# CS125 : Introduction to Computer Science

## Lecture Notes #14
## Objects and Methods

## Lecture 14 : Objects and Methods

Passing objects to methods via reference variables

What we will do now is to look at a program whose methods have references to arrays as parameters and as return values.

```
public class Lecture14
{
   public static void main(String[] args)
   {
      int numberOfScores;
      System.out.print("How many scores are there?: ");
      numberOfScores = Keyboard.readInt();
      int[] scores;
      scores = new int[numberOfScores];
      readData(scores);
      printData(scores);
      System.out.println(scores[2]);
      badInit(scores);
      System.out.println(scores[2]);
      System.out.println(scores.length);
      scores = getArray(4);
      System.out.println(scores.length);
   }

   public static void readData(int[] arr)
   {
      for (int i = 0; i < arr.length; i++)
      {
         System.out.print("Enter score for index #" + i + ": ");
         arr[i] = Keyboard.readInt();
      }
   }

   public static void printData(int[] arr)
   {
      for (int i = 0; i < arr.length; i++)
         System.out.println("Score at index #" + i + " is " + arr[i] + ".");
   }
```

```
   public static void badInit(int[] arr)
   {
      arr = new int[3];
      for (int i = 0; i < arr.length; i++)
         arr[i] = -1;
   }

   public static int[] getArray(int n)
   {
      int[] temp;
      temp = new int[n];
      return temp;
   }

}  // end of the class Lecture14
```

Note the statements:

```
readData(scores);
printData(scores);
badInit(scores);
```

in `main`. Each of those statements is a method call, to a method that appears later in the program. For each of those method calls, the reference variable `scores` is an argument for the method. To use a reference variable as an argument, we simply need to put the reference variable name in the method parenthesis, like we would do for variables of primitive types. Of course, any variable name is an expression, one that evaluates to the value stored in the variable. Reference variables are no different; as we've already discussed a little bit, a reference variable name, when used as an expression, evaluates to the *memory address* stored in the reference variable. So, in each of the three method calls above, the value we are sending to the method as an argument, is a memory address – specifically, the memory address stored in the reference variable `scores`.
Note also that in the method signatures for `readData`, `printData`, and `badInit`, the parameter that matches the reference variable argument has `int[]` as the type, just like the variable `scores` does when we first declare that array reference variable in main:

```
public static void main(String[] args)
{
     ...
     int[] scores;
     ...
}

public static void readData(int[] arr)
             .
             .
             .
public static void printData(int[] arr)
             .
             .
             .
public static void badInit(int[] arr)
```

Since the argument is of type `int[]` – a type that holds memory addresses as values – the parameter is also of type `int[]`. The argument type and the parameter type match – as they should – and the value that gets copied into the parameter, is a value of type `int[]`, i.e. a memory address.

So, with respect to type matching, the parameter-passing rules apply for reference variables just as they did for primitive type variables:

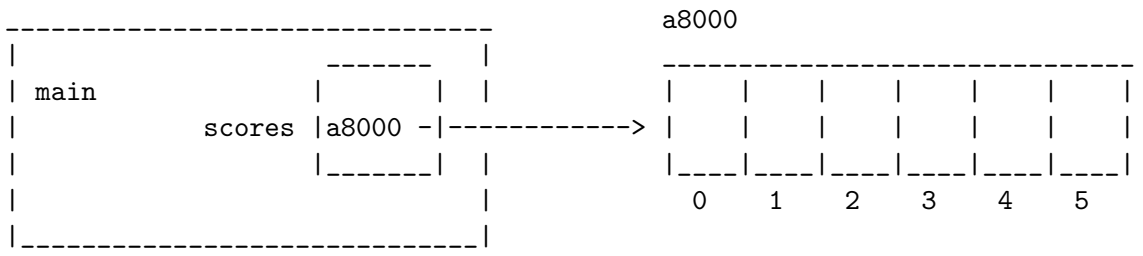
- the parameter type and the argument type must match, whether that common type is `int` or `double` or `int[]`. The fact that now our type is a non-primitive type doesn't change that.
- The parameter "type and variable name" pair will look just like a variable declaration of that type. Our parameters previously were pairs such as `int x` or `boolean notDone` or `char choice`, but now that our parameter type can be `int[]`, we just have a pair such as `int[] arr` – still a "type and variable name" pair, even though the type is now a non-primitive type.
- The argument will still be some expression that evaluates to a value of the needed type. In our example program, the expression we use in the method call is simply a variable name, but that variable name will evaluate to some value of the appropriate type – a memory address, in the case of the methods calls we are discussing – just like any other variable evaluates to the value the variable holds.

In our code above, we send an expression of type `int[]` as an argument. That expression is evaluated to produce a "value of type `int[]`" (the value of any reference variable type is a memory address), and that value is sent to the method and copied into the parameter variable of type `int[]`. This is no different than our earlier method examples, when we would, say, send an expression of type `boolean` as an argument, and that expression would be evaluated to produce a value of type `boolean` and that value would be sent to the method and stored in the parameter variable of type `boolean`.

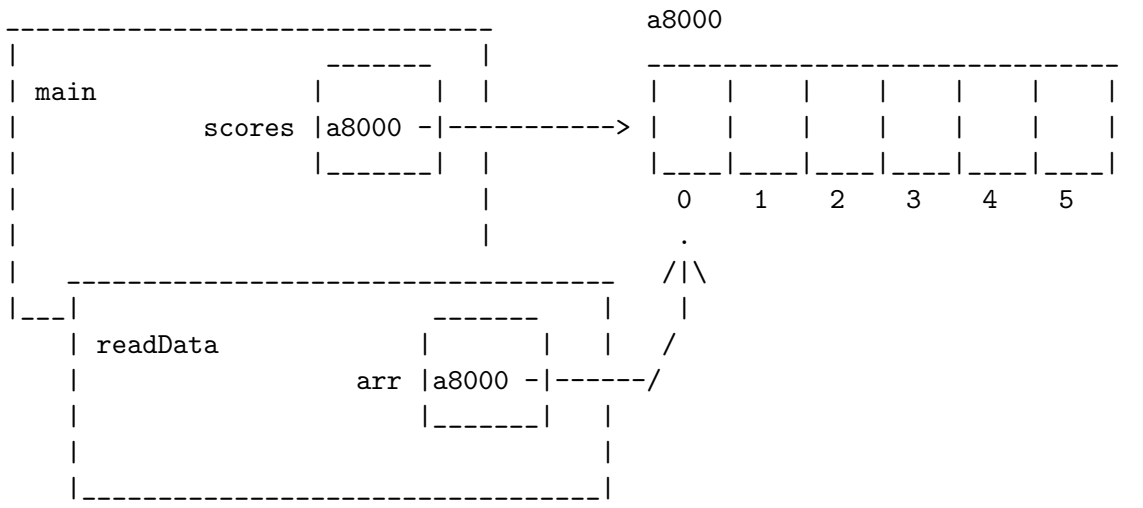
That is the very important point here – fundamentally, parameters of type `int[]` or `double[]` or any other non-primitive type work exactly the same way as parameters of primitive types, namely, that the value of the method call's argument is *copied* into the method's parameter.

