Looping Lab

- Due No Due Date
- Points 1
- Submitting a website url

(https://github.com/learn-co-curriculum/phase-0-intro-to-js-2-looping-code-along) (https://github.com/learn-co-curriculum/phase-0-intro-to-js-2-looping-code-along/issues/new)

Learning Goals

- Build a `for` loop.
- Build a `while` loop.
- Explain the purpose of a loop.
- Understand when to use each type of loop.

Introduction

In an earlier lesson, we learned about *repetition statements* — statements that allow us to break out of the default sequence and repeat a section of code multiple times. We also learned about two of JavaScript's repetition constructs: the *while* loop and the *do...while* loop. In this lesson we will learn about a third: the *for* loop.

Let's say we have a bunch of gifts to wrap. They all happen to be the same size and shape, so for every gift, we need to cut a similarly sized piece of wrapping paper, fold it up over the edges of the gift, tape it together, and add a nice little card. Then we set the wrapped gift aside and move on to the next gift.

In programming terms, we can think of our **collection** of gifts as an `Array` and the act of wrapping them as a function. For example:

```
const gifts = ["teddy bear", "drone", "doll"];

function wrapGift(gift) {
```

```
  console.log(`Wrapped ${gift} and added a bow!`);
}
```

We could then call `wrapGift()` on each gift individually:

```
wrapGift(gifts[0]);
wrapGift(gifts[1]);
wrapGift(gifts[2]);
```

However, this isn't very efficient or extensible. It's a lot of repetitive code to write out, and if we had more gifts we'd have to write a whole new line for each.

This is where loops come in handy! With a loop, we can just write the repeated action **once** and perform the action on **every item in the collection**.

This is a code-along, so follow along with the instructions in each section. There are tests to make sure you're coding your solutions correctly. If you haven't already, **fork and clone** this lab into your local environment. Navigate into its directory in the terminal, then run `code .` to open the files in Visual Studio Code.

The `for` loop

Of the loops in JavaScript, the `for` loop is the most common. The `for` loop is made up of four statements in the following structure:

```
for ([initialization]; [condition]; [iteration]) {
  [loop body]
}
```

- Initialization
  - Typically used to initialize a **counter** variable.
- Condition
  - An expression evaluated before each pass through the loop. If this expression evaluates to `true`, the statements in the loop body are executed. If the expression evaluates to `false`, the loop exits.
- Iteration

    - An expression executed at the end of each iteration. Typically, this will involve incrementing or decrementing a counter, bringing the loop ever closer to completion.
- Loop body
    - Code that runs on each pass through the loop.

Usage: Use a `for` loop when you know how many times you want the loop to run (for example, when you're looping through elements in an array).

Examples

Let's take a look at an example and get some practice using the Node debugger. Enter the code below into the `index.js` file.

> **Note:** You can, of course, copy/paste the code rather than typing it in yourself, but we recommend typing it in for now. The act of typing code develops muscle memory and helps your brain understand and internalize the syntax.

```
for (let age = 30; age < 40; age++) {
  console.log(`I'm ${age} years old. Happy birthday to me!`);
  debugger;
}
```

In the above code, `let age = 30` is the **initialization**: we're creating a variable, `age`, that we'll use in the next three phases of the loop. Notice that we use `let` instead of `const` because we need to increment the value of `age`.

The **condition** for the above loop is `age < 40`, or, in other words, "Run the code in the loop body until `age` is NOT less than `40`." As long as the condition evaluates to `true`, the code in the loop body is executed, the value of `age` is incremented, and the condition is reevaluated. As soon as the condition evaluates to `false`, the loop ends.

The **iteration** is `age++`, which increments the value of `age` by `1` after every pass through the loop. We initialized `age` as `30`, and it retains that value during the first pass through the loop. At the end of the first pass, we increment `age` to `31`, check whether the condition still holds `true`, and, since it does, run the loop body again with `age` as `31`. After that second loop, we increment `age` to `32`, and so on.

The **loop body** is the set of statements that we want to run when the condition evaluates to `true`.

Let's take a look at what's happening in our loop using debugger. Run `node inspect index.js` in the terminal to start the debugger. You should see the following:

```
[~/.../intro_to_js_2/phase-0-intro-to-js-2-looping-code-along // ♥ > node inspect index.js
 < Debugger listening on ws://127.0.0.1:9229/06e35971-bcb0-4d98-96bf-87f0b9dc0951
 < For help, see: https://nodejs.org/en/docs/inspector
 < Debugger attached.
 Break on start in index.js:2
   1 // Code your solutions in this file
 > 2 for (let age = 30; age < 40; age++) {
   3     console.log(`I'm ${age} years old. Happy birthday to me!`);
   4     debugger;
 debug>
```

