Jain College of Engineering and Research

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (AIML)

OOP with JAVA BCS306A

Module -3

Chapter -1 Inheritance

- In object-oriented programming, inheritance is a fundamental concept that enables the creation of hierarchical classifications.
- It involves the creation of a general class (superclass) defining common traits for a group of related items.
- Other, more specific classes (subclasses) can then inherit from this superclass, adding unique elements while retaining the inherited traits.
- In Java terminology, the inherited class is the superclass, and the inheriting class is the subclass.

1.1 Inheritance Basics

To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.

```
// A simple example of inheritance.

// Create a superclass.
class A {
  int i, j;

  void showij() {
    System.out.println("i and j: " + i + " " + j);
  }
}

// Create a subclass by extending class A.
class B extends A {
  int k;

  void showk() {
        System.out.println("k: " + k);
      }


  void sum() {
    System.out.println("i+j+k: " + (i+j+k));
  }
}

class SimpleInheritance {
  public static void main(String[] args) {
    A superOb = new A();
    B subOb = new B();

    // The superclass may be used by itself.
    superOb.i = 10;
    superOb.j = 20;
    System.out.println("Contents of superOb: ");
    superOb.showij();
    System.out.println();

    /* The subclass has access to all public members of
       its superclass. */
    subOb.i = 7;
    subOb.j = 8;
    subOb.k = 9;
    System.out.println("Contents of subOb: ");
    subOb.showij();
    subOb.showk();
    System.out.println();

    System.out.println("Sum of i, j and k in subOb:");
    subOb.sum();
  }
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24
```

The subclass **B** includes all of the members of its superclass, **A.** This is why **subOb** can access **i** and **j** and call **showij( )**. Also, inside **sum( )**, **i** and **j** can be referred to directly, as if they were part of **B.**

Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

The general form of a **class** declaration that inherits:

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

1.2 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**. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain
   private to their class.

   This program contains an error and will not
   compile.
*/

// Create a superclass.
class A {
  int i; // default access
  private int j; // private to A

  void setij(int x, int y) {
    i = x;
    j = y;
  }
}

// A's j is not accessible here.
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);
  }
}
```

1.3 A More Practical Example

Here, the final version of the **Box** class developed in the preceding chapter will be extended to include a fourth component called **weight**. Thus, the new class will contain a box's width, height, depth, and weight.

```
// 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;
  }
}

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);
  }
}
```

The output from this program is shown here:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

2. Using super

- In Java, the `super` keyword is used to refer to the immediate parent class object.
- **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.

2.1 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' constructor.

```
// A complete implementation of BoxWeight.

 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;
  }
 }

// BoxWeight now fully implements all constructors.
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();
      }
    }
```

This program generates the following 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
```

```
Note: // construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
  super(ob);
  weight = ob.weight;
}
```

2.2 A Second Use for super

The second form of **super,** 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.

This second 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 this simple class hierarchy:

```
// 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
```

3. Creating a Multilevel Hierarchy

Given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. In this case, **C** inherits all aspects of **B** and **A**. To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**. **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**, which holds the cost of shipping such a parcel.

```
// Extend BoxWeight to include shipping costs.

// Start with Box.
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;
  }
}

// Add weight.
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;
  }
}

// Add shipping costs.
class Shipment extends BoxWeight {
  double cost;

  // construct clone of an object
  Shipment(Shipment ob) { // pass object to constructor
    super(ob);
    cost = ob.cost;
  }

  // constructor when all parameters are specified
  Shipment(double w, double h, double d,
           double m, double c) {
    super(w, h, d, m); // call superclass constructor
    cost = c;
  }

  // default constructor
  Shipment() {
    super();
    cost = -1;
  }

  // constructor used when cube is created
  Shipment(double len, double m, double c) {
    super(len, m);
    cost = c;
  }
}

class DemoShipment {
  public static void main(String[] args) {
    Shipment shipment1 =
               new Shipment(10, 20, 15, 10, 3.41);
    Shipment shipment2 =
               new Shipment(2, 3, 4, 0.76, 1.28);

    double vol;

    vol = shipment1.volume();
    System.out.println("Volume of shipment1 is " + vol);
    System.out.println("Weight of shipment1 is "
                        + shipment1.weight);
    System.out.println("Shipping cost: $" + shipment1.cost);
    System.out.println();
    vol = shipment2.volume();
    System.out.println("Volume of shipment2 is " + vol);
    System.out.println("Weight of shipment2 is "
                        + shipment2.weight);
    System.out.println("Shipping cost: $" + shipment2.cost);
  }
}
```

