Chapter 11

Implications of Inheritance

The decision to support inheritance and the principle of substitutability in a language sets off a series of chain reactions that end up impacting almost every aspect of a programming language. In this chapter, we will illustrate this point by considering some of the implications of the decision to support in a natural fashion the idea of the polymorphic variable.

The links between inheritance and other language features can be summarized as follows:

- In order to make the most effective use of object-oriented techniques, the language must support the polymorphic variable. A polymorphic variable is a variable that is declared as one type but actually maintains a value derived from a subtype of the declared type.

- Because at compile time we cannot determine the amount of memory that will be required to hold the value assigned to a polymorphic variable, all objects must reside on the heap, rather than on the stack.

- Because values are heap resident, the most natural interpretation of assignment and parameter passing uses reference semantics, rather than copy semantics.

- Similarly, the most natural interpretation of equality testing is to test object identity. However, since often the programmer requires a different meaning for equality, two different operators are necessary.

- Because values reside on the heap, there must be some memory management mechanism. Because assignment is by reference semantics, it is difficult for the programmer to determine when a value is no longer being used. Therefore a garbage collection system is necessary to recover unused memory.

Each of these points will be more fully developed in the following sections.

11.1 The Polymorphic Variable

As we will see in Chapter 12, a great deal of the power of the object-oriented features of Java comes through the use of a *polymorphic variable*. A polymorphic variable is declared as maintaining a value of one type, but in fact holds a value from another type. We have seen many such examples in the sample programs presented in Part 1. For instance, much of the standard user interface code thinks only that an application is an instance of class Frame, when in fact each program we created used inheritance to create a new type of application. Similarly, in the pin ball game construction program (Chapter 7) a variable was declared as holding a value of type PinBallTarget, when in fact it would hold a Hole or a ScorePad.

Figure 11.1 provides a class hierarchy consisting of three classes, Shape and two subclasses Circle and Square. In the small test program shown below variable named form is declared as type Shape, then assigned a value of type Circle. As expected, when the function describe() is invoked, the method that is executed is the procedure in class Circle, not the function inherited from class Shape. We will use this example class in the subsequent discussion in this chapter.

```
class ShapeTest {
    static public void main (String [ ] args) {
        Shape form = new Circle (10, 10, 5);
        System.out.println("form is " + form.describe());
    }
}
```

11.2 Memory Layout

Before we can describe the impact of the polymorphic variable on memory management, it is first necessary to review how variables are normally represented in memory in most programming languages. From the point of view of the memory manager, there are two major categories of memory values. These are *stack based* memory locations, and *heap based* memory values.

Stack based memory locations are tied to procedure entry and exit. When a procedure is started, space is allocated on a run-time stack for local variables. These values exist as long as the procedure is executing, and are erased, and the memory recovered, when the procedure exits. Figure 11.2 shows, for example, a snapshot of the run-time stack for the following simple recursive algorithm:

```
class FacTest {
    static public void main (String [ ] args) {
        int f = factorial(3);
```

```
class Shape {
    protected int x;
    protected int y;

    public Shape (int ix, int iy)
        { x = ix; y = iy; }

    public String describe ()
        { return "unknown shape"; }
}

class Square extends Shape {
    protected int side;

    public Square (int ix, int iy, int is)
        { super(ix, iy); side = is; }

    public String describe ()
        { return "square with side " + side; }
}

class Circle extends Shape {
    protected int radius;

    public Circle (int ix, int iy, int ir)
        { super(ix, iy); radius = ir; }

    public String describe ()
        { return "circle with radius " + radius; }
}
```

Figure 11.1: Shape classes and Polymorphic Variable

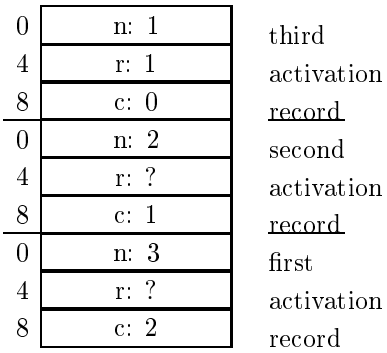


Figure 11.2: A Snapshot of the Activation Frame Stack

```
        System.out.println("Factorial of 3 is " + f);
    }

    static public int factorial (int n) {
        int c = n - 1;
        int r;
        if (c > 0)
            r = n * factorial(c);
        else
            r = 1;
        return r;
        }
}
```

The snapshot is taken after the procedure has recursed three times, just as the innermost procedure is starting to return. The data values for three procedures are shown. In the innermost procedure the variable `r` has been assigned the value 1, while the two pending procedures the value of `r` has yet to be determined.

