

# CS1Ah Lecture Note 16

## Arrays and Bounded Iteration

### 16.1 Using arrays

Consider the following simple program for setting up and printing out the first five prime numbers. Lines 3 to 7 of the program define five integer variables which are given the values of the first five prime numbers. Lines 8 to 12 of the program print out the values of these variables.

```
1  class Primes {
2      public static void main(String[] args) {
3          int p0 = 2;
4          int p1 = 3;
5          int p2 = 5;
6          int p3 = 7;
7          int p4 = 11;
8          System.out.println(p0);
9          System.out.println(p1);
10         System.out.println(p2);
11         System.out.println(p3);
12         System.out.println(p4);
13     }
14 }
```

The program is very simple, and it will work, but it would be better if we could make clear that the variables `p0` to `p4` were part of a related structure. Such a related structure is an array, which can be *indexed* to access particular numbers. We can define an array `p` which will store the first five prime numbers.

	<code>p[0]</code>	<code>p[1]</code>	<code>p[2]</code>	<code>p[3]</code>	<code>p[4]</code>
<code>p</code>	2	3	5	7	11

Arrays are indexed from zero. We access the elements of the array using a notation where the index is written after the name of the array and it is enclosed in square brackets. For example, we would use `p[0]` to access the zeroth entry in `p`, which has the value 2. We would use `p[4]` to access the last entry in the array, which has the value 11.

An array is an ordered collection of entities of the same kind. These might be values of the same type or they might be objects of the same class. An array has a fixed length. The array `p` has length 5. The index is said to be “out of bounds” if it is too small or too large. For the array `p` the allowable range of index values is zero to four. Using an value outside this range as an index into the array is an exceptional case (throwing an `ArrayIndexOutOfBoundsException`).

The time taken to access or update a value stored at some index in an array is small and independent of the position of the index. Later we will consider other data structures for storing sequences of elements (for example, *linked lists*) where accesses or updates of a value at a position given by a numerical index can take time proportional to the distance of the position from the start of the sequence.

The following program has the same effect as the previous one, it prints the first five prime numbers.

```

1  class Primes2 {
2      public static void main(String[] args) {
3          int[] p = new int[5];
4          p[0] = 2;
5          p[1] = 3;
6          p[2] = 5;
7          p[3] = 7;
8          p[4] = 11;
9          int i = 0;
10         while (i < p.length) {
11             System.out.println(p[i]);
12             i++;
13         }
14     }
15 }
```

On line 3, variable `p` is declared to be of type `int[]`, an array of `ints`, and is initialised to the value `new int[5]`, an array of length 5. Arrays in Java, like objects, are always handled by using references, so, strictly speaking, we should say that variable `p` stores a reference to the array created when the expression `new int[5]` is evaluated. Lines 4 to 8 show how array elements can be updated by using array index notation on the left-hand side of assignments, and line 11 shows array index notation used for accessing array elements. In addition, line 10 shows how to find the length of an array.

## 16.2 Initialising Arrays

When arrays are created using `new`, each element is automatically initialised to some default value of the element type. In the case of `int` arrays, the default value is 0. It is common to want to specify initial values for each element of a new array, so Java provides special syntax. Using this syntax, we can rewrite the previous example as:

```

1  class Primes3 {
2      public static void main(String[] args) {
```

```

3         int[] p = { 2, 3, 5, 7, 11 };
4         int i = 0;
5         while (i < p.length) {
6             System.out.println(p[i]);
7             i++;
8         }
9     }
10 }

```

When Java evaluates the expression `{2, 3, 5, 7, 11}`, it creates a new array in memory of length 5, initialises each element to the values given. The resulting value of the expression is a reference to this new array.

