Notes on Types

S. Arun-Kumar

Department of Computer Science and Engineering
Indian Institute of Technology
New Delhi, 110016
email: sak@cse.iitd.ernet.in

April 24, 2002

Chapter 1

Inheritance in Java

The use of abstract data types is intended to reduce code-duplication and encourage code-reuse and separate compilation of components of a large software system. To this end, most object-oriented languages provide two different facilities for this purpose,

1. **Polymorphism** by which a form of generic programming is supported. Generic components may be instantiated to specialized ones by initializing the type parameters of the generic program.
2. **Inheritance** by which new classes of objects may be derived from existing generic ones and customized by new and special functions, methods or properties not present in the generic class. The derived class is said to be a sub-class of the generic class. Equivalently the generic class A is said to be a super-class of the derived class.

The above features of object-oriented languages do encourage code-reuse and discourage code-duplication. However, a closer look at inheritance is required as it raise various issues. If a class A is specialized to class B, then it should be possible to avoid duplication of code from class A into class B, by merely establishing a relationship to that effect. Class B may then be said to *inherit* all the functions and methods of class A.
The most basic principle on which inheritance is based is the following property implemented in all obect-oriented languages.

> **Principle 1**. *If class A is a super-class of class B, then every object of class B is also an object of class A, though not vice-versa.*

The consequences of this property are explored through examples in the sequel. The examples are all working Java code.

1.1 Simple Inheritance

Consider the following *Java* class which allows the creation of "circle objects" from a class called `nCircle`. Along with it are defined some relevant properties computed as functions (such as area). Note the following peculiarties of Java classes[1]

1. The constructor `nCircle` which creates a fresh object of a class has the same name as the class.
2. There are two declarations of the constructor `nCircle`;

[1] They are not necessary for object-oriented languages in general, but they are mandatory in the case of Java

(a) the parameterless one will create a *default* circle object with radius 1.0, and

(b) the one with the formal parameter `r` will create a circle object with a given radius

3. The corresponding source file for this class will also be called `nCircle.java` and the compiled file will be called `nCircle.class`.

4. The methods `getRadius` and `findArea` are obvious.

5. Since there are no methods prefixed with the word `private`, all methods of this class are visible from outside the class and may be called by client programs.

6. The parameter `radius` is `private` to the object created and hence will not be accessible to any other object.

7. `Math.PI` is the value of π availbale from the class `Math` which is a predefined Java class.

8. Comments in Java are like comments in C or C++. Single-line comments begin with `//` and go upto the end of the line, and multi-line comments are bracketed by the pair of symbols `/*` and `*/` respectively.

```
public class nCircle
{  private double radius;

   public nCircle (double r)
    {  radius = r;}

   public nCircle ()  // default circle
    {  radius = 1.0;}

   public double getRadius ()
    { return radius; }

   public double findArea ()
    { return radius*radius*Math.PI; }
}
```

Given below is a simple "client" program which uses the class `nCircle` to create "circle" objects and print out their properties.

```
public class TestnCircle
{  public static void main (String[] args)
   {  // create and display mynCircle
        nCircle mynCircle = new nCircle(4.0);
        System.out.print ("mynCircle:\t");
        printnCircle (mynCircle);

        // create and display yournCircle
        nCircle yournCircle = new nCircle (6.0);
        System.out.print ("yournCircle:\t");
        printnCircle (yournCircle);
```

```
   }

   public static void printnCircle (nCircle c)
   {
        System.out.println ("radius = " + c.getRadius() +
                            "; area = "+c.findArea ());
   }
}
```

We now create a class `nCylinder` of "cylinder" objects. For this purpose we use the class `nCircle` with an additional height parameter to create the objects of the `nCylinder` class.

```
public class nCylinder extends nCircle
/*                     ^^^^^^^
   nCylinder is a sub-class of nCircle
   Hence nCircle is a super-class of nCylinder
*/
{
    private double height;

    // default constructor
    public nCylinder () {

    // super refers to the super-class constructor nCircle
          super ();
    //    ^^^^^
          height = 1.0;
    }

    // constructor
    public nCylinder (double r, double h) {
          super (r);
          height = h;
    }

    public double getHeight ()
    { return height; }


    // Assume nCylinder open on both ends and find surface area

    public double findArea ()
    //            ^^^^^^^^
    // This findArea overrides the findArea in the nCircle class

    { return 2.0*Math.PI*getRadius ()*height;}
    //                   ^^^^^^^^^
```

```
    // Method getRadius () inherited from the super-class nCircle

    public double findVolume ()
    { return super.findArea () * height; }
    //       ^^^^^^^^^^^^^^^
    // overrides the local method by calling the method of same name
    // from the super-class nCircle

}
```

Note the following features of the new class `nCylinder`.

1. The dependence on the class `nCircle` is obvious from the very first declaration of this class. `nCylinder` is therefore said to be a **sub-class** of `nCircle`. Equivalently `nCircle` is said to be the **super-class** of `nCylinder`. Hence every object of class `nCylinder` is also an object of class `nCircle`.

2. The reference `super` in the constructor `nCylinder` refers to the fact that the constructor `nCircle` is being invoked with an additional attribute viz. `height`. Note the use of the names in Java to facilitate this.

