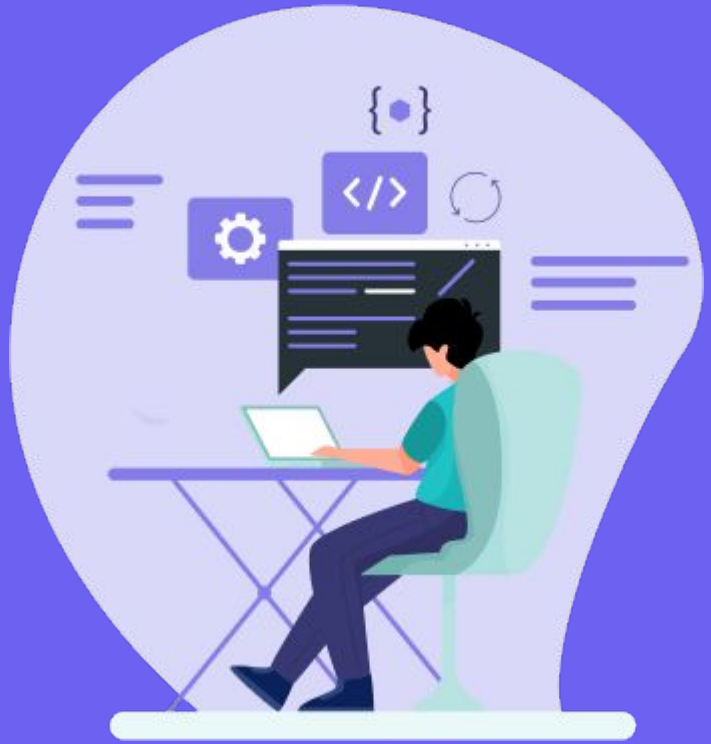
Asynchronous Programming in JS - 1

Relevel
by Unacademy



Synchronous

Synchronous, in general, means one at a time.
Synchronous is the sequential flow of program execution in **a programming context**.
The program/code will be read in sequence from top to bottom and executed line by line, with one thing getting executed at a time.
The Javascript engine waits for the code to get executed completely before moving to the next line.
So Synchronous programming executes each line of code at a time in a sequence.

Example:

Cons of synchronous programming:
Blocks code; The next line of code is not executed until the previous is complete.

```
console.log('Hi');

function greet(){
    console.log('Welcome to
Grandline');
}
greet();

// Hi
// Welcome to Grandline
```

Asynchronous

Asynchronous, in general, is not occurring at the same time. Asynchronous in a programming context is the execution of code/function without waiting for the other code/functions to execute. This enables the Javascript engine to execute the code without halting the code execution process.

Asynchronous

In Javascript, there are different ways to achieve asynchronous execution.

Callbacks: We need to note that Callbacks are used inside Browser API/Web API functions or events. These APIs internally accept callback functions to achieve asynchronous programming. They have methods such as setTimeout and event handlers like click, mouse over, scroll, etc.

Promises: Promises are objects in JS that let us perform asynchronous operations.

Async/Await: Async Await keywords were introduced in Javascript in ECMAScript 2017. This made asynchronous programming easier and is considered a better alternative to promises.

CallBacks

Callbacks are functions that are passed as arguments to other functions and then invoked in the outer function to perform its operation.

Also, callbacks take time to execute and are not executed immediately; they are passed at first and are invoked later.

Callbacks should be implemented asynchronously. Else the event loop in the JS engine has to wait until the callback is executed, which may cause undesired output, or our program may become unresponsive.

Example:

```
function greet(callbackFn){
    callbackFn();
    console.log('Welcome to
Grandline');
}

greet(function(){
    console.log('Hi')
});

// Hi
// Welcome to Grandline
```

```
// Same code with arrow syntax

function greet(callbackFn){
    callbackFn();
    console.log('Welcome to
Grandline');
}

greet(() => console.log('Hi'));

// Hi
// Welcome to Grandline
```

Call Stack

Call Stack is the stack data structure maintained by the Javascript engine. It operates in a LIFO manner (Last in First Out). It has the purpose of tracking the current function being executed. Call Stack is also known as Function execution stack.

