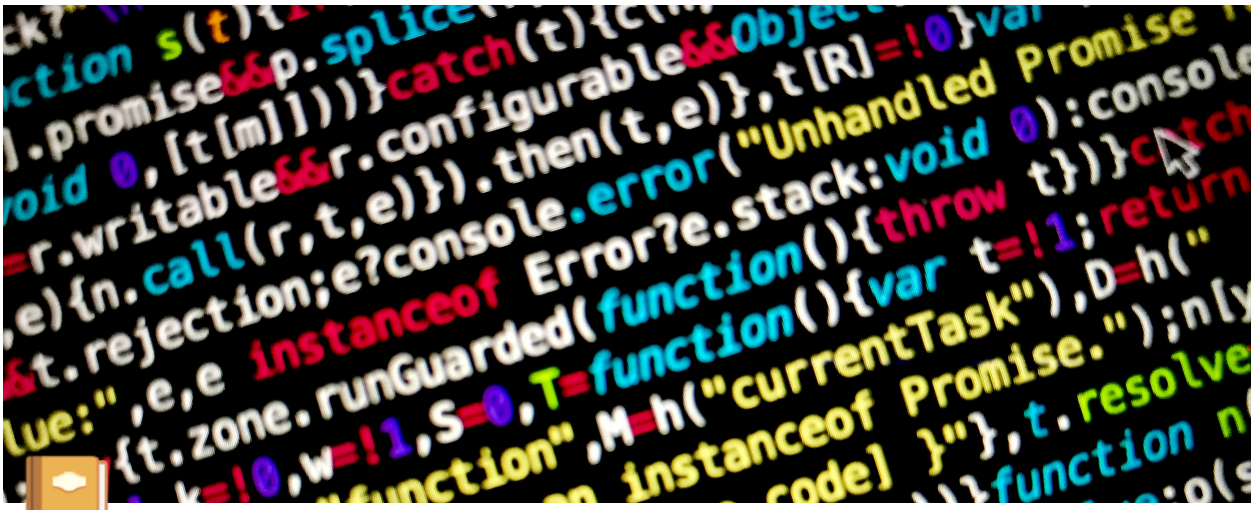


Javascript Boot Camp

Introduction

Javascript is a Dynamic Programming Language and Multi Paradiagm Language .

Dynamic Programming Language : We don’t declare the type of variable

Multiparadiagm : Its supports both Functional & OOP Concepts

Java Script Single threaded Language . Why ?

The JavaScript within a chrome browser is implemented by V8 engine.

The V8 engine has two parts:

1. Memory Heap
2. Call Stack

Memory Heap: It is used to allocate the memory used by your JavaScript program. Remember memory heap is not the same as the heap data structures, they are totally different. It is the free space inside your OS.

Call Stack: Within the call stack, your JS code is read and gets executed line by line.

Now, JavaScript is a single-threaded language, which means it has only one call stack that is used to execute the program. The call stack is the same as the stack data structure that you might read in Data structures. As we know stacks are FILO that is First In Last Out. Similarly, within the call stack, whenever a line of code gets inside the call stack it gets