There are a number of advantages of stack based memory allocation. All local variables can be allocated or deallocated as a block, for example, instead of one by one. This block is commonly called an *activation record*. Internally, variables can be described by their numeric offset within the activation record, rather than by their symbolic address. These numeric offsets have been noted in Figure 11.2. Most machines are much more efficient at dealing with numeric offsets than with symbolic names. Notice that each new activation record creates a new set of offsets, so that the offset is always relative to the activation frame in which a variable appears.

There is one serious disadvantage to stack based allocation. This is that these numeric offsets associated with variables must be determined at compile time, not a run time. In order to do this, the compiler must know the amount of memory to assign to each variable. In Figure 11.2, the compiler only knows that variable c can be found at address 8 because it knows that variable r, which starts at location 4, requires only four bytes.

But, this is exactly the information we do not know for a polymorphic variable. The storage requirements for a polymorphic variable value are determined when the value is created at run-time, and can even change during the course of execution. Recall the classes shown in Figure 11.1, and the memory requirements for the procedure main in the sample program described earlier. Here the variable form, which is declared as holding a Shape, can at one moment be holding a Circle, at another a Square, and so on. Both the subclasses add additional data fields that are not found as part of the parent class. Thus the translator cannot know, at compile time, exactly how much memory will be required to hold the variable form.

In Java, the solution to this problem is that objects are not stored on the activation record stack, but are instead stored on the heap. A heap is an alternative memory management system, one that is not tied to procedure entry and exit. Instead, memory is allocated on the heap when explicitly requested (to create a new object, using the new operator), and is freed, and recycled, when no longer needed. At run-time, when the memory is requested, the Java system knows precisely how much memory is required to hold a value. In this fashion the Java language avoids the need to predict, at compile time, the amount of memory that will be needed at run-time.

The code generated by the compiler, however, must still be able to accesses variables through numeric offsets, even through the actual heap addresses will not be known until run time. The solution to this dilemma is to use one level of indirection. Local variables are represented on the stack as pointer values. The size of a pointer is known at compile time, and is independent of the size of the object it points to. This pointer field is filled when an object is created.

It is said that the programming language Java has no pointers. This is true as far as the language the programmer sees is concerned. But ironically, this is only possible because *all* object values are, in fact, represented internally by pointers.

11.2.1 An Alternative Technique

It should be noted that the solution to this problem selected by the designers of Java is not the only possibility. This can be illustrated by considering the language C++, that uses an entirely different approach. C++ treats assignment of *variables* and assignment of *pointers* very differently.

The designers of C++ elected to store variable values on the stack. Thus, the memory allocated to a variable of type Shape is only large enough to hold a value of type Shape, not the additional fields added by the subclass Circle. During the process of assignment these extra fields are simply sliced off and discarded. Of course, the resulting value is then no

longer a Circle. For this reason, when we try to execute a member function, such as the describe function, the code executed will be that associated with class Shape, not the value associated with class Circle, as in Java. The programmer who uses both C++ and Java should be aware of this subtle, but nevertheless important difference.

11.3 Assignment

In Section 11.2 we described why values in Java are most naturally maintained on the heap, rather than being held in the activation record stack. Because to the compiler the underlying "value" of a variable is simply a pointer into the heap, the most natural semantics for assignment simply copy this pointer value. In this manner, the right and left sides of an assignment statement end up referring to the same object. This is often termed *reference* semantics (sometimes also called *pointer* semantics). The consequences of this interpretation of assignment are subtle, but are again a key point for Java programmers to remember. Suppose we create a simple class Box as follows:

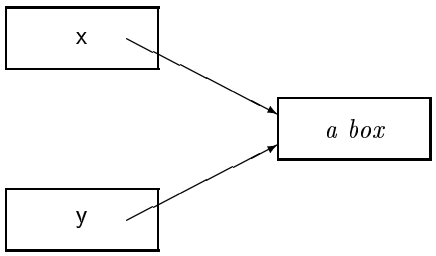
```
public class Box {
    private int value;

    public Box () { value = 0; }
    public void setValue (int v) { value = v; }
    public int getValue () { return value; }
}
```

