Our Flounder

Computer Science Correspondence School

discere domi

Lecture 12

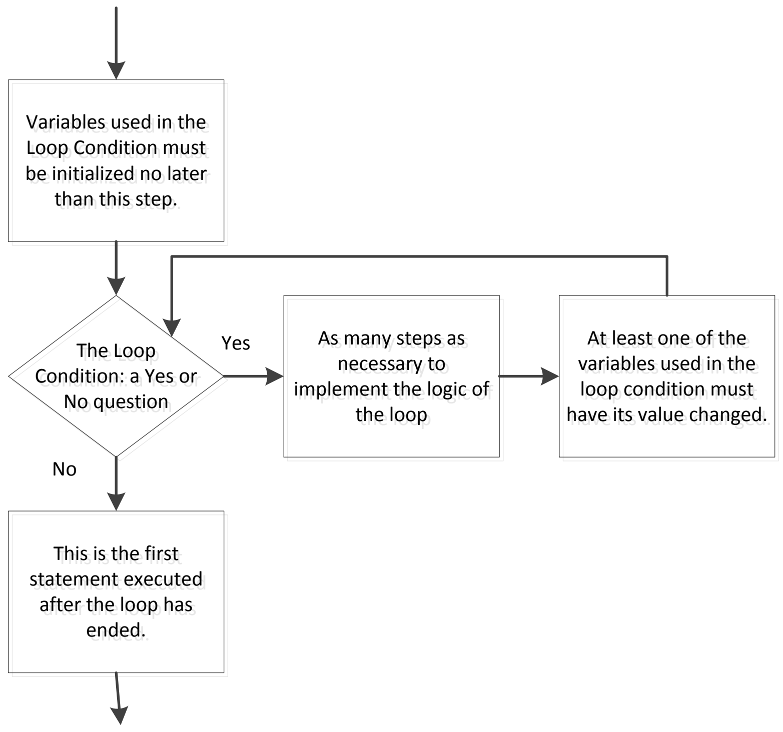
CISC 131
Introduction to Programming and Problem Solving
Spring 2020
for Loops

Due: Monday, April 20, 2020, at start of class
Points: none

Start of Today's Lecture

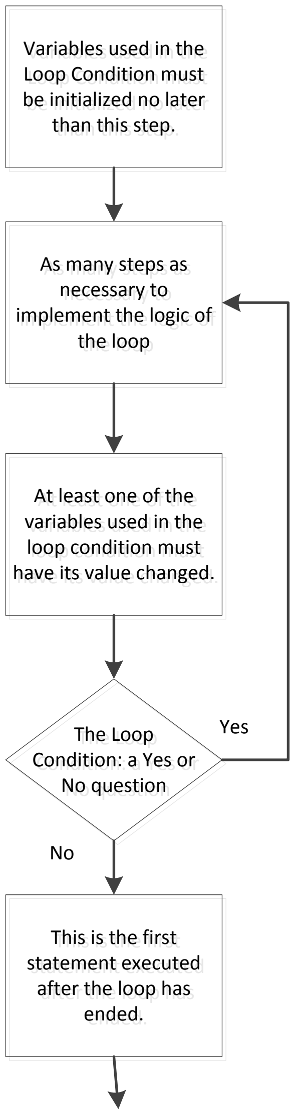
Loops are essential to programming. Each loop has a condition that, when it becomes false, causes the loop to terminate. If the computer were not able to repeat groups of instructions, it would be worthless. All modern languages have syntax that supports several ways to indicate a loop. Without regard to any particular language or syntax, all loops can take three possible forms:

- *Recursive,* in which a function calls itself either directly or indirectly. While this type is often a convenient way to describe an algorithm, it is usually not a good way to implement an algorithm. It can often make excessive use of system resources which can cause a program to abort. Any recursive loop can be converted into a non-recursive loop and *vice versa* although going from non-recursive to recursive can take some effort.
- A *pre-condition loop* is structured in such a way that the question (the loop condition) is asked *before* the loop body is executed. See flowchart below.