For example, let's assume that the first integer inputted by the program from the user was a `6`, and thus the array that gets allocated in `main` is of size `6`. If the array we allocated in `main` was

placed at memory address `a8000` by the sytem, then `scores`, the variable of type `int[]`, would hold the memory address `a8000`. (The array object could have been located anywhere in memory; we chose `a8000` just to pick *something*, for our example, but the object might have been located elsewhere instead.)




So, `scores` is a variable that holds the memory address `a8000`. And that means that when we have `scores` as an argument to a method, the expression `scores` (a single variable name, but an expression nonetheless) gets evaluated to the value `a8000`, and that value is sent to the method, and is written into the parameter, and so now the parameter of the method `readData` also holds the memory address `a8000`. This is no different than when you send an `int` variable as an argument to an `int` parameter, and afterwards the `int` variable and the `int` parameter hold copies of the same `int` value. Above, we sent an `int[]` variable as an argument to an `int[]` parameter, and so both the `int[]` variable and the `int[]` parameter hold copies of the same `int[]` value – i.e. the same memory address:

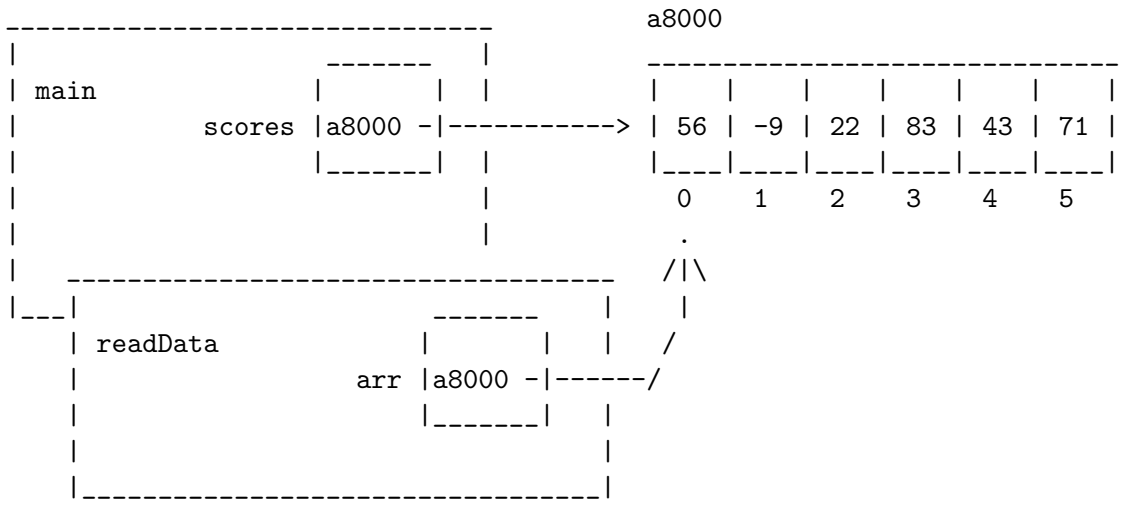



Now, what prevents us from accessing the array through `arr` in exactly the same way we access it through `scores`? The answer is, nothing! Both of those variables – the one in `main` and the one in `readData` – are variables of type `int[]` which hold the memory address of the same dynamically-allocated array. The only difference between them is that one of them is in the scope of `main` and one is in the scope of `readData`. But they hold the same memory address, and thus are pointing to the same array object, and thus conceptually they are completely interchangable.

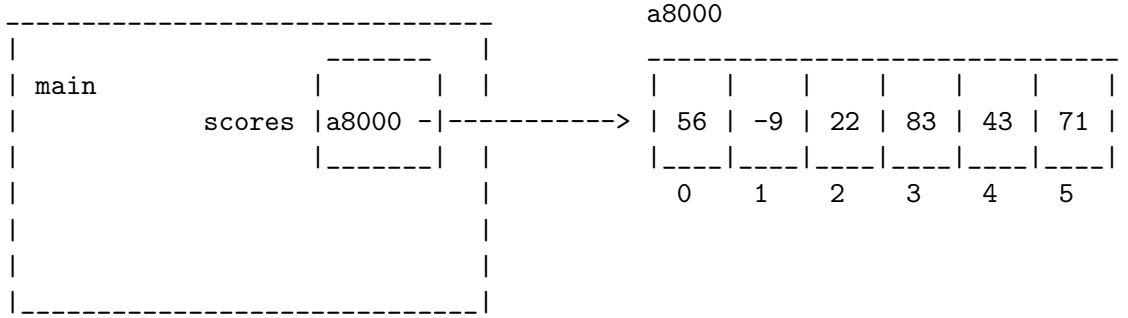
The `readData` code will write into the array object shown above just as easily as if we had put that code in `main` and used `scores` instead of `arr` as the reference variable that array access depended on. This illustrates a very important quality of objects and reference variables: *It doesn't matter what reference variable you access an object through.* There could be fifty different reference

variables to the same object inside your program; with the exception of the names and/or scopes of the variables being different from each other, all of those reference variables are exactly equivalent. Accessing the object – for reading *or* writing – works the same way regardless of which of the fifty reference variables we use, and we will be accessing the same object regardless of which of the fifty reference variables we use.

