**Programming language:** In simple terms, programming languages are set of instructions or code which tells a computer what it needs to do. So basically, we provide a logic or instruction to the computer to perform some task to get the desired output from it. When we need to write a CD or burn a CD or when we need to paste something in pen drive these all instruction is given through some software which involves some instructions or set of code and this software communicate to the hardware. Programming languages are high-level languages that need to be converted into machine level language because a computer can only understand machine level language or binary language (0 and 1). So we write the instructions in human-readable form and then we hit the compile button to convert this into machine level language which a computer can understand and then the computer performs the task. This conversion is done by the compiler which scans the complete code in one go and if it finds any error it immediately throws all errors. Examples are [Java](https://www.geeksforgeeks.org/java/), [C](https://www.geeksforgeeks.org/c-programming-language/), [C++](https://www.geeksforgeeks.org/c-plus-plus/), [C#](https://www.geeksforgeeks.org/csharp-programming-language/). Programming languages are most widely used to make software or drivers.

**Scripting Language:**As the name suggest, it’s all about giving the script to perform some certain task. Scripting languages are basically the subcategory of programming languages which is used to give guidance to another program or we can say to control another program, so it also involves instructions. It basically connects one language to one another languages and doesn’t work standalone. [Javascript](https://www.geeksforgeeks.org/javascript-tutorial/" \t "_blank), [PHP](https://www.geeksforgeeks.org/php/), [Perl](https://www.geeksforgeeks.org/perl-tutorial/), [Python](https://www.geeksforgeeks.org/python-programming-language/), VBScript these all are the examples of scripting language. Scripting languages need to be interpreted (Scanning the code line by line, not like compiler in one go) instead of compiled. There is ***no scope of compiler*** in scripting languages. Scripting languages are most widely used to create a website.

**Markup Languages:** Markup languages are completely different from programming languages and scripting languages. Markup languages prepare a structure for the data or prepare the look or design of a page. These are ***presentational*** languages and it doesn’t include any kind of logic or algorithm, for example, HTML. [HTML](https://www.geeksforgeeks.org/html-tutorials/) is not asking any kind of question to the computer or it’s not comparing things and it’s not asking any logical question. It’s just used to represent a view inside a web browser. It tells the browser how to structure data for a specific page, layout, headings, title, table and all or styling a page in a particular way. So basically it involves formatting data or it controls the presentation of data. Examples of Markup languages are HTML, [CSS](https://www.geeksforgeeks.org/css-tutorials/)or XML. These languages are most widely used to design a website.

From the above definition, we can summarize Programming language, Scripting language, and Markup languages from below images.

**Conclusion:** So we can say that *all the scripting languages are programming languages but all the programming languages are not scripting languages*. C cannot be called a scripting language, it is just a programming language but we can call JavaScript or Php programming or scripting languages. Also, there is no need to compile scripting languages it only needs to be interpreted. Scripting languages are generally slower than programming languages because compiled programs are first converted into machine code. On the other hand, markup languages are just used to define the structure of data which doesn’t require any logic or algorithm

JavaScript is an **interpreted** language. This means we do not have to compile the JavaScript source code before sending it to the browser. An interpreter can take the raw JavaScript code and run it for you.

JavaScript is also a dynamically typed language, unlike **C** and **C++**. This means variables declared using var can store any type of data type like int, string, boolean and also complex data types like object and array.

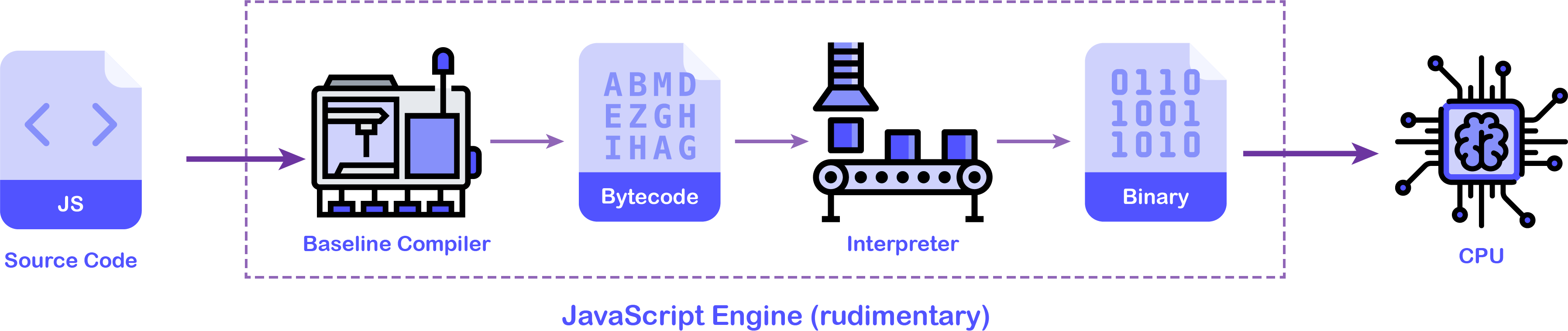
The **lack of type system** is what makes JavaScript slow to run. A statically typed language can produce a much efficient machine code because of the information it has about the data like its **type** and **size**.

