

<WA1/>

2020

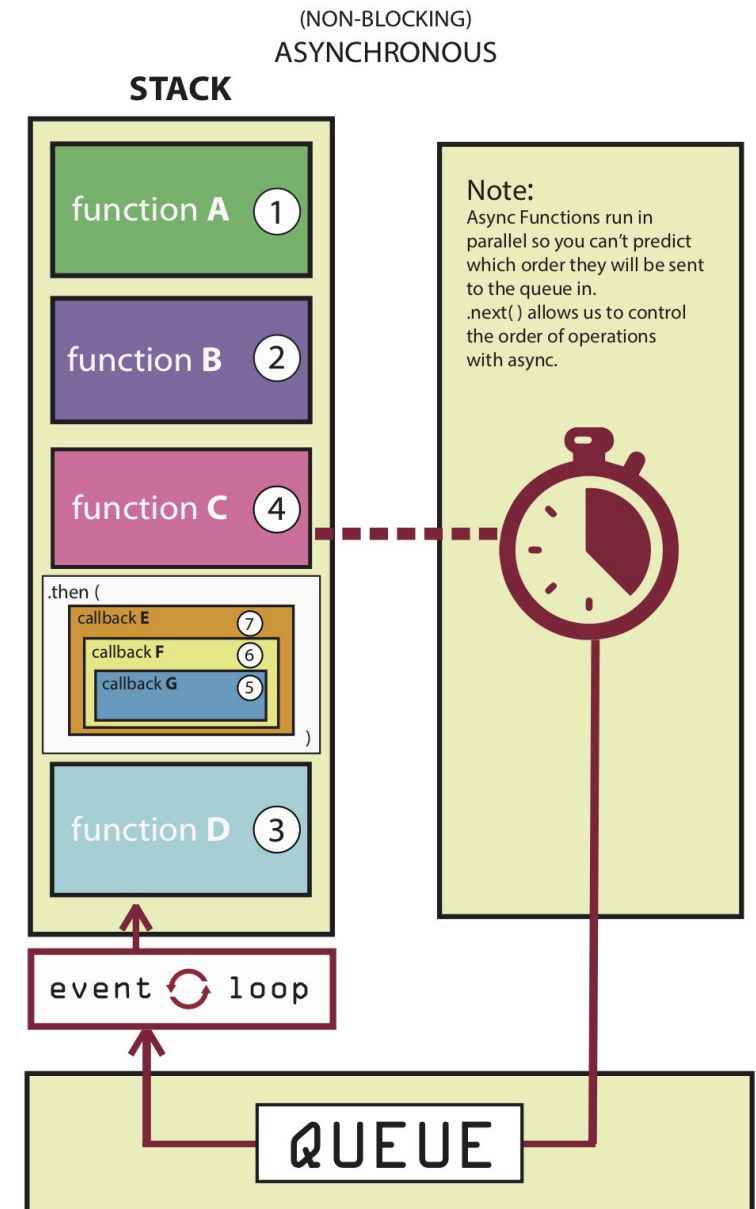
# Asynchronous Programming in JS

“The” language of the Web

Enrico Masala

Fulvio Corno

Luigi De Russis





## JavaScript: The Definitive Guide, 7th Edition Chapter 11. Asynchronous JavaScript

### Mozilla Developer Network

- [Learn web development JavaScript » Dynamic client-side scripting » Asynchronous JavaScript](#)
- [Web technology for developers » JavaScript » Concurrency model and the event loop](#)
- [Web technology for developers » JavaScript » JavaScript Guide » Using Promises](#)

JavaScript – The language of the Web

# ASYNCHRONOUS PROGRAMMING

# Asynchronicity

- JavaScript is single-threaded and inherently synchronous
  - i.e., code cannot create threads and run in parallel in the JS engine
- Callbacks are the most fundamental way for writing asynchronous JS code
- How can they work asynchronously?
  - e.g., how can `setTimeout()` or other async callbacks work?
- Thanks to the Execution Environment
  - e.g., browsers and Node.js
- and the Event Loop

```
const deleteAfterTimeout = (task) =>
{
  // do something
}
// runs after 2 seconds
setTimeout(deleteAfterTimeout, 2000,
task)
```

