1 Arrays in Java

Sometimes, you need to declare a whole group of variables, and you do not want to name them one by one. For example, you may wish to store the first 1000 prime numbers. Or, you may wish to store the temperature for each day in this year. You certainly do not want to give a name to each varible. Instead, what you want to do is declare an *array*.

The general form of a (one-dimensional) array declaration in Java is as follows:

```
<type name>[] <array variable name>;
```

Alternatively, you can use the form more common in other languages too (although it is not reccomended):

```
<type name> <array variable name>[];
```

A few examples would be:

```
int[] phys495Grades;
float[] dailyExchangeRates;
char[] phys495LetterGrades;
```

Arrays are distinct from regular variables. An array variable is only a *name* for an array of that particular type. The declaration does *not* create the array itself; it only creates a name that can *refer* to an array.

The actual array needs to be created using the `new` operator. An example is probably in order:

```
int[] problemSet1Grades;

problemSet1Grades = new int[10];
```

The first statement declares the name `problemSet1Grades` as a name that can refer to an array of integers. The second statement creates a new array of 10 integers (why 10?) in memory, and makes the name `problemSet1Grades` a reference to that new array.

Of course, these two can be combined in a single statement as follows:

```
int[] problemSet1Grades = new int[10];
```

2 Accessing Elements of an Array

How can one access individual variables in an array? This is done by following the array name with a pair of square brackets containing the index of the element. Note that array indices range from 0 to size of array minus one. So an array of 10 elements will have indices 0 through 9.

3 Reusing Array Variables

As mentioned, an array variable is only a reference to the actual array in memory. As such, one can create a new array, and link the array variable to the new array, while discarding the old one. Here is an example:

```
int[] primeNumbers; // This only declares the name.

primeNumbers = new int[10]; // This creates a new array.

primeNumbers = new int[50]; // This discards the old array.
```

4 Initializing Arrays

If need be, you can also initialize the values in the array when creating the array. Here is an example:

```
char[] phys495LetterGrades = {'A', 'A', 'B', 'B', 'C',
                              'D', 'F', 'F', 'F', 'F'};
```

In this case, there is no need for a new operator, or the number of elements. These are all obvious from the initalization. (Note that this is an array of type char, so the initialization values are characters in single quotes.)

5 Finding out the Length of an Array

Since array variables are just references, it is usually necessary to find out the length of an array. The length of an array can be accessed with <array name>.length. Here is a short example that will print the contents of the above declared array:

```
char[] phys495LetterGrades = {'A', 'A', 'B', 'B', 'C',
                              'D', 'F', 'F', 'F', 'F'};

for (int i=0; i < phys495LetterGrades.length; i++) {
    System.out.println(phys495LetterGrades[i];
}
```

6 Multi-Dimensional Arrays

An array is analogous to a vector in mathematics. It has more than one component. Similarly, a two-dimensional array is analogous to a matrix in mathematics. I will only give an example here, since there aren't many instances where one would use a multi-dimensional array:

```
float[][] dailyTemperature = new float[12][31];
```

7 Introducing the `String` Class

We have already effectively used the `String` class in many cases. Any time you type a string in double quotes in a Java program, you effectively use the `String` class.

However, there is more to the `String` class than that. The `String` class in Java contains many *methods* that allows one to perform various operations on the string that the `String` object contains.

8 Declaring String Variables

A `string` variable is declared just as one declares other variables. The general form is simply:

```
String <string variable name>;
```

Just as in the case of arrays, when one declares a `String` variable (which is an object, as opposed to a basic type), all that is created is *a name*. The declaration does *not* create the object itself.

You can make the newly created `String` variable actually point to a `String` object. One way of doing this is by assigning a literal string to the `String` object. This can be done either in the declaration (initialization) or later (assignment). An example for each is below:

```
String instructorName="Yasar Safkan";
String courseCode;

courseCode = "PHYS495";
```

9 String Concatenation

Two `String` objects can be concatenated by using the addition operator, "+". You have already seen its use. The "+" operator concatenates the two `String`s it operates on, and produces a new `String` object as a result. The two operands remain untouched. Here is a simple example:

```
String department="PHYS";
String code="495";
String lectureCode;

lectureCode = department + code;
```

The "+" operator works when one operand is a `String` and the other is a basic type. The basic type will be automatically converted to a `String` and concatenated with the other `String`.

Another possibility is to use the "+=" operator. It works exactly as you would expect it to. The `String` on the right gets appended to the `String` on the left.

10 String Comparison

In order to compare basic types, you normally use the "==" operator. You can use the "==" operator on two `String`s but it will not quite work as you would expect it to. The "==" operator will produce a `true` result *if and only if* the two `String` variables refer to the *same* `String` in memory. It will produce a `false` result for any two distinct `String` objects even if their contents are identical.

The proper way to compare two `String`s for equality is to use the `equals()` *method* of the `String` class.