So, in `readData`, the expression `arr.length` will evaluate to `6`, and thus the body of the `for`-loop will run six times. If the user-inputted values for those six `Keyboard.readInt()` calls are, respectively, `56`, `-9`, `22`, `83`, `43`, and `71`, then afterwards, our picture will look like this:




When we write `56` into the array cell `arr[0]`, we are also writing it into the array cell `scores[0]`, because *they are the exact same array cell!!!*. So, when we return from `readData`, and the parameter reference variable `arr` goes away, the local reference variable `scores` is back in scope, and the actual array object is still accessible through `scores`, and that value `56` is still sitting in the cell with index `0`. That doesn't change. All that happens when we return from `readData` is that our second reference variable, `arr`, goes out of scope. Everything else stays the same:



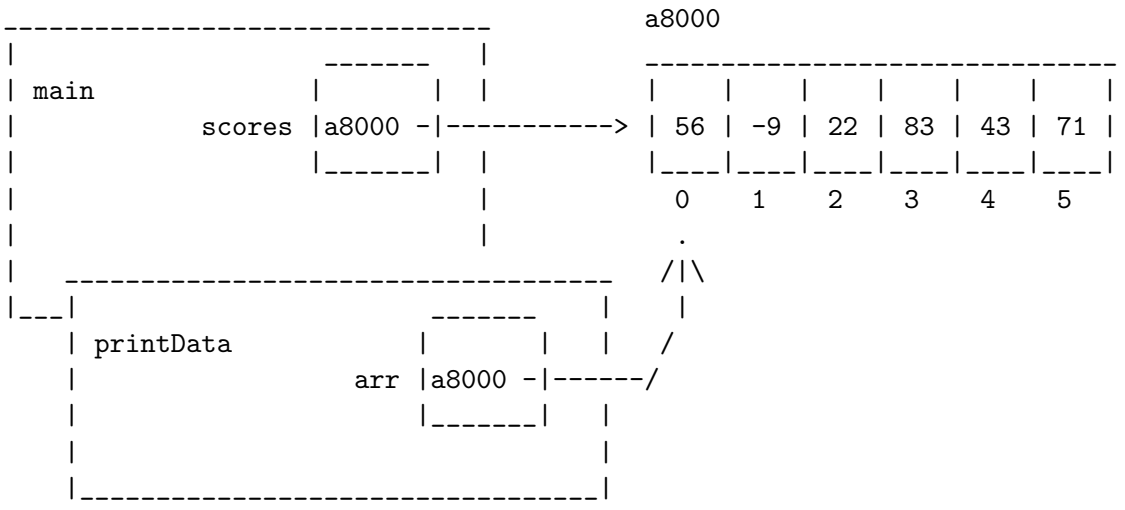

In short, we were able to change our array object from within the method `readData(...)`, even though the array object was created in `main()`. Since objects are not bound by scope rules, we can read and write the same object from any method that has a reference to that object.

Note that we never actually have an object as an argument to a method. Instead of passing in the object itself to a method, we pass in a reference to it. Reference variables follow the same rules as variables of the primitive types – the value the argument evaluates to is *copied* into the

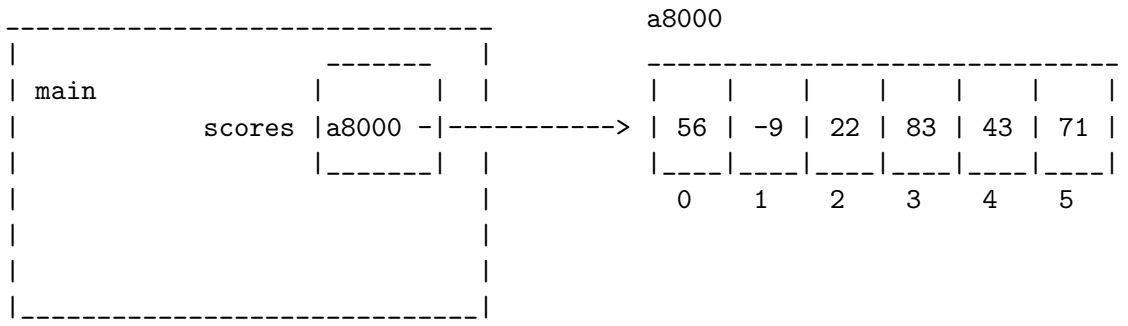
method parameter. If that argument was a variable, that means there are now two copies of the data – one in the argument variable, and one in the parameter. That was how things worked for variables/parameters of type `int`, it's how it worked if they were of type `boolean`, and it's how it works if they are of type `int[]`. The only difference is the type of value that is copied – a value of type `int` in the first case, a value of type `boolean` in the second case, and a memory address in the third case. This is called *passing by value* because from the method call, we were sending a value to be copied into a parameter.

But, we are passing around the *address* of the object from argument to parameter, instead of passing the actual object itself. We never make a copy of the object; we only make a copy of its address, and then use that copy of the object's address to access the object from the new method (that new method being `readData(...)`, in our above example). We can then refer to the original object from our method – and even change it.

So, let's look at another example. When we call the `printData(...)` method, we again pass the value of `scores` – `a8000` in our example – as an argument, to a reference variable parameter:

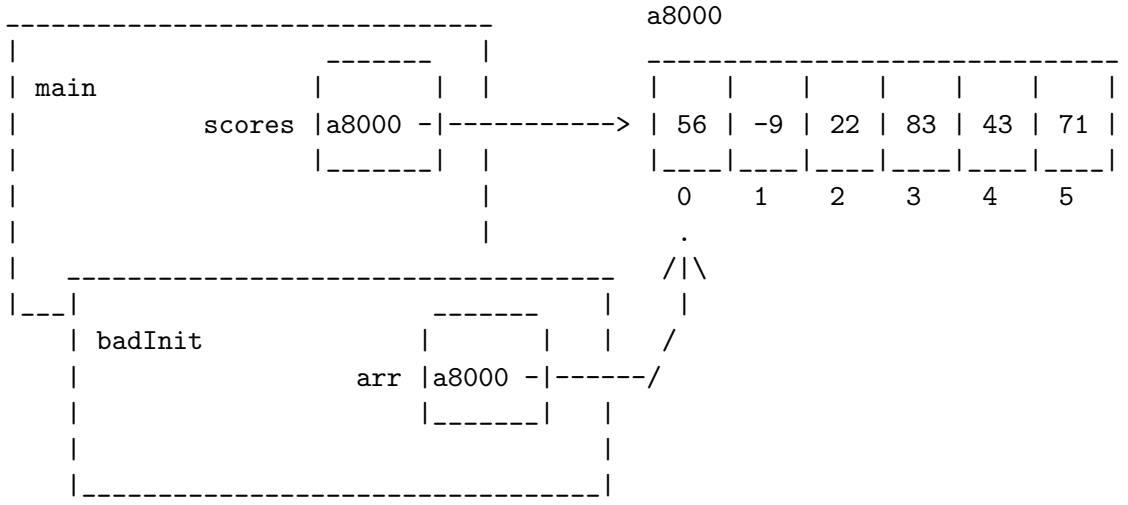



Now, as the `printData(...)` method successively prints out `arr[0]`, `arr[1]`, and so on, the method is actually printing out `scores[0]`, `scores[1]`, and so on – because `arr[0]` and `scores[0]` are the same array cell, `arr[1]` and `scores[1]` are the same array cell, and so on. We eventually return back to `main()`, having printed the entire array pointed to by scores, via the use of a completely different method (`printData(...)`), rather than via code written inside the `main()` method.




Note that altering the object from a different method, as we did with `readData(...)`, is something we could not do with variables of primitive types that were arguments in the method call, nor can we do it with reference variables that are arguments in the method call. Again, this is because arguments in Java are *passed by value*. In the `Add3` method in Lecture Notes #11, changing `x`, `y`, and `z` did not change `a`, `b`, or `c`, since we only sent the *values* of `a`, `b`, and `c` to the `Add3` method – we did not send the actual variables themselves. When you call a method, the arguments are expressions that evaluate to values, and you simply copy those values into the method parameters. Likewise here, changing the value of `arr` – that value being the memory address stored inside `arr` – will not change the value held in `scores`.

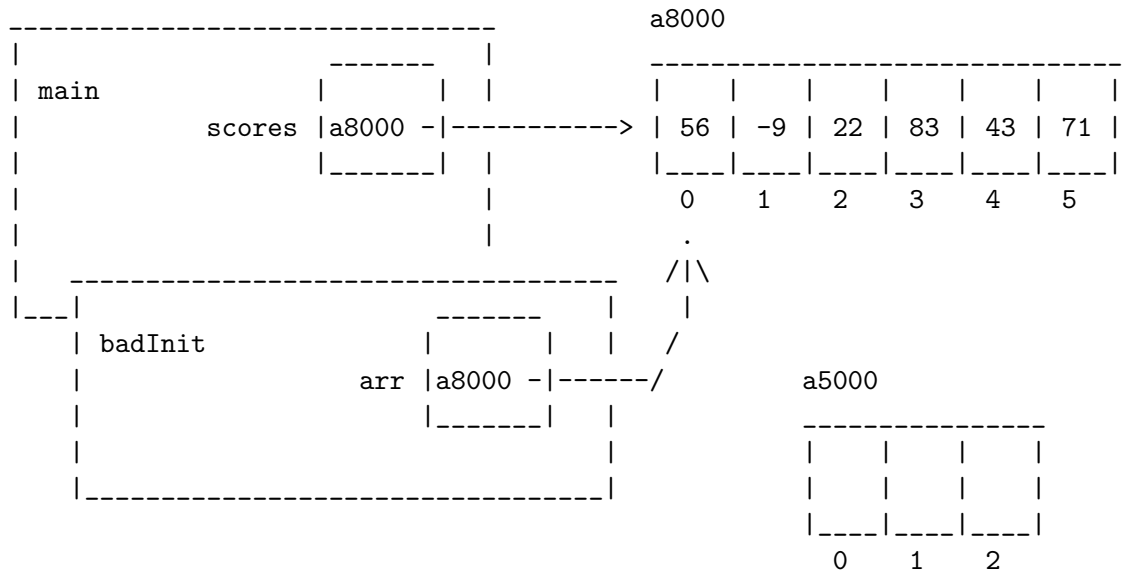
As an example, let's consider the three lines in our `main()` that appear after the call to `printData(...)`. When we print out `scores[2]`, we will print `22` to the screen, since `22` is the value stored at cell `2` of the array pointed to by `scores`. Next, we call the method `badInit(...)`, and end up with the following picture:




So far, this is not very different from the calls to `readData(...)` and `printData(...)`. What *is* different is what happens inside the `badInit(...)` method. The first line of the method definition is:

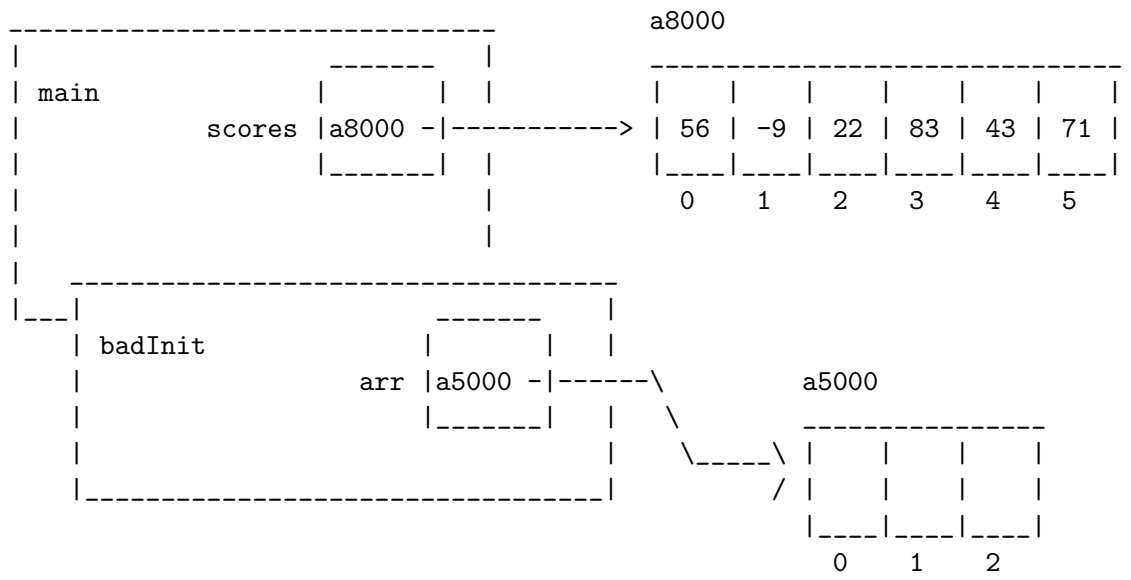
```
arr = new int[3];
```

As we have previously seen, a statement like that will, (1) allocate a new integer array object of size `3`, and (2) write the location of that array into the variable `arr`. So, as the expression `new int[3]` gets evaluated, we have a picture like this:




(The picture assumes that the new array object was stored at memory address `a5000`; it could have been stored at plenty of other different places as well but we just picked one for the example.)