Now imagine that we create a new box, assign it to a variable x, and set the internal value to 7. We then assign the box held by x to another variable named y. Since both x and y now hold non-null values of type Box, the programmer might assume that they are distinct. But, in fact, they are exactly the same box, as can be verified by changing the value held in the y box and printing the value held by the x box:

```
public class BoxTest {
    static public void main (String [ ] args) {

        Box x = new Box();
        x.setValue (7);      // set value of x

        Box y = x;     // assign y the same value as x
        y.setValue (11); // change value of y

        System.out.println("contents of x " + x.getValue());
        System.out.println("contents of y " + y.getValue());
```

```
    }
}
```

The key observation is that the two variables, although assigned separate locations on the activation record stack, nevertheless point to the same location on the heap:



11.3.1 Clones

If the desired effect of an assignment is indeed a copy, then the programmer must indicate this. One way would be to explicitly create a new value, copying the internal contents from the existing value:

```
    // create new box with same value as x
Box y = new Box (x.getValue());
```

If making copies is a common operation, it might be better to provide a method in the original class:

```
public class Box {
    ...
    public Box copy() { // make copy of box
        Box b = new Box();
        b.setValue (getValue());
        return b;
    }
    ...
}
```

A copy of a box is then created by invoking the `copy()` method:

```
    // create new box with same value as x
Box y = x.copy();
```

There is no general mechanism in Java to copy an arbitrary object, however the base class Object does provide a protected method named clone() that creates a bit-wise copy of the receiver, as well as an interface Cloneable that represents objects that can be cloned. Several methods in the Java library require that arguments be values that are cloneable.

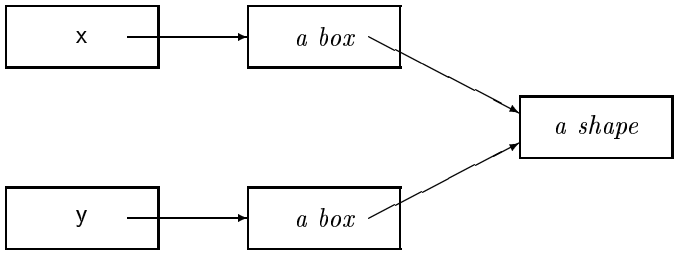
To create a class that is cloneable, the programmer must not only override the clone method to make it public, but also explicitly indicate that the result satisfies the Cloneable interface. The following, for example, shows how to create a cloneable box.

```
public class Box implements Cloneable {
    private int value;

    public Box () { value = 0; }
    public void setValue (int v) { value = v; }
    public int getValue () { return value; }

    public Object clone () {
        Box b = new Box();
        b.setValue (getValue());
        return b;
        }
}
```

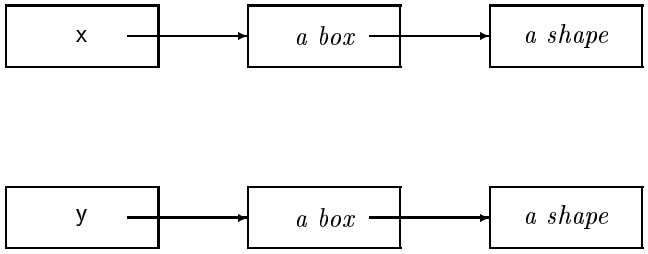
The clone method is declared as yielding a result of type Object. This property cannot be modified when the method is overridden. As a consequence, the result of cloning a value must be cast to the actual type before it can be assigned to a variable.

```
public class BoxTest {
    static public void main (String [ ] args) {

        Box x = new Box();
        x.setValue (7);

        Box y = (Box) x.clone(); // assign copy of x to y

        y.setValue (11); // change value of x

        System.out.println("contents of x " + x.getValue());
        System.out.println("contents of y " + y.getValue());
    }
}
```

As always, there are subtleties here that can trap the unwary programmer. Consider the object that is being held by our box. Imagine, instead of simply an integer, that it is something more complex, such as a Shape. Should a clone also clone this value, or just copy it? A copy results in two distinct boxes, but ones that share a common value. This is called a *shallow copy*.



Cloning the contents of the box (which must therefore be itself a type that is cloneable) results in two box values which are not only themselves distinct, but which point to values that are also distinct. This is termed a *deep copy*.



