**Java For and While Loops**

This handout introduces the basic structure and use of Java for and while loops with example code an exercises. See also the associated CodingBat [java loop practice problems](http://codingbat.com/java) using strings and arrays. Written by [Nick Parlante](http://codingbat.com/about.html).

**Java Loop**

With loops, you get to leverage the power in the computer. Working with computers, you quickly learn that they lack any sort of insight to solve problems on their own. On the other hand, the computer is happy to execute the code we specify a million times over, which is its own sort of power. Loops are the way to do this.

A while loop has a boolean test and a body containing statements, like this:

int count = 0;

while (count < 100) { // test: boolean test within (..)

System.out.println("count:" + count); // body: statements within {..}

count = count + 1;

}

System.out.println("all done!");

An if-statement looks at the test one time and then maybe executes the body once. The while-loop extends this idea, executing the body again and again, checking the test each time. The while-loop follows these steps:

1. Check if the test is true or false. If false, the loop "exits" and does not execute the body. If the test is true, continue with step 2.
2. Execute the body statements, starting at the top and proceeding down through them all.
3. Go back to step 1 and check the test again. If it is true, run the body again. Each time the body finishes, "loop around" and check the test again. Each run of the body is called an "iteration" of the loop.

Eventually the test is false, the loop exits, and the program continues with the line after the while-loop. For example, the above while-loop prints:

count:0

count:1

count:2

count:3

...

count:98

count:99

all done!

The count variable starts at 0, and increases by 1 on each iteration. On the 100th iteration of the loop body, the body prints "99" and then increases count to 100. The program loops back to the top and checks the test (count < 100) again. On all previous checks, the test was true. This time the test is false, and the loop exits without executing the body. We use while-loops to execute some code until some condition is satisfied.

**How To Write a While Loop**

To write a while-loop, we think about three parts...

* **test** -- a boolean test of what should be true before each iteration. Or put another way, the test is the "green light" condition that says that each iteration can go ahead. (The phrase "green light" is a good mnemonic for what the test does.) Eventually, the test should become false and the loop can exit. Think about the precondition that describes the state before each iteration runs -- how are things arranged, what is true? (Also known as an "invariant".) In the above example, the test (count < 100) means that each iteration can proceed, so long as (count < 100) is true.
* **work** -- code in the body that does something for each iteration such as printing or drawing something. Some loops do not have any identifiable work in the body. In the above example, the work is the println() that prints something out for each iteration.
* **increment** -- code in the body that advances things so we are closer to making the test false. At the end of the body, the code will loop around to the top of the body. Sometimes, some cleanup is required at the end of the body to set things up for the next iteration. In the above example, the line "count = count + 1;" accomplishes the increment. That can also be written as "count++;".

**While Loop Example**

Here's a while loop example that uses a loop to see how man times you can divide a number by 2:

// Given a num, returns how many times can we divide it by 2 to get down to 1.

int count2Div(int num) {

int count = 0; // count how many divisions we've done

while (num >= 1) {

num = num / 2;

count++;

}

return count;

}

**Zero Loop Iterations**

Since the while-loop checks the test before every iteration, it is possible for the body to not run even once:

int count = 100;

while (count < 100) { // the test is false the very first time

count = count + 1 // this line never runs

}

**Infinite Loop**

The most famous sort of bug you can get with loops is an "infinite loop", where through some mis-arrangement, running the loop body fails to get any closer to making the test false. Typically, this comes down to some sort of logical disconnect between the body and the test. For example, suppose the body fails to make the count bigger by accident:

int count = 0;

while (count < 100) {

System.out.println("count:" + count);

count = count \* 1; // OOPS, does not change count

}

Or suppose we forget to put in the count line entirely, so the loop just spins in place with count stuck at zero:

int count = 0;

while (count < 100) {

System.out.println("count:" + count);

// OOPS, forget to put in a line that changes count at all

}

Prints when run:

count:0

count:0

count:0

count:0

...

--never stops--

In some cases, we want the code to run on and on without stopping. In that case, we might write an infinite loop on purpose by giving **true** as the test:

while (true) { // deliberate infinite loop

...

...

}

Many computer people like to think of themselves as part of a whimsical field that does not take itself too seriously compared to more established disciplines. For example, Apple Computer headquarters is located on "1 Infinite Loop, Cupertino CA".