The output of this program is shown here:

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28
```

4. When Constructors Are Executed

Given a subclass called **B** and a superclass called **A**, is **A**'s constructor executed before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.

Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used. If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed.

The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are executed.

// Create a super class.
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.");
  }
}

// Create another subclass by extending B.
class C extends B {
  C() {
    System.out.println("Inside C's constructor.");
  }
}

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

```
        }
```

The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

5. Method Overriding

Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

Conditions for Method Overriding:

Same method name

Same method type signature (parameters and return type)

Result of Method Overriding:

The method in the subclass is said to override the method in the superclass.

When the overridden method is called through the subclass, it always refers to the version defined in the subclass.

Key Point:

The version of the method defined by the superclass is hidden when called through the subclass.

Consider the following:

```
    // Method overriding.

    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
  }
}
```

The output produced by this program is shown here:

```
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**.
If you wish to access the superclass version of an overridden method, you can do so by using **super**.

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.
For example, consider this modified version of the preceding example:

```
// Methods with differing type signatures are overloaded – not
// overridden.
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);
  }
}

// Create a subclass by extending class A.
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // overload show()
  void show(String msg) {
    System.out.println(msg + k);
  }
}
```

```
class Override {
  public static void main(String[] args) {
    B subOb = new B(1, 2, 3);

    subOb.show("This is k: "); // this calls show() in B
    subOb.show(); // this calls show() in A
  }
}
```

The output produced by this program is shown here:

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

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

6. Dynamic Method Dispatch

- Method overriding in Java forms the foundation for 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 crucial for achieving run-time polymorphism in Java.
- A superclass reference variable can refer to a subclass object.
- When an overridden method is called through a superclass reference, Java determines the version to execute based on the type of the object being referred to at run time.
- It is the type of the object (not the type of the reference variable) that determines which version of an overridden method will be executed.
- Different versions of an overridden method are called when different types of objects are referred to through a superclass reference variable.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
  void callme() {
    System.out.println("Inside A's callme method");
  }
}
```

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

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

class Dispatch {
  public static void main(String[] args) {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C

    A r; // obtain a reference of type A r =

    a; // r refers to an A object
    r.callme(); // calls A's version of callme

    r = b; // r refers to a B object r.callme(); //
    calls B's version of callme

    r = c; // r refers to a C object r.callme(); //
    calls C's version of callme
              }
            }
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

6.1 Applying Method Overriding

- The following program creates a superclass called **Figure** that stores the dimensions of a two-dimensional object. It also defines a method called **area( )** that computes the area of an object.
- The program derives two subclasses from **Figure**.
- The first is **Rectangle** and the second is **Triangle**.
- Each of these subclasses overrides **area( )** so that it returns the area of a rectangle and a triangle, respectively.

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

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

7. Using Abstract Classes

- Java abstract class is a class that can not be initiated by itself, it needs to be subclassed by another class to use its properties.
- An abstract class is declared using the “abstract” keyword in its class definition.
- The main purpose of an abstract class is to provide a common structure for its subclasses, ensuring that they implement certain essential methods.
- An abstract class can also have constructors, data members, and static methods
- To declare an abstract method, use this general form:
  abstract *type name*(*parameter-list*);

Using an abstract class, you can improve the **Figure** class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares **area( )** as abstract inside **Figure**. This, of course, means that all classes derived from **Figure** must override **area( )**.

