

Java

Inheritance

Inheritance

- Same inheritance concept of C++ in Java with some modifications
 - One class inherits the other using ***extends*** keyword
 - The classes involved in inheritance are known as ***superclass*** and ***subclass***
 - ***Multilevel*** inheritance but no ***multiple*** inheritance
 - There is a special way to call the superclass's ***constructor***
 - There is automatic ***dynamic method dispatch***
- Inheritance provides code reusability (code of any class can be used by extending that class)

Simple Inheritance

```
3 class A {
4     int i, j;
5
6     void showij() {
7         System.out.println(i+" "+j);
8     }
9 }
10
11 class B extends A{
12     int k;
13
14     void showk() {
15         System.out.println(k);
16     }
17
18     void sum() {
19         System.out.println(i+j+k);
20     }
21 }
```

```
23 public class SimpleInheritance {
24     public static void main(String[] args) {
25         A superOb = new A();
26         superOb.i = 10;
27         superOb.j = 20;
28         superOb.showij();
29         B subOb = new B();
30         subOb.i = 7;
31         subOb.j = 8;
32         subOb.k = 9;
33         subOb.showij();
34         subOb.showk();
35         subOb.sum();
36     }
37 }
```

Inheritance and Member Access

```
1 class M {  
2     int i;  
3     private int j;  
4  
5     void set(int x, int y) {  
6         i = x;  
7         j = y;  
8     }  
9 }  
10  
11 class N extends M {  
12     int total;  
13  
14     void sum() {  
15         total = i + j;  
16         // Error, j is not accessible here  
17     }  
18 }  
19
```

```
20 public class SimpleInheritance2 {  
21     public static void main(String[] args) {  
22         N obj = new N();  
23         obj.set(10, 20);  
24         obj.sum();  
25         System.out.println(obj.total);  
26     }  
27 }
```

- 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

Practical Example

```
3  class Box {  
4      double width, height, depth;  
5  
6      Box(Box ob) {  
7          width = ob.width; height = ob.height; depth = ob.depth;  
8      }  
9  
10     Box(double w, double h, double d) {  
11         width = w; height = h; depth = d;  
12     }  
13  
14     Box() { width = height = depth = 1; }  
17  
18     Box(double len) { width = height = depth = len; }  
21  
22     double volume() { return width * height * depth; }  
25 }  
26  
27 class BoxWeight extends Box {  
28     double weight;  
29  
30     BoxWeight(double w, double h, double d, double m) {  
31         width = w; height = h; depth = d; weight = m;  
32     }  
33 }
```

Superclass variable reference to Subclass object

```
34
35 ▶ public class RealInheritance {
36 ▶     public static void main(String[] args) {
37         BoxWeight weightBox = new BoxWeight(w: 3, h: 5, d: 7, m: 8.37);
38         System.out.println(weightBox.weight);
39         Box plainBox = weightBox; // assign BoxWeight reference to Box reference
40         System.out.println(plainBox.volume()); // OK, volume() defined in Box
41         System.out.println(plainBox.weight); // Error, weight not defined in Box
42         Box box = new Box(w: 1, h: 2, d: 3); // OK
43         BoxWeight wbox = box; // Error, can't assign Box reference to BoxWeight
44     }
45 }
46
```

Using super to call Superclass Constructors

super() must always be the first statement executed inside a subclass' constructor

```
3 class BoxWeightNew extends Box {
4     double weight;
5
6     BoxWeightNew(BoxWeightNew ob) {
7         super(ob);
8         weight = ob.weight;
9     }
10
11    BoxWeightNew(double w, double h, double d, double m) {
12        super(w, h, d);
13        weight = m;
14    }
15
16    BoxWeightNew() {
17        super(); // must be the 1st statement in constructor
18        weight = 1;
19    }
20
21    BoxWeightNew(double len, double m) {
22        super(len);
23        weight = m;
24    }
25
26    void print() {
27        System.out.println("Box(" + width + ", " + height +
28            ", " + depth + ", " + weight + ")");
29    }
30 }
```

