass*.

```
class TwoDShape {
   double width, height;
   void showDim() {System.out.println("Width and height are " +
                          width + " and " + height);  }           }

                  // A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
   String style;
   double area() { return width * height / 2; }
   void showStyle() { System.out.println("Triangle is " + style);}
   }
```

```
class Shapes { public static void main(String args[]) {
Triangle t1 = new Triangle();
Triangle t2 = new Triangle();
t1.width = 4.0;      t1.height = 4.0;     t1.style = "filled";
t2.width = 8.0;      t2.height = 12.0;    t2.style = "outlined";
System.out.println("Info for t1:");   t1.showStyle();   t1.showDim();
System.out.println("Area is " + t1.area());
System.out.println();
System.out.println("Info for t2:");   t2.showStyle();   t2.showDim();
System.out.println("Area is " + t2.area());
}}
```

☞ Here, ***TwoDShape*** defines the attributes of a "***generic***" two-dimensional shape, such as a square, rectangle, triangle, and so on.

☞ The ***Triangle*** class creates a specific type of ***TwoDShape***, in this case, a triangle. ***Triangle*** includes all of the members of its superclass, ***TwoDShape*** (Also, inside ***main( )***, objects t1 and t2 can refer to members of superclass).

☞ Being a *superclass* for a *subclass* does not mean that the *superclass* cannot be used by itself. For example, the following is perfectly valid:

```
TwoDShape shape = new TwoDShape();
shape.width = 10;   shape.height = 20;   shape.showDim();
```

- ▶ Of course, an object of ***TwoDShape*** has no knowledge of or access to any *subclasses* of ***TwoDShape***.

❒ ***General form:*** The general form of a class declaration that inherits a *superclass* is:

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

☞ ***Restriction 1:*** A *subclass* can have only one *superclass* that you create. Java does not support the inheritance of multiple superclasses into a single subclass. (This differs from C++, which allows multiple superclass/base). [You can, however, create a hierarchy of inheritance in which a ***subclass*** becomes a ***superclass*** of another ***subclass***. Of course, no class can be a ***superclass*** of itself.]

☞ ***Restriction 2:*** Even though a *subclass* includes all of the members of its *superclass*, it cannot access those members of the *superclass* that have been declared ***private*** (similar as C++). Remember that a class member that has been declared private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

- ▶ ***Accessing private members through accessor methods :*** Here is a rewrite of the TwoDShape and Triangle classes that uses methods to access the private instance variables width and height:

```
class TwoDShape {          private double width;     // these are
                           private double height;    // now private

          // Accessor methods for width and height.
                           double getWidth() { return width; }
                           double getHeight() { return height; }

          /* . . other codes – methods . . */          }   }

          // A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
          String style;
          double area() { return getWidth() * getHeight() / 2; }   /* Use accessor methods provided by superclass */
          /* . . other codes – methods . . */          }   }
```

❒ ***When to make an instance variable private:*** There are no hard and fast rules, but here are two general principles

- ▶ If an instance variable is to be used ***only by methods defined within its class***, then it should be made ***private***.
- ▶ If an instance variable must be ***within certain bounds***, then it should be ***private*** and made available only through ***accessor*** methods. This way, you can

4.2 Constructors and Inheritance *(Recall C/C++ 11.17 Inheritance with Constructor-Destructor. NOT SIMILAR.)*

Both superclasses and subclasses can have their own constructors. The constructor for the superclass constructs the superclass portion of the object, and the constructor for the subclass constructs the subclass part. Here two cases arise:

- ▶ Only *sub-class* defines the constructor
- ▶ Both *super-classes* and *sub-classes* defines constructors

- **Only *sub-class* defines the constructor:** If only ***sub-class*** defines the constructor and ***super-classes*** doesn't, then The ***super-class*** portion of the object is ***constructed automatically*** using its ***default constructor*** when we simply construct the subclass object.

☞ ***Example:*** Consider the previous ***TwoDShape's "accessor version"***

```
class TwoDShape {     /*Similar codes with  Accessor methods */          }
// A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
    private String style;
    // Constructor
    Triangle(String s, double w, double h){setWidth(w);  setHeight(h);  style = s; }
    double area() { return getWidth() * getHeight() / 2; }
    void showStyle() { System.out.println("Triangle is " + style); }       }
```

```
class Shapes3 {
public static void main(String args[]) {
  Triangle t1 = new Triangle("filled", 4.0, 4.0);
  System.out.println("Info for t1: ");
  t1.showStyle();
  t1.showDim();
  System.out.println("Area is " + t1.area());
}}
```

▶ Here, ***Triangle***'s constructor initializes the members of ***TwoDClass*** that it inherits along with its own style field.

