JavaScript

Stephen P Levitt

School of Electrical and Information Engineering
University of the Witwatersrand

2021

Outline

1. Functions Recap

2. Closures
   - Practical Examples of Closures

3. Asynchronous Programming
   - Callbacks
   - Problems with Callbacks
   - Promises

Functions as Values

Functions are first-class citizens (can be easily passed as arguments to other functions; can be returned from functions; can be assigned to variables or stored in data structures)

```
// illustrating a function *declaration*
function aFunction() {
    console.log("in a function");
};

aFunction(); // invoking the function
```

Function Declaration Hoisting

```
foo()

function foo () {
  let a = 2
  console.log(a)
}
```

Function Expressions

```
// illustrating a function expression
const f = function aFunction () {
  console.log('in a function')
}

f() // invoking the function
```

Anonymous Function Expressions

```
// function expressions can be anonymous
const f2 = function () {
  console.log('in a function')
}

f2() // invoking the function
```

Immediately Invokable Function Expressions - IIFE

```
// function is only ever called once
(function anotherFunction () {
  console.log('in a function')
})()
// // function expression is invoked immediately by trailing ()

anotherFunction() // error, aFunction not defined
```

Read the code and answer the following questions

```javascript
const sayHello = function (name) {
  let text = 'Hello ' + name
  let say = function print () { console.log(text) }
  return say
}

const greet = sayHello('Thabo')
greet()

// Output: Hello Thabo
```

- Identify the local variables for `sayHello`
- When do these variables go out of scope?
- What is returned from `sayHello`? Use the correct term.
- At what point is the code in the `print` function executed?
- Explain the output.

> “*Closure is when a function can remember and access its lexical scope even when it’s invoked outside its lexical scope.*”
>
> — Kyle Simpson in *You Don’t Know JS*

> “*A closure is the combination of a function and the lexical environment within which that function was declared.*”

All variables in the outer function form part of the closure

```
function sayAlice () {
  const sayAlert = function greeting () { console.log(alice) }
  // Local variable is hoisted and ends up within closure
  const alice = 'Hello Alice'
  return sayAlert
}

sayAlice()() // immediately invoke the returned function expression

// Output: Hello Alice
```

Modifying variables within the closure

```
function say5 () {
  let num = 5
  const say = function () { console.log(num) }
  num++
  return say
}

say5()() // immediately invoke the returned function expression

// What is the output?
```

With each new call of the outer function a new closure is created

```
const namer = function (name) {
  return function (obj) {
    obj.name = name
    console.log(obj)
  }
}

let anObj = { groupNum: 12 }

const nameFrancis = namer('Francis')
const nameRyan = namer('Ryan')

nameFrancis(anObj)   // name set to Francis
nameRyan(anObj)      // name changed to Ryan

// What is the output?
```

With each new call of the outer function a new closure is created

```
{ groupNum: 12, name: 'Francis' }
{ groupNum: 12, name: 'Ryan' }
```

Closures Share Variables

```
let gPrintNumber, gIncreaseNumber, gSetNumber // globals

function setupSomeGlobals () {
  let num = 5
  // Store references to functions through global variables
  gPrintNumber = function () { console.log(num) }
  gIncreaseNumber = function () { num++ }
  gSetNumber = function (x) { num = x }
}

setupSomeGlobals()

gPrintNumber()
gIncreaseNumber()
gPrintNumber()
gSetNumber(44)
gPrintNumber()
```

Closures Share Variables

```
// Output:
// 5
// 6
// 44
```

Closures — Recap

- Inner functions have closure over their lexical scope
- All variables in the outer function form part of the closure
- The scope closed over has the state resulting from the completion of the outer function
- With each new call of the outer function a new closure is created
- All inner functions share access to the same variables

Give the output produced by the following code

```
function itemList () {
  let items = []
  let i = 0
  while (i < 10) {
    let item = function () {
      console.log(i) // should show its number
    }
    items.push(item)
    i++
  }

  return items
}

let list = itemList()
list[0]()
list[5]()
```

How are closures used in practice?