The Anatomy of the JavaScript engine

EcmaScript specification tells how JavaScript should be implemented by the browser so that a JavaScript program runs exactly the same in all the browsers, but it does not tell how JavaScript should run inside these browsers. It is up to the browser vendor to decide.

Every browser provides a [**JavaScript engine**](https://en.wikipedia.org/wiki/JavaScript_engine) that runs the JavaScript code. The Netscape browser used the **[SpiderMonkey](https://en.wikipedia.org/wiki/SpiderMonkey" \t "_blank)** JavaScript engine. This engine was a rudimentary interpreter with no optimizations. Running the JavaScript code with this engine was slow but it worked.

Image for post



(**Netscape/SpiderMonkey** JavaScript engine)

As you can see from the diagram above, the job of the first JavaScript engine was to take the JavaScript source code and compile it to the binary instructions (*machine code*) that a CPU can understand.

A rudimentary JavaScript engine contains a **baseline compiler** whose job is to compile JavaScript source code into an intermediate representation (***IR***) which is also called the **bytecode** and feeds this bytecode to the interpreter.

The interpreter takes this bytecode and converts to the machine code which is eventually run on the machine’s hardware (CPU).

💡 *This is just like how****Java****works but the bytecode generation is done by the programmer and bytecode is shared universally rather than the source code.*

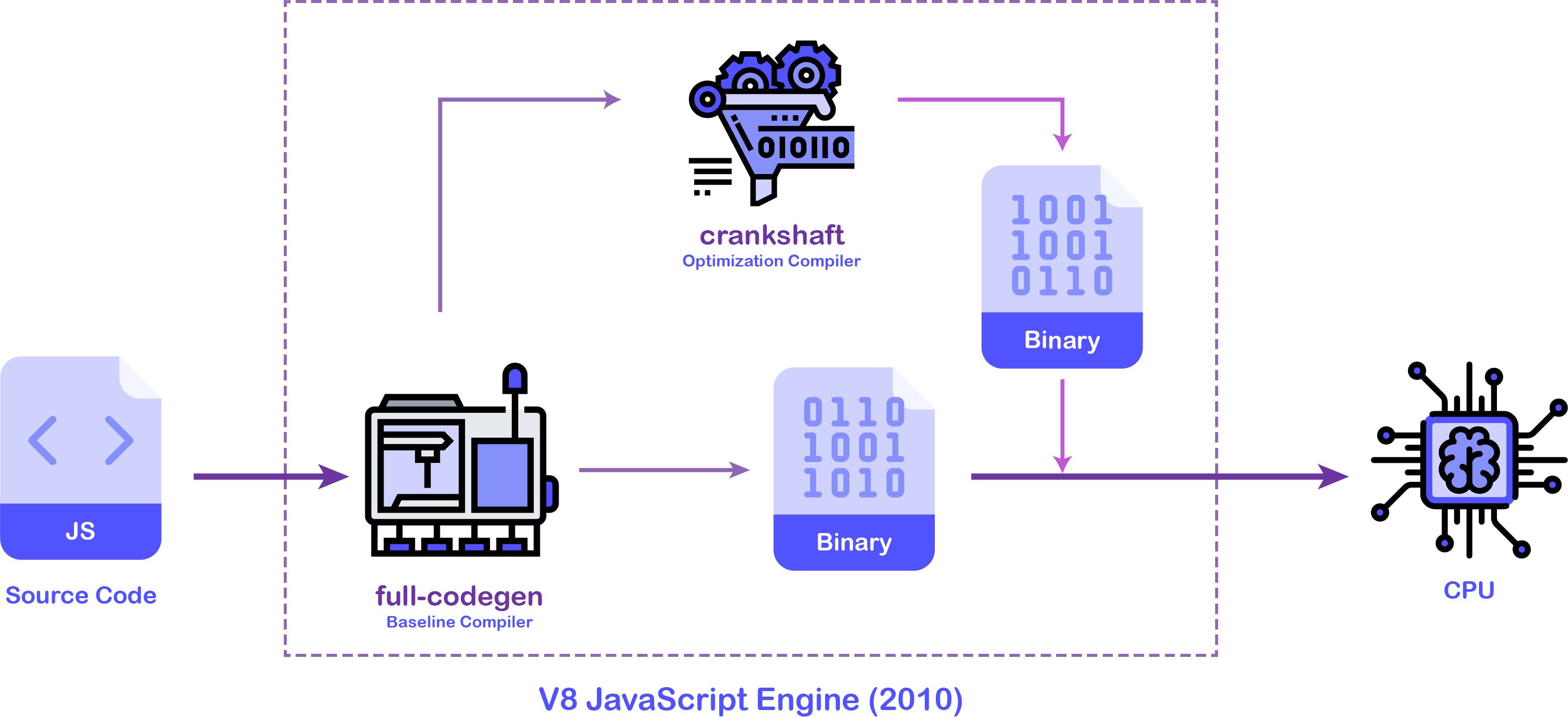
A baseline compiler’s job is to compile code as fast as possible and generate **less-optimized** bytecode (*or machine code in other cases*). Since the interpreter has an unoptimized bytecode to work with, the application speed will be slow, however, the application bootstrap time will be very less.

