CS472 WAP Lecture 6: Asynchronous Programming

Maharishi International University - Fairfield, Iowa



All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi International University.

Process and Thread

When you execute a computer program, an instance of this program lives in the memory, we call it: **process**.

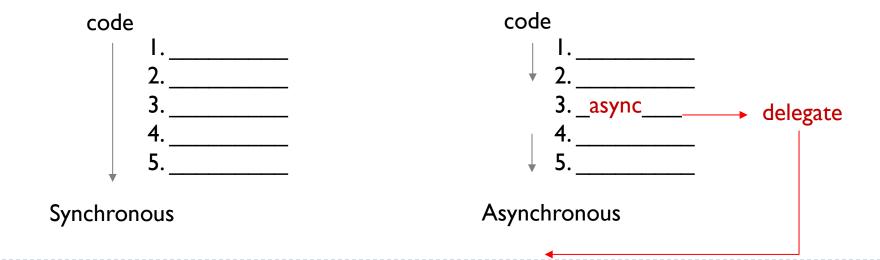
Code runs as a **thread**, a process may have multiple threads. A thread is instructions to be executed by the CPU.



Synchronous vs Asynchronous

Asynchronous means more than one task running simultaneously. Synchronous means one task is executed at a time.

JavaScript/ V8 is Synchronous



Timers

- >setTimeout allows to run a function once after the interval of time.
- runs.



How Timers Work?

The Async API handles the setTimeout(callback, delay) function, which accepts a callback function and delay.

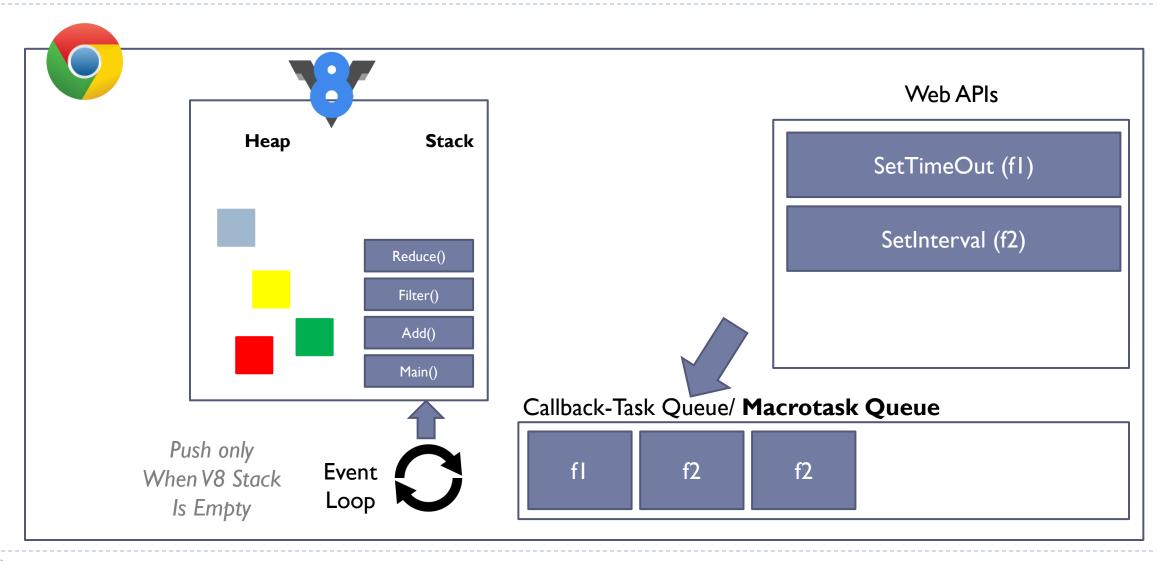
It waits for a given delay in milliseconds and then pushes the callback function to a queue (Macrotask).