**Loop and a Half - "break"**

Sometimes, the computation we want does not fit into the tidy while-test-body structure. For example, suppose we want to do operations A, B, C and D, but the natural place for the test is after B and before C. So the sequence we want is A, B, test, C, D. It's awkward to use the while loop as shown so far to solve this sort of problem. To position the test wherever we want, not just at the top, we will use the "break" statement, shown in the example below.

For example, suppose we have a Frog object that responds to dive() and float() methods that move the frog around in a pond. The frog also responds to an isSunny() method that returns true if the frog is in the sun. We want to write a "find sun" loop that does this: float, check if there is sun. If there is sun, we're done. Otherwise, dive and try the cycle again. So the sequence is float, sun check, dive, float, sun check, dive, .... until the sun check returns true. The key feature here is that the test does not naturally fit at the top of the loop. Instead the test most cleanly goes in the middle of the loop.

The code for this problem demonstrates uses an while-break structure. An if statement in the middle of the while loop checks for the exit condition. If the exit condition is true, the code calls "break" which exits the while loop immediately:

// suppose we have a variable "frog"

while (true) { // use "true" as test -- exit is handled by the if-break below

frog.float();

// Exit using if-break

// Check for exit condition, and if true call break

if (frog.isSunny()) {

break; // this exits the loop

}

frog.dive();

}

// The break exits the loop, leading to this line

With the while-break, we have the power of a while loop, but now we can position the test anywhere we want instead of just at the top. In fact, a standard while loop of this form:

while (*test*) {

*body*

}

Is equivalent to a while-break loop where the break is positioned at the very top:

while (true) {

if (!(*test*)) {

break;

}

*body*

}

As a matter of good coding style, we would not write a loop using while-break unless there was no other way. The standard while loop is shorter, more readable, and can express what we want 90% of the time. We use the while-break version where necessary to deal with cases where the test does not go at the top.

Notice that the normal while loop test is a "green light" condition -- if it is true, the loop continues; if it is false the loop exits. With a while-break, the test is the opposite. The while-break test is a "red light" condition. If it is true, the loop exits.

**For Loops**

The for-loop is a variant of the while-loop, specialized to deal with a particular looping problem. The for-loop is specialized for the case where we want to step a variable through a series of values. For example, the following for-loop steps the variable "i" through the values 0, 1, 2, ... 99:

for (int i=0; i<100; i++) {

System.out.println("i:" + i);

}

This loop prints:

i:0

i:1

i:2

i:3

i:4

...

i:99

The for-loop has four parts -- init, test, increment, and body -- separated by semi-colons (;)

for ( init ; test ; increment ) {

body

}

The for-loop is used specifically to step a variable through a sequence of values, such as 0, 1, 2, 3, .. 99. In fact, most for-loops adhere to that pattern so closely, that they will look similar to the 0..99 for-loop code above.

The for-loop follows four steps:

**1. Init**

The init code runs once to set things up at the very start of the loop. Typically, this looks like "int i = 0", declaring a variable to use for the counting, and setting its start value. The variable exists only within the loop (down through the } of the body).

It is idiomatic to use the variable name "i" for a counter variable. Although we avoid single-letter variable names for general purpose storage, this use of "i" is standard in a for-loop like this. Indeed, this pattern is so standard that other programmers would likely be a little confused if you used a variable name other than "i" in a for-loop. The convention extends to using the names "j", and "k" for loop variables if "i" is already in use. Never use lowercase L "l", since it looks just like the digit one "1" in many fonts.

**2. Test**

The boolean test is evaluated. If it is true the loop body will execute (green light, just like a while-loop). Typically, the test is a boolean expression such as "i<100", showing what should be true for the loop to continue.

**3. Loop-body**

If the test was true, the body runs once. The body is a series of statements, just like in a while-loop. The loop body will very often make some use of the loop variable, such as "i" in this example.

**4. Increment**

Finally, the increment code executes just after the body, and then the program loops back to the test, (step 2). The increment advances the state of things in preparation for the next iteration. The increment code advances the state, and then the test (2) looks at the state. Eventually, we advance the state so far that the test is false and the loop exits. This is analogous to the "increment" idea in the while-loop. In the while-loop, the increment is just a conceptual goal, but in the for-loop, the increment has its own slot in the syntax. When writing a for-loop, we are less likely to forget to do the increment, since the syntax has an explicit slot for it.