💡 ***[SpiderMoney](https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey" \t "_blank)****JavaScript has evolved into a piece of complex machinery to produce highly optimized machine code and currently used in the*[***Firefox***](https://www.mozilla.org/en-US/firefox/new/)*browser. You can follow*[***this documentation***](https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Getting_SpiderMonkey_source_code)*for the source code.*

When it comes to a highly dynamic and interactive web application, the user experience is very poor with this model of JavaScript execution. This problem was faced by Google’s Chrome browser while displaying **Google Maps** on the web. To increase the JavaScript performance on the web, they had to come up with a better approach.

Google Chrome from the early days uses the [**V8**](https://en.wikipedia.org/wiki/V8_(JavaScript_engine)) JavaScript engine. In the beginning, to improve the JavaScript performance, they added two pieces in their JavaScript engine pipeline as shown below.

Image for post



(**2010 V8** JavaScript Engine)

In the **2010 version** of the **V8** JavaScript engine, there were two main pieces of machinery that did the heavy lifting for the engine. The **full-codegen** was the **baseline compiler** whose job was to spit out unoptimized machine code as fast as possible for faster application bootstrap.

As the application was running, the **crankshaft** compiler would kick in and optimize the source code and replace the parts of the machine code generated by the baseline compiler. This optimization would result in better application performance as better and better machine code is generated.

💡 *However, this process comes with the cost of large CPU overhead and memory consumption. Hence V8 has to come up with another model.*

The above version of the JavaScript engine does not contain an interpreter. This is a [**JIT**](https://en.wikipedia.org/wiki/Just-in-time_compilation) (*Just-In-Time*) compilation model as code is compiled to the machine level on the fly and later optimized, also to the machine code.

How JavaScript is optimized?

There are various criteria for optimizing JavaScript code. Before JavaScript code is passed to the interpreter or baseline compiler, it has to first get parsed into an Abstract Syntax Tree ([**AST**](https://en.wikipedia.org/wiki/Abstract_syntax_tree)) which is a tree-like structure of the code.

When we run a JavaScript application, we do not need all the code at the application startup time. For example, if we have a function that is called on the user action, like a button click, that code can be parsed later.

Identifying things that need to be parsed immediately and generating machine code is the best strategy for faster application bootstrap.

