Programming review, July 14

Matthew Meisel

Significant portions of this material are modified from *Surverying the Field of Computing*, 4th ed., by Carl Burch, 2002.

Arrays

An **array** holds a sequence of values of the same type. Arrays are useful to use when storing related data. Each individual value in the array is called an **array element** or, informally, a **component**.

An array is declared similarly to a single variable. An array declaration uses the following format.

<typeOfElement>*`[]` *<variableToDefine>;

For example, the following creates a variable `score` of type `double` that can name an array of numbers.

```
double[] score;
```

We need an extra step when we declare arrays to reserve a chunk of memory for our array, and we do this with the `new` keyword.

```
score = new double[3];
```

Sometimes, it is convenient to combine the above two statements into one. This allows us to declare an array and reserve memory for it in the same step.

```
double[] score = new double[3];
```

To work with an array, we must refer to individual elements using their **array indices**. When an array is created with the code above, the array elements are automatically numbered from 0 up to one less than the array length.

If you declare an array of length 3 (like in the above example), the indices of the array will be 0, 1, and 2. If you try to access an undefined array index, there will be a **run-time error** called and `ArrayIndexOutOfBounds Exception` and your program will crash.

You can assign a value to an array element using a simple declaration statement.

```
score[0] = 23.4;
```

Alternatively, any kind of numerical expression can be coded inside the brackets.

```
Std.out.println("How long to you want your array to be?");
int length;
length = Std.in.readInt();

double[] testArray;
testArray = new double[length];

Std.out.println("Enter a number 0 through" + (length-1) );
int index;
index = Std.in.readInt();

Std.out.println("Enter an int to be placed in that index of the array");
testArray[index] = Std.in.readInt();
```

Finally, Java provides a useful technique for finding the length of an array. The statement `array.length`, where `array` is the name of your array, will return the length of your array.

```
Std.out.println("The length of your array is " + testArray.length);
```

Notice that the length is always an integer.

The `for` loop

A **`for` loop** is a type of loop in Java. It is meant for executing a sequence of statements *for* every value in a set, especially for iterating over some statements *for* every integer in a range. A `for` loop in Java has the following format.

```
for (<initialAssignment>; <conditionalStatement>; <updateAssignment>) {
        <statementsToRepeat>;
}
```

The first time through the loop, the initial assignment is executed. Then, the conditional statement is evaluated. If the conditional statement is true, the statements to repeat are executed, and then the update assignment is, too. The conditional statement is rechecked, and the loop is repeated as long as the conditional is true. If it is false, the loop "breaks," and the program continues with the code after the loop.

Remember that when we test equality inside a conditional statement, we must use the double equals sign (`==`). The single equals sign is used to assign a variable a value, not to test equality.

The following code prints out the first ten perfect squares.

```
for ( int x=1; x<11; x++ ) {
       Std.out.println( (x*x) );
}
```

Notice that we declare and assign the **counter variable** `x` inside the loop itself. Because we defined it inside the loop, the variable is destroyed as soon as the loop is finished, and we cannot access it again.

The symbol `x++` means "add 1 to the value of `x`." Similarly, `x--` means "subtract 1 from the value of `x`." The addition of the `++` and `--` operators to the language C was one of the more minor changes in the creation of C++, but, nevertheless, that is how C++ got its name.

Exercise: Using a `for` loop, write a program that prints the first ten powers of 2. Recall that the Java method `Math.pow(a,b)` returns the value of (a^b).

Using `for` loops and arrays together

Often times, programmers use `for` loops and arrays together do perform the following functions: 1) assign a value to each element of an array, or 2) read each value of an array. In the following example, we read each value of an array using a `for` loop.