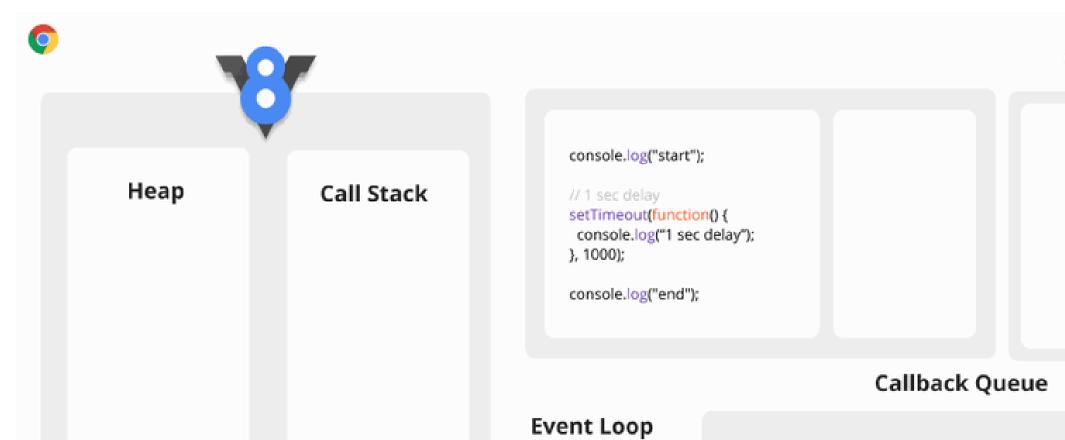
When the V8 stack is empty, the event-loop picks the function and pushes it to the V8 stack to be executed.

The V8 stack runs all code synchronously.

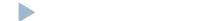


Chrome – The Event Loop





Web APIs



The Event-Loop

The event loop is endlessly running a single-threaded loop and waits for different events. When the JavaScript Engine is ready to accept a new task, the event loop takes callback functions from the macrotask queue and assigns them to the JS Engine to execute. Since the event loop runs on the main thread, if the V8 main thread is busy for a long time, the event loop is blocked for that time.

The macrotask queue is a queue of the callback functions waiting to be executed.

The event loop only pushes a callback function to the stack when the stack is empty or when the main thread is not busy.





```
let timerId = setTimeout(func, [delay], [arg1], [arg2], ...)
> Func: Function or a string of code to execute.
> Delay: delay before run, in milliseconds (1000 ms = 1 second), by default 0.
>arg1, arg2...: Arguments for the function
function sayHi() {
    alert('Hello');
setTimeout(sayHi, 1000);
> With arguments:
function sayHi(phrase, who) {
    alert(phrase + ', ' + who);
setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

Pass a function, but don't run it

Novice developers sometimes make a mistake by adding brackets ()
// wrong!
setTimeout(sayHi(), 1000);

- >doesn't work,
 - > setTimeout expects a reference to a function.
 - ▶ here sayHi() runs the function,
 - > result of its execution is passed to setTimeout.
 - > result of sayHi() is undefined (the function returns nothing), so nothing is scheduled
- function call versus function binding
 - > sayHi() versus sayHi
 - > execute the function versus reference to the function
 - fundamental concept!!



Canceling with clearTimeout



A call to setTimeout returns a "timer identifier" timerId that we can use to cancel the execution.

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

schedule the function and then cancel it

```
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier
clearTimeout(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
```



setInterval



The setInterval method has the same syntax as setTimeout:

```
let timerId = setInterval(func, [delay], [arg1], [arg2], ...)
```

Repeatedly calls the function after the given interval of time.

```
To stop further calls, we should call clearInterval (timerId).
// repeat with the interval of 2 seconds
let timerId = setInterval(() => alert('tick'), 2000);
// after 5 seconds stop
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Canceling with clearInterval

A call to setInterval returns a "timer identifier" timerId that we can use to cancel the execution.

schedule the function and then cancel it

```
let timerId = setInterval(() => alert("never happens"),1000);
alert(timerId); // timer identifier
clearInterval(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
```



Zero delay setTimeout



- >There's a special use case: setTimeout(func, 0), or just setTimeout(func).
- > schedules the execution of func as soon as possible.
 - > after the current code is complete.

```
setTimeout(() => alert("Hello"), 0);
alert("World");
```

Main Point: Timout callbacks

The asynchronous global methods setTimeout and setInterval take a function reference as an argument and then callback the function at a specified time. Science of Consciousness: Accepting an assignment and carrying it out at a designated time is a fundamental capability required for intelligent behavior. A clear awareness and mind promotes good memory and the ability to successfully execute tasks.



