**Java Arrays and Loops**

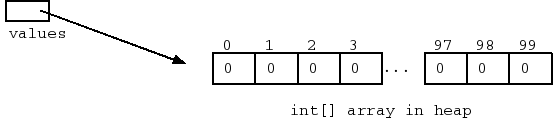
This page introduces arrays and loops in Java with example code, on creating, accessing, and looping with arrays. See also the associated CodingBat [java array problems](http://codingbat.com/java), to practice array ideas or study for an exam. Written by [Nick Parlante](http://codingbat.com/about.html).

An "array" is a way to store a collection of "elements". Arrays use square brackets [ ] for their syntax. Each array is declared with a type that indicates the type of element that it stores, like this: the "int[]" type is an array of int elements, a "String[]" type is an array of string elements. Arrays are allocated in the heap with the "new" operator and accessed through pointers (just like objects). The following line declares a variable "values" that can point to an int array. The code does not allocate the array yet:

int[] values; // declare an int array variable "values"

The expression "new int[100]" allocates a new array in the heap, sized to hold 100 int values.:

values = new int[100]; // allocate the array, store the pointer

In the heap, the array is a block of memory made of all the elements together. Each element in the array is identified by a zero-based index number: 0, 1, 2, ... and so on. An attempt to access an index outside the 0..length-1 range will fail with a runtime exception. When first created, all of the elements in the array are set to zero. So the memory drawing for the code above looks like:

Arrays use square brackets as a convenient syntax to refer to individual elements: "values[2]" is the element at index 2 within the "values" array. This square bracket syntax is an easy way to get or set any particular element in the array. Here is a longer example that declares an array variable, allocates the array with "new", and then uses the [ ] syntax to access particular elements:

int[] values; // declare in array variable

values = new int[100]; // allocate the array (initially all zero)

values[0] = 4; // store a 4 into element 0

values[1] = 8; // store an 8 into element 1

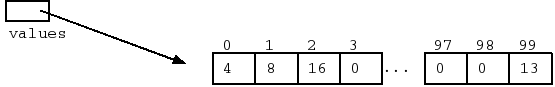
// read ints out of values[0] and values[1], do

// some math, store result into values[2]

values[2] = 2\*values[0] + values[1];

// store a 13 into the last element in the array

values[99] = 13;



The length of an array is set when it is created with "new" and that length never changes for the lifetime of the array. In contrast, Java Lists can grow and shrink over time -- this is a big feature that Lists have that arrays do not. The length of an array can be accessed as a special ".length" attribute. For example, with the above "values" array, we can access the size of the array as "values.length". It is common to use a 0...length-1 for-loop to iterate over all the elements in array:

int[] values = new int[100];

// Loop over all the elements in the values array

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

// Do something with values[i], such as print it

System.out.println( values[i] );

}

Note that the the syntax to access the length is just "values.length", not "values.length()". Why do we need .length? Don't we know how big the array is from when we created it -- length 100 or whatever? The answer is that .length is useful, since it allows us to write a method that takes an array parameter, where the array is of any size. For example, the method computeSum() below takes an int[] array argument, and computes the sum of the ints in the array:

public int computeSum(int[] nums) {

int sum = 0;

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

sum = sum + nums[i];

}

return sum;

}

Since the code uses the .length of the array passed in, it will work for any size array that we pass in.

**Example Array Code**

Here is code that iterates over a non-empty array to find the index of the smallest element:

int findSmallest(int[] values) {

int minIndex = 0; // start with 0th element as min

for (int i=1; i<values.length; i++) {

if (values[i] < values[minIndex]) {

minIndex = i;

}

}

return minIndex;

}

**Object Array Code**

Suppose we have an Account class that has a constructor with zero arguments. Here's some example code that allocates an array of 100 Account objects:

Account[] accounts;

// 1. allocate the array (initially all the pointers are null)

accounts = new Account[100];

// 2. allocate 100 Account objects, and store their pointers in the array

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

accounts[i] = new Account();

}

// 3. call a method on one of the accounts in the array

account[0].deposit(100);

**Array Literals**

There's a special syntax to specify an array and its elements right in the code using curly braces { ... }. This is called an "array constant" or an "array literal". This is a very convenient form if you know what all the elements are at compile time.