```
// Using abstract methods and classes.
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);
  }
  // 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 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());
  }
}
```

As the comment inside **main( )** indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract. And, all subclasses of **Figure** must override **area( )**. To prove this to yourself, try creating a subclass that does not override **area( )**. You will receive a compile-time error.

Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**.

The variable **figref** is declared as a reference to **Figure**, which means that it can be used to refer to an object of any class derived from **Figure**. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

8. Using final with Inheritance

The keyword **final** has two uses. This use was described in the preceding chapter. The uses of **final** apply to inheritance. Both are examined here.

8.1 Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. 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.
- Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it "knows" they will not be overridden by a subclass.
- When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call.

8.2 Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, 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
  //...
}
```

9. Local Variable Type Inference and Inheritance

- Local variable type inference to the Java language, which is supported by the context-sensitive keyword **var**. It is important to have a clear understanding of how type inference works within an inheritance hierarchy. Recall that a superclass reference can refer to a derived class object, and this feature is part of Java's support for polymorphism.
- However, it is critical to remember that, when using local variable type inference, the inferred type of a variable is based on the declared type of its initializer.
- Therefore, if the initializer is of the superclass type, that will be the inferred type of the variable.
- It does not matter if the actual object being referred to by the initializer is an instance of a derived class. For example, consider this program:

```
// When working with inheritance, the inferred type is the declared
// type of the initializer, which may not be the most derived type of
// the object being referred to by the initializer.

class MyClass {
  // ...
}

class FirstDerivedClass extends MyClass {
  int x;
  // ...
}
```

```
class SecondDerivedClass extends FirstDerivedClass {
int y;
  // ...
}

class TypeInferenceAndInheritance {

  // Return some type of MyClass object.
  static MyClass getObj(int which) {
    switch(which) {
      case 0: return new MyClass();
      case 1: return new FirstDerivedClass();
      default: return new SecondDerivedClass();
   }
 }

  public static void main(String[] args) {

    // Even though getObj() returns different types of
    // objects within the MyClass inheritance hierarchy,
    // its declared return type is MyClass. As a result,
    // in all three cases shown here, the type of the
    // variables is inferred to be MyClass, even though
    // different derived types of objects are obtained.

    // Here, getObj() returns a MyClass object.
     var mc = getObj(0);

    // In this case, a FirstDerivedClass object is returned.
     var mc2 = getObj(1);

    // Here, a SecondDerivedClass object is returned.
    var mc3 = getObj(2);

    // Because the types of both mc2 and mc3 are inferred
    // as MyClass (because the return type of getObj() is
    // MyClass), neither mc2 nor mc3 can access the fields
    // declared by FirstDerivedClass or SecondDerivedClass.
//    mc2.x = 10; // Wrong! MyClass does not have an x field.
//    mc3.y = 10; // Wrong! MyClass does not have a y field.
  }
}
```

10. The Object Class

- 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. (Deprecated by JDK 9.)
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**.
- **equals( )** and **toString( )**. The **equals( )** method compares two objects.
- It returns **true** if the objects are equal, and **false** otherwise.
- The precise definition of equality can vary, depending on the type of objects being compared.
- 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.

Chapter -2
Interfaces

- Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body.
- In practice, this means that you can define interfaces that don't make assumptions about how they are implemented.
- 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 the complete set of methods required by the interface.
- However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.
- Interfaces are designed to support dynamic method resolution at run time

2.1 Defining an Interface

An interface is defined much like a class. This is a simplified 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;
}
```

Here is an example of an interface definition. It declares a simple interface that contains one method called **callback( )** that takes a single integer parameter.

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

10.1.1 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 required by the interface.
- The general form of a class that includes the **implements** clause looks like this:

class *classname* [extends *superclass*] [implements *interface* [,*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.

Here is a small example class that implements the **Callback** interface shown earlier:

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

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

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **Client** implements **callback( )** and adds the method **nonIfaceMeth( )**:

```
class Client implements Callback {
  // Implement Callback's interface
  public void callback(int p) {
    System.out.println("callback called with " + p);
  }

  void nonIfaceMeth() {
    System.out.println("Classes that implement interfaces " +
                       "may also define other members, too.");
  }
}
```

2.1.2 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**.
- 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.
- Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Here is an example that demonstrates a nested interface:

```
// 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");
       }
     }
```

2.1.3 Applying Interfaces

- The stack can also be held in an array, a linked list, a binary tree, and so on.
- No matter how the stack is implemented, the interface to the stack remains the same.
- That is, the methods **push( )** and **pop( )** define the interface to the stack independently of the details of the implementation.

- Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples.
- First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**.
- This interface will be used by both stack implementations.