Sometimes, JavaScript code contains unnecessary complex logic that can be simplified. For example, a for loop to increment an integer can be inlined using + operations n number of times. This process is called [**Loop unrolling**](https://en.wikipedia.org/wiki/Loop_unrolling). Similar optimizations can be made using [**function inlining**](https://en.wikipedia.org/wiki/Inline_expansion)

The lack of type system in JavaScript is what makes JavaScript engine produce less optimized machine code. Hence, based on already defined values, a JavaScript engine can guess the data types of the variables and generate better machine code.

(**2017 V8** JavaScript Engine)

As you can see from the above figure, the **V8** team introduced a new interpreter pipeline **Ignition** whose job was to generate the bytecode from the JavaScript source code using a baseline compiler and later interpret that bytecode using an interpreter.

The **TurboFan** optimization compiler can optimize this bytecode in the background (*in separate threads*) as the application is running and generate a very optimized machine code that will be replaced eventually.

**Turbofan** receives the profiling data from the **Ignition** interpreter and looks for the code that is **Hot**. It can make the

engine for **Internet Explorer**.

Some JavaScript engines might look complex perhaps because they have multiple baseline and optimization compilers but in a nutshell, guesses on how to optimize the code better (*by guessing the data types*) and optimize or de-optimize the code.

What about other JavaScript engines?

We have seen a broad overview of how the **V8 JavaScript engine** works. A similar model is followed by other browser vendors like **SpiderMonkey** engine for **Firefox** browser and [**Chakra**](https://en.wikipedia.org/wiki/Chakra_(JScript_engine)) they follow the same model of optimization.

**V8** is one of the most popular JavaScript engines, perhaps because it is developed by Google. But the **V8** engine is constantly evolving and becoming faster. Apart from Google Chrome, [**Chromium**](https://en.wikipedia.org/wiki/Chromium_(web_browser)) project, [**Electron.js**](https://electronjs.org/)**,** and server-side JavaScript runtime [**Node.js**](https://nodejs.org/) use the V8 engine.

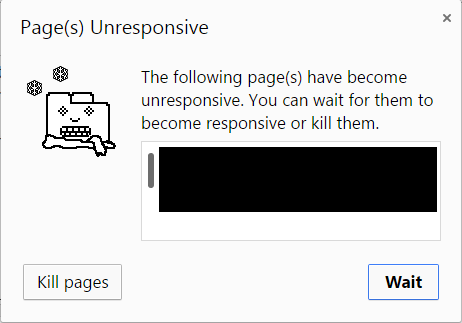
💡 *Microsoft**is planning to ship the*[***Edge browser***](https://support.microsoft.com/en-in/help/4501095/download-microsoft-edge-based-on-chromium)*based on****Chromium****.*

JavaScript at runtime

There are many passionate developers, working on front-end or back-end, who devote their lives to protect the realm of JavaScript. JavaScript is very easy to understand and is an essential part of front-end development.

But unlike other programming languages, **it’s single-threaded** langauge at runtime. That means the code execution will be done one piece at a time. Since code execution is done sequentially, any code that takes a long time to execute will block anything that needs to be executed after that. Hence sometimes you see below the screen while using Google Chrome.

Image for post



(Chrome’s page unresponsive dialog)

When you open a website in the browser, it uses a single JavaScript execution thread. That thread is responsible to handle everything, like scrolling the web page, printing something on the web page, listen to DOM events (*like when the user clicks a button*), and doing other things.

But when JavaScript execution is **blocked**, the browser will stop doing all those things, which means the browser will simply freeze and won’t respond to anything until that task is completed.

You can see that in action using below eternal while loop.

while(true){}

Any code after the above statement won’t be executed as the while loop will loop infinitely until system is out of resources. This can also happen in an infinitely **recursive function call**.

Thanks to modern browsers, as not all open browser tabs rely on single JavaScript thread. Instead, they use separate JavaScript thread **per tab** or **per domain**. In the case of Google Chrome, you can open multiple tabs with different websites and run above the eternal while loop.

That will only freeze the current tab where that code was executed but other tabs will function normally. Any tab having page opened from the same domain / same website will also freeze as Chrome implements a **one-process-per-site** policy and a process uses the same JavaScript execution thread.