# Non-Blocking Code!

- Asynchronous techniques are very useful, particularly for web development
- For instance: when a web app runs executes an intensive chunk of code without returning control to the browser, the browser can appear to be frozen
  - this is called blocking, and it should be the exception!
    - the browser is blocked from continuing to handle user input and perform other tasks until the web app returns control of the processor
- This may happen outside browsers, as well
  - e.g., reading a long file from the disk/network, accessing a database and returning data, accessing a video stream from a web cam, etc.
- Most of the JS execution environments are, therefore, deeply asynchronous
  - with non-blocking primitives
  - JavaScript programs are event-driven, typically

# Back to Callbacks

- The most fundamental way for writing asynchronous JS code
- Great for "simple" things!

```
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('Task description: ', (answer) => {
  let description = answer;

  rl.close();
});
```

# Handling Errors in Callbacks

- No “official” ways, only best practices!
- Typically, the first parameter of the callback function is for storing any error, while the second one is for the result of the operation
  - this is the strategy adopted by Node.js, for instance

```
fs.readFile('/file.json', (err, data) => {  
  if (err !== null) {  
    console.log(err);  
    return;  
  }  
  //no errors, process data  
  console.log(data);  
});
```

# Beware: Callback Hell!

- If you want to perform multiple asynchronous actions in a row using callbacks, you must keep passing new functions to handle the continuation of the computation after the previous action
  - every callback adds a level of nesting
  - when you have lots of callbacks, the code starts to be complicated very quickly

```
const readline = require('readline');
const rl = readline.createInterface(...);

rl.question('Task description: ', (answer) => {
  let description = answer;

  rl.question('Is the task important? (y/n)', (answer) => {
    let important = answer;

    rl.question('Is the task private? (y/n)', (answer) => {
      let private = answer;

      rl.question('Task deadline: ', (answer) => {
        let date = answer;
        ...
      })
    })
  })

  rl.close();
});
```

# Callback Hell

```
window.addEventListener('load', () => {  
  document.getElementById('button').addEventListener('click', () => {  
    setTimeout(() => {  
      items.forEach(item => {  
        //your code here  
      })  
    }, 2000);  
  })  
})
```



# Promises

- A core language feature to simplify asynchronous programming
  - a possible solution to callback hell, too!
  - a fundamental building block for "newer" functions (async, ES2017)
- It is an **object** representing the eventual **completion** (or **failure**) of an asynchronous operation
  - i.e., an asynchronous function returns *a promise to supply the value* at some point in the future, instead of returning immediately a final value
- Promises standardize a way to handle errors and provide a way for errors to propagate correctly through a chain of promises

# Promises

- Promises can be created or consumed
  - many Web APIs expose Promises to be consumed!
- When consumed:
  - a Promise starts in a **pending** state
    - the caller function continues the execution, while it waits for the Promise to do its own processing, and give the caller function some “responses”
  - then, the caller function waits for it to either return the promise in a fulfilled state or in a rejected state

```
let duration = 10;

const waitPromise = new Promise((resolve, reject) => {
  if (duration >= 0) {
    // the promise can be fulfilled!
    resolve("It works!");
  } else {
    // time travel? we reject the promise
    reject(new Error("It doesn't work."));
  }
});

waitPromise.then((result) => {
  console.log("Success: ", result);
}).catch((error) => {
  console.log("Error: ", error);
});
```

# Creating a Promise

- A Promise object is created using the new keyword and its constructor
- The constructor takes an *executor function*, as its parameter
- This function takes two functions as parameters:
  - resolve, called when the asynchronous task completes successfully and returns the results of the task as a value
  - reject, called when the task fails and returns the reason for failure (an error object, typically)

```
const myPromise = new Promise((resolve,
reject) => {

    // do something asynchronous which
    eventually call either:

        resolve(someValue); // fulfilled

    // or

        reject("failure reason"); // rejected

});
```

# Creating a Promise

- You can also provide a function with “promise functionality”
- Simply have it return a promise!

```
function wait(duration) {  
  // Create and return a new promise  
  return new Promise((resolve, reject) => {  
    // If the argument is invalid,  
    // reject the promise  
    if (duration < 0) {  
      reject(new Error('Time travel not yet  
implemented'));  
    } else {  
      // otherwise, wait asynchronously and then  
      // resolve the Promise; setTimeout will  
      // invoke resolve() with no arguments:  
      // the Promise will fulfill with  
      // the undefined value  
      setTimeout(resolve, duration);  
    }  
  });  
}
```