String[] words = new String[] { "hello", "foo", "bar" };

int[] squares = new int[] { 1, 4, 9, 16 };

**Array Code Patterns**

A "pattern" or "idiom" is a phrase or section of code that we see or use often since it solves a common problem. With arrays, there are a few very common patterns for iterating over the elements in an array. You want to be familiar with these basic patterns since many array problems can be solved with variations and combinations of these basic patterns.

**1. For-All Loop**

The "for-all" or "for-each" loop, as we have already seen, is a very common idiom for arrays. The goal is to iterate over all the elements from 0 to length-1, doing some operation once for each element.

// For-All

// Do something for every element

public void forAll(int[] nums) {

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

System.out.println( nums[i] );

}

}

We can change the init/test/increment in the for-loop to iterate through the elements backwards, from length-1 down to 0.

// For-All variant

// Go through the elements backwards

// by adjusting the for-loop

public void forAllBackwards(int[] nums) {

for (int i = nums.length-1; i>=0; i--) {

System.out.println( nums[i] );

}

}

With this sort of code, we have to be careful with the exact specification of the for-loop to hit the right elements in the array. It's easy to be off-by-one -- for example, the backwards loop above has to init i to "length-1" instead of "length" to start with the last element. The test should be "i>=0" instead of "i>0" to include the 0th element. Oftentimes in loop and array problems, getting these "edge" cases correct is the trickiest part. When writing a loop, look at the code and think how it will behave for the very first and very last elements. If it works for those, it will most likely work for all the elements in between. In Java, if the running code tries to access an index that is too big or too small to fit in the array, the program will get an ArrayOutOfBounds exception on that line. Other examples of for-all operations that look at every element in the array include: sum up the int values in an array, call a method on every object in an array, count the number of times a particular value appears in an array.

**2. Search Loop**

Another common problem is searching through all the elements in an array to find a particular "target" value. The code is basically the standard for-all loop with an if-statement inside looking at each element. The example code below takes array and target arguments and searches through the array for an element that is the same as the target. The method returns the index where the target appears in the array, or -1 if the target does not appear in the array.

In the loop, an if-statement checks if each element is the same as the target value. If the test is true we have found the target and can stop looking. In the example below, the if-statement uses a "return" to exit the loop and method immediately when a matching element is found. Another solution might use a "break". The point is, once the target is found, we do not need to keep iterating the loop.

How do we know if the target does not appear in the array? This requires a little logic. The search-loop looks at every element in the array, tests each one, and exits the method when a match is found. Therefore, the only way the program can get to the line just after the search-loop is if none of the elements matched. Therefore, the code just after the search-loop only runs when no elements match. In this example, the "return -1;" will run in the case that none of the elements matched.

// Search

// Searches through the given array looking

// for the given target. Returns the index number

// of the target if found, or -1 otherwise.

// (classic search-loop example)

public int search(int[] nums, int target) {

// Look at every element

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

if (nums[i] == target) {

return i; // return the index where the target is found

}

}

// If we get here, the target was not in the array

return -1;

}

For search problems, we generally write the loop in the standard way to go through all the elements, and then add if/break/return logic inside the loop to react to each element. As a matter of style, the search-loop solves two problems. It must iterate over all the elements, and it must figure out if the target is found or not. One of those problems – iterating over all the elements -- is just the standard for-all problem, and we are happy to use the standard pattern code to handle it. Or put another way, if part of a problem is common and part is unique, use standard code for the common part and add in code for the unique part. The strength of standard code is, after all, that it is familiar and easy to use, so we use it where we can.