```
// Define an integer stack interface.
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item
}
```

The following program creates a class called **FixedStack** that implements a fixed-length version of an integer stack:

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
  private int[] stck;
  private int tos;

  // allocate and initialize stack
  FixedStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class IFTest {
  public static void main(String[] args) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);
    // push some numbers onto the stack
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<5; i++)
       System.out.println(mystack1.pop());
      System.out.println("Stack in mystack2:");
      for(int i=0; i<8; i++)
         System.out.println(mystack2.pop());
    }
  }
```

Following is another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition.
In this implementation, each stack is constructed with an initial length.
If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```
// Implement a "growable" stack.
class DynStack implements IntStack {
  private int[] stck;
  private int tos;

  // allocate and initialize stack
  DynStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  public void push(int item) {
    // if stack is full, allocate a larger stack
    if(tos==stck.length-1) {
      int[] temp = new int[stck.length * 2]; // double size
      for(int i=0; i<stck.length; i++) temp[i] = stck[i];
      stck = temp;
      stck[++tos] = item;
    }
    else
      stck[++tos] = item;
  }


  // Pop an item from the stack
  public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class IFTest2 {
  public static void main(String[] args) {
    DynStack mystack1 = new DynStack(5);
    DynStack mystack2 = new DynStack(8);

    // these loops cause each stack to grow
```

```
        for(int i=0; i<12; i++) mystack1.push(i);
        for(int i=0; i<20; i++) mystack2.push(i);

        System.out.println("Stack in mystack1:");
        for(int i=0; i<12; i++)
           System.out.println(mystack1.pop());

        System.out.println("Stack in
        mystack2:"); for(int i=0; i<20; i++)
           System.out.println(mystack2.pop());
     }
   }
```

2.2 Default Interface Methods

- A default method lets you define a default implementation for an interface method. In other words, by use of a default method, it is possible for an interface method to provide a body, rather than being abstract. During its development, the default method was also referred to as an *extension method*
- A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. Recall that there must be implementations for all methods defined by an interface.
- The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided. Thus, the addition of a default method will not cause preexisting code to break.
- Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used.
- One last point: As a general rule, default methods constitute a special-purpose feature. Interfaces that you create will still be used primarily to specify *what* and not *how*. However, the inclusion of the default method gives you added flexibility.

2.2.1 Default Method Fundamentals

An interface default method is defined similar to the way a method is defined by a **class.** The primary difference is that the declaration is preceded by the keyword **default.**

For example, consider this simple interface:

```
public interface MyIF {
  // This is a "normal" interface method declaration.
  // It does NOT define a default implementation.
  int getNumber();

  // This is a default method. Notice that it provides
  // a default implementation.
  default String getString() {
    return "Default String";
  }
}
```

MyIF declares two methods. The first, **getNumber( )**, is a standard interface method declaration. It defines no implementation whatsoever. The second method is **getString( )**, and it does include a default implementation. In this case, it simply returns the string "Default String". Pay special attention to the way **getString( )** is declared. Its declaration is preceded by the **default** modifier. This syntax can be generalized. To define a default method, precede its declaration with **default**.

Because **getString( )** includes a default implementation, it is not necessary for an implementing class to override it. In other words, if an implementing class does not provide its own implementation, the default is used. For example, the **MyIFImp** class shown next is perfectly valid:

```
// Implement MyIF.
class MyIFImp implements MyIF {
  // Only getNumber() defined by MyIF needs to be implemented.
  // getString() can be allowed to default.
  public int getNumber() {
    return 100;
  }
}
```

The following code creates an instance of **MyIFImp** and uses it to call both **getNumber( )** and **getString( )**.

