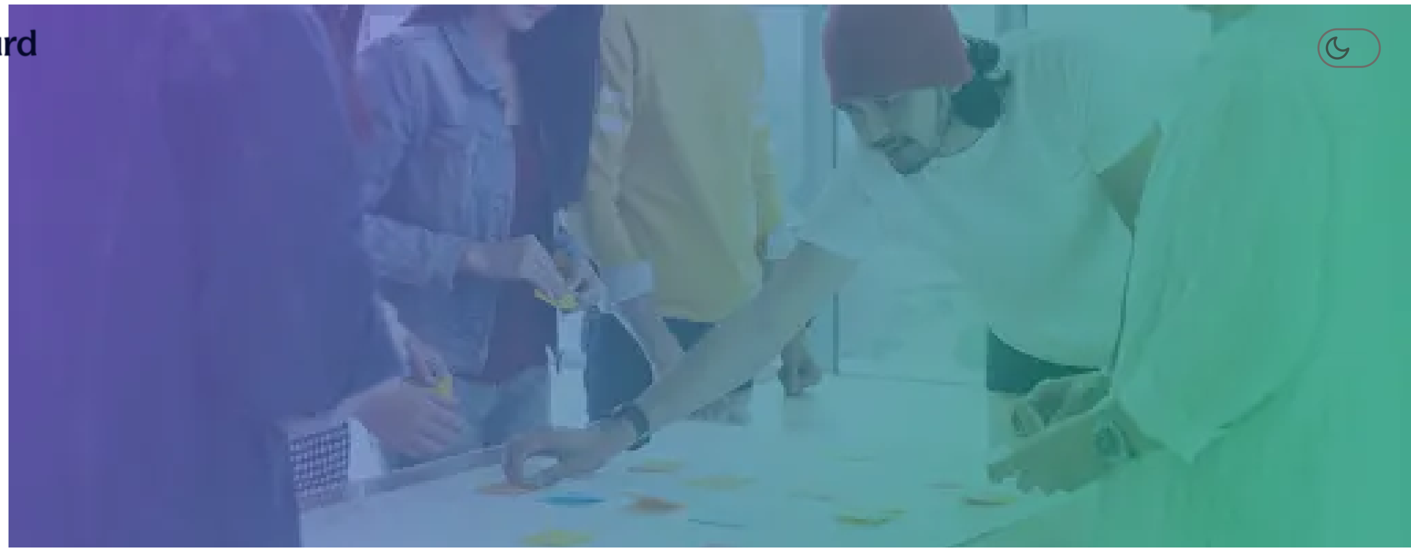
Springboard



Loops

Goals

- Learn why loops are used and how to control them
- Explain how to implement for and while loops for repetitive operations
- Understand the terms iterator and iterable
- Learn how to loop through strings in both directions
- Learn special tricks and know what things to avoid with loops

Loops

The Purpose Of Loops

What if you wanted to print odd numbers up to 10 to the console?

```
console.log(1); console.log(3); console.log(5); console.log(7);
console.log(9);
```

- There is too much repetition here!
- We want our code to be DRY (Don't Repeat Yourself!).

- Those numbers are hard-coded!
- We want to take a more programmatic approach by abstracting the problem.

Let's Fix Our Example

Let's solve the previous slide with a for loop!

```
for (let i = 1; i <= 10; i += 2) { console.log(i); }
```

What's happening here?

- First a variable i is initialised to 1.
- We tell i to keep going until it equals 10
- After each iteration (loop cycle), we tell i to go up by 2
- Inside the loop block, we print the value of i for that iteration
- Our code prints "1, 3, 5, 7, 9" then i gets incremented to 11
- 11 is not <= 10, so the loop stops.

For Loops

Most for loops follow a general structure!

for (let i = 0; i < 3; i++)

1. Initialisation. Declare or assign a counter variable, also known as an iterator (often named i).
2. Condition. This expression is evaluated before each iteration (loop cycle), if it resolves to true, the code inside the loop runs, otherwise the loop exits.
3. Final-Expression. This is where we increment the iterator. This happens after the loop code runs, and before the next condition gets evaluated.

Looping "Through" Things

Obviously, the primary use of loops is not to count numbers.

The main functionality is the ability to iterate through special sequences called iterable objects.

The two main iterable objects in JavaScript are strings and arrays.

Looping over arrays

A World Without Loops

```
const cities = [ "San Francisco", "Berlin", "Tokyo", "Moscow",
"Buenos Aires" ]; console.log(cities[0]); // "San Francisco"
console.log(cities[1]); // "Berlin" console.log(cities[2]); //
"Tokyo" console.log(cities[3]); // "Moscow"
console.log(cities[4]); // "Buenos Aires"
```

Iterating Through An Array

```
const cities = [ "San Francisco", "Berlin", "Tokyo", "Moscow",
"Buenos Aires" ]; for (let i = 0; i < cities.length; i++) {
console.log(cities[i]); }
```

```
const cities = [ "San Francisco", "Berlin", "Tokyo", "Moscow",
"Buenos Aires" ]; let i = 0; while (i < cities.length) {
console.log(cities[i]); i++ }
```

For ... Of Loops

By default, for loops in JavaScript iterate through indices

With a for...of loop, we can iterate through the elements directly!

```
const cities = [ "San Francisco", "Berlin", "Tokyo", "Moscow",
"Buenos Aires" ]; for (let city of cities) { console.log(city);
}
```

