# **Promise**

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

Syntax

new Promise(executor);

### **Parameters**

**executor**

A function that is passed with the arguments resolve and reject. The executor function is executed immediately by the Promise implementation, passing resolve and reject functions (the executor is called before the Promise constructor even returns the created object). The resolve and reject functions, when called, resolve or reject the promise, respectively. The executor normally initiates some asynchronous work, and then, once that completes, either calls the resolve function to resolve the promise or else rejects it if an error occurred. If an error is thrown in the executor function, the promise is rejected. The return value of the executor is ignored.

**Description**

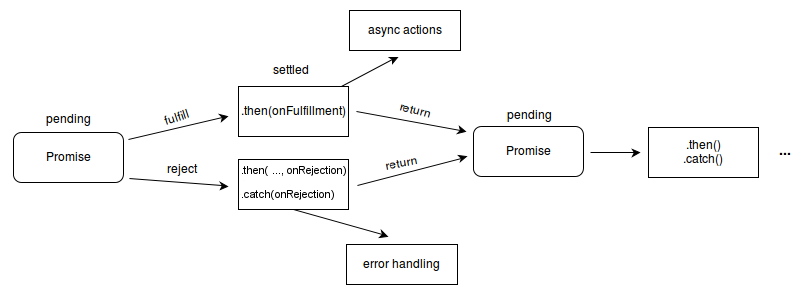
A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

A Promise is in one of these states:

* pending: initial state, neither fulfilled nor 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 associated handlers queued up by a promise's then method are called. (If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.)

As the [Promise.prototype.then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) and [Promise.prototype.catch()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch) methods return promises, they can be chained.



## **Properties**

**Promise.length**

Length property whose value is always 1 (number of constructor arguments).

[**Promise.prototype**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/prototype)

Represents the prototype for the Promise constructor.

## **Methods**

# Promise.all()

The **Promise.all()** method returns a single [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) that fulfills when all of the promises passed as an iterable have been fulfilled or when the iterable contains no promises. It rejects with the reason of the first promise that rejects.

It is typically used after having started multiple asynchronous tasks to run concurrently and having created promises for their results, so that one can wait for all the tasks being finished.

**JavaScript Demo: Promise.all()**

1

var promise1 = Promise.resolve(3);

2

var promise2 = 42;

3

var promise3 = new Promise(function(resolve, reject) {

4

setTimeout(resolve, 20000, 'foo');

5

});

6

​

7

Promise.all([promise1, promise2, promise3]).then(function(values) {

8

console.log(values);

9

});

10

// expected output: Array [3, 42, "foo"]

11

## ​Syntax

Promise.all(iterable);

### **Parameters**

**iterable**

An [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols" \l "The_iterable_protocol) object such as an [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array).

### **Return value**

* An **already resolved** [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) if the iterable passed is empty.
* An **asynchronously resolved** [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) if the iterable passed contains no promises. Note, Google Chrome 58 returns an **already resolved** promise in this case.
* A **pending** [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) in all other cases. This returned promise is then resolved/rejected **asynchronously** (as soon as the stack is empty) when all the promises in the given iterable have resolved, or if any of the promises reject. See the example about "Asynchronicity or synchronicity of Promise.all" below. Returned values will be in order of the Promises passed, regardless of completion order.

## **Description**

This method can be useful for aggregating the results of multiple promises.

### **Fulfillment**

The returned promise is fulfilled with an array containing **all**the values of the iterable passed as argument (also non-promise values).

* If an empty iterable is passed, then this method returns (synchronously) an already resolved promise.
* If all of the passed-in promises fulfill, or are not promises, the promise returned by Promise.all is fulfilled asynchronously.

### **Rejection**

If any of the passed-in promises reject, Promise.all asynchronously rejects with the value of the promise that rejected, whether or not the other promises have resolved.

## **Examples**

### **Using Promise.all**

Promise.all waits for all fulfillments (or the first rejection).

