**Loops**

A *loop* is a programming tool that repeats a set of instructions until a specified condition, called a *stopping condition* is reached. As a programmer, you'll find that you rely on loops all the time! You’ll hear the generic term *iterate* when referring to loops; iterate simply means "to repeat".

When we need to reuse a task in our code, we often bundle that action in a function. Similarly, when we see that a process has to repeat multiple times in a row, we write a loop. Loops allow us to create efficient code that automates processes to make scalable, manageable programs.

As illustrated in the diagram, loops iterate or repeat an action until a specific condition is met. When the condition is met, the loop stops and the computer moves on to the next part of the program.

# Repeating Tasks Manually

Before we write our own loops let's take a moment to develop an appreciation for loops. The best way to do that is by showing you how cumbersome it would be if a repeated task required you to type out the same code every single time.

# The For Loop

Instead of writing out the same code over and over, loops allow us to tell computers to repeat a given block of code on its own. One way to give computers these instructions is with a for loop.

The typical for loop includes an iterator variable that usually appears in all three expressions. The iterator variable is initialized, checked against the stopping condition, and assigned a new value on each loop iteration. Iterator variables can have any name, but it’s best practice to use a descriptive variable name.

A for loop contains three expressions separated by ; inside the parentheses:

1. an initialization starts the loop and can also be used to declare the iterator variable.
2. a stopping condition is the condition that the iterator variable is evaluated against— if the condition evaluates to true the code block will run, and if it evaluates to false the code will stop.
3. an iteration statement is used to update the iterator variable on each loop.

The for loop syntax looks like this:

for (let counter = 0; counter < 4; counter++) { console.log(counter); };

In this example, the output would be the following:

0 1 2 3

Let’s break down the example:

* The initialization is let counter = 0, so the loop will start counting at 0.
* The stopping condition is counter < 4, meaning the loop will run as long as the iterator variable, counter, is less 4.
* The iteration statement is counter++. This means after each loop, the value of counter will increase by 1. For the first iteration counter will equal 0, for the second iteration counter will equal 1, and so on.
* The code block is inside of the curly braces, console.log(counter), will execute until the condition evaluates to false. The condition will be false when counter is greater than or equal to 4 — the point that the condition becomes false is sometimes called the stop condition.

This for loop makes it possible to write 0, 1, 2, and 3 programmatically.

**Looping in Reverse**

What if we want the for loop to log 3, 2, 1, and then 0? With simple modifications to the expressions, we can make our loop run backward!To run a backward for loop, we must:

* The set the iterator variable to the highest desired value in the initialization expression.
* Set the stopping condition for when the iterator variable is less than the desired amount.
* The iterator should decrease in intervals after each iteration.

# Nested Loops

When we have a loop running inside another loop, we call that a nested loop. One use for a nested for loop is to compare the elements in two arrays. For each round of the outer for loop, the inner for loop will run completely.

Let’s look at an example of a nested for loop:

const myArray = [6, 19, 20]; const yourArray = [19, 81, 2]; for (let i = 0; i < myArray.length; i++) { for (let j = 0; j < yourArray.length; j++) { if (myArray[i] === yourArray[j]) { console.log('Both loops have the number: ' + yourArray[j]) } } };

Let's think about what's happening in the nested loop in our example. For each element in the outer loop array, myArray, the inner loop will run in its entirety comparing the current element from the outer array, myArray[i], to each element in the inner array, yourArray[j]. When it finds a match, it prints a string to the console.

Now it's your turn to write a nested loop!

**The While Loop**

You’re doing great! We're going to teach you about a different type of loop: the while loop. To start, let's convert a for loop into a whileloop:

// A for loop that prints 1, 2, and 3 for (let counterOne = 1; counterOne < 4; counterOne++){ console.log(counterOne); } // A while loop that prints 1, 2, and 3 let counterTwo = 1; while (counterTwo < 4) { console.log(counterTwo); counterTwo++; }

Let's break down what's happening with our while loop syntax:

* The counterTwo variable is declared before the loop. We can access it inside our while loop since it's in the global scope.
* We start our loop with the keyword while followed by our stopping condition, or *test condition*. This will be evaluated before each round of the loop. While the condition evaluates to true, the block will continue to run. Once it evaluates to false the loop will stop.
* Next, we have our loop's code block which prints counterTwo to the console and increments counterTwo.

What would happen if we didn't increment counterTwo inside our block? If we didn't include this, counterTwo would always have its initial value, 0. That would mean the testing condition counterTwo < 4 would always evaluate to true and our loop would never stop running! This is called an *infinite loop* and it's something we always want to **avoid**. Infinite loops can take up all of your computer's processing power potentially freezing your computer.

So you may be wondering when to use a while loop! The syntax of a for loop is ideal when we know how many times the loop should run, but we don't always know this in advance. Think of eating like a while loop: when you start taking bites, you don't know the exact number you'll need to become full. Rather you'll eat while you're hungry. In situations when we want a loop to execute an undetermined number of times, while loops are the best choice.

# Do...While Statements

In some cases, you want a piece of code to run at least once and then loop based on a specific condition after its initial run. This is where the do...while statement comes in.

A do...while statement says to do a task once and then keep doing it until a specified condition is no longer met. The syntax for a do...while statement looks like this:

let countString = ''; let i = 0; do { countString = countString + i; i++; } while (i < 5); console.log(countString);

In this example, the code block makes changes to the countString variable by appending the string form of the i variable to it. First, the code block after the dokeyword is executed once. Then the condition is evaluated. If the condition evaluates to true, the block will execute again. The looping stops when the condition evaluates to false.

Note that the while and do...while loop are different! Unlike the while loop, do...whilewill run at least once whether or not the condition evaluates to true.

const firstMessage = 'I will print!'; const secondMessage = 'I will not print!'; // A do while with a stopping condition that evaluates to false do { console.log(firstMessage) } while (true === false); // A while loop with a stopping condition that evaluates to false while (true === false){ console.log(secondMessage) };

# The break Keyword

Imagine we're looking to adopt a dog. We plan to go to the shelter every day for a year and then give up. But what if we meet our dream dog on day 65? We don't want to keep going to the shelter for the next 300 days just because our original plan was to go for a whole year. In our code, when we want to stop a loop from continuing to execute even though the original stopping condition we wrote for our loop hasn't been met, we can use the keyword break.

The break keyword allows programs to “break” out of the loop from within the loop's block.

Let’s check out the syntax of a breakkeyword:

for (let i = 0; i < 99; i++) { if (i > 2 ) { break; } console.log('Banana.'); } console.log('Orange you glad I broke out the loop!');

This is the output for the code:

Banana. Banana. Banana. Orange you glad I broke out the loop!

break statements can be especially helpful when we’re looping through large data structures! With breaks, we can add test conditions besides the stopping condition, and exit the loop when they're met.

**Review**

Great job! In this lesson, we learned how to write cleaner code with loops. You now know:

* Loops perform repetitive actions so we don’t have to code that process manually every time.
* How to write for loops with an iterator variable that increments or decrements
* How to use a for loop to iterate through an array
* A nested for loop is a loop inside another loop
* while loops allow for different types of stopping conditions
* Stopping conditions are crucial for avoiding infinite loops.
* do...while loops run code at least once— only checking the stopping condition after the first execution
* The break keyword allows programs to leave a loop during the execution of its block