In this case, the increment is "i++" which increments the variable "i" by one. Since "i" is the common variable choice for a for-loop, it is very common for the increment to be "i++". It is idiomatic to write the increment as "i++" as opposed to the similar "i = i + 1".

**For Loop Examples**

The examples below demonstrate use the basic for-loop structure and change the init, test, and increment to get all sorts of different series: 0, 1, 2, ..98, 99 -or- 5, 10, 15, ..90, 95, 100 -or- 100, 98, 96, .. 2, 0.

/\*

Classic for-loop counting up from 0 to 99

0, 1, 2, 3, ... 99

-init int i = 0

-test i<100

-increment i++

\*/

for (int i=0; i<100; i++) {

System.out.println(i);

}

/\*

For-loop to print the values

0, 2, 4, 6, .. 98

-increment by i+=2, instead of i++

(same as i = i + 2)

\*/

for (int i=0; i<100; i+=2) {

System.out.println(i);

}

/\*

For-loop from 99 down to 0

99, 98, 97, ... 0

-init i=99

-test i>=0

-increment i--

\*/

for (int i=99; i>=0; i--) {

System.out.println(i);

}

/\*

For-loop from 0 to 100 by 5's

0, 5, 10, 15, .. 100

-test i<=100

-increment i+=5

\*/

for (int i=0; i<=100; i+=5) {

System.out.println(i);

}

**Test Style < vs. <=**

In the for-loop test, we can choose between < and <= -- which is the better way to write it? If the for-loop stops at 1 less than some nice round value, such as stopping at 99, then it's better to write the test using < and the nice round value -- "i < 100" -- instead of the equivalent "i <= 99". If the loop goes up to and includes the nice round value, then write the loop with <=. So a loop up to and including the value 100 would use the test "i<=100" instead of the equivalent "i<101".

**For vs. While Loops**

The for-loop can actually be understood as equivalent to a while-loop version, like this:

init

while (test) {

body

increment

}

So for example, we could write the standard 0..99 loop like this, although for real code we would prefer the for-loop in this problem:

int i = 0;

while (i<100) {

System.out.println(i);

i++;

}

As a matter of style, if we are solving a series-of-values 0, 1, 2, 3, ... 100 type problem, then the for-loop is preferable. That's exactly what it does. If the looping problem does not fit into the series-of-values pattern, then the standard while-loop is best.

**Loop With 2 Exit Tests**

When writing a loop that looks for something, we often have two exit conditions. Suppose as a sort of James Bond example, we have a Vault object with something locked up inside of it, and it responds to a tryCode(int code) message that returns true if that code number opens the vault. The codes are between 0 and 1 million. We want to write a method findCode() that takes a vault, tries all the codes, and returns the first code that opens the vault, or -1 if no code opened the vault. The best way to write this is with the standard for-loop on the outside to iterate the range 0...1 million, and then inside the loop use an if-statement to check for additional exit conditions.

public int findCode(Vault vault) {

for (int i=0; i<=1000000; i++) {

if (vault.tryCode(i)) {

// found the code!

return(i);

}

}

return(-1); // if we get here, we did not find the code

}

So in this case, reaching 1 million is one exit condition, and tryCode() returning true is the other.

**Only Test at the Top**

The while loop checks the test, runs the body, checks the test, runs the body, ... and so on until the test is false. The test is only checked once for each iteration, just before the body runs. The test is not checked while the body runs, although the English phrasing "while (count < 100) { ..." can give the false impression that the test is checked continuously. Suppose we have the loop:

count = 0;

while (count < 100) {

count = count + 10;

count = count - 10;

count = count + 1;

}

This loop changes count up and down inside the loop. For example, at some point we imagine that the line "count = count + 10" will make the test (count < 100) false. However, that will not exit the loop, because the test is not checked at that time. The test is only checked at the very top of the loop, just before each run of the body. So in this strange example, there will be times that count is greater than 100, and the loop will continue to run, exiting only when count is >= 100 at the top of the loop.

[CodingBat.com](http://codingbat.com/) code practice.