Asynchronous JS
Event-Loop

CS445 Modern Asynchronous Programming

Maharishi University of Management

Department of Computer Science

Teaching Faculty: Assistant Professor Umur Inan

Prepared by: Associate Professor Asaad Saad

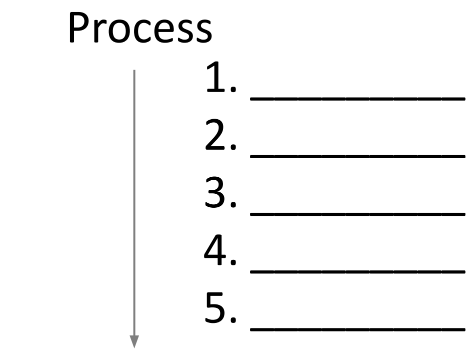
Introduction

JavaScript executes code in a single thread, which brings a risk of blocking the thread if a single line takes long time to process. This means until that line finishes, the next line of code won’t be processed.

For example, handling of AJAX request should be done on a different thread, otherwise our main thread would be blocked until the network response is received.

Synchronous vs Asynchronous

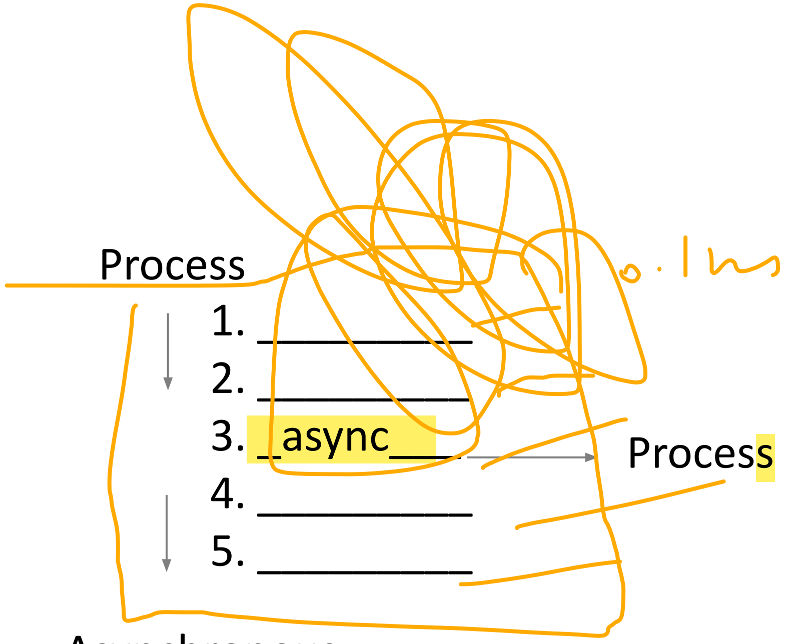
- *Asynchronous* means more than one process running simultaneously.
- *Synchronous* means one process is executing at a time.
- **JavaScript/ V8 is Synchronous**



Process

1. __________
2. __________
3. __________
4. __________
5. __________

Synchronous



Process

1. __________
2. __________
3. __async___
4. __________
5. __________

Process

0.1ms

Asynchronous

Web APIs

Web APIs are APIs that extends JavaScript functionality to perform asynchronous tasks. For example, **setTimeout** is a Web API that performs some action after a given delay.

Using Web APIs, some JavaScript jobs can be transferred to other threads.

Web APIs is not a part of the JavaScript standard. They are not included in the JavaScript engine. Instead, they are provided by the browser or server-side JavaScript frameworks like Node.js

JavaScript Timers

Both `setTimeout` or `setInterval` schedule a certain function to execute after a specified delay in milli-seconds (when possible).

```
// call a given function after delay in ms
setTimeout(fn, ms);
// call a function repeatedly every ms
setInterval(fn, ms);
```

How Timers Work?

`setTimeout(callback, delay)` function takes a **`callback`** and stores it temporarily. It waits for **`delay`** given in milliseconds and then pushes the **`callback`** function in V8 stack **once the stack it is empty**. That's when the callback function gets executed.