Variables used in the Loop Condition must be initialized no later than this step.
The Loop Condition: a Yes or No question
Yes
No
As many steps as necessary to implement the logic of the loop
At least one of the variables used in the loop condition must have its value changed.
This is the first statement executed after the loop has ended.

Pre-condition Loop

- A *post-condition* loop is structured so that the loop condition is evaluated *after* the loop body has been executed. See flowchart below


Variables used in the Loop Condition must be initialized no later than this step.
As many steps as necessary to implement the logic of the loop
At least one of the variables used in the loop condition must have its value changed.
The Loop Condition: a Yes or No question
Yes
No
This is the first statement executed after the loop has ended.

Post-condition Loop

The flowcharts shown above illustrate the two rules that should be followed when writing loops:

- All variables used in the loop condition must have been initialized[1] before the loop is reached
- At least one of those variables must have its value changed each time through the loop.

Pre-condition loops differ from post-condition loops in one important way: the statements in the body of a post-condition loop will always be executed at least one time. This is because the statements in the body are executed *before* the loop condition is evaluated. Because the statements in the body of a pre-condition loop are executed only *after* the condition has been

[1] Remember that *declaration* of a variable and *initialization* of a variable are very different things.

evaluated, the statements in the body of a pre-condition loop will be executed zero or more times.

Post-condition loops should be used only when there is a guarantee that the loop needs to be executed at least one time. This doesn't happen very often. Almost all of the loops that you will write will be pre-condition loops and, moreover, a pre-condition loop can be used in all circumstances.

The *while* loops that you have been writing are *pre-condition* loops. They are a good way of representing situations where you can't tell in advance how long the loop will need to run. For example, if the loop is counting the number of *html* elements that have the same *html id* prefix, you don't know when you write the program whether the loop will run zero times or a thousand times.

There are other situations where you can specify exactly how many times the loop run. This is almost always the case when processing characters in a *String* or elements in an array. The loop will, generally execute *length* times. In addition, these kinds of loops use a variable to identify the character number or array element that is being processed in the loop body. Each time through the loop, the next character or array element is processed. When a *while* loop is used for this kind of processing, the loop has three parts:

a. Statement(s) before the *while* statement that initialize all of the variables used in the loop condition.
b. The keyword *while* followed by the condition inside parentheses.
c. The body of the loop which must contains a statement that causes at least one of the variables used in the loop condition to have its value changed.

Conceptually, this makes the while loop look like this:

```
a while ( b ) { … c }
```

Most languages, including Javascript, have a useful and more concise syntax for expressing a loop that does this sort of processing and it is usually implemented with the keyword *for*. The *for* loop is a pre-condition loop just like the *while* loop is but the *for* loop has a more concise syntax.

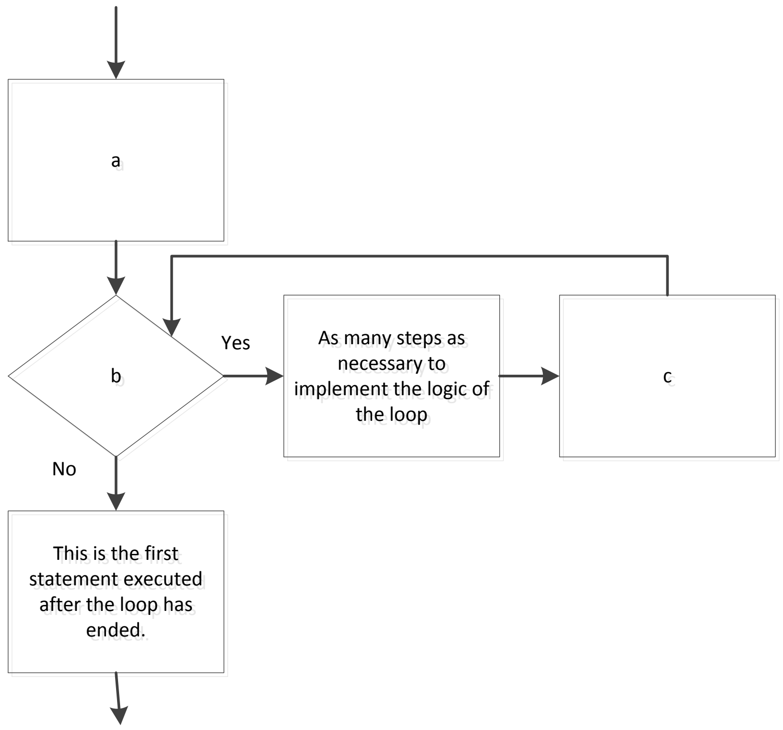
Using the parts a, b, and c shown above, the *for* loop syntax is:

```
for (a; b; c) { loop body }
```

