Arrays

An *array* is an indexed sequence of contiguous slots in memory, all of the same type. The type of the elements is called the *base type* of the array. The indexes are numbers, starting with 0.

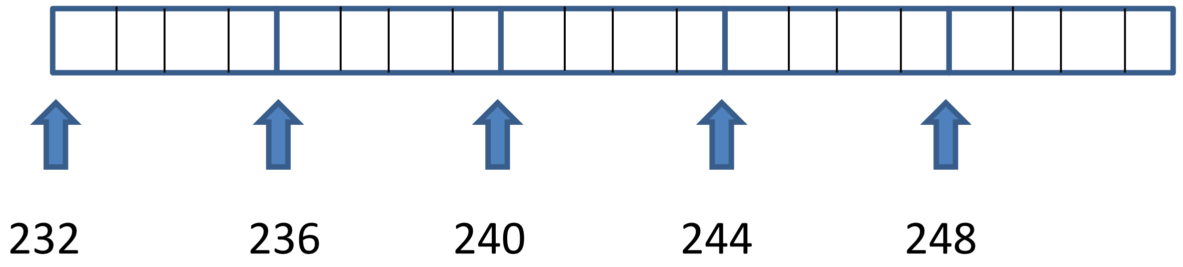
If you are used to lists in Python you can think of an array as a fixed-length list, though it is actually more than that.

Everything in Java has a type. The type of an array of elements that have base type T is T[ ]. For example, we would declare A to be an array of ints with

```
int [ ] A;
```

All arrays with the same base type have the same type (the length is not part of the type).

The fact that arrays are contiguous is very important. Suppose we have an array of (4-byte) integers starting at address 232:



The index-0 element of this array is stored at its starting address 232 and occupies the bytes 232, 233, 234, and 235. The index-1 element is stored at address 236, the index-2 element at 240, and so forth.

In general, the index-n element is stored at address 232+n*4

The important fact to remember about arrays is that the system can retrieve the element at any index in an array in one step. This is not true for lists or most other structures.

Back to Java.

We *construct* an array of N elements with base type T with

```
new T[N];
```

Altogether, a typical line of Java code that declares and constructs an array of 10 Strings is

```
String [ ]  page =  new String[10];
```

You can then refer to page[0], page[1], etc. up to page[9].

A common error is to declare an array but forget to construct it. For example:

```
int [ ] A;
A[0] = 23;  // There is no memory allocated
            // for A.
```

You can't refer to any of the elements of an array until you have actually constructed it. This works:

```
int [] A;
A = new int[100];
A[0] = 23;
```

If you have a small number of specific values you want to put into an array, there is another way to construct it.

```
        int [ ] A = {23, 45, 67};
```

creates an array of 3 elements and puts 23 into A[0], 45 into A[1], and 67 into A[2]. This is only practical for initializing small arrays, but it is sometimes useful.

Clicker Question:

What will this code do:

```
int [ ] A = new int[3];
A[0] = 11;
A[1] = 13;
A[2] = 17;
A[3] = 19;
for (int i = 0; i < 3; i++)
        System.out.println( A[i] );
```

A. It will print 11, 13, and 17 on different lines
B. It will print 11, 13, 17, and 19 on different lines
C. It will print 11, 13, 17, 19 and change the length of A to 4.
D. It will crash and burn.

Answer D: As soon as you try to access A[3] your program will crash.

You can find the length (number of entries) of any array A with A.length Remember that this is just the allocated size of the array; nothing says that all of those entries have useful data. Your program needs to manage the data in your arrays.

If you are trying to count the instances of each letter 'a' through 'z’ in some file, you might declare an array

```
int[ ] Counts = new int[26];
```

initialize all of the entries of Counts to 0

```
for( int i = 0; i < 26; i++)
        Counts[i] = 0;
```

and each time you see an instance of the ith letter increment Counts[i].

On the other hand, if you are using an array Primes to keep track of the prime numbers you have found, there is nothing particularly meaningful about indexes you use. Here you probably want to keep a large array Primes:

```
        int[ ] Primes = new int[1000];
```

and also have a variable to keep track of how many entries of this array you are currently using:

```
        int size;
```

If (size == 0) you haven't yet found any primes; otherwise the entries of Primes from index 0 to index size-1 are the primes you have found.

When you find a new prime p you add it to the array and increment size:

```
Primes[size] = p;
size += 1;
```

Your program will crash if you try to access an entry of the array beyond its length, so test for this:

```
if (size < Primes.length) {
    Primes[size] = p;
    size += 1;
}
```

One more clicker question: what will this print?

```
int [] A = new int[10];
size = 0;
size++;
A[size] = 11;
size++;
A[size]= 13;
for (int i = 0; i <=size; i++)
        System.out.println( A[i] );
```

A. It will print 11 13 on one line.
B. It will print 11, then 13 on separate lines.
C. It will print 0, 11, 13 all on separate lines.
D. It will print 11, 13, 0 all on separate lines.

Answer: It will print 0, 11, 13. If you look closely at the print loop it will print A[0], A[1], and A[2]. The code before the print loop sets A[1] to 11 and A[2] to 13; it doesn’t set A[0] to anything. Java initializes all of the entries of a new array to 0, so A[0] will be 0 when it is printed.