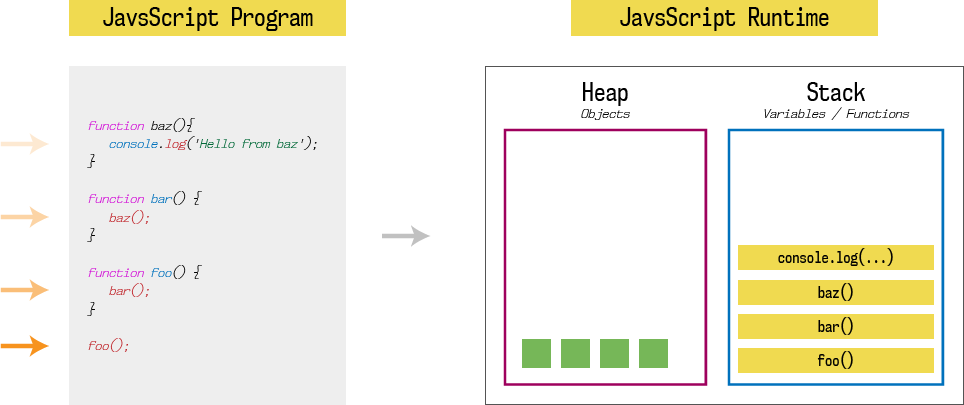
To visualize, how JavaScript executes a program, we need to understand JavaScript runtime and different components that play a part in it. So lets write a simple JavaScript program to visualize this.

(source: [gist.github.com](https://gist.github.com/thatisuday/7d2432656da8d66e3226e3bb75d963f1))

Here we have a simple JavaScript program that has three functions, viz. foo, bar and baz. The function foo calls the function bar and then function bar calls the function baz which logs something to the console using the console.log function provided by the runtime.

When we run this program, first the function foo gets called and then the call chain begins until the console.log() is executed. Let’s visualize this using a diagram and inspect various components of the runtime.

Image for post



(JavaScript Runtime Environment)

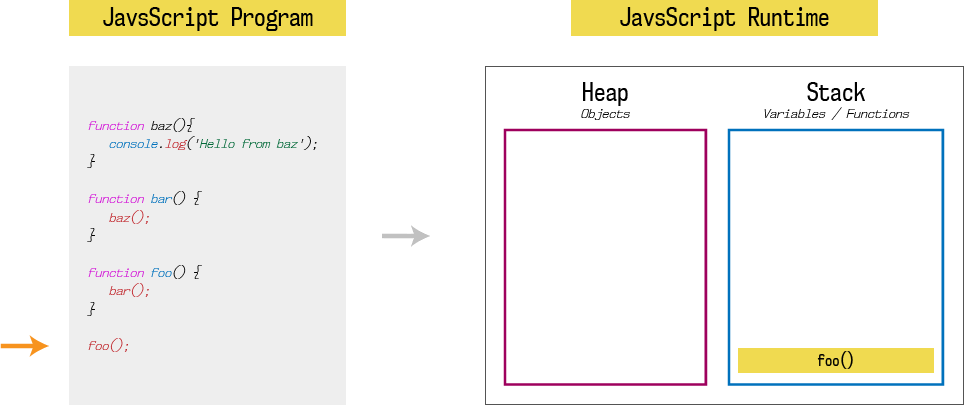
Like any other programming language, JavaScript runtime has one stack and one heap storage. A **heap** is a free memory storage unit where you can store memory in random order. Data that is going to persist in for a considerable amount of time go inside the heap. Heap is managed by the JavaScript runtime and cleaned up by the garbage collector. I am not going to explain much more about the heap, you can read it [here](https://hashnode.com/post/does-javascript-use-stack-or-heap-for-memory-allocation-or-both-cj5jl90xl01nh1twuv8ug0bjk).

What we are interested in is **stack**. A stack is **LIFO** (last in, first out) data storage that stores the current function execution **context** of a program. In the above example, when our program is loaded into the memory, it starts execution from the first function call which is foo().

Hence, the first stack entry is foo(). Since foo function calls bar function, second stack entry is bar(). Since bar function calls baz function, third stack entry is baz(). And finally, baz function calls console.log, fourth stack entry is console.log('Hello from baz').

Until a function returns something (*while the function is executing*), it won’t be popped out from the stack. The stack will pop entries one by one as soon as that entry (*function*) returns some value, and it will continue pending function executions.

