



(<https://bit.ly/2eF8uO2>)

JavaScript Promises for Dummies

Jecelyn Yeen (@jecelyn) (@jecelynyeen) (<https://twitter.com/jecelynyeen>) December 01, 2016

Jecelyn Yeen(@jecelyn)

For Dummies

JavaScript Promises

Code(<https://github.com/chybie/js-async-await-promise>)

Demo(<https://jsbin.com/nifocu/1/edit?js,console>)

Project(<https://jsbin.com/nifocu/1/edit?js,console>)

Javascript Promises are not difficult. However, lots of people find it a little bit hard to understand at the beginning. Therefore, I would like to write down the way I understand promises, in a dummy way.

Related Course: [Getting Started with JavaScript for Web Development](https://scotch.io/courses/getting-started-with-javascript)
(<https://scotch.io/courses/getting-started-with-javascript>)

#Understanding Promises



kid. Your mom **promises** you that she'll get you a **new phone**

(<https://bit.ly/2kDfzJ>) You don't know if you will get that phone until next week. Your mom can either *really buy you a brand new phone*, or *stand you up* and withhold the phone if she is not

A promise has 3 states. They are:

Pending: You don't know if you will get that phone until next week.

Fulfilled: Your mom really buy you a brand new phone.

Rejected: You don't get a new phone because your mom is not happy.

Creating a Promise

↳ JavaScript.

JAVASCRIPT

```
/* ES5 */
var isMomHappy = false;

// Promise
var willIGetNewPhone = new Promise(
    function (resolve, reject) {
        if (isMomHappy) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone); // fulfilled
        } else {
            var reason = new Error('mom is not happy');
            reject(reason); // reject
        }
    }
);
```

The code is quite expressive in itself.

1. We have a boolean `isMomHappy`, to define if mom is happy.
2. We have a promise `willIGetNewPhone`. The promise can be either `resolved` (if mom get you a new phone) or `rejected` (mom is not happy, she doesn't buy you one).



(<https://bit.ly/2eF8uO2>)

Standard syntax to define a new `Promise`, refer to MDN (https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise), the syntax look like this.

JAVASCRIPT

```
// promise syntax look like this
new Promise(/* executor*/ function (resolve, reject) { ... } );
```

to remember is, when the result is successful, call `resolve(success_value)`, if the result fails, call `reject(reason)`. In our example, if mom is happy, we will get a phone. Therefore, we call `resolve` function with `phone` variable. If mom is not happy, we will call `reject` function with a reason `reject(reason)`;

Consuming Promises

Now that we have created the promise, let's consume it.

JAVASCRIPT

```
/* ES5 */
...
// call our promise
var askMom = function () {
    willIGetNewPhone
        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
            // output: { brand: 'Samsung', color: 'black' }
        })
        .catch(function (error) {
            // oops, mom don't buy it
            console.log(error.message);
            // output: 'mom is not happy'
        });
};

askMom();
```

1. We have a function call `askMom`. In this function, we will consume our promise `willIGetNewPhone`.

2. We want to take some action once the promise is resolved or rejected, we use `.then` and `.catch` to handle our action.



In this example, we have `function(fulfilled) { ... }` in `.then`. What is the value of `fulfilled`? The `fulfilled` value is exactly the value you pass in your promise, `give(your_success_value)`. Therefore, it will be `phone` in our case. And we have `function(error){ ... }` in `.catch`. What is the value of `error`? As you can guess, the `error` value is exactly the value you pass in your promise `reject(your_fail_value)`. Therefore, it will be `reason` in our case.

Let's run this code in a browser console and see the result!

Open this link in your browser: <https://jsbin.com/nifocu/1/edit?js,console> (<https://jsbin.com/nifocu/1/edit?js,console>)



#Chaining Promises

Promises are chainable.

Let's say, you, the kid, **promise** your friend that you will **show them** the new phone when your mom buy you one.

That is another promise. Let's write it!

