**Loops**

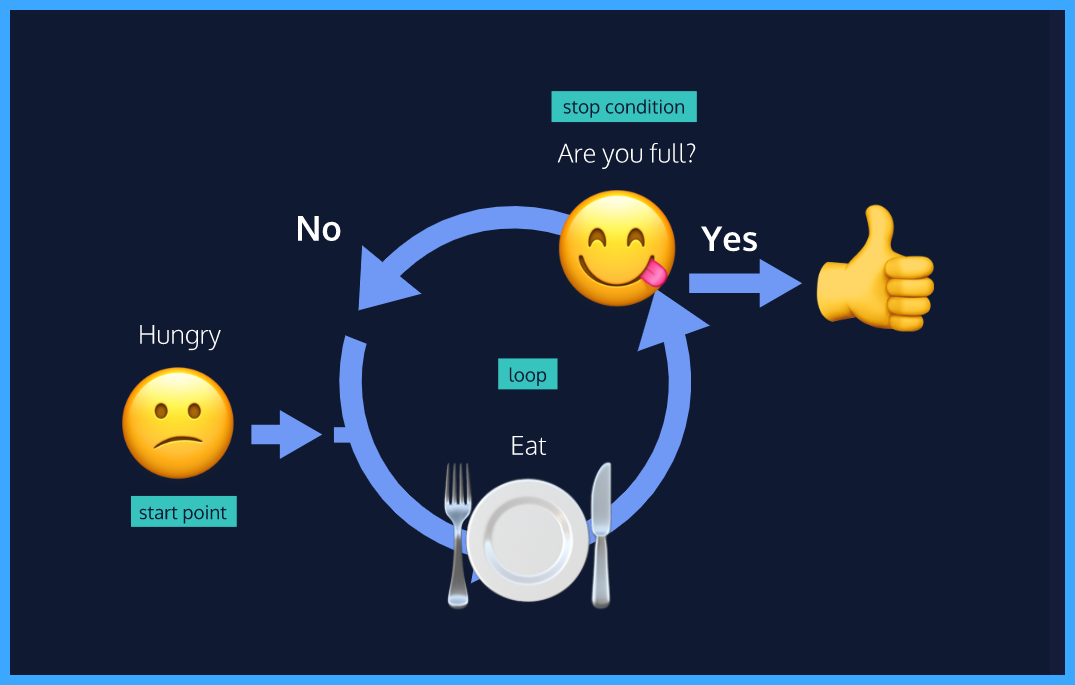
A *loop* is a programming tool that repeats a set of instructions until a specified condition, called a *stopping condition* is reached. As a programmer, you’ll find that you rely on loops all the time! You’ll hear the generic term *iterate* when referring to loops; iterate simply means “to repeat”.

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.

**Instructions**

Check out the diagram to see how loops work in programming.



**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.

**Instructions**

**1.**

Create the variable vacationSpots, and assign its value to an array of three strings naming places you’d like to visit.

Stuck? Get a hint

**2.**

Next, console.log() each item in vacationSpots. Since we don’t know loops yet, we have to console.log() each element in the array separately.

Stuck? Get a hint

**3.**

Nice work! Now imagine that the vacation list had 100 places on it— logging each array element to the console by hand would be a tedious task! In the next exercise, we will learn how to make things more efficient with for loops.

const vacationSpots = ['kolkata', 'mum', 'chennai'];

console.log(vacationSpots[0]);

console.log(vacationSpots[1]);

console.log(vacationSpots[2]);

# 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 than 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.

**Instructions**

**1.**

Now, make your own! Create a program that loops from 5 to 10 and logs each number to the console.

for (let counter = 5; counter < 11; counter++){

  console.log(counter);

}

**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:

* 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.

**Instructions**

**1.**

Make a for loop that loops backwards printing 3 to 0 to the console. Use the >= comparison operator in your stopping condition and the -- operator in your iteration statement.

Hint

Start your initialization variable in your initialization expression to the first number you want to print, 3.

Set the condition to check that the iterator variable remains greater than or equal to 0.

Change the iteration statement so the iterator variable decreases by increments of 1 after each round of the loop. To increase by 1, we use can the operator ++. To decrease by 1, we can use the operator --.

for (let counter = 3; counter >= 0; counter--){

  console.log(counter);

}

# Looping through Arrays

for loops are very handy for iterating over data structures. For example, we can use a for loop to perform the same operation on each element on an array. Arrays hold lists of data, like customer names or product information. Imagine we owned a store and wanted to increase the price of every product in our catalog. That could be a lot of repeating code, but by using a for loop to iterate through the array we could accomplish this task easily.

To loop through each element in an array, a for loop should use the array’s .length property in its condition.

Check out the example below to see how for loops iterate on arrays:

const animals = ['Grizzly Bear', 'Sloth', 'Sea Lion'];

for (let i = 0; i < animals.length; i++){

console.log(animals[i]);

}

This example would give you the following output:

Grizzly Bear

Sloth

Sea Lion

In the loop above, we’ve named our iterator variable i. This is a variable naming convention you’ll see in a lot of loops. When we use i to iterate through arrays we can think of it as being short-hand for the word **i**ndex. Notice how our stopping condition checks that i is less than animals.length. Remember that arrays are zero-indexed, the index of the last element of an array is equivalent to the length of that array minus 1. If we tried to access an element at the index of animals.length we will have gone too far!

With for loops, it’s easier for us to work with elements in arrays.

**Instructions**

**1.**

Write a for loop that iterates through our vacationSpots array using i as the iterator variable.