Now run `cont` at the `debug` prompt to start executing the loop and advance to the breakpoint. It should look like this:

```
[debug> cont
 < I'm 30 years old. Happy birthday to me!
 break in index.js:4
   2 for (let age = 30; age < 40; age++) {
   3     console.log(`I'm ${age} years old. Happy birthday to me!`);
 > 4     debugger;
   5   }
 debug>
```

Note, in the first line, that the message has been logged and that `age` is equal to 30. To verify that, enter `repl` at the `debug` prompt to open the REPL, then enter `age`. You should see `30` output. Now type `Ctrl-C` to exit the repl, and enter `cont` at the `debug` prompt to advance to the next iteration of the loop. You should see the following:

```
[debug> cont
 < I'm 31 years old. Happy birthday to me!
 break in index.js:4
   2 for (let age = 30; age < 40; age++) {
   3     console.log(`I'm ${age} years old. Happy birthday to me!`);
 > 4     debugger;
   5   }
 debug>
```

Note that the logged output now shows that age is 31. You can continue to step through the iterations by running `cont` at the `debug` prompt. At any point you can verify the value of `age` by entering the REPL and typing `age` at the prompt, then exit the REPL to continue stepping through. When you're done, enter `.exit` or `Ctrl-C` twice to exit the debugger.

Now let's remove `debugger;` from our code and execute it by running `node index.js`. You should see the following:

```
~/.../intro_to_js_2/phase-0-intro-to-js-2-looping-code-along // ♥ > node index.js
I'm 30 years old. Happy birthday to me!
I'm 31 years old. Happy birthday to me!
I'm 32 years old. Happy birthday to me!
I'm 33 years old. Happy birthday to me!
I'm 34 years old. Happy birthday to me!
I'm 35 years old. Happy birthday to me!
I'm 36 years old. Happy birthday to me!
I'm 37 years old. Happy birthday to me!
I'm 38 years old. Happy birthday to me!
I'm 39 years old. Happy birthday to me!
~/.../intro_to_js_2/phase-0-intro-to-js-2-looping-code-along // ♥ >
```

Using `for` with Arrays

The `for` loop is often used to iterate over every element in an array. Let's rewrite our gift-wrapping action above as a `for` loop. Enter the following code into `index.js`:

```
const gifts = ["teddy bear", "drone", "doll"];

function wrapGifts(gifts) {
  for (let i = 0; i < gifts.length; i++) {
    console.log(`Wrapped ${gifts[i]} and added a bow!`);
    debugger;
  }

  return gifts;
}

wrapGifts(gifts);
```

We started our counter, `i`, at `0` because arrays have zero-based indexes. Our condition states that we should run the code in the loop body while `i` is less than `gifts.length` (`3` in the above example). Our iteration, `i++`, increments our counter by `1` at the end of each pass through the loop.

Run `node inspect index.js` in the terminal to enter the debugger, and `cont` at the `debug` prompt to advance to the breakpoint. You should see `Wrapped teddy bear and added a bow!` logged. In our loop body, we reference `gifts[i]`. Since `i` starts out as `0`, during the first pass through

the loop `gifts[i]` is `gifts[0]`, which is `'teddy bear'`. Continue stepping through the loop by entering `cont` at the `debug` prompt and remember you can enter the REPL at any point to check the values of our variables, `i` and `gifts[i]`.

When you're done, remove the `debugger` and execute the code by running `node index.js`. You should see the following logged to the terminal:

```
Wrapped teddy bear and added a bow!
Wrapped drone and added a bow!
Wrapped doll and added a bow!
```

Assignment

In the previous section, the `wrapGifts()` function allowed us to take any array of gifts and loop over them, logging our own message. Let's practice that with a slightly different idea. To complement our gift wrapping function, your task is to create a thank you card creator.

> **Note**: Recall the difference between logging and returning values from a function. When we log information we are simply outputting text to a terminal or console. When we return data from a function we will be able to reference and use that information elsewhere because the data is being passed out of the function.

In `index.js`, build a function named `writeCards()` that accepts two arguments: an array of string names, and an event name. Create a `for` loop with a counter that starts at `0` and increments at the end of each loop. The `for` loop should stop once it has iterated over the length of the array.

As with our previous `wrapGifts()` function, you will create a custom message for each name inside the loop. Unlike that example, however, instead of simply logging the messages to the console, you will collect them in a *new* array and **return** this array at the end of the function. (Refer back to the Array Methods lesson if you need a refresher on how we can add an element to an array.) The overall process should be:

1. create a new, empty array to hold the messages;
2. iterate through the input array and, inside the loop, build out the 'thank you' message for each name using string interpolation, then add that message to the new array you created;
3. after the loop finishes and all of the messages have been added to the new array, return the new array.