...

```
// 2nd promise
var showOff = function (phone) {
    return new Promise(
        function (resolve, reject) {
            var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';
            (https://bit.ly/2eF8uO2)             resolve(message);
        }
    );
};
```



LEARN ANGULAR
BY BUILDING
The practical Angular book

[Get the Book](#)

, you might realize we didn't call the `reject`. It's optional.
In this sample like using `Promise.resolve` instead.

```
// shorten it
...

// 2nd promise
var showOff = function (phone) {
    var message = 'Hey friend, I have a new ' +
        phone.color + ' ' + phone.brand + ' phone';

    return Promise.resolve(message);
};
```

Let's chain the promises. You, the kid can only start the `showoff` promise after the `willIGetNewPhone` promise.

...



```
// call our promise
var askMom = function () {
    willIGetNewPhone
        .then(showOff) // chain it here
        .then(function (fulfilled) {
            console.log(fulfilled);
            // output: 'Hey friend, I have a new black Samsung phone.'
        })
        .catch(function (error) {
            // oops, mom don't buy it
            console.log(error.message);
            // output: 'mom is not happy'
        });
};
```

chain the promise.

es are Asynchronous

hronous. Let's log a message before and after we call the

```
// call our promise
var askMom = function () {
    console.log('before asking Mom'); // log before
    willIGetNewPhone
        .then(showOff)
        .then(function (fulfilled) {
            console.log(fulfilled);
        })
        .catch(function (error) {
            console.log(error.message);
        });
    console.log('after asking mom'); // log after
}
```

What is the sequence of expected output? Probably you expect:

1. before asking Mom
2. Hey friend, I have a new black Samsung phone.
3. after asking mom



(<https://bit.ly/2eF8uO2>)

Final output sequence is:

1. before asking Mom
2. after asking mom
3. Hey friend, I have a new black Samsung phone.

 A screenshot of a browser's developer tools console. On the left, there is a snippet of JavaScript code involving promises. On the right, the console window shows the output of the code execution. The output includes a blue arrow indicating the current execution step, followed by the text "you got a new phone" in green, which corresponds to the fulfilled state of the promise.


```

n () {
  ore asking Mom');
}

f)

on (fulfilled) {
  you got a new phone
  log(fulfilled);

  ion (error) {
    mom don't buy it
    log(error.message);

    er asking mom');
  }
}
  
```

Why? Because life (or JS) waits for no man.

You, the kid, wouldn't stop playing while waiting for your mom promise (the new phone). Don't you? That's something we call **asynchronous**, the code will run without blocking or waiting for the result. Anything that need to wait for promise to proceed, you put that in `.then`.

#Promises in ES5, ES6/2015, ES7/Next

ES5 - Majority browsers

The demo code is workable in ES5 environments (all major browsers + NodeJs) if you include Bluebird (<http://bluebirdjs.com/docs/getting-started.html>) promise library. It's because ES5 doesn't support promises out of the box. Another famous

promise library is Q (<https://github.com/kriskowal/q>) by Kris Kowal.

ES6 / ES2015 - Modern browsers, NodeJs v6

The demo code works out of the box because ES6 supports promises natively. In



functions, we can further simplify the code with **fat arrow =>** and

.

of ES6 code:

(<https://bit.ly/2eF8uO2>)

JAVASCRIPT

```
/* ES6 */
const isMomHappy = true;

// Promise
const willIGetNewPhone = new Promise(
  (resolve, reject) => { // fat arrow
    if (isMomHappy) {
      const phone = {
        brand: 'Samsung',
        color: 'black'
      };
      resolve(phone);
    } else {
      const reason = new Error('mom is not happy');
      reject(reason);
    }
  }
);

const showOff = function (phone) {
  const message = 'Hey friend, I have a new ' +
    phone.color + ' ' + phone.brand + ' phone';
  return Promise.resolve(message);
};

// call our promise
const askMom = function () {
  willIGetNewPhone
    .then(showOff)
    .then(fulfilled => console.log(fulfilled)) // fat arrow
    .catch(error => console.log(error.message)); // fat arrow
};

askMom();
```

Notes that all the `var` are replaced with `const`. All the `function(resolve, reject)` has been simplified to `(resolve, reject) =>`. There are a few benefits come with these changes. Read more on:-

- JavaScript ES6 Variable Declarations with let and const
(<https://strongloop.com/strongblog/es6-variable-declarations/>)
- An introduction to Javascript ES6 arrow functions
(<https://strongloop.com/strongblog/an-introduction-to-javascript-es6-arrow->

functions/)

ES7 - Async Await make the syntax look prettier

ES7 introduce `async` and `await` syntax. It makes the asynchronous syntax look easier to understand, without the `.then` and `.catch`.