- When the Javascript engine runs the code, the first thing that happens is an execution context is created precisely Global Execution Context (GEC) whenever the code execution is started.
- So the first thing that's pushed into the call stack is GEC.
- After the execution of code, the execution context is popped out.
- Remember, each function has its execution context.
- *Execution context* is the environment in which the code is executed.

Example:

```
function greet(){

    return function() {
        console.log('Welcome to Grandline');
    }
}

const result = greet();
result(); // Welcome to Grandline
```

Explanation

- First, the Global execution context will be created and pushed into the call stack.
- The JS engine starts reading the code from top to bottom and creates a memory space for the function definition with the label greet. It does not execute the function as it's still not invoked.
- Next it reads const result = greet(); . As soon as it reads greet(), a local execution context for greet() is pushed into the call stack above the GEC, and the output of the greet function is stored into the result variable. It does not execute the inner function as it is still not invoked; it simply returns the function and stores it. As soon as it returns the function, the local execution context of greet is popped out, and only GEC is present in the stack.
- Now the next line is read, and the local execution context for the result() is pushed into the call stack.

Explanation

- After its execution, where it logs Welcome to Grandline, the local execution context for the result() is also popped out of the stack.
- Only the Global execution context that was pushed down the call stack will be present, and as there is no further code to execute, this will also be popped, and the call stack is empty now.

Callback queue

Callback functions do not enter the call stack for execution directly instead a callback function is registered in the Web API and then the result of callback is stored now the callback is entered to **Callback Queue**. It is a queue data structure that has entries in **FIFO manner (First in First out)** and stores all the callback functions ready for execution and once the call stack is empty they are deleted from the queue and pushed to the stack for execution.

Callbacks with Async Example:

```
console.log('Hi');

function callback(){
    console.log('Welcome to Grandline');
}

setTimeout(callback, 2000);

console.log('Join our Pirate Crew & be our Nakama');

// Hi
// Join our Pirate Crew & be our Nakama
// Welcome to Grandline (with a delay of 2 seconds)
```

Explanation

- First the Global execution context will be created and pushed into the call stack.
- The JS engine now starts reading the code from top to bottom and executes the console.log('Hi'); line, further it creates a memory space for the function definition with the label callback. It does not execute the function as it is still not invoked.
- Next it reads setTimeout(callback, 2000); . In this piece of code we are using the WEB API method setTimeout(). We pass a timer to wait for 2000 milliseconds (2 seconds) and a callback function.
- A callback is registered in the Web API and it will store the result of the callback function but this callback will wait inside the callback queue till the timer is expired and the call stack is empty; it enters the call stack only when both of these conditions are fulfilled.

Explanation

- It is a common scenario to have nested callbacks and in this case all the nested callbacks will wait inside the callback queue in FIFO manner (First in First out) and then be pushed to the call stack i.e the function that came first to the queue will be deleted from the queue and pushed to the stack first.

Explanation

- Here the Event Loop comes into the picture as the job of the call stack is to track the current function in execution. It's the job of Event Loop to keep a check if the call stack is empty and if there is any callback function waiting to be executed in the callback queue.
- Also Javascript does not wait for anyone; it executes whatever is there in the call stack. So even if we are waiting for the timer to expire there is no function in the call stack at this moment except the GEC so the JS engine will start executing the next line and log console.log('Join our Pirate Crew & be our Nakama'); to the console. Now after logging this the call stack is having GEC and as there is nothing to execute the GEC will be popped out.
- Now the call stack is empty and the timer is also expired so the callback function is eligible to enter the call stack, now the callback gets deleted from the callback queue and callback execution context is created and the callback function is then pushed to the stack.

Explanation

- Callback function is executed and it logs Welcome to Grandline .
  After this the callback function is popped out from the call stack and it is empty again.

