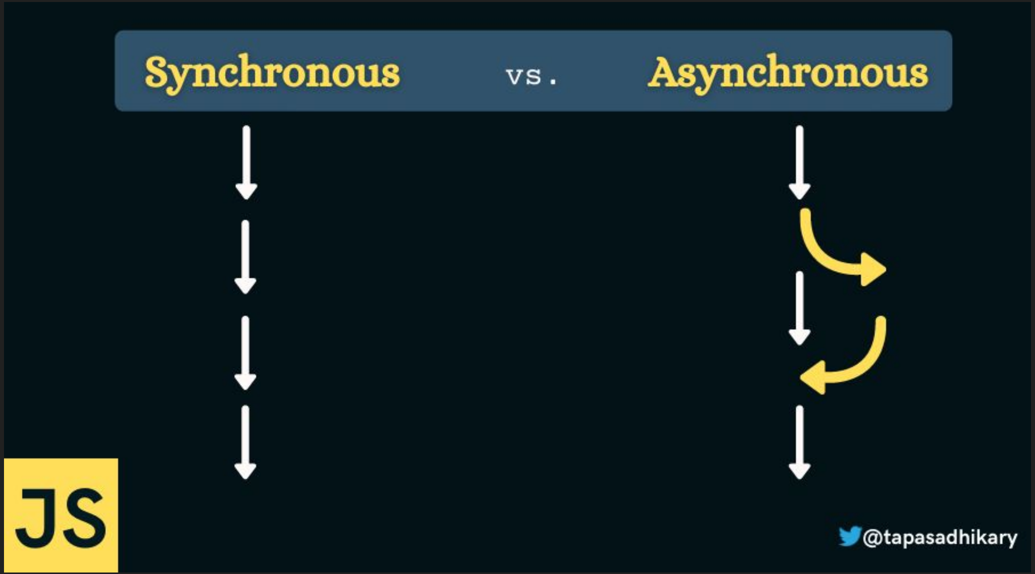
Welcome Geeks

JS Next Level

- synchronous and asynchronous JS
- call stack, event loop



Synchronous JS

- Synchronous JS - Function Execution and Call Stack

So what happens when you define a function and then invoke it?

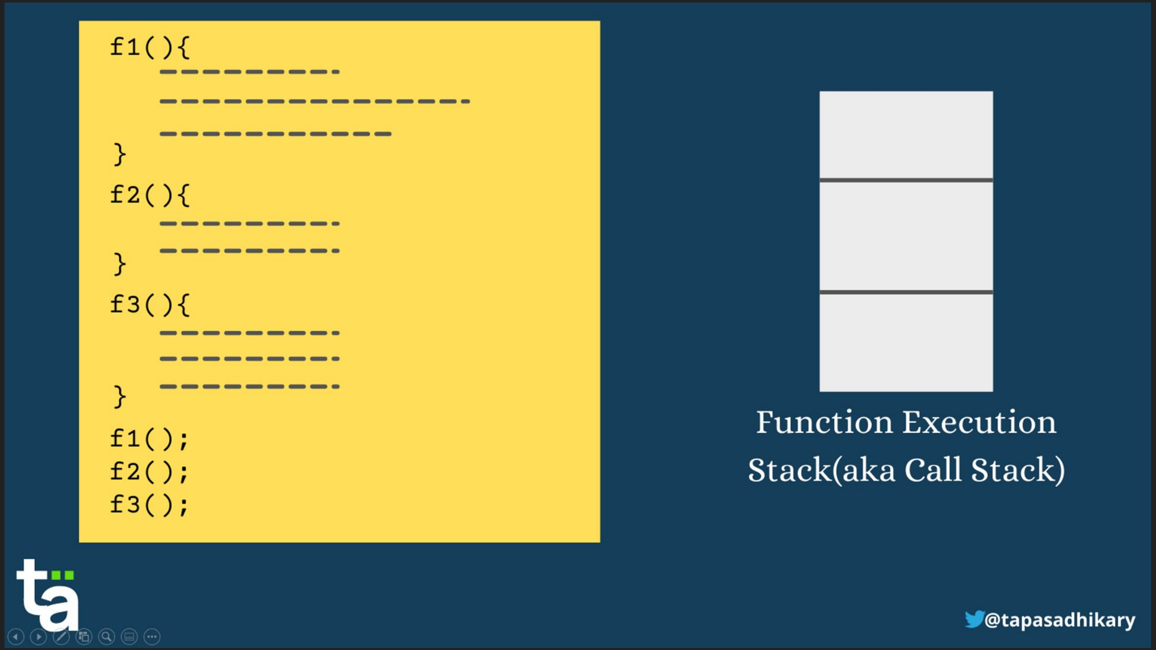
The JavaScript engine maintains a stack data structure called function execution stack. The purpose of the stack is to track the current function in execution.