(<https://bit.ly/2eF8uO2>)

```
/* ES7 */
const isMomHappy = true;
```



```
// Promise
BY EXAMPLE
The practical Angular book
Get the Book
(https://bit.ly/2eF8uO2)

// Promise
const willIGetNewPhone = new Promise(
  (resolve, reject) => {
    if (isMomHappy) {
      const phone = {
        brand: 'Samsung',
        color: 'black'
      };
      resolve(phone);
    } else {
      const reason = new Error('mom is not happy');
      reject(reason);
    }
  }
);

// 2nd promise
async function showOff(phone) {
  return new Promise(
    (resolve, reject) => {
      var message = 'Hey friend, I have a new ' +
        phone.color + ' ' + phone.brand + ' phone';

      resolve(message);
    }
  );
}

// call our promise
async function askMom() {
  try {
    console.log('before asking Mom');

    let phone = await willIGetNewPhone;
    let message = await showOff(phone);

    console.log(message);
    console.log('after asking mom');
  }
  catch (error) {
    console.log(error.message);
  }
}

(async () => {
  await askMom();
})();
```

1. Whenever you need to return a promise in a function, you prepend `async` to that function. E.g. `async function showOff(phone)`

2. Whenever you need to call a promise, you prepend with `await`. E.g.

```
let phone = await willIGetNewPhone; and let message = await showOff(phone); .
```

3. Use `try { ... } catch(error) { ... }` to catch promise error, the **rejected** promise.



Promises and When to Use Them?

(<https://bit.ly/2eTSUOZ>)

Promises? How's the world look like before promise? Before questions, let's go back to the fundamental.

Function vs Async Function

In these two examples, both examples perform addition of two numbers. One uses a normal function, the other adds remotely.

Normal Function to Add Two Numbers

JAVASCRIPT

```
// add two numbers normally

function add (num1, num2) {
    return num1 + num2;
}

const result = add(1, 2); // you get result = 3 immediately
```

Async Function to Add Two numbers

JAVASCRIPT

```
// add two numbers remotely

// get the result by calling an API
const result = getAddResultFromServer('http://www.example.com?num1=1&num2=2');
// you get result = "undefined"
```

If you add the numbers with a normal function, you get the result immediately. However, when you issue a remote call to get the result, you need to wait, you can't get the result immediately.

Or put it this way, you don't know if you will get the result because the server might be down, slow in response, etc. You don't want your entire process to be blocked while waiting for the result.



Must we use promise for asynchronous call? Nope. Prior to Promise, we use callback. Callback is just a function you call when you get the return result. Let's

s example to accept a callback.

JAVASCRIPT

```
// add two numbers remotely
// get the result by calling an API

function addAsync (num1, num2, callback) {
  // use the famous jQuery getJSON callback API
  return $.getJSON('http://www.example.com', {
    num1: num1,
    num2: num2
  }, callback);
}

addAsync(1, 2, success => {
  // callback
  const result = success; // you get result = 3 here
});
```

<, why do we need promises then?

What if You Want to Perform Subsequent Async Action?

Let's say, instead of just add the numbers one time, we want to add 3 times. In a normal function, we do this:-



scotch (/)

LOGIN (/LOGIN)

SIGN UP (/REGISTER)

```
// add two numbers normally

let resultA, resultB, resultC;

function add (num1, num2) {
    return num1 + num2;
}

resultA = add(1, 2); // you get resultA = 3 immediately
(resultB = add(resultA, 3)); // you get resultB = 6 immediately
(resultC = add(resultB, 4)); // you get resultC = 10 immediately
```



LEARN ANGULAR

BY BUILDING

The practical Angular book

[Get the Book](#)