- Both ***super-classes*** and ***sub-classes*** defines constructors(introducing ***super***): In this case both the ***superclass*** and ***subclass*** constructors must be executed. And you must use another of Java's keywords, ***super***, which has two general forms:

super(parameter-list);

A ***subclass*** can call a constructor defined by its ***superclass*** by use of the above form of ***super***.

❖ Here, **parameter-list** specifies any parameters needed by the constructor in the ***superclass***.

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

```
class TwoDShape {  /*Similar codes with  Accessor methods */
                   /* Parameterized constructor for TwoDShape*/
                   TwoDShape(double w, double h) {
                                width = w; height = h; }
               }
class Triangle extends TwoDShape { /* other codes */
        /* Use super( ) to execute the TwoDShape constructor. */
        Triangle(String s, double w, double h) {
                    super(w, h);      // call superclass constructor
                    style = s } /* this constructor only initialize style*/
        /* other codes */      }
class Shapes4 {public static void main(String args[]){/*… */}}
```

➲ Here, ***Triangle()*** calls ***super()*** with the parameters ***w*** and ***h***. This causes the ***TwoDShape()*** constructor to be called, which initializes **width** and **height** using these values.

⇨ ***Triangle*** no longer initializes these values itself. It need only initialize the value unique to it: i.e. "***style***".

[This leaves TwoDShape free to construct its subobject in any manner that it so chooses. Furthermore, TwoDShape can add functionality about which existing subclasses have no knowledge, thus preventing existing code from breaking.]

super.member

Here ***super*** acts like ***this*** reference, except that it always refers to the ***superclass*** of the ***subclass*** in which it is used.

❖ ***member*** can be either a ***method*** or an ***instance variable***.

❖ This form of ***super*** is most applicable to situations in which member names of a ***subclass*** hide members by the same name in the ***superclass***. Consider following

```
class A { int i; }
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(); }}
```

OUTPUT:

```
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***.

➲ ***super*** can also be used to call methods that are hidden by a ***subclass***.

☞ Any form of constructor defined by the ***superclass*** can be called by ***super()***. The constructor executed will be the one that matches the arguments. For example:

```
class TwoDShape {  /*Similar codes with  Accessor methods */
        TwoDShape() { width = height = 0.0; }/* A default constructor.*/
/* Parameterized constructor */
        Triangle(String s, double w, double h) {
                    super(w, h);     // call superclass constructor
                    style = s } /* this constructor only initialize style*/
                    /* other codes */  }
/* constructor for equal width and height.*/
        TwoDShape(double x) { width = height = x; }
}
```

```
class Triangle extends TwoDShape { private String style;
        Triangle() { super();
                    style = "none"; }   /* default constructor.*/
    // Constructor
        Triangle(String s, double w, double h) {
                    super(w, h);  // call superclass constructor
                    style = s; }
    // One argument constructor.
        Triangle(double x) {
                    super(x);     // call superclass constructor
                    style = "filled"; }
/* other codes */}
```

☛ NOTE:

➲ When a subclass calls ***super()***, it is calling the constructor of its immediate ***superclass***. Thus, ***super()*** always refers to the ***superclass*** immediately above the calling class. This is true even in a multilevel hierarchy.

➲ Also, ***super()*** must always be the first statement executed inside a subclass constructor.

4.3 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 traits found in all of its ***superclasses***. In this case, **C *inherits all aspects*** of ***B*** and ***A***. Consider following example:

- ***Eg: Triangle*** is used as a ***superclass*** to ***ColorTriangle***. Because of inheritance, ***ColorTriangle*** can make use of the previously defined classes of ***Triangle*** and ***TwoDShape***, adding only the extra information it needs for its own, specific application.

- Notice that, ***super()*** always refers to the constructor in the closest ***superclass***. The ***super()*** in ***ColorTriangle*** calls the ***constructor*** in ***Triangle***. The ***super()*** in ***Triangle*** calls the constructor in ***TwoDShape***.

☞ In a class hierarchy, if a ***superclass*** constructor requires parameters, then all ***subclasses*** must pass those parameters (Even a ***subclass*** don't have its own parameter).