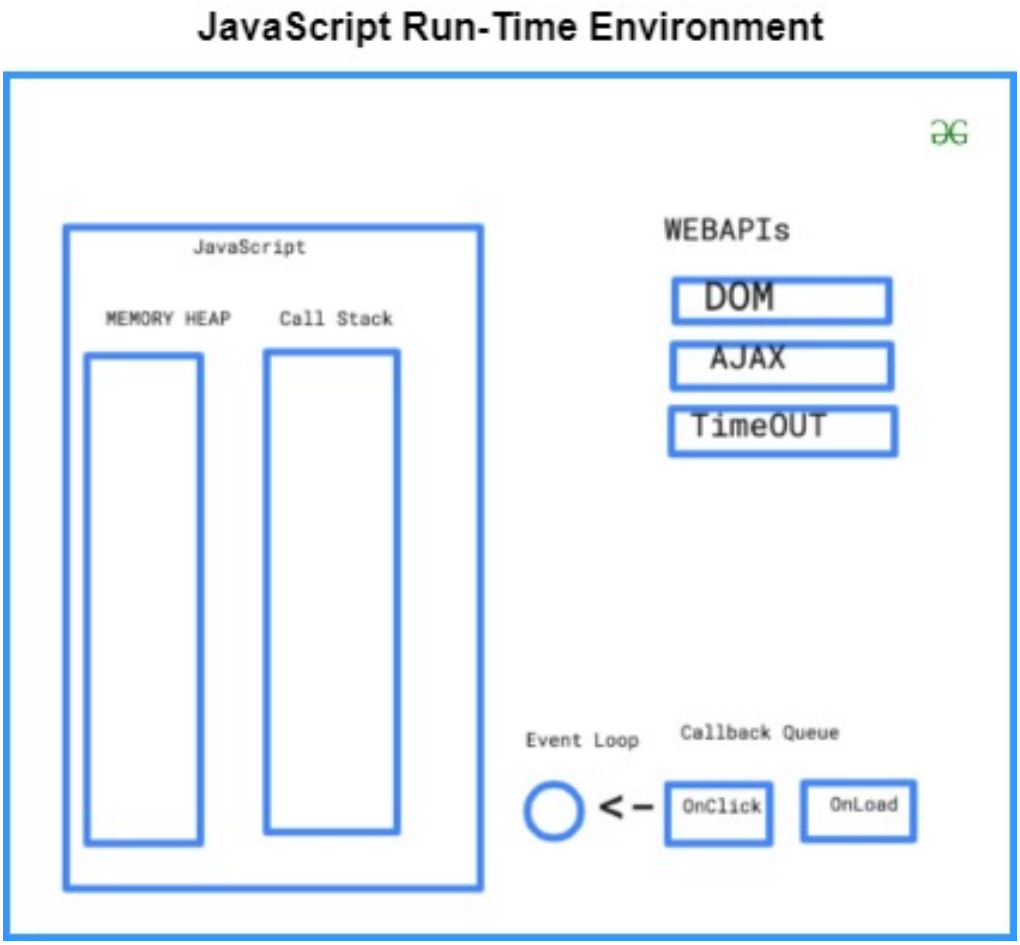
executed and move out of the stack. In this way, JavaScript is a single-thread language because of only one call stack.

JavaScript is a single-threaded language because while running code on a single thread, it can be really easy to implement as we don’t have to deal with the complicated scenarios that arise in the multi-threaded environment like deadlock.

Since, JavaScript is a single-threaded language, it is synchronous in nature. Now, you will wonder that you have used async calls in JavaScript so is it possible then?

So, let me explain to you the concept of async call within JavaScript and how it is possible with single-threaded language. Before explaining it you let’s discuss briefly why we require the async call or asynchronous calls. As we know within the synchronous calls, all the work is done line by line i.e. first one task is executed then the second task is executed, no matter how much time one task will take. This arises the problem of time wastage as well as resource wastage. These two problems are overcome by asynchronous calls, where one doesn’t wait for the one call to complete instead it runs another task simultaneously. So, when we have to do things like image processing or making requests over the network like API calls, we use async calls.

Now, coming back to the previous question of how to use async call within JS. Within JS we have a lexical environment, syntax parser, an execution context (memory heap and call stack) that is used to execute the JS code. But except these browsers also have Event Loops, Callback queue, and WebAPIs that is also used to run the JS code. Although these are not part of JS it also helps to execute the JS properly as we sometimes used the browser functions within the JS.



As you can see in the above diagram, DOM, AJAX, and Timeout are not actually part of JavaScript but the part of RunTime Environment or browser, so these can be run asynchronously within the WebAPI using the callback queue and again put in the call stack using event loop to execute.

Let us take an example to be very clear of the concept. Suppose we have the following piece of code that we want to execute in the JS run-time environment.

Let Vs Const vs Var

Let is scoped , Var is not scoped

```
 if (true){
    var var1 = " hello var";
}
console.log(var1)
```

```
if (true){
    let var2 = " hello let ";
    console.log(var2)
}
// console.log(var2) // Will not work bcs its let
```

```
//.Const can not be changed unless it is a object

const const_variable = 456
const_variable = 567  // Will not work

const const_object = { name:"Thor"}
const_object.name = "thor odinson"
```

```
console.log(x)
var x = "cool example"

//output
undefined
```

HOISTING

JavaScript **Hoisting** refers to the process whereby the interpreter appears to move the *declaration* of functions, variables or classes to the top of their scope, prior to execution of the code.

Hoisting allows functions to be safely used in code before they are declared.

Variable and class *declarations* are also hoisted, so they too can be referenced before they are declared. Note that doing so can lead to unexpected errors, and is not generally recommended.

```
printName("Rotimi")
// HOISTING IN JS

function printName(name){
    console.log(name)
}
```

Variable hoisting

Hoisting works with variables too, so you can use a variable in code before it is declared and/or initialized.

