# JavaScript Callbacks

*"I will call back later!"*

A callback is a function passed as an argument to another function

This technique allows a function to call another function

A callback function can run after another function has finished

## Function Sequence

JavaScript functions are executed in the sequence they are called. Not in the sequence they are defined.

This example will end up displaying "Goodbye":

function myFirst() {  
  myDisplayer("Hello");  
}  
  
function mySecond() {  
  myDisplayer("Goodbye");  
}  
  
myFirst();  
mySecond();

## Sequence Control

Sometimes you would like to have better control over when to execute a function.

Suppose you want to do a calculation, and then display the result.

You could call a calculator function (myCalculator), save the result, and then call another function (myDisplayer) to display the result:

function myDisplayer(some) {  
  document.getElementById("demo").innerHTML = some;  
}  
  
function myCalculator(num1, num2) {  
  let sum = num1 + num2;  
  return sum;  
}  
  
let result = myCalculator(5, 5);  
myDisplayer(result);

Or, you could call a calculator function (myCalculator), and let the calculator function call the display function (myDisplayer):

function myDisplayer(some) {  
  document.getElementById("demo").innerHTML = some;  
}  
  
function myCalculator(num1, num2) {  
  let sum = num1 + num2;  
  myDisplayer(sum);  
}  
  
myCalculator(5, 5);

The problem with the first example above, is that you have to call two functions to display the result.

The problem with the second example, is that you cannot prevent the calculator function from displaying the result.

Now it is time to bring in a callback.

## JavaScript Callbacks

A callback is a function passed as an argument to another function.

Using a callback, you could call the calculator function (myCalculator) with a callback (myCallback), and let the calculator function run the callback after the calculation is finished:

### Example

function myDisplayer(some) {  
  document.getElementById("demo").innerHTML = some;  
}  
  
function myCalculator(num1, num2, myCallback) {  
  let sum = num1 + num2;  
  myCallback(sum);  
}  
  
myCalculator(5, 5, myDisplayer);

In the example above, myDisplayer is a called a **callback function**.

It is passed to myCalculator() as an **argument**.

## Note

When you pass a function as an argument, remember not to use parenthesis.

Right: myCalculator(5, 5, myDisplayer);

Wrong: ;

### Example

// Create an Array  
const myNumbers = [4, 1, -20, -7, 5, 9, -6];  
  
// Call removeNeg with a callback  
const posNumbers = removeNeg(myNumbers, (x) => x >= 0);  
  
// Display Result  
document.getElementById("demo").innerHTML = posNumbers;  
  
// Keep only positive numbers  
function removeNeg(numbers, callback) {  
  const myArray = [];  
  for (const x of numbers) {  
    if (callback(x)) {  
      myArray.push(x);  
    }  
  }  
  return myArray;  
}

### Example

// Create an Array  
const myNumbers = [4, 1, -20, -7, 5, 9, -6];  
  
// Call removeNeg with a callback  
const posNumbers = removeNeg(myNumbers, (x) => x >= 0);  
  
// Display Result  
document.getElementById("demo").innerHTML = posNumbers;  
  
// Keep only positive numbers  
function removeNeg(numbers, callback) {  
  const myArray = [];  
  for (const x of numbers) {  
    if (callback(x)) {  
      myArray.push(x);  
    }  
  }  
  return myArray;  
}

**JavaScript Promise Tutorial – How to Resolve or Reject Promises in JS**

Promises are important building blocks for asynchronous operations in JavaScript. You may think that promises are not so easy to understand, learn, and work with. And trust me, you are not alone!

Promises are challenging for many web developers, even after spending years working with them.

In this article, I want to try to change that perception while sharing what I've learned about JavaScript Promises over the last few years. Hope you find it useful.

# What is a Promise in JavaScript?

A Promise is a special JavaScript object. It produces a value after an asynchronous (aka, async) operation completes successfully, or an error if it does not complete successfully due to time out, network error, and so on.

Successful call completions are indicated by the resolve function call, and errors are indicated by the reject function call.

