CS125 : Introduction to Computer Science

Lecture Notes #13
Reference Variables and Objects

Lecture 13 : Reference Variables and Objects

Reference variables

In reality, the line:

```
int[] scores = new int[10];
```

is quite similar to the line:

```
int b = 2;
```

in that, in both cases, we have put a declaration and an initialization on the same line. The latter could have been done on two lines as well:

```
int b;
b = 2;
```

and likewise, the former could also have been done on two lines:

```
int[] scores;
scores = new int[10];
```

That is, rather than thinking of the single line:

```
int[] scores = new int[10];
```

as some big mess of syntax that you need to create an array, take note that it is really two different things put together:

- a variable declaration `int[] scores;`
- an initialization `scores = new int[10];`

and that the line:

```
int[] scores = new int[10];
```

is declaring and then initializing a variable, just as the line:

```
int b = 2;
```

is doing. However, the type of this varible is `int[]`, and that was not one of our eight primitive types (`int` is a primitive type, but not `int[]`). So, since the type `int[]` of this variable `score` is not one of our eight primitive types, `score` is not a primitive type variable. Instead, it is something known as a *reference variable*. Any variable which is *not* of a primitive type will be a reference variable; array variables merely happen to be the first example of this that we've seen. But before the semester is over, we will see many more examples of variables which are not of primitive types – that is, we will see many more examples of reference variables.

So, how are primitive type variables and reference variables different? Well, the integer variable declaration will produce an integer variable, but the reference variable declaration does not produce an array. Instead, what it produces is a *reference* (that's why it is called a reference variable). If a primitive-type variable is similar to a box that holds a piece of data, then think of a reference variable as being similar to an "arrow":

```
                --------
                |      |
b - - - - - |  2   |
                |______|


                 --------
                 |      |
scores - - - |   ---|----->
                 |______|
```

That is, you can store an integer value *inside* the `int` variable, but you don't store any data value of your own inside the reference variable, because it is not designed to hold your data. Instead, its job is to refer you to some other memory location where some important data *is* located – in this case, where the array itself is located.

Objects and reference variables

There are four things that every variable you declare in Java will have:

1. A name
2. A type
3. A location in memory where it will be stored, i.e. an address. (You don't choose this; the compiler and run-time environment choose it.
4. A value that the variable stores.

Since we have no idea what the value in the variable is, until we initialize the variable, we're going to set that property aside for now, and focus on just the first three properties, all of which are well-defined as soon as the variable is declared.

It does not matter whether you are talking about a primitive-type variable or a reference variable. In the declaration `int b;`, the variable name is `b`, the type is `int`, and there will be some location where this variable gets stored. (Let's assume for the sake of this discussion that the compiler chooses address `a80` for that variable.) In the declaration `int[] x;`, the variable name is `x`, the type is `int[]` (i.e., "reference to integer array"), and there will be some location where this variable gets stored. (Let's assume for the sake of this discussion that the compiler chooses address `a88` for that variable.)

So, let's look at the section of memory where these variables are stored:

```
     |          .
     |          .
     |          .
     |__________________________  _____
a80  |__________________________       |
a81  |__________________________       | b
a82  |__________________________       |
a83  |__________________________  _____|
a84  |__________________________
a85  |__________________________
a86  |__________________________
a87  |__________________________  _____
a88  |__________________________       |
a89  |__________________________       | x
a90  |__________________________       |
a91  |__________________________  _____|
a92  |__________________________
a93  |__________________________
a94  |__________________________
a95  |__________________________
a96  |__________________________
a97  |__________________________
a98  |__________________________
a99  |__________________________
a100 |__________________________
a101 |__________________________
a102 |__________________________
a103 |__________________________
a104 |__________________________
a105 |__________________________
a106 |__________________________
     |        .
     |        .
     |        .
     |        .
```

Variables of type `int` take up 32 bits, so the variable `b` needs four 8-bit memory cells to store its value. Likewise, we'll assume reference variables need 32 bits as well, so therefore the reference variable `x` would also need 32 bits, i.e. four 8-bit memory cells.

The way these variables differ is in the *data* they store. When you execute the assignment statement `b = 2;`, the actual value 2 (or rather, the encoded version of it) gets stored in the memory location corresponding to the variable `b` (which starts at the memory location with address `a80` in our example).

```
      |         .
      |         .
      |         .
      |__________________________  _____
a80   |                                 |
a81   |  32 bits that, together,        |  b
a82   |   encode the integer 2          |
a83   |__________________________  _____|
a84   |__________________________
a85   |__________________________
a86   |__________________________
a87   |__________________________  _____
a88   |__________________________       |
a89   |__________________________       | x
a90   |__________________________       |
a91   |__________________________  _____|
a92   |__________________________
a93   |__________________________
a94   |__________________________
a95   |__________________________
a96   |__________________________
a97   |__________________________
a98   |__________________________
a99   |__________________________
a100  |__________________________
a101  |__________________________
a102  |__________________________
a103  |__________________________
a104  |__________________________
a105  |__________________________
a106  |__________________________
      |       .
      |       .
      |       .
      |       .
```

Now, remember that in an assignment statement, our variable type and value type had to match. You could only assign `char` values to `char` variables. You could only assign `boolean` values to `boolean` variables. And so on. So, we have this reference variable, `x`. We want to have an assignment statement such as `x = ?;` but so far we have not discussed what we would replace the question mark with. What values can be written into reference variables?

