A mini-session on loops in JavaScript

15th October 2025

In the video for this mini-session which can be found on YouTube at:
https://youtu.be/28ieBlCD4bE?si=aUd-xJDLxBnZLnL2

I am using two resources:

1. https://www.wscubetech.com/resources/javascript/compiler to run all of the examples of code in and,
2. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide which is Mozilla's primer of JavaScript so when I encounter an error I can check the syntax.

Each time I run a snippet of code. I use the following checklist:

1. Do brackets match at the beginning and the end of an expression?
2. Is there a semi-colon at the end of each expression (including in a **for** loop) and commas between elements in an array?
3. Are there quotation marks around strings, both when declaring and using them?
4. Are my logical operators (+=, ==, ===, etc.,) the ones I intended to use?
5. Spelling: Are variables, constants, commands spelt consistently and correctly throughout the whole code?
6. Are all variables declared before use? (Including inside the **for** loop).

Why loops?

When programming, we often face repetitive tasks, for example, processing 1000 orders in our online shop or drawing 100 lines on a screen. Writing code to do these things is inefficient and introduce errors, which is where loops come in.

A loop is a control structure that allows us to execute a block of code repeatedly based on a condition.[1]

Two types of loops: For and while

[1] Ada Lovelace, poet Bryon's daughter, first thought of loops after watching how a loom worked saying: *The Analytical Engine weaves algebraic patterns, just as the Jacquard-loom weaves flowers and leaves,* The Sketch of the Analytical Machine*, Taylors Scientific Memoirs, 1843*.

Broadly speaking we have two types, the **for** loop and the while **loop**.

Consider this snippet of code:

```
const rhyme = ['the', 'cat', 'sat', 'on', 'the', 'mat'];

console.log (rhyme);
```

This gives us:

```
['the', 'cat', 'sat', 'on', 'the', 'mat']
```

If we access the element in the array by number:

```
const rhyme = ['the', 'cat', 'sat', 'on', 'the', 'mat'];

console.log (rhyme[0]);
```

This gives us:

```
the
```

However, the rhyme array is six elements, to get the rest of the array we would have to repeat the line like so:

```
const rhyme = ['the', 'cat', 'sat', 'on', 'the', 'mat'];

console.log (rhyme[0]);
console.log (rhyme[1]);
console.log (rhyme[2]);
console.log (rhyme[3]);
console.log (rhyme[4]);
```

```
console.log (rhyme[5]);
```

This gives us:

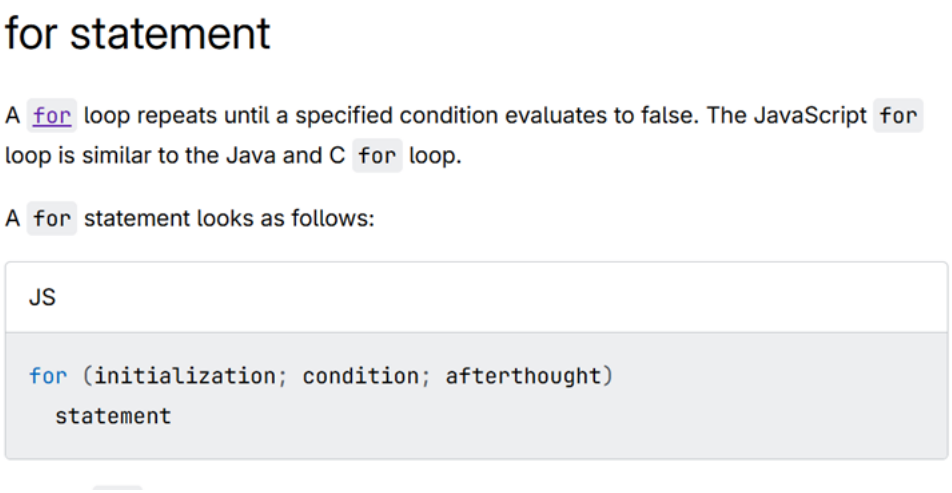
```
the
cat
sat
on
the
mat
```

This cutting and pasting of the line of code is awkward and can lead to errors, but is not terrible given we only have six entries. However, what if we have a whole document of words that we have read in? We would need to repeat the same code over and over.