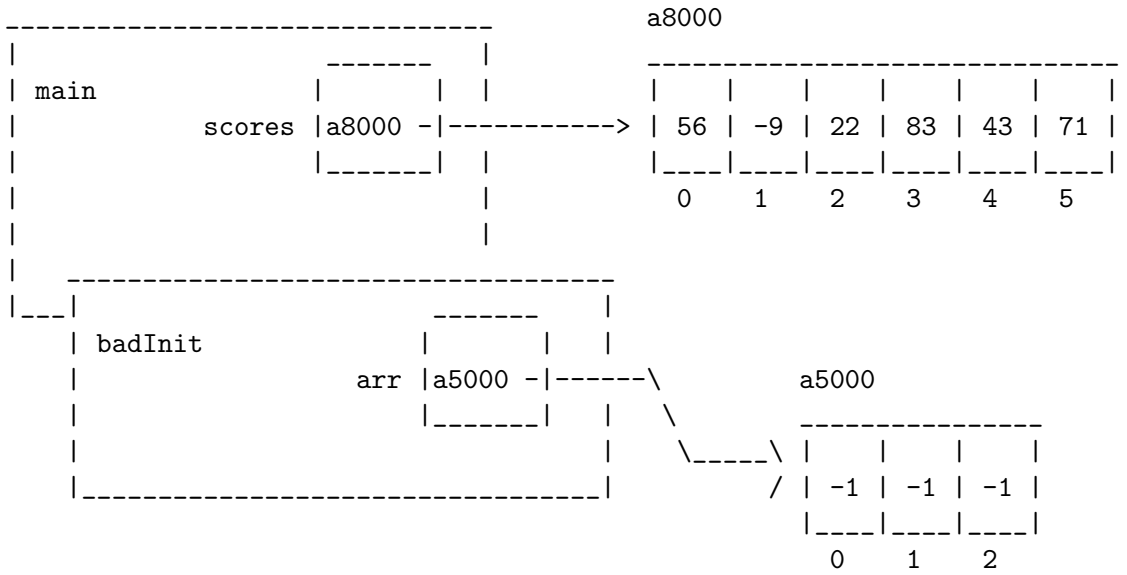
The array at `a5000` is allocated as part of the evaluation of the expression `new int[3]`. Once the array has been allocated, the expression evaluates to the address of the array – `a5000` – and then that address is written into `arr` by the assignment statement, giving the following picture once the assignment statement is complete:



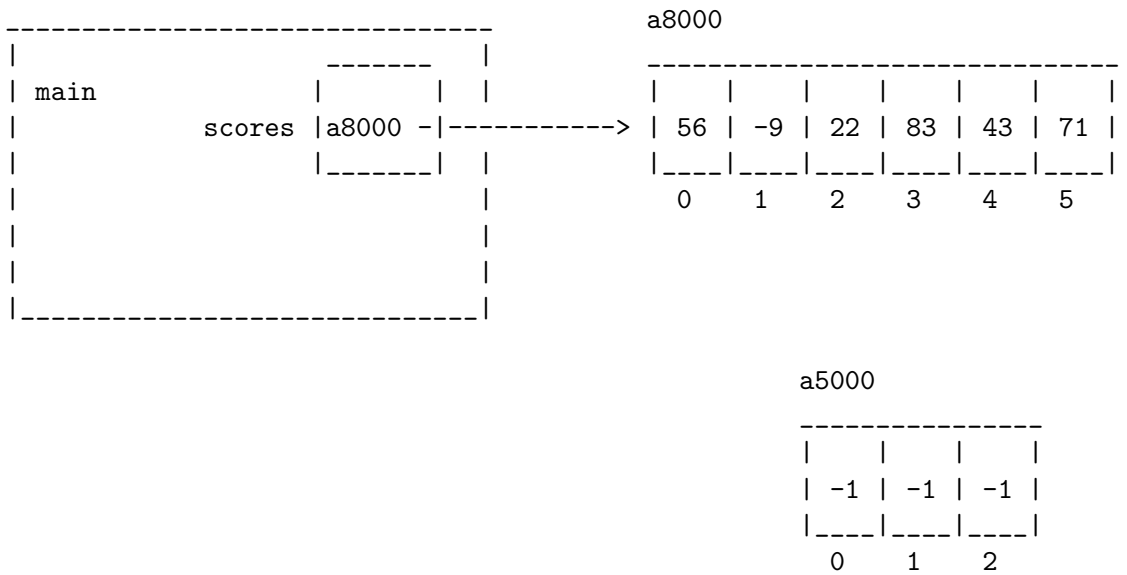

We have changed the value of `arr` by pointing it to a *different* object – but `scores` still holds the *same* address and thus still points to the *same* object. We can't change `scores` from `badInit` because `scores` is a local variable, and local variables are bound by scope. There is no way from within `badInit(...)` or any other non-`main()` method, to reassign a variable that is local to `main()`. This is just like in Lecture Notes #11, when we could not change the value of `a` in `main` by messing with the value of `x` in `Add3`.