What would it mean to disregard the above advice? We could make up a while-loop that mixes together the logic of the for-all and the search into one:

// Another way to write search that combines

// the "end of array" and "found" logic in one

// while loop. As a matter of style, we prefer

// the above version that uses the standard

// for-all loop.

public int searchNotAsGood(int[] nums, int target) {

int i = 0;

while (i<nums.length && nums[i]!=target) {

i++;

}

// get here either because we found it, or hit end of array

if (i==nums.length) {

return -1;

}

else {

return i;

}

}

The above loop works, but I think it is not as good as the earlier version that uses the standard for-all loop. The beauty of the for-all idiom is that it is easy to write, easy to read, and hard to get wrong. We should take advantage of that clarity where we can.

**Heads/Tails Array Loop Example**

Here is a more complex search loop problem. Given the results of a series of heads/tails coin tosses as a boolean[] array (true=heads, false=tails), figure out if there were ever 10 heads in a row. This is essentially the search problem again: we go through the array looking for a something, stopping if we find it:

// Given an array of booleans representing a series

// of coin tosses (true=heads, false=tails),

// returns true if the array contains anywhere within it

// a string of 10 heads in a row.

// (example of a search loop)

public boolean searchHeads(boolean[] heads) {

int streak = 0; // count the streak of heads in a row

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

if (heads[i]) { // heads : increment streak

streak++;

if (streak == 10) {

return true; // found it!

}

}

else { // tails : streak is broken

streak = 0;

}

}

// If we get here, there was no streak of 10

return false;

}

**3. Min/Max Loop**

A Max or Min loop searches through all the elements looking for an element that is the most or least among all the elements according to some metric. The max-loop differs from the search-loop since the max-loop must look at every element, while the search-loop can exit as soon as it finds a match.

The basic idea with the max-loop is to keep a local variable that stores max-so-far as we go through the loop, and each element is compared to the max-so-far. If an element is larger than the max-so-far, its value becomes the new max-so-far. At the end of the loop, we have looked at every element and remembered the largest.

What should the initial value of max-so-far be before we look at anything? One possibility is to just use the first value from the array. Although probably not the largest in the array, it will be compared to every element by the time the loop is done, so it's a fine starting candidate.

// Find-Max

// Given a non-empty array of ints, returns

// the largest int value found in the array.

// (does not work with empty arrays)

public int findMax(int[] nums) {

int maxSoFar = nums[0]; // use nums[0] as the max to start

// Look at every element, starting at 1

for (int i=1; i<nums.length; i++) {

if (nums[i] > maxSoFar) {

maxSoFar = nums[i];

}

}

return maxSoFar;

}

A slightly different strategy is to use an extremely small value as the initial max-so-far, such as the constant Integer.MIN\_VALUE which is the most negative possible int value, about -2 billion.

// Find-Max variant

// Use very small number, Integer.MIN\_VALUE,

// as the initial max value.

public int findMax2(int[] nums) {

int maxSoFar = Integer.MIN\_VALUE; // about -2 billion

// Look at every element

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

if (nums[i] > maxSoFar) {

maxSoFar = nums[i];

}

}

return maxSoFar;

}

Finding a max or min value in an array does not make sense if the array is empty (length of zero) -- there are no elements, so the question of which element is biggest does not make sense. Sometimes, a method will stipulate that empty arrays may not be passed in and the method does not promise to behave correctly given an empty array. The burden is on the caller not to pass in an empty array. Sometimes, the method may back up this stipulation by actually detecting an "illegal" argument and raising a descriptive exception, such as the IllegalArgumentException. How do the two versions of findMax() above behave if passed an empty array? (The first will crash with an ArrayOutOfBounds exception. The second will return the value -2 billion.) A final variant of find-max returns the index of the max element in the array, rather than the value of the max element. The easiest solution is probably to use two local variables -- one for the max value and one for the index.

// Find-Max variant -- rather than finding the max value, find the

// \*index\* of the max value

public int findMaxIndex(int[] nums) {

int maxIndex = 0; // say the 0th element is the biggest

int maxValue = nums[0];

// Look at every element, starting at 1

for (int i=1; i<nums.length; i++) {

if (nums[i] > maxValue) {

maxIndex = i;

maxValue = nums[maxIndex];

}

}

return maxIndex;

}

[CodingBat.com](http://codingbat.com/) code practice.