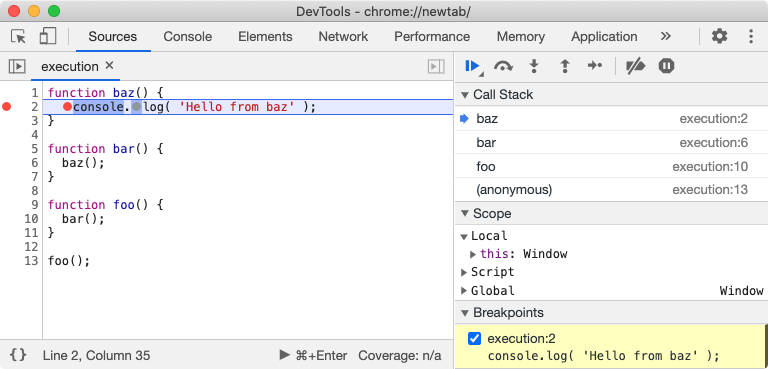
Image for post



(Stack frames)

Each entry in the stack is called a **stack frame**. A stack frame contains the information of the function call such as arguments of the function call, locals of the function, return address (*where the return value will be consumed*), and other information of the function.

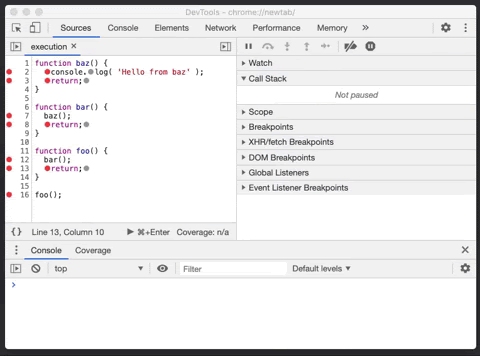
Image for post



(Chrome’s DevTools Snippet)

As you can see from the above screenshot, when we add a breakpoint at the console.log function call, Chrome’s DevTools displays the **Call Stack** (*on right*) that contains the stack frames up until the current function execution.

Image for post



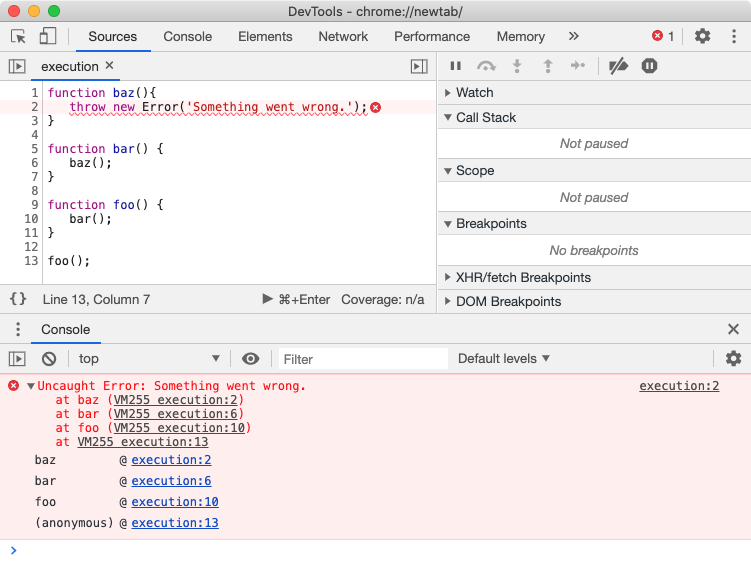
(Chrome’s DevTools)

If any function call at a given stack frame produces an error, JavaScript will print a **stack trace** which is nothing but a snapshot of code execution up until that stack frame.

(source: [gist.github.com](https://gist.github.com/thatisuday/155b0f6a157ef1ecbdef358e69c7edee))

In the above program, we threw error inside the baz function. Therefore when JavaScript encounters the error, it will print the below stack trace to display what went wrong and where.

Image for post



(Chrome’s DevTools Snippet)

As you can see from the above screenshot, Chrome’s DevTools not only display the error message but also shows the stack track up until the stack frame where the error was thrown. If baz function calls another function after the error is thrown, it won’t be pushed to the stack.

