CS125 : Introduction to Computer Science

Lecture Notes #16
Classes, Reference Variables, and `null`

Lecture 16 : Classes, Reference Variables, and `null`

`Clock` references as parameters

Last time, we saw this example, where we stored our information for a clock, in a `Clock` object, and passed a reference to that `Clock` object to the `printClock(...)` method, rather than passing two `int` values and a `boolean` value to the `printClock(...)` method:

```
public class ClockTest
{
   public static void main(String[] args)
   {
      // declare reference variables
      Clock home;
      Clock office;

      // allocate object, assign address to reference variable,
      //    initialize instance variables of object
      home = new Clock();
      home.hour = 2;
      home.minutes = 15;
      home.AM = true;

      // allocate object, assign address to reference variable,
      //    initialize instance variables of object
      office = new Clock(); // object created
      office.hour = 7;
      office.minutes = 14;
      office.AM = false;

      // print the time on each of the two clocks
      printClock(home);
      printClock(office);
   }

   public static void printClock(Clock c)
   {
      // print variables for clock
      System.out.print("Time is " + c.hour + ":");
      if (c.minutes < 10)
         System.out.print("0");
      System.out.print(c.minutes + " ");
      if (c.AM == true)
         System.out.println("AM.");
      else    // AM == false
         System.out.println("PM.");
   }
}  // end of class
```

So far, the use of the `Clock` class:

```
public class Clock
{
   public int hour;
   public int minutes;
   public boolean AM;
}
```

has not allowed us to do anything we couldn't do before. In our first example yesterday, we called a method to print out the clock information, and that's exactly what we are doing now, as well.

In the first example last time, we could copy the values of our local variables, into the parameters of a new method, and so though we could not read `main()`'s clock-related variables from `printClock(...)`, we "effectively" had read-access to those variables anyway, since we could copy the data of those variables from `main()` to `printClock(...)` as part of the method call.

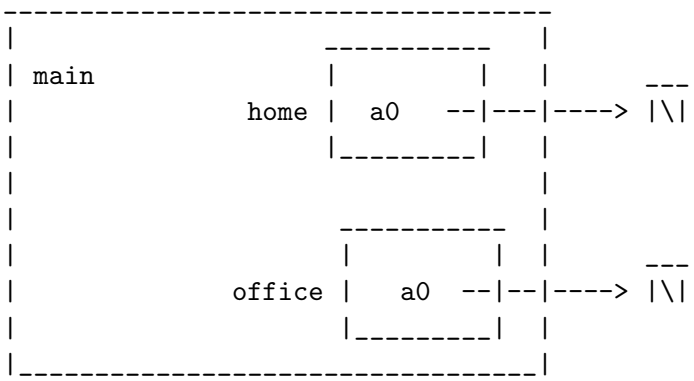
What we could not do, however, was *write* to the variables of `main()`, from `printClock(...)`. The method `printClock(...)` could store copies of the values, but if it changed its own copies, that had no effect on the original variables back in `main()`. So `main()` could send the values of local variables to other methods, but those other methods could not read – or write – the actual local variables of `main()`.

However, now that we've moved the clock-related data from being stored in local variables in `main()`, to being stored in instance variables in objects, we can now access those objects from other methods, just as we did in Lecture Notes #14 when we accessed an array created in `main()`, from other methods. In Lecture Notes #14, our non-`main()` methods were able to both read *and* write to the array object created in `main()`, since those other methods had their own references to that array object. Similarly, we can design methods like `printClock(...)`, to have `Clock` references as parameters – and then since a method like `printClock(...)` has its own reference to the `Clock` object created in `main()`, the method could read *and* write the instance variables of the `Clock` object created in `main()`, just as the methods in Lecture Notes #14 could read and write the array cells of the array object created in `main()`. Again, the important concept here is that objects are *not bound by the scope rules*. Any method can access any existing object, as long as the method has a reference to that object.

So, let's add in a new method to our `ClockTest` class – one that will assign to the variables of a `Clock` object. We will keep the `Clock` class the same for this entire notes packet; for now, we are only changing the code inside the `ClockTest` class, via the addition of a method `setTime(...)`:

```
public class ClockTest
{
   public static void main(String[] args)
   {
      // declare reference variables
      Clock home;
      Clock office;

      // allocate objects and assign addresses to the reference variables
      home = new Clock(); // object created
      office = new Clock(); // object created

      // set the time on each of the two clocks
      setTime(home, 2, 15, true);
      setTime(office, 7, 14, false);

      // print the time on each of the two clocks
      PrintClock(home);
      PrintClock(office);

   }  // end main

   public static void setTime(Clock c, int theHour, int theMinutes, boolean theAM)
   {
      c.hour = theHour;
      c.minutes = theMinutes;
      c.AM = theAM;
   }

   public static void PrintClock(Clock c)
   {
      // print variables for clock
      System.out.print("Time is " + c.hour + ":");
      if (c.minutes < 10)
         System.out.print("0");
      System.out.print(c.minutes + " ");
      if (c.AM == true)
         System.out.println("AM.");
      else    // AM == false
         System.out.println("PM.");
   }

}  // end of class
```

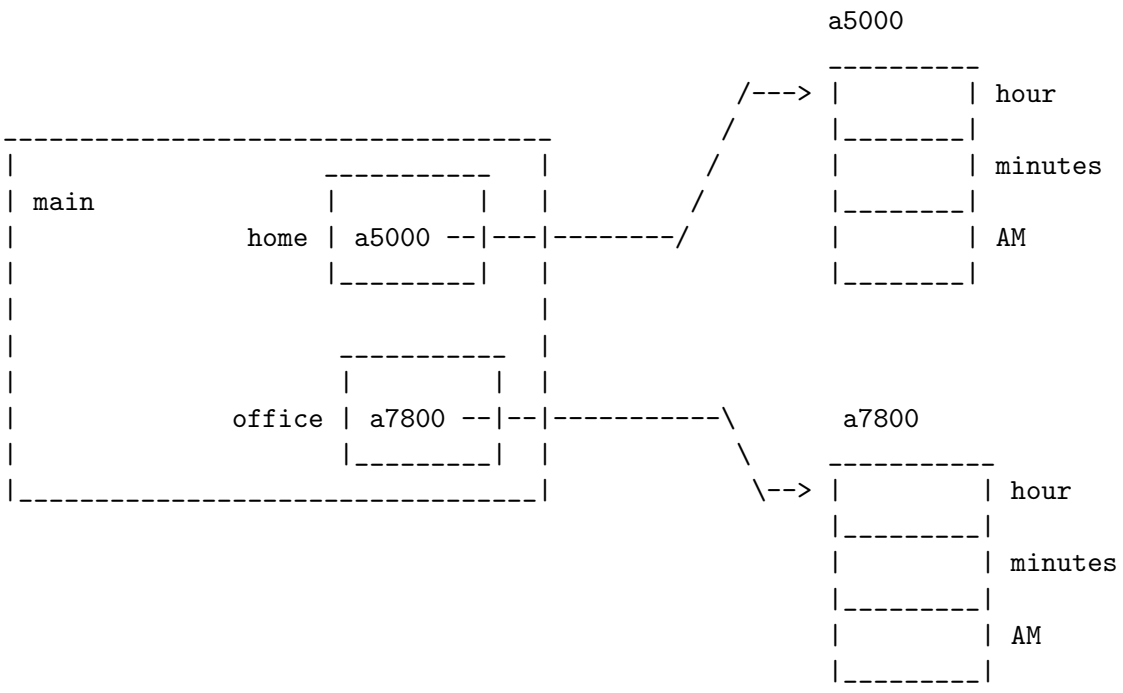
Let's trace though the beginning of this program, to see what happens. First of all, we declare the two reference variables in `main()`:



Then we allocate two objects. The first allocation occurs in the statement:

```
home = new Clock();
```

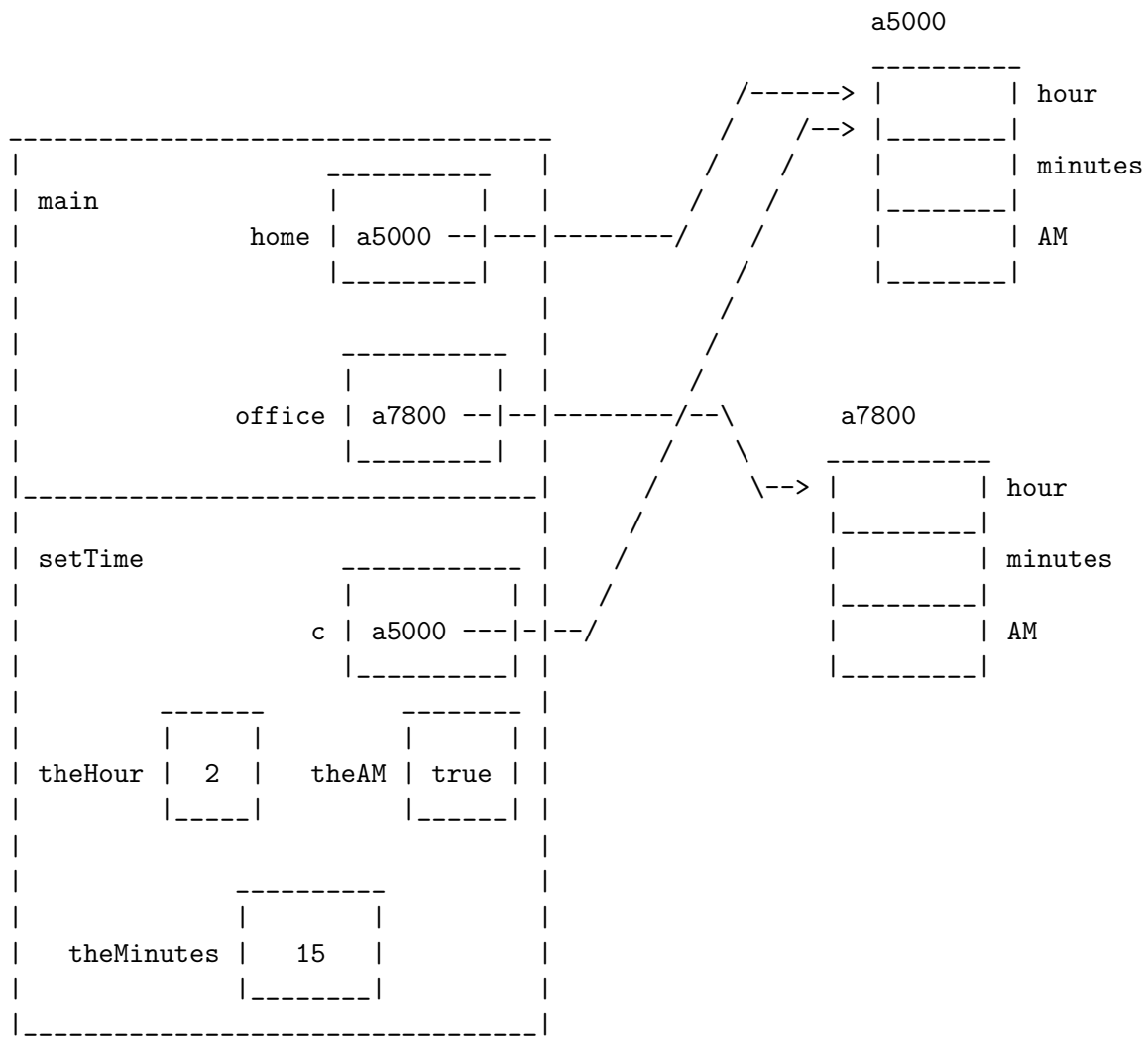
and thus the address of that object is written into the reference variable `home`. Likewise, the statement after that one will allocate a second object and store its address in the reference variable `office`. Let's assume these two objects are allocated at addresses `a5000` and `a7800`, respectively, just for the sake of assuming some addresses for our example. In real life, the objects could have been there, but could also have been almost anywhere else in memory instead.



Next, we have our first call to `setTime(...)`:

```
setTime(home, 2, 15, true);
```

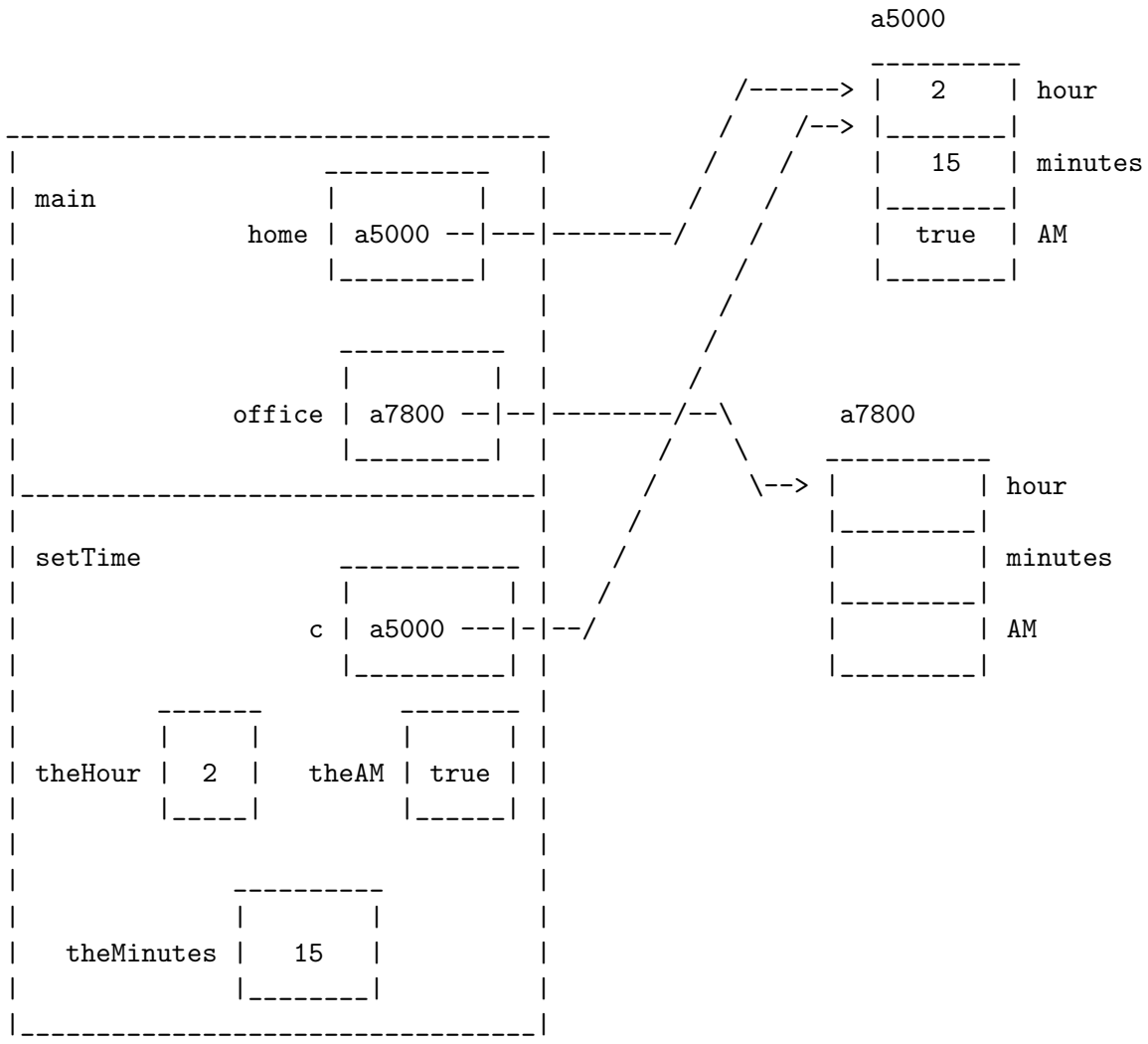
Each of the four arguments in that method call are evaluated, to obtain the four values `a5000`, `2`, `15`, and `true`. Those four values are then copied into the four parameters of the `setTime(...)` method, and control is transferred to the `setTime(...)` method:



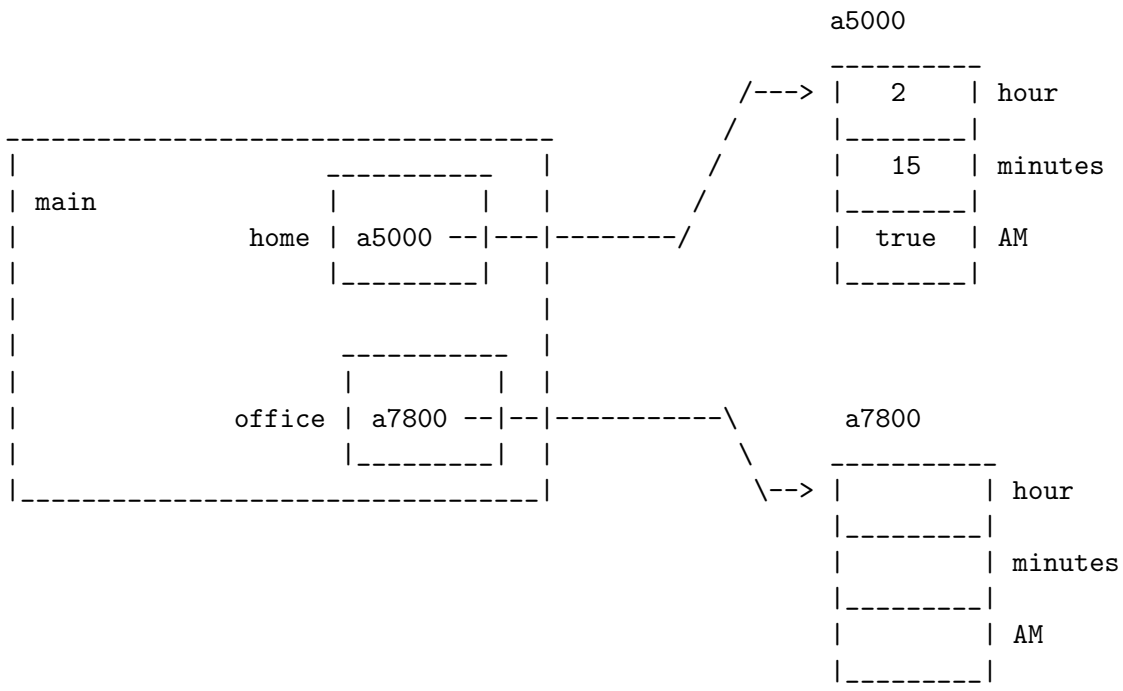
And now, when we run the three lines of the `setTime` method:

```
c.hour = theHour;
c.minutes = theMinutes;
c.AM = theAM;
```

we are writing into the instance variables of the object that the reference variable `c` points to, and thus, also writing into the instance variables of the object that `home` points to, since `c` and `home` point to the same object:



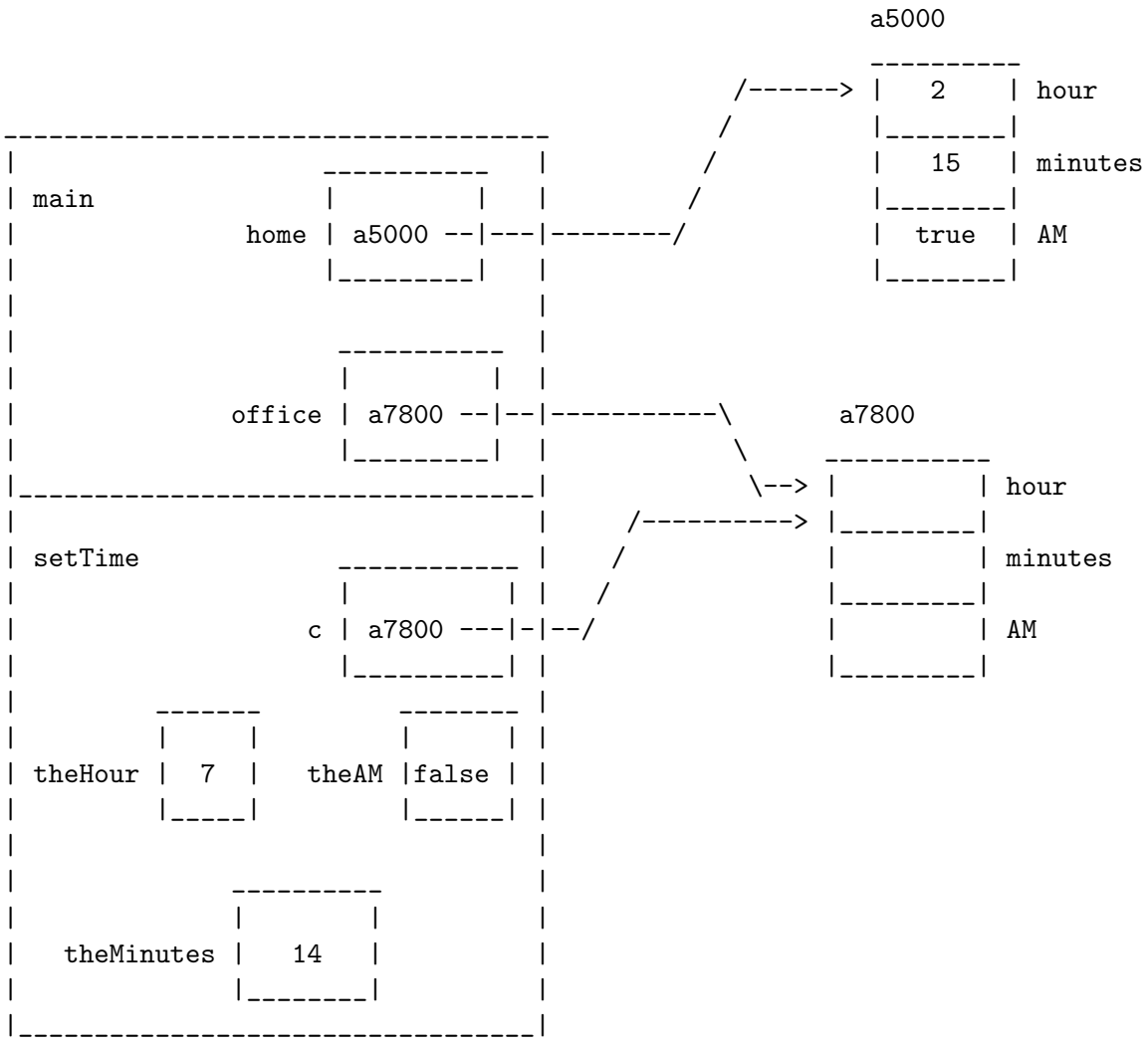
As a result, once we return from `setTime(...)` back to `main()`, the object `home` points to has been initialized:



The same would happen with the next line of code in `main()` – the second call to `setTime(...)`:

```
setTime(office, 7, 14, false);
```

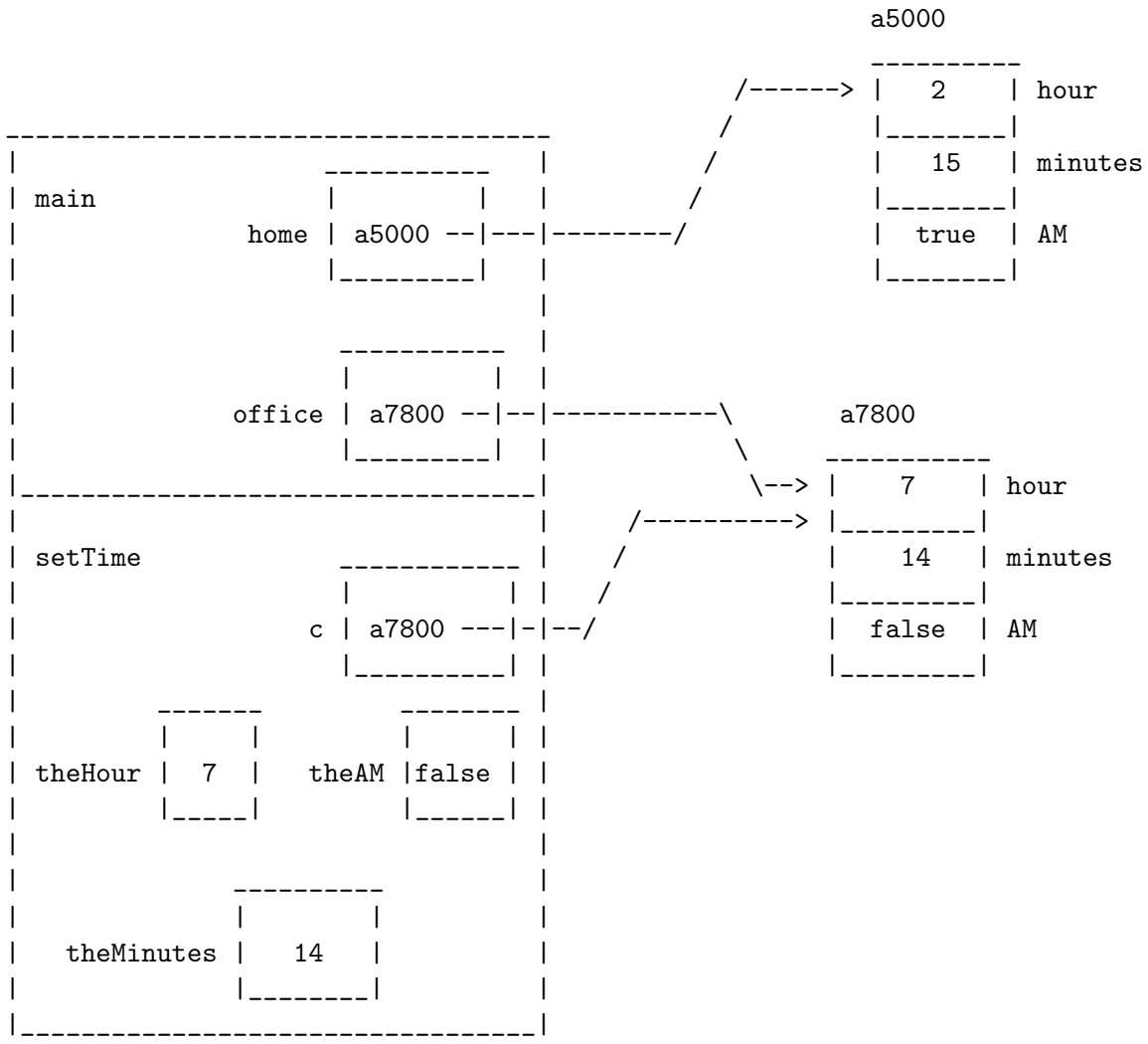
Each of the four arguments in that method call are evaluated, to obtain the four values `a7800`, `7`, `14`, and `false`. Those four values are then copied into the four parameters of the `setTime(...)` method, and control is transferred to the `setTime(...)` method:



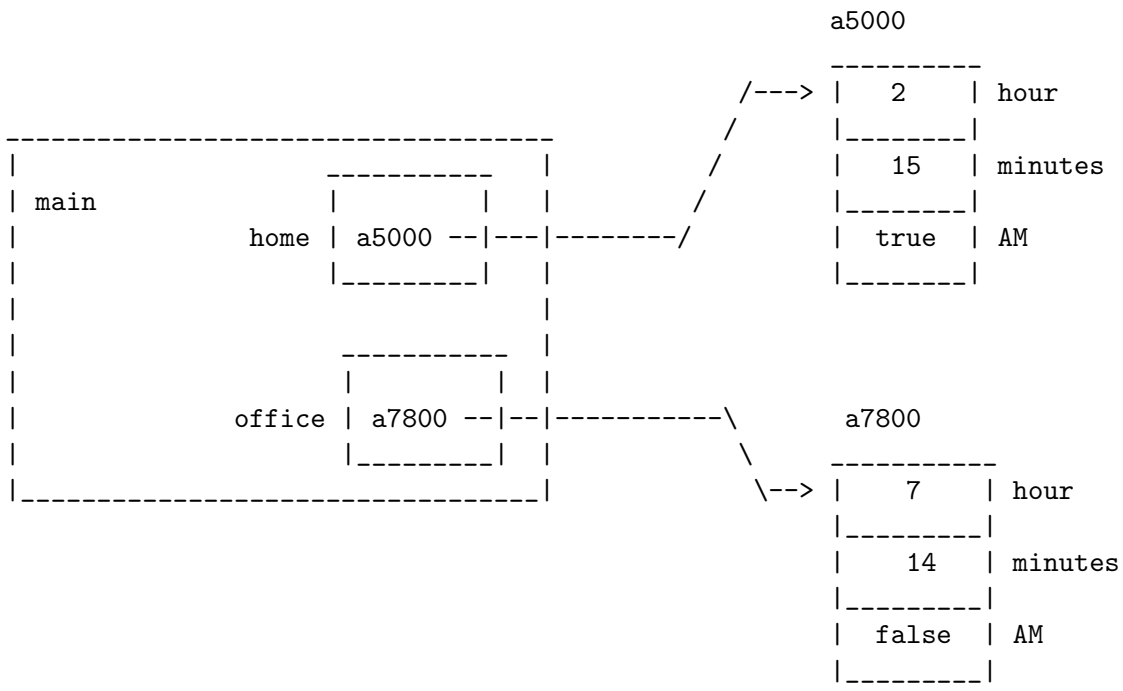
And now, when we run the three lines of the `setTime` method:

```
c.hour = theHour;
c.minutes = theMinutes;
c.AM = theAM;
```

