Promise

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

This entry is for the Promise constructor. To learn about promises, read Using promises first. The constructor is primarily used to wrap functions that do not already support promises.

Syntax

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

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

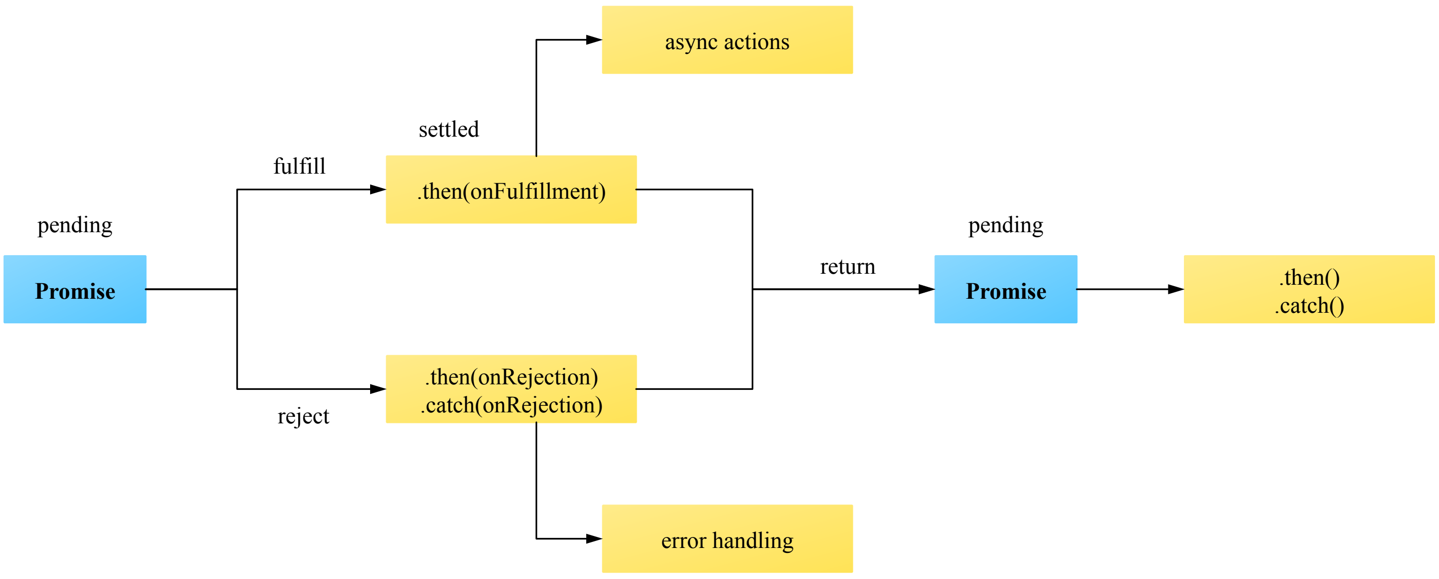
A **`Promise`** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning 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()` and `Promise.prototype.catch()` methods return promises, they can be chained.



Not to be confused with: Several other languages have mechanisms for lazy evaluation and deferring a computation, which they also call "promises", e.g. Scheme. Promises in JavaScript represent processes which are already happening, which can be chained with callback functions. If you are looking to lazily evaluate an expression, consider the arrow function with no arguments: `f = () => expression` to create the lazily-evaluated expression, and `f()` to evaluate.

Note: A promise is said to be *settled* if it is either fulfilled or rejected, but not pending. You will also hear the term *resolved* used with promises — this means that the promise is fulfilled. States and fates contains more details about promise terminology.

Properties

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

`Promise.prototype`
: Represents the prototype for the `Promise` constructor.

Methods

`Promise.all(iterable)`
: Returns a promise that either fulfills when all of the promises in the iterable argument have fulfilled or rejects as soon as one of the promises in the iterable argument rejects. If the returned promise fulfills, it is fulfilled with an array of the values from the fulfilled promises in the same order as defined in the iterable. If the returned promise rejects, it is rejected with the reason from the first promise in the iterable that rejected. This method can be useful for aggregating results of multiple promises.

`Promise.race(iterable)`
: Returns a promise that fulfills or rejects as soon as one of the promises in the iterable fulfills or rejects, with the value or reason from that promise.

`Promise.reject(reason)`
: Returns a `Promise` object that is rejected with the given reason.

`Promise.resolve(value)`
: Returns a `Promise` object that is resolved with the given value. If the value is a thenable (i.e. has a `then` method), the returned promise will "follow" that thenable, adopting its eventual state; otherwise the returned promise will be fulfilled with the value. Generally, if you don't know if a value is a promise or not, `Promise.resolve(value)` it instead and work with the return value as a promise.

`Promise` prototype

Properties

`Promise.prototype.constructor`
: Returns the function that created an instance's prototype. This is the `Promise` function by default.