Promise, Async & Await

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.

```
async1(function(){
    async2(function(){
        async3(function(){
            async4(function(){
            });
        });
    });
});
```

Asynchronous Code

To write a nice asynchronous code away from the callback hell we can use one of these code structures:

- Promises (ES6)
- Function Generators (ES6)
- Async & Await (ES7, Node 8)
- util.promisify (Node 8)
- Observables (ES7)

Introduction

- Java Script 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.

The Promise Object

The **Promise** object is used for asynchronous computations. A **Promise** represents a value which may be available now, or in the future, or never.

```
new Promise( function(resolve, reject) { ... } );
```

A Promise has one of these states:

- **pending**: initial state, not fulfilled or rejected.
- fulfilled: meaning that the operation completed successfully.
- rejected: meaning that the operation failed.

A pending promise can either be fulfilled with a value, or rejected with a reason (error).

state: "fulfilled"
result: value

Ject (error)

new Promise(executor)

state: "pending" result: undefined

state: "rejected"

result: error

Create a Promise Object

```
var giveMePizza = function(){
       return new Promise(function(resolve, reject){
               if(false){
                      resolve("This is true"); // then() will be called
               } else {
                      reject("This is false"); // catch() will be called
       })
giveMePizza()
       .then(data => console.log(data))
       .catch(err => console.error(err));
console.log('I will run immediately after calling giveMePizza() and before any
result arrives');
```

The callback from the Promise constructor gives us two parameters, resolve and reject functions, that will affect the state of the Promise object. If everything works, call resolve, otherwise call reject. Note that you can pass in values to resolve and reject which will be further passed on to the respective handlers, then and catch.

Create a Promise Object (cont.)

```
let promise = new Promise(function(resolve, reject) {
    // the function is executed automatically when the promise is constructed
    // after 1 second signal that the job is done with the result "done".
    setTimeout(() => resolve("done"), 1000);
});
                       new Promise(executor)
                                               resolve("done"
                         state: "pending"
                                                                    state: "fulfilled"
                        result: undefined
                                                                    result: "done"
let promise = new Promise(function (resolve, reject) {
// after 1 second signal that the job is finished with an error
    setTimeout(() => reject(new Error("Whoops!")), 1000);
});
                        new Promise(executor)
                                                reject(error
                         state: "pending"
                                                                   state: "rejected"
                         result: undefined
                                                                   result: error
```

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) => {
    reject("error");
    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

Consumers: then, catch, finally

• A Promise object serves as a link between the executor and the consuming functions, which will receive the result or error. Consuming functions can be registered (subscribed) using methods .then, .catch and .finally.

```
let promise = new Promise(function (resolve, reject) {
   const random = Math.random();
   console.log(' random: ', random);
   if (random > 0.5)
      setTimeout(() => resolve("done"), 1000);
   else
      setTimeout(() => reject(new Error("Woops!")), 1000)
});

promise.then(result => console.log(result))
   .catch(error => console.log(error))
   .finally(() => console.log('Promise ready!'));
```

Synchronous or Asynchronous?

The promise executor function is the function you pass to the new Promise(). It is executed synchronously.

The callbacks you attach with then, catch, and finally are always called asynchronously, whether the promise is already settled or not.



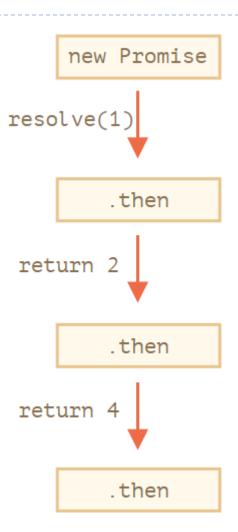
Microtask Queue

All then and catch and finally registered callback functions are pushed into the microtask queue which has a higher priority than the macrotask queue.



Promises chaining

```
new Promise(function(resolve, reject) {
    setTimeout(() => resolve(1), 1000); // (*)
}).then(function(result) { // (**)
    alert(result); // 1
    return result * 2;
}).then(function(result) { // (***)
    alert(result); // 2
    return result * 2;
}).then(function(result) {
    alert(result); // 4
    return result * 2;
});
```



Async /await

- It's a special syntax to work with promises in a more comfortable fashion
- The async keyword: when you put async keyword in front of a function declaration, it turns the function into an async function.
- The await keyword: await only works inside async functions. await can be put in front of any async promise-based function to pause your code on that line until the promise fulfills, then return the resulting value.