- For emulating private data leading to the module pattern
- For use with browser callbacks

Exercise

Give the output of this program, and explain it

```
setTimeout(() => console.log('A'), 0) // timeout of zero seconds
console.log('B')
```

Exercise

Give the output of this program, and explain it

```
setTimeout(() => console.log('A'), 0) // timeout of zero seconds
console.log('B')
```

Does this make it more obvious?

```
setTimeout(() => console.log('A'), 5000) // timeout of five seconds
console.log('B')
```

The Call Stack

Example

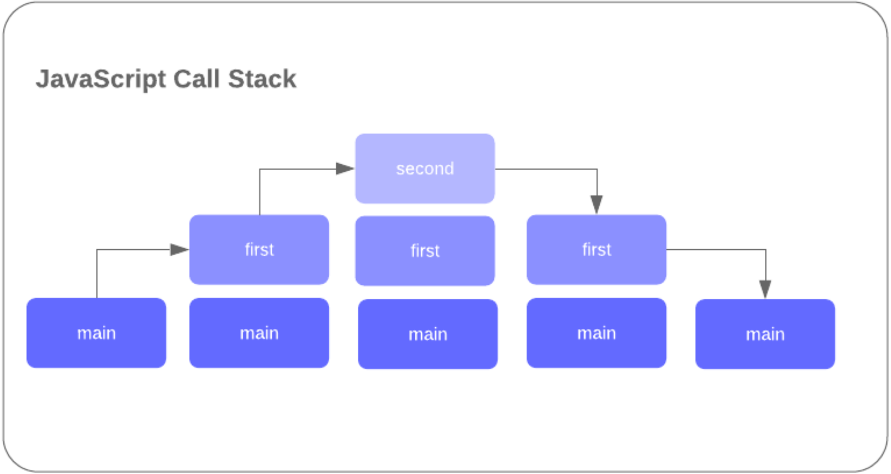
```
// Call stack example

function first () {
  console.log('in first')
  second()
}

function second () {
  console.log('in second')
}

first()
```

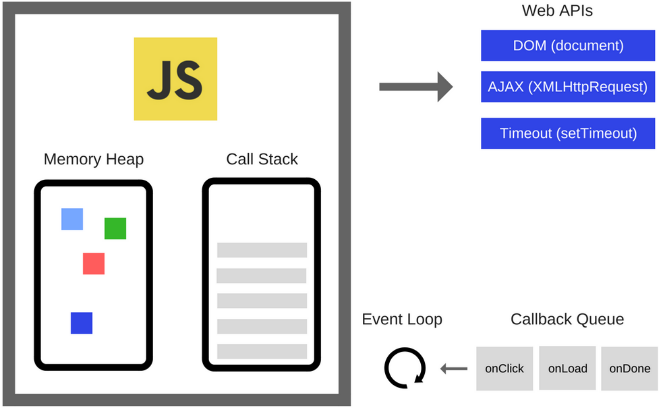
Visualizing the Call Stack



Source: JavaScript Call Stack

https://www.carnaghan.com/javascript-call-stack/

Event Loop Overview



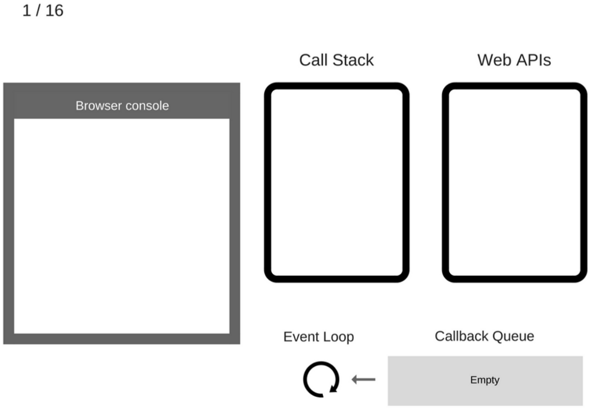
Source: How JavaScript Works

Consider the following

```
console.log('Hi')
setTimeout(function cb1 () {
  console.log('cb1')
}, 5000)
console.log('Bye')
```