## 16.3 The for loop

In addition to the **while** loop construct seen previously, Java provides a control structure called the **for** loop. The previous example rewritten with a **for** loop is:

```

1  class Primes4 {
2      public static void main(String[] args) {
3          int[] p = { 2, 3, 5, 7, 11 };
4          for(int i = 0 ; i < p.length ; i++) {
5              System.out.println(p[i]);
6          }
7      }
8  }

```

The general form of a **for** loop is:

```

for(init; cond; update) {
    body
}

```

The parts are as follows:

**The initialisation** *init*: This typically defines a *loop control variable* whose value changes on each iteration of the loop. The initialisation part of line 4 of the example is `int i = 0`, which defines the `int` variable `i` and gives it an initial value of 0. The initialisation statement in the header of a **for** loop is only executed once.

**The condition** *cond*: This typically uses the value of the loop control variable to determine whether or not the work of the loop is complete. In the example we test whether or not the index has reached the end of the array. The index `i` must be less than the length of the array (written as `i < p.length`).

**The loop body** *body*: Line 5 of the example is the *body* of the **for** loop. The body is executed repeatedly while the condition evaluates to true. If the loop condition is false initially then the loop body will not be executed at all.

**The update** *update*: After each execution of the body of the **for** loop the loop control variable is updated. Frequently this is a simple adjustment to the value held by the loop control variable. In the example above *i* is incremented (*i++*).

The flow of control in a **for** loop is illustrated in Figure 16.1.

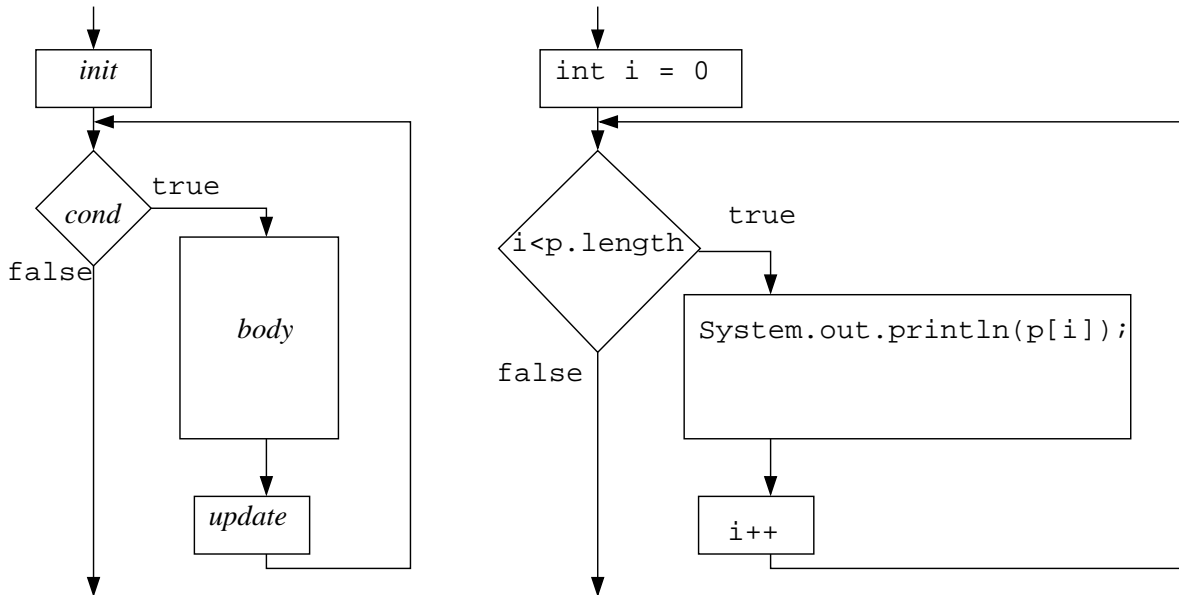


Figure 16.1: Flowchart for the **for** loop

The initialisation, the condition and the update make up the header of the **for** loop. Together these define where the loop starts, when it stops, and how it gets there. The intention of the **for** loop is that this information is neatly collected together in one place (line 4 of the example above). It is possible to assign new values to the loop control variable within the body of the **for** loop but this rather reduces the usefulness of the **for** loop because the update statement in the header of the loop ought to tell a reader of the program all they need to know about how the loop makes progress in steps towards the completion of its work.