The expression `arr.length` later in `badInit(...)` will now evaluate to `3`, since the array object that the reference variable `arr` points to, is of size `3`. So, in that `for`-loop inside `badInit(...)`,

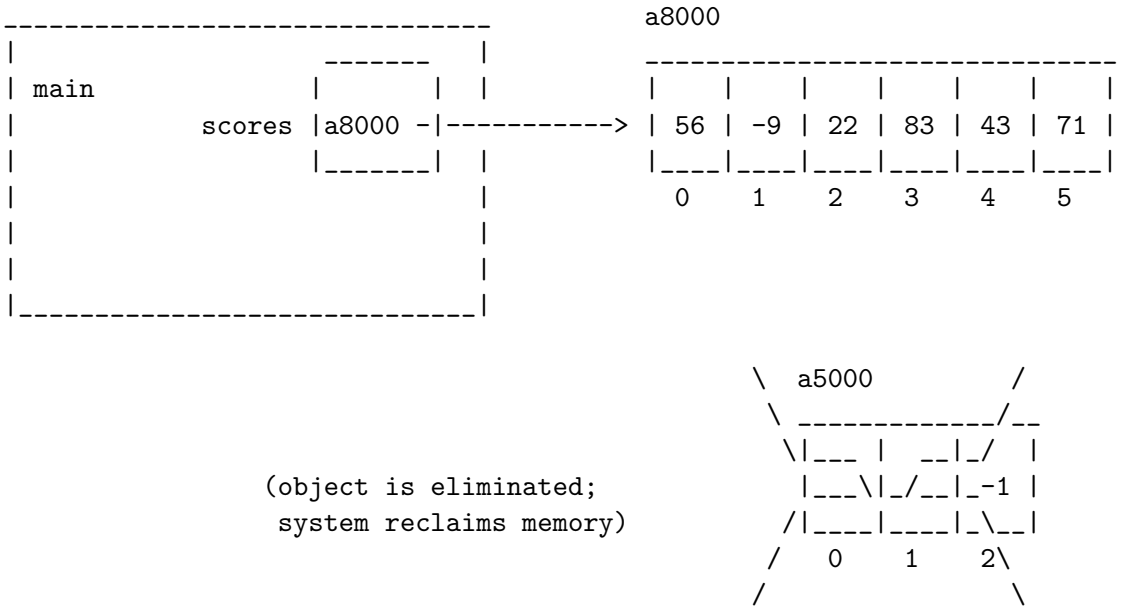
the body of the loop will run three times, and thus three assignments will be run – one for each of the three cells in the array pointed to by `arr`:



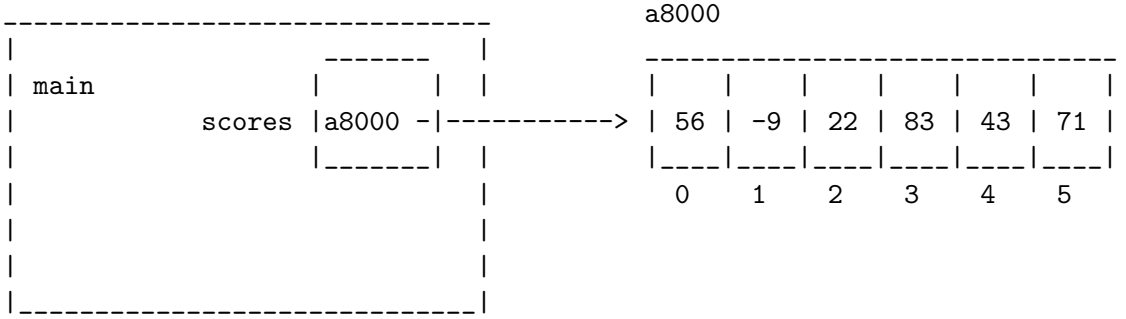

Now, with the `for`-loop complete, we can return from the method `badInit(...)` back to `main()`. When we return from a method, the local variables and parameter variables of that method go out of scope. Thus, the variable `arr` in `badInit(...)` goes out of scope, and thus there are no more references to the array at `a5000`:




And, since there are no longer any references in the program to the array object at `a5000`, the system will reclaim that memory, since your program is no longer using it:




So now we are back in `main()`. The variable `scores` still holds the address `a8000`, there is still an array of size `6` at the address `a8000`, and that array still holds the six values we inputted earlier. The first printing of `scores.length` (the third-to-last statement inside the definition of `main()`) will print the value `6` to the screen, just as it would have done at any earlier point in `main()` after the array was allocated.

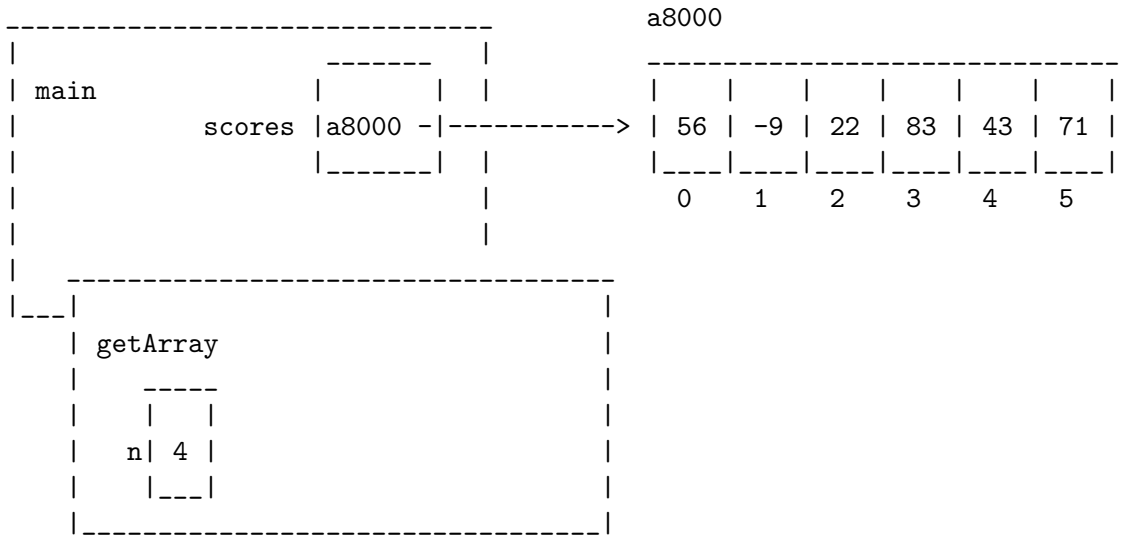



The only thing `main()` has access to that we could have changed in `badInit(...)`, is the non-local data of `main()` – specifically, the object at `a8000` to which `main()` has a reference. Since we did not change that object within `badInit(...)` – we reassigned `arr` before making changes to the cells of the array `arr` pointed to – we therefore have not changed anything within `badInit(...)` that `main()` had access to, and so there are no permanent effects of the work we did in `badInit(...)`.