Understanding the Event Loop

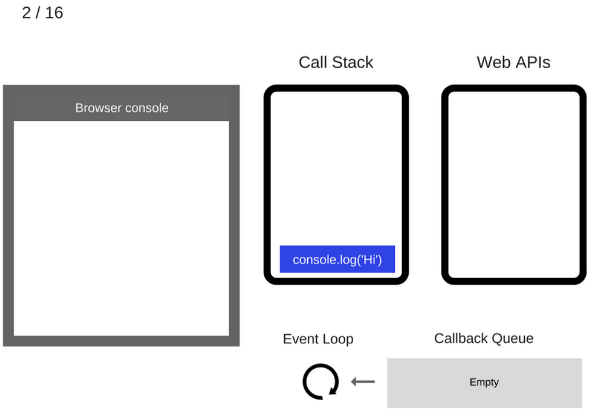
Initial state



Source: How JavaScript Works

Understanding the Event Loop

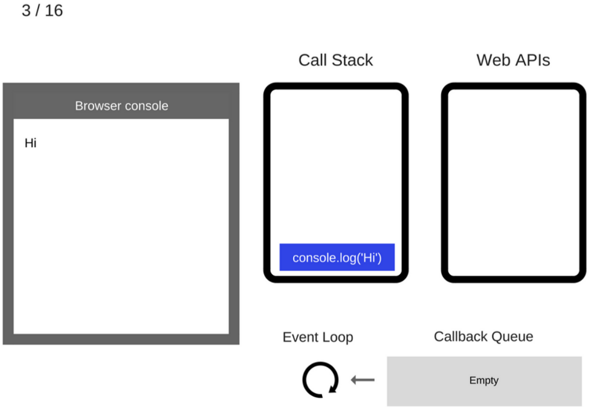
```
console.log('Hi') // added to call stack
```



Source: How JavaScript Works

Understanding the Event Loop

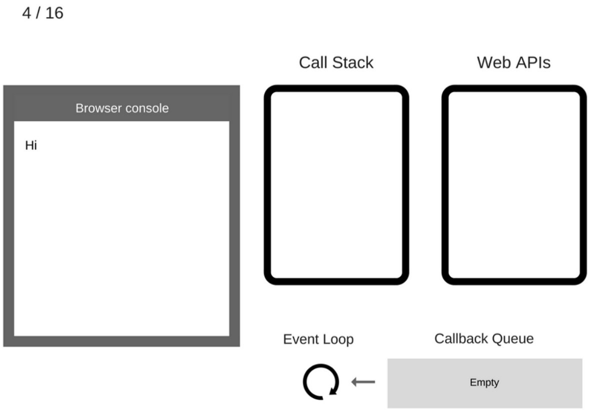
```
console.log('Hi') // executed
```



Source: How JavaScript Works

Understanding the Event Loop

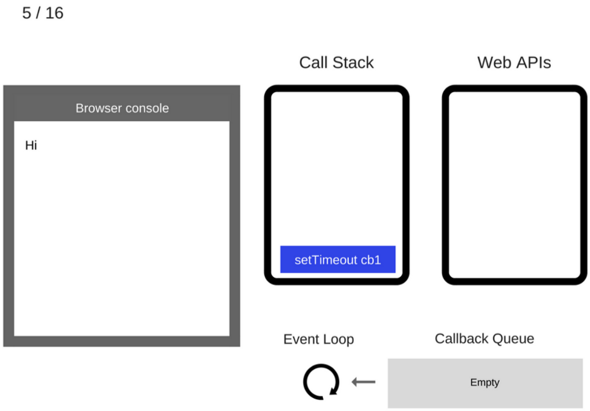
```
console.log('Hi') // removed from call stack
```



Source: How JavaScript Works

Understanding the Event Loop

```
setTimeout(function cb1 () {...}) // added to call stack
```