If the *iterable* contains non-promise values, they will be ignored, but still counted in the returned promise array value (if the promise is fulfilled):

// this will be counted as if the iterable passed is empty, so it gets fulfilled

var p = Promise.all([1,2,3]);

// this will be counted as if the iterable passed contains only the resolved promise with value "444", so it gets fulfilled

var p2 = Promise.all([1,2,3, Promise.resolve(444)]);

// this will be counted as if the iterable passed contains only the rejected promise with value "555", so it gets rejected

var p3 = Promise.all([1,2,3, Promise.reject(555)]);

// using setTimeout we can execute code after the stack is empty

setTimeout(function() {

console.log(p);

console.log(p2);

console.log(p3);

});

// logs

// Promise { <state>: "fulfilled", <value>: Array[3] }

// Promise { <state>: "fulfilled", <value>: Array[4] }

// Promise { <state>: "rejected", <reason>: 555 }

# [Note:-](https://stackoverflow.com/questions/28916710/what-do-double-brackets-mean-in-javascript-and-how-to-access-them)

# What do the double brackets [[ ]] mean, and how do I retrieve the value of [[PromiseValue]]?

Ans:

Promise {

[[PromiseStatus]]: "resolved",

[[PromiseValue]]: child

}

It's an internal property. You cannot access it directly. Native promises may only be unwrapped in then with promises or asynchronously in generally.

Quoting the specification:

They are defined by this specification purely for expository purposes. An implementation of ECMAScript must behave as if it produced and operated upon internal properties in the manner described here. **The names of internal properties are enclosed in double square brackets [[ ]]**. When an algorithm uses an internal property of an object and the object does not implement the indicated internal property, a TypeError exception is thrown.

## **Seriously though - what are they?**

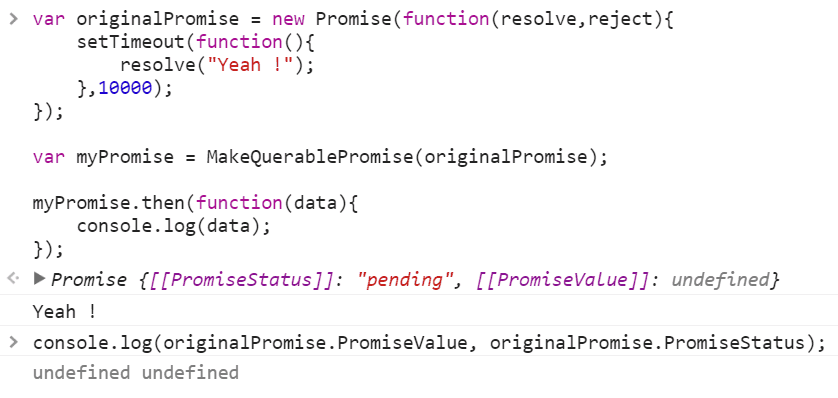
Very nice! As the above quote says they're just used in the spec - so there is no reason for them to really appear in your console.

Don't tell anyone but these are really **private symbols**. The reason they exist is for other internal methods to be able to access [[PromiseValue]]. For example when io.js decides to return promises instead of taking callbacks - these would allow it to access these properties fast in cases it is guaranteed. They are not exposed to the outside.

### **Can I access them?**

Not unless you make your own Chrome or V8 build. Maybe in ES7 with access modifiers. As of right now, there is no way as they are not a part of the specification and will break across browsers - sorry.

# How to check if a Javascript promise has been fulfilled, rejected or resolved



## Workaround

function MakeQuerablePromise(promise) {

// Don't modify any promise that has been already modified.

if (promise.isResolved) return promise;//isResolved is not a property of promise but we can attach this property and change it inside this function.

// Set initial state

var isPending = true;

var isRejected = false;

var isFulfilled = false;

// Observe the promise, saving the fulfillment in a closure scope.

var result = promise.then(

function(v) {

isFulfilled = true;

isPending = false;

promise.isResolved=true;

return v;

},

function(e) {

isRejected = true;

isPending = false;

throw e;

}

);

result.isFulfilled = function() { return isFulfilled; };

result.isPending = function() { return isPending; };

result.isRejected = function() { return isRejected; };

return result;

}

Asynchronicity or synchronicity of Promise.all

This following example demonstrates the asynchronicity (or synchronicity, if the iterable passed is empty) of Promise.all:

Promise.all resolves synchronously **if and only if** the *iterable* passed is empty:

var p = Promise.all([]); // will be immediately resolved

var p2 = Promise.all([1337, "hi"]); // non-promise values will be ignored, but the evaluation will be done asynchronously

console.log(p);

console.log(p2)

setTimeout(function() {

console.log('the stack is now empty');

console.log(p2);

});

// logs

// Promise { <state>: "fulfilled", <value>: Array[0] }

// Promise { <state>: "pending" }

// the stack is now empty

// Promise { <state>: "fulfilled", <value>: Array[2] }

Promise.all fail-fast behavior

Promise.all is rejected if any of the elements are rejected. For example, if you pass in four promises that resolve after a timeout and one promise that rejects immediately, then Promise.all will reject immediately.

