8. Memory Representation of Primitive Values and Objects

By now, we have seen how to assign primitive and object values to primitive and object variables. In this chapter, we will focus a bit more in-depth on such assignments. In particular, we will study the difference between the way these two kinds of values are stored in memory, and the implication this difference has for the assignment statement.

Storing Primitive Values and Variables

Consider how the assignment:

```
int i = 5;
```

is processed by the computer. From a programmer’s point of view, of course, the variable `i` gets assigned the value 5. However, let us look at under the hood and see what Java exactly does when processing this statement.

Java creates an integer value, 5, and stores it in a memory block. A memory block is simply a set of contiguous memory cells that store some program value. It also creates a memory block for the variable `i`. When the assignment statement is executed, the contents of the value block are copied into the variable block.

The two blocks have the same size because the value and the variable have the same type. As a result, the value “fits” exactly in the variable. The size of the block for an integer value is 1 memory word or 32 bits, as we saw earlier. Figure 1 illustrates this discussion.

The statement:

```
double d = 5.5;
```

is processed similarly except that Java manipulated blocks of 2 words instead of 1 word, because the size of a double is 2 words.

Assignment of one variable to another is handled similarly:
double e = d;

Each memory block is identified by its memory address, which is listed on its left in the figure. While we think in terms of high-level specifiers such as `i` and `5`, the processor, in fact, works in terms of these addresses. The compiler converts these names to addresses, so that the human and processor can speak different languages. It is for this reason that a compiler is also called a translator.

In summary:
values and variables of same type take same amount of fixed storage.

the storage consists of one or more consecutive memory words that together form a memory block.

values and variables of different types may take different storage.

Storing Object Values and Variables

We cannot apply these rules directly to objects. The reason is that, as we saw before, instances of the same class can have different physical structures, which, in turn, means they occupy different sized memory blocks. For example, as we saw earlier, instances of AnAnotherLine can have different physical structures as their location instance variable can be assigned ACartesianPoint or APolarPoint. If a variable and value of an object type occupied the same amount of space, then what should we do if the variable holding some instance, i1, of a class, another instance, i2, of the class that is of a different size? One solution is to move the variable to a different area of memory. This means, given a variable, we would have to search some table to determine its location, which makes variable accesses very time inefficient. Another alternative is to allocate the variable the maximum sized object that can assigned to it. However, this approach is space inefficient – in fact, when you study recursive structures, you will see that this approach is not even feasible as there may not be any limits on the maximum-sized structure that can be assigned to a variable.

A compromise solution is to mix elements of the above two solutions. As in the second solution, we will allocate to an object variable a fixed-size memory block. However, this storage area will be very small – exactly one memory word. As in the first solution, we will incur some runtime overhead in accessing an object variable. However, this overhead will be smaller than the search overhead of the first solution. In
our solution, the memory block associated with an object does not contain a copy of the value assigned to it. Instead, it contains a “pointer” to the value.

To illustrate and explain in detail, consider again the ACartesianPoint class we saw before:

```java
public class ACartesianPoint implements Point {
    int x,y;
    public ACartesianPoint(int initX, int initY) {
        x = initX;
        y = initY;
    }
    ...
}
```

Consider the following statement in which a new instance of ACartesianPoint is created and stored in a variable of type Point:

```java
Point p1 = new ACartesianPoint(50,100);
```

As before, both values and the variables are allocated memory. However, each assignment copies into the variable’s block, not the contents of the value block, but instead its address. All Java addresses are 1 word long, so all object variables are allocated a 1-word block, regardless of their types, which was not the case with the primitive variables. For example, double variable, d, and integer variable, i, we saw above were of different sizes.

All objects, however, are not of the same size. When a new object is created, a composite memory block consisting of a series of consecutive blocks, one for each instance variable of the object, is created. Thus, since ACartesianPoint has two int instance variables, a block consisting of two integer variables is created. The sizes of two objects of different types can be different depending on the number and the types of their instances variables. Figure 3 illustrates the above discussion.

Since an object variable stores addresses, it also called a pointer variable or reference variable, and the address stored in it a pointer or reference.

Variable reference is more complicated when the variable is a pointer variable. Consider:

```java
System.out.println(i)
```

Java accesses memory at the address associated with i, and uses the value stored in the println. In contrast, consider:

```java
System.out.println(p1)
```

Java accesses memory at the address associated with p1, finds another address there, and then uses this address to find the integer value. Thus, we do not go directly from a variable address to its value, but instead, indirectly using the value address or pointer. In some languages, the programmer is responsible for doing the indirection or dereferencing. For instance, in Pascal, given an integer pointer variable p1, we need to type:
to refer to the value to which it refers. Thus, the equivalent statement in Pascal would be:

\[ \text{writeln}(p1^) \]

Java, however, automatically does the dereferencing for us. In fact, we cannot directly access the address stored in it. Thus, we are not even aware that the variable is a pointer variable. Sometimes, the term *pointer* is used for a variable that must be explicitly dereferenced and *reference* for a variable that is automatically dereferenced. For this reason, some people say that Java has no pointer variables. However, we will use these two terms interchangeably. Dereferencing increases memory access time – this is the reason why we do not use it for primitive variables, which explains why primitives and objects are treated differently.

The special value, *null*, we saw before, can be assigned to a pointer variable:

```java
Point p2 = null;
```

In fact, if we do not initialize a pointer variable, this is the value stored in its memory block. It denotes the absence of a legal object assigned to the variable. This value is not itself a legal object, and does not have a class. If we try to access a member of this value:

```java
null.toString();
```

we get a *NullPointerException*, which some of you may have already seen. Later, will see how we can determine whether the value of a pointer variable is *null* or not, and hence, whether we can access its member.
**Pointer Assignment**

Assignment can be tricky with pointers. Consider:

```java
Point p1 = new A CartesianPoint(50, 50);
Point p2 = p1;
p1.setX(100);
System.out.println(p2.getX());
```

When `p1` is assigned to `p2`, the pointer stored in `p1` is copied into `p2`, not the object itself. Both variables share the same object, and thus, the code will print 100 and not 50, as you might expect.

Sharing allows us to create graph structures, such as the one shown in Figure 4, which may also be represented as shown in Figure 5. You will study such structures in more depth in a data structure course. They support useful applications such as two Web pages pointing to the same page.