5 / 16

Call Stack

Web APIs

Browser console

Hi

setTimeout cb1

Event Loop

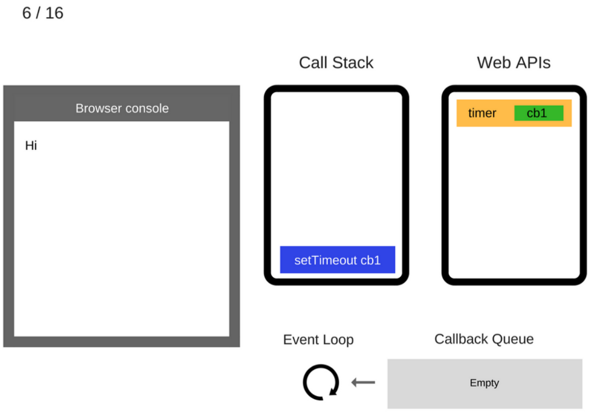
Callback Queue

Empty

Source: How JavaScript Works

Understanding the Event Loop

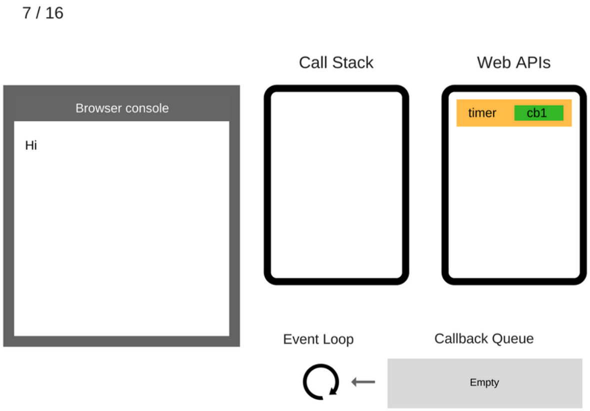
```
setTimeout(function cb1 () {...}) // executed
```



Source: How JavaScript Works

Understanding the Event Loop

```
setTimeout(function cb1 () {...}) // removed from call stack
```