The stack will become empty when all synchronous function calls are executed.

time slin

Concurrency

Example

```
setTimeout(hideBanner, 5000);

// called when the timer goes off
function hideBanner() {
   document.getElementById("mwp").style.display = "none";
}
```

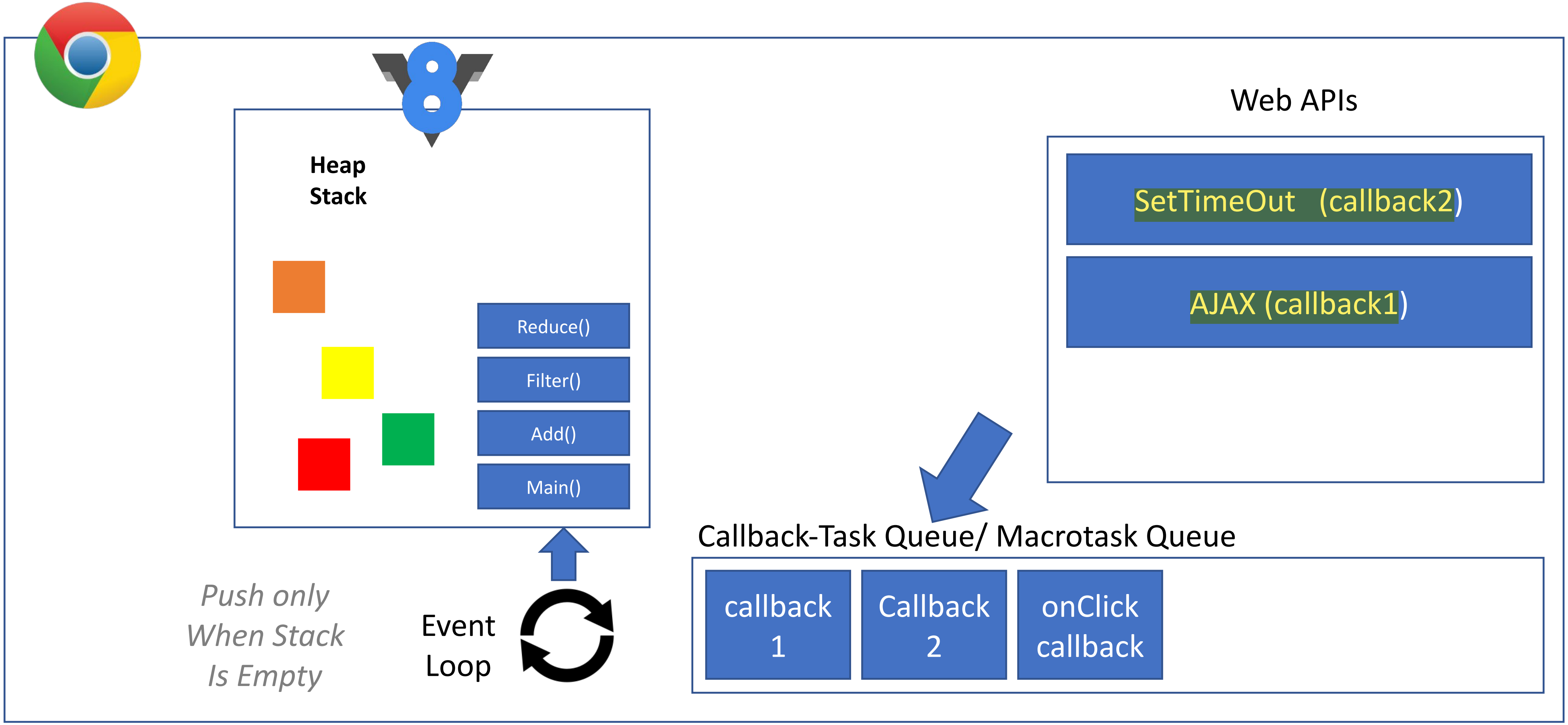
Example

```
let count = 0;
const timerId = setInterval(increment, 1000);

// called every 1000 ms
function increment() {
   document.getElementById("mwp").innerHTML = ++count;
   if(count === 10) clearInterval(timerId);
}
```

Common Timer Errors

```
function multiply(a, b) {
   return a * b;
}

setTimeout(~~multiply(num1 , num2)~~, 5000);
setTimeout(multiply, 5000, num1, num2);
```

Chrome – The Event Loop



If you block the stack, browser can't run the render queue

The Event-Loop

The event loop is endlessly running single-threaded loop that runs on the main JavaScript thread and listens for the different events. Its job is to accept callback functions and execute them on the main thread. Since event loop runs on the main thread, if the main thread is busy, event loop is basically dead for that time.

