JavaScript Call Stack

Summary: in this tutorial, you will learn about the JavaScript Call Stack which is a mechanism to keep track of the function calls.

Introduction to JavaScript Call Stack

A call stack is a way for the JavaScript engine to keep track of its place in code that calls multiple functions. It has information on what function is being run and what functions are invoked from within that function.

The JavaScript engine also uses a **call stack** to manage execution contexts:

- The global execution context
- Function execution contexts

The call stack works based on the last-in-first-out (LIFO) principle.

When you execute a script, the JavaScript engine creates a global execution context and pushes it on top of the call stack.

Whenever a function is called, the JavaScript engine creates a function execution context for the function, pushes it on top of the call stack, and starts executing the function.

If a function calls another function, the JavaScript engine creates a new function execution context for the function being called and pushes it on top of the call stack.

When the current function completes, the JavaScript engine pops it off the call stack and resumes the execution where it left off.

The script will stop when the call stack is empty.

JavaScript call stack example

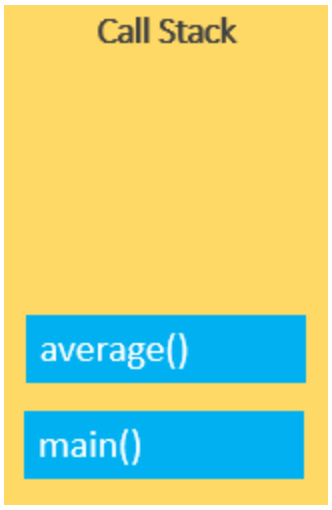
Let's start with the following example:

```
function add(a, b) {
    return a + b;
}

function average(a, b) {
    return add(a, b) / 2;
}

let x = average(10, 20);
```

When the JavaScript engine executes this script, it places the global execution context (denoted by `main()` or `global()` function on the call stack.

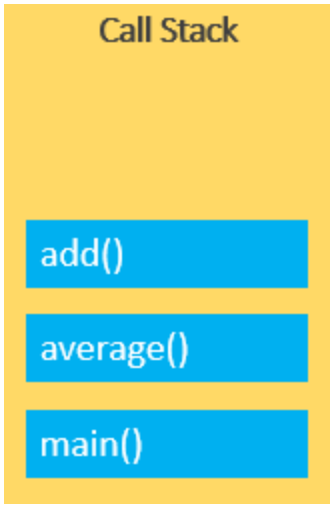
The global execution context enters the creation phase and moves to the execution phase.

The JavaScript engine executes the call to the `average(10, 20)` function and creates a function execution context for the `average()` function and pushes it on top of the call stack:

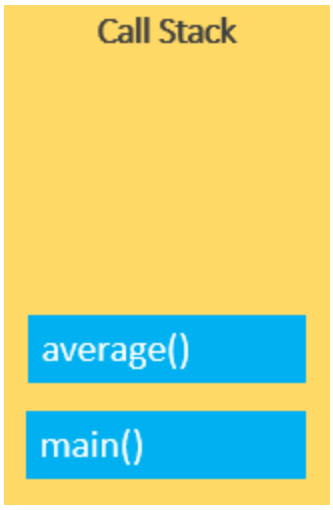


The JavaScript engine starts executing the `average()` since because the `average()` function is on the top of the call stack.

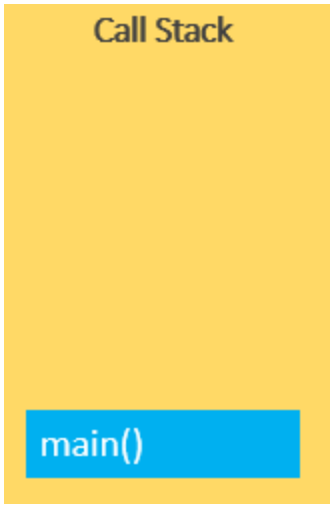
The `average()` function calls `add()` function. At this point, the JavaScript engine creates another function execution context for the `add()` function and places it on the top of the call stack:



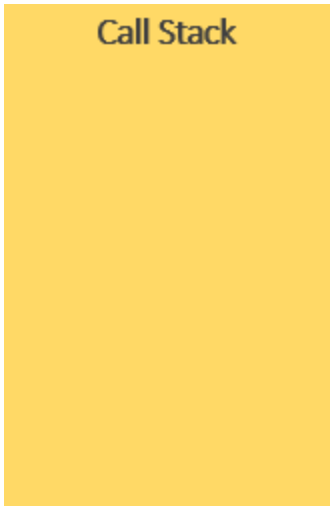
JavaScript engine executes the `add()` function and pops it off the call stack:



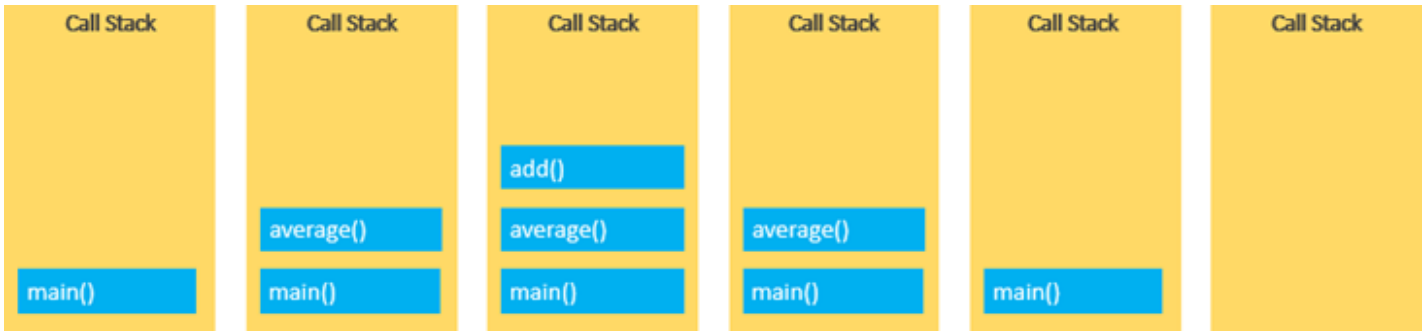
At this point, the `average()` function is on the top of the call stack, the JavaScript engine executes and pops it off the call stack.



Now, the call stack is empty so the script stops executing:



The following picture illustrates the overall status of the Call Stack in all steps:



Stack Overflow

The call stack has a fixed size, depending on the implementation of the host environment, either the web browser or Node.js.

If the number of execution contexts exceeds the size of the stack, a stack overflow error will occur.

For example, when you execute a recursive function that has no exit condition, the JavaScript engine will issue a stack overflow error:

```
function fn() {
    fn();
}

fn(); // stack overflow
```

Asynchronous JavaScript

JavaScript is a single-threaded programming language. This means that the JavaScript engine has only one call stack. Therefore, it only can do one thing at a time.

When executing a script, the JavaScript engine executes code from top to bottom, line by line. In other words, it is synchronous.

Asynchronous means the JavaScript engine can execute other tasks while waiting for another task to be completed. For example, the JavaScript engine can:

- Request for data from a remote server.
- Display a spinner

- When the data is available, display it on the webpage.

To do this, the JavaScript engine uses an event loop, which will be covered in the following tutorial.

Summary

- JavaScript engine uses a call stack to manage execution contexts.
- The call stack uses the stack data structure that works based on the LIFO (last-in-first-out) principle.