**JS MILESTONE 8**

FUNCTION EXPRESSIONS

Recursion

Closures

● Closures and Variables

● The this Object

● Memory Leaks

Mimicking Block Scope

Synchronous Programming

Asynchronous Programming

Callback

Avoid callback hell

Promise

Bluebird Promises

Async Await

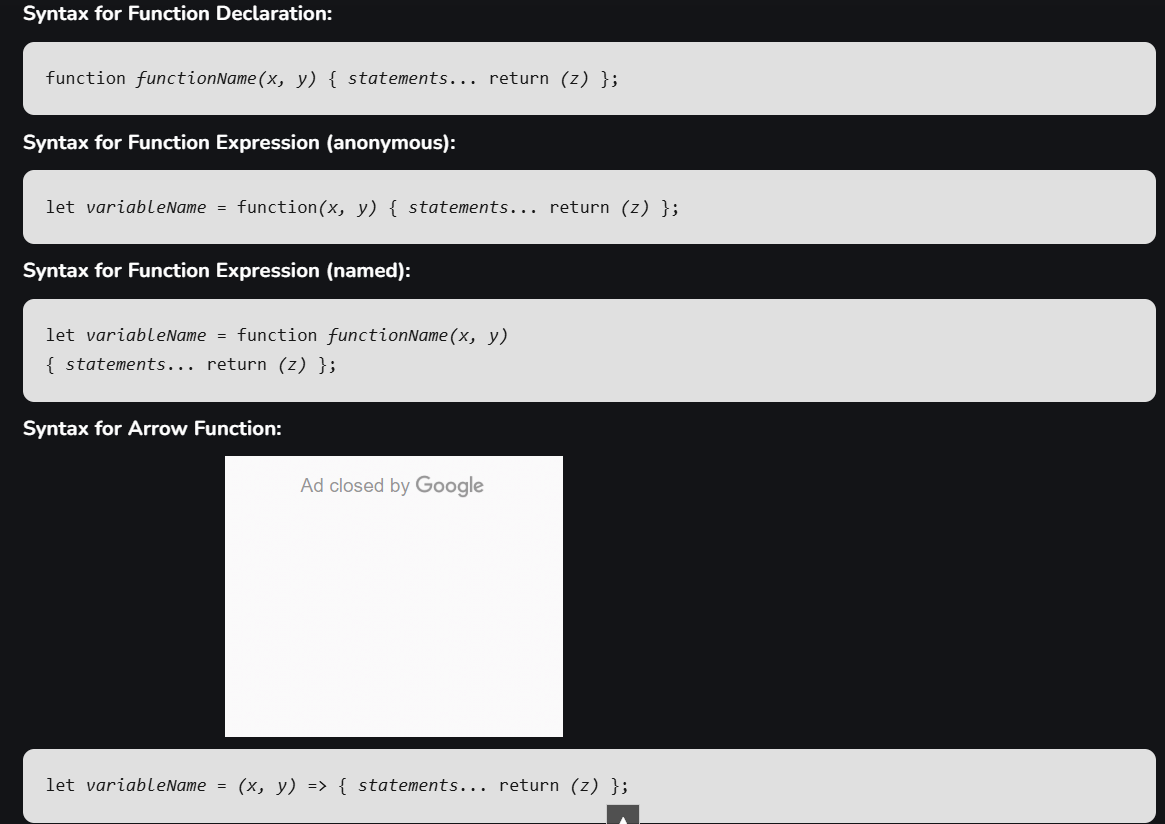
Convert Callback to promise

Convert promise to async await

Why async needed in js?

**FUNCTION EXPRESSION:**

* Function expression is used for callbacks and closures. So that function can be used as values when passed as a parameter.
* It can be named or anonymous.
* Function can be used as a value for a variable.
* Function expression has to be defined first before calling it or before using it as a parameter.



**RECURSION:**

* A programming technique where a function calls itself to solve smaller instances of a problem.
* Base case and recursive case must be given in a recursion.

**CLOSURES:**

* It is a function that remembers and accesses variables from its outer scope even after the scope has exited.
* A closure is created when a function is defined inside another function, and the inner function "remembers" the variables from the outer function, even after the outer function is done executing.

function outerFunction(outerVariable) {

return function innerFunction(innerVariable) {

console.log(`Outer Variable: ${outerVariable}`);

console.log(`Inner Variable: ${innerVariable}`);

};

}

const closure = outerFunction("outside");

closure("inside");

**MEMORY LEAKS:**

* A memory leak in JavaScript occurs when objects are allocated memory, but that memory isn't released even after the objects are no longer required.
* It may occur sue to uncleared intervals and timers, unremoved eventlisteners.
* This can be overcome by using clearInterval() and clearTimeout()
* Event listeners can be removed by using .remove(), .removeEventistener()
* In closures, if any eventlisteners are added, removing reference to outer function doesn’t delete the reference of eventlisteners.
* This can be overcome by removing eventlisteners at correct time.