# Consuming a Promise

- When a Promise is **fulfilled**, the **then()** callback is used
- If a Promise is **rejected**, instead, the **catch()** callback will handle the error
- **then()** and **catch()** are instance methods defined by the Promise object
  - each function registered with **then()** is invoked only once
- You can omit **catch()**, if you are interested in the result, only

```
waitPromise.then((result) => {
    console.log("Success: ", result);
}).catch((error) => {
    console.log("Error: ", error);
});

// if a function returns a Promise...
wait(1000).then(() => {
    console.log("Success!");
}).catch((error) => {
    console.log("Error: ", error);
});
```

# Consuming a promise

- `p.then(onFulfilled[, onRejected]);`
  - Callbacks are executed asynchronously (inserted in the event loop) when the promise is either fulfilled (success) or rejected (optional)
- `p.catch(onRejected);`
  - Callback is executed asynchronously (inserted in the event loop) when the promise is rejected
- `p.finally(onFinally);`
  - Callback is executed in any case, when the promise is either fulfilled or rejected.
  - Useful to avoid code duplication in then and catch handlers
- All these methods return Promises, too!

# Chaining Promises

- One of the most important benefits of Promises
- They provide a natural way to express a sequence of asynchronous operations as a **linear chain of `then()`** invocations
  - **without having to nest** each operation within the callback of the previous one
    - the "callback hell" seen before
- Important: Always return results, otherwise callbacks won't get the result of a previous promise

```
getRepoInfo()  
  .then(repo => getIssue(repo))  
  .then(issue => getOwner(issue.ownerId))  
  .then(owner => sendEmail(owner.email,  
    'Some text'))  
  .catch(e => {  
    // just log the error  
    console.error(e)  
  })  
  .finally(_ => logAction());  
});
```

# Example Chaining

- Useful, for instance, with I/O API such as `fetch()`, which returns a Promise

```
const status = (response) => {  
  if (response.status >= 200 && response.status < 300) {  
    return Promise.resolve(response) // static method to return a fulfilled Promise  
  }  
  return Promise.reject(new Error(response.statusText))  
}  
const json = (response) => response.json()  
  
fetch('/todos.json')  
  .then(status)  
  .then(json)  
  .then((data) => { console.log('Request succeeded with JSON response', data) })  
  .catch((error) => { console.log('Request failed', error) })
```



# Promises... in Parallel

```
Promise.all(promises)
  .then(results => console.log(results));
})
.catch(e => console.error(e));
```

- What if we want to execute several asynchronous operations in parallel?
- `Promise.all()`
  - takes an array of Promise objects as its input and returns a Promise
  - the returned Promise will be rejected if at least one of the input Promises is rejected
  - otherwise, it will be fulfilled with an array of the fulfillment values for each of the input promises
  - the input array can contain non-Promise values, too: if an element of the array is not a Promise, it is simply copied unchanged into the output array
- `Promise.race()`
  - returns a Promise that is fulfilled or rejected when **the first** of the Promises in the input array is fulfilled or rejected
  - if there are any non-Promise values in the input array, it simply returns the first of those

# Simplyfing writing with async / await

- ECMAScript 2017 (**ES8**) introduces two new keywords **async** **await**
  - write promise-based asynchronous code that **looks like** synchronous code
- Prepend **async** keyword to any function means that it will return a promise
- Prepend **await** when calling an async function (or a function returning a Promise) makes the calling code stop until the promise is resolved or rejected

```
const sampleFunction = async () => {  
  return 'test'  
}  
sampleFunction().then(console.log) // This will log 'test'
```

