Repetition, Looping
CS101

Last time we looked at how to use if-then statements to control the flow of a program. In this section we will look at different ways to repeat blocks of statements. Such repetitions are called loops and are a powerful way to perform some task over and over again that would typically be too much work to do by hand. There are several ways to construct loops. We will examine the while and for loop constructs here.

While Loop

The while loop allows you to direct the computer to execute the statement in the body of the loop as long as the expression within the parentheses evaluates to true. The format for the while loop is:

```
while (boolean_expression) {
        statement1;
        …
        statement N;
}
```

As long as the Boolean expression evaluates to true, statements 1 through N will continue to be executed. Generally one of these statements will eventually make the Boolean expression become false, and the loop will exit. Here is an example that prints the numbers from 1 to 10:

```
int x=1;

while (x<=10) {
        System.out.println(x);
        x++;                    // Same as x=x+1
}
```

If we wanted to print out 1,000,000 numbers we could easily do so by changing the loop! Without the while loop, we would need 1,000,000 different print statements, certainly an unpleasant task for a programmer.

There are two types of while loops that we can construct. The first is a *count-based* loop, like the one we just used above. The loop continues, incrementing a counter each time, until the counter reaches some maximum number. The second is an event-based loop, where the loop continues indefinitely until some event happens that makes the loop stop. Here is an example of an event based loop:

```
int sum=0, x=0;

while (x!=-9999) {
        x = Console.readInt("Enter an integer");
        if (x != -9999) {
                sum = sum + x;
        }
}
System.out.println("The sum of the numbers you entered is " + sum);
```

This loop will input a number and add it to sum as long as the number entered is not –9999. Once –9999 is entered, the loop will exit and the sum will be printed. This is an event-based loop because the loop does not terminate until some event happens – in this case, the special value of –9999 is entered. This value is called a *sentinel* because it signals the end of input. Note that it becomes possible to enter the sentinel value as data, so we have to make sure we check for this if we don't want it to be added to the sum.

What is wrong with the following code? Hint: It results in what is called an infinite loop.

```
int x=1, y=1;

while (x<=10) {
        System.out.println(y);
        y++;
}
```

Exercise: Write a program that outputs all 99 stanzas of the "99 bottles of beer on the wall" song.

For example, the song will initially start as:

```
99 bottles of beer on the wall, 99 bottles of beer,
take one down, pass it around,
98 bottles of beer on the wall.
```

Write a loop so that we can output the entire song, starting from ninety-nine and counting down to zero. Try to work this out on your own, before looking at the program on the next page.

Here is a working program to solve this exercise:

```
class Beer
{
 public static void main(String[] args)
 {
     int numBeers = 99;
     while (numBeers > 0) {
        System.out.print(numBeers + " bottles of beer on the wall, ");
        System.out.println(numBeers + " bottles of beer,");
        System.out.println("take one down, pass it around,");
        System.out.println(numBeers-1 +
                                " bottles of beer on the wall.");
        numBeers = numBeers - 1;
    }
 }
}
```

Example: What is the output of this code?

```
            int j=0, k=0, x=5;

            while (j<x) {
                  System.out.print("*");
                  j++;
            }
            System.out.println();         // Prints a newline
            j=0;
            while (j<x-2) {
                  System.out.print("*");
                  k=0;
                  while (k<x-2) {
                        System.out.print(".");
                        k++;
                  }
                  System.out.println("*");
                  j++;
            }
            j=0;
            while (j<x) {
                  System.out.print("*");
                  j++;
            }
            System.out.println();
      }
```

Let’s trace through this code line by line, keeping track of the value in each variable as we go. When we start, j=0, k=0, and x=5.

The first while loop continues while (j<x). 0 is less than 5, so the Boolean condition is true and we enter the loop and we output a single *. J is incremented to 1 and then we return to the top of the while loop. 1 is still less than 5, so we continue and output

another *. J is incremented to 2, and then we return to the top of the while loop. 2 is still less than 5, so we continue and output another *. J is incremented to 3. 3 is less than 5 so we continue and output *. J is incremented to 4. 4 is still less than 5 and we output another *. J is incremented to 5. Finally, 5 is no longer less than 5 (it is equal to five) so we exit the first while loop.

This loop has just made our program output five asterisks: *****

The code now continues by resetting j to 0.
The next loop continues while j < x-2. In this case j=0 and x=5. So we compare 0 < 3, which is true.

The program outputs another *. So far we have output:

```
*****
* ←  Cursor is here
```

The next chunk of code is another loop:

```
k=0;
while (k<x-2) {
      System.out.print(".");
      k++;
}
```

This code should look familiar – it is almost identical to the first while loop that we examined that printed out five *'s. However, in this case it is looping from k=0 up to k=x-2. Since x=5, it is looping from k=0 to k=3 and will output three "."'s.

Finally the program prints out another * with a newline to give:

```
*****
*...*
```

Next, the program increments j by one. We now have j=1, k=3, and x=5. The program returns to the top of the second while loop and compares j to x-3. In this case, it compares 1 < 3 which is true, so we continue the loop and repeat the above process another time to get:

```
*****
*...*
*...*
```

At the bottom of the loop, j is incremented again to 2. We compare 2 < 3 which is true, so we continue the loop and repeat the above process one more time to get:

```
*****
*...*
*...*
*...*
```

We increment j one more time to 3. Now when we return to the top of the loop, we compare 3 < 3 which is not true and we skip the inner loop and drop down to the last while loop.

The last while loop is:

```
j=0;
while (j<x) {
      System.out.print("*");
      j++;
}
```

This continues while j < x. Initially, j =0 and x=5. Since 0 < 5 the condition is true, so we enter the loop. Notice that this is now identical to the very first while loop we encountered, which will print out five *'s. The complete figure printed out by this program is then:

```
*****
*...*
*...*
*...*
*****
```

This last example illustrates the concept of nested loops. It is possible, and often desirable, to insert one loop inside another loop. When this is done, it is called a nested loop.

Here are some sample exercises for you to try. See if you can do them yourself before looking at the solutions on the next page.

Exercise: Write a program that inputs from the user how many numbers she would like to enter. The program should then input that many numbers and compute the average of the numbers.

Exercise: Write a program that finds and prints all of the prime numbers between 3 and 100. A prime number is a number such that one and itself are the only numbers that evenly divide it (e.g., 3,5,7,11,13,17, …)

Here is pseudocode we developed to see if a number "n" is prime:

```
CHECK IF N IS PRIME:
    if n = 1 then stop, return back "PRIME"
    x ← 2                // The arrow means assign the number 2 to x
    while (x < n)
        if (n mod x) = 0 then stop, return back "NOT PRIME"
        x ← x + 1
    return back "PRIME"
```

Here is pseudocode to output all prime numbers between 3 and 100:

```
PRIMES 3-100:
    ctr ← 3
    while (ctr <=100)
        Check if ctr is prime using above algorithm
        If prime, print ctr
        cr ← ctr + 1
```

Putting these two pieces together yields the following algorithm:

```
PRIMES 3-100:
    ctr ← 3
    while (ctr <= 100)
        IsPrime ← TRUE          // Assume that ctr is prime
        x ← 2
        while ((x < ctr) && (IsPrime is TRUE))
            if (ctr mod x) = 0 then IsPrime = FALSE
            x ← x + 1
        if (IsPrime is TRUE) then print ctr
        ctr ← ctr + 1
```

Here is a solution for Average, there are many other equally valid ways to write a valid solution!

```
class Average
{
 public static void main(String[] args)
 {
        int howMany;
        double temp;
        double ave;
        double sum=0;
        int i;

        howMany = Console.readInt("How many numbers to enter? ");
        i=1;
        while (i <= howMany) {
                temp = Console.readDouble("Enter Number " + i + ": ");
                sum = sum + temp;
                i = i + 1;
        }
        ave = sum / howMany;
        System.out.println("The average is " + ave);
 }
}
```

Solution for prime numbers:

```
class Prime
{
 public static void main(String[] args)
 {
        int ctr = 3;
        int x = 2;
        boolean isPrime;

        while (ctr <= 100)
        {
                isPrime = true; // Assume current value in ctr is prime
                x = 2;
                while ((x<ctr) && (isPrime==true))
                {
                        // See if we can show value in ctr is not prime
                        if ((ctr % x)==0) {
                                isPrime = false;
                        }
                        x = x + 1;
                }
                // If isPrime is still true, then ctr is prime
                if (isPrime) {
                        System.out.println(ctr + " is prime!");
                }
                ctr = ctr + 1;
        }
 }
}
```