3. However the notion of area in the two classes must necessarily be different. Hence the *same* name `findArea` is used in `nCylinder` to define the surface area of a hollow cylinder, thus overriding (and hiding) the use of the method of the *same* name in the super-class `nCircle`.

4. However, while computing a new attribute viz. volume of a hollow cylinder it is necessary to invoke the `findArea` from the super-class to compute it.

The following is a simple and routine class meant to test the new `nCylinder` class.

```
public class TestnCylinder
{  public static void main (String[] args)
   {  // create and display myCircle
        nCylinder mynCylinder = new nCylinder(4.0, 5.0);
        System.out.print ("mynCylinder:\t");
        printnCylinder (mynCylinder);

        // create and display yournCylinder
        nCylinder yournCylinder = new nCylinder (6.0, 7.0);
        System.out.print ("yournCylinder:\t");
        printnCylinder (yournCylinder);

   }

   public static void printnCylinder (nCylinder c)
   {
        System.out.println ("radius = " + c.getRadius() +
                            "; area = "+ c.findArea ()  +
                                    "; volume = " + c.findVolume());
```

```
   }
}
```

1.2 Anomalies

The real fun begins now. For example, it is possible to construct various scenarios. For the scenarios we construct we assume the following.

1. Since the two classes `nCircle` and `nCylinder` are different classes they could have been developed by different teams who have no communication with each other.
2. No programmer of any class has any idea of the internal details of a class developed by any other person or team.

These assumptions are not unreasonable[2] at all given the premise that the main purpose of object-oriented programming is to facilitate such distributed programmming of large software systems.

1. Since any object of the class `nCylinder` (e.g. `yournCylinder`) is also implicitly a member of the super-class `nCircle`, it is possible to add the following code which "casts" a cylinder to a circle.

   ```
   // Casting the nCylinder into a nCircle
   nCircle yournCircle = (nCircle)yournCylinder;
   System.out.print ("yournCircle: \t");
   printnCircle (yournCircle);
   ```

   Note that the area that is output is the surface area of the cylinder since `yournCylinder` is originally an `nCylinder` and not an `nCircle`. To get a real circle you need to create one afresh (either from `nCircle` or `nCylinder`).

2. The `ncylinder` class contains no public method to find the area of the base of an `nCylinder` object[3]. One way to compute it is by dividing the surface area by the height. But one must watch out that the cylinder has not only a non-zero height, but is significant enough that there is no overflow or underflow.

3. It is perfectly possible for one to create an `ncylinder` object of height 0, but there is no way that can be regarded as a circle without an explicit conversion. Moreover the area of such a cylinder would be 0 and not the area of the circle.

To a large extent these anomalies are created because of the mismatch between the notion of types in programming (and mathematics) and their mismatch with the notion of classes and objects.

[2] though they may seem unreasonable for such silly programs like circles and cylinders

[3] This is just to emphasise the fact that it is always possible that the user of a class needs to extend it in some way or the other for their own purposes, which the original developer of the class may not be able to anticipate. This is one major reason why classes have to be extensible.

1.3 Subclassing and Subtyping

While the Principle 1 works perfectly well for types, we have seen that it does not work so well for classes. Hence classes[4] are not types. Ideally the Principle 1 will work for classes only under the following conditions.

1. Any extension of a class that creates a sub-class should only *specialize* the class rather than extend it in such a way as to include more elements than the original class did.

2. Pragmatically, such extensions should be such that functions from a super-class are not overridden.

3. However advocates of object-oriented programming claim that in general when specializing a class (e.g. specializing a class `polygon` to a sub-class `square`) some methods need to be redefined in order to make their computations more efficient (e.g. the area of a square is easier to calculate than that of a general polygon, hence the `findArea` method in `polygon` should be overridden by a the `findArea` method of `square`).

4. But how does one enforce that overridden methods are *semantically identical* to their namesakes in the parent class? The `findArea` method in our `nCylinder` example is a case of classic negligence.

5. While any form of semantic enforcement might be impossible, what has been advocated by many language theorists is the enforcement of type-checking rules which are "checkable" at compile-time. These are the following

   Contravariance rule on arguments The input arguments of the overriding function (viz. `findArea` in `nCylinder`) must be *super-types* of the input arguments of the overridden function (viz. `findArea` in `nCircle`).

   Covariance rule on results The result of the overriding function must be a *sub-type* of the result of the overridden function.

   The reason for enforcing these rules come from the fact that if $f : A \longrightarrow B$ is a function of type $A \longrightarrow B$ then for f to be overridden by $f' : A' \longrightarrow B'$ it is necessary that $A' \supseteq A$ and $B' \subseteq B$. While it is understandable that in the specialized subclass in which the method f' is used, $B' \subseteq B$ would hold, the condition $A' \supseteq A$ ensures that every element of A regardless of its pedigree does have the function f defined on it. Since any element on which the overridden function f' may be applicable should also be an element on which f is equally applicable. This is especially necessary when objects of a sub-class are "cast" into their super-class. However, none of the most commonly used object-oriented languages enforce even these type-checking rules.

[4] especially when they are badly designed and there is no way for a client of the class to know they are badly designed without having tried to use them!