Iteration

Preview of Coming Attractions

In this unit be sure to look for

- a five-step process for developing while statements
- tracing while statements

The Need to Get Your Act Together

As you probably know already, developing a while statement can require a great deal of care. Because the body of a while is executed repeatedly, a single mistake can come back to haunt us not just once, not just twice, but an untold number of times! That's serious.

In order to help keep you out of trouble, we'll explain a somewhat general process for developing while statements. The process is informal, intuitive, sensible, straightforward, and helpful. If you find yourself facing a particularly difficult while statement, fall back on the process and let it help you out.

A Process for Developing While Statements

1. Make sure you understand what is given to the while statement and what its goal is. (You should think of the goal as what the while statement is supposed to accomplish.) In particular, make sure you understand what must be true of variable values
    1.1 as execution first reaches the while statement
    1.2 upon successful completion of the while statement.
   (You might think of 1.1 as a precondition or "requires" for the while and think of 1.2 a postcondition or "ensures" for the while.)
2. Think of an action that, if repeated a sufficient number times, will accomplish the goal of the while statement. The action will eventually morph into the body of the while statement and may consist of several steps.
3. Think of an appropriate continuation condition to go with the action from step 2.

4. Write a skeleton of the while statement using an informal statement of the continuation condition from step 3 and an informal expression of the action, from step 2, for the body of the while.
5. Gradually refine the skeleton into the final while statement.

A First Example

Let's see how the process works on a real example. Informally, the problem is, given variable **`int`** `n` and variable **`double`** `approx`, assign to `approx` the summation of the first `n` terms in $(1/1)+(1/3)+(1/5)+(1/7)+\ldots$. To get started, notice that iteration is necessary here because the number of terms to be added is not fixed, but rather depends on the value of `n`.

Step 1 — Understand the Given and the Goal

Variables `n` and `approx` are given. `n` determines the number of terms to be added together. Let's assume that at least one term should appear in the summation; otherwise, there really isn't much to do. So we assume that `n >= 1` when execution reaches the while statement. For now, we really don't care what value `approx` has when execution reaches the while statement, because the while statement will produce a value for `approx`.

When the while statement is finished, let's ensure that `n` has the same value as before the while and ensure that `approx` has, for its value, the appropriate summation.

Step 2 — Think of an Appropriate Repeatable Action

We need to compute the sum of `n` terms. So, one obvious piece of the action will be to add another term to `approx`. But, the terms aren't there for the picking — they must be computed. Hence, another piece of the action will be to compute another term. Let's compute and add the terms in the order $(1/1)$, then $(1/3)$, then $(1/5)$, and so on. It looks like our action sequence is:

compute the next term and add it to `approx`

Step 3 — Think of a Continuation Condition

The action from step 2 can be repeated as long as there are additional terms to be computed and added to `approx`. So, the continuation condition will be something like

there are more terms to process

Step 4 — Write a Skeleton

Given the action sequence and continuation condition from steps 2 and 3, here's a simple skeleton:

```
while (there are more terms to process) {
    compute the next term and add it to approx;
}
```

Step 5 — Refine the Skeleton Into a While Statement

The skeleton does a nice job of getting us organized. Now we just need to add some flesh to the bones by growing some good code. Adding the terms, one at a time, to `approx` should be easy. The more interesting challenge is computing the next term. For this problem, a little notation defining the terms goes a long way: $t_1 = (1/1)$ defines the first term; $t_2 = (1/3)$ defines the second term; $t_3 = (1/5)$ defines the third term, and so on. Can you see the pattern? What defines the *i-th* term? It looks like for all $i \geq 1$, $t_i = (1/(2i - 1))$.

To take advantage of this insight, let's introduce a *new* variable for the while loop of type **`int`** and call it `termNum`. `termNum` will be used to keep track of the number of the current term being processed. To be very precise, we'll make sure that the while statement has the following key property:

The Key Property

> ***EVERY TIME*** execution reaches the continuation condition of the while statement, the value of `termNum` will equal the number of the *last* term processed; that is, the number of the last term computed and added to `approx`. In other words, every time execution reaches the continuation condition, `approx` will be the summation $t_1 + t_2 + t_3 + \cdots + t_{termNum}$.