```
class TwoDShape {
        private double width;
        private double height;
        TwoDShape() {      width = height = 0.0;            }
        TwoDShape(double w, double h) { width = w; height = h; }
        TwoDShape(double x) { width = height = x; }

        double getWidth() { return width; }          // Accessor method
        double getHeight() { return height; }        //Accessor method

        void setWidth(double w) { width = w; }
        void setHeight(double h) { height = h; }
        void showDim() {System.out.println("Width and height are " +
                                    width + " and " + height); }
        }

// Extend TwoDShape.
class Triangle extends TwoDShape {
        private String style;
        Triangle() { super(); style = "none"; }
        Triangle(String s, double w, double h) { super(w, h);  style = s; }
        Triangle(double x) { super(x); style = "filled"; }

        double area() {return getWidth() * getHeight() / 2;}
        void showStyle() {System.out.println("Triangle is " + style);}
        }
```

```
// Extend Triangle.
class ColorTriangle extends Triangle {
    private String color;
    ColorTriangle(String c, String s,double w, double h) {
                super(s, w, h); color = c; }
    String getColor() { return color; }
    void showColor() {
                System.out.println("Color is " + color);}
    }

class Shapes { public static void main(String args[]) {
    ColorTriangle t1 =
          new ColorTriangle("Blue", "outlined", 8.0, 12.0);
    /* ColorTriangle inherits Triangle, which is descended from
TwoDShape, so ColorTriangle includes all members of Triangle and
                            TwoDShape.*/

    System.out.println("Info for t1: ");
    t1.showStyle();
    t1.showDim();
    t1.showColor();
    System.out.println("Area is " + t1.area());

}}
```

❒ ***CONSTRUCTORS EXECUTION IN CLASS HIERARCHY:*** constructors complete their execution in order of derivation, from ***superclass*** to ***subclass***. Also, since ***super()*** must be the first statement executed in a ***subclas's*** constructor, this order is the same whether or not ***super()*** is used. If ***super()*** is't used, then the default (parameterless) constructor of each ***superclass*** will be executed.

```
class A { A() { System.out.println("Constructing A."); } }
class B extends A { B() { System.out.println("Constructing B."); } }
class C extends B { C() { System.out.println("Constructing C."); } }
class OrderOfConstruction {
    public static void main(String args[]) { C c = new C();   }}
```

OUTPUT:

```
Constructing A.
Constructing B.
Constructing C.
```

☞ A ***superclass*** has no knowledge of any ***subclass***, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the ***subclass***. Therefore, it must complete its execution first *(Recall examples of C/C++ 11.17).*

4.4 Superclass References and Subclass Objects

In java ***type compatibility*** is strictly enforced. Although, ***automatic type promotions*** occurs only for **`primitive type`**. But a ***`reference`*** variable for one class type cannot normally refer to an object of another class type. For example, consider the following:

```
class X { int a; X(int i) { a = i; } }
class Y { int a; Y(int i) { a = i; } }
```

```
class IncompatibleRef { public static void main(String args[]) { X x = new X(10);
                                                                  X x2;
                                                                  Y y = new Y(5);
                                                                  x2 = x;    // OK, both of same type
                                                                  x2 = y;   /* Error, not of same type */}}
```

☞ Here, even though ***`class X`*** and ***`class Y`*** are structurally the same, it is not possible to assign an ***X*** reference to a ***Y*** object because they have different types. In general, an object reference variable can refer only to objects of its type.

❒ ***Exception to Java's strict type enforcement:*** A reference variable of a **superclass** can be assigned a reference to an object of any **subclass** *derived from* that **superclass**. In other words, 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(j);  b = i; }        }

class SupSubRef {public static void main(String args[]) {X x = new X(10);
                                                          X x2;
                                                          Y y = new Y(5, 6);
```

```
    x2 = x;              // OK, both of same type
    x2 = y;              // still Ok because Y is derived from X

    // X references know only about X members
    x2.a = 19;           // OK
    x2.b = 27;           // Error, X doesn't have a b member
 }}
```

☞ Here, Y is now derived from X; thus, it is permissible for x2 to be assigned a reference to a Y object.

☞ NOTE: 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***, because the ***superclass*** has no knowledge of what a ***subclass*** adds to it. This is why ***x2*** can't access ***b*** even when it refers to a ***Y*** object.