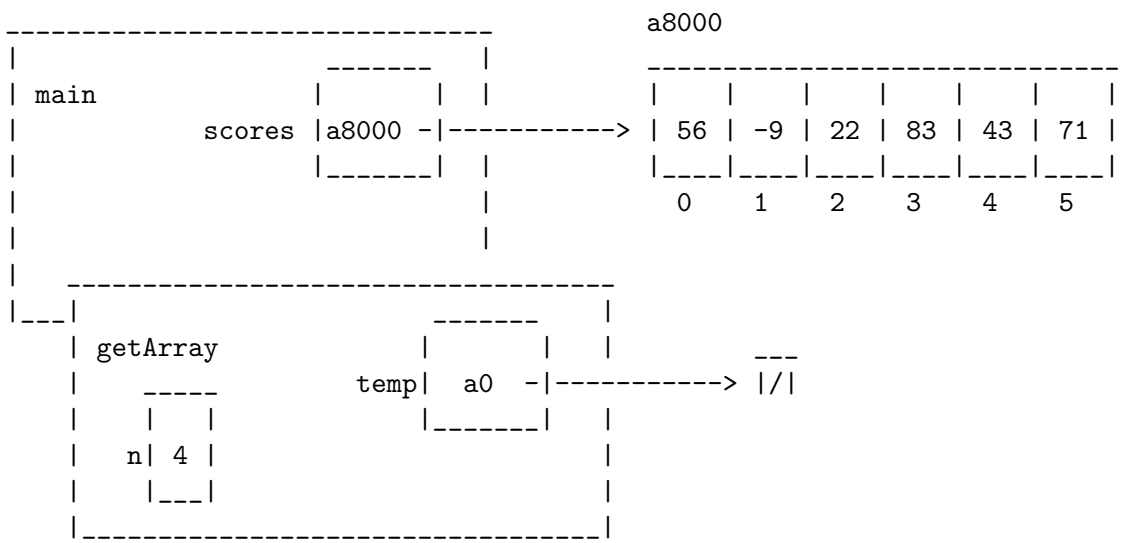
We will now inspect the next statement in `main()`:

```
scores = getArray(4);
```

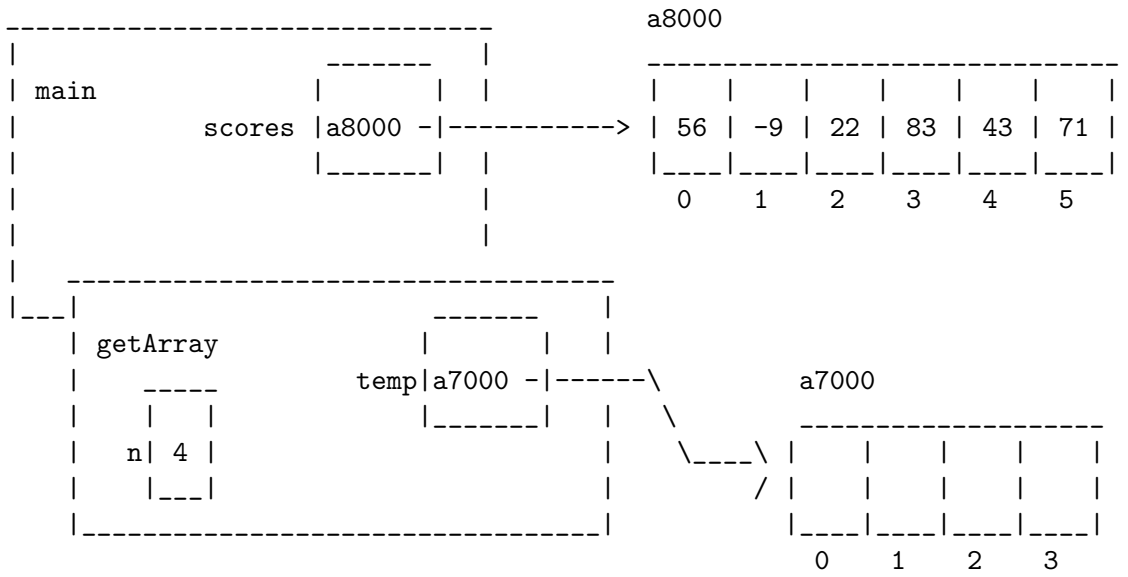
As far as passing parameters goes, this method call is a bit more straightforward than the previous three, since the one parameter is an integer, and we've seen parameters of primitive types before:




Within the method, we first declare a local reference variable of type `int[]`. As with any reference variable declared via a variable declaration statement, this reference variable is automatically initialized to `null`:




Then the next statement will allocate an array object of size `4`, and write the memory address of that array object into the reference variable `temp`. (We'll assume that the array object is allocated at memory address `a7000`, just to pick something for our example.)




Finally, the last statement of the method:

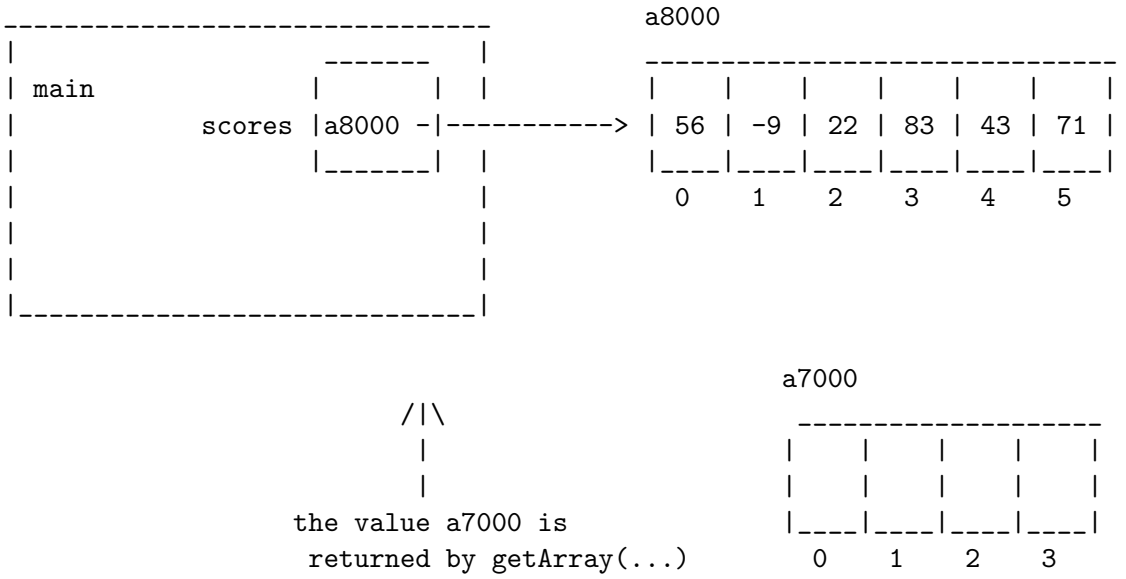
```
return temp;
```

is the interesting part. Remember that in any return statement of the form:

```
return expr;
```

we evaluate the expression, and return that value as our return value. As we have already discussed, reference variables evaluate to the memory addresses they hold. So, our expression `temp` in our return statement, evaluates to `a7000`, and that is what we return. This matches our return type; the return type is `int[]` and our return value is a value of type `int[]`, i.e. a memory address. (Remember, the value of any reference type, is a memory address.)