It is possible to change this behaviour by handling possible rejections:

var p1 = new Promise((resolve, reject) => {

setTimeout(() => resolve('p1\_delayed\_resolution'), 1000);

});

var p2 = new Promise((resolve, reject) => {

reject(new Error('p2\_immediate\_rejection'));

});

Promise.all([

p1.catch(error => { return error }),

p2.catch(error => { return error }),

]).then(values => {

console.log(values[0]) // "p1\_delayed\_resolution"

console.error(values[1]) // "Error: p2\_immediate\_rejection"

})

# Promise.allSettled()

The **Promise.allSettled()** method returns a promise that resolves after all of the given promises have either resolved or rejected, with an array of objects that each describes the outcome of each promise.

**JavaScript Demo: Promise.allSettled()**

1

const promise1 = Promise.resolve(1)

2

const promise2 = new Promise((resolve, reject) => setTimeout(reject, 1000,'fo'));

3

const promises = [promise1, promise2];

4

​

5

Promise.allSettled(promises).

6

then((results) => results.forEach((result) => console.log(result)));

7

​

8

// expected output:

9

// "fulfilled"

10

// "rejected"

11

## Syntax

Promise.allSettled(iterable);

### **Parameters**

**iterable**

An [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/iterable) object, such as an [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array), in which each member is a Promise.

### **Return value**

A **pending** [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) that will be **asynchronously** fulfilled once every promise in the specified collection of promises has completed, either by successfully being fulfilled or by being rejected. At that time, the returned promise's handler is passed as input an array containing the outcome of each promise in the original set of promises.

For each outcome object, a status string is present. If the status is fulfilled, then a value is present. If the status is rejected, then a reason is present. The value (or reason) reflects what value each promise was fulfilled (or rejected) with.

# Promise.race()

The **Promise.race()** method returns a promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or reason from that promise.

**JavaScript Demo: Promise.race()**

1

var promise1 = new Promise(function(resolve, reject) {

2

setTimeout(resolve, 500, 'one');

3

});

4

​

5

var promise2 = new Promise(function(resolve, reject) {

6

setTimeout(resolve, 100, 'two');

7

});

8

​

9

Promise.race([promise1, promise2]).then(function(value) {

10

console.log(value);

11

// Both resolve, but promise2 is faster

12

});

13

// expected output: "two"

14

## ​ Syntax

Promise.race(iterable);

### **Parameters**

**iterable**

An iterable object, such as an [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array). See [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/iterable).

### **Return value**

A **pending** [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) that **asynchronously**yields the value of the first promise in the given iterable to fulfill or reject.

## Description

The race function returns a Promise that is settled the same way (and takes the same value) as the first promise that settles amongst the promises of the iterable passed as an argument.

If the iterable passed is empty, the promise returned will be forever pending.

If the iterable contains one or more non-promise value and/or an already settled promise, then Promise.race will resolve to the first of these values found in the iterable.

### **Asynchronicity of Promise.race**

This following example demonstrates the asynchronicity of Promise.race:

// we are passing as argument an array of promises that are already resolved,

// to trigger Promise.race as soon as possible

var resolvedPromisesArray = [Promise.resolve(33), Promise.resolve(44)];

var p = Promise.race(resolvedPromisesArray);

// immediately logging the value of p

console.log(p);

// using setTimeout we can execute code after the stack is empty

setTimeout(function(){

console.log('the stack is now empty');

console.log(p);

});

// logs, in order:

// Promise { <state>: "pending" }

// the stack is now empty

// Promise { <state>: "fulfilled", <value>: 33 }

If the iterable contains one or more non-promise value and/or an already settled promise, then Promise.race will resolve to the first of these values found in the array:

var foreverPendingPromise = Promise.race([]);

var alreadyFulfilledProm = Promise.resolve(666);

var arr = [foreverPendingPromise, alreadyFulfilledProm, "non-Promise value"];

var arr2 = [foreverPendingPromise, "non-Promise value", Promise.resolve(666)];

var p = Promise.race(arr);

var p2 = Promise.race(arr2);

console.log(p);

console.log(p2);

setTimeout(function(){

console.log('the stack is now empty');

console.log(p);

console.log(p2);

});

// logs, in order:

// Promise { <state>: "pending" }

// Promise { <state>: "pending" }

// the stack is now empty

// Promise { <state>: "fulfilled", <value>: 666 }

// Promise { <state>: "fulfilled", <value>: "non-Promise value" }

# **Promise.reject()**

The **Promise.reject()** method returns a Promise object that is rejected with a given reason.