we are writing into the instance variables of the object that the reference variable `c` points to, and thus, also writing into the instance variables of the object that `office` points to, since `c` and `office` now point to the same object. Our second method call has written a different value into the parameter `c`, and so by writing to the instance variables of what `c` *now* points to, we are writing to a different object than we were writing to when we called `setTime(...)` the first time:



And now, once we return from `setTime(...)` back to `main()`, the object `office` points to has also been initialized:



That is one of the big benefits of using objects – since we can access objects from any method as long as that method has a reference to the object, we can read or write the same object from many different methods. We could not have written a separate method to initialize local variables within `main()`, but we *can* write a separate method to initialize the instance variables of some object. So, by moving the variables from being local to `main()`, to being in an object, we gain the ability for other methods to read and write those variables freely, rather than being forced to (for example) put all the variable assignment code we ever want to run, into `main()`.

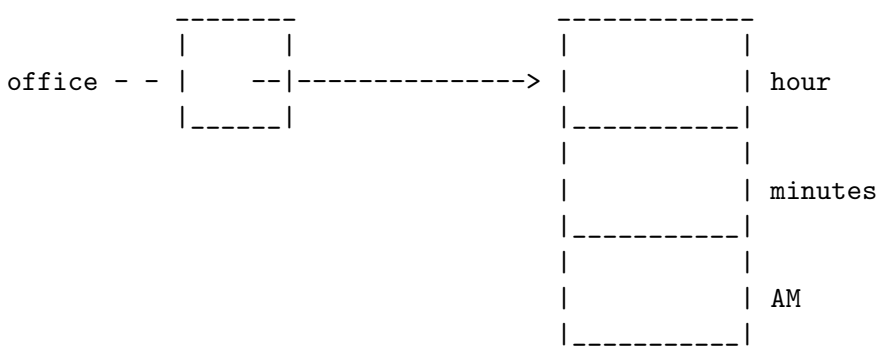
The `NullPointerException`

Consider the following code:

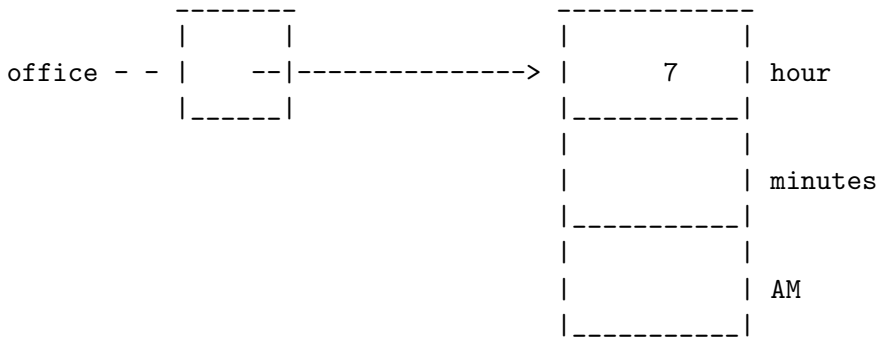
```
Clock office;
office = new Clock();
office.hour = 7;
```

We have discussed how this code works before. First we declared the reference variable (which is automatically initialized to `null`, like any other reference variable):

then we allocate a `Clock` object for the reference variable to point to:



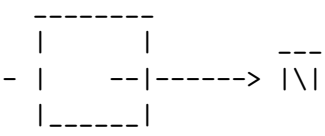
and finally, when we have the statement `office.hour = 7;` we are using the "dot syntax" on the variable `office` to "follow the arrow" from `office` to the object it points to, and once we have "arrived" at the object that `office` points to, we find an `hour` variable there, and can assign it the value `7`:



That's how things are *supposed* to work. But what if we leave out the allocation?

```
Clock office;
office.hour = 7;
```

In that case, we only declare the reference (which is initialized to `null`), so when we then use the "dot syntax" on the `office` variable, we "follow the arrow" for the following picture:



and when we arrive at the spot that `office` points to, there is *no object there of any kind!!!*. So the line `office.hour = 7;` can't do anything, since there's no `hour` variable there to assign. We "follow the arrow" to a location where there isn't an `hour` variable.

This is a problem – we have an assignment statement in our code that we cannot complete! So, the program will crash, and will announce to you that you have a `NullPointerException`. We have mentioned "exception error messages" before – we said that if you access an array with an illegal index, the program crashes, and notifies you that you had an `ArrayIndexOutOfBoundsException`. It's a similar situation here, in that the program will crash, and the error messages printed when it crashes will tell you of some illegal condition that was encountered when the program ran. In this case, you are trying to access an object at the `null` location, and since there *isn't* any object at the `null` location, you have a problem – you are asking for the impossible!. Hence, you have triggered a `NullPointerException`.

This is a common error in Java, and one you are likely to encounter frequently. Whenever you do encounter it, it means that some reference variable you are using the "dot syntax" on, points to `null` instead of an object, and thus using the "dot syntax" on that reference variable doesn't make sense, given the value inside the reference variable. It might mean you haven't assigned a value to the reference variable yet, or it could mean you incorrectly assigned to the reference variable when you did assign to it, i.e. you wrote `null` into the reference variable at some point, when you didn't mean to do so. In any case, the `NullPointerException` is the end result.

Note that this can also occur with arrays – only in that case, it's the "bracket syntax", as well as the "dot syntax", that you are worried about. First of all, you cannot ask for the length of an array that doesn't exist:

```
int[] scores;   // right now, the variable "scores" holds the value "null"
int num;
num = scores.length;
```

The last line will trigger a `NullPointerException`, again because you are using the "dot syntax" on a reference variable that holds the value `null`. However, you cannot use the bracket syntax on such a reference variable, either. So in the following code:

```
int[] scores;   // right now, the variable "scores" holds the value "null"
scores[0] = 7;
```

you will again get a `NullPointerException` from the last line, since the syntax "`scores[0]`" is basically doing the same thing that a line such as `office.hour` would do – it is giving you access to a variable *within* the object the reference points to. The difference between the two situations is basically one of naming. The `length` variable of an array object and the `hour` variable of a `Clock` object, actually have names, and so you access them from their references using the "dot syntax". But the individual cells of an array are accessed using integer indices – i.e. the "names" of those

variables, are numbers, not actual names. And so in that case, we use the "bracket syntax" instead of the "dot syntax", since if we used the "dot syntax", we'd get things like the last three lines of the following code:

```
int[] scores;
scores = new int[6];
scores.0 = 7;
scores.5 = 14;
System.out.println(scores.5);
```

and that looks a little strange. So that's why we use the bracket syntax instead, if our "names" are actually integer indices, rather than real names like `hour` or `length`:

```
int[] scores;
scores = new int[6];
scores[0] = 7;
scores[5] = 14;
System.out.println(scores[5]);
```

The point here is that the "dot syntax" and the "bracket syntax" *mean* the same thing – "follow the arrow, to the object this reference points to" – and thus they can both trigger a `NullPointerException` if the reference you are using the "dot syntax" or "bracket syntax" on, holds the value `null` instead of the address of an actual object.

Arrays of non-primitive types

Just as you could create arrays whose cells held values of primitive types, you likewise can create arrays whose cells hold reference variables of a particular type. The syntax for doing this is the same as it was for primitive types; the only difference is that, when you needed a type before, you used a primitive type, and now you could also use a non-primitive type.

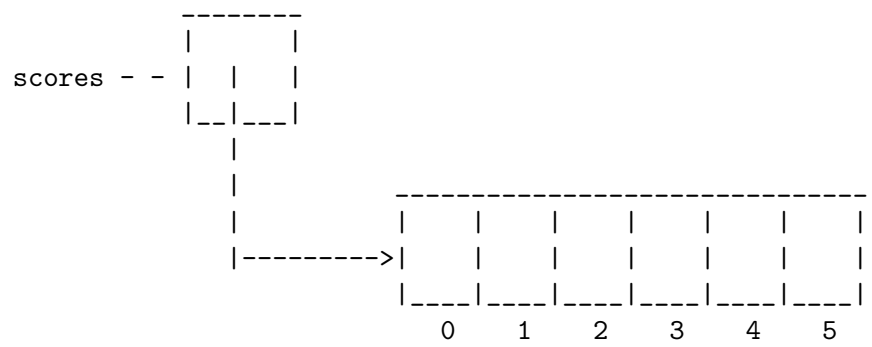
For example, you could declare a reference to a `Clock` array using the same syntax you declare a reference to an `int` array:

```
// Type varname;
int[] scores;
Clock[] times;
```

and then just as you can allocate an integer array object for the integer array reference to point to, you can allocate a `Clock` array object for the `Clock` array reference to point to.

```
// varname = expr;
scores = new int[6];
times = new Clock[6];
```

There is a concern, however. Remember that when you create an array, you still need to initialize it. If we had the integer array declaration and allocation above, we would have the following picture:



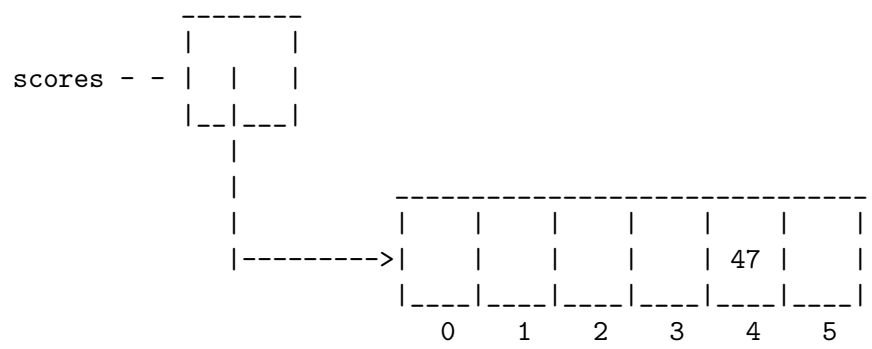
Now consider the statement:

```
System.out.println(scores[4]);
```

What happens if we now run that statement? What will get printed? We really have no idea, since we never initialized `scores[4]`. If we initialize `scores[4]` and *then* run the `System.out.println(...)` statement, however:

```
scores[4] = 47;
System.out.println(scores[4]);
```

then you actually know what value will be printed – the value you had written into that cell a statement earlier, namely, `47`.



In this respect, array cells – which are effectively variables – aren't any different than stand-alone variables. After all, when we discussed arrays, we said to treat an array cell just like any other variable. And with any other variable, you'd have to initialize it before printing it, or else you have no idea what value you'll actually print:

```
int x;
System.out.println(x);  // what gets printed? who knows?
x = 47;
System.out.println(x);  // this time, 47 gets printed
```

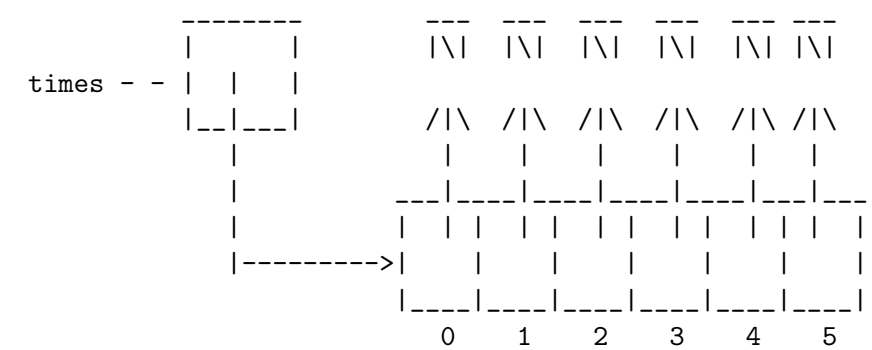
The only difference between the first example and the second one, is that in the first example, the integer we want to print is part of an array, and in the second example (the four lines of code above), the integer is a stand-alone local variable. In either case, however, we want to write a value into the variable before reading the variable; if we read the variable before we ever initialize the variable, we don't have any idea what value we'll see there. (The compiler might even complain, in some cases.)

So what does all this have to do with objects and classes? Well, consider again the declaration of a `Clock` reference variable and the allocation of a `Clock` array object:

```
Clock[] times;
times = new Clock[6];
```

When you create this array, you are creating an array of six *reference variables*. But just as with the integers above, you will have not initialized those six array cells yet. So you have an array of six un-initialized reference variables.

Well, not quite. We did say, that the virtual machine automatically initializes all reference variables to `null`. So, what we really have, is six array cells, all of which point to `null`:



So now, imagine trying to initialize the `hour` variable of one of the clocks:

```
times[4].hour = 7;
```

This syntax is no different than if we had done the following:

```
Clock office;
office.hour = 7;
```

As we have already discussed, the code immediately above will generate a `NullPointerException` – since we have not assigned the variable `office` to point to a new object, it points to `null` and thus there is no `hour` variable to assign to. And as we pointed out with integers above, there is no real difference between a stand-alone integer variable and an integer variable inside an array, other than the syntax you use to obtain the "name" of the variable. It's no different with reference variables – there's no real difference between a stand-alone `Clock` reference variable, and a `Clock` reference variable that is one of the cells of an array, other than the syntax you use to obtain the `Clock` reference variable.

