

Subclassing in Java

When a new class is defined in terms of an existing class, the new class *extends* the existing class. The new class *inherits* all public and protected members of its *parent* (or *base*) class. The new class may add new methods or fields. It may also redefine inherited methods or fields.

```
class Point {
    int x,y;
    Point(int xin, int yin) {
        x = xin; y = yin;
    }
    static float dist(
        Point P1, Point P2) {
        return (float) Math.sqrt(
            (P1.x-P2.x)*(P1.x-P2.x)+
            (P1.y-P2.y)*(P1.y-P2.y));
    }
}
```

```
class Point3 extends Point {
    int z;
    Point3(int xin, int yin,
        int zin) {
        super(xin,yin); z=zin;
    }
    static float dist(
        Point3 P1, Point3 P2) {
        float d=Point.dist(P1,P2);
        return (float) Math.sqrt(
            (P1.z-P2.z)*(P1.z-P2.z)+
            d*d);
    }
}
```

Note that although `Point3` redefines `dist`, the old definition of `dist` is still available by using the parent class as a qualifier (`Point.dist`).

The same is true for fields that are hidden when a field in a parent is redeclared.

Non-static methods are automatically *virtual*: a redefined method is automatically used in all inherited methods including those defined in parent classes that think they are using an earlier definition of the class.

Example:

```
class C {
    void DoIt(){PrintIt();}
    void PrintIt()
        {println("C rules!");}
}
class D extends C {
    void PrintIt()
        {println("D rules!");}
    void TestIt() {DoIt();}
}
D dvar = new D();
dvar.TestIt();
D rules! is printed.
```

Static methods in Java are *not* virtual (this can make them easier to implement efficiently).

Abstract Classes and Methods

Sometimes a Java class is not meant to be used by itself because it is intentionally incomplete.

Rather, the class is meant to be starting point for the creation (via subclassing) of more complete classes.

Such classes are *abstract*.

Example:

```
abstract class Shape {
    Point location;
}
class Circle extends Shape {
    float radius;
}
```

Methods can also be made abstract to indicate that their actual definition will appear in subclasses:

```
abstract class Shape {
    Point location;
    abstract float area();
}
class Circle extends Shape {
    float radius;
    float area(){
        return Math.pi*radius*radius;
    }
}
```

Subtyping and Inheritance

We can use a subtyping mechanism, as found in C++ or Java, for two different purposes:

- We may wish to inherit the actual implementations of classes and members to use as the basis of a more complete or extended class. To inherit an implementation, we say a given class “extends” an existing class:

```
class Derived extends Base
{ ... };
```

Class **Derived** contains all of the members of **Base** *plus* any others it cares to add.

- We may wish to inherit an *interface*—a set of method names and values that will be available for use. To inherit (or claim) an interface, we use a Java interface definition. An interface doesn’t implement anything; rather, it gives a name to a set of operations or values that may be available within one or more classes.

Why are Interfaces Important?

Many classes, although very different, share a common subset of values or operations. We may be willing to use any such class as long as only interface values or operations are used.

For example, many objects can be ordered (or at least partially-ordered) using a “less than” operation.

If we always implement less than the same way, for example,

```
boolean lessThan(Object o1,  
                  Object o2);
```

then we can create an interface that admits all classes that know about the `lessThan` function:

```
interface Compare {  
    boolean lessThan(Object o1,  
                     Object o2);  
}
```

Now different classes can each implement the `Compare` interface, proclaiming to the world that they know how to compare objects of the class they define:

```
class IntCompare implements Compare {  
    public boolean lessThan(Object i1,  
                           Object i2){  
        return ((Integer)i1).intValue() <  
               ((Integer)i2).intValue();  
    }  
class StringCompare implements  
    Compare {  
    public boolean lessThan(Object i1,  
                           Object i2){  
        return  
        ((String)i1).compareTo((String)i2)<0;  
    }  
}
```

The advantage of using interfaces is that we can now define a method or class that only depends on the given interface, and which will accept *any* type that implements that interface.

```
class PrintCompare {  
    public static void printAns(  
        Object v1, Object v2, Compare c){  
        System.out.println(  
            v1.toString() + " < " +  
            v2.toString() + " is " +  
            new Boolean(c.lessThan(v1,v2))  
                .toString());  
    }  
}  
class Test {  
    public static void  
    main(String args[]){  
        Integer i1 = new Integer(2);  
        Integer i2 = new Integer(1);  
        PrintCompare.printAns(  
            i1,i2,new IntCompare());  
        String s2 = "abcdef";  
        String s1 = "xyzaa";  
        PrintCompare.printAns(  
            s1,s2,new StringCompare());  
    }  
}
```

Since classes may have many methods and modes of use or operation, a given class may implement many different interfaces. For example, many classes support the `Cloneable` interface, which states that objects of the class may be duplicated (cloned).

Multiple Inheritance

We have seen that a class may be derived from a given parent class. It is sometimes useful to allow a class to be derived from more than one parent, inheriting members of all parents. This is *multiple inheritance*; it is allowed by C++ and Python, but not by Java.

The basic idea is that sometimes we want a “composite” object formed from more than one source. Hence a **Computer** object can be viewed as both a **PhysicalObject** (with height, weight, color, cost, etc.) and also a **CPUImplementation** (with memory size, processor design, processor speed, I/O ports, etc.)

Using multiple inheritance we merge aspects of a **PhysicalObject** and a **CPUImplementation**, and perhaps add additional data:

```
class PhysicalObject {
    float height, width, weight;
    Color outsideColor;
    ... }
class CPUImplementation {
    CPUClass CPUKind;
    int memorySize, CPUSpeed;
    ...
}
class Computer: PhysicalObject,
CPUImplementation {
    String myURL; ...
}
```

The advantages of multiple inheritance are obvious—you can build a class from many sources rather than just one.

There are problems though:

- If the same name appears in more than one parent, which is used? For example, if both parents contain a “copyright” field, which do you get? C++ forbids access to fields common to several parents, though accidental clashes of member names are certainly possible. Python relies on order of specification of the parents (which can be somewhat arbitrary).
- Access to fields and methods can be less efficient. A method in a parent class can’t know how fields in derived class will be allocated when multiple parents may exist. Hence some form of indirection may be needed. For example, in class **PhysicalObject**, we may believe

that **height** is the first field allocated, while in **CPUImplementation** we may believe that **CPUKind** is allocated first. But in class **Computer**, which contains both **height** and **CPUKind**, both fields *can’t* come first.