CallBacks hell

- As mentioned earlier, it is a common scenario to have nested callbacks and all the nested callbacks will wait inside the callback queue and be pushed to the call stack in FIFO manner (First in First out).
- This complex nesting of callbacks makes our code difficult to maintain and debug.
- It is also prone to more errors if not written carefully.
- This scenario in which we have complex nested callbacks is referred to as Callback hell. Here, each callback takes arguments that are the result of previous callbacks.
- This kind of callback structure forms a pyramid like structure.

```
// Callbacks hell example
function cheese(){
    console.log('Add lots of cheese');
}
function patty(cheese){
    console.log('Add veggies and patties');
    cheese();
}
function bun(patty){
    console.log('Take two fresh buns');
    patty(cheese);
}
function burger(bun){
    setTimeout(() => {
        bun(patty);
        console.log('Our Burger is ready');
    }, 2000);
}
burger(bun);
console.log('Lets make a Burger');
```

```
//Output

// Lets make a Burger// (after delay of 2000ms)

// Take two fresh buns
// Add veggies and patties
// Add lots of cheese
// Our Burger is ready
```

Explanation

- First the GEC is pushed to the call stack and all the function definitions are stored in the memory with their respective function names as labels.
- Then we invoke burger(bun); which is a normal/synchronous function and it will go inside the call stack above the GEC and create its local execution context and start its execution.
- It accepts bun callback function as an argument and encounters a setTimeout() function inside burger() function definition now the burger() will be popped out of the call stack as its executed and setTimeout() will be pushed on the stack with its local context and the Anonymous callback

```
A() => {
        bun(patty);
        console.log('Our Burger is ready');
}
```

  will be registered in the Web API and then pushed to the callback queue.
- The job of setTimeout() is to notify the JS engine to have a timer of 2000ms and it has already pushed the callback function to the queue so setTimeout() is executed and is popped out of the stack.

Explanation

- Now only GEC is present in the call stack and as JS engine does not wait for the callback to get completed and timer to get expired it moves to the next line and logs Lets make a Burger , at this moment there is nothing to execute and the GEC is popped out from the stack and the call stack is empty.
- The event loop is checking at every instance of time if the call stack is empty and if there is some callback waiting in the callback queue.
- As the stack is empty the Anonymous callback A() is pushed to the stack and its deleted from the callback queue.It starts executing the function and encounters the invocation of bun(patty) function as this is synchronous in nature now the bun(patty) will get pushed to the stack above the callback A(), gets executed and logs Take two fresh buns on the console and invokes patty(cheese).
- Now patty(cheese) will be pushed to the stack above bun(patty) and logs
- Add veggies and patties and invokes cheese();

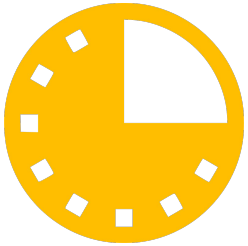
Explanation

- cheese() will be pushed above patty(cheese) and will log
- Add lots of cheese.
- Execution of cheese() is done and it will be popped out from the stack and then patty(cheese) and bun(patty) will be popped out as well.
- At this moment only the anonymous callback function A() resides in the call stack and finally logs Our Burger is ready now all the executions are done and the call stack as well callback is empty.
- The point to note here is that all the result of the code in the callback function of setTimeout() will be stored and is executed only after the timer expires.

Each callback is waiting for the result of the previous callback and this creates a nested Callback Hell. Such callback hell creates confusion and it is difficult to understand the flow to code execution.

We can prevent the callback hell by using Promises or await inside of async functions. We will see promises and use of Async and Await in the next class.

Timers in JS



In Javascript there are two Timing Events:

- **setTimeout()**
- **setInterval()**

These timing events are not part of the JS engine but actually they are part of Browser and are available to us in the Global context associated with the Window object of the browser and in NodeJS they are associated with the Global object.

setTimeout()

setTimeout(): It is basically a function that is used to execute a function or code after a specified delay.

Syntax: **setTimeout(function, milliseconds, args)**

function: It is a mandatory field and its the function that will be executed after the delay or expiry of timer.

milliseconds: It is an optional field. It is the delay time we express in milliseconds. It is the time for which the timer should wait before the passed function or code is executed.

args: It is an optional field and is the arguments that are passed to the function defined in the setTimeout.

Example:

```
function burger(){
    setTimeout(() => {
        console.log('Our Burger
is ready');
    }, 2000);
}

burger();
// (after delay of 2 seconds)
// Our Burger is ready
```

clearTimeout()

clearTimeout() is the method used to cancel/clear the timeout even before it is executed.

- It accepts the variable to which we assign the setTimeout(). This variable returns the reference of setTimeout(). Here we use clearTimeout(getFood) with getFood variable.