You can create a promise using the promise constructor like this:

let promise = new Promise(function(resolve, reject) {

// Make an asynchronous call and either resolve or reject

});

In most cases, a promise may be used for an asynchronous operation. However, technically, you can resolve/reject on both synchronous and asynchronous operations.

# Understanding Promise States

Just to review, a promise can be created with the constructor syntax, like this:

let promise = new Promise(function(resolve, reject) {

// Code to execute

});

The constructor function takes a function as an argument. This function is called the executor function.

// Executor function passed to the

// Promise constructor as an argument

function(resolve, reject) {

// Your logic goes here...

}

The executor function takes two arguments, resolve and reject. These are the callbacks provided by the JavaScript language. Your logic goes inside the executor function that runs automatically when a new Promise is created.

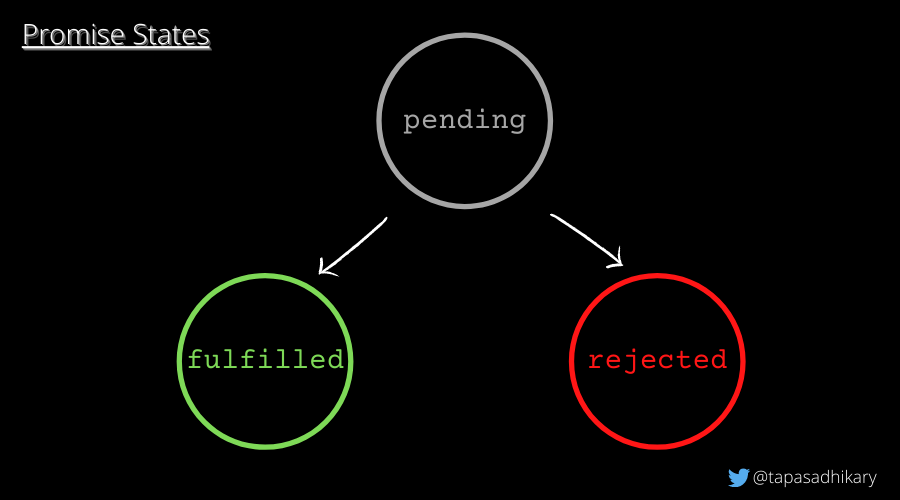
For the promise to be effective, the executor function should call either of the callback functions, resolve or reject. We will learn more about this in detail in a while.

The new Promise() constructor returns a promise object. As the executor function needs to handle async operations, the returned promise object should be capable of informing when the execution has been started, completed (resolved) or retuned with error (rejected).

A promise object has the following internal properties:

1. state – This property can have the following values:

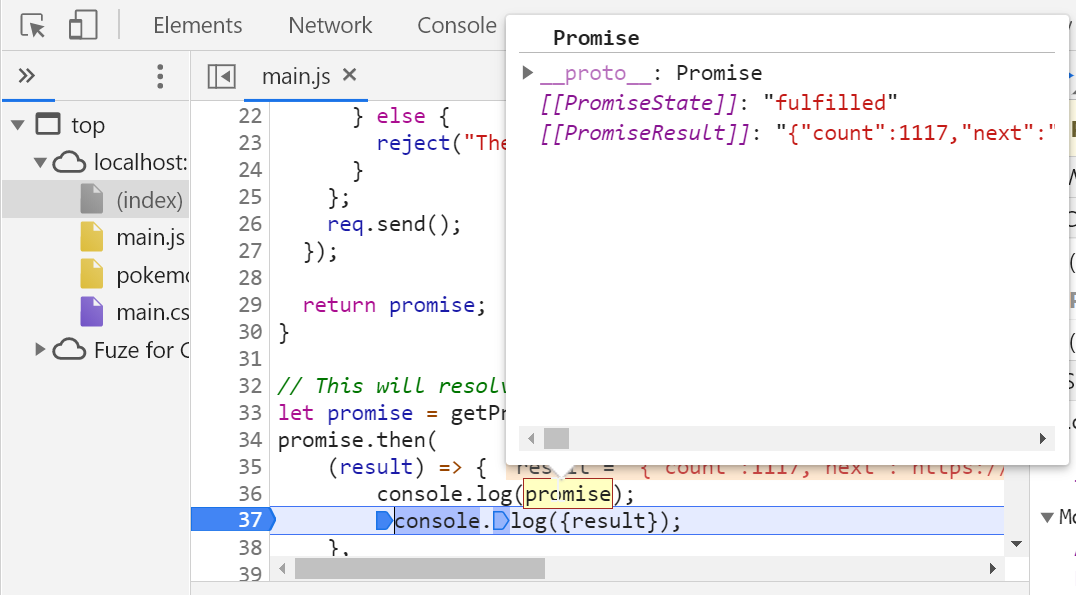
* pending: Initially when the executor function starts the execution.
* fulfilled: When the promise is resolved.
* rejected: When the promise is rejected.