**MIMICKING BLOCK SCOPE:**

* Before the introduction of let and const in ES6, JS had only function scope and global scope.
* Variables declared with var are function scoped.
* To mimic block scope in older JS, we need to use Immediately invoked function Expression (IIFEs)
* BLOCK SCOPE – a variable that is accessible only within {…}.
* Var can be accessed even after a block. Let is not accessible after a block.

Here var i is shared across all iterations. So when i=0,callback is queued

i=1, callback is queued

i=2, callback is queued.

i=3, condition fails

After 1sec, the queued callback is executed. When seeing i=3. So callback (ie 3 console is logged as 3)

for (var i = 0; i < 3; i++) {

setTimeout(function () {

console.log(i); }, 1000); }

* To overcome above problem either let should be used. Because let creates new instance foe each iteration.
* Before introduction of let IIFEs is used.

Here let i is not shared across all iterations. So when

i=0,callback is queued, and remembers value of i as 0

i=1, callback is queued, and remembers value of i as 1

i=2, callback is queued, and remembers value of i as 2

i=3, condition fails

After 1sec, the queued callback is executed. When seeing callback code, is separately logs 0,1,2 as there are different I for each callback

for (let i = 0; i < 3; i++) {

setTimeout(function () {

console.log(i); }, 1000); }

* IIFE

Here var i is shared across all iterations. So when

i=0, i is passed as argument so j=i=0. j is remembered after callback, because j is a parameter and parameter is scoped to func

same for i=1, i=2

i=3, condition fails

After 1sec, the queued callback is executed. When seeing callback code, is separately logs 0,1,2 as there are different I for each callback

for (var i = 0; i < 3; i++) {

(function (j) {

setTimeout(function () {

console.log(j); // Logs the value of j

}, 1000); }) (i);

// Pass the current value of i as argument

}

**SYNCHRONOUS PROGRAMMING:**

* Synchronous programming in JS is where JS engine executes code sequentially, one line at a time.
* Each line of code waits for the previous one to finish before proceeding to the next.

EX: console.log("Hi");

console.log("Geek");

console.log("How are you?");

* 2nd, 3rd console works after executing 1st console.

**ASYNCHRONOUS PROGRAMMING:**

* Allows multiple tasks to run independently of each other.
* In asynchronous, a task can be initiated and while waiting for it to complete, other tasks can be proceeded.

EX: console.log("Hi");

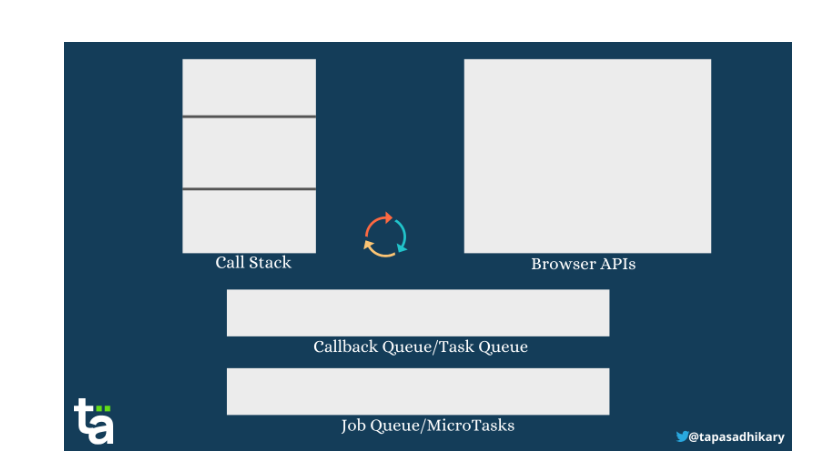
setTimeout(() => {

console.log("Geek");

