Java Array Facts

For CS302 by Andrew Weinrich 2007 pages.cs.wisc.edu/~weinrich

1. An array is a way to collect multiple *values* by using only one *variable*.

Remember the difference between a variable and a value: a variable is just a *name*, a value is a *number* or *pointer* attached to it. Arrays let us deal with multiple *values* by using only one *variable*.

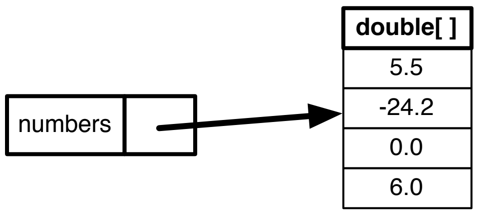
An array is a special kind of object that collects several *values* into a fixed *order*. For example, if we have several variables of the same type:

```
double number1 = 5.5;
double number2;
double number3 = -24.2;
double number4 = 6.0;
```

We can use an array to store these four values with only one variable:

```
double numbers[] = new double[4];
numbers[0] = 5.5;
numbers[1] = -24.2;
numbers[3] = 6.0;
```

Here's what the memory diagram of this code looks like:



The array adds an extra level of indirection between the variable and the values.

2. Arrays are objects

Arrays are actual objects, and are created on the *heap*, not the *stack*. We can tell that they are objects because we create them with the `new` keyword. When creating an array, we must specify the *type* of the array and the *number* of elements it will contain:

```
type[] arrVariable = new type[size];
```

3. An array has a certain number of elements in a fixed order.

The separate values stored in an array are called *elements*. Each of these elements is identified by its numeric *index.*

4. Elements of arrays are accessed with the special notation `arr[x]`.

To access an element of an array, we use a combination of the array variable name and the index of a specific value. This tells the virtual machine to follow the pointer stored in the *variable* to the array *object*, and return the value at the given *index*. Borrowing a term from mathematics, the index is sometimes called the *subscript*.

5. Arrays indices start at 0.

The valid indices of an array are `0` through `size-1`, where `size` is the number of elements in the array. So if the size of an array is `10`, the index of the last element is `9`.

6. Accessing an invalid array index causes an exception

Whenever we try to access an element of an array, the Virtual Machine makes sure that the index is valid. If it is not, the VM raises an error that will crash the program.

7. Arrays elements have default values

When an array is created with the `new` keyword, the elements will be given the same default values as member variables. For primitive types, this is `0` or `0.0`; for objects, it is the `null` pointer.

8. The size of arrays can be decided at run time

If we had to use constant sizes when creating arrays, they would only serve as a convenience. But we can use integer variables for the size when creating an array:

```
int arraySize = ...;
double[] arr = new double[arraySize];
```

9. Arrays have a `length` field

We can always find out the length of an array by accessing the special `length` property. See the next fact for an example of how to use this property.

10. Arrays and `for` loops are made for each other

A very common task is to iterate over all the elements in an array and perform some operation, such as printing or totaling. This is a perfect place to use a `for` loop:

```
int[] arr = ....
for (int i = 0; i < arr.length; i++) {
    S.o.p(arr[i]);
}
```

We use an index variable that ranges over all the valid indices of the array, starting at `0` and going up to `length-1`.

11. Arrays can be created with literal syntax

If we know what elements we want to put into an array, we can use a special syntax to create a literal array. This is like creating a literal `String` or number. To create a literal array, use a pair of curly braces, with the values of the array elements inside, separated by commas:

```
String[] strings = { “AA”, “BB”, “CC” };
int[] numbers = { 1, 2, 7, 8, 123, 0 };
```

12. Array elements can be used directly

We do not have to pull an element out of an array into an individual variable before we use it:

```
String[] strings = new String[10];
// fill in elements of the string array
for (int i = 0; i < strings.length; i++)
    S.o.p(strings[i].toUpperCase();
int[] numbers = { .... };
int sum = numbers[2] + numbers[3];
```

However, this code will not compile:

```
// tying to call a method on an array
S.o.p(strings.toUpperCase());
// trying to use an array as a number
int average = numbers / numbers.length;
```

13. All elements in an array must be of the same type

When we create an array, we have to specify the type of value it contains. This can be either a primitive type or an object reference type. Assignment of a different type will not work:

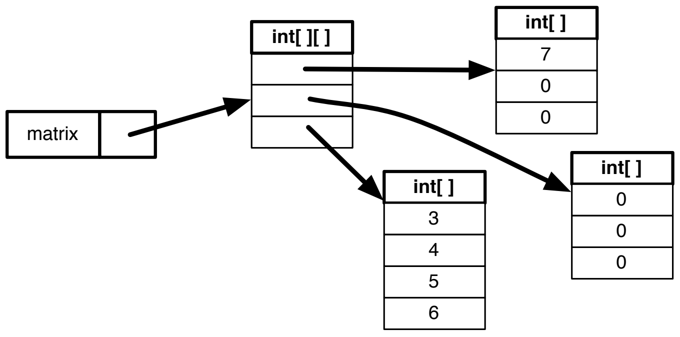
```
int[] numbers = new int[10];
numbers[5] = 5.6;  // will not compile
```

14. Arrays can be nested

We can create multidimensional arrays by using two pairs of brackets.

```
int[][] matrix = new int[3][3];
matrix[0][0] = 7;
int x = matrix[1][2];
int[] rowB = matrix[1];
matrix[2] = { 3, 4, 5, 6 };
```

This does not actually create a two dimensional matrix; instead, it creates an array of array references. Arrays can be nested to any depth, but more than 2 dimensions is rare. The memory diagram for this code looks like this:



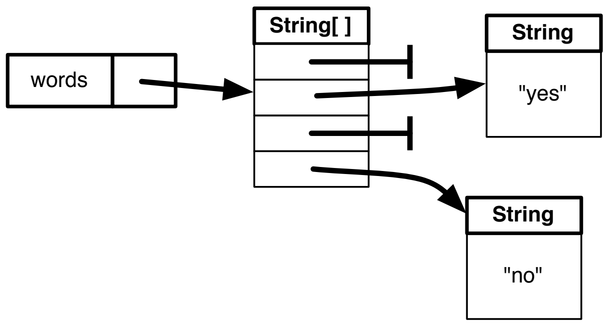
Note that while multi-dimensional arrays are created to be “square”, the VM does not enforce equal row lengths after creation. A multi-dimensional array with uneven row lengths is called a *jagged array*.

15. Arrays can be partially filled

If we have an array of objects, it is possible for the array to be *partially filled*, meaning that not every slot in the array points to an actual object. When using partially filled arrays, we have to be careful that we do not try to use an element in the array that is null:

```
// all elements initialized to null
String[] words = new String[4];
words[1] = “yes”;
words[3] = “no”;
// this line will crash
S.o.p(words[2].toUpperCase());
```

Here is the memory diagram produced by this code:



Arrays of primitive values cannot be partially filled.