**Logic: A Simplified Overview, Part Two**

**Loop Logic**

Computers are great at repeating things over and over again; it's one of the things that they do best. The computer's ability to repeat things precisely, again and again, has been classified in imperative programming languages as LOOP LOGIC.

There are really only three kinds of loops, with many, many variations.

Loop Type 1: Repeat an action WHILE, or for as long as, a condition is true.  
Loop Type 2: Repeat an action FOR a particular length of time, or FOR a given number of repetitions.  
Loop Type 3: Repeat an action until something occurs to stop it.

**Explanations**

Loop Type 1: Repeat an action WHILE, or for as long as, a condition is true.

For example, I could be testing to see whether a bowl of rice has rice in it. If the bowl of rice has rice in it, then I would eat a grain of rice. As long as the bowl still had rice in it, I would continue to eat a grain of rice each time the loop was executed. Once the bowl was found to be empty, the condition would become false and the loop would end.

Example:

while (theGrainsOfRiceInTheBowl > 0) {

eatAGrainOfRice();

}

Loop Type 2: Repeat an action for a particular length of time, or for a given number of repetitions.

For example, I might wish to sort 3 shrimp into a basket. I could create a loop, called a FOR loop, which would allow me to perform my action three times, then stop. Each time through the loop, I would sort one shrimp into the basket. A local counter variable would be in place, which would be INCREMENTED (have 1 added to it) each time through the loop. When the counter reached 3, the loop would end.

Example:

for (myCounter = 0; myCounter < 3; myCounter++) {

sortShrimpIntoBasket();

}

In FOR loops, like the above example, the condition statement has three parts, which can be defined as (initialize; test; increment/decrement). Let's look at this portion of the FOR loop in greater detail.

The first part, initialize, declares and initializes a local counter variable which may possess any name; in this example, the counter has been initialized to 0. Note: the local variable being declared here does NOT require the var keyword.

The second part, test, evaluates whatever condition is placed there. In this example, I am testing the counter and seeing whether it is less than 3. If the counter is less than three, the condition is true and the statements between the curly-braces are executed. If the counter is greater than or equal to 3, than the condition is false and the loop ends without executing any further code.

The third part, increment/decrement, is executed after the lines of code between the curly-braces have been executed. In this example, I am incrementing (or adding 1 to) my counter variable. Each time through the loop, then, the counter increases in number until it reaches 3; at that point, the condition check in the center becomes false and the loop ends, preventing further incrementation.

In my live classes, I would act out the part of the computer running the loop. Obviously, that is not practical here. As an exercise, however, I am going to write out the iterations of the loop.

myCounter is declared and initialized to 0. Is myCounter less than 3? Yes, so the loop executes, and sorts a shrimp into the basket. The third part of the structure is executed, incrementing the myCounter variable by 1. myCounter, then, is now equal to 1.

Is myCounter less than 3? Yes, it is, so the loop executes, and sorts a shrimp into the basket. The myCounter variable is incremented again, and is now equal to 2.

Is myCounter less than 3? Yes, it is, so the loop executes yet again, and sorts a shrimp into the basket. The myCounter variable is incremented again, and is now equal to 3.

Is myCounter less than 3? No, myCounter is equal to 3, so the condition is false, and the loop ends.

You would be amazed at the powerful things you can accomplish with this sort of mind-numbing repetition. The FOR loop is essential to code of any sophistication or ability.

Loop Type 3: Repeat an action until something occurs to stop it.

In my live classes, I act out the part of a computer-generated stopwatch, which keeps ticking away the seconds until someone presses a pretend button on my hand. You get the idea.

This sort of "recursive" loop is usually created in JavaScript using the setTimeout() method of the Window object, which recursively calls its own function over and over again. A demonstration of code for this sort of loop is outside the scope of this class.

One thing you may have noticed about loops is that, regardless of the type of loop under consideration, a loop performs ONE action or one SET of actions at a time; these actions are repeated  
1) WHILE a condition is in place,  
2) FOR a given number of repetitions, or  
3) until something occurs to stop the loop.

We have barely scratched the surface of loop logic; there are a tremendous number of variations on the loop statements I've mentioned here.

Loops are central to performing a wide variety of activities in JavaScript (as well as every other imperative programming language around). You might use loops to generate HTML code (creating repetitive structures like table rows, for instance), or for calculations of one sort or another, or to create game logic, or to access information from complex objects or arrays, or for any one of a vast number of other purposes. As I mentioned earlier, the use of logic, especially loop logic, is an entire world unto itself which is entirely beyond the scope of this introductory lecture.