The Standard One-Dimensional Array Data Structure

**Declare and instantiate** an standard array of size 10 that can hold integers:

**int [ ] a = new int [10];**

Note the order for declaring and instantiating an array:

<data type> <empty brackets> <variable name> = new <data type> [ <physical size> ] ;



int [ ] a = new int [10];





By default Java initializes all the array elements (values) of an array of integers to zero.

As soon as we construct the array with the code above, the Java code a.length holds the integer value 10.

(Note: it is a.length NOT a.length() length is NOT a method call but a variable.)

For an array of n elements the index values are identified with the integers from 0 and going up to n-1.

If the array has been instantiated to hold 10 values then the indices go from 0 to 9.

**To store values** in the array you must use square bracket [ ] notation.

We can store values in this array with 10 separate assignment statements, by accessing and assigning a value to each element:

a[0] = 5; a[1] = 8; a[2] = 11; etc.

a[0] identifies the memory location and the = 5; assigns the value to the memory location.

If we try to do a[10] = 35, then Java throws an ArrayIndexOutOfBounds exception, because for this array the last usable index is 9.

(THIS IS EASY ISN’T IT! )

[KEEP GOING]It can be easier to use a loop to store values in an array because we can use the loop control variable as the index of the location where we want to store the value.

Let’s say for some reason we wanted the int values 5, 8, 11, 14, 17, 20, 23, and 26, stored in an array.

First we need to declare an array and this time the array will be named nums. So we use the following code to construct the array:

int [ ] nums = new int [8];

Then, to fill the array we just need assignment statements (even though we could read from the keyboard): (Did you notice the pattern in the numbers above?)

int num = 5;

for (int i = 0; i < nums.length; i++) // note that nums.length will hold the value 8 and the loop will run the

{ // correct number of times.

nums[i] = num; // Note the loop control variable i is in the [ ] representing the index

num += 3;

}

The array nums has a **physical** length of 8 (the size it was instantiated to). If we use nums.length then we access the value 8. In our loop, we want to use nums.length, not 8, because we may decide to change our code and instantiate an array of a different size for nums, and we don’t want to have to change the loop code.

**To print the values** of the array using a loop with 1 line of code rather than using 10 separate System.out.println statements, we could use the following:

for (int i = 0; i < nums.length; i++)

{

System.out.println(“The value at index ” + i + “ is ” + nums[i] );

}

Note: this code assumes the array is full of values, therefore we can use nums.length as the upper limit of the loop.

(Now let’s learn about how to copy values from one array to another.)**Copying.**

If there are two arrays a and b of the same size and type and the array **a** already contains values and **b** may or may not contain values (it doesn’t matter), then we can copy the values **from** **array a to array b** as follows:

for (int i = 0; i < a.length; i++)

{

b[i] = a[i];

}

We could have used b.length in place of a.length above, but it doesn’t matter which one we use since the arrays are both the same size. Either way the loop would run the correct number of times.

Sometimes we want to track the **logical size of an array**. If a program begins to fill an array with values but doesn’t fill it all the way, we need to know where it stopped filling the array so other values can be added later in the unfilled positions or so if we go to print the values we won’t print garbage values. For this example, we will assume that the array b has not been filled. To perform the copy, we use a separate integer variable that counts how many locations in the array have been filled. This variable then holds the index of where the next value needs to be added. Consider this code that copies only the even numbers from array a to array b:

int evenCount = 0;

for (int i = 0; i < a.length; i++) // use a loop to inspect all values in the array named a

{

if ( a[i] % 2 == 0 ) // checking to see if a[i] is evenly divisible by two

{

b[evenCount] = a[i]; // evenCount holds the index of the next available location to store a value

evenCount ++; // update evenCount so it holds the correct value for the next add

}

}

Here is now what the two arrays look like after the even ints in array a have been copied to b:









If we didn’t use evenCount and used i in its place, then array b would look like this with gaps!





If you want to generate random integers between 50 and 100 inclusive, what do you do?

50 100

- 50 - 50 subtract 50 from both numbers

0 50 multiply by 51 for it to be inclusive

then you use (int) (Math.random() \* 51) + 50 add back the 50 you subtracted

If you want random integers between -50 and -25 inclusive, then …

-50 -25

+ 50 + 50 add 50 to both numbers

0 25 multiply by 26 for it to be inclusive

then you use (int) (Math.random() \* 26) - 50 subtract off the 50 you added