However, JavaScript only hoists declarations, not initializations! This means that initialization doesn't happen until the associated line of code is executed, even if the variable was originally initialized then declared, or declared and initialized in the same line.

Until that point in the execution is reached the variable has its *default* initialization (`undefined` for a variable declared using `var`, otherwise uninitialized).

Note: Conceptually variable hoisting is often presented as the interpreter "splitting variable declaration and initialization, and moving (just) the declarations to the top of the code".

Below are some examples showing what can happen if you use a variable before it is declared.

`var` hoisting

Here we declare and then initialize the value of a `var` after using it. The default initialization of the `var` is `undefined`.

```
console.log(num); // Returns 'undefined' from hoisted var declaration (not 6)
var num; // Declaration
num = 6; // Initialization
console.log(num); // Returns 6 after the line with initialization is executed.
```

The same thing happens if we declare and initialize the variable in the same line.

```
console.log(num); // Returns 'undefined' from hoisted var declaration (not 6)
var num = 6; // Initialization and declaration.
console.log(num); // Returns 6 after the line with initialization is executed.
```

If we forget the declaration altogether (and only initialize the value) the variable isn't hoisted. Trying to read the variable before it is initialized results in a `ReferenceError` exception.

```
console.log(num); // Throws ReferenceError exception - the interpreter doesn't know about `num`.
num = 6; // Initialization
```

Note however that initialization also causes declaration (if not already declared). The code snippet below will work, because even though it isn't hoisted, the variable is initialized and effectively declared before it is used.

```
a = 'Cran'; // Initialize a
b = 'berry'; // Initialize b

console.log(`${a}${b}`); // 'Cranberry'
```

`let` and `const` hoisting

Variables declared with `let` and `const` are also hoisted but, unlike `var`, are not initialized with a default value. An exception will be thrown if a variable declared with `let` or `const` is read before it is initialized.

```
console.log(num); // Throws ReferenceError exception as the variable value is uninitialized
let num = 6; // Initialization
```

Note that it is the order in which code is *executed* that matters, not the order in which it is written in the source file. The code will succeed provided the line that initializes the variable is executed before any line that reads it.

How the code is run in JS

```
console.log(x)
printNmae("Jay")
var x = 10
function printName(name){
  console.log(name)
  }
```

Usually , JS is has memory and code . It reads the code and puts in memory and starts executing as below

Memory Section in Javascipt theoretically Look like this

#line of Execution	Code	Code
1	x : Undefined	undefined
2	PrintName()	Jay

#line of Execution	Code	Code
3	x =10	

Note : After the execution , Memory is deleted

All the execution is loaded into Call Stack . After the last line of execution , the stack will be cleared . Since its LIFO .

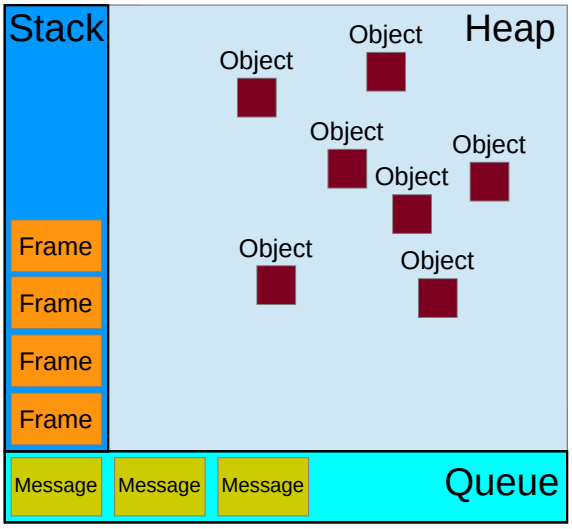
The event loop

JavaScript has a runtime model based on an **event loop**, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. This model is quite different from models in other languages like C and Java.

Runtime concepts

The following sections explain a theoretical model. Modern JavaScript engines implement and heavily optimize the described semantics.

Visual representation



Stack

Function calls form a stack of *frames*.

```
function foo(b) {
  const a = 10;
  return a + b + 11;
}

function bar(x) {
  const y = 3;
  return foo(x * y);
}

const baz = bar(7); // assigns 42 to baz
```

Order of operations:

1. When calling `bar`, a first frame is created containing references to `bar`'s arguments and local variables.
2. When `bar` calls `foo`, a second frame is created and pushed on top of the first one, containing references to `foo`'s arguments and local variables.
3. When `foo` returns, the top frame element is popped out of the stack (leaving only `bar`'s call frame).
4. When `bar` returns, the stack is empty.