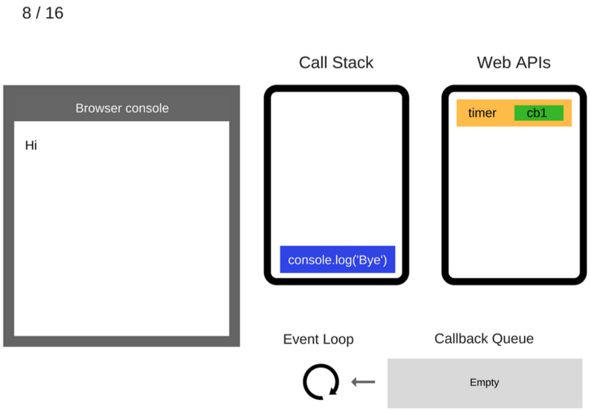


```
Hi
```

Source: How JavaScript Works

Understanding the Event Loop

```
console.log('Bye') // added to call stack
```

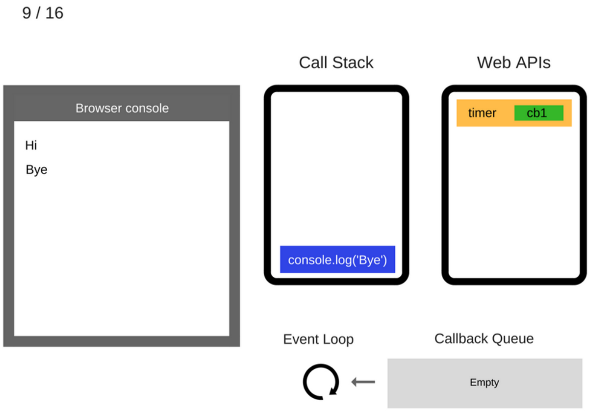


```
Hi
```

Source: How JavaScript Works

Understanding the Event Loop

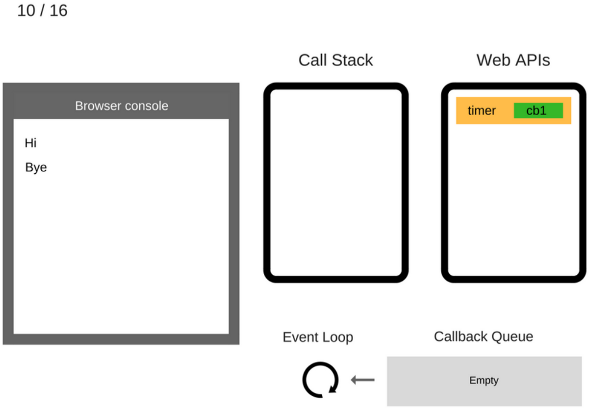
```
console.log('Bye') // executed
```



Source: How JavaScript Works

Understanding the Event Loop

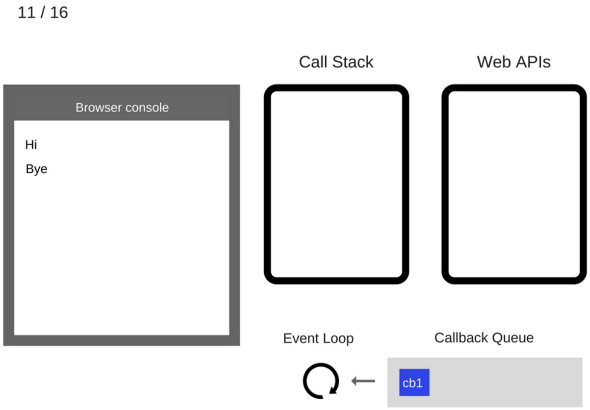
```
console.log('Bye') // removed from call stack
```



Source: How JavaScript Works

Understanding the Event Loop

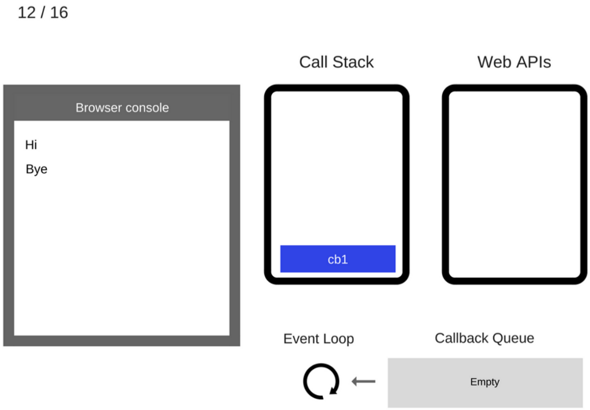
After 5 seconds, `cb1` pushed onto the Callback/Task Queue



Source: How JavaScript Works

Understanding the Event Loop

The Event Loop takes `cb1` from the Queue and pushes it to the call stack

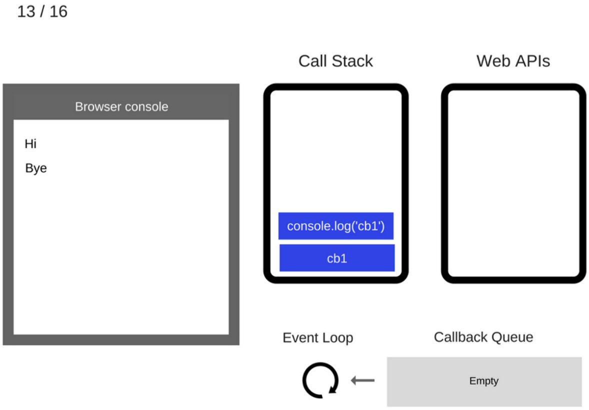


```
Hi
Bye
```

Source: How JavaScript Works

Understanding the Event Loop

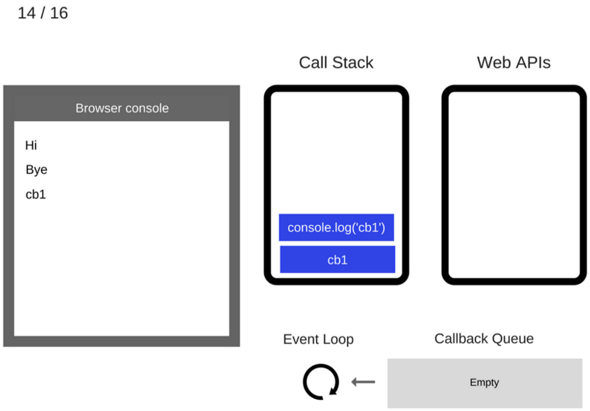
`cb1` executed and adds `console.log('cb1')` to the call stack