Whether a copy should be shallow or deep is something that must be determined by the programmer when they override the clone interface.

11.3.2 Parameters are a form of Assignment

Note that a variable passed as an argument to a member function can be considered to be a form of assignment, in that the parameter passing, as with assignment, results in the same value being accessible through two different names. Thus, the issues raised in Section 11.3 regarding assignment apply equally to parameter values. Consider the function sneeky in the following example, which modifies the value held in a box that is passed through a parameter value.

```
public class BoxTest {
    static public void main (String [ ] args) {
```

```
        Box x = new Box();
        x.setValue (7);

        sneeky (x);

        System.out.println("contents of x " + x.getValue());
    }

    static void sneeky (Box y) {
        y.setValue (11); // change value of parameter
        }
}
```

A programmer who passes a box to this function, as shown in the main procedure, could subsequently see the resulting change in a local variable.

A Java programmer should always keep in mind that when a value is passed to a procedure, a certain degree of control over the variable is lost. In particular, the procedure is free to invoke any method applicable to the parameter type, which could result in the state of the variable being changed, as in this example.

11.4 Equality Test

For basic data types the concept of equality is relatively simple. The value 7 should clearly be equivalent to $3 + 4$, for example, because we think of integers as being unique entities–there is one and only one 7 value. This is true even when there are two syntactic representations for the same quantity, for example the ASCII value of the letter *a* is 141, and thus `'\141'` is the same as `'a'`.

The situation becomes slightly more complicated when the two values being defined are not the same type. For example, should the value 2 (an integer constant) be considered equal to 2.0 (a floating-point constant)? The Java language says that the two values are equivalent in this case, since the integer value can be *converted* into a floating point value, and the two floating values compared. Thus, all the following expressions will yield a true result.

```
7 == (3 + 4)
'a' == '\141'
2 == 2.0
```

When the concept of equality testing is expanded to include objects, the most natural interpretation becomes less obvious. In Section 11.3 it was argued that because objects are

internally represented by pointers, the natural interpretation of assignment uses reference semantics. One interpretation of equality testing follows the same reasoning. That is, two objects can be considered equal if they are identically the same. This form of equality testing is often termed as testing object *identity*, and is the interpretation provided by the operator == and its inverse, the operator ! =. This can cause certain anomalies. For example, although 7 is equal to 3 + 4, the following code fragment will nevertheless show that an Integer value 7 is a distinct object from a different Integer object, even if it has the same value:

```
Integer x = new Integer(7);
Integer y = new Integer(3 + 4);
if (x == y)
    System.out.println("equivalent");
else
    System.out.println("not equivalent");
```

The Java compiler does apply type checking rules to the two arguments, which will help detect many programming errors. For example, although a numeric value can be compared to another numeric, a numeric cannot be compared to a different type of object (for example, a String). Two object values can be compared if they are the same type, or if the class of one can be converted into the class of the second. For example, a variable that was declared to be an instance of class Shape could be compared with a variable of type Circle, since a Circle would be converted into a Shape. A particular instance of the conversion rule is one of the more frequent uses of the == operator; namely, any object can be compared to the constant null, since the value null can be assigned to any object type.

Often object identity is not the relation one would like to test, and instead one is interested in object *equality*. This is provided by the method equals, which is defined by the base class Object and redefined by a number of classes. The following, for example, would return true using the equals function, but would not be true using the == operator:

```
String a = "abc";
String b = "abc";
Integer c = new Integer(7);
Integer d = new Integer(3 + 4);
if (a.equals(b))
    System.out.println("strings are equal");
if (c.equals(d))
    System.out.println("integers are equal");
```

Because the equals function is defined in class Object, it can be used with any object type. However, for the same reason, the argument is declared only as type Object. This

means that any two objects can be compared for equality, even if there is no possible way that for either to be assigned to the other:

```
if (a.equals(c)) // can never be true
    System.out.println("string equal to integer");
```

The developer of a class is free to override the equals operator and thereby allow comparison between objects. Since the argument is an Object it must first be tested to ensure it is the correct type. By convention, the value false is returned in cases where the type is not correct. The following, for example, would be how one could define a function to compare two instances of class Circle (Figure 11.1).

```
class Circle extends Shape {
    ...
    public boolean equals (Object arg) {
        if (arg instanceof Circle) {
                // convert argument to circle
            Circle argc = (Circle) arg;
            if (radius == argc.radius)
                return true;  // just test radius
        }
        return false; // return false otherwise
    }
}
```