And the answer is: memory addresses. The data that gets stored inside reference variables are bit strings that tell the machine to go to particular memory locations. So, just as we've labelled our memory locations with addresses like `a5` and `a7`, those exact same kinds of labels – or rather, their bit string encodings – are what get stored in our reference variables.

The question is, how can we obtain memory addresses to actually store in our reference variables? And the answer is, we do this by using the keyword `new`. That is, we would use an expression such as the following to create an array and return the address of that array:

```
    new int[3];
```

You have seen this expression before, as part of an array creation line:

```
int[] x = new int[3];
```

and we said at the start of this notes packet that the above line could actually be split into two parts:

```
int[] x;
x = new int[3];
```

so, this expression is used in the second line above and whatever the expression evaluates to is being written into the variable `x`.

What this expression is doing is telling the computer to set aside enough memory for an integer array of size `3`. This process is known as an *allocation*, and thus we say that we are *allocating memory*. This chunk of memory we ask the system to set aside for us is called an *object* – in Java, any memory you allocate using `new` is an *object*, and any memory you obtain through a variable declaration statement is a *variable*. You never create objects using variable declarations statements – you only create objects by using `new`. Variables are declared (using variable declaration statements), and objects are allocated.

All expressions involving the Java keyword "`new`", will set aside some amount of memory for an object, and then inform the program of where the start of that memory is located in the machine. The expression actually *evaluates to a memory address.* So, for example, if the collection of memory cells set aside for the array begins at address `a100`, then the expression will evaluate to `a100`, since that is the starting location of the cells that were set side for the array.

```
     |         .
     |         .
     |         .
     |__________________________  _____
a80  |                                 |
a81  |  32 bits that, together,        |  b
a82  |   encode the integer 2          |
a83  |__________________________  _____|
a84  |__________________________
a85  |__________________________
a86  |__________________________
a87  |__________________________  _____
a88  |__________________________       |
a89  |__________________________       | x
a90  |__________________________       |
a91  |__________________________  _____|
a92  |__________________________
a93  |__________________________
a94  |__________________________
a95  |__________________________
a96  |__________________________
a97  |__________________________
a98  |__________________________
a99  |__________________________  _____
a100 |__________________________       | The memory allocated for
a101 |__________________________       |  the array object begins
a102 |__________________________       |  here (and ends somewhere
a103 |__________________________       |  further down, beyond the
a104 |__________________________       |  edge of our picture).
a105 |__________________________       |
a106 |__________________________       |
     |       .                      .
     |       .                      .
     |       .                      .
```

So, we would have used this expression as follows:

```
x = new int[3];
```

and when the expression `new int[3]` evaluates to `a100`, that address would be what is stored in the reference variable `x`:

```
     |         .
     |         .
     |         .
     |__________________________  _____
a80  |                                 |
a81  |  32 bits that, together,        |  b
a82  |   encode the integer 2          |
a83  |__________________________  _____|
a84  |__________________________
a85  |__________________________
a86  |__________________________
a87  |__________________________  _____
a88  |  32 bits that, together,        |  x
a89  |   encode the memory             |
a90  |     address a100                |
a91  |__________________________  _____|
a92  |__________________________
a93  |__________________________
a94  |__________________________
a95  |__________________________
a96  |__________________________
a97  |__________________________
a98  |__________________________
a99  |__________________________  _____
a100 |__________________________  cell |
a101 |__________________________  0 of |
a102 |__________________________  array|
a103 |__________________________  _____|
a104 |__________________________  cell |
a105 |__________________________  1 of |  the entire array
a106 |__________________________  array|  (12 memory cells)
a107 |__________________________  _____|
a108 |__________________________  cell |
a109 |__________________________  2 of |
a110 |__________________________  array|
a111 |__________________________  _____|
a112 |__________________________
     |       .
     |       .
     |       .
```

Finally, when you have a statement such as:

```
    x[2] = 17;
```

which uses the square brackets, that tells the machine to (1) read the memory address stored in `x`, and then (2) taking that location as the start of the array, jump to the start of cell 2 of the array, and then (3) write `17` into that cell. So, step (1) would read the value `a100` from `x`, and then step (2) says that, since `a100` is cell 0, `a100 + 2 * sizeOfInt = a100 + 2 * 4 memory cells`