The macrotask queue is a queue of the callback function waiting to be executed. The event loop pushes oldest queued callback functions (FIFO) from macrotask queue on to the main call stack one at the time where they are executed synchronously. Event loop only pushes a callback function to the stack when the stack is empty or when the main thread is not busy.

Blocking the Event-Loop/ Rendering UI

```
<input placeholder="Open console and enter your text" />

<script>
    document.querySelector('input').addEventListener('keyup',
    e => {
        let n = 0
        for (let i = 0; i < 9e7; i++) { n = n + e.which }
        console.log(e.key)
    })
</script>
```

Callbacks and Events Queue

```
// In what order the results will be printed and why?

console.log(1);
const a = setTimeout(function(){ console.log(2); }, 1000);
const b = setTimeout(function(){ console.log(3); }, 0);
console.log(4);
```

Be Careful!

Accepting a callback function does not mean the function will be asynchronous.

```
console.log(`Start`);
[1, 2, 3].forEach(i => console.log(i));
console.log(`Finish`);
```

```
Start
1
2
3
Finish
```

Asynchronous Example

```
console.log(`Start`);
[1, 2, 3].map(i => setTimeout(_ => console.log(i)), 0);
console.log(`Finish`);
```

```
Start
Finish
1
2
3
```

The Boomerang Effect (Callback Hell)

As you may have noticed, asynchronous programming relies on callback functions that are usually passed as arguments.

This can turn your code into "**callback spaghetti**", making it visually hard to track which context you are in. This style also makes debugging your application difficult, reducing even more the maintainability of your code.

Callback Hell Example

```
function fn(callback) {
    setTimeout(() => {
            console.log('result of fn()');
            callback();
       }, 1000 ); // 1 second delay
}

fn(()=> console.log('fn() is done!'));
```

Promise to the Rescue!

Promises make our job a little easier when it comes to writing complicated asynchronous programs.

Different states of a promise

A Promise has one of three states:

- **Pending**
- **Fulfilled**
- **Rejected**

Create Promise Object

A **`Promise`** represents a value which may be available now, or in the future, or never.

```
const promiseInstance = new Promise(function(resolve, reject){
     resolve();
     reject();
});
```

Consume Promise Object

```
promiseInstance.then(fn) // when resolve()
               .catch(fn) // when reject()
               .finally(fn) // always
```

Both catch and finally handlers are optional.

How Promises can make our code easy to read

As the `Promise.prototype.then()` and `Promise.prototype.catch()` methods return promises, they can be chained.

```
const pizzaPromise = makeMePizza('Pepperoni'); // returns Promise

pizzaPromise.then(eat).then(drink).catch(cry).finally(sleep)
```

Creating a promise

```
const makeMePizza = function(){
    return new Promise(function(resolve,reject){
    if(everythingWorks){
        resolve("Here is your pizza!"); // then() will be called
    } else {
        reject("Sorry no more cheese!"); // catch() will be called
    }
    })
}
makeMePizza()
    .then(data => console.log(data))
    .catch(err => console.error(err));
console.log('Finish my homework');
```

How Do Promises work?

The biggest misconception about Promises in JavaScript is that they are asynchronous, but not everything of Promises is asynchronous.

Only the parts of **`resolve`** and **`reject`** are going to be asynchronous.

```
const promise = new Promise((resolve, reject) => {
   console.log(`Promise starts`)
   resolve(`Promise result`)
   console.log(`Promise ends`)
})

console.log(`Code starts`)
promise.then(console.log)
console.log(`Code ends`)
```

Example

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => { resolve('Promise results')}, 1000); // resolve after 1 second
});

console.log('Code starts');

promise.then(console.log)

console.log('I love JS');
```

What happens when we change the timer to 0

Queue Example

```
setTimeout(() => console.log('setTimeout results'), 0);   4

const promise = new Promise((resolve) => resolve(`Promise results`));

console.log('Code starts');   1

promise.then(console.log);   3

console.log('I love JS');   2
```

Promises and the Event-Loop

`then` and **`catch`** as well as **`finally`** methods of a promise register the callback functions passed to them and these callbacks are provided to the event loop when the promise is resolved or rejected. These callbacks are added to the **`microtask`** queue which has higher priority than **`macrotask`** queue.