Using super to call Superclass Constructors

```
31
32 public class SuperTest {
33     public static void main(String[] args) {
34         BoxWeightNew box1 = new BoxWeightNew(10, 20, 15, 34.3);
35         BoxWeightNew box2 = new BoxWeightNew(2, 3, 4, 0.076);
36         BoxWeightNew box3 = new BoxWeightNew();
37         BoxWeightNew cube = new BoxWeightNew(3, 2);
38         BoxWeightNew clone = new BoxWeightNew(box1);
39         box1.print();
40         box2.print();
41         box3.print();
42         cube.print();
43         clone.print();
44     }
45 }
46
47
```


Using super to access Superclass hidden members

```
3 class C {
4     int i;
5     void show() {
6     }
7 }
8
9 class D extends C {
10     int i; // this i hides the i in C
11
12     D(int a, int b) {
13         super.i = a; // i in C
14         i = b; // i in D
15     }
16
17     void show() {
18         System.out.println("i in superclass: " + super.i);
19         System.out.println("i in subclass: " + i);
20         super.show();
21     }
22 }
23
24 public class UseSuper {
25     public static void main(String[] args) {
26         D subOb = new D(1, 2);
27         subOb.show();
28     }
29 }
```

Multilevel Inheritance

```
3 class X {
4     int a;
5     X() {
6         System.out.println("Inside X's constructor");
7     }
8 }
9
10 class Y extends X {
11     int b;
12     Y() {
13         System.out.println("Inside Y's constructor");
14     }
15 }
16
17 class Z extends Y {
18     int c;
19     Z() {
20         System.out.println("Inside Z's constructor");
21     }
22 }
23
24 public class MultilevelInheritance {
25     public static void main(String[] args) {
26         Z z = new Z();
27         z.a = 10;
28         z.b = 20;
29         z.c = 30;
30     }
31 }
```

Inside X's constructor
Inside Y's constructor
Inside Z's constructor

Method Overriding

```
3 class Base {
4     int a;
5     Base(int a) {
6         this.a = a;
7     }
8     void show() {
9         System.out.println(a);
10    }
11 }
12
13 class Child extends Base {
14     int b;
15
16     Child(int a, int b) {
17         super(a);
18         this.b = b;
19     }
20
21     // the following method overrides Base class's show()
22     @Override // this is an annotation (optional but recommended)
23     void show() {
24         System.out.println(a + ", " + b);
25     }
26 }
27
28 public class MethodOverride {
29     public static void main(String[] args) {
30         Child o = new Child(a: 10, b: 20);
31         o.show();
32         Base b = o;
33         b.show(); // will call show of Override
34     }
35 }
```

Dynamic Method Dispatch

```
3 class P {
4     void call() {
5         System.out.println("Inside P's call method");
6     }
7 }
8 class Q extends P {
9     void call() {
10        System.out.println("Inside Q's call method");
11    }
12 }
13 class R extends Q {
14     void call() {
15        System.out.println("Inside R's call method");
16    }
17 }
18
19 public class DynamicDispatchTest {
20     public static void main(String[] args) {
21         P p = new P(); // object of type P
22         Q q = new Q(); // object of type Q
23         R r = new R(); // object of type R
24         P x;           // reference of type P
25         x = p;          // x refers to a P object
26         x.call();       // invoke P's call
27         x = q;          // x refers to a Q object
28         x.call();       // invoke Q's call
29         x = r;          // x refers to a R object
30         x.call();       // invoke R's call
31     }
32 }
```

For practical example please
refer to **FindAreas.java**

Abstract Class

- ***abstract class A***
- contains abstract method ***abstract method f()***
- No instance can be created of an abstract class
- The subclass must implement the abstract method
- Otherwise the subclass will be a abstract class too

Abstract Class

```
3  abstract class S {
4      // abstract method
5      abstract void call();
6      // concrete methods are still allowed in abstract classes
7      void call2() {
8          System.out.println("This is a concrete method");
9      }
10 }
11
12 class T extends S {
13     void call() {
14         System.out.println("T's implementation of call");
15     }
16 }
17
18 class AbstractDemo {
19     public static void main(String args[]) {
20         //S s = new S(); // S is abstract; cannot be instantiated
21         T t = new T();
22         t.call();
23         t.call2();
24     }
25 }
```

