**Synchronous vs. Asynchronous JavaScript -** [Vivian Yim](https://medium.com/@vivianyim?source=post_page-----de4918e8ad62--------------------------------) Jan 28, 2022

JavaScript is a single-threaded language, but at the same time, can be made to be non-blocking, asynchronous, and concurrent.

Chart, waterfall chart

Description automatically generated

The time it takes to run asynchronous programs is equal to its most time-consuming task since it is able to execute its task in parallel compared to synchronous programs.

**Synchronous JavaScript:** As the name suggests, synchronous execution is when statements are executed one at a time in the order in which they appear. Each statement waits for the previous to finish, also known as “blocking”.

For example, take a look at the following code:

Shape

Description automatically generated with medium confidence

In the statements above, the console will log out “First”, “Second”, and “Third” in the exact order it is written. However, to understand more complicated synchronous code, consider this new example.

Text

Description automatically generated

In the example above the console will still log out “First”, “Second”, “Third” in the exact same order as the first example. However, this example is meant to demonstrate how JavaScript utilizes the call stack to keep track of the global execution context and the function execution contexts. The call stack operates on a Last In, First Out(LIFO) principle which means the last function that gets pushed onto the stack is the first to be popped out.

When the code is run:

1. Function definitions are parsed and evaluated.
2. function print\_everything() will then get called. A stack frame — a bundle of contiguous data including the function, its parameters, variables, etc — will be created and pushed onto the top of the call stack.
3. function print\_everything() will begin executing since it is at the top of the call stack and invoke function print\_first().
4. A new stack frame for print\_first() will be created, and then the function will be pushed onto the stack and start executing.
5. Within print\_first(), a call to console.log() is made. Another function execution is created and pushed onto the stack.
6. Console.log() prints out “First” to the console, and then gets popped off the stack when this is completed
7. function print\_first() then finishes execution, and is popped off the stack as well
8. Steps 3–7 will repeat for both print\_second() and print\_third().
9. The end result is “First”, “Second”, and “Third” printed in sequential order

I’ll expand more on the importance of the call stack in a future blog where I dive more into the implementation of asynchronous JavaScript!

While synchronous, blocking code is straightforward to write and easy to understand, there can be some nasty side effects — especially for web applications.

Say your application needs to fetch a big chunk of data from an API, which can take minutes to return. If this were run in a blocking context, then your application would be totally unresponsive while waiting for this call to complete, leading to a terrible user experience! This is where asynchronous execution can come to the rescue.

**Asynchronous JavaScript:**In an asynchronous, or non-blocking, environment, code can be executed immediately instead of waiting for the previous to finish. Having code that runs asynchronously allows a user to fetch data from the server, execute a function with a delay, and deal with multiple requests simultaneously. This helps to reduce the waiting time for the user.

Text

Description automatically generated

In the example above, the output will first be “First”, followed by “and then followed by “and then Hello World” two seconds later. The important thing here, however, is that we can do some work in the downtime between the two console.log statements!

Table

Description automatically generated

This image shows how asynchronous functions can handle more operations

**Why Asynchronous?**

In the past websites were often static HTML pages. Pages would appear frozen or unresponsive since fetching data from a Web API could take an indeterminate amount of time depending on several factors, like the size of the data and the speed of the network connection. However, present-day web applications are able to prevent blocking behavior by utilizing asynchronous code which makes them more dynamic and interactive. This allows a user to continue using the browser normally while the asynchronous operations are being processed.

The most common triggers for asynchronous JavaScript operations are:

1. Event Loop
2. Callback
3. Promises
4. Async/Await

I will be covering these topics in a separate post!