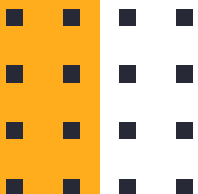






INHERITANCE

Inheritance

- Allows the creation of hierarchical classifications
- In the terminology 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 instance variables and methods defined by the superclass and adds its own, unique elements
 - 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();
    // 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

- A subclass can be a superclass for another subclass
- The general form of a **class** declaration that inherits a superclass is shown here:

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



Java does not support the inheritance of multiple superclasses into a single subclass



Member Access and Inheritance



- Subclass cannot access those members of the superclass that have been declared as **private**

// Create a superclass.

class A

{

int i; // public by default

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();
```

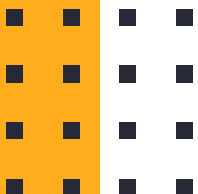
```
        System.out.println("Total is " + subOb.total);
```

```
    }
```

```
}
```

Using super

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**
- **super** has two general forms
- The first calls the superclass' constructor
- The second is used to access a member of the superclass that has been hidden by a member of a subclass



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

super(parameter-list);

- *parameter-list* specifies any parameters needed by the constructor in the superclass

```
class Box
{
    double width;
    double height;
    double depth;
    // This is the constructor for Box.
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
}
```



// BoxWeight now uses super to initialize its Box attributes.


```
class BoxWeight extends Box
{
    double weight; // weight of box
    // initialize width, height, and depth using super()
    BoxWeight(double w, double h, double d, double m)
    {
        super(w, h, d); // call superclass constructor
        weight = m;
    }
}
```

A Second Use for 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

// 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();
    }
}
```



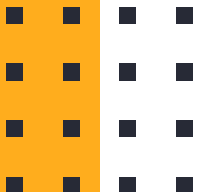
This program displays the following:

i in superclass: 1

i in subclass: 2

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**
- **C** inherits all aspects of **B** and **A**



When Constructors Are Called ?


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

// 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.");
    }
}
```



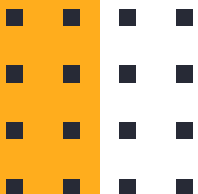
```
class CallingCons
{
    public static void main(String args[])
    {
        C c = new C();
    }
}
```





The output from this program is shown here:
Inside A's constructor
Inside B's constructor
Inside C's constructor

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass



- 
- 
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass
 - The version of the method defined by the superclass will be hidden

```
class A
{
    int i, j;
    A(int a, int b)
    {
        i = a;
        j = b;
    }
    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



If you wish to access the superclass version of an overridden function, you can do so by using super



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

- 
- 
- 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***

// Methods with differing type signatures are overloaded - not overridden.

```
class A {
```

```
    int i, j;
```

```
    A(int a, int b) {
```

```
        i = a;
```

```
        j = b;
```

```
    }
```

```
    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



Method overriding is one of the ways in which Java supports Runtime Polymorphism. **Dynamic method dispatch** is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- When an overridden method is called through a superclass reference, Java determines which version (superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on 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
- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.



```
// A Java program to illustrate Dynamic Method
// Dispatch using hierarchical inheritance
class A
{
    void m1()
    {
        System.out.println("Inside A's m1 method");
    }
}
```




```
class B extends A
```

```
{
```

```
    // overriding m1()
```

```
    void m1()
```

```
    {
```

```
        System.out.println("Inside B's m1 method");
```

```
    }
```

```
}
```

```
class C extends A
```

```
{
```

```
    // overriding m1()
```

```
    void m1()
```

```
    {
```

```
        System.out.println("Inside C's m1 method");
```

```
    }
```


```
}
```



// Driver class

class Dispatch

```
{      public static void main(String args[])
      {
        // object of type A
        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 ref; // obtain a reference of type A
        ref = a; // now ref refers to a A object
        ref.m1(); // calling A's version of m1()
        ref= b; // now refers to B object
        ref.m1(); // calling B's version of m1()
        ref= c; // now refers to C object
        ref.m1(); // calling C's version of m1()
      }
}
```





Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

In C++, if a class has at least one pure virtual function, then the class becomes abstract. Unlike C++, in Java, a separate keyword *abstract* is used to make a class abstract.

Points to Remember

- 1) An abstract class must be declared with an abstract keyword.
- 2) It can have abstract and non-abstract methods.
- 3) It cannot be instantiated. we can have references of abstract class type though.
- 4) It can have constructors and static methods also.
- 5) It can have final methods which will force the subclass not to change the body of the method.

abstract method in java

A method declared without a body (no implementation) within an abstract class is an **abstract method**. In other words, if you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, then you can declare the method in the parent class as an abstract.

Key Features of Abstract Method

Listed below are key features of Abstract Method:

1. Abstract methods don't have an implementation (body), they just have method signature as shown in the above example
2. If a class has an abstract method it should be declared abstract, the vice versa is not true
3. Instead of curly braces, an abstract method will have a semicolon (;) at the end
4. If a regular class extends an abstract class, then the class must have to implement all the abstract methods of that class or it has to be declared abstract as well

Example on abstract class and abstract method

```
abstract class Shape    // here shape is abstract class
{
abstract void draw();  // abstract method draw
}
class Rectangle extends Shape
{
void draw()
{
System.out.println("drawing rectangle");
}
}
class Circle1 extends Shape
{
void draw()
{
System.out.println("drawing circle");
}
}
class TestAbstraction1
{
public static void main(String args[])
{
Shape s=new Circle1(); // you can create shape class reference variable but not
object of type shape
s.draw(); // here draw method refers to circle object.
}
}
```



“ Code Never Lies ”

Thank you!