(<https://bit.ly/2eF8uQ2>)

resultB = add(resultA, 3); // you get resultB = 6 immediately

resultC = add(resultB, 4); // you get resultC = 10 immediately

console.log('total' + resultC);

console.log(resultA, resultB, resultC);

ith callbacks?

```
// add two numbers normally
// get the result by calling an API

let resultA, resultB, resultC;

function addAsync (num1, num2, callback) {
    // use the famous jQuery getJSON callback API
    return $.getJSON('http://www.example.com', {
        num1: num1,
        num2: num2
    }, callback);
}

addAsync(1, 2, success => {
    // callback 1
    resultA = success; // you get result = 3 here

    addAsync(resultA, 3, success => {
        // callback 2
        resultB = success; // you get result = 6 here

        addAsync(resultB, 4, success => {
            // callback 3
            resultC = success; // you get result = 10 here

            console.log('total' + resultC);
            console.log(resultA, resultB, resultC);
        });
    });
});
```

Demo: <https://jsbin.com/barimo/edit?html,js,console> (<https://jsbin.com/barimo/edit?html,js,console>)

The syntax is less user friendly. In a nicer term, It looks like a pyramid, but people usually refer this as "callback hell", because the callback nested into another callback. Imagine you have 10 callbacks, your code will nested 10 times!



(<https://bit.ly/2eF8uO2>)

From Callback Hell

to rescue. Let's look at the promise version of the same example.

JAVASCRIPT

```
// add two numbers remotely using observable

let resultA, resultB, resultC;

function addAsync(num1, num2) {
    // use ES6 fetch API, which return a promise
    return fetch(`http://www.example.com?num1=${num1}&num2=${num2}`)
        .then(x => x.json());
}

addAsync(1, 2)
    .then(success => {
        resultA = success;
        return resultA;
    })
    .then(success => addAsync(success, 3))
    .then(success => {
        resultB = success;
        return resultB;
    })
    .then(success => addAsync(success, 4))
    .then(success => {
        resultC = success;
        return resultC;
    })
    .then(success => {
        console.log('total: ' + success)
        console.log(resultA, resultB, resultC)
    });
}
```

Demo: <https://jsbin.com/qafane/edit?js,console> (<https://jsbin.com/qafane/edit?js,console>)

With promises, we flatten the callback with `.then`. In a way, it looks cleaner because of no callback nesting. Of course, with ES7 `async` syntax, we can even further enhance this example, but I leave that to you. :)

#New Kid On the Block: Observables



(<https://bit.ly/2eF8uO2>)

own with promises, there is something that has come about to deal with async data called Observables .

s are lazy event streams which can emit zero or more values, and may or may not finish.

(<https://cycle.js.org/streams.html>)

es between promises and observable are:

- are cancellable
- are lazy

the same demo written with Observables. In this example, I am using the RxJS Observable API (<https://reactivex.io/rxjs/class/es6/Observable.js~Observable.html>) for the

JAVASCRIPT

```
let Observable = Rx.Observable;
let resultA, resultB, resultC;

function addAsync(num1, num2) {
    // use ES6 fetch API, which return a promise
    const promise = fetch(`http://www.example.com?num1=${num1}&num2=${num2}`)
        .then(x => x.json());

    return Observable.fromPromise(promise);
}

addAsync(1,2)
    .do(x => resultA = x)
    .flatMap(x => addAsync(x, 3))
    .do(x => resultB = x)
    .flatMap(x => addAsync(x, 4))
    .do(x => resultC = x)
    .subscribe(x => {
        console.log('total: ' + x)
        console.log(resultA, resultB, resultC)
    });
}
```

Demo: <https://jsbin.com/dosaviwalu/edit?js,console>
(<https://jsbin.com/dosaviwalu/edit?js,console>)

Notes:

- `Observable.fromPromise` converts a promise to observable stream.
- `.do` and `.flatMap` are among some of the operators available for Observables



or times. (https://bit.ly/2eF8uO2)

zy. Our `addAsync` runs when we `.subscribe` to it.

o more funky stuff easily. For example, `delay` add function by
t one line of code or retry so you can retry a call a certain number

JAVASCRIPT

```
...
addAsync(1,2)
  .delay(3000) // delay 3 seconds
  .do(x => resultA = x)
...
...
```

it Observables in future post!

ary

ir with callbacks and promises. Understand them and use them.
DON'T WORRY about Observables, just yet. All three can factor into your development
depending on the situation.

