Array of objects

An array of objects is really an array of object references, and objects themselves do not reside inside the array, only object references reside inside the array. Here is a simple diagram for representing one such situation in memory.

Let’s use an array of Student objects as an example as follows:

```java
Student[] sa = new Student[n]; // for some n
```

The line above would generate a situation like this;

```
+------+
| 800  |
+------+ |
     v
800: | null | null | ... | null |
+-----------------------------+
0 1 n-1
```

If we then create a Student object and assign it to each location of the array, s1 through sn, like this:

```java
sa[0] = new Student(...); // s1
...
sa[n-1] = new Student(...); // sn
```

then it would look like this:

```
+-------------+
|     |        |
| 800: | null | null | ... | null |
| 0 1  n-1 |
```

where s1 is one of the Student objects that would look like this assuming that Student class has three fields (two ints and a double):

```
+-------------+
|            |
|    | (32 bits) | (an int field) |
|    | (32 bits) | (an int field) |
|    | (64 bits) | (a double field) |
```

So, there is a level of indirection from an array to the actual object that is being referenced by an object reference in the array just like a variable of object type has a reference into an object. Not surprising, right? After all, an array is an array of variables.

Remember that an object reference is a piece of memory large enough to hold a 32-bit memory address? Well, each of the elements in the array sa is one such location. So, if you create an array of 100 Student objects, then there will be 100 object references of 32 bits each and 100 times the number of bits that are
required to represent each individual Student object. If a Student object has two int fields and a double field in it, then each object would take up \((32 + 32 + 64)\) bits. 100 of them will take up \((32 + 32 + 64)\) x 100 bits. So, an array of 100 Student objects would take up \((100 \times 32) + ((32 + 32 + 64) \times 100)\) bits altogether. And, one more object reference into the array object itself. After all, an array is an object and the name of the array would be an object reference itself.

If you declare a variable, say \(s\), of type Student, then \(s\) itself would be either null or an object reference plus the Student object itself that is referenced by the object reference. So, a variable like \(s\) would take up \(32 + (32 + 32 + 64)\) bits if it is referencing an actual Student object. If its value is null, it would just occupy 32 bits for the null value.