❒ **Calling *constructors* in class hierarchy and superclass-subclass reference :** An important place where ***subclass references*** are assigned to ***superclass variables*** is when constructors are called in a class hierarchy.

☞ Sometimes a **class** to define a **constructor** that takes an object of the **class** as a **parameter**. This allows the class to construct a copy of an object. Subclasses of such a class can take advantage of above feature. For example, ***`TwoDShape`*** and ***`Triangle`*** with constructors that take an **object** as a **parameter**.

- Here, ***t2*** is constructed from ***t1*** and is, thus, identical.
- Pay special attention to this ***Triangle*** constructor:

```
// Construct an object from an object.
Triangle(Triangle ob) {
   super(ob);  // pass object to TwoDShape constructor
   style = ob.style;      }
```

It receives an object of type ***Triangle*** and it passes that object (through ***super***) to following TwoDShape constructor:

```
// Construct an object from an object.
TwoDShape(TwoDShape ob) {
   width = ob.width;
   height = ob.height;     }
```

- The key point is that ***TwoDshape()*** is expecting a ***TwoDShape*** object. However, ***Triangle()*** passes it a ***Triangle*** object. The reason this works is because, as explained, a ***superclass reference can refer to a subclass object***. (i.e. Here the type promotion occurs !!!).

```
class TwoDShape {
        private double width;
        private double height;
        TwoDShape() { width = height = 0.0; }
        TwoDShape(double w, double h) { width = w; height = h; }
        TwoDShape(double x) { width = height = x; }
          // Construct an object from an object. Object parameter Constructor
        TwoDShape(TwoDShape ob) { width = ob.width; height = ob.height; }
        // Accessor methods
        // Other methods            }

                        // A subclass of TwoDShape for triangles.
class Triangle extends TwoDShape {
        private String style;
        Triangle() { super(); style = "none"; }
        Triangle(String s, double w, double h) { super(w, h);  style = s; }
        Triangle(double x) { super(x);  style = "filled"; }
// Construct an object from an object. Object parameter Constructor
       Triangle(Triangle ob) { super(ob); // pass object to TwoDShape constructor
                                    style = ob.style; }
       /* Other methods */       }

class Shapes7 { public static void main(String args[]) {
                              Triangle t1 = new Triangle("outlined", 8.0, 12.0);
           // make a copy of t1
                              Triangle t2 = new Triangle(t1);          }}
```

- Thus, it is perfectly acceptable to pass ***TwoDShape()*** a reference to an object of a class derived from ***TwoDShape***. Because the ***TwoDShape()*** constructor is initializing only those portions of the subclass object ***that are members*** of ***TwoDShape***, ***it doesn't matter that the object might also contain other members added by derived classes.***

4.5 Method Overriding (Recall C/C++ *virtual function* 13.2)

In a class hierarchy, when a method in a ***subclass*** has the same **return type** and **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***. Consider the following:

```
class A{ int i, j;
        A(int a, int b) { i = a; j = b; }
/* following method will be overridden */
        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
```

- When ***show()*** is invoked on an object of type ***B***, the version of ***show()*** defined within ***B*** is used. That is, the version of ***show()*** inside ***B*** overrides the version declared in ***A***.

- ***<u>Use super to access superclass version of an overridden method:</u>*** If you want to access the ***superclass*** version of an ***overridden method***, you can do so by using ***super***. For example, in this version of ***B***, the superclass version of ***show()*** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
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(){  super.show(); // this calls A's show()
                                      System.out.println("k: " + k); }
                  }
```

Substituting this version of ***show()*** into the previous program, we will see the following output:

```
i and j: 1 2
k: 3
```

- Here, ***super.show()*** calls the ***superclass*** version of ***show()***.

- This ***show()*** in ***B*** overrides the one defined by ***A***. Use ***super*** to call the version of ***show()*** defined by **superclass *A***.

- ***<u>Signature of method and Method-overriding:</u>*** Method overriding occurs only when the signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For example, consider this modified version of the preceding example:

```
class A { /*……  ……*/
          void show() { System.out.println("i and j: " + i + " " + j); }            }

class B extends A {                     /*……  ……*/
            /* Because signatures differ, this show( ) simply overloads show( ) in superclass A. */
  void show(String msg){ System.out.println(msg + k); } /* signature changed*/ }
```