Here is an example of what a call to the `writeCards()` function might look like:

```
writeCards(["Charlie", "Samip", "Ali"], "birthday");
```

If we were to call the function using this function call, it should produce the following array as the return value:

```
[
  "Thank you, Charlie, for the wonderful birthday gift!",
  "Thank you, Samip, for the wonderful birthday gift!",
  "Thank you, Ali, for the wonderful birthday gift!",
];
```

> **Top Tip**: The debugger isn't just for debugging code — you can also use it to help you write your function! Try building the structure of the loop, putting the `debugger` inside the loop body. Even before you start writing the code, you can enter the debugger's REPL and try out code until you figure out how to create the message and add it to an array. Once it's working in the REPL, transfer the code to `index.js`, exit the debugger, and run the tests.

The `while` loop

Recall from the earlier lesson that the `while` loop is similar to a `for` loop, repeating an action in a loop based on a condition. Both will continue to loop until that condition evaluates to `false`. Unlike `for`, `while` only requires condition and loop statements:

```
while ([condition]) {
  [loop body]
}
```

The initialization and iteration statements of the `for` loop have not disappeared, though. In fact, we could rewrite our original `for` loop gift wrapping example using a `while` loop and achieve the exact same result:

```
const gifts = ["teddy bear", "drone", "doll"];

function wrapGifts(gifts) {
  let i = 0; // the initialization moves OUTSIDE the body of the loop!
```

```
  while (i < gifts.length) {
    console.log(`Wrapped ${gifts[i]} and added a bow!`);
    i++; // the iteration moves INSIDE the body of the loop!
  }

  return gifts;
}

wrapGifts(gifts);
// LOG: Wrapped teddy bear and added a bow!
// LOG: Wrapped drone and added a bow!
// LOG: Wrapped doll and added a bow!
// => ["teddy bear", "drone", "doll"]
```

Notice that we've just moved the initialization and iteration statements — declaring the `i` variable *outside* the loop, and incrementing it *inside* the loop.

> **CAUTION**: When using `while` loops, it is easy to forget to involve iteration. Leaving iteration out can result in a condition that *always* evaluates to `true`, causing an infinite loop!

Using a `while` loop enables us to check conditions that aren't based on a counter. Take a look at the following ***pseudocode*** ***(https://en.wikipedia.org/wiki/Pseudocode)*** for an example of how we could use `while` in a program for planting a garden:

```
function plantGarden() {
  let keepWorking = true;
  while (keepWorking) {
    chooseSeedLocation();
    plantSeed();
    waterSeed();
    keepWorking = checkForMoreSeeds();
  }
}
```

We can imagine that *while* we have seeds, we take the same steps over and over: choose a location for a seed; plant it; water it. Then, check if there are more seeds. If *not*, do not keep working.

When to Use `for` and `while`

JavaScript, like many programming languages, provides a variety of looping options. Loops like `for` and `while` are actually just slight variations of the same process. By providing a variety, we as programmers have a larger vocabulary to work with.

Often, you will see `while` loops simply being used as an alternative to `for` loops:

```
let countup = 0;
while (countup < 10) {
  console.log(countup++);
}
```

This is perfectly fine as an alternative way to describe:

```
for (let countup = 0; countup < 10; countup++) {
  console.log(countup);
}
```

If you're feeling a bit lost about when to use a `for` vs. a `while` loop, take a deep breath. Most of the time, a regular `for` loop will suffice. It's by far the most common looping construct in JavaScript. A general heuristic for choosing which loop to use is to first try a `for` loop. If that doesn't serve your purposes, then go ahead and try a `while` loop. Also, remember that you can always refer to the **documentation on these loops (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration)** at any time.

Just don't forget: with `while`, make sure you are updating the condition on each loop so that the loop eventually terminates!

Assignment

To get more acquainted with `while`, your task is to write a function, `countDown`, that takes in any positive integer and, starting from that number, counts down to zero using `console.log()`. Note that this means that running `countDown(10);` would actually log *11* times:

```
10
9
8
7
6
5
4
3
2
1
0
```

Remember the workflow:

1. Install the dependencies using `npm install`.
2. Run the tests using `npm test`.
3. Read the errors; vocalize what they're asking you to do.
4. Write code; repeat steps 2 and 3 often until a test passes.
5. Repeat as needed for the remaining tests.

Conclusion

After some time programming in JavaScript, writing a `for` loop will come as naturally to you as wrapping one gift after another. Just as you slowly become comfortable using different words and vocabulary to better express yourself, you will become more acquainted with concepts like `for` and `while` until you are able to discern the nuanced differences in usage between them.

Resources

- Codecademy
  - `for` loop (http://www.codecademy.com/glossary/javascript/loops#for-loops)
  - `while` loop (http://www.codecademy.com/glossary/javascript/loops#while-loops)
- MDN

- `for` loop (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for)
- `while` loop (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/while)
- Loops and iteration (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration)