JavaScript Event Loop

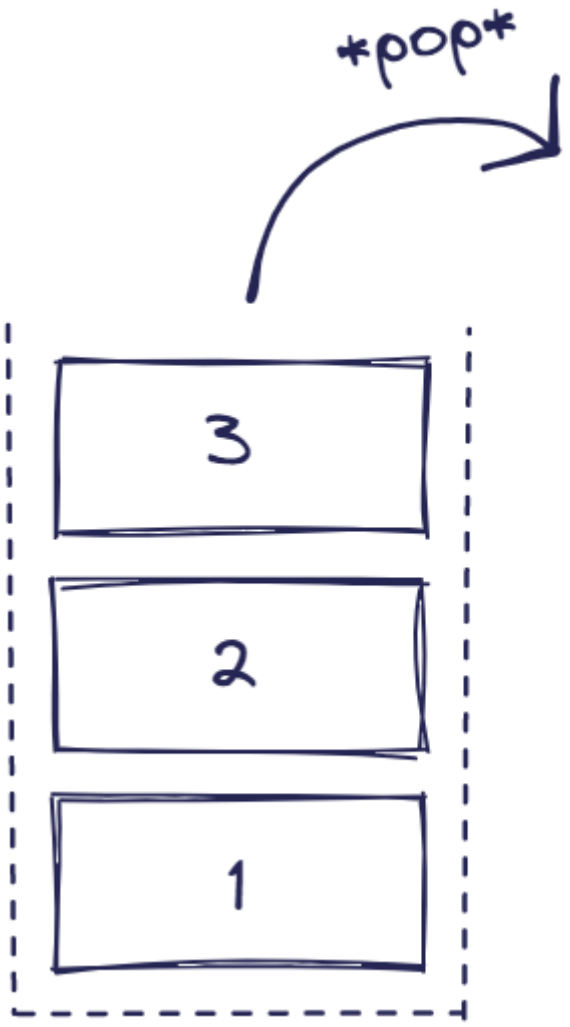
Learning Objectives

- Understand Call Stacks
- Understand the Event Loop

Material

Call Stack

A stack is a list-like data structure where the first element added is the last to be removed.



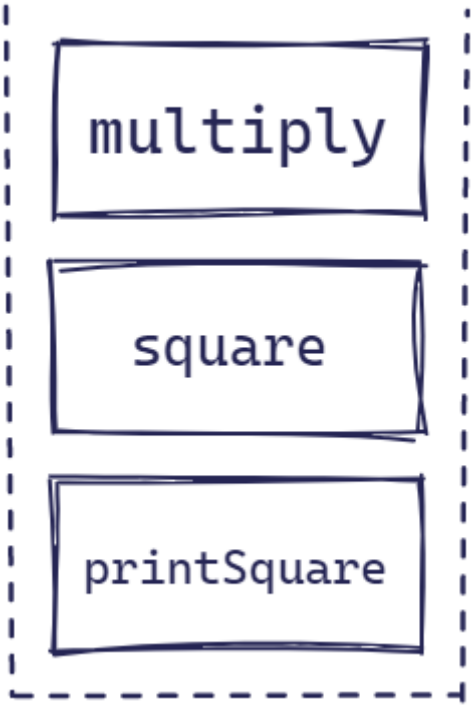
A programme's call stack is used to keep track of where a running subroutine (function) should hand back control once it has finished executing.

Take this code:

```
function multiply(a, b) {
    if (typeof a !== "number" || typeof b !== "number") {
        throw new Error("a and b must be numbers");
    }
    return a * b;
}
function square(n) {
    return multiply(n, n);
}
function printSquare(n) {
    const result = square(n);
    console.log(result);
}

printSquare(4);
```

Here, `multiply` is the last function to be called - whilst `multiply` is executing, the call-stack will look like this:



So `multiply` knows to hand its return value to `square` to continue the programme.

When a programme errors, we usually see the state of the call-stack printed in the logs:

```
function multiply(a, b) {
    if (typeof a !== "number" || typeof b !== "number") {
        throw new Error("a and b must be numbers");
    }
    return a * b;
}
function square(n) {
    return multiply(n, n);
}
function printSquare(n) {
    const result = square(n);
    console.log(result);
}

printSquare("not a number");
```

Which logs this when run:

```
Error: a and b must be numbers
    at multiply (C:\Users\admin.charlie\code\whiteboard\a.js:3:15)
    at square (C:\Users\admin.charlie\code\whiteboard\a.js:8:12)
    at printSquare (C:\Users\admin.charlie\code\whiteboard\a.js:11:20)
    at Object.<anonymous>
```

This tells us that `printSquare` called `square` which called `multiply` which errored.

The Event Loop

JavaScript is a single-threaded language. This means it has one call stack. JavaScript executes code in order and must finish executing a piece of code before moving onto the next. We call this synchronous, or blocking, execution. Other languages, such as C++ and Java, are multi-threaded and can execute multiple pieces of code at the same time. We refer to this as asynchronous or non-blocking execution.

The problem with single-threaded code is that, if a particular piece of code takes a long time to execute, it blocks any other code from running and so stalls our programme.

In the Browser (for front-end JavaScript) and Node (for back-end JavaScript), JavaScript runs inside a runtime environment. The runtime includes additional components which are not part of JavaScript. These include:

- APIs (e.g. web APIs for browser JavaScript: DOM, Timers, Fetch etc.)
- the Callback Queue - which holds callback functions from events which have just completed

- the Event Loop - which monitors the Callback Queue and the Call Stack and places callbacks from the Callback Queue onto the Call Stack when it is empty.

The Event Loop is what allows asynchronous (non-blocking) operations to occur — despite the fact that JavaScript is single-threaded.

Asynchronous (async) functions such as setting times, reading files etc. are recognised by Node.js and are executed in a separate area from the Call Stack. Node polls (regularly checks) the computer for the completion of the async operation and, once the operation is complete, the callback is placed into the callback queue. The Event Loop waits for the Call Stack to be empty and then moves the pending callback onto the Call Stack. It waits as otherwise it would randomly interrupt the execution of whatever sequence of function calls were queued up on the stack.

Timers are useful for illustrating the event loop. JavaScript does not have a built-in timer feature. It uses the Timer API provided by the Node.js or browser runtime to perform time-related operations. For this reason, timer operations are asynchronous.

`setTimeout(callback, ms)` can be used to schedule code execution after a designated amount of milliseconds. It accepts a callback function as its first argument and the millisecond delay as the second argument. When `setTimeout` is called, the timer is started in Web APIs part of the Node/Browser runtime. This frees the Call Stack up to continue executing code and only when the timer is complete and the Call Stack empty, does the callback get pushed to the Call Stack for execution.

Core Assignments

Loupe

Experiment writing scripts with Loupe until you feel comfortable with how the event loop operates.

Timing Predictions

For each of the following, predict **what** will be outputted to the console, and **when** it will happen. Run the code to test your predictions.

```
setTimeout(() => {
    console.log("hello");
}, 1000);
setTimeout(() => {
    console.log("goodbye");
}, 500);
```

```
setTimeout(() => {
    console.log("hello");
}, 0);
console.log("goodbye");
```

```
setTimeout(() => {
    console.log("hello");
    setTimeout(() => {
        console.log("goodbye");
    }, 2000);
}, 1000);
```

```
setTimeout(() => console.log("hello"), 1000);

let x = 1;
while (true) {
    x++;
}
```

```
const s = new Date();

setTimeout(function () {
    console.log("hello");
}, 100);

while (true) {
    if (new Date() - s >= 3000) {
        console.log("goodbye");
        break;
    }
}
```

Additional Resources

- What the heck is the event loop anyway? | Philip Roberts | JSConf EU