The body of the while statement will need to make sure that the value of `termNum` is properly updated. And, the continuation condition now can be stated formally as `termNum < n`.

Putting these ideas together, here is the first version of the while statement:

```
while (termNum < n) {
   // adjust the value of termNum
   termNum = termNum + 1;

   // compute the next term and add it to approx
   approx = approx + (1.0 / (2 * termNum – 1));
}
```

We're almost there. The last detail concerns possibly setting the values of the variables just prior to the while statement. Remember the key idea: ***EVERY TIME*** execution reaches the continuation condition of the while statement the value of `termNum` must be the number of the last term computed and added to `approx` and `approx` must be the corresponding summation. ***This includes the very first time execution reaches the continuation condition, even before the body of the while statement has been executed even once.*** So, which term should be processed just before the very first time execution reaches the continuation condition? The simplest thing is to process term $t_1 = (1/1)$ *only* just prior to the while statement. This leads to:

A Final Solution

```
int termNum = 1;  // processing term t1
double approx = 1.0;  // approx = t1

while (termNum < n) {
   // adjust the value of term_num
   termNum = termNum + 1;

   // compute the next term and add it to approx
   approx = approx + (1.0 / (2 * termNum – 1));
}
```

Still Awake?

10-1. Explain why the value of `approx` should be set to 1.0 just prior to the while statement. Consider the key property.

10-2. Would it be okay to use `termNum <= n` for the continuation condition? Explain your answer.

10-3. Could the final solution start with `approx = 0.0` just before the while statement? If so, explain what other changes would have to be made to the final solution and why.

10-4. The final solution computes and sums the terms in the order $t_1, t_2, t_3, \ldots$. Develop another while statement that solves the same problem by computing and summing the terms in the order $\ldots, t_3, t_2, t_1$.

10-5. In the sequence of terms $(1/1), (1/3), (1/5), (1/7), \ldots$ the denominators are 1, 3, 5, 7, … . Suppose that we had an additional variable of type **`int`** named `denominator` and that we ensured that, every time execution reached the continuation condition, the value of `denominator` was equal to the denominator of the last term computed and added to `approx`. Develop another while statement that solves the same problem using the `denominator` variable.

A Second Example

The second example uses the `SimpleReader` and `String` components. Informally, for this problem, we are given a `SimpleReader` variable named `input` connected to a file that contains a sequence of number, string pairs, and the task is to count how many of these pairs have the property that the number is equal to the length of the string. Iteration is necessary here because the number of pairs to be considered is not fixed, but rather depends on the value of file to which `input` is connected.

Step 1 — Understand the Given and the Goal

Variable `input` is given and has been connected to a file that contains the number, string pairs. Let's assume that the format of this file is a sequence of lines organized as:

```
n1
string1
n2
string2
n3
string3
…
nk
stringk
```

for some `k >= 0`. (If `k=0`, then the file is empty; that is, the file contains no pairs.) Let's also assume that we have a variable of type **`int`** named `count` that we will use to store the number of pairs to be counted.

When the while statement is finished, let's ensure that all pairs have been read from `input` so that `input` is empty and ensure that `count` has, for its value, the number of (`n`, `string`) pairs where `n` = length of `string`.

Step 2 — Think of an Appropriate Repeatable Action

The repeatable action for this problem is fairly obvious

> get the next number and string from `input`
> increment `count` if the number matches the length of the string

Step 3 — Think of a Continuation Condition

The action from step 2 can be repeated as long as there are additional pairs to read from `input`; that is, as long as `input` is not empty. So, the continuation condition will be something like

`input` is not empty

Step 4 — Write a Skeleton

Given the action sequence and continuation condition from steps 2 and 3, here's a simple skeleton:

```
while (input is not empty) {
    get the next number and string from input
    increment count if number matches the length of string
}
```

Step 5 — Refine the Skeleton Into a While Statement

To get a number-string pair from `input`, we need two *new* variables, say `n` of type **`int`** and `str` of type `String`. Then, the statements

```
n = input.nextInteger();
str = input.nextLine();
```

can be used to get another number and string. The `length` operation for `String` variables can be used to check whether the pair satisfies the required property. The key property for our solution will be:

The Key Property

EVERY TIME execution reaches the continuation condition of the while statement, the value of `count` will be exactly the number of pairs that have been read from `input` and that satisfy the required property.

