Subtyping and Inheritance in Java

Prakash Panangaden

November 16, 2006

1 Inheritance

One of the fundamental advances in the object-oriented paradigm is the ability to *reuse* code. It often happens that you find yourself coding a small variation of something that you had coded before. If your code is organized into classes, you might observe the following patterns. The new code that you want to write is a new class, but it looks just like an old class that you wrote a while ago except for a couple of methods. It is to handle situations like this that we have the notion of inheritance.

The following code is a basic example.

```
class myInt {

  //Instance variable
  private int n;

  //Constructor
  public myInt(int n){
    this.n = n;
  }

  //Instance methods
  public int getval(){
    return n;
  }

  public void increment(int n){
    this.n += n;
  }

  public myInt add(myInt N){
    return new myInt(this.n + N.getval());
  }
```

```
  public void show(){
    System.out.println(n);
  }
}
```

This class just has an integer in each object together with some basic methods. It is not a class we would really write. It is here for illustrative purposes only.

Now imagine that we decide to have "integers" made out of complex numbers. These numbers are called gaussian integers used by the great mathematician Gauss for his work in number theory. He discovered that as an algebraic system, these numbers behaved very much like ordinary integers. We might want to extend our ordinary integers to deal with these gaussian integers. Here is the code that does it. The keyword `extends` in the class declaration tells the system that you want all the code that worked for class `myInt` to be present in class `gaussInt`. We say that `gaussInt` *inherits* from `myInt`. We also say that `myInt` is the *superclass* and that `gaussInt` is the *subclass*.

```
class gaussInt extends myInt {

  //Instance variable
  private int m; //Represents the imaginary part

  /* We do not need the real part that is already present because we
     have inherited all the data and methods of myInt.  Thus the
     private int n is also present in every instance of a gaussInt. */

  //Constructor
  public gaussInt(int x, int y){
    super(x); //Special keyword
    this.m = y;
  }

  //Instance methods

  //This method is overridden from the superclass
  public void show(){
    System.out.println(
    "realpart is: " + this.getval() +" imagpart is: " + m);
  }

  public int realpart(){
    return getval();
  }
```

```
  /*The method getval is defined in the superclass. It is not defined
    here but it is inherited by this class so we can use it. */

  public int imagpart(){
    return m;
  }

  //This is an overloaded method
  public gaussInt add(gaussInt z){
    return new gaussInt(z.realpart() + realpart(),
                        z.imagpart() + imagpart());
  }

  public static void main(String[] args){
    gaussInt kreimhilde = new gaussInt(3,4);
    kreimhilde.show();
    kreimhilde.increment(2);
    kreimhilde.show();
  }
}//class gaussInt
```

There are a couple of things to note. In the constructor, we first want to use the constructor for the superclass; this is done with the keyword `super`. The `super` keyword invokes the superclass constructor and then continues with whatever is written next. The picture that you should have in mind is that an instance of `gaussInt` *contains* an instance of `myInt`.

Now you really want to inherit some of the methods unchanged, but some methods need to be modified. Obviously the `show` method of `myInt` is no use for `gaussInt`. In the subclass, you can give a new definition for an old method name. Thus in the above class, we have inherited `getval` and `increment` unchanged but we have modified `show`. This is called *overriding*. From class `gaussInt` you cannot use the `show` method of the superclass as it is well hidden.

There is also a more subtle phenomenon called *overloading*. Look at the `add` method in the two classes. At first sight, this looks like overriding. However the types of the arguments expected by the two definitions of the `add` method are different. The types of the arguements are called the *signature* of the method. Now when we use the same name for a method *but with a different signature* we get two different methods *with the same name*. Both methods are available from the subclass. In this case, from the class `gaussInt,` we can use both `add` methods. How does the system know which one to use? It looks at the types of the actual arguments and decides which one to use. Thus if you want to add an ordinary `myInt` to a `gaussInt`, then the `add` method of the superclass is used.

2 Subtyping and Interfaces

One of the features of object-oriented languages is a sophisticated type system that offers a possibility called "subtype polymorphism." In this section we will explain what this is and how it works in Java. One very important caveat when reading the extant literature: there is terrible confusion about the words "subtyping" and "inheritance". **They are not the same thing!** Yet one sees in books phrases like "inhertitance (i.e. subtyping) is very easy ..."! My only explanation is that lots of people really are clueless. Do not learn from them.