Because **for** loops typically specify some pre-determined number of cycles round a loop, we usually use them for *bounded iteration*, repeating something a limited number of times by counting. By contrast, the **while** loop is usually used when the number of iterations is determined by a condition, and may even be *unbounded* (loops forever).

In fact, the difference is one of convention and good style only; the **for** loop in Java can be used to do anything that can be done with the **while** loop. Similarly the **while** loop can be used to do anything that can be done with the **for** loop.

## 16.4 Further prime numbers

The program which we have so far makes only limited use of arrays. We can show their usefulness in a better light by implementing an algorithm to generate the prime numbers up to any given limit. The algorithm is the Sieve of Eratosthenes:

- Collect all of the numbers from 2 to the limit.

- Step through the numbers in turn sieving out multiples.
- The numbers remaining are all primes.

An implementation in Java is:

```

1  class Sieve {
2      public static void main( String[] args) {
3          final int N = 1000;
4          boolean[] sieved = new boolean[N];
5          for ( int i = 2 ; i < N ; i++) {
6              if (sieved[i]) continue;
7              System.out.println(i);
8              for ( int j = 2 ; i * j < N ; j++) {
9                  sieved[i * j] = true;
10             }
11         }
12     }
13 }
    
```

The implementation makes use of an array `sieved` of boolean values. Every element of this array is initially the value `false`, since this is default value for `boolean` arrays.

As we loop through the numbers from 2 (the smallest prime) to `N` (the limit) we test to see if the number has previously been sieved out. If so, we *continue* from the point of updating the loop control variable and we do not execute the rest of the loop body. If not, we have a prime number which we can print out (line 7) before we continue to remove all of its multiples from the sieve (lines 8 to 10). The general flow of control for the `continue` statement in loop bodies is shown in Figure 16.2. This figure also shows

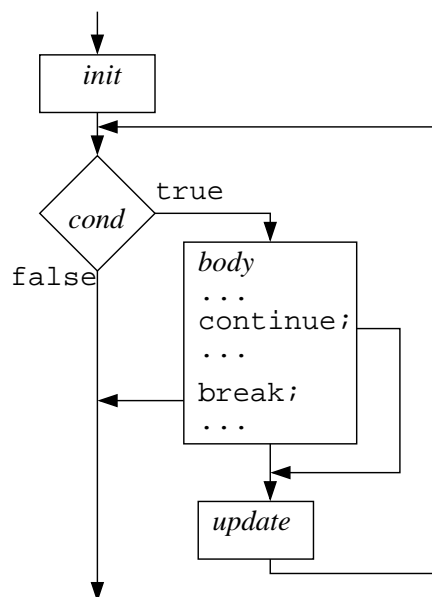


Figure 16.2: Flowchart showing effect of `continue` and `break`

how the `break` statement provides a means of immediately exiting a `for` loop.

## 16.5 Multi-dimensional arrays

An ordinary array stores a single row of data. Sometimes we want to organise data elements in tables, storing several rows together, or perhaps even several tables together. To do this we can use a *multi-dimensional* arrays. Multi-dimensional arrays in Java are treated as arrays of arrays. To access the  $j$ th (column) element of the  $i$ th (row) element of a two-dimensional array  $q$ , we write  $q[i][j]$ .

