# UNIT III

**Inheritance** – Hierarchical abstractions, Base class object, subclass, subtype, substitutability, forms of inheritance- specialization, specification, construction, extension, limitation, combination, benefits of inheritance, costs of inheritance. Member access rules, super uses, using final with inheritance, polymorphism- method overriding, abstract classes.

## INTRODUCTION

- ✓ Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications.
- ✓ Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
- ✓ In the terminology of Java, a class that is inherited is called a *superclass.*
- ✓ The class that does the inheriting is called a *subclass.*
- ✓ Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

## Inheritance Basics

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

```
// 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();

superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();


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
```

*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.*

## Subclass, Subtype, Substitutability

In programming languages, inheritance means:

- Data and behavior of parent class are part of child
- Child class may include data and behavior not in parent

**Subclass & Subtype**

- To assert that one class is a subclass of another is to simply say it was built using inheritance.
- To assert that one class is a subtype of another is to say that it preserves the purpose of the original.

The two concepts are independent:

- Not all subclasses are subtypes
- Sometimes subtypes are constructed without being subclasses

**Substitutability:**

In order to satisfy substitutability it has to satisfy following properties:

- Instances of the subclass must posses all data areas associated with the parent class.
- Instances of the subclass must implement, through inheritance at least (if not explicitly overridden) all functionality defined for the parent class.
- Thus, an instance of the a child class can mimic the behavior of the parent class.

Note: Subtypes must satisfy the substitution principle, which means that any code that relies on B’s specification will function properly if A’s implementation is substituted for B.

## FORMS OF INHERITANCE

- ✓ Inheritance can be used in a variety of different ways and for different purposes
- ✓ Many of these types of inheritance are given their own special names
- ✓ We will describe some of these specialized forms of inheritance
    - Specialization
    - Specification
    - Construction
    - Generalization or Extension
    - Limitation
    - Combination

**Specialization Inheritance**

- ✓ By far the most common form of inheritance is specialization
- ✓ Each child class overrides a method inherited from the parent in order to specialize the class in some way

Example:

```
class First
{
void show()
{
System.out.println("Hello");
}
void showA()
{
System.out.println("Welcome");
}
}

class Second extends First
{
void show()
{
System.out.println("Hi");
}
void showA()
{
System.out.println("Hello");
}
}
```

**Specification Inheritance**

If the parent class is abstract, we often say that it is providing a specification for the child class.

```
abstract class First
{
abstarct void show();
abstract void showA();
}
```

```
class Second extends First
{
void show()
{
System.out.println(“Hi”);
}
void showA()
{
System.out.println(“Hello”);
}
}
```

**Inheritance for Construction**

- ✓ If the parent class is used as a source for behavior, but the child class has no *is-a* relationship to the parent, then we say the child class is using inheritance for construction
- ✓ Generally not a good idea, since it can break the principle of substitutability, but nevertheless sometimes found in practice (More often in dynamically typed languages, such as Smalltalk)

```
class First
{
void show()
{
System.out.println(“Hello”);
}
void showA()
{
System.out.println(“Welcome”);
}
}

class Second extends First
{
void set(int a, int b)
{
.........
.........
}

void add()
{
..........
........... }
```

**Inheritance for Generalization or Extension**

- ✓ If a child class generalizes or extends the parent class by providing more functionality, but does not override any method, we call it inheritance for generalization
- ✓ The child class doesn't change anything inherited from the parent, it simply adds new features

```
class First

{
void set(int a, int b)
{
.........
.........
}

void add()
{
..........
...........
 }

class Second extends First
{
void show()
{
..............
}
}
```

**Inheritance for Limitation**

- ✓ If a child class overrides a method inherited from the parent in a way that makes it unusable (for example, issues an error message), then we call it inheritance for limitation
- ✓ Generally not a good idea, since it breaks the idea of substitution. But again, it is sometimes found in practice

```
class First

{
void set(int a, int b)
{// initialization}
```

```
void add()
{
..........
...........
 }

class Second extends First
{
void set()
{
//display
}
}
```

**Inheritance for Combination**

The child class inherits features from more than one parent class. Although multiple inheritance is not supported directly by Java, it can be achieved using interfaces.

```
interface A
{
……
}

interface B
{
…….
}

class C implements A, B
{
……..
}
```

**Summary of Forms of Inheritance**

- *Specialization*
  - The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.
- *Specification*
  - The parent class defines behaviour that is implemented in the child class but not in the parent class.

- *Construction*
  - The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class.
- *Generalization*
  - The child class modifies or overrides some of the methods of the parent class.
- *Extension*
  - The child class adds new functionality to the parent class, but does not change any inherited behavior.
- *Limitation*
  - The child class restricts the use of some of the behavior inherited from the parent class.
- *Combination*
  - The child class inherits features from more than one parent class. This is multiple inheritance.

