Using promises

A Promise is an **object representing the eventual completion or failure of an asynchronous operation**. Since most people are consumers of already-created promises, this guide will explain the consumption of returned promises before explaining how to create them.

Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.

Chaining

A common need is to execute two or more asynchronous operations back to back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step. In the old days, doing several asynchronous operations in a row would lead to the classic callback pyramid of doom:

```
doSomething(function (result) {
  doSomethingElse(result, function (newResult) {
    doThirdThing(newResult, function (finalResult) {
      console.log(`Got the final result: ${finalResult}`);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

With promises, we accomplish this by creating a promise chain. The API design of promises makes this great, because callbacks are attached to the returned promise object, instead of being passed into a function.

Here's the magic: the `then()` function returns a **new promise**, different from the original:

```
const promise = doSomething();
const promise2 = promise.then(successCallback, failureCallback);
```

This second promise (`promise2`) represents the completion not just of `doSomething()`, but also of the `successCallback` or `failureCallback` you passed in — which can be other asynchronous functions returning a promise.

With this pattern, you can create longer chains of processing, where each promise represents the completion of one asynchronous step in the chain. In addition, the arguments to then are optional, and `catch(failureCallback)` is short for `then(null, failureCallback)` — so if your error handling code is the same for all steps, you can attach it to the end of the chain:

```
doSomething()
  .then(function (result) {
    return doSomethingElse(result);
  })
  .then(function (newResult) {
    return doThirdThing(newResult);
  })
  .then(function (finalResult) {
    console.log(`Got the final result: ${finalResult}`);
  })
  .catch(failureCallback);
```

You might see this expressed with arrow functions instead:

```
doSomething()
  .then((result) => doSomethingElse(result))
  .then((newResult) => doThirdThing(newResult))
  .then((finalResult) => {
    console.log(`Got the final result: ${finalResult}`);
  })
  .catch(failureCallback);
```

Important: **Always return results**, otherwise callbacks won't catch the result of a previous promise (with arrow functions, `() => x` is short for `() => { return x; }`)

Therefore, whenever your operation encounters a promise, return it and defer its handling to the next then handler.

```
const listOfIngredients = [];

doSomething()
  .then((url) =>
    fetch(url)
      .then((res) => res.json())
      .then((data) => {
        listOfIngredients.push(data);
```

```
      }),
  )
  .then(() => {
    console.log(listOfIngredients);
  });

// OR

doSomething()
  .then((url) => fetch(url))
  .then((res) => res.json())
  .then((data) => {
    listOfIngredients.push(data);
  })
  .then(() => {
    console.log(listOfIngredients);
  });
```

Nesting

Nesting is a control structure to limit the scope of `catch` statements. Specifically, a nested `catch` only catches failures in its scope and below, not errors higher up in the chain outside the nested scope. When used correctly, this gives greater precision in error recovery:

```
doSomethingCritical()
  .then((result) =>
    doSomethingOptional(result)
      .then((optionalResult) => doSomethingExtraNice(optionalResult))
      .catch((e) => {}),
  ) // Ignore if optional stuff fails; proceed.
  .then(() => moreCriticalStuff())
  .catch((e) => console.error(`Critical failure: ${e.message}`));
```

The inner error-silencing catch handler only catches failures from `doSomethingOptional()` and `doSomethingExtraNice()`, after which the code resumes with `moreCriticalStuff()`. Importantly, if `doSomethingCritical()` fails, its error is caught by the final (outer) catch only, and does not get swallowed by the inner catch handler.

Chaining after a catch

It's possible to chain after a failure, i.e. a `catch`, which is useful to accomplish new actions even after an action failed in the chain. Read the following example:

```
new Promise((resolve, reject) => {
  console.log("Initial");

  resolve();
})
  .then(() => {
    throw new Error("Something failed");

    console.log("Do this");
  })
  .catch(() => {
    console.error("Do that");
  })
  .then(() => {
    console.log("Do this, no matter what happened before");
  });
```

This will output the following text:

```
Initial
Do that
Do this, no matter what happened before
```

Note: "Do this" is not displayed because the "Something failed" error caused a rejection.

Common mistakes

```
// Bad example! Spot 3 mistakes!

doSomething()
  .then(function (result) {
    // Forgot to return promise from inner chain + unnecessary nesting
    doSomethingElse(result).then((newResult) => doThirdThing(newResult));
  })
  .then(() => doFourthThing());
// Forgot to terminate chain with a catch!
```

The first mistake is to not chain things together properly. This happens when we create a new promise but forget to return it. As a consequence, the chain is broken — or rather, we have two

independent chains racing. This means `doFourthThing()` won't wait for `doSomethingElse()` or `doThirdThing()` to finish, and will run concurrently with them.

The second mistake is to nest unnecessarily, enabling the first mistake. Nesting also limits the scope of inner error handlers, which—if unintended—can lead to uncaught errors.

The third mistake is forgetting to terminate chains with catch.

A good rule is to always either return or terminate promise chains, and as soon as you get a new promise, return it immediately, to flatten things:

```
doSomething()
  .then(function (result) {
    // If using a full function expression: return the promise
    return doSomethingElse(result);
  })
  // If using arrow functions: omit the braces and implicitly return the result
  .then((newResult) => doThirdThing(newResult))
  // Even if the previous chained promise returns a result, the next one
  // doesn't necessarily have to use it. You can pass a handler that doesn't
  // consume any result.
  .then((/* result ignored */) => doFourthThing())
  // Always end the promise chain with a catch handler to avoid any
  // unhandled rejections!
  .catch((error) => console.error(error));
```

Error handling

You might remember seeing `failureCallback` three times in the pyramid of doom earlier, compared to only once at the end of the promise chain:

```
doSomething()
  .then((result) => doSomethingElse(result))
  .then((newResult) => doThirdThing(newResult))
  .then((finalResult) => console.log(`Got the final result: ${finalResult}`))
  .catch(failureCallback);
```

If there's an exception, the browser will look down the chain for `.catch()` handlers or `onRejected`. This is very much modeled after how synchronous code works:

```
try {
  const result = syncDoSomething();
  const newResult = syncDoSomethingElse(result);
  const finalResult = syncDoThirdThing(newResult);
  console.log(`Got the final result: ${finalResult}`);
} catch (error) {
  failureCallback(error);
}
```

This symmetry with asynchronous code culminates in the `async/await` syntax:

```
async function foo() {
  try {
    const result = await doSomething();
    const newResult = await doSomethingElse(result);
    const finalResult = await doThirdThing(newResult);
    console.log(`Got the final result: ${finalResult}`);
  } catch (error) {
    failureCallback(error);
  }
}
```

It builds on promises — for example, `doSomething()` is the same function as before, so there's minimal refactoring needed to change from promises to `async/await`.

Promise rejection events

If a promise rejection event is not handled by any handler, it bubbles to the top of the call stack, and the host needs to surface it. On the web, whenever a promise is rejected, one of two events is sent to the global scope. The two events are:

```
unhandledrejection
```

Sent when a promise is rejected but there is no rejection handler available.

```
rejectionhandled
```

Sent when a handler is attached to a rejected `promise` that has already caused an unhandledrejection event.

In both cases, the event (of type `PromiseRejectionEvent`) has as members a `promise` property indicating the promise that was rejected, and a `reason` property that provides the reason given for the promise to be rejected.

In Node.js, handling promise rejection is slightly different. You capture unhandled rejections by adding a handler for the Node.js `unhandledRejection` event (notice the difference in capitalization of the name), like this:

```
process.on("unhandledRejection", (reason, promise) => {
  // Add code here to examine the "promise" and "reason" values
});
```

Composition

There are four composition tools for running asynchronous operations concurrently: `Promise.all()`, `Promise.allSettled()`, `Promise.any()`, and `Promise.race()`.

We can start operations at the same time and wait for them all to finish like this:

```
Promise.all([func1(), func2(), func3()]).then(([result1, result2, result3]) => {
  // use result1, result2 and result3
});
```

If one of the promises in the array rejects, Promise.all() immediately rejects the returned promise and aborts the other operations. This may cause unexpected state or behavior. Promise.allSettled() is another composition tool that ensures all operations are complete before resolving.

These methods all run promises concurrently — a sequence of promises are started simultaneously and do not wait for each other. Sequential composition is possible using some clever JavaScript:

```
[func1, func2, func3]
  .reduce((p, f) => p.then(f), Promise.resolve())
```

```
  .then((result3) => {
    /* use result3 */
  });
```

In this example, we reduce an array of asynchronous functions down to a promise chain. The code above is equivalent to:

```
Promise.resolve()
  .then(func1)
  .then(func2)
  .then(func3)
  .then((result3) => {
    /* use result3 */
});
```

This can be made into a reusable compose function, which is common in functional programming:

```
const applyAsync = (acc, val) => acc.then(val);
const composeAsync =
  (...funcs) =>
  (x) =>
    funcs.reduce(applyAsync, Promise.resolve(x));
```

The `composeAsync()` function accepts any number of functions as arguments and returns a new function that accepts an initial value to be passed through the composition pipeline:

```
const transformData = composeAsync(func1, func2, func3);
const result3 = transformData(data);
```

Sequential composition can also be done more succinctly with async/await:

```
let result;
for (const f of [func1, func2, func3]) {
  result = await f(result);
}
/* use last result (i.e. result3) */
```

However, before you compose promises sequentially, consider if it's really necessary — it's always better to run promises concurrently so that they don't unnecessarily block each other unless one promise's execution depends on another's result.

Creating a Promise around an old callback API

A `Promise` can be created from scratch using its constructor. This should be needed only to wrap old APIs.

In an ideal world, all asynchronous functions would already return promises. Unfortunately, some APIs still expect success and/or failure callbacks to be passed in the old way. The most obvious example is the `setTimeout()` function:

```
setTimeout(() => saySomething("10 seconds passed"), 10 * 1000);
```

Mixing old-style callbacks and promises is problematic. If `saySomething()` fails or contains a programming error, nothing catches it. This is intrinsic to the design of setTimeout.

Luckily we can wrap `setTimeout` in a promise. The best practice is to wrap the callback-accepting functions at the lowest possible level, and then never call them directly again:

```
const wait = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

wait(10 * 1000)
  .then(() => saySomething("10 seconds"))
  .catch(failureCallback);
```

The promise constructor takes an executor function that lets us resolve or reject a promise manually. Since `setTimeout()` doesn't really fail, we left out reject in this case. For more information on how the executor function works, see the `Promise()` reference.

Timing

Guarantees

In the callback-based API, **when and how the callback gets called** depends on the API implementor. For example, the callback may be called synchronously or asynchronously:

```
function doSomething(callback) {
  if (Math.random() > 0.5) {
    callback();
  } else {
    setTimeout(() => callback(), 1000);
  }
}
```

To avoid surprises, functions passed to `then()` will never be called synchronously, even with an already-resolved promise:

```
Promise.resolve().then(() => console.log(2));
console.log(1);
// Logs: 1, 2
```

Instead of running immediately, the passed-in function is put on a microtask queue, which means it runs later, just before control is returned to the event loop; i.e. pretty soon:

```
const wait = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

wait(0).then(() => console.log(4));
Promise.resolve()
  .then(() => console.log(2))
  .then(() => console.log(3));
console.log(1); // 1, 2, 3, 4
```

Task queues vs. microtasks

Promise callbacks are handled as a microtask whereas setTimeout() callbacks are handled as task queues.

```
const promise = new Promise((resolve, reject) => {
  console.log("Promise callback");
  resolve();
}).then((result) => {
  console.log("Promise callback (.then)");
});

setTimeout(() => {
  console.log("event-loop cycle: Promise (fulfilled)", promise);
}, 0);

console.log("Promise (pending)", promise);
```

The code above will output:

```
Promise callback
Promise (pending) Promise {<pending>}
Promise callback (.then)
event-loop cycle: Promise (fulfilled) Promise {<fulfilled>}
```