Promise states

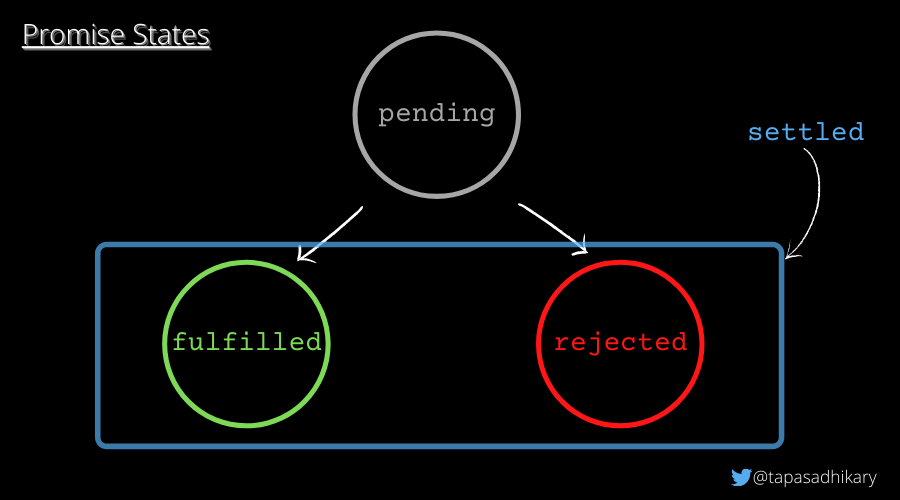
2.  result – This property can have the following values:

* undefined: Initially when the state value is pending.
* value: When resolve(value) is called.
* error: When reject(error) is called.

These internal properties are code-inaccessible but they are inspectable. This means that we will be able to inspect the state and result property values using the debugger tool, but we will not be able to access them directly using the program.

Able to inspect the internal properties of a promise

A promise's state can be pending, fulfilled or rejected. A promise that is either resolved or rejected is called settled.

A settled promise is either fulfilled or rejected

### How promises are resolved and rejected

Here is an example of a promise that will be resolved (fulfilled state) with the value I am done immediately.

let promise = new Promise(function(resolve, reject) {

resolve("I am done");

});

The promise below will be rejected (rejected state) with the error message Something is not right!.

let promise = new Promise(function(resolve, reject) {

reject(new Error('Something is not right!'));

});

An important point to note:

A Promise executor should call only one resolve or one reject. Once one state is changed (pending => fulfilled or pending => rejected), that's all. Any further calls to resolve or reject will be ignored.