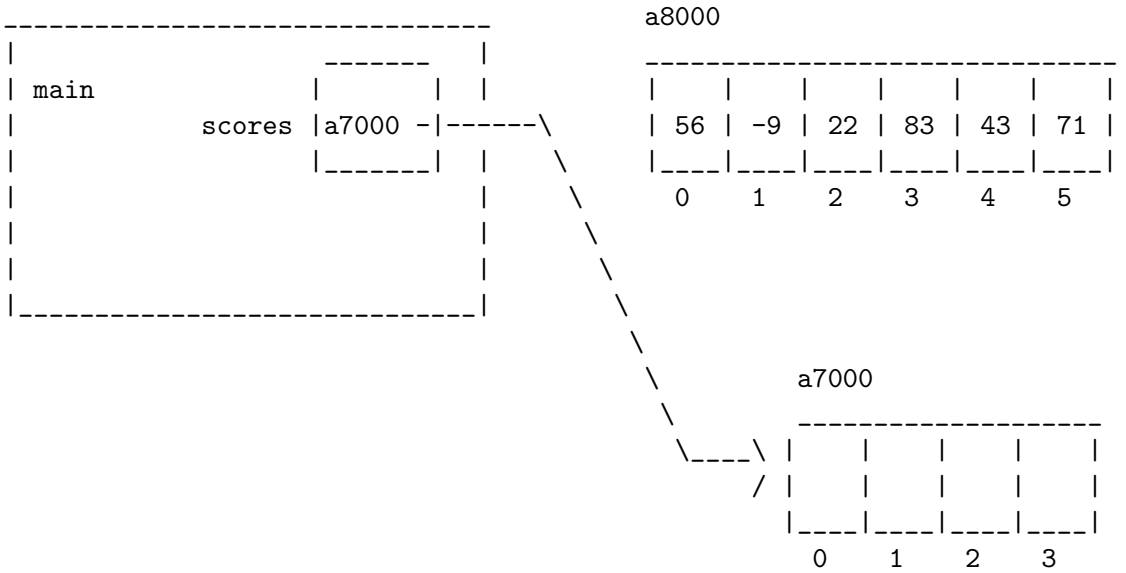
So, the method `getArray(...)` now ends, and thus `temp` and `n` go out of scope, and the method returns `a7000` as its return value.



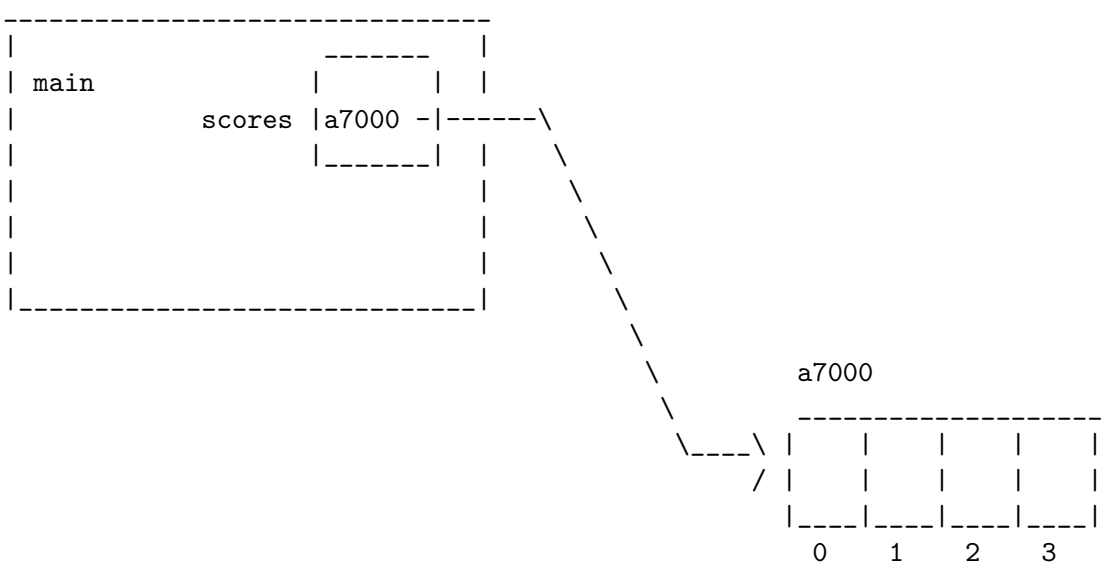

And thus the right-hand side of the statement:

```
scores = getArray(4);
```

evaluates to `a7000`, and so that value is what the assignment statement writes into the reference variable `scores`. That gives us the following picture:




That is, `scores` now points to the array object that was allocated in the `getArray(...)` method, and there are no longer any references holding `a8000`. That means the system can reclaim the memory being used for the array object at `a8000`, since no reference variable in the program is pointing to it anymore:




And that is what we are left with at that point in our `main()` method. The reference variable `scores` points to an array of size `4`, and so the final print statement of the `main()` method will print `4`, since `scores.length` will now evaluate to `4` (whereas before the `getArray(...)` call, it evaluated to `6`, since at that time, `scores` pointed to a completely different array object.

The general rule to remember is that *variables* – whether primitive-type variables or reference variables – follow the "pass by value" rule. Parameters of methods merely hold copies of the argument values, and are not tied in any way to the arguments themselves. So if a parameter is a reference variable, it holds a memory address. If we have a reference as an argument, we are sending a memory address value – NOT an object – to be copied into a reference variable parameter. If we return a reference variable, we are returning a memory address value, and so the method call expression evaluates to that memory address value (as in the `getArray(...)` call in our `main()`).