**How JavaScript Works**

The language can be described by the following phrases:

1. **Multi-Paradigm**

High abstractions, with intuitive syntax

1. **Interpreted**

Translate each of the instructions (lines of code) one-by-one.

This is different from compiled languages like C, where you’ll have to translate the whole file into Machine code or Binary, and then the machine will directly execute this file (.exe)

However, JavaScript is not entirely interpreted, as modern engines has used some of the features of a compiler to add additional performance on the language.

1. **Dynamic Weakly Typed language**

This means we do not use any explicit type definition (like C).

**Data types become known at runtime**.

1. **Multi-Paradigm**

Allows us to combine coding styles including Declerative functional or Imparative Object-Oriented approaches.

1. **Prototype Inheritance based**

JavaScript is based on Prototype Inheritance. Everything in JavaScript is an object, and each Object holds a link to its prototype. It then becomes a prototype chain (nested objects)

**Implementation details**

How the interpreters should be implemented, how to allocate memory,… are instead handled by the browsers. 2 of the most popular engines are SpiderMonkey from Firefox and V8 from Chrome. They both do:

1. **Just-In-Time (JIT) Compilation**

- In the case of V8, it will compile JavaScript code into native machine code before running it (instead of interpreter) 🡪 improve performance in browsers and .node

1. **Single-threaded**

JavaScript can run just one task at the same time. We can test this by putting while(true){} on the web console, and that we cannot press on any buttons.

So how can we do so many things on the web? 🡪 We need the **event loop**

When executing the JavaScript code, 2 regions of memory are allocated on the machine: Stack and Heap

- Call Stack is designed to create frames for functions and pop them as they return values.

- What if we get into more complicatied things, like an object that might be referenced by multiple function calls outside of the local context 🡪 Heap, which stores objects in our code. The special thing about the Heap is that it is

1. **Garbage-collected**

This means that V8 or the JS Runtime will automatically free up memories when it’s no longer referenced 🡪 Don’t need to manually allocate and then free up memory in C language.

1. **Non-Blocking Event loop**

In the most basic sense, it’s just a while loop that waits for the messages from a queue. Then it processes their synchronous instructions to completion.

while (queue.waitForMessage()) {

queue.processNextMessage;

}

We create events by, for example, clicking a button. Then the runtime will process whatever JavaScript defines as the callback for that event. This is what makes JavaScript non-blocking. Because the only thing it ever does is listen to events and handle callbacks, so it’s never actually waiting for the return value of a function.

The only thing it’s actually waiting for is the CPU to process your synchronous code.

Imagine the first iteration of the event loop:

- It will first handle all the synchronous code in the script.

- It checks whether there are any messages or callbacks in the queue ready to be executed.

- We can demonstrate this behaviour very simply by adding a setTimeOut on the top of the script for 0 seconds

setTimeOut(() => console.log(‘do this first?’), 0)

console.lof(‘let there be JavaScript!’)

//let there be JaveScript

//do this first?

🡪 The setTimeOut will not be executed until JavaScript finishes the first iteration of synchronous code.

\*What makes it special is that you can offload long running jobs to completely separate thread pools. In the browser you can make a HTTP call that takes a few seconds to resolve, or in nodejs you might need to interact with the file system, and you can do these things without blocking the main JavaScript thread.

**\*\*\*Promises and Microtask queue.**

If we go back to our code and add a Promise.resolve after setTimeOut

setTimeOut(() => console.log(‘do this first?’), 0)

Promise.resolve()

.then(v => console.log(‘do this second?’);

console.lof(‘let there be JavaScript!’)

//let there be JaveScript

//do this second?

//do this first?

There’s something called the micro task queue for promises which has priority over the main task queue used for DOM APIs and set timeouts, etc. 🡪 the handler for the promise will be called back first in this case.

🡪 For each round of the event loop:

1. Run sync code
2. Run Promise or microtask callbacks
3. Run async task callbacks (setTimeOuts/DOM APIs)