let promise = new Promise(function(resolve, reject) {

resolve("I am surely going to get resolved!");

reject(new Error('Will this be ignored?')); // ignored

resolve("Ignored?"); // ignored

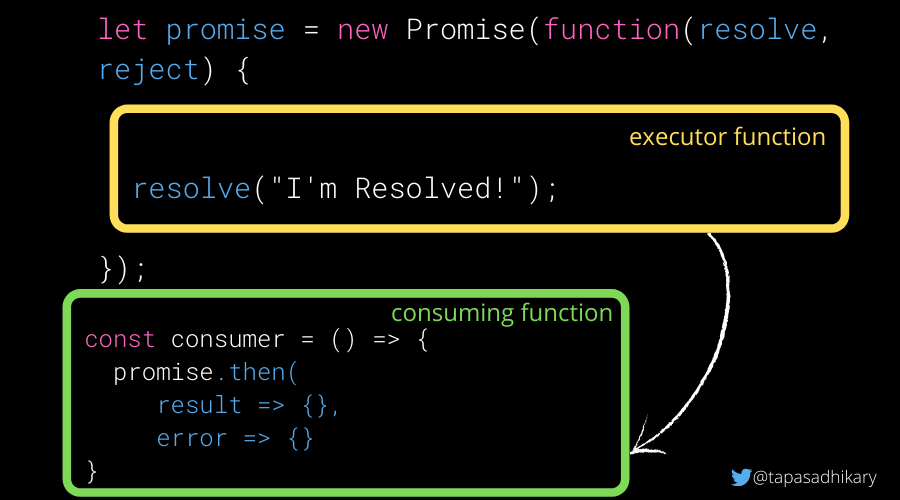
});

In the example above, only the first one to resolve will be called and the rest will be ignored.

# How to handle a Promise once you've created it

A Promise uses an executor function to complete a task (mostly asynchronously). A consumer function (that uses an outcome of the promise) should get notified when the executor function is done with either resolving (success) or rejecting (error).

The handler methods, .then(), .catch() and .finally(), help to create the link between the executor and the consumer functions so that they can be in sync when a promise resolves or rejects.

The executor and consumer functions

## How to Use the .then() Promise Handler

The .then() method should be called on the promise object to handle a result (resolve) or an error (reject).

It accepts two functions as parameters. Usually, the .then() method should be called from the consumer function where you would like to know the outcome of a promise's execution.

promise.then(

(result) => {

console.log(result);

},

(error) => {

console.log(error);

}

);

If you are interested only in successful outcomes, you can just pass one argument to it, like this:

promise.then(

(result) => {

console.log(result);

}

);

If you are interested only in the error outcome, you can pass null for the first argument, like this:

promise.then(

null,

(error) => {

console.log(error)

}

);

However, you can handle errors in a better way using the .catch() method that we will see in a minute.

Let's look at a couple of examples of handling results and errors using the .then and .catch handlers. We will make this learning a bit more fun with a few real asynchronous requests. We will use the [PokeAPI](https://pokeapi.co/) to get information about Pokémon and resolve/reject them using Promises.

First, let us create a generic function that accepts a PokeAPI URL as argument and returns a Promise. If the API call is successful, a resolved promise is returned. A rejected promise is returned for any kind of errors.

We will be using this function in several examples from now on to get a promise and work on it.

function getPromise(URL) {

let promise = new Promise(function (resolve, reject) {

let req = new XMLHttpRequest();

req.open("GET", URL);

req.onload = function () {

if (req.status == 200) {

resolve(req.response);

} else {

reject("There is an Error!");

}

};

req.send();

});

return promise;

}

Utility method to get a Promise

Example 1: Get 50 Pokémon's information:

const ALL\_POKEMONS\_URL = 'https://pokeapi.co/api/v2/pokemon?limit=50';

// We have discussed this function already!

let promise = getPromise(ALL\_POKEMONS\_URL);

const consumer = () => {

promise.then(

(result) => {

console.log({result}); // Log the result of 50 Pokemons

},

(error) => {

// As the URL is a valid one, this will not be called.

console.log('We have encountered an Error!'); // Log an error

});

}

consumer();

Example 2: Let's try an invalid URL

const POKEMONS\_BAD\_URL = 'https://pokeapi.co/api/v2/pokemon-bad/';

// This will reject as the URL is 404

let promise = getPromise(POKEMONS\_BAD\_URL);

const consumer = () => {

promise.then(

(result) => {

// The promise didn't resolve. Hence, it will

// not be executed.

