Arrays

Introduction

In this article from my free Java 8 course, I will be discussing arrays. The array is an extremely powerful 'tool' that allows you to store multiple objects or primitive data types in one place.

Arrays

An array is a special type of object in Java. Imagine it like a container that can hold a number of primitive data types or objects. It stores each one in its own 'compartment' and allows you to access them by providing the location of the 'compartment', called an index.

Let's say we wanted to create an array of Person objects. You would do it like this:

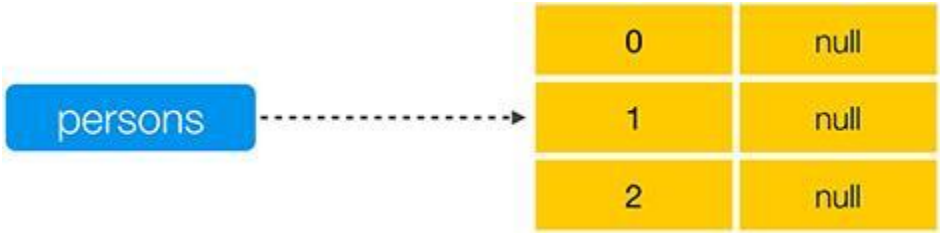
```
Person[] persons = new Person[4];
```

Example 1

The first part `Person[] persons` defines a Person-array reference variable named `persons`. The second part, `= new Person[4]` creates a Person-array object and assigns it to our reference variable `persons`. `[4]` indicates that the array will be able refer to a maximum number of four Person objects. You can't change the size of an array after you've created it. So once you've initialized the array, that's it, it's stuck at that size.

When instantiating an array of objects,such as a Person array in our case, you create **one** array object, acting as container, *offering space to store references* to the actual objects.

After instantiation, your array will be empty. Each cell will contain `null`, which means it is not referencing any object, as illustrated by Example 2. If we tried to access the reference at this position we would get an error once we ran the program. As a side note, the proper term for an error like this is an 'exception' (you can read more about Exceptions in Java here).



Example 2

Now, let's fill our array. When we do this, we're filling it with reference variables to `Person` objects.

```
persons[0] = new Person();
persons[1] = new Person();
persons[2] = new Person();
persons[3] = new Person();
```

Example 3

Example 4

Syntactically, you could also put the square brackets of the `persons` array variable after the variable declaration, as Example 5 shows you:

```
Person persons[] = new Person[4];
```

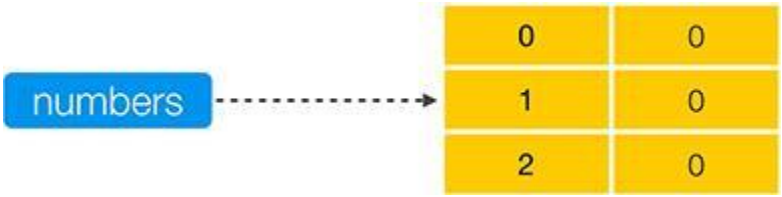
Example 5

The code in Example 5 is a flaw of the Java Programming Language and it's highly recommend never to do that. Reading the code is less clear - since if you only read part of the line, you could come to the wrong conclusion that it creates a reference variable of type Person. Put the square brackets **directly after** the Person class, to clearly indicate that this is a reference variable of type Person array, and **not** Person.
We can also create an array to store a primitive data type like an `int`:

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

Example 6

You are probably not used to seeing `new` in front of a **primitive** int. However, this is syntactically correct; it creates an array object that can hold three values of type `int`, **and not a primitive type**. However, while an array of objects stores spaces for references, an array of primitives stores the primitive values themselves. Therefore, after initialization, the cells of the array will be pre-filled with `0`, the default value for an `int`, and not `null`, as Example 7 illustrates:



Example 7

Remember, whether it is storing object references or primitive data types, an array is an object. Since arrays are objects, you can call methods on them. For example, you can call `myInts.toString()` on the array in Example 6. You can also access the array's `public` attribute, `length`, that tells you the static length of the array. You might remember that in a previous article I talked about why you should make your instance variables `private` inside a class when using an object-oriented approach. This is yet another flaw in Java, a place where Java itself unfortunately violates basic object-oriented principles.