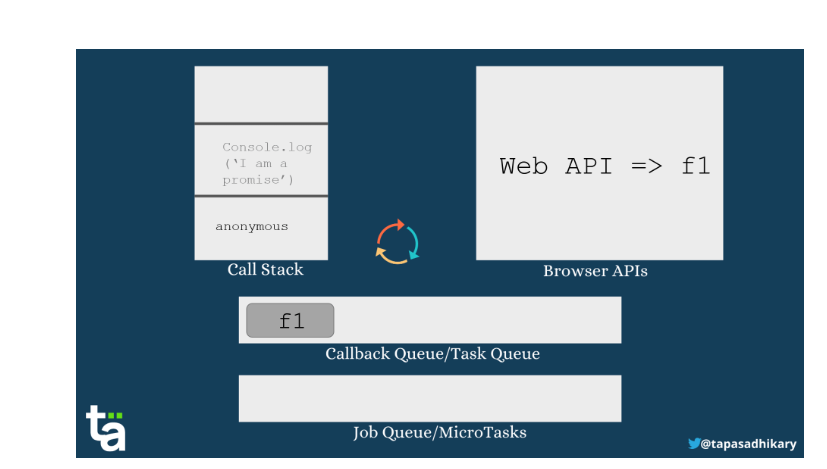
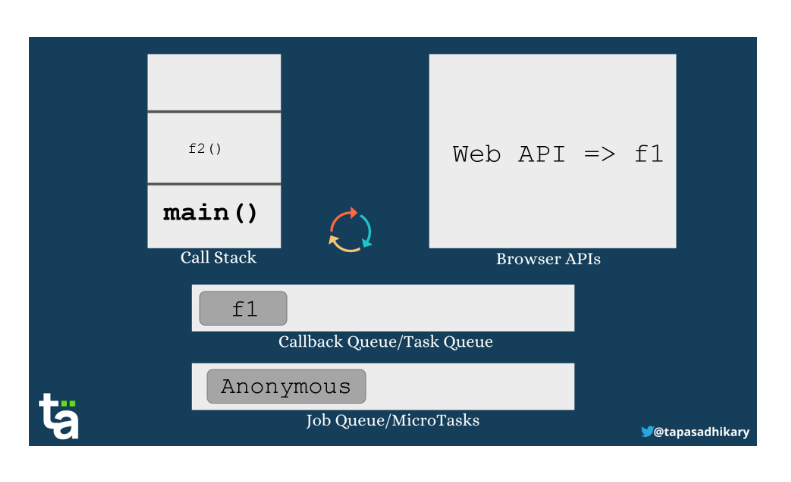
}, 2000);

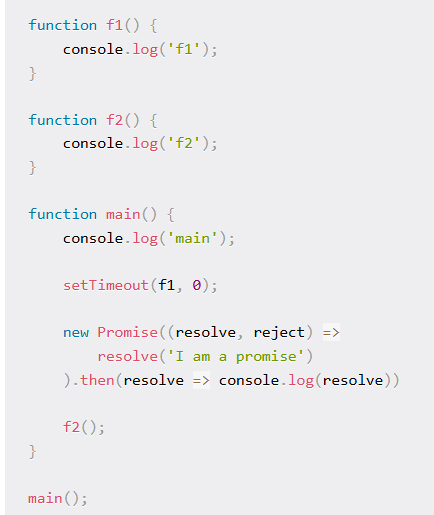
console.log("End");

* Hi End Greek is the output for the above code.
* Working –
  + When program encounters setTimeOut, it is pushed to callstack.
  + Without interrupting main code, it wiats in stack.
  + Once time is over, it pushes callback function to callback queue.
  + The event loop continuously checks the call stack. If it’s empty, it pushes the functions from the callback queue onto the stack for execution.
* Promises can also be used to perform asynchronous operations.



* Call stack 🡪 if any browser API (setTimeOut) / if any Job queue/ microtasks (Promises)
* Executes call stack 1st and push promise (Job Queue) to call stack even if setTimeout (callback queue) has 0 as delay.
* Only then setTimeOut is pushed to stack.





**OUTPUT**

main

f2

I am a promise

f1

**CALLBACK:**

* A callback is a function passed as an argument to another function. It shouldn’t have paranthesis
* A callback function can run after another function has finished.
* JavaScript functions are executed in the sequence they are called. Not in the sequence they are defined.
* EX: function myDisplayer(some) {  
    document.getElementById("demo").innerHTML = some;  
  }  
    
  function myCalculator(num1, num2, myCallback) {  
    let sum = num1 + num2;  
    myCallback(sum);  
  }  
    
  myCalculator(5, 5, myDisplayer);
* Here **myDisplayer** is callback function.

**CALLBACK HELL:**

* Callback hell is multiple nested callbacks. It is difficult to manage and complex to understand.
* To overcome this promise, async/ await is used.
* Async/ await and promises are both used for handling asynchronous operations in JS.
* Promise provide simple way to work with callbacks
* Async/ await are more readable.
* The async keyword is used to create a function that returns a Promise, while the await keyword pauses the function’s execution until the Promise is done.

**PROMISE:**

* Promise represents the eventual completion (success)/ failure of an asynchronous operation and its resulting value.