Because the type of the argument is not necessarily the same as the type of the receiver, unusual situations can occur if the programmer is not careful. For example, suppose we defined equals in class Shape from Figure 11.1 to test equality of the x and y values, but forgot to also override the function in class Square. It could happen in this situation that if we tried to compare a square and a circle, the comparison would be true one way and false the other.

```
Square s = new Square(10, 10, 5);
Circle c = new Circle(10, 10, 5);
if (s.equals(c)) // true, since method in shape is used
    System.out.println("square equal to circle");
if (c.equals(s)) // false, since method in circle is used
    System.out.println("circle equal to square");
```

When overriding the equals method the programmer should be careful to avoid this problem, and ensure that the resulting functions are both symmetric (if x is equal to y, then y is equal to x) and associative (if x is equal to y and y is equal to z, then x is equal to z).

A good rule of thumb is to use == when testing numeric quantities, and when testing an object against the constant null. In all other situations the function equals should be used.

11.5 Garbage Collection

In Section 11.2 it was argued that support for polymorphic variables naturally implies that values are allocated on the heap, rather than on the stack. Memory of any type is always a finite resource, which must be managed if a program is to avoid running out of storage. In order to prevent this problem, both stack and heap based memory is recycled, with new memory requests being assigned the same locations as previous memory values that are no longer being used.

Unlike stack based memory allocation, heap based memory management is not tied to procedure activation, and is thus not automatically recovered when a procedure returns. This means an alternative mechanism must be introduced.

Two different approaches to the recovery of heap based memory values are found in programming languages. In languages such as Object Pascal or C++ it is up to the programmer to explicitly indicate when a memory value is no longer being used, and can therefore be reused to satisfy new memory requests. In Object Pascal, for example, this is accomplished by means of the statement dispose:

```
var
    aShape : Shape;
begin
    new (aShape); (* allocate a new shape *)
    ...
    dispose (aShape); (* free memory used by variable *)
end.
```

In such languages, if an object is to be freed, it must be "owned" by some variable (the variable that will be the target for the free request). But Java values are simply references, and are shared equally by all variables that refer to the same value. If a programmer assigns a value to another variable, or passes the value as an argument, the programmer may no longer be aware of how many references a value might have.

Leaving to the programmer the responsibility for freeing memory in this situation exposes a program to a number of common errors, such as freeing the same memory location twice. Even more commonly, programmers avoid committing this error by simply never freeing memory, causing long-running programs to slowly degrade as they consume more and more memory resources.

For these reasons, the designers of Java elected a different approach. Rather than having the programmer indicate when a value is no longer needed, a run-time system is provided that periodically searches the memory being used by a program to discover which heap

values are being accessed and, more importantly, which heap values are no longer being referenced by any variable and can therefore be recovered and recycled. This mechanism is known as the *garbage collection* system.

The use of a garbage collection system in Java is a compromise. The task of garbage collection does exact a toll in execution time. However, in return a garbage collection system greatly simplifies the programming process, and eliminates several categories of common programming errors. For most programs, the improvements in reliability are well worth the execution time overhead.

11.6 Chapter Summary

The idea of a polymorphic variable is an extremely powerful concept, one we will explore in detail in later chapters. However, the decision to support the concept of a polymorphic variable raises a number of subtle and difficult issues in other aspects of the language. In this chapter we have investigated some of these, showing how inheritances alters the way the language must handle storage management, the concept of assignment, and the testing of two values for equality.

Study Questions

1. What is a polymorphic variable?
2. From the language implementation point of view, what are the two major categories of memory values?
3. How does the idea of a polymorphic variable conflict with the ability to determine memory requirements at compile time?
4. What does it mean to say that Java uses reference semantics for assignment?
5. What must a programmer do to create a class that supports the Cloneable interface?
6. What is the difference between a deep and shallow copy?
7. In what way is passing a parameter similar to an assignment?
8. What is the difference between the == operator and the equals() method?
9. What task is being performed by the garbage collection system in the Java run-time library?
10. What are some advantages of a language that uses garbage collection? What are some disadvantages?

Exercises

1. Rewrite the Shape classes shown in Figure 11.1 so that they support the cloneable interface.

2. Rewrite the class Box so that it holds values that are cloneable, and when cloned it created a deep copy.