Here are the demo code for all mom promise to buy phone examples:

- Demo (ES5): [\(https://jsbin.com/habuwuyeqo/edit?html,js,console\)](https://jsbin.com/habuwuyeqo/edit?html,js,console)
- Demo (ES6): [\(https://jsbin.com/cezedu/edit?js,console\)](https://jsbin.com/cezedu/edit?js,console)
- Demo (ES7): [\(https://goo.gl/U3fPmh\)](https://goo.gl/U3fPmh)
- Github example (ES7): [\(https://github.com/chybie/js-async-await-promise\)](https://github.com/chybie/js-async-await-promise)

That's it. Hopefully this article smoothen your path to tame the JavaScript promises.
Happy coding!



(<https://bit.ly/2eF8uO2>)

(/[@jecelyn])

JECELYN YEEN (/@JECELYN)

Coder. Diver. Board Game Lover.

Speak English, Mandarin, JavaScript, Typescript, C# and more.

GDE | Angular | Web Technologies

(<https://developers.google.com/experts/people/jecelyn-yeen>)

[VIEW MY 20 POSTS \(/@JECELYN\)](#)

🔥 POPULAR

Code(<https://github.com/chybie/js-async-await-promise>)



(<https://bit.ly/2eF8uO2>)

Demo(<https://jsbin.com/nifocu/1/edit?js,console>)

Project(<https://jsbin.com/nifocu/1/edit?js,console>)

([@jecelyn](#))

Jecelyn Yeen ([@jecelyn](#))

20 posts ([@jecelyn](#))

Coder. Diver. Board Game Lover.

Speak English, Mandarin, JavaScript, Typescript, C#
and more.

GDE | Angular | Web Technologies

(<https://developers.google.com/experts/people/jecelyn-yeen>).



([@jecelyn](#)) (<https://twitter.com/jecelynyeen>) (<https://github.com/chybie>) (<https://about.me/jecelyn>)

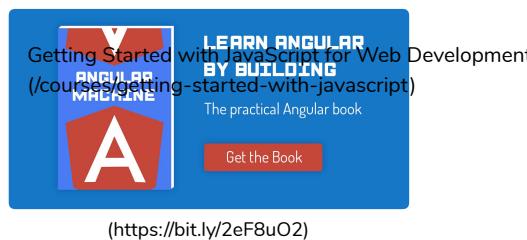
LATEST VIDEO COURSES

(</courses/getting-started-with-angular-2>)

(</courses/getting-started-with-react>)

Getting Started with Angular v2+ (</courses/getting-started-with-angular-2>)

Getting Started with React (</courses/getting-started-with-react>)



(<https://bit.ly/2eF8uO2>)

Get to Know Git (/courses/get-to-know-git)

Join Scotch

High Quality Content

The best tutorials and content that you'll find for web development. Guides, courses, tutorials, and more great content to learn with.

Build Real Apps

We won't just go over concepts and "Hello Worlds"; we'll **build real apps together** that you can use at your job or for your portfolio.

Not Just How, But Why

There are many different ways to code the same project. We'll show **best practices** and why certain choices are better than others.

Scotch Free

Write your own posts

Watch free lessons

Like favorite posts

Bookmark content for reference

Free (</registering?type=free>)

Scotch School

All of the free features

Access to **all premium content**

Downloadable videos

No ads across all of Scotch

Track **completed** content

\$20 (/registering?type=monthly)



(<https://bit.ly/2eF8uO2>)

Top shelf learning. Informative tutorials explaining
the code **and the choices behind it all.**

(<https://github.com/scotch->

(<https://tinyurl.com/scotchdevelopment>)

Brought to you from Las Vegas and DC By... (/about)

(<http://scotch.io/@nick>)

[FAQ \(/faq\)](#) [Privacy \(/privacy\)](#) [Terms \(/terms\)](#) [Rules \(/rules\)](#)

2017 © Scotch.io, LLC. All Rights Super Duper Reserved.

Proudly hosted by Digital Ocean (<https://m.do.co/c/7a59e9361ab7>)

Chris Sevilleja

Nick Cerminara