Async functions

- async can be placed before a function. An async function always returns a promise:
 - When no return statement defined, or return without a value. It turns a resolving a promise equivalent to return Promise. Resolve()
 - When a return statement is defined with a value, it will return a resolving promise with the given return value, equivalent to return Promise. Resolve (value)
 - When an error is thrown, a rejected promised will be returned with the thrown error, equivalent to return Promise.Reject(error)

```
console.log('start');
async function f() {
    return 1;
}
f().then(console.log);
console.log('end');
```

Await

- The keyword await makes JavaScript wait until that promise settles and returns its result.
- await literally suspends the function execution until the promise settles, and then
 resumes it with the promise result. That doesn't cost any CPU resources, because
 the Java Script engine can do other jobs in the meantime: execute other scripts,
 handle events, etc.
- It's just a more elegant syntax of getting the promise result than promise. then.

```
console.log('start');
async function foo() {
    return ' done ! ';
}
async function bar() {
    console.log(' insidebar - start');
    let result = await foo();
    console.log(result);
    console.log('insidebar - end');
}
bar();
console.log('end');
```

Await (cont.)

```
1. Can't use await in regular functions. If we
   try to use await in a non-async function,
   there would be a syntax error:
async function foo() {
    return 'done!';
function bar() {
    let result = await foo(); // Syntax error
    console.log(result);
bar();
```

```
2. await won't work in the top-level code
// syntax error in top-level code
async function baz() {
    return 'baz...';
let result = await baz(); //Syntax Error
console.log(result);
```

Error Handling in Async functions

- 1. Use await inside async function
- 2. Chain async function call with a .catch() call.

```
async function thisThrows() {
    throw new Error("Thrown from thisThrows()");
async function run() {
    try {
        await thisThrows();
    } catch (e) {
        console.log('Caught the error....');
        console.error(e);
    } finally {
        console.log('We do cleanup here');
run();
```

```
async function thisThrows() {
    throw new Error("Thrown from thisThrows()");
}
thisThrows()
    .catch(console.error)
    .finally(() => console.log('We do cleanup here'));
```

Example - Promise

```
let Studied = true;
let willPassTheExam = function(){
       return new Promise(function(resolve, reject) {
               if (true) resolve('Pass');
              else reject(new Error('Fail'));
       })
};
let askMe = function () {
       willPassTheExam()
               .then(function(results){ console.log(results); })
               .catch(function (error) { console.log(error); });
};
askMe();
console.log('Finish')
```

Example - Async & Await

```
async function askMe() {
       try {
               console.log('Before/ taking the exam');
               let results = await willPassTheExam();
3
               console.log('After taking the exam');
               console.log(results);
       } catch (error) {
              console.log(error);
askMe();
console.log('Finish')
```

Fetch API

- The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a **global** fetch() method that provides an easy, logical way to fetch resources **asynchronously** across the network.
- This kind of functionality was previously achieved using XMLHttpRequest.
- Syntax:

```
let promise = fetch(url, [options]);
```

- url the URL to access.
- options optional parameters: method, headers etc.
- Without options, this is a simple GET request, downloading the contents of the url.

Using fetch() - GET

```
let promise = fetch('http://localhost:3000/products')
```

The browser starts the request right away and returns a promise that the calling code should use to get the result.

- Getting a response is usually a two-stage process.
 - First, the promise, returned by fetch, resolves with an object of the built-in Response class as soon as the server responds with headers.

```
promise.then(response => console.log(response.ok, response.status));
```

- Second, to get the response body, we need to use an additional method call.
 - response.text() read the response and return as text,
 - response.json() parse the response as JSON,
 - ..

```
promise.then(response => response.json())
    .then(result => console.log(result));
```

Using fetch() – POST with options

```
fetch('http://localhost:3000/products', {
        method: 'POST',
        body: JSON.stringify({
            title: 'cs472',
            description: 'This is CS472 Lesson 14 Demo',
            price: 100,
        }),
        headers: {
            'Content-type': 'application/json; charset=UTF-8',
        },
    })
    .then((response) => response.json())
    .then((json) => console.log(json));
```

Using Async & Await

```
async function fetchProductById(id) {
    let response = await fetch("http://localhost:3000/products/"
                                                               + id);
    if (response.ok) {
       let json = await response.json();
       console.log(json);
   } else {
       console.log("HTTP-Error: " + response.status);
fetchProductById(1);
// Remember to use try/catch in case of network problem
```