Source: How JavaScript Works

Understanding the Event Loop

```
console.log('cb1') // executed
```

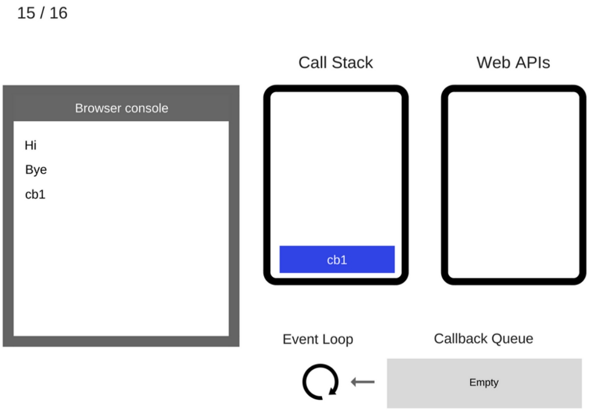


```
Hi
Bye
cb1
```

Source: How JavaScript Works

Understanding the Event Loop

```
console.log('cb1') // removed from call stack
```

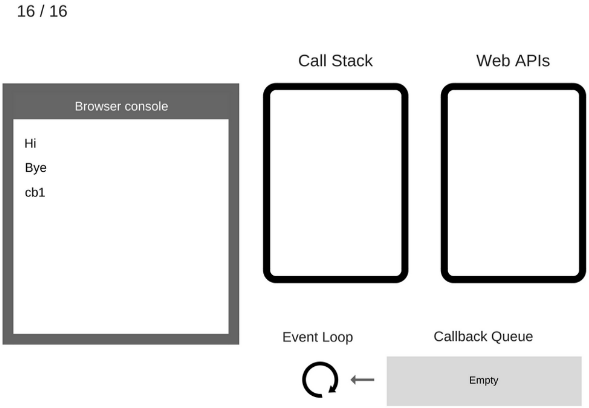


```
Hi
Bye
cb1
```

Source: How JavaScript Works

Understanding the Event Loop

`cb1` removed from call stack



Source: How JavaScript Works

Exercise

Give the output of this program, and explain it

```
setTimeout(() => console.log('A'), 0) // timeout of zero seconds
console.log('B')
```

B will always be printed before A

Callback Hell

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

Why Does This Happen?

asyncOperation2 can only take place after asyncOperation1 has completed.
asyncOperation3 can only take place after asyncOperation2 has completed.

```
asyncOperation1(function callback1() {
    asyncOperation2(function callback2() {
        asyncOperation3(function callback3() {
        })
    })
})
```

Execution Order

```
doA(() => {
    doB()
    doC(() => {
        doD()
    })
    doE()
})
doF()
```

```
first(() => {
    third()
    fourth(() => {
        sixth()
    })
    fifth()
})
second()
```

Inversion of Control Issues

```
doA()
ajax( "..", function doC(...){ // ajax(..) is some arbitrary Ajax function
    // doC is invoked by a third party service - not under our control
    // ...
} );
doB()
```

There is an implicit contract between our code and that of the third-party service

Inversion of Control Issue Example

- Building an ecommerce checkout system
- A third-party function needs to be called so that sales can be tracked

```
analytics.trackPurchase( purchaseData, function(){
    chargeCreditCard()
    displayThankyouPage()
} )
```

Preventing Multiple Calls

```
let tracked = false

analytics.trackPurchase( purchaseData, function() {
    if (!tracked) {
        tracked = true
        chargeCreditCard()
        displayThankyouPage()
    }
} )
```

Trusting the Third Party

They should not

- Call the callback too early (before it's been tracked)
- Call the callback too late (or never)
- Call the callback too few or too many times
- Fail to pass along any necessary environment/parameters to your callback
- Swallow any errors/exceptions that may happen

Promises: An Analogy

Promises

- A Promise is an object that is used as a placeholder for the eventual results of a deferred, and usually asynchronous, computation.
- A Promise has three states:
  - Pending – the result is not ready
  - Resolved or fulfilled – the result is available
  - Rejected – an error occurred
- A Promise is *settled* when it is resolved or rejected

Using Promises: The Fetch API

- `fetch()` is the modern way of sending requests over HTTP
- It is a two-stage process: first the header is requested followed by the body

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

  - `url` – the URL to access
  - `options` – optional parameters: method, headers etc.