```
// Use the default method.
class DefaultMethodDemo {
  public static void main(String[] args) {

    MyIFImp obj = new MyIFImp();

    // Can call getNumber(), because it is explicitly
    // implemented by MyIFImp:
    System.out.println(obj.getNumber());

    // Can also call getString(), because of default
    // implementation:
    System.out.println(obj.getString());
  }
}
```

The output is shown here:

```
100
Default String
```

A More Practical Example

- **IntStack** is widely used and many programs rely on it. Further assume that we now want to add a method to **IntStack** that clears the stack, enabling the stack to be re-used.
- Thus, we want to evolve the **IntStack** interface so that it defines new functionality, but we don't want to break any preexisting code. In the past, this would be impossible, but with the inclusion of default methods, it is now easy to do.
- For example, the **IntStack** interface can be enhanced like this:

```
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item

  // Because clear( ) has a default, it need not be
  // implemented by a preexisting class that uses IntStack.
  default void clear() {
    System.out.println("clear() not implemented.");
  }
}
```

Here, the default behavior of **clear( )** simply displays a message indicating that it is not implemented. This is acceptable because no preexisting class that implements **IntStack** would ever call **clear( )** because it was not defined by the earlier version of **IntStack**.
However, **clear( )** can be implemented by a new class that implements **IntStack**. Furthermore, **clear( )** needs to be defined by a new implementation only if it is used. Thus, the default method gives you

- a way to gracefully evolve interfaces over time, and
- a way to provide optional functionality without requiring that a class provide a placeholder implementation when that functionality is not needed.

2.3 Use static Methods in an Interface

Another capability added to **interface** by JDK 8 is the ability to define one or more **static** methods. Like **static** methods in a class, a **static** method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a **static** method. Instead, a **static** method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form:

InterfaceName.staticMethodName

Notice that this is similar to the way that a **static** method in a class is called.

The following shows an example of a **static** method in an interface by adding one to **MyIF**, shown in the previous section. The **static** method is **getDefaultNumber( )**. It returns zero.

```
public interface MyIF {
  // This is a "normal" interface method declaration.
  // It does NOT define a default implementation.
  int getNumber();

   // This is a default method. Notice that it provides
   // a default implementation.
   default String getString() {
     return "Default String";
   }

  // This is a static interface method.
  static int getDefaultNumber() {
    return 0;
  }
}
```

The **getDefaultNumber( )** method can be called, as shown here:

```
int defNum = MyIF.getDefaultNumber();
```

As mentioned, no implementation or instance of **MyIF** is required to call **getDefaultNumber( )** because it is **static**.

One last point: **static** interface methods are not inherited by either an implementing class or a subinterface.

2.4 Private Interface Methods

- The key benefit of a private interface method is that it lets two or more default methods use a common piece of code, thus avoiding code duplication.
- For example, here is another version of the **IntStack** interface that has two default methods called **popNElements( )** and **skipAndPopNElements( )**.
- The first returns an array that contains the top *N* elements on the stack.
- The second skips a specified number of elements and then returns an array that contains the next *N* elements. Both use a private method called **getElements( )** to obtain an array of the specified number of elements from the stack.

```
// Another version of IntStack that has a private interface
// method that is used by two default methods.
interface IntStack {
  void push(int item); // store an item
  int pop(); // retrieve an item



  // A default method that returns an array that contains
  // the top n elements on the stack.
  default int[] popNElements(int n) {
    // Return the requested elements.
    return getElements(n);
  }

  // A default method that returns an array that contains
  // the next n elements on the stack after skipping elements.
  default int[] skipAndPopNElements(int skip, int n) {

    // Skip the specified number of elements.
    getElements(skip);

    // Return the requested elements.
    return getElements(n);
  }

  // A private method that returns an array containing
  // the top n elements on the stack
  private int[] getElements(int n) {
    int[] elements = new int[n];

    for(int i=0; i < n; i++) elements[i] = pop();
    return elements;
  }
}
```

Notice that both **popNElements( )** and **skipAndPopNElements( )** use the private **getElements( )** method to obtain the array to return. This prevents both methods from having to duplicate the same code sequence. Keep in mind that because **getElements( )** is private, it cannot be called by code outside **IntStack**. Thus, its use is limited to the default methods inside **IntStack**. Also, because **getElements( )** uses the **pop( )** method to obtain stack elements, it will automatically call the implementation of **pop( )** provided by the **IntStack** implementation. Thus, **getElements( )** will work for any stack class that implements **IntStack**.