Synchronous JS

- When the JavaScript engine invokes a function, it adds it to the stack, and the execution starts.
- If the currently executed function calls another function, the engine adds the second function to the stack and starts executing it.
- Once it finishes executing the second function, the engine takes it out from the stack.
- The control goes back to resume the execution of the first function from the point it left it last time.
- Once the execution of the first function is over, the engine takes it out of the stack.
- Continue the same way until there is nothing to put into the stack.

Synchronous JS

```
function f1() {
  // some code
}
function f2() {
  // some code
}
function f3() {
  // some code
}

// Invoke the functions
 one by one
f1();
f2();
f3();
```



```
f1(){
   ----------
   -----------------
   -------------
}
f2(){
   ----------
}  ----------
f3(){
   ----------
   ----------
}  ----------
f1();
f2();
f3();
```

Function Execution Stack(aka Call Stack)

Asynchronous JS

- Browser API/Web API events or functions. These include methods like setTimeout, or event handlers like click, mouse over, scroll, and many more.

- Promises. A unique JavaScript object that allows us to perform asynchronous operations.

Asynchronous JS

```
function printHello() {
  console.log(’print hello’);
}

setTimeout(printHello, 5000);
```

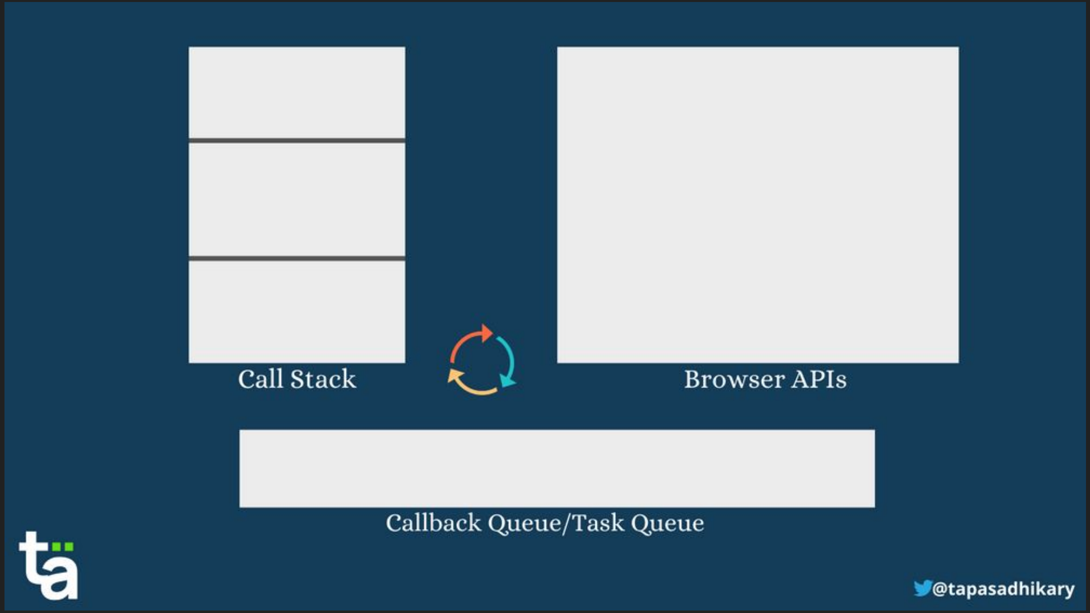
The setTimeout function executes a function after a certain amount of time has elapsed. In the code above, the text print me logs into the console after a delay of 5 seconds.