💡 *If you want to learn more about JavaScript’s stack track and how to get more out of it, read*[***this***](https://v8.dev/docs/stack-trace-api)*V8’s documentation on stack trace API.*

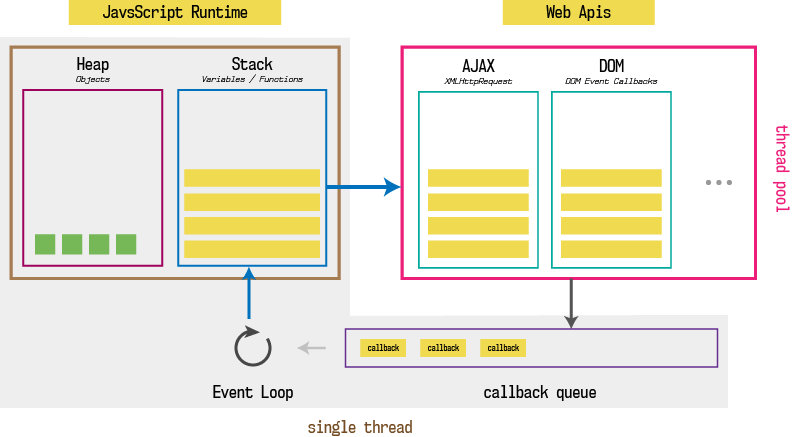
Since JavaScript is single-threaded, it has only one stack and one heap per process. Hence, if any other program wants to execute something, it has to wait until the previous program is completely executed. This thread is commonly known as **main thread** or **main execution thread**.

So let’s think of one scenario. What if a browser sends an HTTP request to load some data over the network or to load an image to display on the web page. Will the browser freeze until that request is resolved? If it does, then it’s very bad for user experience.

A browser comes with a JavaScript engine that is responsible to execute any JavaScript contained inside a web application (*web page*). For example, Google Chrome uses [**V8**](https://v8.dev/) JavaScript engine.

But guess what, the browser uses more than just the JavaScript engine. This is what browser under the hood looks like.

Image for post



(browser under the hood)

Looks really complex but it is very if you understand one piece at a time and they work together in harmony. JavaScript runtime actually consists of 2 more components viz. **event loop** and **callback queue**. The callback queue is also called a **message queue** or **task queue**.

Apart from JavaScript engine, browser contains different applications which can do a variety of things like send HTTP requests, listen to DOM events, delay execution using setTimeout or setInterval, caching, database storage, and much more. These features of the browser help us create rich web applications and better user experience.

But think about this, if the browser had to use the same JavaScript thread for the execution of these tasks, then user experience would be terrible. For example, if the browser had to use the same JavaScript thread to perform a task when the HTTP network response is received, then the web page would be irresponsive for seconds or even for minutes.

Hence browser implements their own logic to perform these operations such as sending HTTP requests and listening to their responses. These operations do not block the JavaScript main execution thread since they are spawned on different threads managed by the browser and JavaScript has no idea of it.

A browser may use a low-level language like C or C++ to implement these features for performance benefits and give us the clean JavaScript API to execute these operations from the JavaScript. For example, [fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) API is provided by the browser to send HTTP requests. These APIs are known as **Web APIs** since they are not part of the JavaScript specifications.

These Web APIs are **asynchronous**. That means you can instruct these APIs to do something in the background and return data once done, meanwhile we can continue further execution of JavaScript code. While instructing these APIs to do something in the background, we have to provide a **callback function**. Responsibility of a callback function is to execute some JavaScript code in the main Javascript thread once Web API is done with its work. Let’s understand how all pieces work together.

So when you call a function, it gets pushed to the stack. If that function contains a Web API call, JavaScript will delegate control of it to the Web API with a callback function and move to the next lines until the function returns something. Now the callback function is with the Web API which is performing its operation on a separate thread, separate from the main thread.

Once the function hits the return statement, that function is popped from the stack and move to the next stack entry. Meanwhile, Web API is doing its job in the background and remembers what callback function is associated with that job. Once the job is done, Web API binds the result of that job to the callback function and publishes a message to the **message queue** (*AKA****callback queue***) with that callback function.

The only job of the **event loop** is to look at callback queue and once there is something pending in callback queue, push that callback to the **stack**. The event loop pushes one callback function at a time, to the stack, **once the stack is empty**. Later, the stack will execute the callback function.

Let’s see how everything works step by step using setTimeout Web API. The setTimeout Web API is mainly used to execute something after a few seconds (any time period). This execution happens once all the code in the program is done executing (*when the stack is empty*). The syntax for setTimeout function is as below.

setTimeout(callbackFunction, timeInMilliseconds);

The callbackFunction is a callback function which will execute after timeInMilliseconds. Let’s modify our earlier program and use this API.