Recall that a type system classifies values into collections that reflect some structural or computational similarity. There is no reason that this classification should be into disjoint collections. Thus, for example, a value like 2 can be both an integer and a floating-point number. When a value can be in more than one type we say that we have a system that is *polymorphic* - from the Greek, meaning "having many shapes." A type system that allows procedures to gain in generality by exploiting the possibility that a value may be in many types is called a polymorphic type system.

The kind of polymorphism that one sees in Java is called "subtype polymorphism"[1] which is based on the idea that there may be a relation between types called *the subtyping relation*. Do not confuse subtyping with the notion of subset. We will say that a type A is a subtype of a type B if *whenever* the context requires an element of type B it can accept an element of type A. We write $A \lhd B$ to indicate this.

Here is the basic example of subtyping valid in many languages – but not in Sml. Consider the types `int` and `float`. It is easy to see that `int` is a subtype of `float`, in symbols $int \lhd float$. Whenever you need a floating-point value an integer can be used but definitely not the other way. For example, we may have a method for computing the prime factors of an integer, obviously such a method would not even make sense of a floating-point number.

How do we set up the subtyping relation? There are some built in instances of subtyping – such as $int \lhd float$ – but, clearly, this is not worth making such a fuss about. Where subtyping really comes into its own is with user-defined types. In Java, subtyping occurs automatically when you have inheritance; *this does not mean that subtyping and inheritance are the same thing*. You can also have instances of subtyping without any inheritance as we shall see.

Thus, if we declare a class B to be an extension of class A, we will have - in addition to all the inheritance - that $B \lhd A$. In other words, if at some point you have a method `foo` which expects an argument of type A; `foo` will always accept an object of type B. If we extend class B with class C then we have

$$C \lhd B \lhd A.$$

If we extend A with another class D then we will have $D \lhd A$ but there will be no subtyping relation between B and D or C and D. A method expecting an object of type A will accept objects of type B, C or D. This gives increased generality to the code.

[1] There are other kinds, most notably *parametric* polymorphism, which we have seen in Sml.

It is often the case that the inheritance hierarchy is not very "wide". In other words it is unlikely that a class A can be sensibly extended in many incompatible ways. This is because the code has to be inherited and usually only a few methods are modified. If we are modifying all or almost all the methods then it is clear that we are not really using inheritance. We are really trying to get the generality offered by subtype polymorphism.

Java supports subtype polymorphism independently of inheritance. This is done by **interfaces**. An interface is a declaration of a collection of method names - *without any method bodies* - and perhaps some constants. They describe a (sub)set of methods that a class might have. There is no code in an interface, hence there is nothing to inherit. A concrete class[2] may be declared to `implement` an interface. This is really a subtyping declaration. An interface names a new type (not a class) and when we say that a class P implements interface I we have set up the subtyping relation $P \lhd I$. Because there was no code in the interface, P does not inherit any code; but *it must have a method of the same name and type as every method name in the interface* I.

A class can implement many interfaces and thus one can have a complex type hierarchy with multiple subtyping. One often hears the phrase "interfaces allow Java to fake multiple inheritance". This is a source of confusion. What interfaces allow is multiple subtyping. Java definitely does not have multiple inheritance (C++ does have true multiple inheritance); wat it has is multiple subtyping.

Here is an example of the use of inheritance. Imagine that you have written code to draw the graph of a mathematical function. You want this to be abstracted on the function. You do not want to write code to plot a graph of the *sine* function and another different - but almost identical - piece of code to plot a graph of the *exp* function. You want the function - a piece of code - to be a *parameter*. We can do this with objects because objects can be passed around like any other piece of data but yet they carry code. What kind of object is a mathematical function? It expects a `double` argument and returns a `double` result. There might be other methods associated with function objects but the `plot` method does not care. It only cares that there is a method - called, say, `y` - such that `f.y(3.14159)` returns a double. So instead of having to know all about the details of the class of mathematical functions we just define an interface `plottable` with the one method `y` in it with the appropriate type. When we define our mathematical function objects we can make them as complicated as we please as long as we declare that they implement `plottable` and ensure that they really do have the method `y`. If we have another type of object - say `fin-data` - for financial data we would expect to define a totally different class with no relation to mathematical function objects. However, `fin-data` could also have a method `y` and be declared to implement `plottable`. Then our `plot` method works on totallly unrelated classes. We have achieves generality for our plotting method through subtype polymorphism in a situation far more general than could have been achieved by inheritance.

In between interfaces and classes are *abstract classes* that have some methods defined and some that are left blank as in interfaces. You can extend them as you would any class.

[2]We use the adjective "concrete" to mean that the class is completely defined, i.e. it has all the actual code for the methods.