**What Is a Promise?**

Intro

The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

A Promise is in one of these states:

* *pending*: initial state, neither(ни) fulfilled nor(ни) 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). When either of these options happens, the result is passed to a method called "then".

A Promise object is created using the “new” keyword and its constructor. This constructor takes as its argument a function, called the "executor function". This function should take two functions as parameters. The first of these functions (resolve) is called when the asynchronous task completes successfully and returns the results of the task as a value. The second (reject) is called when the task fails, and returns the reason for failure, which is typically an error object.

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

// do something asynchronous which eventually calls either:

//

resolve(“success”); // fulfilled

// or

reject("failure reason"); // rejected

});

myPromise.then((successMessage) => {

console.log(successMessage); // success

});

Any promise we have, using ES2016, we can await. That’s literally all await means: it functions in exactly the same way as calling `.then()` on a promise (but without requiring any callback function).

const request = async (url) => {

const res = await fetch(url);

const json = await res.json();

return json;

};

## Chain Flow

Promises are not just a mechanism for a single-step this-then-that sort of operation. That's the building block, of course, but it turns out(оказывается) we can string multiple(обьеденить) Promises together to represent a sequence(последовательность) of async steps.

Every time you call then(..) on a Promise, it creates and returns a new Promise, which we can *chain* with.

var p = Promise.resolve( 21 );

var p2 = p.then( function(v){

console.log( v ); // 21

// fulfill `p2` with value `42`

return v \* 2;

} );

// chain off `p2`

p2.then( function(v){

console.log( v ); // 42

} );

By returning v \* 2 (i.e., 42), we fulfill the p2 promise that the first then(..) call created and returned. When p2's then(..) call runs, it's receiving the fulfillment from the return v \* 2 statement. Of course, p2.then(..) creates yet another promise, which we could have stored in a p3 variable.

But we can avoid creating variable p2 or p3.

var p = Promise.resolve( 21 );

p

.then( function(v){

console.log( v ); // 21

// fulfill the chained promise with value `42`

return v \* 2;

} )

// here's the chained promise

.then( function(v){

console.log( v ); // 42

} );

So now the first then(..) is the first step in an async sequence, and the second then(..) is the second step. This could keep going for as long as you needed it to extend.

.catch()

To avoid losing an error to the silence of a forgotten/discarded Promise, some developers have claimed that a "best practice" for Promise chains is to always end your chain with a final catch(..), like:

var p = Promise.resolwve( 42 );

p.then(

function fulfilled(msg){

// numbers don't have string functions,

// so will throw an error

console.log( msg.toLowerCase() );

}

)

.catch( handleErrors );

Because we didn't pass a rejection handler to the then(..), the default handler was substituted, which simply propagates the error to the next promise in the chain. As such, both errors that come into p, and errors that come after p in its resolution (like the msg.toLowerCase() one) will filter down to the final handleErrors(..).

**What happens if handleErrors(..) itself also has an error in it? Who catches that?**

You can't just stick another catch(..) on the end of that chain, because it too could fail.

.done()

Another more common suggestion is that Promises should have a done(..) added to them, which essentially marks the Promise chain as "done." done(..) doesn't create and return a Promise. Any exception inside a done(..) rejection handler would be thrown as a global uncaught error.

var p = Promise.resolve( 42 );

p.then(

function fulfilled(msg){

// numbers don't have string functions,

// so will throw an error

console.log( msg.toLowerCase() );

}

)

.done( null, handleErrors );

// if `handleErrors(..)` caused its own exception, it would

// be thrown globally here

## Promise Patterns

### Promise.all([ .. ])

Say you wanted to make two Ajax requests at the same time, and wait for both to finish, despite(несмотря на) of their order, before making a third Ajax request.

Consider:

// `request(..)` is a Promise-aware Ajax utility,

// like we defined earlier in the chapter

var p1 = request( "http://some.url.1/" );

var p2 = request( "http://some.url.2/" );

Promise.all( [p1,p2] )

.then( function(msgs){

// both `p1` and `p2` fulfill and pass in

// their messages here

return request(

"http://some.url.3/?v=" + msgs.join(",")

);

} )

.then( function(msg){

console.log( msg );

} );

Promise.all([ .. ]) expects a single argument, an array, consisting generally of Promise instances. The promise returned from the Promise.all([ .. ]) call will receive a fulfillment message (msgs in this snippet) that is an array of all the fulfillment messages from the passed in promises, in the same order as specified (regardless of fulfillment order).

### Promise.race([ .. ])

Similar to Promise.all([ .. ]), Promise.race([ .. ]) will fulfill if and when any Promise resolution is a fulfillment, and it will reject if and when any Promise resolution is a rejection.

Let's revisit our previous concurrent Ajax example, but in the context of a race between p1 and p2:

// `request(..)` is a Promise-aware Ajax utility,

// like we defined earlier in the chapter

var p1 = request( "http://some.url.1/" );

var p2 = request( "http://some.url.2/" );

Promise.race( [p1,p2] )

.then( function(msg){

// either `p1` or `p2` will win the race

return request(

"http://some.url.3/?v=" + msg

);

} )

.then( function(msg){

console.log( msg );

} );

Because only one promise wins, the fulfillment value is a single message, not an array as it was for Promise.all([ .. ]).