console.log({result});

},

(error) => {

// A rejected prmise will execute this

console.log('We have encountered an Error!'); // Log an error

}

);

}

consumer();

## How to Use the .catch() Promise Handler

You can use this handler method to handle errors (rejections) from promises. The syntax of passing null as the first argument to the .then() is not a great way to handle errors. So we have .catch() to do the same job with some neat syntax:

// This will reject as the URL is 404

let promise = getPromise(POKEMONS\_BAD\_URL);

const consumer = () => {

promise.catch(error => console.log(error));

}

consumer();

If we throw an Error like new Error("Something wrong!")  instead of calling the reject from the promise executor and handlers, it will still be treated as a rejection. It means that this will be caught by the .catch handler method.

This is the same for any synchronous exceptions that happen in the promise executor and handler functions.

Here is an example where it will be treated like a reject and the .catch handler method will be called:

new Promise((resolve, reject) => {

throw new Error("Something is wrong!");// No reject call

}).catch((error) => console.log(error));

## How to Use the .finally() Promise Handler

The .finally() handler performs cleanups like stopping a loader, closing a live connection, and so on. The finally() method will be called irrespective of whether a promise resolves or rejects. It passes through the result or error to the next handler which can call a .then() or .catch() again.

Here is an example that'll help you understand all three methods together:

let loading = true;

loading && console.log('Loading...');

// Gatting Promise

promise = getPromise(ALL\_POKEMONS\_URL);

promise.finally(() => {

loading = false;

console.log(`Promise Settled and loading is ${loading}`);

}).then((result) => {

console.log({result});

}).catch((error) => {

console.log(error)

});

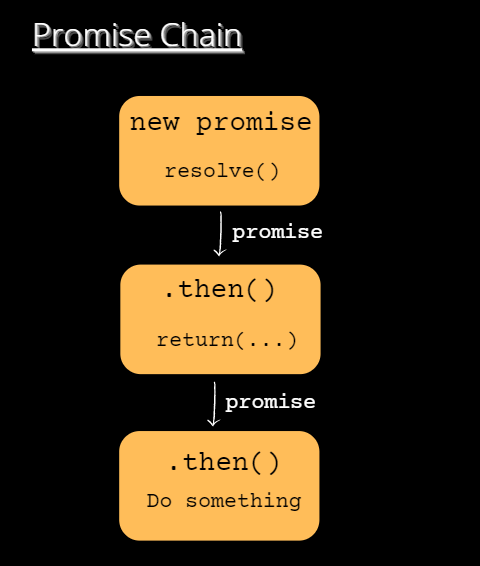
To explain a bit further:

* The .finally() method makes loading false.
* If the promise resolves, the .then() method will be called. If the promise rejects with an error, the .catch() method will be called. The .finally() will be called irrespective of the resolve or reject.

# What is the Promise Chain?

The  promise.then() call always returns a promise. This promise will have the state as pending and result as undefined. It allows us to call the next .then method on the new promise.

When the first .then method returns a value, the next .then method can receive that. The second one can now pass to the third .then() and so on. This forms a chain of .then methods to pass the promises down. This phenomenon is called the Promise Chain.

Promise Chain

Here is an example:

let promise = getPromise(ALL\_POKEMONS\_URL);

promise.then(result => {

let onePokemon = JSON.parse(result).results[0].url;

return onePokemon;

}).then(onePokemonURL => {

console.log(onePokemonURL);

}).catch(error => {

console.log('In the catch', error);

});

Here we first get a promise resolved and then extract the URL to reach the first Pokémon. We then return that value and it will be passed as a promise to the next .then() handler function. Hence the output,

https://pokeapi.co/api/v2/pokemon/1/

The .then method can return either:

* A value (we have seen this already)
* A brand new promise.

It can also throw an error.