Note that the arguments and local variables may continue to exist, as they are stored outside the stack — so they can be accessed by any nested functions long after their outer function has returned.

Heap

Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.

Queue

A JavaScript runtime uses a message queue, which is a list of messages to be processed. Each message has an associated function that gets called to handle the message.

At some point during the event loop, the runtime starts handling the messages on the queue, starting with the oldest one. To do so, the message is removed from the queue and its corresponding function is called with the message as an input parameter. As always, calling a function creates a new stack frame for that function's use.

The processing of functions continues until the stack is once again empty. Then, the event loop will process the next message in the queue (if there is one).

Event loop

The **event loop** got its name because of how it's usually implemented, which usually resembles:

```
while (queue.waitForMessage()) {
  queue.processNextMessage();
}
```

`queue.waitForMessage()` waits synchronously for a message to arrive (if one is not already available and waiting to be handled).

"Run-to-completion"

Each message is processed completely before any other message is processed.

This offers some nice properties when reasoning about your program, including the fact that whenever a function runs, it cannot be preempted and will run entirely before any other code runs (and can modify data the function manipulates). This differs from C, for instance, where if a function runs in a thread, it may be stopped at any point by the runtime system to run some other code in another thread.

A downside of this model is that if a message takes too long to complete, the web application is unable to process user interactions like click or scroll. The browser mitigates this with the "a script is taking too long to run" dialog. A good practice to follow is to make message processing short and if possible cut down one message into several messages.

Adding messages

In web browsers, messages are added anytime an event occurs and there is an event listener attached to it. If there is no listener, the event is lost. So a click on an element with a click event handler will add a message — likewise with any other event.

The function `setTimeout` is called with 2 arguments: a message to add to the queue, and a time value (optional; defaults to `0`). The *time value* represents the (minimum) delay after which the message will be pushed into the queue. If there is no other message in the queue, and the stack is empty, the message is processed right after the delay. However, if there are messages, the `setTimeout` message will have to wait for other messages to be processed. For this reason, the second argument indicates a *minimum* time — not a *guaranteed* time.

Here is an example that demonstrates this concept (`setTimeout` does not run immediately after its timer expires):

```
const seconds = new Date().getTime() / 1000;

setTimeout(() => {
  // prints out "2", meaning that the callback is not called immediately after 500 milliseconds.
  console.log(`Ran after ${new Date().getTime() / 1000 - seconds} seconds`);
}, 500)

while (true) {
  if (new Date().getTime() / 1000 - seconds >= 2) {
    console.log("Good, looped for 2 seconds");
    break;
  }
}
```

Zero delays

Zero delay doesn't mean the call back will fire-off after zero milliseconds.
Calling `setTimeout` with a delay of `0` (zero) milliseconds doesn't execute the callback function after the given interval.

The execution depends on the number of waiting tasks in the queue. In the example below, the message `"this is just a message"` will be written to the console before the message in the callback gets processed, because the delay is the *minimum* time required for the runtime to process the request (not a *guaranteed* time).

The `setTimeout` needs to wait for all the code for queued messages to complete even though you specified a particular time limit for your `setTimeout`.

```
(() => {

  console.log('this is the start');

  setTimeout(() => {
    console.log('Callback 1: this is a msg from call back');
  }); // has a default time value of 0

  console.log('this is just a message');

  setTimeout(() => {
    console.log('Callback 2: this is a msg from call back');
  }, 0);

  console.log('this is the end');

})();

// "this is the start"
// "this is just a message"
// "this is the end"
// "Callback 1: this is a msg from call back"
// "Callback 2: this is a msg from call back"
```

Several runtimes communicating together

A web worker or a cross-origin `iframe` has its own stack, heap, and message queue. Two distinct runtimes can only communicate through sending messages via the `postMessage` method. This method adds a message to the other runtime if the latter listens to `message` events.

Never blocking

A very interesting property of the event loop model is that JavaScript, unlike a lot of other languages, never blocks. Handling I/O is typically performed via events and callbacks, so when the application is waiting for an IndexedDB query to return or an XHR request to return, it can still process other things like user input.

Legacy exceptions exist like `alert` or synchronous XHR, but it is considered good practice to avoid them. Beware: exceptions to the exception do exist (but are usually implementation bugs, rather than anything else).