Week 7 Description

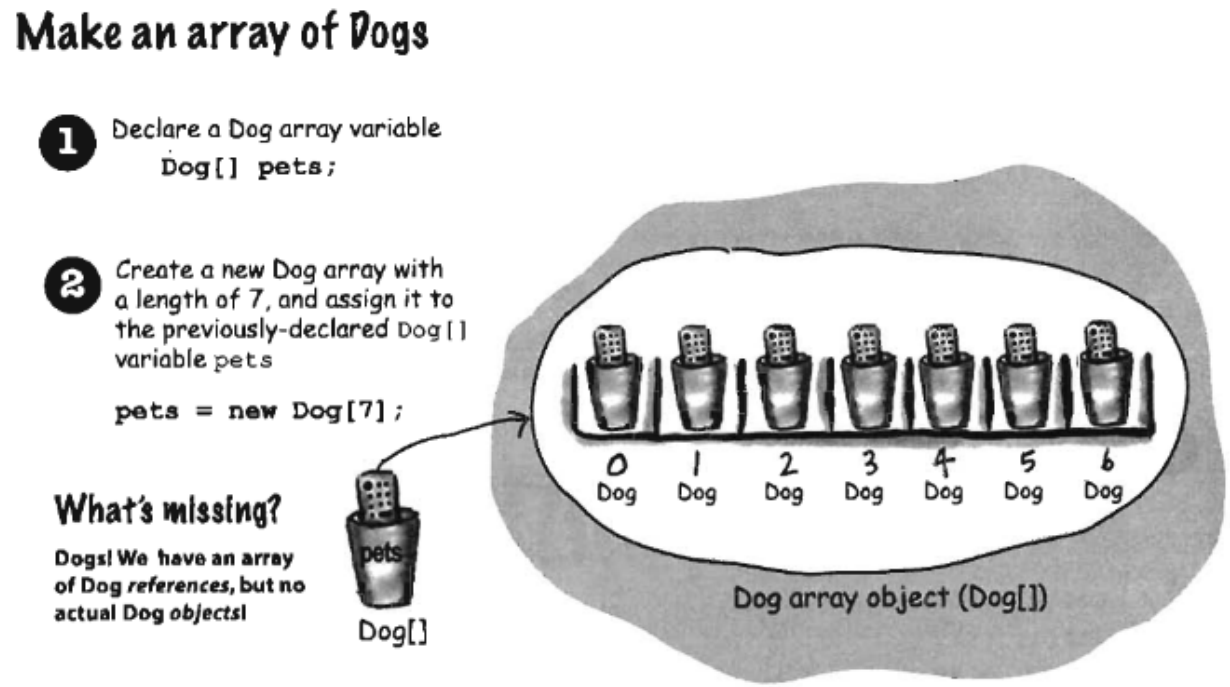
We will learn how to split arrays.

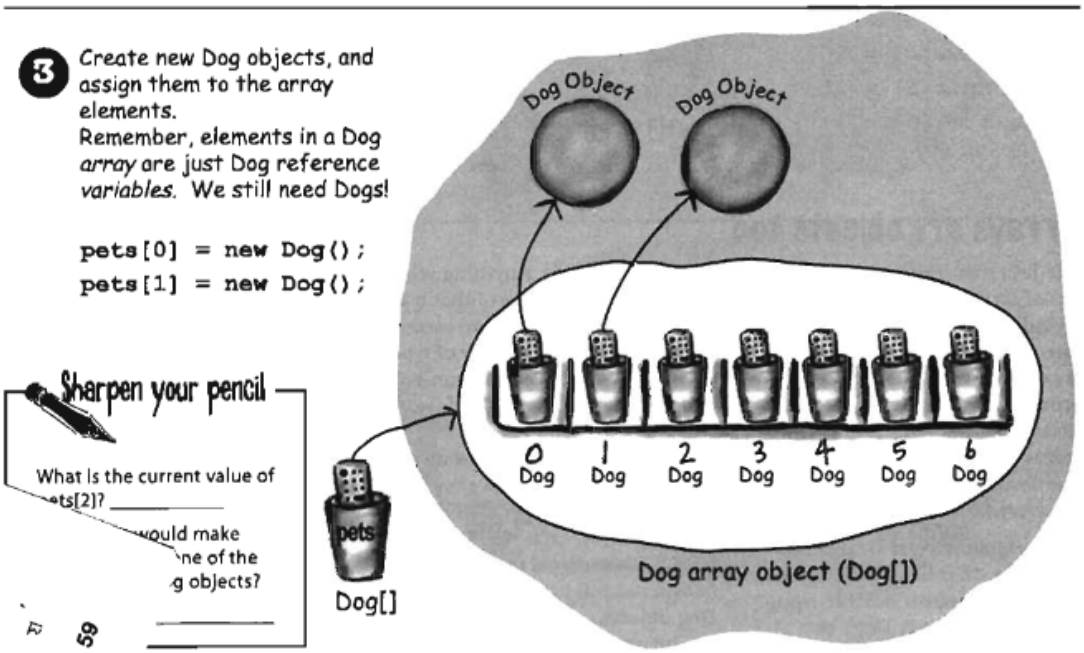
We also continue our study of arrays by working with array of objects and two-dimensional arrays.

Array of objects

Just as we can make arrays of primitive types, we can make arrays of objects. Remember that arrays themselves are objects, but they can be declared to hold primitives or object references.

When working with arrays of objects, the same principles apply as with arrays of primitives: the array declaration does not create the objects that go into it, you must do this separately:





If we want to initialize all the dogs in the pets array, we can use a for loop like this:

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

If we want to do the same thing to all the dogs in the pets array, we can use a for loop:

```
//make all dogs bark
for (int i = 0; i < pets.length; i++) {
    pets[i].bark();
}
```

or a for each loop like this:

```
//make all dogs bark - use for each loop
for (Dog pet : pets) {
    pet.bark();
}
```

String.split()

Now that we've covered arrays, we can discuss the String split method. The split method is useful for splitting up a String object into smaller strings. The method returns an array of Strings. The following snippet demonstrates the use of this method:

```
/* The split method takes a parameter to use as a delimiter and returns a
String array containing the strings (tokens) delimited by that parameter
*/
        String sentence = "This is a string object";
        String words[] = sentence.split(" ");
        for(int i = 0; i < words.length; i++){
            System.out.println(i + " " + words[i]);
        }
```

Two dimensional arrays

We have seen that an array is an object that contains several variables, called the array elements, that are all of the same data type. We can have an array of ints, an array of doubles, an array of Strings, arrays of objects, etc.

We can also create two-dimensional arrays, which we tend to think of in spreadsheet format, where there are rows and columns of data. In Java, it's important to realize that a two-dimensional array is actually an array of arrays (each of the elements in the first array is an array variable that refers to another array).

The following code:

```
double[][] anArray;
anArray = new double[3][];
```

creates an array of three elements, each of which is an array variable that refers to an array of doubles.

We have not yet created any array objects of type double[ ]. We can do so in the usual way, and assign each one to elements of the array we have already created:

```
anArray[0] = new double[4];
anArray[1] = new double[4];
anArray[2] = new double[4];
```

We have now created one array of type double[ ][ ] and three arrays of type double[ ]. The arrays of type double[ ] do not have to be the same size in Java, but we'll only discuss this relatively simple case.

As I mentioned above, we usually visualize the two-dimensional array called anArray as if it's a spreadsheet. In this case, it would have three rows, one for each of the three arrays it contains (anArray[0], anArray[1] and anArray[2]). It would also have four columns, one for each of the four doubles which are the elements in the arrays. By default, each array element is initialized to 0 for numeric primitive types, to false for booleans, and to null for class types, so our spreadsheet looks like this at this point, because our doubles currently contain only the initial values:

anArray[0]==> 0.0 0.0 0.0 0.0

anArray[1]==> 0.0 0.0 0.0 0.0

anArray[2]==> 0.0 0.0 0.0 0.0

There are three rows, and the rows are all of length 4, because of the way we declared the arrays.