Given the array of primes as before, we might construct a table of multiples of primes which begins like this:

	$q[0][0]$	$q[0][1]$	$q[0][2]$	$q[0][3]$	$q[0][4]$
$q[0]$	2	3	5	7	11
	$q[1][0]$	$q[1][1]$	$q[1][2]$	$q[1][3]$	$q[1][4]$
$q[1]$	4	6	10	14	22

Treating multi-dimensional arrays as arrays of arrays has the advantage that elements can be of different sizes: arrays do not have to be rectangular, as the next example program shows. Do you know the triangle it prints?

```

1  class Triangle {
2      public static void main( String[] args) {
3          int[][] triangle = maketriangle(15);
4          for (int i = 0; i < triangle.length; i++) {
5              for (int j = 0; j < triangle[i].length; j++) {
6                  System.out.print(triangle[i][j]);
7                  System.out.print(" ");
8              }
9              System.out.println();
10         }
11     }
12     static int[][] maketriangle(int rows) {
13         int[][] t = new int[rows][];
14         for (int i = 0; i < t.length; i++) {
15             t[i] = new int[i+1];
16             t[i][0] = 1;
17             if (i>0) {
18                 int j = 1;
19                 for ( ; j < t[i-1].length; j++) {
20                     t[i][j] = t[i-1][j-1] + t[i-1][j];
21                 }
22                 t[i][j] = 1;
23             }
24         }
25         return t;
26     }
27 }

```

## 16.6 The Arrays class

Arrays are a commonly used container used to hold items in sequence. Because the order of items is apparent, useful actions can be performed on arrays which manipulate the order. The chief example is *sorting*, a ubiquitous procedure for which computer scientists have studied and invented many algorithms. Some easy-to-use implementations of sorting are provided in the Arrays Java library class:

```

1  import java.util.Arrays;
2  class ArgsSort {
3      public static void main(String[] args) {
4          Arrays.sort(args);
5          for (int i=0; i<args.length; i++) {
6              System.out.println(args[i]);
7          }
8      }
9  }
```

On line 4 the static method `Arrays.sort` is invoked on the array `args`, which sorts it into ascending order. For strings, the order is the *lexicographic* (or dictionary) order, where shorter strings should appear before longer ones which they prefix: `a`, `as`, `asp`, `aspect`, etc.

Another ubiquitous problem in computing is *searching*. The obvious way to search an array is to examine each element in turn to see if it is equal to the one being searched for (the *key*), from the start of the array until the end. This is called a *linear search*. For a large array, a linear search can be time consuming. We can do much better if we know something about the order of the elements in the array beforehand. On a sorted array, we can perform a *binary search* which repeatedly divides the part of the array being searched in two. The Arrays class has a method to do this:

```

1  import java.util.Arrays;
2  class ArgsSearch {
3      public static void main(String[] args) {
4          String key = args[0];
5          args[0] = "";
6          Arrays.sort(args);
7          int pos = Arrays.binarySearch(args, key);
8          if (pos >= 0) {
9              System.out.println("Found.");
10         } else {
11             System.out.println("Not found.");
12         }
13     }
14 }
```

The `ArgsSearch` program searches for the string given as the first command line argument in the remainder of the arguments. To avoid copying the array into a new

array to do the search, the first element of `args` is made empty on line 5. Without this line, the search would always succeed! Then the array is sorted on the next line, using `Arrays.sort` again. On line 7, we call the `Arrays.binarySearch` method to search for the key in the sorted array.

We will examine sorting and searching algorithms in more detail in later lectures.

## 16.7 Array equality, filling and copying

Like objects, arrays are handled in Java using *references*. This means that an array variable actually holds a *pointer* to a memory position where the array begins. If `a` and `b` are different names for the same array (they point to the same location in memory), then `a == b`. If `a` and `b` are different arrays, then we can test whether they have the same length and contents with `Arrays.equals(a,b)`.

The Sieve example began with an array of booleans, and relied on the default initialization which makes every element `false`. If an array `a` is to be filled with some other value `v`, the method `Arrays.fill(a,v)` saves writing the obvious loop.

Sometimes we want to make a copy of an array, or a part of an array, in another array. The recommended way to copy from `a[i]...a[i+n-1]` into `b[j]...b[j+n-1]` is with the method call `System.arraycopy(a,i,b,j,n)`. Again this saves writing a loop.

*Authored by Stephen Gilmore, David Aspinall and Paul Jackson.  
Paul Jackson, 2002/11/12 09:11:45.*