Here is an example where we have created a promise chain with the .then methods which returns results and a new promise:

// Promise Chain with multiple then and catch

let promise = getPromise(ALL\_POKEMONS\_URL);

promise.then(result => {

let onePokemon = JSON.parse(result).results[0].url;

return onePokemon;

}).then(onePokemonURL => {

console.log(onePokemonURL);

return getPromise(onePokemonURL);

}).then(pokemon => {

console.log(JSON.parse(pokemon));

}).catch(error => {

console.log('In the catch', error);

});

In the first .then call we extract the URL and return it as a value. This URL will be passed to the second .then call where we are returning a new promise taking that URL as an argument.

This promise will be resolved and passed down to the chain where we get the information about the Pokémon. Here is the output:

Output of the promise chain call

In case there is an error or a promise rejection, the .catch method in the chain will be called.

A point to note: Calling .then multiple times doesn't form a Promise chain. You may end up doing something like this only to introduce a bug in the code:

let promise = getPromise(ALL\_POKEMONS\_URL);

promise.then(result => {

let onePokemon = JSON.parse(result).results[0].url;

return onePokemon;

});

promise.then(onePokemonURL => {

console.log(onePokemonURL);

return getPromise(onePokemonURL);

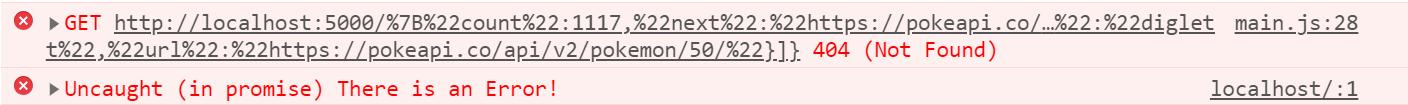
});

promise.then(pokemon => {

console.log(JSON.parse(pokemon));

});

We call the .then method three times on the same promise, but we don't pass the promise down. This is different than the promise chain. In the above example, the output will be an error.



# How to Handle Multiple Promises

Apart from the handler methods (.then, .catch, and .finally), there are six static methods available in the Promise API. The first four methods accept an array of promises and run them in parallel.

1. Promise.all
2. Promise.any
3. Promise.allSettled
4. Promise.race
5. Promise.resolve
6. Promise.reject

Let's go through each one.

## The Promise.all() method

Promise.all([promises]) accepts a collection (for example, an array) of promises as an argument and executes them in parallel.

This method waits for all the promises to resolve and returns the array of promise results. If any of the promises reject or execute to fail due to an error, all other promise results will be ignored.

Let's create three promises to get information about three Pokémons.

const BULBASAUR\_POKEMONS\_URL = 'https://pokeapi.co/api/v2/pokemon/bulbasaur';

const RATICATE\_POKEMONS\_URL = 'https://pokeapi.co/api/v2/pokemon/raticate';

const KAKUNA\_POKEMONS\_URL = 'https://pokeapi.co/api/v2/pokemon/kakuna';

let promise\_1 = getPromise(BULBASAUR\_POKEMONS\_URL);

let promise\_2 = getPromise(RATICATE\_POKEMONS\_URL);

let promise\_3 = getPromise(KAKUNA\_POKEMONS\_URL);

Use the Promise.all() method by passing an array of promises.

Promise.all([promise\_1, promise\_2, promise\_3]).then(result => {

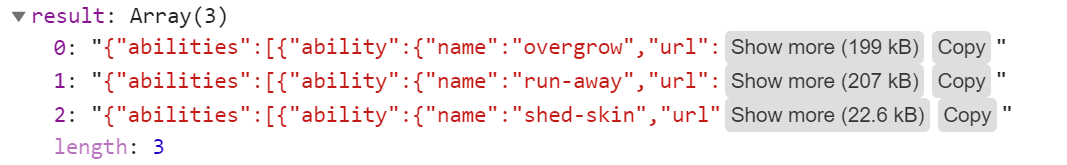
console.log({result});

}).catch(error => {

console.log('An Error Occured');

});

Output:



As you see in the output, the result of all the promises is returned. The time to execute all the promises is equal to the max time the promise takes to run.

