Chapter 42: Promises

Section 42.1: Introduction

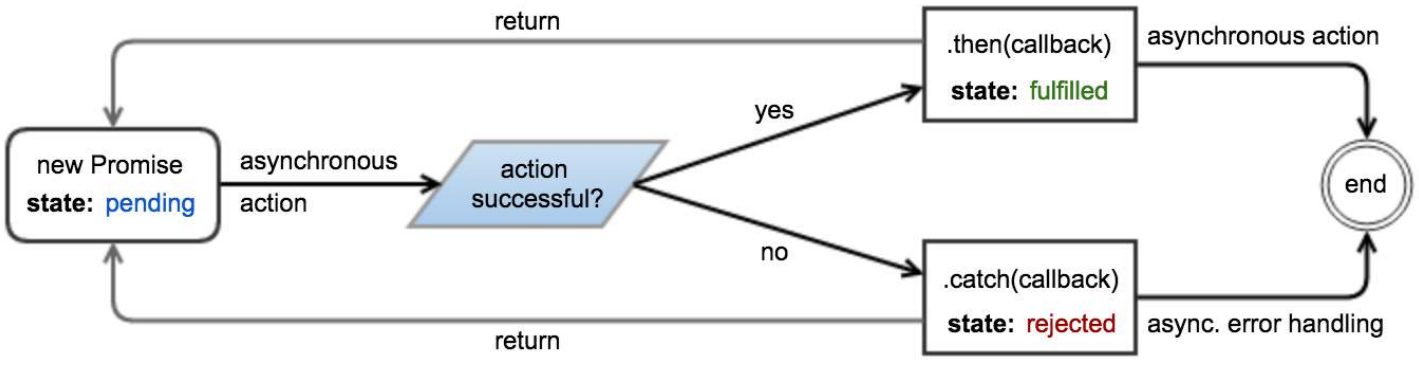
A `Promise` object represents an operation which *has produced or will eventually produce* a value. Promises provide a robust way to wrap the (possibly pending) result of asynchronous work, mitigating the problem of deeply nested callbacks (known as "callback hell").

States and control flow

A promise can be in one of three states:

- *pending* — The underlying operation has not yet completed, and the promise is *pending* fulfillment.
- *fulfilled* — The operation has finished, and the promise is *fulfilled* with a *value*. This is analogous to returning a value from a synchronous function.
- *rejected* — An error has occurred during the operation, and the promise is *rejected* with a *reason*. This is analogous to throwing an error in a synchronous function.

A promise is said to be *settled* (or *resolved*) when it is either fulfilled or rejected. Once a promise is settled, it becomes immutable, and its state cannot change. The `then` and **`catch`** methods of a promise can be used to attach callbacks that execute when it is settled. These callbacks are invoked with the fulfillment value and rejection reason, respectively.



Example

```
const promise = new Promise((resolve, reject) => {
    // Perform some work (possibly asynchronous)
    // ...

    if (/* Work has successfully finished and produced "value" */) {
        resolve(value);
    } else {
        // Something went wrong because of "reason"
        // The reason is traditionally an Error object, although
        // this is not required or enforced.
        let reason = new Error(message);
        reject(reason);

        // Throwing an error also rejects the promise.
        throw reason;
    }
});
```

The `then` and **`catch`** methods can be used to attach fulfillment and rejection callbacks:

```
promise.then(value => {
    // Work has completed successfully,
    // promise has been fulfilled with "value"
}).catch(reason => {
    // Something went wrong,
    // promise has been rejected with "reason"
});
```

Note: Calling `promise.then(...)` and `promise.catch(...)` on the same promise might result in an `Uncaught exception in Promise` if an error occurs, either while executing the promise or inside one of the callbacks, so the preferred way would be to attach the next listener on the promise returned by the previous `then` / `catch`.

Alternatively, both callbacks can be attached in a single call to `then`:

```
promise.then(onFulfilled, onRejected);
```

Attaching callbacks to a promise that has already been settled will immediately place them in the microtask queue, and they will be invoked "as soon as possible" (i.e. immediately after the currently executing script). It is not necessary to check the state of the promise before attaching callbacks, unlike with many other event-emitting implementations.

Live demo

Section 42.2: Promise chaining

The `then` method of a promise returns a new promise.

```
const promise = new Promise(resolve => setTimeout(resolve, 5000));

promise
    // 5 seconds later
    .then(() => 2)
    // returning a value from a then callback will cause
    // the new promise to resolve with this value
    .then(value => { /* value === 2 */ });
```

Returning a `Promise` from a `then` callback will append it to the promise chain.

```
function wait(millis) {
    return new Promise(resolve => setTimeout(resolve, millis));
}

const p = wait(5000).then(() => wait(4000)).then(() => wait(1000));
p.then(() => { /* 10 seconds have passed */ });
```

A `catch` allows a rejected promise to recover, similar to how `catch` in a `try`/`catch` statement works. Any chained `then` after a `catch` will execute its resolve handler using the value resolved from the `catch`.

```
const p = new Promise(resolve => {throw 'oh no'});
p.catch(() => 'oh yes').then(console.log.bind(console));  // outputs "oh yes"
```

If there are no `catch` or `reject` handlers in the middle of the chain, a `catch` at the end will capture any rejection in the chain:

```
p.catch(() => Promise.reject('oh yes'))
  .then(console.log.bind(console))      // won't be called
  .catch(console.error.bind(console));  // outputs "oh yes"
```

On certain occasions, you may want to "branch" the execution of the functions. You can do it by returning different promises from a function depending on the condition. Later in the code, you can merge all of these branches into one to call other functions on them and/or to handle all errors in one place.

```
promise
    .then(result => {
        if (result.condition) {
            return handlerFn1()
                .then(handlerFn2);
        } else if (result.condition2) {
            return handlerFn3()
                .then(handlerFn4);
        } else {
            throw new Error("Invalid result");
        }
    })
    .then(handlerFn5)
    .catch(err => {
        console.error(err);
    });
```

Thus, the execution order of the functions looks like:

```
promise --> handlerFn1 -> handlerFn2 --> handlerFn5 ~~> .catch()
         |                            ^
         V                            |
         -> handlerFn3 -> handlerFn4 -^
```

The single `catch` will get the error on whichever branch it may occur.

Section 42.3: Waiting for multiple concurrent promises

The `Promise.all()` static method accepts an iterable (e.g. an `Array`) of promises and returns a new promise, which resolves when **all** promises in the iterable have resolved, or rejects if **at least one** of the promises in the iterable have rejected.

```
// wait "millis" ms, then resolve with "value"
function resolve(value, milliseconds) {
    return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "millis" ms, then reject with "reason"
function reject(reason, milliseconds) {
    return new Promise((_, reject) => setTimeout(() => reject(reason), milliseconds));
}

Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    resolve(3, 7000)
]).then(values => console.log(values)); // outputs "[1, 2, 3]" after 7 seconds.

Promise.all([
    resolve(1, 5000),
    reject('Error!', 6000),
    resolve(2, 7000)
]).then(values => console.log(values)) // does not output anything
```

```
.catch(reason => console.log(reason)); // outputs "Error!" after 6 seconds.
```

Non-promise values in the iterable are "promisified".

```
Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    { hello: 3 }
])
.then(values => console.log(values)); // outputs "[1, 2, { hello: 3 }]" after 6 seconds
```

Destructuring assignment can help to retrieve results from multiple promises.

```
Promise.all([
    resolve(1, 5000),
    resolve(2, 6000),
    resolve(3, 7000)
])
.then(([result1, result2, result3]) => {
    console.log(result1);
    console.log(result2);
    console.log(result3);
});
```

Section 42.4: Reduce an array to chained promises

This design pattern is useful for generating a sequence of asynchronous actions from a list of elements.

There are two variants :

- the "then" reduction, which builds a chain that continues as long as the chain experiences success.
- the "catch" reduction, which builds a chain that continues as long as the chain experiences error.

The "then" reduction

This variant of the pattern builds a `.then()` chain, and might be used for chaining animations, or making a sequence of dependent HTTP requests.

```
[1, 3, 5, 7, 9].reduce((seq, n) => {
    return seq.then(() => {
        console.log(n);
        return new Promise(res => setTimeout(res, 1000));
    });
}, Promise.resolve()).then(
    () => console.log('done'),
    (e) => console.log(e)
);
// will log 1, 3, 5, 7, 9, 'done' in 1s intervals
```

Explanation:

1. We call `.reduce()` on a source array, and provide `Promise.resolve()` as an initial value.
2. Every element reduced will add a `.then()` to the initial value.
3. `reduce()`'s product will be Promise.resolve().then(...).then(...).
4. We manually append a `.then(successHandler, errorHandler)` after the reduce, to execute `successHandler` once all the previous steps have resolved. If any step was to fail, then `errorHandler` would execute.

Note: The "then" reduction is a sequential counterpart of `Promise.all()`.

The "catch" reduction

This variant of the pattern builds a `.catch()` chain and might be used for sequentially probing a set of web servers for some mirrored resource until a working server is found.

```
var working_resource = 5; // one of the values from the source array
[1, 3, 5, 7, 9].reduce((seq, n) => {
    return seq.catch(() => {
        console.log(n);
        if(n === working_resource) { // 5 is working
            return new Promise((resolve, reject) => setTimeout(() => resolve(n), 1000));
        } else { // all other values are not working
            return new Promise((resolve, reject) => setTimeout(reject, 1000));
        }
    });
}, Promise.reject()).then(
    (n) => console.log('success at: ' + n),
    () => console.log('total failure')
);
// will log 1, 3, 5, 'success at 5' at 1s intervals
```

Explanation:

1. We call `.reduce()` on a source array, and provide `Promise.reject()` as an initial value.
2. Every element reduced will add a `.catch()` to the initial value.
3. `reduce()`'s product will be `Promise.reject().catch(...).catch(...)`.
4. We manually append `.then(successHandler, errorHandler)` after the reduce, to execute `successHandler` once any of the previous steps has resolved. If all steps were to fail, then `errorHandler` would execute.

Note: The "catch" reduction is a sequential counterpart of `Promise.any()` (as implemented in `bluebird.js`, but not currently in native ECMAScript).

Section 42.5: Waiting for the first of multiple concurrent promises

The `Promise.race()` static method accepts an iterable of Promises and returns a new Promise which resolves or rejects as soon as the **first** of the promises in the iterable has resolved or rejected.

```
// wait "milliseconds" milliseconds, then resolve with "value"
function resolve(value, milliseconds) {
    return new Promise(resolve => setTimeout(() => resolve(value), milliseconds));
}

// wait "milliseconds" milliseconds, then reject with "reason"
function reject(reason, milliseconds) {
    return new Promise((_, reject) => setTimeout(() => reject(reason), milliseconds));
}

Promise.race([
    resolve(1, 5000),
    resolve(2, 3000),
    resolve(3, 1000)
])
.then(value => console.log(value)); // outputs "3" after 1 second.
```

```
Promise.race([
    reject(new Error('bad things!'), 1000),
    resolve(2, 2000)
])
.then(value => console.log(value)) // does not output anything
.catch(error => console.log(error.message)); // outputs "bad things!" after 1 second
```

Section 42.6: "Promisifying" functions with callbacks

Given a function that accepts a Node-style callback,

```
fooFn(options, function callback(err, result) { ... });
```

you can promisify it *(convert it to a promise-based function)* like this:

```
function promiseFooFn(options) {
    return new Promise((resolve, reject) =>
        fooFn(options, (err, result) =>
            // If there's an error, reject; otherwise resolve
            err ? reject(err) : resolve(result)
        )
    );
}
```

This function can then be used as follows:

```
promiseFooFn(options).then(result => {
    // success!
}).catch(err => {
    // error!
});
```

In a more generic way, here's how to promisify any given callback-style function:

```
function promisify(func) {
    return function(...args) {
        return new Promise((resolve, reject) => {
            func(...args, (err, result) => err ? reject(err) : resolve(result));
        });
    }
}
```

This can be used like this:

```
const fs = require('fs');
const promisedStat = promisify(fs.stat.bind(fs));

promisedStat('/foo/bar')
    .then(stat => console.log('STATE', stat))
    .catch(err => console.log('ERROR', err));
```

Section 42.7: Error Handling

Errors thrown from promises are handled by the second parameter (`reject`) passed to `then` or by the handler passed to **`catch`**:

```
throwErrorAsync()
```

```
  .then(null, error => { /* handle error here */ });
// or
throwErrorAsync()
  .catch(error => { /* handle error here */ });
```

Chaining

If you have a promise chain then an error will cause `resolve` handlers to be skipped:

```
throwErrorAsync()
  .then(() => { /* never called */ })
  .catch(error => { /* handle error here */ });
```

The same applies to your `then` functions. If a `resolve` handler throws an exception then the next `reject` handler will be invoked:

```
doSomethingAsync()
  .then(result => { throwErrorSync(); })
  .then(() => { /* never called */ })
  .catch(error => { /* handle error from throwErrorSync() */ });
```

An error handler returns a new promise, allowing you to continue a promise chain. The promise returned by the error handler is resolved with the value returned by the handler:

```
throwErrorAsync()
  .catch(error => { /* handle error here */; return result; })
  .then(result => { /* handle result here */ });
```

You can let an error cascade down a promise chain by re-throwing the error:

```
throwErrorAsync()
  .catch(error => {
      /* handle error from throwErrorAsync() */
      throw error;
  })
  .then(() => { /* will not be called if there's an error */ })
  .catch(error => { /* will get called with the same error */ });
```

It is possible to throw an exception that is not handled by the promise by wrapping the `throw` statement inside a `setTimeout` callback:

```
new Promise((resolve, reject) => {
  setTimeout(() => { throw new Error(); });
});
```

This works because promises cannot handle exceptions thrown asynchronously.

Unhandled rejections

An error will be silently ignored if a promise doesn't have a `catch` block or `reject` handler:

```
throwErrorAsync()
  .then(() => { /* will not be called */ });
// error silently ignored
```

To prevent this, always use a `catch` block:

```
throwErrorAsync()
  .then(() => { /* will not be called */ })
  .catch(error => { /* handle error*/ });
// or
throwErrorAsync()
  .then(() => { /* will not be called */ }, error => { /* handle error*/ });
```

Alternatively, subscribe to the `unhandledrejection` event to catch any unhandled rejected promises:

```
window.addEventListener('unhandledrejection', event => {});
```

Some promises can handle their rejection later than their creation time. The `rejectionhandled` event gets fired whenever such a promise is handled:

```
window.addEventListener('unhandledrejection', event => console.log('unhandled'));
window.addEventListener('rejectionhandled', event => console.log('handled'));
var p = Promise.reject('test');

setTimeout(() => p.catch(console.log), 1000);

// Will print 'unhandled', and after one second 'test' and 'handled'
```

The `event` argument contains information about the rejection. `event.reason` is the error object and `event.promise` is the promise object that caused the event.

In Nodejs the `rejectionhandled` and `unhandledrejection` events are called `rejectionHandled` and `unhandledRejection` on `process`, respectively, and have a different signature:

```
process.on('rejectionHandled', (reason, promise) => {});
process.on('unhandledRejection', (reason, promise) => {});
```

The `reason` argument is the error object and the `promise` argument is a reference to the promise object that caused the event to fire.

Usage of these `unhandledrejection` and `rejectionhandled` events should be considered for debugging purposes only. Typically, all promises should handle their rejections.

Note: Currently, only Chrome 49+ and Node.js support `unhandledrejection` and `rejectionhandled` events.

Caveats

Chaining with `fulfill` and `reject`

The `then(fulfill, reject)` function (with both parameters not `null`) has unique and complex behavior, and shouldn't be used unless you know exactly how it works.

The function works as expected if given `null` for one of the inputs:

```
// the following calls are equivalent
promise.then(fulfill, null)
promise.then(fulfill)

// the following calls are also equivalent
promise.then(null, reject)
promise.catch(reject)
```

However, it adopts unique behavior when both inputs are given:

```
// the following calls are not equivalent!
promise.then(fulfill, reject)
promise.then(fulfill).catch(reject)

// the following calls are not equivalent!
promise.then(fulfill, reject)
promise.catch(reject).then(fulfill)
```

The `then(fulfill, reject)` function looks like it is a shortcut for `then(fulfill).catch(reject)`, but it is not, and will cause problems if used interchangeably. One such problem is that the `reject` handler does not handle errors from the `fulfill` handler. Here is what will happen:

```
Promise.resolve() // previous promise is fulfilled
    .then(() => { throw new Error(); }, // error in the fulfill handler
        error => { /* this is not called! */ });
```

The above code will result in a rejected promise because the error is propagated. Compare it to the following code, which results in a fulfilled promise:

```
Promise.resolve() // previous promise is fulfilled
    .then(() => { throw new Error(); }) // error in the fulfill handler
    .catch(error => { /* handle error */ });
```

A similar problem exists when using `then(fulfill, reject)` interchangeably with `catch(reject).then(fulfill)`, except with propagating fulfilled promises instead of rejected promises.

Synchronously throwing from function that should return a promise

Imagine a function like this:

```
function foo(arg) {
  if (arg === 'unexepectedValue') {
    throw new Error('UnexpectedValue')
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

If such function is used in the **middle** of a promise chain, then apparently there is no problem:

```
makeSomethingAsync().
  .then(() => foo('unexpectedValue'))
  .catch(err => console.log(err)) // <-- Error: UnexpectedValue will be caught here
```

However, if the same function is called outside of a promise chain, then the error will not be handled by it and will be thrown to the application:

```
foo('unexpectedValue') // <-- error will be thrown, so the application will crash
  .then(makeSomethingAsync) // <-- will not run
  .catch(err => console.log(err)) // <-- will not catch
```

There are 2 possible workarounds:

Return a rejected promise with the error

Instead of throwing, do as follows:

```
function foo(arg) {
  if (arg === 'unexepectedValue') {
    return Promise.reject(new Error('UnexpectedValue'))
  }

  return new Promise(resolve =>
    setTimeout(() => resolve(arg), 1000)
  )
}
```

Wrap your function into a promise chain

Your `throw` statement will be properly caught when it is already inside a promise chain:

```
function foo(arg) {
  return Promise.resolve()
    .then(() => {
      if (arg === 'unexepectedValue') {
        throw new Error('UnexpectedValue')
      }

      return new Promise(resolve =>
        setTimeout(() => resolve(arg), 1000)
      )
    })
}
```

Section 42.8: Reconciling synchronous and asynchronous operations

In some cases you may want to wrap a synchronous operation inside a promise to prevent repetition in code branches. Take this example:

```
if (result) { // if we already have a result
  processResult(result); // process it
} else {
  fetchResult().then(processResult);
}
```

The synchronous and asynchronous branches of the above code can be reconciled by redundantly wrapping the synchronous operation inside a promise:

```
var fetch = result
  ? Promise.resolve(result)
  : fetchResult();

fetch.then(processResult);
```

When caching the result of an asynchronous call, it is preferable to cache the promise rather than the result itself. This ensures that only one asynchronous operation is required to resolve multiple parallel requests.

Care should be taken to invalidate cached values when error conditions are encountered.

```
// A resource that is not expected to change frequently
var planets = 'http://swapi.co/api/planets/';
// The cached promise, or null
```

```
var cachedPromise;

function fetchResult() {
    if (!cachedPromise) {
        cachedPromise = fetch(planets)
            .catch(function (e) {
                // Invalidate the current result to retry on the next fetch
                cachedPromise = null;
                // re-raise the error to propagate it to callers
                throw e;
            });
    }
    return cachedPromise;
}
```

Section 42.9: Delay function call

The `setTimeout()` method calls a function or evaluates an expression after a specified number of milliseconds. It is also a trivial way to achieve an asynchronous operation.

In this example calling the `wait` function resolves the promise after the time specified as first argument:

```
function wait(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

wait(5000).then(() => {
    console.log('5 seconds have passed...');
});
```

Section 42.10: "Promisifying" values

The `Promise.resolve` static method can be used to wrap values into promises.

```
let resolved = Promise.resolve(2);
resolved.then(value => {
    // immediately invoked
    // value === 2
});
```

If `value` is already a promise, `Promise.resolve` simply recasts it.

```
let one = new Promise(resolve => setTimeout(() => resolve(2), 1000));
let two = Promise.resolve(one);
two.then(value => {
    // 1 second has passed
    // value === 2
});
```

In fact, `value` can be any "thenable" (object defining a `then` method that works sufficiently like a spec-compliant promise). This allows `Promise.resolve` to convert untrusted 3rd-party objects into trusted 1st-party Promises.

```
let resolved = Promise.resolve({
    then(onResolved) {
        onResolved(2);
    }
});
resolved.then(value => {
    // immediately invoked
    // value === 2
});
```

The `Promise.reject` static method returns a promise which immediately rejects with the given `reason`.

```
let rejected = Promise.reject("Oops!");
rejected.catch(reason => {
    // immediately invoked
    // reason === "Oops!"
});
```

Section 42.11: Using ES2017 async/await

The same example above, Image loading, can be written using async functions. This also allows using the common **`try`/`catch`** method for exception handling.

Note: as of April 2017, the current releases of all browsers but Internet Explorer supports async functions.

```
function loadImage(url) {
    return new Promise((resolve, reject) => {
        const img = new Image();
        img.addEventListener('load', () => resolve(img));
        img.addEventListener('error', () => {
            reject(new Error(`Failed to load ${url}`));
        });
        img.src = url;
    });
}

(async () => {

    // load /image.png and append to #image-holder, otherwise throw error
    try {
        let img = await loadImage('http://example.com/image.png');
        document.getElementById('image-holder').appendChild(img);
    }
    catch (error) {
        console.error(error);
    }

})();
```

Section 42.12: Performing cleanup with finally()

There is currently a proposal (not yet part of the ECMAScript standard) to add a **`finally`** callback to promises that will be executed regardless of whether the promise is fulfilled or rejected. Semantically, this is similar to the **`finally`** clause of the **`try`** block.

You would usually use this functionality for cleanup:

```
var loadingData = true;

fetch('/data')
    .then(result => processData(result.data))
    .catch(error => console.error(error))
    .finally(() => {
        loadingData = false;
    });
```

It is important to note that the **finally** callback doesn't affect the state of the promise. It doesn't matter what value it returns, the promise stays in the fulfilled/rejected state that it had before. So in the example above the promise will be resolved with the return value of `processData(result.data)` even though the **finally** callback returned **undefined**.

With the standardization process still being in progress, your promises implementation most likely won't support **finally** callbacks out of the box. For synchronous callbacks you can add this functionality with a polyfill however:

```
if (!Promise.prototype.finally) {
    Promise.prototype.finally = function(callback) {
        return this.then(result => {
            callback();
            return result;
        }, error => {
            callback();
            throw error;
        });
    };
}
```

Section 42.13: forEach with promises

It is possible to effectively apply a function (`cb`) which returns a promise to each element of an array, with each element waiting to be processed until the previous element is processed.

```
function promiseForEach(arr, cb) {
  var i = 0;

  var nextPromise = function () {
    if (i >= arr.length) {
      // Processing finished.
      return;
    }

    // Process next function. Wrap in `Promise.resolve` in case
    // the function does not return a promise
    var newPromise = Promise.resolve(cb(arr[i], i));
    i++;
    // Chain to finish processing.
    return newPromise.then(nextPromise);
  };

  // Kick off the chain.
  return Promise.resolve().then(nextPromise);
};
```

This can be helpful if you need to efficiently process thousands of items, one at a time. Using a regular **for** loop to create the promises will create them all at once and take up a significant amount of RAM.

Section 42.14: Asynchronous API request

This is an example of a simple `GET` API call wrapped in a promise to take advantage of its asynchronous functionality.

```
var get = function(path) {
  return new Promise(function(resolve, reject) {
    let request = new XMLHttpRequest();
    request.open('GET', path);
    request.onload = resolve;
    request.onerror = reject;
    request.send();
  });
};
```

More robust error handling can be done using the following `onload` and `onerror` functions.

```
request.onload = function() {
  if (this.status >= 200 && this.status < 300) {
    if(request.response) {
      // Assuming a successful call returns JSON
      resolve(JSON.parse(request.response));
    } else {
      resolve();
  } else {
    reject({
      'status': this.status,
      'message': request.statusText
    });
  }
};

request.onerror = function() {
  reject({
    'status': this.status,
    'message': request.statusText
  });
};
```