Asynchronous JS

```
function printHello() {
  console.log(’print hello’);
}

function test() {
  console.log(’test’);
}

setTimeout(printHello, 5000);
test();
```

Asynchronous JS

Callback Queue comes into play here!

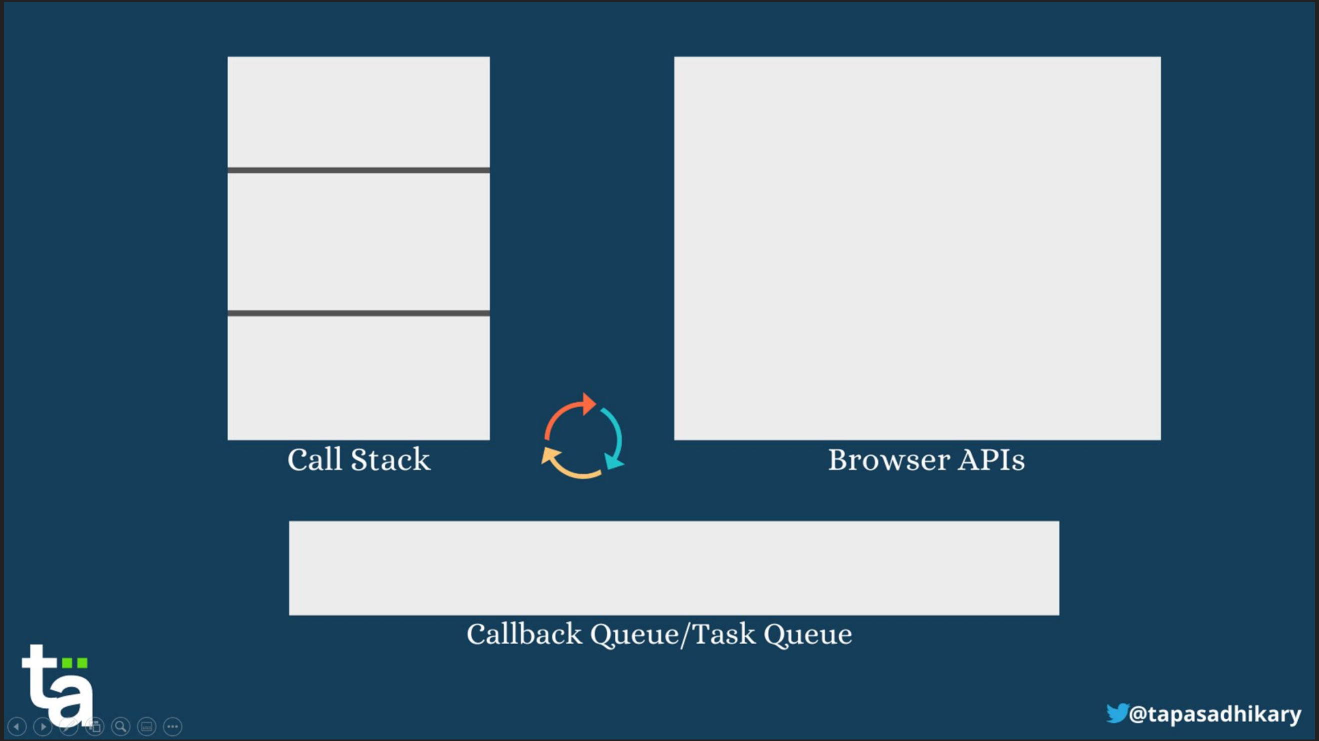


Asynchronous JS

```
function f1() {
    console.log('f1');
}

function f2() {
    console.log('f2');
}

function main() {
    console.log('main');

    setTimeout(f1, 0);

    f2();
}

main();
```

Asynchronous JS

```
function f1() {
    console.log('f1');
}

function f2() {
    console.log('f2');
}

function main() {
    console.log('main');

    setTimeout(f1, 0);

    f2();
}

main();
```