Note that all three parts are (initialization, condition, and value change) are all contained inside the parentheses. The value change Any loop that can be written using a *while* statement can be written using a *for* statement and any loop written using a *for* statement can be written using a *while* statement.

The following flowchart shows how *a, b,* and *c* are used in the *for* loop. Note that the *c* part is done after all other statements in the body have been executed. It is important to realized that the *c* part of the *for* loop can be any assignment statement. Oftentimes the *c* part is an increment[2] but it is not limited to that. It could be anything. The statement that you put in *c* all depends on what you want to happen as the last statement in the body of the loop. Just make sure it is an assignment statement or your loop may be infinite.

[2] Adding one to someting


a
b
Yes
No
As many steps as necessary to implement the logic of the loop
c
This is the first statement executed after the loop has ended.

Here is an example comparing a *while* loop to a *for* loop. Both loops sum the integers from one through six.

```
var i;                                var i;
var sum;                              var sum;

sum = 0;                              sum = 0;
i = 1;                  // a          for (i=1; i<7; i=i+1)  // a,b,c
while ( i < 7 )         // b          {
{
 sum = sum + i;                        sum = sum = i;
 i = i + 1;             // c
}                                     }
```

Although either *while* or *for* can be used to implement any kind of logic, usually *while* loops are used when it is not known when the loop condition will become *false*. The *for* loop is usually used when doing character processing in a String or accessing array elements.

The following page contains some examples of functions implemented with a *while* and the equivalent function implemented with a *for*.

return a copy of the array parameter

```
function arrayCopyUsingWhile ( copyMe )
{
 var i;
 var result;

 result = new Array ( copyMe.length );

 i = 0;
 while ( i < copyMe.length )
 {
  result[i] = copyMe[i];
  i = i + 1;
 }

 return result;
}
```

```
function arrayCopyUsingFor ( copyMe )
{
 var i;
 var result;

 result = new Array ( copyMe.length );
 for ( i=0; i<result.length; i=i+1 )
 {
  result[i] = copyMe[i];
 }
 return result;
}
```

return the number of times *countMe* occurs in the *String* sourcd

```
function countUsingWhile ( source, countMe )
{
 var i;
 var count;

 i = 0;
 count = 0;
 while ( i < source.length )
 {
  if ( source.charAt(i) === countMe )
  {
   count = count + 1;
  }
  i = i + 1;
 }
 return count;
}
```

```
function countUsingFor ( source, countMe )
{
 var i;
 var count;

 count = 0;
 for ( i=0; i < source.length; i=i+1 )
 {
  if ( source.charAt(i) === countMe )
  {
   count = count + 1; }
  }
 return count;
}
```

As practice, take some of the array and *String* processing functions that you have written with *while* loops and re-write them with *for* loops. Both versions should produce the same results.

You might wonder why change from *while* to *for*. It has nothing to do with correctness. The *for* statement is more commonly used for array and *String* processing and it is what more advanced programmers would use. You are becoming a more advanced programmer, so incorporate the *for* statement into your programs.