# async functions

- The async function declaration defines an asynchronous function
  - a function that is an AsyncFunction object
- Asynchronous functions operate in a separate order than the rest of the code via the **event loop**, returning an **implicit Promise** as their result
  - but the syntax and structure of code using async functions looks like standard synchronous functions.

```
async function name([param[, param[, ...param]]]) {  
  statements }
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async\\_function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function)

# await

- The await operator can be used to wait for a Promise. It can *only be used inside an async function*
- await **blocks** the code execution within the async function **until the Promise is resolved**
- When resumed, the value of the await expression is that of the fulfilled Promise
- If the Promise is rejected, the await expression **throws** the rejected value
  - If the value of the expression following the await operator is not a Promise, it's converted to a resolved Promise

```
returnValue = await expression;
```

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await>

# Example: async / await

```
function resolveAfter2Seconds() {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve('resolved');  
    }, 2000);  
  });  
}  
  
async function asyncCall() {  
  console.log('calling');  
  const result = await resolveAfter2Seconds();  
  console.log(result);  
}  
  
asyncCall();
```

} Return a  
promise

} async is needed to use await

} Looks like  
sequential  
code

```
> "calling"  
//... 2 seconds  
> "resolved"
```

# Example: async / await

```
function resolveAfter2Seconds() {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve('resolved');  
    }, 2000);  
  });  
}  
  
async function asyncCall() {  
  console.log('calling');  
  const result = await resolveAfter2Seconds();  
  return 'end';  
}  
  
asyncCall().then(console.log);
```

} Implicitly returns a Promise

} Can use Promise methods

```
> "calling"  
//... 2 seconds  
> "end"
```

# Examples... Before and After

```
const makeRequest = () => {  
  return getAPIData()  
    .then(data => {  
      console.log(data);  
      return "done";  
    })  
};  
  
let res = makeRequest();
```

```
const makeRequest = async () => {  
  console.log(await getAPIData());  
  return "done";  
};  
  
let res = makeRequest();
```

# Examples... Before and After

```
function getData() {  
    return getIssue()  
        .then(issue =>  
            getOwner(issue.ownerId))  
        .then(owner =>  
            sendEmail(owner.email, 'Some text'));  
}
```

// assuming that all the 3 functions  
above return a Promise

```
async function getData = {  
    const issue = await getIssue();  
    const owner = await  
        getOwner(issue.ownerId);  
    await sendEmail(owner.email, 'Some  
text');  
}
```



# Converting Promise-based Function to async/await with Visual Studio Code



Umar Hansa @umaar · Sep 28, 2018

Visual Studio Code can now convert your long chains of Promise.then()'s into async/await! 🎉 Works very well in both JavaScript and TypeScript files. .catch() is also correctly converted to try/catch ✅

```
TS promise-async-await.ts x
1
2 function example() {
3   return Promise.resolve(1)
4     .then(() => {
5       return Promise.resolve(2);
6     }).then((value) => {
7       console.log(value)
8       return Promise.reject(3)
9     }).catch(err => {
10      console.log(err);
11    })
12 }
13
14 function get() {
15   return fetch('https://umaar.com')
16     .then(res => res.text())
17     .catch(err => console.log('Error', err))
18 }
19
```

GIF

<https://twitter.com/i/status/1045655069478334464>

# Chaining with async/await

- Simpler to read, easier to debug
  - debugger would not stop on asynchronous code

```
const getFirstUserData = async () => {  
  const response = await fetch('/users.json'); // get users list  
  const users = await response.json(); // parse JSON  
  const user = users[0]; // pick first user  
  const userResponse = await fetch(`/users/${user.name}`); // get user data  
  const userData = await user.json(); // parse JSON  
  return userData;  
}  
getFirstUserData()
```

# Promises or async/await? Both!

- If the output of `function2` is dependent on the output of `function1`, use `await`.
- If two functions can be run in parallel, create two different async functions and then run them in parallel `Promise.all(promisesArray)`
- Instead of creating huge async functions with many `await asyncFunction()` in it, it is better to create **smaller** async functions (not too much blocking code)
- If your code contains blocking code, it is better to make it an async function. The callers can decide on the level of asynchronicity they want.

<https://medium.com/better-programming/should-i-use-promises-or-async-await-126ab5c98789>

# License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give [appropriate credit](#), provide a link to the license, and [indicate if changes were made](#). You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for [commercial purposes](#).
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the [same license](#) as the original.
  - **No additional restrictions** — You may not apply legal terms or [technological measures](#) that legally restrict others from doing anything the license permits.
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>