So, in our array example, each of the cells indexed `0` through `5` is a reference variable of type `Clock`, holding `null`, just as in the above example, `office` is a reference variable of type `Clock`, holding `null`. And so, just as the line:

```
office.hour = 7;
```

will generate a `NullPointerException`, likewise, the line:

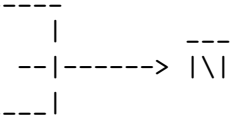
```
times[4].hour = 7;
```

will generate a `NullPointerException` as well, since the only difference between the two is in how the `Clock` reference variable was obtained.

Sometimes, people get confused on this point, since they know the array has been allocated. Indeed it has, but if the array is allocated, that only means the five `Clock` reference *variables* have been created – it doesn't mean any of them actually point to `Clock` *objects*. It's no different than when you declare one `Clock` reference variable – that doesn't mean the reference variable automatically points to a `Clock` object. So, just because your *array* reference variable points to an array object, it does not mean all the references *inside* that array object, themselves point to objects. That is why drawing pictures is so useful; if you remember that this line:

```
Clock[] times;
```

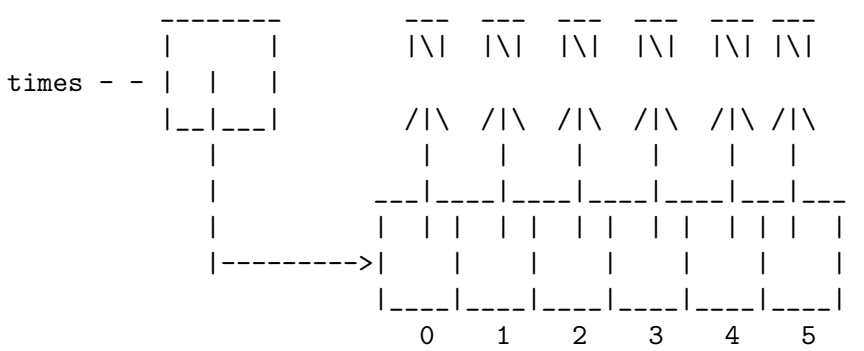
gives you this picture:



and that with these *two* lines:

```
Clock[] times;
times = new Clock[5];
```

you get the following picture:

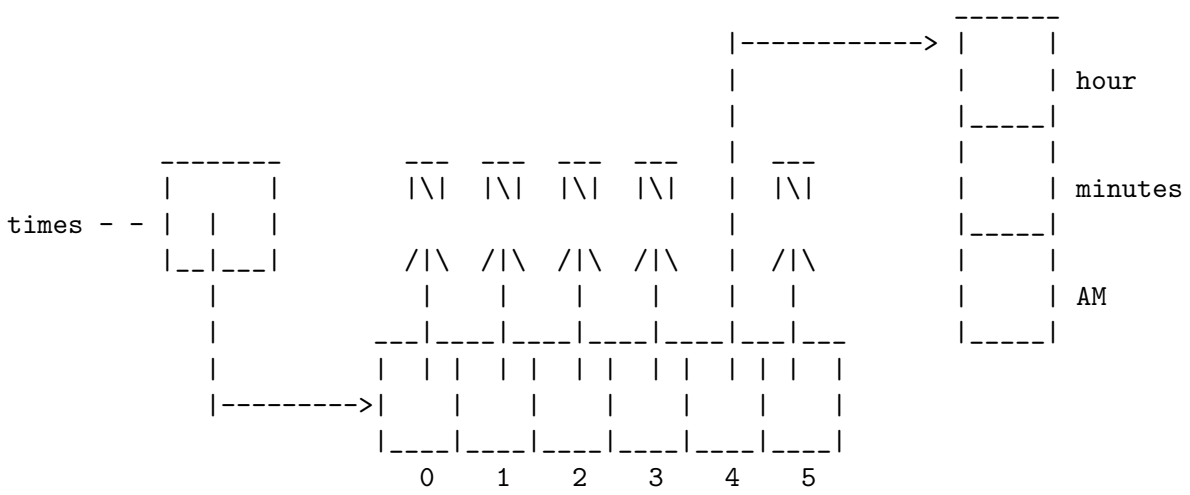


then that might help you remember that the individual `Clock` reference variables also need to be assigned to point to objects, just as the array reference variable `times` was assigned to point to an object.

So, what you need to do if you want to write into an `hour` variable, is to have the following *three* lines (the third one is the one we've added to the previous two examples):

```
Clock[] times;
times = new Clock[5];
times[4] = new Clock();
```

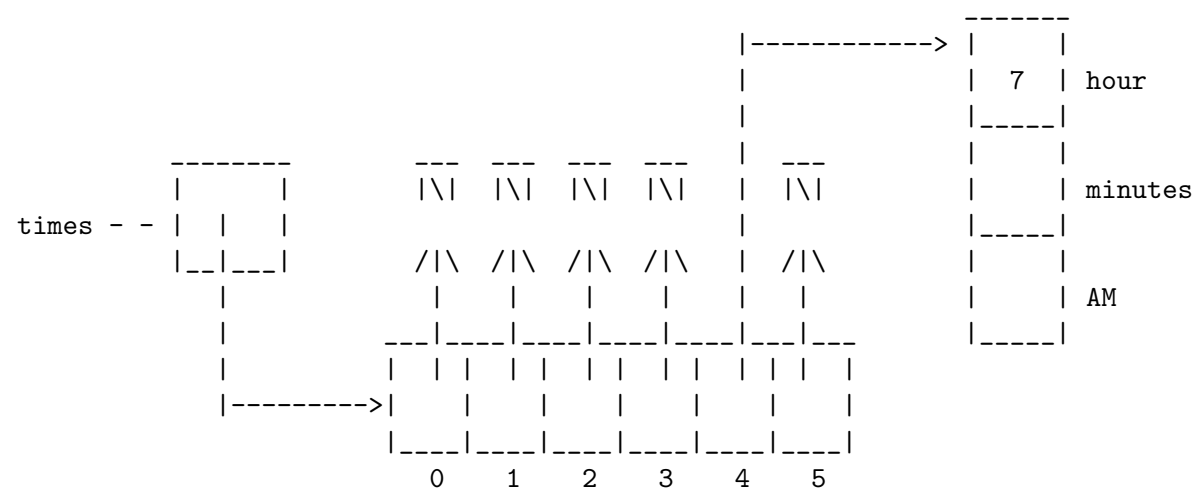
Now that we are also allocating an object for `times[4]` to point to, we get the following picture:



And so, if we add a fourth line to our code – the assignment to an `hour` instance variable that we wanted to perform earlier:

```
Clock[] times;
times = new Clock[5];
times[4] = new Clock();
times[4].hour = 7;
```

this time, when we follow the arrow from the `Clock` reference at `times[4]`, to an object, there is indeed a `Clock` object at the end of that arrow, rather than `null` – and thus there is indeed an `hour` variable there, and thus the assignment works just fine:



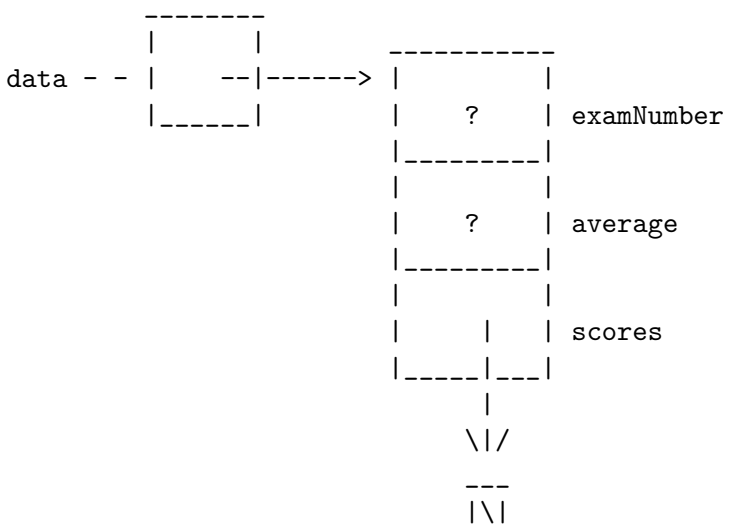
Note that this same problem could happen in reverse, if you had a class which had instance variables that were not of a primitive type. For example, consider the following class:

```
public class ExamScores
{
   public int examNumber;
   public double average;
   public int[] scores;
}
```

One easy mistake to make, is to write code such as the following:

```
ExamScores data;
data = new ExamScores();
data.examNumber = 1;
data.average = 80.3;
data.scores[0] = 89;
data.scores[1] = 57;
data.scores[2] = 95;
```

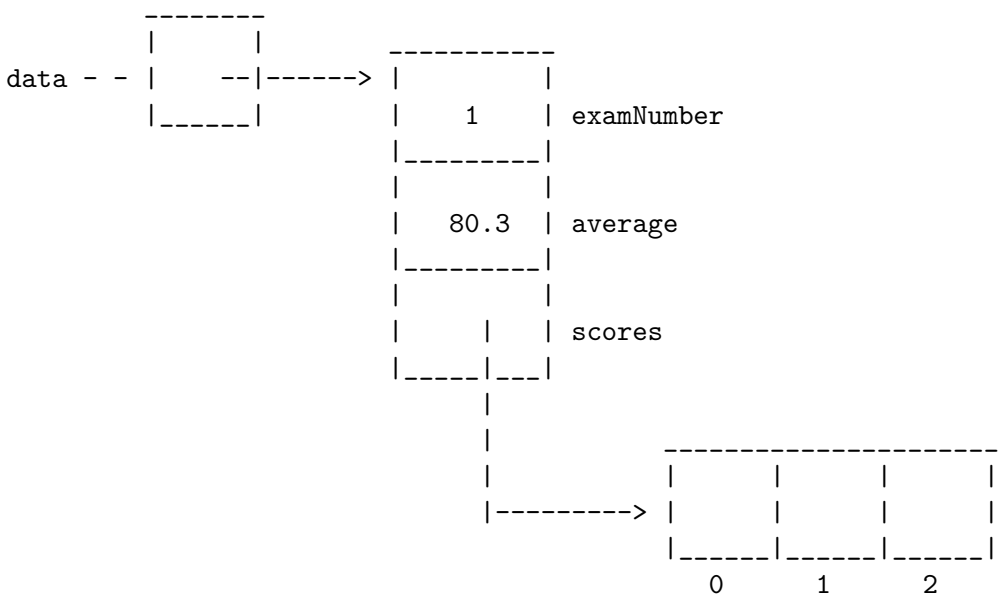
Each of the last three lines is a problem, because the reference variable `scores` is `null`. When we create a new `ExamScores` object, we get the following:



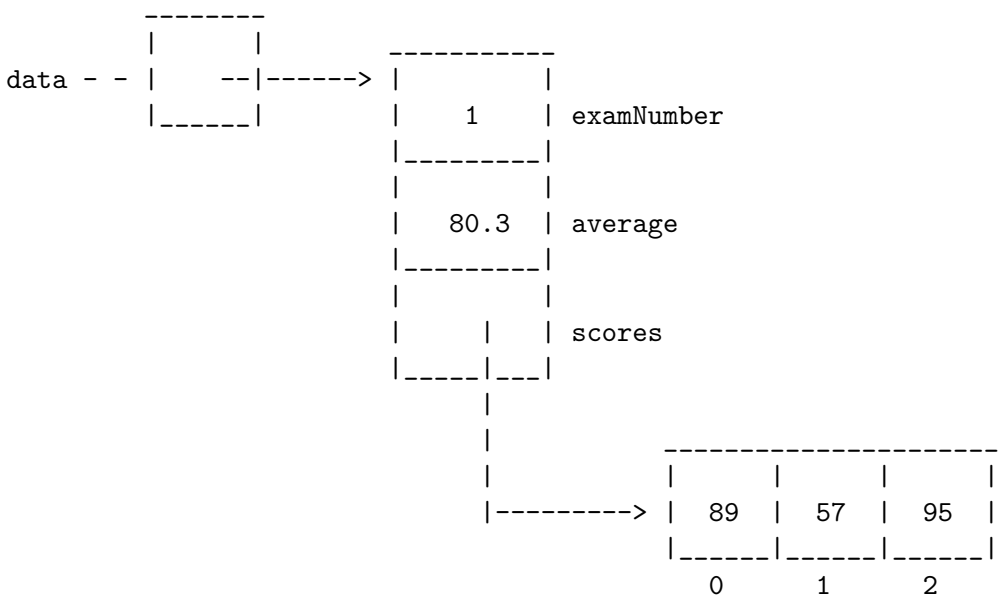
The `int` variable `examNumber` and the `double` variable `average`, are uninitialized, and so we do not know what values those variables hold. The `scores` variable, being a reference variable, is initialized to `null`. And as we discussed earlier, if we then try and access individual cells of the array object `data.scores` points to, when `data.scores` doesn't actually point to an array object in the first place, we would get a `NullPointerException`. We need to not just allocate the `ExamScores` object, but also, since `scores` is a reference variable, we need to allocate an object for that reference variable to point to as well, as in the following code (which is the earlier code, with the indicated line added in the middle):

```
ExamScores data;
data = new ExamScores();
data.examNumber = 1;
data.average = 80.3;
data.scores = new int[3]; // this is the line we have added to the earlier example
data.scores[0] = 89;
data.scores[1] = 57;
data.scores[2] = 95;
```

Running the above code up through the allocation of the integer array (i.e. ignoring the last three lines for now) would give us the following picture:



and thus we can run the last three lines as well, since now `data.scores` does indeed point to an array object, and we can write the cells indexed `0`, `1`, and `2` at that object:



So, when dealing with references – whether local reference variables, or parameter reference variables, or reference variables stored in the cells of an array, or reference variables that are the instance variables of objects – keep in mind that you need to always initialize a reference to point to an object, before you can use the "dot syntax" or the "bracket syntax" on that reference.