To state the continuation condition, we use the `atEOS` operation for `SimpleReader` variables. This operation is specifically designed to test for emptiness. Applied to `input`, the operation call is `input.atEOS()`, which you can read as "`input` is at the end of stream".

Here's the first version of the while statement:

```
while (! input.atEOS ()) {
   // get the next number and string
   n = input.nextInteger();
   str = input.nextLine();

   // check if n is equal to the length of str
   // and if it is increment count
   if (n == str.length()) {
      count = count + 1;
   }
}
```

Once again, we need to make sure that the key property holds the ***very first time execution reaches the continuation condition before the body of the while statement has been executed even once***. Just prior to the while, no pairs have been read from `input`, so `count` should be 0.

A Final Solution

```
int count = 0; // nothing counted yet

while (! input.atEOS ()) {
   // get the next number and string
   int n = input.nextInteger();
   String str = input.nextLine();

   // check if n is equal to the length of str
   // and if it is increment count
   if (n == str.length()) {
      count = count + 1;
   }
}
```

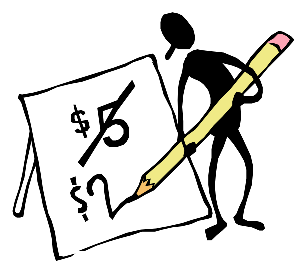
Still Awake?

10-6. If the variables `n` and `str` were declared before the while loop, what values should they be initialized to? Why?

10-7. Explain how you would modify the problem statement and solution if the input contained 3-tuples (instead of pairs), each 3-tuple had two integers and a string, and you needed to count how many 3-tuples have the property that the length of the string is between the two integers.

Tricky Tracing Ahead

Tracing while statements is a double-edged sword. Because the details of while statements can be quite intricate and difficult to get right, tracing on sample values would appear to be an attractive idea. However, because the values of variables can change each time the body of the while is executed, tracing while statements can be very tricky. Great care is needed to make sure that unexpected results during tracing are due to a mistake in the while-statement design and not due to erroneous tracing!



A systematic, though tedious way to trace while statements is to use the "cross 'em out" technique. For each iteration of the body of the while, the values of the variables from the previous iteration are crossed out and new values recorded beside the old. Here's an example, where variables `x`, `y`, and `m` are of type **`int`**. (Note: the '%' operator is the Java remainder operator. For instance, 10 % 3 is 1 because 10 divided by 3 leaves a remainder of 1.)

Statement	Variable Values
	x = 21 y = 15 m = 2917
while (y != 0) {	
	x = ~~21~~ ~~15~~ 6 y = ~~15~~ ~~6~~ 3 m = ~~2917~~ ~~6~~ 3
m = x % y;	
	x = ~~21~~ ~~15~~ 6 y = ~~15~~ ~~6~~ 3 m = ~~6~~ ~~3~~ 0
x = y;	
	x = ~~15~~ ~~6~~ 3 y = ~~15~~ ~~6~~ 3 m = ~~6~~ ~~3~~ 0
y = m;	
	x = ~~15~~ ~~6~~ 3 y = ~~6~~ ~~3~~ 0 m = ~~6~~ ~~3~~ 0
}	
	x = 3 y = 0 m = 0

Still Awake?

10-8. Complete the following tracing table. Variables `x`, `y`, and `q` are of type **`int`**.

Statement	Variable Values
	x = 103 y = 32 q = 0
while (x >= y) {	
	x = y = q =
x = x - y;	
	x = y = q =
q++; // q = q + 1	
	x = y = q =
}	
	x = y = q =