We can now refer to the individual array elements using the array name followed by two subscripts:

```
anArray[0][0] = 1.0;
anArray[0][1] = 1.5;
anArray[0][2] = 2.0;
anArray[0][3] = 3.0;
anArray[1][0] = 4.5;
anArray[2][3] = 9.9;
```

After executing the assignment statements above, our array looks like:

```
1.0 1.5 2.0 3.0
4.5 0.0 0.0 0.0
0.0 0.0 0.0 9.9
```

Each element of the 2-D array is a variable of type double, and can be used anywhere that you would normally use a double variable in a program.

If we like, we can do the declaration in a single step:

```
double[][] anArray = new double[3][4];
```

new double[3][4] does not just create one array. It creates an array of type double[ ][ ] of length 3, and creates three arrays of type double[ ], each of length 4, and assigns them to the elements of the array of type double[ ][ ].

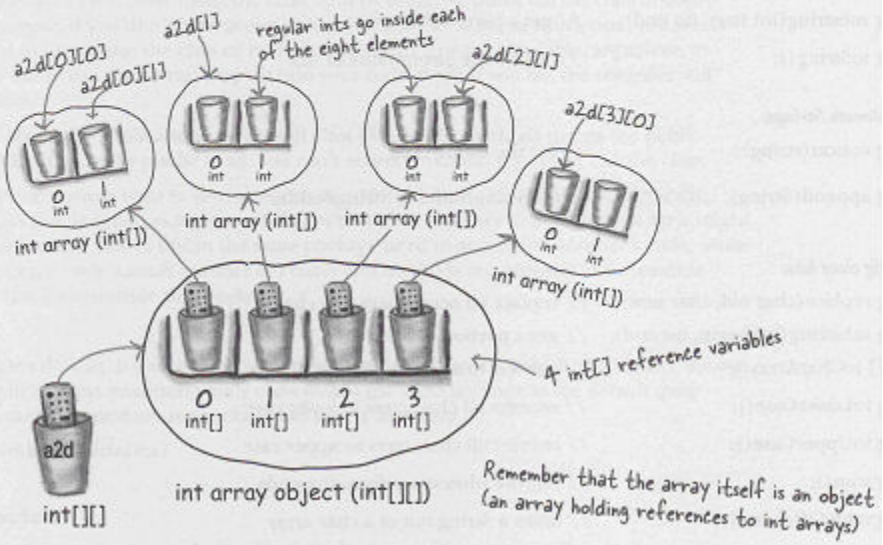
We can now work with each array element, which is a variable of type double and has two subscripts. We can also work with a complete row, which is an array variable of type double[ ].

Multidimensional Arrays

In most languages, if you create, say, a 4 x 2 two-dimensional array, you would visualize a rectangle, 4 elements by 2 elements, with a total of 8 elements. But in Java, such an array would actually be 5 arrays linked together! In Java, a two dimensional array is simply *an array of arrays.* (A three dimensional array is an array of arrays of arrays, but we'll leave that for you to play with.) Here's how it works

```
int[][] a2d  = new int [4][2];
```

The JVM creates an array with 4 elements. *Each* of these four elements is actually a reference variable referring to a (newly created), int array with 2 elements.



Working with multidimensional arrays

- To access the second element in the third array: `int x = a2d[2][1];  // remember, 0 based!`
- To make a one-dimensional reference to one of the sub-arrays: `int[] copy = a2d[1];`
- Short-cut initialization of a 2 x 3 array: `int[][] x = { { 2,3,4 }, { 7,8,9 } };`
- To make a 2d array with irregular dimensions:

```
int[][] y = new int [2][];     //  makes only the first array, with a length of 2
y[0] = new int [3];    // makes the first sub-array 3 elements in length
y[1] = new int [5];    // makes the second sub-array 5 elements in length
```