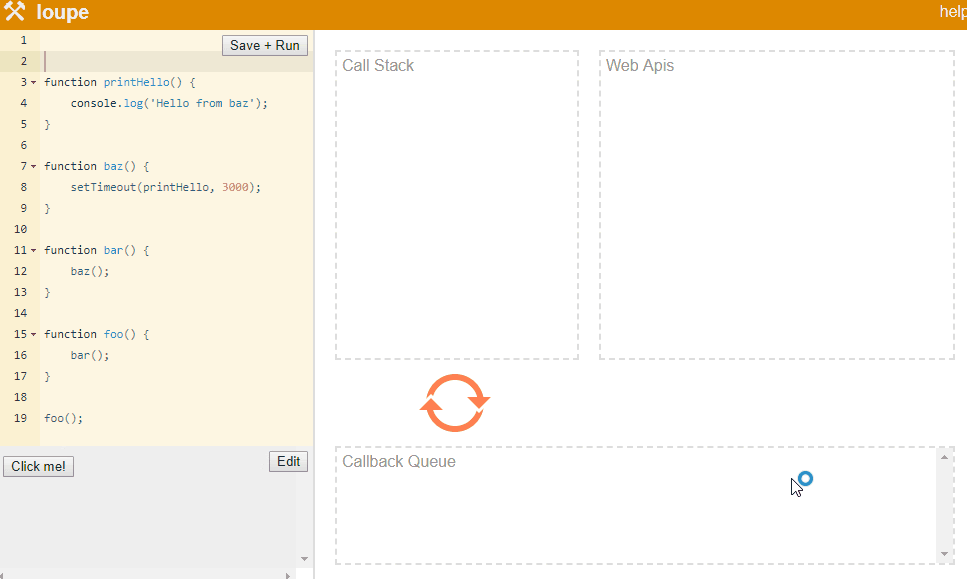
(source: [gist.github.com](https://gist.github.com/thatisuday/6de7c945a0a795596edb22d49f04442a))

The only modification done to the program is, we delayed printHello function execution by *3 seconds*. In this case, the stack will keep building up like foo() => bar() => baz(). Once baz starts executing and hits setTimeout API call, JavaScript will pass the callback function to the Web API and move to the next line.

Since there is no next line, function returns and the stack will pop baz, then bar and then foo function calls. Meanwhile, Web API is waiting for 3 seconds to pass. Once 3 seconds are passed, it will push this callback to callback queue and since the stack is empty, the event loop will put this callback back on the stack where the execution of this callback will happen.

[***Philip Robers***](http://latentflip.com/) has created an amazing online tool to visualize how JavaScript works underneath. Our above example is available at [this link](http://latentflip.com/loupe/?code=ZnVuY3Rpb24gcHJpbnRIZWxsbygpIHsNCiAgICBjb25zb2xlLmxvZygnSGVsbG8gZnJvbSBiYXonKTsNCn0NCg0KZnVuY3Rpb24gYmF6KCkgew0KICAgIHNldFRpbWVvdXQocHJpbnRIZWxsbywgMzAwMCk7DQp9DQoNCmZ1bmN0aW9uIGJhcigpIHsNCiAgICBiYXooKTsNCn0NCg0KZnVuY3Rpb24gZm9vKCkgew0KICAgIGJhcigpOw0KfQ0KDQpmb28oKTs%3D!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D).

Image for post



(<http://latentflip.com/loupe/>)

The event loop and callback queue are the pieces of the same puzzle. They are not part of the Javascript engine, rather they sit outside JavaScript engine and normally provided by the runtime such as a web browser or Node.js. The event loop uses JavaScript engine’s APIs to communicate with it and provide callback functions to execute.

Even loop inside Node.js

When it comes to **Node.js**, it has to do more because the Node promises more. In the case of a browser, we are limited to what we can do in the background. But in node, we can pretty much do most of the things in the background, even it is a simple JavaScript program. But, how does that work?

Node.js uses **Google’s V8 engine** to provide JavaScript runtime and employes its own event loop using the **[libuv](https://github.com/libuv/libuv" \t "_blank)** library (*written in c*). Node follows the same callback approach like Web APIs and works in a similar fashion as the browser.