```
class Overload {
public static void main(String args[]) {
    B subOb = new B(1, 2, 3);
    subOb.show("This is k: ");  // calls show() in B
    subOb.show();           /* calls show() in A */  }}
```

- The version of ***show()*** in ***B*** takes a string parameter. This makes its signature different from the one in ***A***, which takes no parameters. Therefore, no overriding (or name hiding) takes place.

OUTPUT:

```
This is k: 3
i and j: 1 2
```

- ***<u>Signature of a method:</u>*** **Signature** of a method means the "**method name**" and its "**parameter list**".

❐ ***Dynamic method dispatch (run-time overriding):*** Method overriding forms the basis for one of Java's most powerful concepts: **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***. Java implements run-time polymorphism by ***dynamic method dispatch***.

☞ Since a ***superclass reference variable*** can refer to a ***subclass object***. Java uses this fact to resolve calls to overridden methods at ***run time***.

➢ 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.]

☞ Here is an ***example*** that illustrates *dynamic method dispatch*:

class Sup { ***void*** who() { ***System.out.println***("who() in Sup"); } }
class Sub1 ***extends*** Sup{ ***void*** who(){ ***System.out.println***("who() in Sub1"); } }
class Sub2 ***extends*** Sup{ ***void*** who(){ ***System.out.println***("who() in Sub2"); } }
 	
class DynDispDemo { **public static void main(String args[])** {
Sup superOb = **new** Sup();
Sub1 subOb1 = **new** Sub1();
Sub2 subOb2 = **new** Sub2();
	Sup supRef; */* only reference variable. Not object */*
	supRef = superOb; supRef.who();
	supRef = subOb1; supRef.who();
	supRef = subOb2; supRef.who(); }}

	OUTPUT: who() in Sup who() in Sub1 who() in Sub2

- ▶ This program creates a **superclass** called ***Sup*** and two **subclasses** of it, called ***Sub1*** and ***Sub2***. ***Sup*** declares a method called ***who()***, and the **subclasses** override it.
- ▶ Inside the ***main()*** method, objects of type ***Sup***, ***Sub1***, and ***Sub2*** are declared.
- ▶ A reference of type ***Sup***, called ***supRef***, is declared also. The program then assigns a reference to each type of object to ***supRef*** and uses that reference to call ***who()***.
- ▶ As the output shows, the version of ***who()*** executed is ***determined by the type of object being referred to at the time of the call***, ***not by the class type*** of ***supRef***.

❐ ***C++ virtual function and Java Overridden methods:*** **Overridden methods** in Java are equivalent in purpose and similar in operation to **virtual functions** in **C++** (recall C/C++ 13.2).

❐ ***Example(Use of array declaration for overriding methods):*** Notice in ***main()*** that shapes is declared as an array of ***TwoDShape*** objects. However, the elements of this array are assigned ***Triangle***, ***Rectangle***, and ***TwoDShape*** references. This is valid because, as explained, a ***superclass reference*** can refer to a ***subclass object***. The program then cycles through the array, displaying information about each object. Although quite simple, this illustrates the power of both ***inheritance*** and ***method overriding***.

☞ The type of object referred to by a ***superclass reference variable*** is determined at ***run time*** and acted on accordingly.

```
class DynShapes{ public static void main(String args[]){ TwoDShape shapes[] = new TwoDShape[5];
          shapes[0] = new Triangle("outlined", 8.0, 12.0);
          shapes[1] = new Rectangle(10);
          shapes[2] = new Rectangle(10, 4);
          shapes[3] = new Triangle(7.0);
          shapes[4] = new TwoDShape(10, 20, "generic");
          for(int i=0; i < shapes.length; i++) {
                    System.out.println("object is " + shapes[i].getName());
               /* The proper version of area( ) is called for each shape. */
                    System.out.println("Area is " + shapes[i].area());
                    System.out.println();      }      }}
```

[Overridden methods are another way that Java implements the "***one interface, multiple methods***" aspect of polymorphism. By combining ***inheritance*** with ***overridden*** methods, a ***superclass*** can define the ***general form of the methods*** that will be used by all of its subclasses.]

4.6 Abstract Methods and Abstract Classes (Recall C/C++ 13.3 *Abstract Class and Pure Virtual function*)

Abstract class: It is a **superclass** that defines only 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 but does not, itself, provide an implementation of one or more of these methods.

❐ ***Declaring abstract class and abstract method:*** An ***abstract method*** is created by specifying the ***abstract type modifier***. To declare an *abstract method*, use this general form:

```
abstract type name(parameter-list);
```

- ➲ An *abstract method* contains ***no body*** and is, therefore, not implemented by the **superclass**.
- ➲ A subclass must override the *abstract method* —it cannot simply use the version defined in the **superclass**.
- ➲ The ***abstract*** modifier can be used only on ***instance methods***. It cannot be applied to ***static methods*** or to ***constructors***.
- ➲ A class that contains one or more ***abstract*** methods must also be declared as ***abstract*** by preceding its class declaration with the ***abstract modifier***.
    - ▶ Since an abstract class does not define a complete implementation, ***there can be no objects of an abstract*** class. Thus, *attempting to create an object of an abstract class by using* ***new*** *will result in a* **compile-time error**.
- ➲ When a **subclass** inherits an *abstract class*, it ***must implement all of the abstract methods*** in the **superclass**. If it doesn't, then the **subclass** ***must also be specified as abstract***. I.e ***abstract*** attribute is inherited until complete implementation is achieved.

❒ ***Example:*** Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the preceding program declares ***abstract double area()*** inside ***abstract class TwoDShape{ }***.

```
abstract class TwoDShape{       // variable declaration
                                // diffent versions of constructors
                String getName() { return name; }
                void showDim() { System.out.println("Width and height are " +width + " and " + height); }
                                // other normal methods
                                // accessor methods
                abstract double area();// Now, area() is abstract.
                                // other abstact methods
                   }
```

☞ All **subclasses** of **TwoDShape** must override **area()**.

☞ NOTICE that ***TwoDShape*** still includes the ***showDim()*** and ***getName()*** methods and that these are not modified by ***abstract***. It is perfectly acceptable—indeed, quite common—for an ***abstract class*** to contain ***concrete methods*** which a **subclass** is free to use as is. Only those methods declared as ***abstract*** need be overridden by **subclasses**.

4.7 Final

final Prevents Overriding: To prevent 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.

❒ ***final Prevents Inheritance:*** To prevent a class from being ***inherited*** jut precede its 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***: it is illegal for ***B*** to inherit ***A*** since ***A*** is declared as ***final***.

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

❒ *Using **final** with **Data Members:*** ***final*** can also be applied to member variables to turn them into **named constants**. If you precede a class variable's name with ***final***, its ***value cannot be changed throughout the lifetime*** of your program.

☞ You can, of course, give that variable an initial value. For Example:

```
class finl_data {    final int i = 0;
                     final int j= 1;
                     final int k = 2;
                     final int t = 3;
                     String msgs[] = { "Hey", "You", "There", "Sleepin" };
            // Return the message.
                     String get(int i) {  if(i >=0 & i < msgs.length) return msgs[i];
                                          else return "Invalid "; }
               }
```

```
class FinalD {
public  static  void  main(String  args[])  {
         finl_data err = new finl_data();
         System.out.println(err.get (err.i));
         System.out.println(err.get (err.k));
}}
```

▶ Notice how the ***final*** constants are used in ***main()***. Since they are members of the ***finl_data*** class, they must be accessed via an object of that class. They can also be inherited by subclasses and accessed directly inside those subclasses.

☞ NOTE: As a point of style, many Java programmers use uppercase identifiers for final constants.

❒ ***Other uses of final:*** ***final*** member variables can be used as ***static*** or as ***method parameters*** and with ***local variables***.

☞ Making a ***final*** member variable ***static*** lets you refer to the constant ***through its class name*** rather than ***through an object***. For example, if the constants in ***finl_data*** were modified by ***static***, then the ***println()*** statements in ***main()*** could look like this:

```
System.out.println(err.get(finl_data.i));
System.out.println(err.get(finl_data.k));
```

☞ Declaring a **parameter** ***final*** prevents it from being changed ***within the method***.

☞ Declaring a **local** variable ***final*** **prevents** it from being assigned a value **more than once**.

4.8 The Object Class

Object is the **implicit superclass** of all other classes. I.e. all other classes are **subclasses** of ***Object***. 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 can override the others.
- Notice two methods: equals( ) and toString( ):
  - The ***equals( )*** method compares 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. Also, this method is automatically called when an object is output using ***println( )***. Many classes override this method. Doing so allows them to **tailor a description** specifically *for the types of objects that they create*.
- Notice the unusual syntax in the return type for ***getClass( )***. This relates to Java's ***generics*** feature. ***Generics*** allow the **type of data** used by a ***class*** or ***method*** to be ***specified as a parameter***.