## THE BENEFITS OF INHERITANCE

- **Software Reusability**

When behavior is inherited from another class, the code that provides that behavior does not have to be rewritten. With objet oriented techniques, the functions can be written once and reused.

- **Increased Reliability**

Code that is executed frequently will tend to have fewer bugs then code that executed infrequently. When same components are used in two or more applications, the code will be exercised more than code that is developed for a single application. Thus bugs in such code tend to be more quickly discovered and latter applications gain the benefit of using components are more error free. Similarly the costs of maintenance of shared components can be split among many projects.

- **Code Sharing**

Code sharing can occur on several levels with OO techniques. Two or more objects will share the code that they inherit.

- **Consistency of Interface**

When two or more classes inherit from the same super class, the behavior inherit will be same in all cases. Thus it easier to guarantee that interfaces to similar objects are in fact similar, the user is not presented with a confusing collection of objects that are almost the same but behave, and are interacted with, very differently.

- **Software Components**

Inheritance provide programmers with the ability to construct reusable software components. The goal is to permit the development of new and novel applications that nevertheless require little or no actual coding.

- **Rapid Prototyping**

When a s/w system is constructed largely out of reusable components, development time can be concentrated on understanding new and unusual portion of the system. Thus, s/w systems can be generated more quickly and easily, leading to a style of programming known as rapid prototyping or exploratory programming.

- **Polymorphism**

Polymorphism in programming languages permits the programmer to generate high-level reusable components that can be tailored to fit different applications by changes in their low-level parts.

- **Information hiding**

A programmer who reuses a s/w components needs only to understand the nature of the component and its interface. It is not necessary for the programmer to have detailed information concerning matters such as the techniques used to implement the component.

## THE COST OF INHERITANCE

Although the benefits of inheritance in OOP are great, almost nothing is without cost of one sort or another.

- **Execution Speed**

The inherited methods, which must deal with arbitrary subclasses, are often slower than specialized code.

- **Program Size**

The use of any s/w library frequently imposes a size penalty not imposed by systems constructed for a specific project. As memory cost decrease the size of programs becomes less important.

- **Message-Passing Overhead**

A message passing is by nature a more costly operation than simple procedure invocation.

- **Program Complexity**

The overuse of inheritance can often simply replace one form of complexity with another. Understanding the control flow of a program that uses inheritance may require several multiple scans up and down the inheritance graph. This is known as the yo-yo problem.

## Using super

- ✓ However, there will be times when you will want to create a super class 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 super class, it can do so by use of the keyword **super**.

**super** has two general forms.

- ✓ The first calls the super class' constructor.
- ✓ The second is used to access a member of the super class that has been hidden by a member of a subclass.

### Using super to Call Superclass Constructors

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

```
super(parameter-list);
```

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 Box
{
private double width;
private double height;
private double depth;
```

```
Box(Box ob) {
width = ob.width;
height = ob.height;
depth = ob.depth;
}

Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}

Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}

Box(double len) {
width = height = depth = len;
}

double volume() {
return width * height * depth;
}
}

class BoxWeight extends Box
{
double weight; // weight of box
BoxWeight(BoxWeight ob) {
super(ob);
weight = ob.weight;
}

BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
BoxWeight() {
super();
weight = -1;
}

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();
}
```

### A Second Use for super

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used.

This usage has the following general form:

super.*member*

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

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.

```
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();
}
}
```

This program displays the following:

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

## DYNAMIC METHOD DISPATCH

- ✓ 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.
- ✓ Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
- ✓ An important principle: 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 a n 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.

```
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
```

## Why Overridden Methods?

- ✓ Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
- ✓ Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.
- ✓ Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization.
- ✓ Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its own methods, yet still enforces a consistent interface. Thus, 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.
- ✓ Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

## Applying Method Overriding

```
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);

System.out.println("Area is " + f.area());
System.out.println("Area is " + r.area());
System.out.println("Area is " + t.area());
}
}
```

## ABSTRACT CLASSES

- ✓ There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- ✓ That is, sometimes create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement.
- ✓ One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java’s solution to this problem is the *abstract method.*
- ✓ Certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass.

To declare an abstract method, use this general form:

abstract *type name(parameter-list)*;

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

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

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

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


class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

## USING final WITH INHERITANCE

The keyword **final** has three uses.

- First, it can be used to create the equivalent of a named constant.

```
class A
{
final int n=10;
}
```

```
class B extends A
{
void show()
{
System.out.println(n++); //ERROR! Can’t be modified
}
}
```

## Using final to Prevent Overriding

- To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

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

## Using final 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.

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

## 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. |
| 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( )<br>void wait(long *milliseconds*)<br>void wait(long *milliseconds*, int *nanoseconds*) | Waits on another thread of execution. |