Example: Write a method that takes an array and prints out every array element.
Solution:

```
private static void printArray ( double[] array ) {

       for ( int c=0; c<array.length; c++ ) {
              Std.out.println(array[c]);
       }

       return;

}
```

Notice that our method is **void**: it *does* something, but it doesn't *return* anything. Therefore, when we write `return` at the end of our method, we don't write anything after it. The keyword `return` only signifies that it is the end of the method. Void methods are called using a single statement, whereas methods that return a value must be called as part of a large statement.

```
printArray(a);       // void method: does not return a value, so it is
                     // written as a statement alone

int m = max(a,b);    // non-void method: returns a value, so is part of
                     // another statement (in this case, a variable
                     // declaration)
```

Now we will write a program that performs the other function of a `for` loop with an array: to assign a value to each element of an array.

Exercise: Write a program that creates an array of the first *n* even numbers. Asks the user how large they want their array to be. Then, fill the array using a `for` loop.

Searching through an array

Sometimes, it might be useful to find the *location* of a certain element in an array. The simplest way of doing this is to look through the array, element by element, and check to see if the given element is the same as the one we are looking for. The following method takes as a parameter two things: an array of type `int` and an integer in the array that we wish to find. It returns, as an integer: the first index where we find that value. If we don't find that value, it returns -1 (which obviously cannot be an index).

```
private static int findNum ( int[] array, int numToFind ) {

       for ( int c==0; c<array.lenth; c++ ) {
              if (array[c] = numToFind) {
                     return c;
              }
       }
       return -1;

}
```

It is important that your methods always return *some* value. In this case, our method needs to return a value even if it doesn't find the given number. Your methods should always return a default value if the method doesn't do what it's supposed to, and that value—a negative number, in this case—should make it clear to the program that something in the method went wrong.

Notice that our method uses multiple `return` statements. Because the method can only return one value, the first time it encounters a `return` statement, that value is returned and the entire method—along with the `for` loop—is terminated.

Exercise: Write a method that takes as its parameters an integer array and an integer to find (just like above). However, this method should return the *number of times* the given number appears in the array. You will need to create a variable inside the method to count how many times the number appears. Set it to 0 to begin with, and add one to it each time you come across the given number! (What value will the method return if it doesn't find the given number?)

Tricks with `while` loops

We have already seen the basic format of a `while` loop.

```
while (<conditionalStatement>) {
        <statementsToRepeat>
}
```

The first time through the loop, the conditional statement is evaluated. If it is true, then the statements to repeat are executed. Then, the conditional statement is checked again, and, if it is still true, the statements to repeat are executed again. The loop continues until one of two things happens: 1) the conditional statement becomes false, or 2) the keyword `break;` is reached. (You can also use the keyword `break;` inside a `for` loop).

Consider the following code.

```
int c = 2;
while ( c<2000 ) {
        Std.out.println(c);
        c = c*2;
}
```

The first time through the loop, the conditional statement is true: `c`, which equals 2, is less than 2000. The program prints `c`, then multiplies `c` by 2. We have reached the end of the statements to repeat, so the conditional is evaluated again. This time, `c` equals 4, which is still less than 2000, so the statements inside the loop are again repeated. This will continue as `c` grows, but eventually `c` will become greater than 2000, and at the point the loop will stop.

There are a number of tricks you can use with `while` loops. For example, if you want a program to loop for ever, you can use code like this.

```
while ( true ) {
        <statementsToRepeat>
}
```

Because the conditional statement will always be `true` (the keyword true always evaluates as true!), the loop will repeat forever (unless, of course, it comes across the `break;` statement in the loop).

You can also use a `while` loop to guarantee that your user's input has some special property. For example, suppose you want to the user to input an integer that is at least 1, and if the user doesn't input an integer greater than 1, you want to ask them again.

```
int number = 0;
while ( number<=0 ) {
        Std.out.println("Please input an integer greater than 0");
        number = Std.in.readInt();
}
```

We assign number a default value that violates our own condition, and then we write the conditional statement of the while loop that is the *opposite* of the condition we are looking for. Therefore, the loop will execute the first time through, and it will continue to execute as long as the input is illegal. As soon is the input is *legal*, the loop will end—and we have legal input!

Exercise: Write a program that asks a user for an odd integer and continues to ask them for an odd integer until they input one.