Controls Structure for Repetition

So far we have looked at the if statement, a control structure that allows us to execute different pieces of code based on certain conditions. However, the true power of computers lies in their ability to repeat a set of instructions many times in a short period of time. Many algorithms contain directives to repeat a designated set of steps an arbitrary number of times. In Java we have a simple loop structure, called a while statement, which allows us to repeat a piece of code multiple times. Here is the general syntax of a while statement:

```
while (<boolean expression>) {

        stmt1;
        stmt2;
        ...
        stmtn;
}
```

This statement is executed in the following manner:

1) First the boolean expression is evaluated.
2) If it is true, execute statements 1 through n in order.
3) If it is false, skip all the statements inside the while loop braces({}).
4) Go back to step #1.

In essence, you check your boolean condition each time before you decide whether or not to execute the statement in the loop. You continue doing so until you check the condition and it is false.

What we can see from this construct is typically, for it to be effective, the boolean condition must first be true, and eventually become false. Thus, if the boolean condition was in terms of a variable, such as:

```
while (x < 10)  {
...
}
```

For this to be an effective loop, the value of x must change in the body of the loop. If it does not, then the condition will always evaluate to the same thing.

The big problem with this is if it happened to be true to begin with, it would never stop!!! This is known as an infinite loop. One of the most common problems that beginning programmers have when designing loops is having them run forever, so to speak. We will talk about how to detect these later.

Now lets consider trying to write a program to do the following:

1) Prompt the user if they would like to enter a test score.
2) If so, reading it in and then asking them for another test score.
3) Continue the process above until the user says no.
4) Output the average of the test scores entered by the user.

Before we get down to writing this program, let's analyze how we will need to lay everything out.

To compute an average, we must know two pieces of information:

1) The sum of the numbers
2) The number of numbers being averaged.

Notice that we don't actually need to know each individual number. So the idea that we have is the following:

Create two variables, one to store the sum of the numbers and one to store the number of numbers.

Furthermore, we will also need a variable to read in the answer of the user as to whether or not they want to enter another score. Finally, we will also need a variable to read each individual score into.

In general, we want to continue reading in scores until the user says they do not want to enter any, so our general layout will look like the following:

```
<initialize variables>
while (<user says they want to enter another score>) {

        <ask user for score>
        <adjust necessary variables
        <check to see if they want to enter another score>
}
```

The process we have gone through is basically the algorithmic design process. Notice that we haven't truly written any code yet, but we have a very good idea of how we are going to solve this problem.

Whenever you are given a problem to write a program for, you should always go through this algorithmic design process. You want to identify the variables you will need, as well as the general flow of execution. That way, once you actually write your program, it will be much easier to debug(which is simply a fancy word for fix) it.

So, without further due, here is the program:

```
public class Average {

   public static void main(String args[]) {

      Scanner stdin = new Scanner(System.in);

       // Declare and initialize all necessary variables.
      char answer;
      int sum, score, numscores;
      double average;
      numscores = 0;
      sum = 0;

      // Read in whether or not user would like to enter a score.
      System.out.println("Would you like to enter a score?(y/n)");
      answer = (stdin.next()).charAt(0);

      // Continue reading in scores as long as user wants.
      while (answer == 'y') {

         // Get next score.
         System.out.println("Enter the next score.");
         score = stdin.nextInt();

         // Update necessary variables.
         sum = sum + score;
         numscores = numscores + 1;
```

```
            // Prompt user if they want to enter another score or not.
            System.out.println("Would you like to enter a score?(y/n)");
            answer = (stdin.next()).charAt(0);
        }

        average = sum/numscores;
        System.out.println("The average of your scores is" + average);
    }
}
```

Now, there are actually a couple subtle problems with the program above. The first is that even though the statement average = sum/numscores is mathematically correct, two things can go wrong with this statement:

1) If numscores is 0, then we are dividing by a 0, which is a big no no!!!
2) average will always be set to an integer because the quotient of two integers is defined to be an integer in Java as well.

How can we solve the first problem?

Notice that our English write-up of the problem looks a lot like Java code. We can simply use an if statement to check the value of numscores.

How about the second problem?

We know that Java will correctly divide a double by an integer, so we can change our assignment statement to:

average = 1.0*sum/numscores.

The expression on the right is evaluated left to right. 1.0*sum is going to give us a double as an answer, equal to sum. Then this double will get divided by numscores, also giving us a double, which will be the correct answer. So, here is our fix for the last section of our code:

```
if (numscores > 0) {
    average = 1.0*sum/numscores;
    System.out.println("The average of your scores is" + average);
}
else
```

System.out.println(“Sorry, you did not enter any numbers to average.”);

Let’s analyze our code to look for general types of constructs for particular solutions. First of all, the while loop we used is typically known as a sentinel-controlled loop. What this means is that the loop continues to run until some sentinel value, which is meant to indicate the end of the data, is read in. In this case, we simply prompted the user as to whether or not they wanted to enter another value each time. Our sentinel value was the character ‘n’. (Technically it was every character but ‘y’.)

This is a standard technique. Often times you will have to read in several pieces of data, and will be looking for some value that signifies the end of that data. We could have written this program in such a manner that we asked the user to enter all of their scores at once, followed by a –1. Since it is impossible for –1 to be a test score, this would have been the sentinel value we were “looking” for. In any event, in a situation like this, you simply use a while loop that runs until there is equality with the sentinel value.

The second standard technique introduced in this program is the user of an accumulator variable. Often times you will have to sum up a set of values. You can use an accumulator variable to do so. In this program the accumulator variable was sum. The way you typically use an accumulator variable is to initialize it to zero. Then, whenever you encounter a value you need to add into your sum, simply have a line with the following form:

sum = sum + <value>;

In essence, each time you execute a line of this type, you are taking the old value of sum, adding to it a value that you want to, and then storing that sum back into the accumulator variable. It is instructive to trace through a couple examples to convince yourself that this technique works.