CS 4: Lecture 10
Wednesday, February 22, 2006

"for" Loops

"for" loops are a convenient shorthand that can be used to write some "while" loops in a more compact way. The following "for" loop is equivalent to the following "while" loop.

```
for (initialize; condition; next) {       |    initialize;
  statements;                             |    while (condition) {
}                                         |      statements;
                                          |      next;
                                          |    }
```

"for" statements are usually used to iterate while advancing an index variable over a fixed range of values. The following code repeats a statement 100 times.

```
int i;
for (i = 0; i < 100; i++) {
  System.out.println("I will not make Java serve detention for me.");
}                                          // Loop body ends at this "}".
```

A common mistake among beginning Java and C programmers is to get the conditional wrong and do one loop iteration too few. For example, suppose you want to print all the prime numbers in the range 2...n.

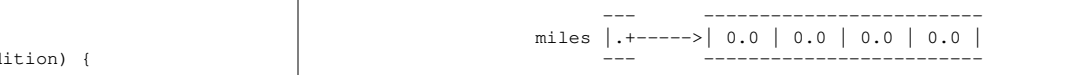
```
for (int i = 2; i < n; i++) {      // ERROR!!!  Condition should be i <= n.
  if (isPrime(i)) {
    System.out.print(i + "is prime.");
  }
}
```

Suppose we correct this method so the loop test is "i <= n". Think carefully: what is the value of i when the loop ends?

MORE ARRAYS

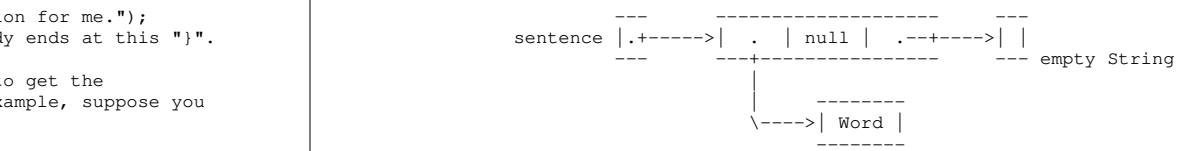
When you construct an array, Java automatically initializes all its values to zero (or false for an array of booleans).

```
double[] miles = new double[4];
```



You can declare an array of objects--but it's really an array of references. Java initializes these references to a value of "null", which means that they don't point to anything. You can set them to point to objects, but they all have to be of the class you used in the definition.

```
String[] sentence = new String[3];
sentence[0] = "Word";
sentence[2] = new String();
```



You can use fields or methods on objects in an array by combining the array notation with the dot notation.

```
sentence[1] = sentence[0].concat(sentence[0]);
```

main()'s Parameter

What is the array of Strings that the main() method takes as a parameter? It's a list of command-line arguments sent to your Java program. In Gild, you can supply these through a menu item. If you have a command-line Java (most common in Unix), you type them on the command line. Consider the following program.

```
class Echo {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++) {
      System.out.println(args[i]);
    }
  }
}
```

If we compile this and type "java Echo kneel and worship Java" on a Unix command line, java prints

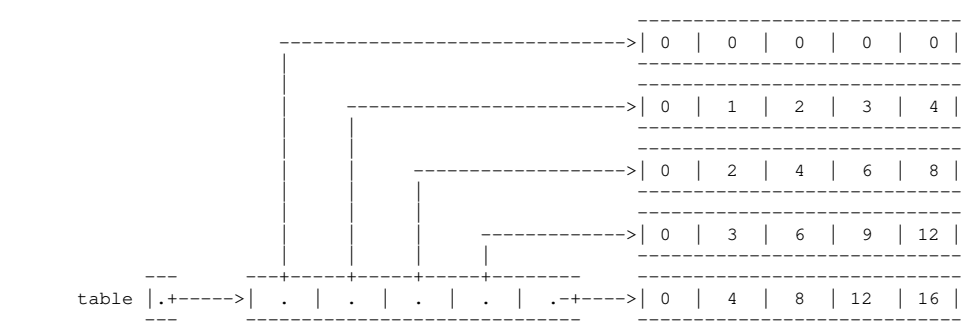
```
kneel
and
worship
Java
```

Multi-Dimensional Arrays

A _two-dimensional_array_ is an array of references to arrays. Java can construct an array of arrays for you with one command. Let's construct a multiplication table.

```
long[][] table;            // table's type is "array of arrays of longs".
table = new long[x][y];    // Construct an array of x arrays of y longs each.
for (int i = 0; i < x; i++) {
  for (int j = 0; j < y; j++) {
    table[i][j] = i * j;
  }
}
```

The array objects look like this:



It's important to understand the order of the array indices. Does table[i][j] mean we look up the jth element of the ith array, or vice versa? It might help you to keep in mind that you can address just one index at a time.

```
long[] row = table[3];
```

Now "row" references an array of longs--the array stored at index "3" of the array of arrays. Then the following two lines access the same long.

```
z = row[1];
z = table[3][1];             // Both lines access the same array element.
```

Think of the dereferencing operation "[1]" as an operation on the array that precedes it, which in this case is "table[3]". You could even write

```
z = (table[3])[1];
```

to remind yourself that we look up index 3 first, then index 1. This is perfectly valid Java code.

You can also construct a three-dimensional array--an array of arrays of arrays--or a four-dimensional or whatever-dimensional array.

```
long[][][][] tensor = new long[3][4][5][2];
```

But if you try to construct a high-dimensional array where each dimension of the array is large, you'll find you run out of memory pretty fast. A 100-by-100-by-100-by-100 array of longs takes about a gigabyte to store.

It's convenient that Java will construct all the arrays for you, at every dimension. But arrays are the only class of object Java will do this for. If you construct an array of other types of objects, like Planet objects, Java will just construct an array of null references; you'll have to write a loop to construct the Planets themselves.

Initializers

When you declare a variable that references an array, you can also construct arrays and initialize array entries by using _initializers_.

```
long[] a = {7, 12, 5, -3};
String[] b = {"milk", "fish", new String()};
int[][] c = {{7, 3, 2}, {x}, {8, 5, 0, 0}, {y + z, 3}};
```

The third line constructs a two-dimensional array:

Observe here that two-dimensional arrays don't have to be rectangular--each array can be a different length.

Alas, you can only use initializer notation in a declaration. Java will proclaim a compiler error if you try to write

```
d = {3, 7};
f({1, 2, 3});
```

How could you loop through an array like "c" above? Since each row has a different length, you need to write a loop that looks up each row's length. Let's look at a nested loop that prints out an array as a String.

```
public static String arrayToString(int[][] array) {
  String out = "[ ";
  for (int i = 0; i < array.length; i++) {
    out = out + "[ ";
    for (int j = 0; j < array[i].length; j++) {
      out = out + array[i][j] + " ";
    }
    out = out + "] ";
  }
  return out + "]";
}

System.out.println(arrayToString(c));
```

This code prints the array like this:

```
[ [ 7 3 2 ] [ 9 ] [ 8 5 0 0 ] [ 6 3 ] ]
```