This is why we need loops of which we have two types.

Two types of loops

1. The **for** loop – when we know how many times we wish to repeat the code.



for statement

A `for` loop repeats until a specified condition evaluates to false. The JavaScript `for` loop is similar to the Java and C `for` loop.

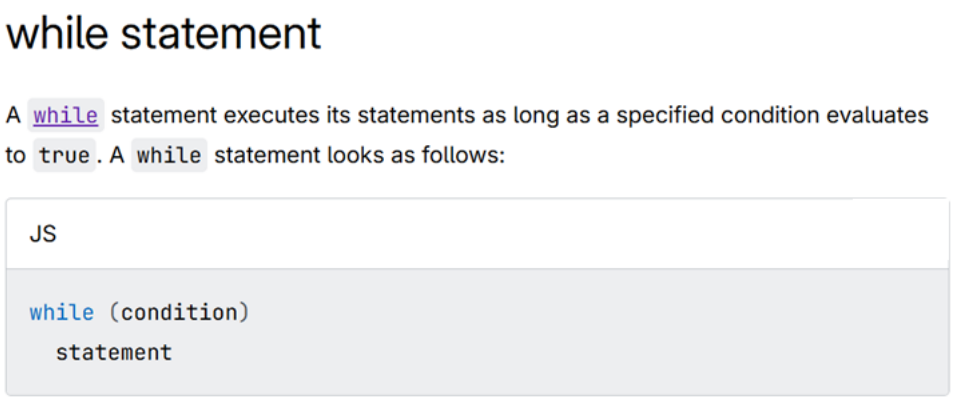
A `for` statement looks as follows:

JS

```
for (initialization; condition; afterthought)
  statement
```

2. The **while** loop – when sometimes we don't know how many times we wish to repeat the code so we set up a (Boolean) condition so that the loop will only

execute when the condition is true/false, once the state changes, so that the loop exits.



while statement

A `while` statement executes its statements as long as a specified condition evaluates to `true`. A `while` statement looks as follows:

JS

```
while (condition)
  statement
```

The **for** loop

We use the for loop when we know how many times we wish to repeat some code. If we look at our code again and write it with a for loop this is what it will look like:

```
const rhyme = ['the', 'cat', 'sat', 'on', 'the', 'mat'];


for (i=0; i<=5; i++)
{
      console.log (rhyme[i]);
}
```

This gives us this:

```
the
cat
sat
on
the
mat
```

Q: Play with making [i] for index higher or lower. What do you get?

A: We miss words off the sentence or we have an undefined element, which means, that the element didn't exist.

To make our output easier to read we can format it, by putting each word (or array element) in a sentence and constructing it space by space like so:

```
const rhyme = ['the', 'cat', 'sat', 'on', 'the', 'mat'];

let sentence ="";
for (i=0; i<=5; i++)
{
    sentence += rhyme[i] + " ";
}
console.log (sentence);
```

This gives us:

```
the cat sat on the mat
```

Q: if we move the console.log (sentence) output command to inside the curly brackets, what happens?

```
const rhyme = ['the', 'cat', 'sat', 'on', 'the', 'mat'];

let sentence ="";
for (i=0; i<=5; i++)
{
    sentence += rhyme[i] + " ";
    console.log (sentence);
}
```

A: It would repeat each iteration of the loop which looks like this:

```
the
the cat
the cat sat
the cat sat on
the cat sat on the
the cat sat on the mat
```

This can be a useful technique to see exactly what is happening when we have a logic problem and we are not sure what is going on.

With our **for** loop set up, we can add an if statement to begin analysing our text sentence in more detail:

```
const rhyme = ['the', 'cat', 'sat', 'on', 'the', 'mat'];

let sentence ="";
let thecount =0;

for (i=0; i<=5; i++)
{
    sentence += rhyme[i] + " ";
    if (rhyme[i] == 'the')
    thecount++;
    }
console.log (sentence);
console.log ('The number of thes is: ' + thecount);
```

This gives us:

```
the cat sat on the mat
The number of thes is: 2
```

And we start to see how useful a for loop can be.