Multidimensional Arrays

You can also create a multidimensional array. A multidimensional array is an array of arrays:

```
int[][] numbers = new int[2][3];
```

Example 8

Conceptually, you can visualize a two dimensional array as a table with row and column indexes, as Example 9 illustrates:

Row Index / Column Index	0	1	2
0	0	42	3
1	6	6	-33

Example 9

Each cell of the first array forms the rows of the table. Each row contains yet another array, where each array forms the cells of each row. Arrays of more than two dimensions are less common, but easily possible, as Example 10 shows you:

```
Person [][][] persons = new Person[2][4][3];
```

Example 10

Multidimensional arrays are read from left to right, with each value acting like a coordinate for each primitive value or object reference. The topmost array in the hierarchy is the leftmost array. It is storing the arrays that are referenced by the subsequent set of square brackets.

Shorthand Notation for Arrays

Besides the way I explained it in Example 1, there is an alternate way to create arrays. The most basic form of it is this:

```
Person[] persons2 = {};
```

Example 11

The code in Example 11 creates an empty array, with `person2` referencing it. The curly braces surround every object that we are putting into the array. This array is size `0`, which isn't really useful in any sense, but technically it's possible.

We can also use this method to actually fill an array without setting the size. When you create the array, you put in as many objects and/or `null` values as you want and that decides how large the array will be.

```
Person[] persons = new Person[3];
```

Example 12

Indexing in an array

Let's return to the array of Person references that we created before. Let's fill it with person objects. First we have to index to the 'compartment' in our array by putting its index value in square brackets like below:

```
persons[0] = new Person();
```

Example 13

Arrays start indexing at `0`, so our four 'compartment' array has indexes at `0`, `1`, `2` and `3`. We could, as we did in Example 3, create four references and assign them each to new objects. Alternatively, we can assign our new reference variables to existing objects, or even to objects that other cells of the array are referencing. Obviously, in such a simple example, it might not be necessary to introduce these complexities, but I'm doing it to demonstrate the concepts.

```
@Test
public void demonstrateArrays() {
     Person[] persons = new Person[4];
     persons[0] = new Person();
     persons[1] = new Person();
     persons[2] = persons[1];
     Person myPerson = new Person();
     persons[3] = myPerson;
}
```

Example 14

For Loops and Arrays

Loops and arrays are always a couple. For an array to be efficient it needs a loop, and loops are well equipped to work arrays. Let's say we wanted to index through every single 'compartment' in our array and apply the same code to each one. We can create a for-loop to do this for us. Our variable `i`, will be used as the value for our index. Now, inside this loop, we could create the objects of type person and access the objects.

```
@Test
public void demonstrateArrays() {
     Person[] persons = new Person[4];
     for(int i = 0; i < 4; i++){
          persons[i] = new Person();
          person[i].helloWorld();
     }
}
```

Example 15

Inside this loop we could utilize each person in the array and have them call the `helloWorld()` method. Loops are an extremely convenient way to repetitively execute an operation on each cell of the array, without having to duplicate the code.

```
for(int i = 0; i < persons.length; i++){
     persons[i] = new Person();
}

Person myPerson = new Person();

Person myPerson2 = null;
Person[] persons2 = {persons[0], null, myPerson, myPerson2};
```