- It does not clear/cancel the delay in the setTimeout it actually cancels the setTimeout() event itself.

Example:

```
const getFood = setTimeout((food) => {
    console.log('Our ' + food +' is ready');
}, 2000, 'Pizza');

clearTimeout(getFood);
// Nothing happens and code just exits
```

setInterval()

setInterval(): It is basically a function that is used to execute a function or code repeatedly after a specified interval of time.

Syntax: **setInterval(function, milliseconds, args)**

function: It is a mandatory field and its the function that will be executed after the delay or expiry of timer.

milliseconds: It is an optional field. It is the interval time we express in milliseconds. It is the interval time after which the passed function or code is executed every time.

args: It is an optional field and is the arguments that are passed after the timer expires, to the function defined in the setInterval.

Example:

```
setInterval((food) => {
    console.log('Our ' + food +' is ready');
}, 2000, 'Pizza');

// Gets logged every 2 seconds

// Our Pizza is ready
// Our Pizza is ready
// Our Pizza is ready
// Our Pizza is ready
// Our Pizza is ready
// ^C
```

clearInterval()

clearInterval() is the method used to stop/clear the interval.

- It accepts the variable more precisely intervalID to which we assign the setInterval(). Here we use clearInterval(getFood) with getFood variable to stop the setInterval().

- It does not clear the delay in the setInterval. It actually stops the setInterval() event itself before execution.

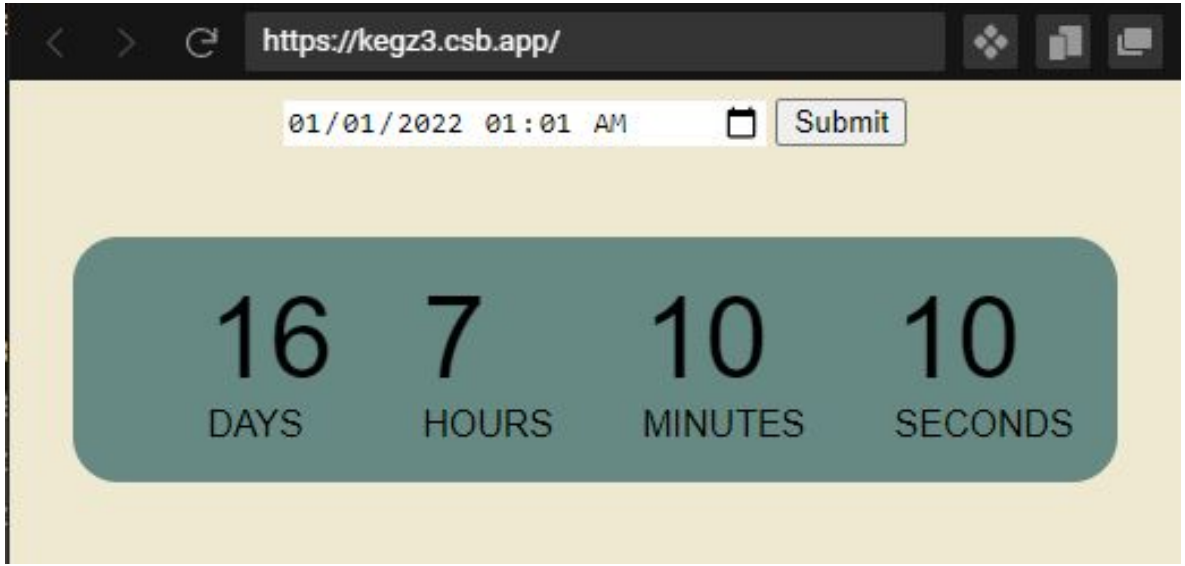
Example:

```
const getfood = setInterval((food) => {
    console.log('Our ' + food +' is ready');
}, 2000, 'Pizza');

clearInterval(getfood);

// Nothing happens and code just exits
```

Implementing the Countdown Timer App

Complete App :

https://codesandbox.io/s/countdown-timer-app-base-complete-kegz3



Practice/HW

1. Program to demonstrate call back hells.
2. Program to demonstrate timing events in JS.
3. Program to demonstrate synchronous and asynchronous functions.

Thank you