= `a108` must be the address of cell 2, so go to `a108`. Finally, since that memory location, `a108`, is `x[2]`, that is the location into which 17 is written:

```
     |         .
     |         .
     |         .
     |__________________________  _____
a80  |                                |
a81  |  32 bits that, together,       |  b
a82  |   encode the integer 2         |
a83  |__________________________  _____|
a84  |__________________________
a85  |__________________________
a86  |__________________________
a87  |__________________________  _____
a88  |  32 bits that, together,       |  x
a89  |   encode the memory            |
a90  |     address a100               |
a91  |__________________________  _____|
a92  |__________________________
a93  |__________________________
a94  |__________________________
a95  |__________________________
a96  |__________________________
a97  |__________________________
a98  |__________________________
a99  |__________________________  _____
a100 |__________________________  cell |
a101 |__________________________  0 of |
a102 |__________________________  array|
a103 |__________________________  _____|
a104 |__________________________  cell |
a105 |__________________________  1 of |  the entire array
a106 |__________________________  array|  (12 memory cells)
a107 |__________________________  _____|
a108 |                            cell |
a109 | 32 bits that, together,    2 of |
a110 |  encode the integer 17     array|
a111 |__________________________  _____|
a112 |__________________________
     |       .
     |       .
     |       .
```

This memory that we request from the system using `new`, we call it *dynamic memory*, we say that we *allocate* that memory from the *heap*, and we call that memory an *object* – i.e. we *declare* variables, but we *allocate* objects. Note that objects *do not have names!!!!*. Therefore, the only way we can actually refer to objects in our program is by having reference variables that store labels of the memory addresses of those objects. This is why reference variables are so important. Without the ability to store the addresses of object locations – the task which reference variables are designed to do – we could never actually reach our objects, and so there would be no point in creating them. (Objects are not bound by normal scope rules, which is *why* we want to create them.)

In Java, there are only three different types of data storage. The first are variables of primitive types, and you've used those already. The second are reference variables, which store memory addresses rather than primitive-type values. And the third kind of data is an object, which is stored somewhere in memory just like a variable, and which holds some kind of data, just like a variable – but which does not have a name of its own, unlike a variable.

Our more typical "abstract" picture

Now, it is not necessary to think about the array that way in order to program in Java. In fact, thinking about specific memory locations and such is usually more a hinderance than a help. So instead, we tend to hide away all those details and instead just draw pictures.

So, as we did earlier, we can draw reference variables as arrows. The declaration:

```
int[] scores;
```

produces a reference variable, which we will draw as follows:

```
           --------
           |      |
scores - - |   ---|----->
           |______|
```

Now, it turns out that reference variables are always automatically initialized by the Java virtual machine. The value a reference variable has when you first declare it, is the value `null`. The value `null` is a "reference variable literal" in the same way that `2` is an `int` literal and `45.4` is a `double` literal. When a reference variable "points to nothing", it is pointing to `null`. We draw it like this:

```
           --------
           |      |         _
scores - - |   ---|-----> |\|
           |______|
```

i.e. a little box with a slash through it. Sometimes the slash is put through the reference variable picture itself:

```
           --------
           |\_    |
scores - - |  \_  |
           |____\_|
```

Either way, that's how we convey "reference variable points to `null`" or "reference variable holds the value `null`" in our pictures. If we want to check if a reference variable "points to nothing" instead of pointing to an actual object, we can use boolean expressions such as:

```
scores == null
```

If that expression evaluates to `true`, then the reference variable points to `null`, just as if we have an integer variable `x`, then if:

```
x == 2
```

evaluates to `true`, we know `x` contains the value `2`. We can also assign a reference variable to store `null` via an assignment statement such as:

```
scores = null;
```

just as the assignment statement:

```
x = 2;
```

would store the value `2` in the integer variable `x`.

Now, we have not created the array yet; that is done with the following expression:

```
 new int[6]
```

giving the following picture:

```
           --------
           |      |        _
scores - - |   ---|-----> |\|
           |______|


                   -------------------------------
                   |    |    |    |    |    |    |
                   |    |    |    |    |    |    |
                   |____|____|____|____|____|____|
                      0    1    2    3    4    5
```

We simply draw the six array cells together floating off in space; we don't worry about where they are located in memory.