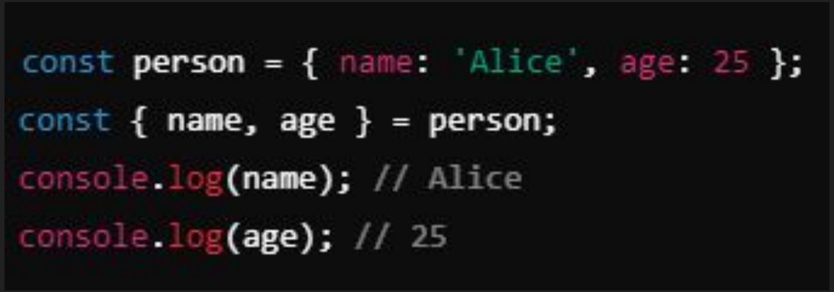


ES6

- let & const
- arrow functions
- promises

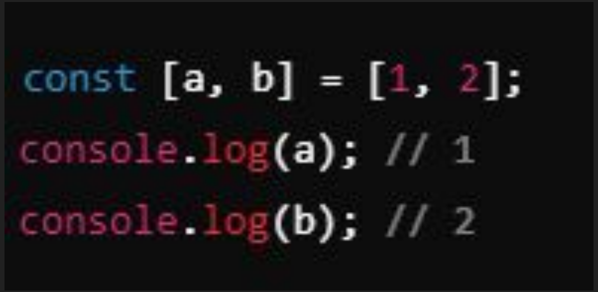
Destructuring

Object Destructuring



```
const person = { name: 'Alice', age: 25 };
const { name, age } = person;
console.log(name); // Alice
console.log(age); // 25
```

Array Destructuring



```
const [a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2
```

Arrow functions

Regular vs Arrow functions

- Syntactical difference
- No duplicate parameters
- Arguments binding

callback functions

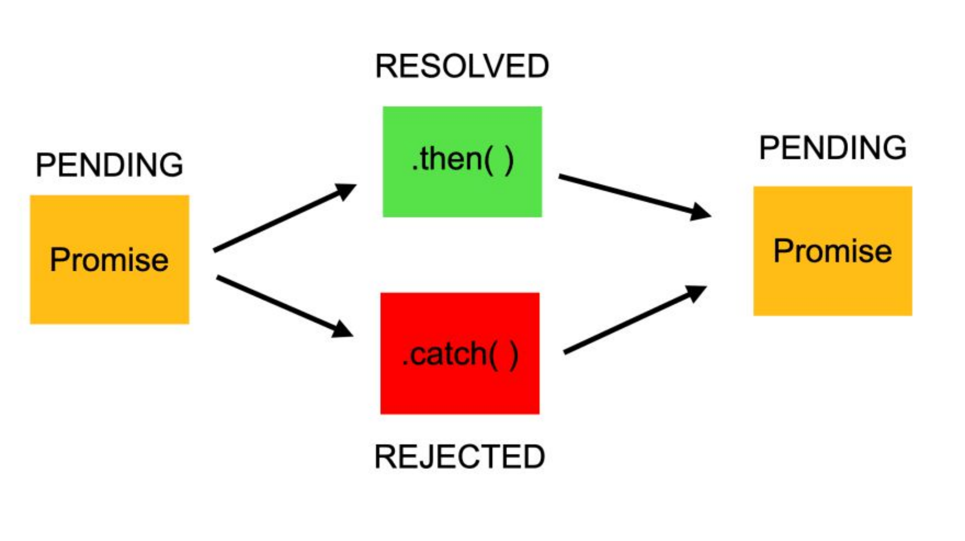
- A callback is a function passed as an argument to another function

```
const message = function() {
    console.log("This message is shown after 3 seconds");
}

setTimeout(message, 3000);
```

callback hell

promises

- Promises are the alternative to callbacks for delivering the results of asynchronous computation.



promises

```
const promise = new Promise((resolve, reject) => {
  // Async operation logic here....
  if (asyncOperationSuccess) {
    resolve(value); // async operation successful
  } else {
    reject(error);  // async operation error
  }
});
```

async await