How does one call a *method* of a class? The general form of doing this is:

```
<class variable name>.<method name>(<method arguments>)
```

You have already seen similar uses. For example, we used the square root method of the `Math` class in previous excercises. That was a little different, however. We could call the `sqrt()` method without ever having created a variable of the `Math` class. That is because the `sqrt()` method is a *static* method of the `Math` class. A *static* method can be called in the form <class name>.<method name>(<method arguments>) without ever creating a variable of that class.

Coming back to the original discussion, here is an example of how one uses the `equals()` method to compare two `String`s, which you can type in and try out:

```
public class CompareString {
    public static void main(String[] args) {
        String string1="PHYS";
        String string2="495";
        String string3="PHYS495";
        String string4;

        string4 = string1 + string2;

        System.out.println("string3 is: " + string3);
        System.out.println("string4 is: " + string4);

        if (string3 == string4) {
            System.out.println("string3 == string4");
        } else {
            System.out.println("string3 != string4");
        }

        if (string3.equals(string4)) {
            System.out.println("string3.equals(string4) == true");
        } else {
            System.out.println("string3.equals(string4) == false");
        }
    }
}
```

What do you think this program will produce as output when run? Here is the result I got when I executed it:

```
string3 is: PHYS495
string4 is: PHYS495
string3 != string4
string3.equals(string4) == true
```

As you can see, although the strings `string3` and `string4` contain identical strings, they are *different* string objects, and the "==" operator does not produce the result that one expects. But, using `equals()` *does* produce the intended result.

Note that the `equals()` method does a *case sensitive* comparison. It will produce a `true` result if and only if the two `Strings` are of the same length, and match character-by-character.

If a *case insensitive* comparison is desired, one should use the `equalsIgnoreCase()` method. While `equalsIgnoreCase()` does a case insensitive comparison, note that it will only work properly for the English language without additional effort.

11 Testing the Beginning or End of a `String`

Sometimes one cares only about the beginning or end of a `String`. (Probably the beginning, more often than not.) The `String` class provides two methods for each purpose: The `startsWith()` and `endsWith()` methods.

Here is an example for `startsWith()`:

```
String courseCode = "PHYS495";

if (courseCode.startsWith("PHYS")) {
    System.out.println("This is a physics course.");
} else {
    System.out.println("This is not a physics course.");
}
```

12 Ordering of `Strings`

The `String` class provides the `compareTo()` method which is used to compare two `Strings`. `string1.compareTo(string2)` will return a *positive* value if `string1` is greater, i.e., `string1` occurs *later* in alphabetical order. It will return a *negative* value if `string1` is smaller, i.e., occurs *first* in alphabetical order. The method will return zero if and only if the two `Strings` being compared are equal.

13 Accessing Individual `characters` in a `String`

The `charAt()` method of the `String` class takes an `integer` value as an argument, and returns a `char`. The `char` returned is the character which is at the position specified as the argument. An example is in order:

```
String courseCode = "PHYS495";

int stringLength = courseCode.length();
char letter;

for (int i=0; i < stringLength; i++) {
    letter = courseCode.charAt(i);

    System.out.println("Character at " + i + " is " + letter);
}
```

This code fragment will print out the characters in `courseCode` one by one. Note that the `length()` method in this code fragment has been used to figure out the length of the `String`. If the argument of `charAt()` is out of range, an exception will be thrown (in other words, it is an error).

14 Searching for "Stuff" in a `String`

Very often, one needs to parse `Strings` in a program. To do that effectively, one needs to locate where certain `char`acters or `Strings` lie within a `String`.

The `indexOf()` method comes in two variants, one which takes a `char` as an argument, and another which takes a `String` as an argument. Both forms return an `int`eger as a result, which is the index of the first occurrence of its argument within the `String`. If the argument does not exist within the `String`, -1 is returned in both cases.

There are two more versions of `indexof()`, which take *two* arguments. The first argument is an integer, which tells the method *at what position* to start the search at. The second argument is again, either the `char` or the `String` to be searched for.

A similar set of methods exist with the name `lastIndexOf()`, which return in response the *last* index rather than the *first* index of the `char` or `String` being sought in the `String`.

15 Extracting Substrings from a `String`

The `substring()` method of the `String` class takes two `int`eger arguments. The first argument is the index of the first character of the substring, and the second argument is the index of the last character of the substring plus one.

```
String courseCode = "PHYS495";

System.out.println(courseCode.substring(0, 4));
System.out.println(courseCode.substring(4, 7));
```

The code fragment above will produce the following output:

16 More about `String`s

The `String` class in Java is quite an extensive class, and contains more methods than outlined here. The authoritative source for documentation is the one distributed by Sun alongside the JDK. You should install *and* learn how to use the Javadoc. The `String` class is only a single class out of the huge Java Class Library. It is not possible to remember every method of every class in the Java Class Library. Instead, it is necessary to learn where to find the documentation and how to use it.