Example 16

Shorthand Notation for Multidimensional-Arrays

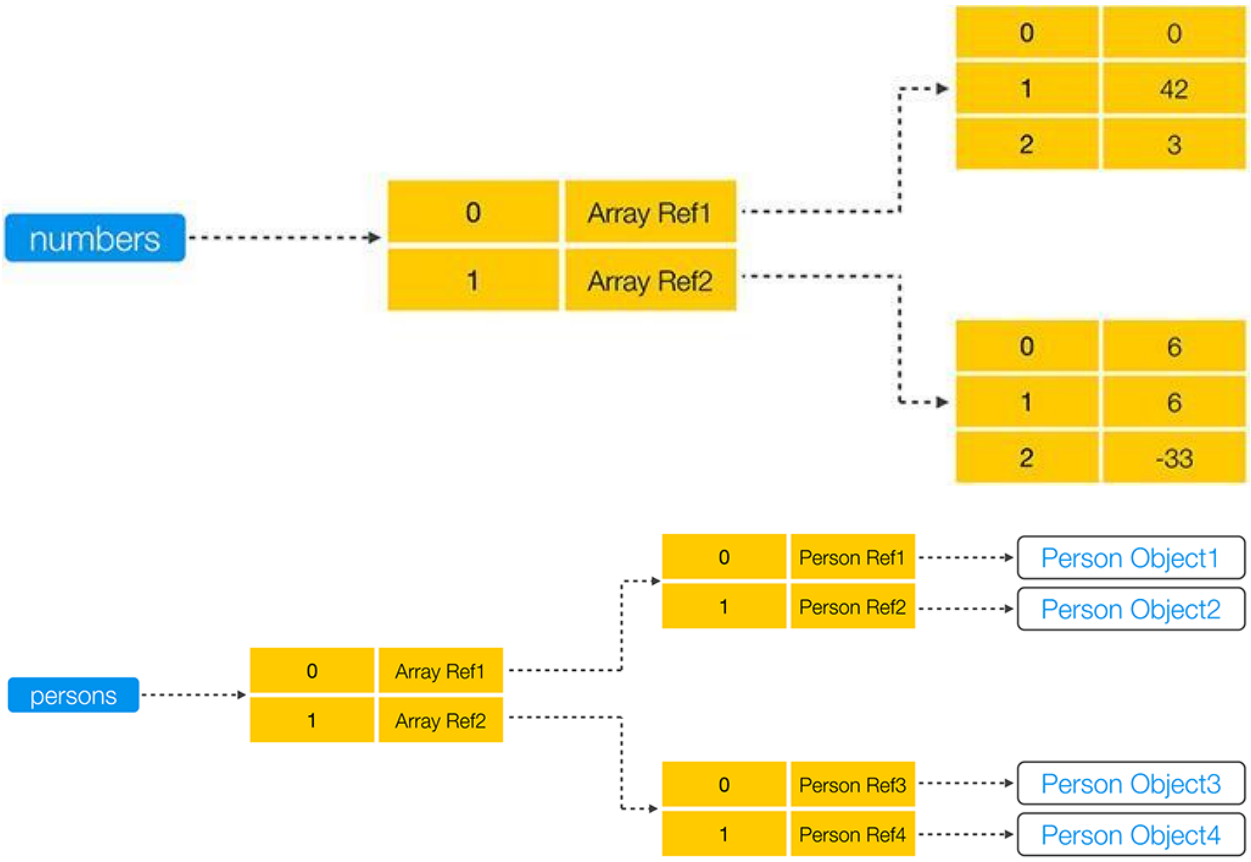
You can also utilize this shorthand notation for multidimensional arrays. To do this, you surround each 'inner array' with curly brackets and separate each set of values with a comma:

```
int[][] numbers = {
     {0,42,3, },
     {6,6,-33, },
};

Person[][] persons = {
     {person1, person2, },
     {person3, person4, },
};
```

Example 17

Examples 18 and 19 show how this will look in memory:



We can also use two for-loops, commonly known as a nested loop, to index a two dimensional array:

```
@Test
public void demonstrateTwoDimensonalArrays() {

     Person[] persons = new Person[4][4];
     for(int i = 0; i < 4; i++) {
          for(int j = 0; j < 4; i++) {
               persons[i][j] = new Person();
               person[i][j].helloWorld();
          }
     }
}
```

Example 20

Utilizing a For-Each Loop

As I mentioned in my last article about the For-Each Loop, for-each loops are extremely useful when applied to arrays. Utilizing the for-each loop we can iterate through every object in the array without having to know the length of the array.

```
for(Person person: persons) {
     // do something to each object in the array
}
```

Example 21

Now we've learned in full about how the for-each loop and the array work, so I've shown you one of the most powerful pairings in Java. You can use arrays to store objects or primitives and iterate through them with loops- an extremely efficient and clean way to code.

Thanks for reading!