For Loops

```
const cities = [ "San Francisco", "Berlin", "Tokyo", "Moscow",
"Buenos Aires" ]; for (let i = 0; i < cities.length; i++) {
console.log(cities[i]); }
```

```
for (let city of cities) { console.log(city); }
```

Iterating Through An Object

```
const city = { name: "San Francisco", state: "California",
population: 871000, bridges: 2 } for (let key in city) {
console.log(key, city[key]); } // For...of loops don't work on
objects!
```

Looping Through Strings

Let’s look at some real code to loop through a string

```
let str = 'pancakes'; // does the string include a "k"? let
includesK = false; for (let i = 0; i < str.length; i++) { if
(str[i] === 'k') { includesK = true; } }
```

- This loop is initialized to zero, because string indices start at zero.
- This loop condition goes up to the last character, str.length - 1

- i gets incremented by 1 every time until it reaches str.length and exits
- Characters can be accessed with bracket notation str[i] becomes “p” then “a” then “n” and so forth.

Note: This is basically the .includes() method!

Looping Backwards

Let’s use the flexibility of loop conditions to go backwards!

```
let myStr = 'live'; let reversed = ''; // reverse it! for (let
i = myStr.length - 1; i >= 0; i--) { reversed += myStr[i]; }
console.log(reversed); // 'evil'
```

- The loop is initialized to the last character (myStr.length - 1)
- The loop condition goes down to the first character at index 0
- i gets decremented by 1 every time until it reaches -1 and exits
- reversed gets re-created each time with a new char from myStr

Let Revisited

Why do we keep using the let keyword for all of our iterators?

Answer: The let keyword is scoped inside the loop block only.

```
for (let i = 0; i < 2; i++) { console.log(i); } // 0 // 1 for
(var i = 0; i < 2; i++) { console.log(i); } // 0 // 1
console.log(i); // 2 <-- why would we want this?!
```

While Loops

The main alternative to for loops are while loops. They are more open-ended than for loops, because of their simpler structure:

```
while (condition) { // do stuff }
```

There is only a single condition now, which is an expression that gets evaluated before each iteration. If it is true, the loop runs.

```
let i = 0; while (i < 3) { console.log(i); i++; // we must
manually increment in the loop body } // 0 // 1 // 2
```

Beware Infinite Loops

One downside of while loops is that it is much easier to get stuck in infinite loops, which can be devastating bugs as they continue to consume memory and can lock up your system if you aren't careful.

Here is an example infinite loop:

```
let i = 0; while (i < 5) { console.log(i * i); // FORGOT TO
INCREMENT i ! }
```

If you get stuck in an infinite loop in the browser, open up a new tab and close that window

Special Keywords

A very useful keyword is ***break***, which immediately exits a loop.

This was introduced when we talked about switch statements!

```
// Find the index of a character, like String.indexOf()! let
idx; for (let i = 0; i < str.length; i++) { if (str[i] === 'q')
{ idx = i; break; // since we no longer need to search the
string } }
```

continue

The continue keyword, used less often, skips to the next iteration.

```
for (let i = 1; i < 5; i++) { if (i > 2 && i < 4) { continue;
// skip these numbers! } console.log(i); } // 1 // 2 // 4
```

Object.keys

- Accepts an object
- Returns an array of the object’s keys
- Keys are in the same order as in a for...in loop

```
let weirdObj = { hereIsAKey: "here is a value!", "10": "number
key!", thisIsABoolean: true, "5": "another number key!",
nullValue: null } Object.keys(weirdObj); // ["5", "10",
"hereIsAKey", "thisIsABoolean", "nullValue"]
```

Object.values

- Accepts an object
- Returns an array of the object’s values
- Values are in the same order as in a for...in loop

```
let weirdObj = { hereIsAKey: "here is a value!", "10": "number
key!", thisIsABoolean: true, "5": "another number key!",
nullValue: null }
```

```
Object.values(weirdObj); // ["another number key!", "number
key!", "here is a value!", true, null]
```

Nested Arrays

```
const ticTacToe = [ ["X", "O", "X"], ["X", "O", "O"], ["X",
"O", "X"] ]; const maze = [ [1, 0, 1, 0, 1, "END"], [0, 1, 0,
0, 1, 0], [0, 0, 1, 0, 0, 0], [1, 0, 1, 0, 0, 0], [0, 1, 0, 0,
1, 0], [1, 1, 1, 0, 0, 1], [0, 1, 0, 0, 1, 0], ["START", 0, 0,
0, 0, 0] ];
```

Iterating Over Nested Arrays

```
const ticTacToe = [ ["X", "O", "X"], ["X", "O", "O"], ["X",
"O", "X"] ]; let xCount = 0; for(let i = 0; i <
ticTacToe.length; i++){ for(let j = 0; j < ticTacToe[i].length;
j++){ if(ticTacToe[i][j] === "X"){ xCount++; } } }
```

Nested Objects

```
let data = { artist: "The Beatles", albums: [ { title: "Abbey
Road", producer: "George Martin", releaseYear: 1969, numTracks:
17, length: "47:23" }, { title: "Sgt. Pepper's Lonely Hearts
Club Band", producer: "George Martin", releaseYear: 1967,
numTracks: 13 } ] };
```

Iterating Over An Array in an Object

```
for (let album of data.albums) { console.log("The title is",
album.title) }
```