Inside the block of the for loop, use console.log() to log each element in the vacationSpots array after the string 'I would love to visit '. For example, the first round of the loop should print 'I would love to visit Bali' to the console.

Hint

To access elements in an array we use the syntax arrayName[index]. The index you’ll be accessing will be held in your i iterator variable. You can use string concatenation or string interpolation to create the string you log to the console. Don’t forget about the zero-indexing of arrays! Make sure your stopping condition doesn’t go too far.

for(let i=0;i<vacationSpots.length;i++){

  console.log(`I would love to visit ${vacationSpots[i]}`);

}

# 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!

**Instructions**

**1.**

Imagine you’re a big-wig programmer for a social media platform! You have been tasked with building a prototype for a mutual followers program. You’ll need two arrays of “friends” from two mock users so that you can extract the names of the followers who exist in both lists. Make a variable called bobsFollowers and set it equal to an array with four strings representing the names of Bob’s friends.

**2.**

Make a variable called tinasFollowers and set it equal to an array with three strings representing the names of Tina’s friends. Make exactly two of these the same as two of the friends in the bobsFollowers array.

**3.**

Create a third variable named mutualFollowers and set it to an empty array.

**4.**

Create a nested loop that iterates through bobsFollowers as the array for the outer loop and tinasFollowers as the array for the inner loop. If the current element from the outer loop is the same as the current element from the inner loop, push that element into the mutualFollowers array.

Hint

It might be helpful to look at the example from the narrative again:

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])

}

}

};

In your nested loop, you’ll want to push the matching element into your mutualFollowers array instead of printing a string.

// Write your code below

let bobsFollowers = ['a','b','c','d'];

let tinasFollowers = ['a','b','e' ];

let mutualFollowers = [];

const myArray = [6, 19, 20];

const yourArray = [19, 81, 2];

for (let i = 0; i < bobsFollowers.length; i++) {

  for (let j = 0; j < tinasFollowers.length; j++) {

    if (bobsFollowers[i] === tinasFollowers[j]) {

      mutualFollowers.push(bobsFollowers[i]);

    }

  }

};

console.log(mutualFollowers);

**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 while loop:

// 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, 1. 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.

**Instructions**

**1.**

Below the cards array, declare a variable, currentCard, with the let keyword but don’t assign it a value.

Stuck? Get a hint

**2.**

Create a while loop with a condition that checks if the currentCard does not have that value 'spade'.

Inside the block of your while loop, add the following line of code:

currentCard = cards[Math.floor(Math.random() \* 4)];

Math.floor(Math.random() \* 4) will give us a random number from 0 to 3. We’ll use this number to index the cards array, and assign the value of currentCard to a random element from that array.

Stuck? Get a hint

**3.**

Awesome! Your loop is running, but you can’t tell because it doesn’t output anything. Let’s add a console.log() statement to our while block. Inside the block, after you assign currentCard a new value, log currentCard to the console.

For fun you can run your code a few times and see how the output changes!

Hint

Add console.log(currentCard) to the correct place in your while loop block.

while (currentCard !== 'spade') {

currentCard = cards[Math.floor(Math.random() \* 4)];

// On this line, log current card to the console

}

const cards = ['diamond', 'spade', 'heart', 'club'];

// Write your code below

let currentCard;

while(currentCard != 'spade'){

  currentCard = cards[Math.floor(Math.random() \* 4)];

  console.log(currentCard)

}

# 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 do keyword 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...while will 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)

};

**Instructions**

**1.**

We’d like a program to simulate part of the cake-baking process. Depending on the recipe, a different number of cups of sugar is required. Create the variable cupsOfSugarNeeded, and assign it a number value of your choosing. The cups of sugar must be added to the batter one at a time. Declare the variable cupsAdded and assign it the value 0.

**2.**

We have a sweet tooth, so we want to add at least one cup of sugar to the batter even if the value of cupsOfSugarNeeded is 0. Create a do...while loop which increments cupsAdded by one while cupsAdded is less than cupsOfSugarNeeded.

Hint

Within the block of our do...while loop, we’ll increment the value of cupsAdded. Our stopping condition will compare cupsAdded to cupsOfSugarNeeded:

do {

cupsAdded++

} while (cupsAdded < cupsOfSugarNeeded);

// Write your code below

let cupsOfSugarNeeded = 5;

let cupsAdded = 0;

do{

  cupsAdded++;

}while(cupsOfSugarNeeded>cupsAdded)

# 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 break keyword:

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 above 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.

**Instructions**

**1.**

Log each element from rapperArray in a for loop with the iterator variable i.

Stuck? Get a hint

**2.**

After the for loop, log the string "And if you don't know, now you know." to the console. Note: since there’s a single quote character, ', in our string, we can use double quotes around the string to make sure character prints.

**3.**

Add a break inside your loop’s block that breaks out of the loop if the element at the current index in the rapperArray is 'Notorious B.I.G.'.

Hint

Add a conditional to your loop’s block that compares 'Notorious B.I.G.' and the element at the current index of the rapperArray.

myArray[myArrayIndex] === 'element value';

const rapperArray = ["Lil' Kim", "Jay-Z", "Notorious B.I.G.", "Tupac"];

// Write you code below

for (let i = 0; i < rapperArray.length; i++){

  console.log(rapperArray[i]);

  if (rapperArray[i] === 'Notorious B.I.G.'){

    break;

  }

}

console.log("And if you don't know, now you know.");

# 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