The **while** loops

The **while** look is very similar and indeed, we could use the same code but just put it inside a while loop.

```
const rhyme = ['the', 'cat', 'sat', 'on', 'the', 'mat'];

let sentence ="";
let thecount =0;
let i=0;

while (i<=5)
{
    sentence += rhyme[i] + " ";
    if (rhyme[i] == 'the')
    thecount++;
    i++
    }
console.log (sentence);
console.log ('The number of thes is: ' + thecount);
```

Again, this gives us:

```
the cat sat on the mat
The number of thes is: 2
```

Alternatively, if we didn't know how big our text was, and we couldn't figure out the specified number of times we need to loop over it without causing a problem, we do so by using a condition.

The condition must be true in order to begin the loop. In this case we initialise the counter or i (for index) outside the loop, and we also initialise our sentence outside the loop too.

Q: Why is this?

A: We do not want to reset out sentence with each iteration. And then we put our Boolean statement inside the loop like this:

```
const rhyme = ['the', 'cat', 'sat', 'on', 'the', 'mat', 'fullstop'];


let sentence ="";
let thecount =0;
let i=0;


while (rhyme[i] != 'fullstop')
{
    sentence += rhyme[i] + " ";
    if (rhyme[i] == 'the')
    thecount++;
    i++
    }
console.log (sentence);
console.log ('The number of thes is: ' + thecount);
```

Q: What have I done to create the condition?

A: I have added another element to my array ‘fullstop’ and I use that to search for the end of the array.

If I didn’t know how long the array (or say, if this was a large file of text that I read in) I ask my loop to search for the unique identifier ‘fullstop’ signalling the end of the array (or file).

Q: What could go wrong with this?

A: I might have chosen a word that is in the text file earlier on and I might miss some text. So, choose something completely unique that you know won’t show up in the text file.

Q: What other ways could we make sure we do not get stuck in an endless loop?

Break or Return

Some programmers recommend using a break or return to exit out of the loop:

```
const rhyme = ['the', 'cat', 'sat', 'on', 'the', 'mat', 'fullstop'];

let sentence ="";
let thecount =0;
let i=0;

while (i<=5)
{
    sentence += rhyme[i] + " ";
    if (rhyme[i] == 'the')
    thecount++;
    if (rhyme[i] == 'on')
    break;
    i++
    }
console.log (sentence);
console.log ('The number of thes is: ' + thecount);
```

Q: Swap out the **break** for **return**. What happens? What is the difference between break and return? What problems could this cause?

While technically not wrong (and this example is a tiny bit contrived), as **break** and **return** exist in JavaScript, this is not a 'clean' way of coding.

In a snippet of code like this one it is easy to see what has happened, but when you have a large program full of many lines of code, it can become difficult with a **break** or a **return** to know where you have ended up after breaking out or returning from the loop and which line of code is being executed next.

A cleaner way of entering and exiting a loop

A cleaner way is the first example which has the condition statement to make sure that the code enters and exits at the same single point rather than a 'break out/return' early approach.

This also helps makes the code easier to read. The logic of the loop is in the first line of the snippet of code rather than being spread throughout the whole loop.

However, if you can make a good case for **break** or **return**, then as the coder that is your prerogative. After all, **break** and **return** have been included in the language for a reason[2].

Further explorations

If you feel that you want to explore more, then refer to the Mozilla JavaScript reference guide and try out the variations of the **for** and **while** loops.

Ask yourself:

- When and why would you use each statement.
- What are the disadvantages and advantages or these variations over the basic **while** and **for** loops?

Have fun!

[2] This is debateable as JavaScript was created by Brendan Eich in 10 days during in 1995, under duress, it was originally a script for the Netscape browser that *looked like Java* and *was easy for beginners* as he says in: Herman, D., (2012), *JavaScript 68 Specific Ways to Harness the Power of JavaScript*. https://github.com/hypnguyen1209/JS-ebook/blob/master/Effective%20JavaScript%2068%20Specific%20Ways%20to%20Harness%20the%20Power%20of%20JavaScript%20by%20David%20Herman%20-%202013.pdf (Retrieved 15th October 2025).