Methods

`Promise.prototype.catch(onRejected)`
: Appends a rejection handler callback to the promise, and returns a new promise resolving to the return value of the callback if it is called, or to its original fulfillment value if the promise is instead fulfilled.

`Promise.prototype.then(onFulfilled, onRejected)`

Appends fulfillment and rejection handlers to the promise, and returns a new promise resolving to the return value of the called handler, or to its original settled value if the promise was not handled (i.e. if the relevant handler `onFulfilled` or `onRejected` is not a function).

Creating a Promise

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 myFirstPromise = new Promise((resolve, reject) => {
  // do something asynchronous which eventually calls either:
  //
  //   resolve(someValue); // fulfilled
  // or
  //   reject("failure reason"); // rejected
});
```

To provide a function with promise functionality, simply have it return a promise:

```
function myAsyncFunction(url) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.onload = () => resolve(xhr.responseText);
    xhr.onerror = () => reject(xhr.statusText);
    xhr.send();
  });
}
```

Examples

Basic Example

```
let myFirstPromise = new Promise((resolve, reject) => {
  // We call resolve(...) when what we were doing asynchronously was succe
  // In this example, we use setTimeout(...) to simulate async code.
  // In reality, you will probably be using something like XHR or an HTML5
  setTimeout(function(){
    resolve("Success!"); // Yay! Everything went well!
  }, 250);
});

myFirstPromise.then((successMessage) => {
  // successMessage is whatever we passed in the resolve(...) function abo
  // It doesn't have to be a string, but if it is only a succeed message,
  console.log("Yay! " + successMessage);
});
```

Advanced Example

This small example shows the mechanism of a `Promise`. The `testPromise()` method is called each time the `<button>` is clicked. It creates a promise that will be fulfilled, using `window.setTimeout()`, to the promise count (number starting from 1) every 1-3 seconds, at random. The `Promise()` constructor is used to create the promise.

The fulfillment of the promise is simply logged, via a fulfill callback set using `p1.then()`. A few logs show how the synchronous part of the method is decoupled from the asynchronous completion of the promise.

```
'use strict';
var promiseCount = 0;

function testPromise() {
    let thisPromiseCount = ++promiseCount;

    let log = document.getElementById('log');
    log.insertAdjacentHTML('beforeend', thisPromiseCount +
        ') Started (<small>Sync code started</small>)<br/>');

    // We make a new promise: we promise a numeric count of this promise,
    let p1 = new Promise(
        // The resolver function is called with the ability to resolve or
        // reject the promise
       (resolve, reject) => {
            log.insertAdjacentHTML('beforeend', thisPromiseCount +
                ') Promise started (<small>Async code started</small>)<br/
            // This is only an example to create asynchronism
            window.setTimeout(
                function() {
                    // We fulfill the promise !
                    resolve(thisPromiseCount);
                }, Math.random() * 2000 + 1000);
        }
    );

    // We define what to do when the promise is resolved with the then() c
    // and what to do when the promise is rejected with the catch() call
    p1.then(
        // Log the fulfillment value
        function(val) {
            log.insertAdjacentHTML('beforeend', val +
                ') Promise fulfilled (<small>Async code terminated</small>
        })
    .catch(
        // Log the rejection reason
       (reason) => {
            console.log('Handle rejected promise ('+reason+') here.');
        });

    log.insertAdjacentHTML('beforeend', thisPromiseCount +
        ') Promise made (<small>Sync code terminated</small>)<br/>');
}
```

This example is started by clicking the button. You need a browser that supports `Promise`. By clicking the button several times in a short amount of time, you'll even see the different promises being fulfilled one after another.

Make a promise!

Loading an image with XHR

Another simple example using `Promise` and `XMLHttpRequest` to load an image is available at the MDN GitHub js-examples repository. You can also see it in action. Each step is commented and allows you to follow the Promise and XHR architecture closely.

Specifications

Specification	Status	Comment
ECMAScript 2015 (6th Edition, ECMA-262) The definition of 'Promise' in that specification.	ST Standard	Initial definition in an ECMA standard.
ECMAScript Latest Draft (ECMA-262) The definition of 'Promise' in that specification.	LS Living Standard	

Browser compatibility

Desktop Mobile

Feature	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari
Basic Support	32	(Yes)	29[1]	No	19	7.1[3]

1. Constructor requires a new operator since version 37.

2. Constructor requires a new operator since version 4.

3. Constructor requires a new operator since version 10.

See also

- Using promises
- Promises/A+ specification
- Venkatraman.R - JS Promise (Part 1, Basics)
- Venkatraman.R - JS Promise (Part 2 - Using Q.js, When.js and RSVP.js)
- Jake Archibald: JavaScript Promises: There and Back Again
- Domenic Denicola: Callbacks, Promises, and Coroutines – Asynchronous Programming Patterns in JavaScript

- Matt Greer: JavaScript Promises ... In Wicked Detail
- Forbes Lindesay: promisejs.org
- Nolan Lawson: We have a problem with promises — Common mistakes with promises
- Promise polyfill
- Udacity: JavaScript Promises