- The browser initiates the request immediately and a promise is returned
- The promise resolves with an object of the built-in `Response` class as soon as the server responds with headers (even if they signal HTTP-errors, such as 404 or 500)
- The promise rejects if `fetch` is unable to make the HTTP-request (network problems, site does not exist)
- To get the response body an additional method call is needed

How fetch works

```
fetch('http://example.com/movies.json') // contains the HTTP response
  .then(response => response.json()) //  contains body content extracted as JSON
  .then(data => console.log(data));  // logs the data to the console
  .catch(e => alert(e)) // if one of the above promises rejects
```

Creating Promises using the Promise Constructor

```
const promise = new Promise((resolve, reject) => {
if (allWentWell) {
  resolve('All things went well!')
} else {
  reject('Something went wrong')
}
})
```

- The executor function is immediately executed when a promise is created
- The promise is resolved by calling `resolve` and rejected by calling `reject`
- A promise can be resolved or rejected only once

```
const promise = new Promise((resolve, reject) => {
  resolve('Promise resolved')  // Promise is resolved
  reject('Promise rejected')   // Promise can't be rejected
})
```

A Better Promise Example

```javascript
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {  // setTimeout used to create an async operation
                      // typical of promises
    if (Math.random() * 100 < 90) {
      console.log('resolving the promise ...')
      resolve('Hello, Promises!')
    }
    reject(new Error('In 10% of the cases, I fail. Miserably.'))
  }, 1000)
})
```

Promise Chaining

- `then()` and `catch()` return a new promise which can be handled by chaining another `then()`
- Promise chaining is used when promises need to be resolved in a sequence (refer to callback hell)

```
const delay = function (ms) {
  return new Promise(resolve => setTimeout(resolve, ms)) // cannot be rejected
}
```

Promise Chaining Example

```
delay(2000)
  .then(() => {
    console.log('Resolved after 2 seconds')
    return delay(1500)
  })
  .then(() => {
    console.log('Resolved after 1.5 seconds')
    throw new Error() // equivalent to: reject()
  })
  .catch(() => console.log('Caught an error'))
  .then(() => console.log('Done'))
```

Manually Creating Resolved Promises

```
Promise.resolve(5)
  .then(n => n * 2)
  .then(n => n + 1)
  .then(n => n.toString())
  .then(n => console.log(n))

// Output: 11
```

Promises Timing — Mixing Synchronous and Asynchronous Code

```
const array = []
array.push('before')

Promise.resolve().then(() => {
  array.push('then')
})

array.push('after')

console.log(array)
```

Promises Timing Continued — Another Example

```
Promise.resolve(5)
  .then(n => {
    console.log('First callback for resolved promise')
    return n * 2
  })
  .then(n => {
    console.log('Second callback for resolved promise')
    return n + 1
  })
  .then(n => n.toString())
  .then(n => console.log(n))

Promise.reject(new Error('Rejected'))
.catch(e => console.log(e.message))
```

Waiting for multiple promises: .all

```
function getUser (id) {
  const users = [
    { id: 1, name: 'Joel' },
    { id: 2, name: 'Carla' },
    { id: 3, name: 'Tsholofelo' }
  ]
  const user = users.find(user => user.id === id)
  return new Promise(resolve => resolve(user))
}

Promise.all([getUser(2), getUser(3)])
  .then(users => {
    const usernames = users.map(user => user.name)
    console.log(usernames)
  })
// Output: [ 'Carla', 'Tsholofelo' ]
```

Waiting for multiple promises continued

```
let addImg = (src) => {
  let imgElement =
    document.createElement("img")
  imgElement.src = src
  document.body.appendChild(imgElement)
}

Promise.all([
  loadImage('images/cat1.jpg'), // loadImage returns a Promise
  loadImage('images/cat2.jpg'), // will reject if it cannot find the image
  loadImage('images/cat3.jpg'), // otherwise contains an image object
]).then((images) => {
  images.forEach(img => addImg(img.src))
}).catch((error) => {
  // handle error later
})
```