**Promise.reject()**

1

function resolved(result) {

2

console.log('Resolved');

3

}

4

​

5

function rejected(result) {

6

console.error(result);

7

}

8

​

9

Promise.reject(new Error('fail')).then(resolved, rejected);

10

// expected output: Error: fail

11

## ​ Syntax

Promise.reject(reason);

### **Parameters**

**reason**

Reason why this Promise rejected.

### **Return value**

A [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) that is rejected with the given reason.

## Description

The static Promise.reject function returns a Promise that is rejected. For debugging purposes and selective error catching, it is useful to make reason an instanceof [Error](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error).

# **Promise.resolve()**

The **Promise.resolve()** method returns a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) object that is resolved with a given value. If the value is a promise, that promise is returned; if the value is a thenable (i.e. has a ["then" method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then)), the returned promise will "follow" that thenable, adopting its eventual state; otherwise the returned promise will be fulfilled with the value. This function flattens nested layers of promise-like objects (e.g. a promise that resolves to a promise that resolves to something) into a single layer.

The **Promise.resolve()** method returns a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) object that is resolved with a given value. If the value is a promise, that promise is returned; if the value is a thenable (i.e. has a ["then" method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then)), the returned promise will "follow" that thenable, adopting its eventual state; otherwise the returned promise will be fulfilled with the value. This function flattens nested layers of promise-like objects (e.g. a promise that resolves to a promise that resolves to something) into a single layer.

## Syntax

Promise.resolve(value);

### **Parameters**

**value**

Argument to be resolved by this Promise. Can also be a Promise or a thenable to resolve.

### **Return value**

A [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) that is resolved with the given value, or the promise passed as value, if the value was a promise object.

## Description

The static Promise.resolve function returns a Promise that is resolved.

Example:

### **Resolving another Promise**

var original = Promise.resolve(33);

var cast = Promise.resolve(original);

cast.then(function(value) {

console.log('value: ' + value);

});

console.log('original === cast ? ' + (original === cast));

// logs, in order:

// original === cast ? true

// value: 33

The inverted order of the logs is due to the fact that the then handlers are called asynchronously. See how then works [here](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then#Return_value).

### **Resolving thenables and throwing Errors**

// Resolving a thenable object

var p1 = Promise.resolve({

then: function(onFulfill, onReject) { onFulfill('fulfilled!'); }

});

console.log(p1 instanceof Promise) // true, object casted to a Promise

p1.then(function(v) {

console.log(v); // "fulfilled!"

}, function(e) {

// not called

});

// Thenable throws before callback

// Promise rejects

var thenable = { then: function(resolve) {

throw new TypeError('Throwing');

resolve('Resolving');

}};

var p2 = Promise.resolve(thenable);

p2.then(function(v) {

// not called

}, function(e) {

console.error(e); // TypeError: Throwing

});

// Thenable throws after callback

// Promise resolves

var thenable = { then: function(resolve) {

resolve('Resolving');

throw new TypeError('Throwing');

}};

var p3 = Promise.resolve(thenable);

p3.then(function(v) {

console.log(v); // "Resolving"

}, function(e) {

// not called

});

## Differences

Let's first take a look at difference between existing & new combinator methods.

### 🔅 Promise.all vs. Promise.allSettled

Both accepts an iterable object but

* Promise.all rejects as soon as a promise within the iterable object rejected.
* Promise.allSettled resolves regardless of rejected promise(s) within the iterable object.

### 🔅 Promise.race vs. Promise.any

Both accepts an iterable object but

* Promise.race short-circuits on the first settled (fulfilled or rejected) promise within the iterable object.
* Promise.any short-circuits on the first fulfilled promise and continues to resolve regardless of rejected promises unless all within the iterable object reject.

## Comparison Table

Now let's take a look at existing/upcoming combinator methods.

|  | **Short-circuit?** | **Short-circuits on?** | **Fulfilled on?** | **Rejected on?** |
| --- | --- | --- | --- | --- |
| Promise.all | ✅ | First rejected promise | All promise fulfilled | First rejected promise |
| Promise.allSettled | ❌ | N/A | Always | N/A |
| Promise.race | ✅ | First settled | First promise fulfilled | First rejected promise |
| Promise.any | ✅ | First fulfilled | First promise fulfilled | All rejected promises |