For practical example please
refer to **FindAreas2.java**

Anonymous Subclass

```
3  abstract class S {  
4      // abstract method  
5      abstract void call();  
6      // concrete methods are still allowed in abstract classes  
7      void call2() {  
8          System.out.println("This is a concrete method");  
9      }  
10 }  
11  
12 class AbstractDemo {  
13     public static void main(String args[]) {  
14         //S s = new S(); // S is abstract; cannot be instantiated  
15         S s = new S() {  
16             void call() {  
17                 System.out.println("Call method of an abstract class");  
18             }  
19         };  
20         s.call();  
21     }  
22 }
```

Using final with Inheritance

To prevent overriding

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

To prevent inheritance

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


Local Variable Type Inference and Inheritance

- A superclass reference can refer to a derived class object in Java
- 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

Local Variable Type Inference and Inheritance

```
1 class A {  
2     int a;  
3 }  
4 class B extends A {  
5     int b;  
6 }  
7 class C extends B {  
8     int c;  
9 }  
10 public class InheritanceVarDemo {  
11     @static A getObject(int type) {  
12         switch(type) {  
13             case 0: return new A();  
14             case 1: return new B();  
15             case 2: return new C();  
16             default: return null;  
17         }  
18     }  
19 }
```

```
19 public static void main(String[] args) {  
20     var x = getObject( type: 0);  
21     var y = getObject( type: 1);  
22     var z = getObject( type: 2);  
23     System.out.println(x.a);  
24     System.out.println(y.b);  
25     // Error, A doesn't have b field  
26     System.out.println(z.c);  
27     // Error, A doesn't have c field  
28 }  
29 }
```

For detail example please refer to
InheritanceVarDemo.java

The inferred type is determined by the return type of `getObject()`, not by the actual type of the object obtained. Thus, all three variables will be of type A

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's toString()

- 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 provide a description specifically for the types of objects that they create

Object's toString()

```
3  class Point {
4      int x, y;
5
6      Point(int x, int y) {
7          this.x = x;
8          this.y = y;
9      }
10
11     @Override
12     public String toString() {
13         return "(" + x + ", " + y + ")";
14     }
15 }
16
17 public class ObjectTest {
18     public static void main(String[] args) {
19         Point p1 = new Point( x: 10, y: 20);
20         // without override toString() method the
21         // following will print something like this
22         // Point@3cd1a2f1
23         System.out.println(p1);
24     }
25 }
```

Object's equals() and hashCode()

- `==` is a reference comparison, whether both variables refer to the same object
- Object's **equals()** method does the same thing
- String class override **equals()** to check contents
- If you want two different objects of a same class to be equal then you need to override **equals()** and **hashCode()** methods
 - **hashCode()** needs to return same value to work properly as keys in Hash data structures

Object's equals() and hashCode()

```
3 import java.util.HashMap;
4 import java.util.Objects;
5
6 class Point {
7     int x, y;
8     Point(int x, int y) {
9         this.x = x;
10        this.y = y;
11    }
12
13    @Override
14    public boolean equals(Object o) {
15        if (o == this) return true;
16        if (!(o instanceof Point)) {
17            return false;
18        }
19        Point p = (Point) o;
20        if (p.x == this.x && p.y == this.y) return true;
21        return false;
22    }
23
24    @Override
25    public int hashCode() {
26        return Objects.hash(x, y);
27    }
28 }
```

```
30 public class ObjectTest {
31     public static void main(String[] args) {
32         Point p1 = new Point(x: 10, y: 20);
33         Point p2 = new Point(x: 10, y: 20);
34         System.out.println(p1.equals(p2));
35         System.out.println(p1 == p2);
36         HashMap m = new HashMap();
37         m.put(p1, "Hello");
38         System.out.println(m.get(p2));
39     }
40 }
41
```