Finally, the `new int[6]` expression would have been part of an assignment statement:

```
        scores = new int[6];
```

and the location of the array is written into `scores`. In our picture, we will show this by having the reference variable `scores` point to the array.

```
           --------
           |      |
scores - - |  |   |
           |__|___|
              |
              |    -------------------------------
              |    |    |    |    |    |    |    |
              |--->|    |    |    |    |    |    |
                   |____|____|____|____|____|____|
                      0    1    2    3    4    5
```

We will draw reference variables and arrays that way – as arrows pointing to other large chunks of memory. But remember that, in reality, the reference variable stores a label – a memory adddress – for some cell in memory. Upon declaration, the reference would hold `null`, which is an address in the machine at which *nothing* will ever be stored. (Generally, the `null` address is either the lowest address in the machine – i.e. `a0` – or the highest address in the machine. We will assume it is `a0` in CS125.) So the real picture after declaration is something like this:

```
              --------
              |      |
scores - -    |  a0  |
              |______|
```

And then after allocating the object and assigning the reference variable to point to the object, the picture would be something like this:

```
              --------
              |      |
scores - -    | a200 |
              |______|



                      a200
                      -------------------------------
                      |    |    |    |    |    |    |
                      |    |    |    |    |    |    |
                      |____|____|____|____|____|____|
                         0    1    2    3    4    5
```

Summary of data storage

- Type of data:
  - Primitive-type variable – primitive type
  - Reference variable – non-primitive type
  - Object – non-primitive type
- Does the item have a name?
  - Primitive-type variable – yes, all variables have names
  - Reference variable – yes, all variables have names
  - Object – no, there is no name; you cannot use objects directly, but instead must have a reference variable pointing to the object and then work through the reference variable
- How is the item created?
  - Primitive-type variable – variable declaration
  - Reference variable – variable declaration
  - Object – you use `new` to dynamically allocate memory from the heap
- What is the scope of the item?
  - Primitive-type variable – the block or method it was declared in
  - Reference variable – the block or method it was declared in
  - Object – as long as a reference points to the object, it still exists. Once no reference points to the object, the object is gone. So there is no particular scope to an object – you can simply use it for as long as it is still accessible from your program in some way.
- What can the item store?
  - Primitive-type variable – values of primitive types
  - Reference variable – memory addresses
  - Object – depends on the type of the object (more on that later, as we study classes; for now, the only object we know is arrays and they store many items of the same type)

One last important piece of info

Since we are manipulating objects through references, it is important to realize that assignment doesn't work the same way.

```
int[] A = new int[10];
int[] B = new int[10];
    ... // assume we have some code that
        // writes values into A's array.
B = A;
```

That last line will *not* copy the values of the array that `A` refers to, into the cells of the array that `B` refers to. Instead, it will change the reference variable `B` to point to the same array that the reference variable `A` points to. That is, the assignment statement above copies the memory location label stored in the reference variable `A`, into the reference variable `B`.

```
// BEFORE

                ________________
  A ----------> | first array    |
                |________________|


                ________________
  B ----------> | second array   |
                |________________|



// AFTER

                ________________
  A ----------> | first array    |
             __ |________________|
              /|
             /  ________________
  B --------/   | second array   |
                |________________|
```

and then the array that the reference variable `B` *used to* point to will be lost and unaccessible, since we no longer have any references to it.

That is, dealing with objects is quite different than dealing with variables of primitive types. Assignment of references is *not* assignment of objects, it is just an assignment of a memory location into a reference. Thus, assignment of one reference to another does *not* automatically copy an object – it simply writes a memory location into a reference. When `B` is declared to be an integer array reference, and we write the line:

```
B = new int[10];
```

the expression involving `new` sends back the location of the array and it is written into the reference variable `B`. Likewise, when we write

```
B = A;
```

the reference variable A stored a memory location and that location is then written into B. It is important to realize this, as it means when we deal with objects in general we cannot copy one into another by assigning one's reference to the other's reference. We need to explicity copy the object somehow. In the case of the array, that would be done through a loop:

```
for (int i = 0; i < A.length; i++)
   B[i] = A[i];
```

That example assumes that the array pointed to by A and the array pointed to by B are the same size. If they were not, then you'd have to write slightly more complicated code to adjust for that possibility. The following code:

```
int[] A = new int[10];
int[] B = new int[8];
B = A;
System.out.println(B.length);
```

will print 10 since even though B starts out pointing to an array of length 8, we then reassign B to point to an array of length 10.

If two reference variables are pointing to the same object, then it doesn't matter which reference you use to access the object. The following code:

```
int[] A = new int[10];
int[] B = A;
B[0] = 19;
A[0] = 3;
System.out.println(B[0]);
```

will print the value 3, because A[0] and B[0] are the exact same array cell, since A and B point to the same array.

```
                  ------------------
  A ----------> |   | .  .  . |  |
             -- |___|_________|__|
              /|  0   .  .  .  9
             /
  B --------/
```

Therefore, when we assign A[0], we are assigning B[0] as well – replacing the 19 with a 3.