## The Promise.any() method

Promise.any([promises]) - Similar to the all() method, .any() also accepts an array of promises to execute them in parallel. This method doesn't wait for all the promises to resolve. It is done when any one of the promises is settled.

Promise.any([promise\_1, promise\_2, promise\_3]).then(result => {

console.log(JSON.parse(result));

}).catch(error => {

console.log('An Error Occured');

});

The output would be the result of any of the resolved promises:

image-162

## The Promise.allSettled() method

romise.allSettled([promises]) - This method waits for all promises to settle(resolve/reject) and returns their results as an array of objects. The results will contain a state (fulfilled/rejected) and value, if fulfilled. In case of rejected status, it will return a reason for the error.

Here is an example of all fulfilled promises:

Promise.allSettled([promise\_1, promise\_2, promise\_3]).then(result => {

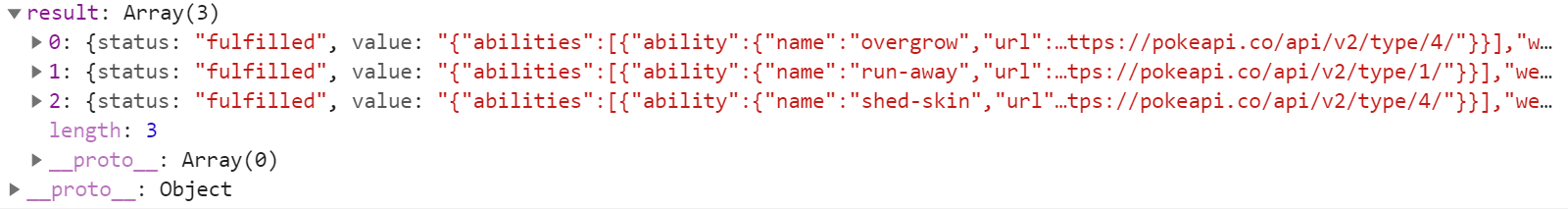
console.log({result});

}).catch(error => {

console.log('There is an Error!');

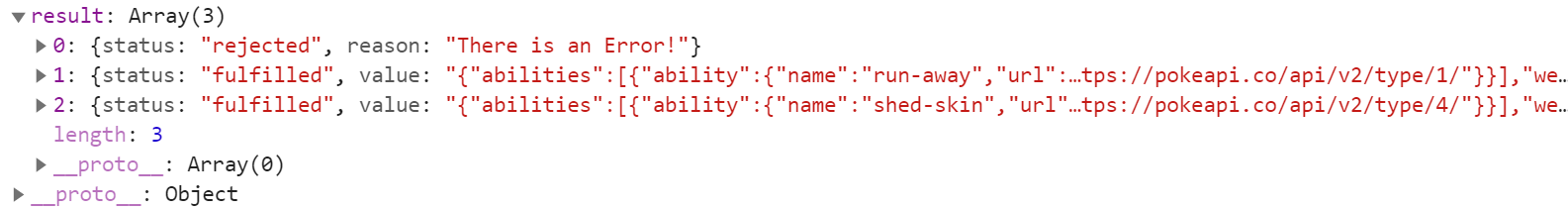
});

Output:



If any of the promises rejects, say, the promise\_1,

let promise\_1 = getPromise(POKEMONS\_BAD\_URL);



## The Promise.race() method

Promise.race([promises]) – It waits for the first (quickest) promise to settle, and returns the result/error accordingly.

Promise.race([promise\_1, promise\_2, promise\_3]).then(result => {

console.log(JSON.parse(result));

}).catch(error => {

console.log('An Error Occured');

});

Output the fastest promise that got resolved:

image-165

## The Promise.resolve/reject methods

Promise.resolve(value) – It resolves a promise with the value passed to it. It is the same as the following:

let promise = new Promise(resolve => resolve(value));

Promise.reject(error) – It rejects a promise with the error passed to it. It is the same as the following:

let promise = new Promise((resolve, reject) => reject(error));