25. Promises for asynchronous programming

This chapter is an introduction to asynchronous programming via Promises in general and the ECMAScript 6 Promise API in particular. The previous chapter explains the foundations of asynchronous programming in JavaScript. You can consult it whenever there is something that you don't understand in this chapter.

- 25.1. Overview
  - 25.1.1. Chaining `then()` calls
  - 25.1.2. Executing asynchronous functions in parallel
  - 25.1.3. Glossary: Promises
- 25.2. Introduction: Promises
- 25.3. A first example
- 25.4. Three ways of understanding Promises
  - 25.4.1. Conceptually: calling a Promise-based function is blocking
  - 25.4.2. A Promise is a container for an asynchronously delivered value
  - 25.4.3. A Promise is an event emitter
- 25.5. Creating and using Promises
  - 25.5.1. Producing a Promise
  - 25.5.2. The states of Promises
  - 25.5.3. Consuming a Promise
  - 25.5.4. Promises are always asynchronous
- 25.6. Examples
  - 25.6.1. Example: promisifying `fs.readFile()`
  - 25.6.2. Example: promisifying `XMLHttpRequest`
  - 25.6.3. Example: delaying an activity
  - 25.6.4. Example: timing out a Promise
- 25.7. Other ways of creating Promises
  - 25.7.1. `Promise.resolve()`
  - 25.7.2. `Promise.reject()`
- 25.8. Chaining Promises

- 25.8.1. Resolving Q with a normal value
- 25.8.2. Resolving Q with a thenable
- 25.8.3. Resolving Q from `onRejected`
- 25.8.4. Rejecting Q by throwing an exception
- 25.8.5. Chaining and errors
- 25.9. Common Promise chaining mistakes
  - 25.9.1. Mistake: losing the tail of a Promise chain
  - 25.9.2. Mistake: nesting Promises
  - 25.9.3. Mistake: creating Promises instead of chaining
  - 25.9.4. Mistake: using `then()` for error handling
- 25.10. Tips for error handling
  - 25.10.1. Operational errors versus programmer errors
  - 25.10.2. Handling exceptions in Promise-based functions
  - 25.10.3. Further reading
- 25.11. Composing Promises
  - 25.11.1. Manually forking and joining computations
  - 25.11.2. Forking and joining computations via `Promise.all()`
  - 25.11.3. `map()` via `Promise.all()`
  - 25.11.4. Timing out via `Promise.race()`
- 25.12. Two useful additional Promise methods
  - 25.12.1. `done()`
  - 25.12.2. `finally()`
- 25.13. Node.js: using callback-based sync functions with Promises
- 25.14. ES6-compatible Promise libraries
- 25.15. Next step: using Promises via generators
- 25.16. Promises in depth: a simple implementation
  - 25.16.1. A stand-alone Promise
  - 25.16.2. Chaining
  - 25.16.3. Flattening
  - 25.16.4. Promise states in more detail
  - 25.16.5. Exceptions
  - 25.16.6. Revealing constructor pattern
- 25.17. Advantages and limitations of Promises
  - 25.17.1. Advantages of Promises
  - 25.17.2. Promises are not always the best choice

- 25.18. Reference: the ECMAScript 6 Promise API
  - 25.18.1. `Promise` constructor
  - 25.18.2. Static `Promise` methods
  - 25.18.3. `Promise.prototype` methods
- 25.19. Further reading

25.1 Overview

Promises are an alternative to callbacks for delivering the results of an asynchronous computation. They require more effort from implementors of asynchronous functions, but provide several benefits for users of those functions.

The following function returns a result asynchronously, via a Promise:

```
function asyncFunc() {
    return new Promise(
        function (resolve, reject) {
            ···
            resolve(result);
            ···
            reject(error);
        });
}
```

You call `asyncFunc()` as follows:

```
asyncFunc()
.then(result => { ··· })
.catch(error => { ··· });
```

25.1.1 Chaining `then()` calls

`then()` always returns a Promise, which enables you to chain method calls:

```
asyncFunc1()
.then(result1 => {
    // Use result1
    return asyncFunction2(); // (A)
})
.then(result2 => { // (B)
    // Use result2
})
.catch(error => {
    // Handle errors of asyncFunc1() and asyncFunc2()
});
```

How the Promise P returned by `then()` is settled depends on what its callback does:

- If it returns a Promise (as in line A), the settlement of that Promise is forwarded to P. That's why the callback from line B can pick up the settlement of `asyncFunction2`'s Promise.
- If it returns a different value, that value is used to settle P.
- If throws an exception then P is rejected with that exception.

Furthermore, note how `catch()` handles the errors of two asynchronous function calls (`asyncFunction1()` and `asyncFunction2()`). That is, uncaught errors are passed on until there is an error handler.

25.1.2 Executing asynchronous functions in parallel

If you chain asynchronous function calls via `then()`, they are executed sequentially, one at a time:

```
asyncFunc1()
.then(() => asyncFunc2());
```

If you don't do that and call all of them immediately, they are basically executed in parallel (a *fork* in Unix process terminology):

```
asyncFunc1();
asyncFunc2();
```

`Promise.all()` enables you to be notified once all results are in (a *join* in Unix process terminology). Its input is an Array of Promises, its output a single Promise that is fulfilled with an Array of the results.

```
Promise.all([
    asyncFunc1(),
    asyncFunc2(),
])
.then(([result1, result2]) => {
    ···
})
.catch(err => {
    // Receives first rejection among the Promises
    ···
});
```

25.1.3 Glossary: Promises

The Promise API is about delivering results asynchronously. A *Promise object* (short: Promise) is a stand-in for the result, which is delivered via that object.

States:

- A Promise is always in one of three mutually exclusive states:
    - Before the result is ready, the Promise is *pending*.
    - If a result is available, the Promise is *fulfilled*.
    - If an error happened, the Promise is *rejected*.
- A Promise is *settled* if “things are done” (if it is either fulfilled or rejected).
- A Promise is settled exactly once and then remains unchanged.

Reacting to state changes:

- *Promise reactions* are callbacks that you register with the Promise method `then()`, to be notified of a fulfillment or a rejection.
- A *thenable* is an object that has a Promise-style `then()` method. Whenever the API is only interested in being notified of settlements, it only demands thenables (e.g. the values returned from `then()` and `catch()`; or the values handed to `Promise.all()` and `Promise.race()`).

Changing states: There are two operations for changing the state of a Promise. After you have invoked either one of them once, further invocations have no effect.

- *Rejecting* a Promise means that the Promise becomes rejected.
- *Resolving* a Promise has different effects, depending on what value you are resolving with:
  - Resolving with a normal (non-thenable) value fulfills the Promise.
  - Resolving a Promise P with a thenable T means that P can't be resolved anymore and will now follow T's state, including its fulfillment or rejection value. The appropriate P reactions will get called once T settles (or are called immediately if T is already settled).

25.2 Introduction: Promises

Promises are a pattern that helps with one particular kind of asynchronous programming: a function (or method) that returns a single result asynchronously. One popular way of receiving such a result is via a callback ("callbacks as continuations"):

```
asyncFunction(arg1, arg2,
  result => {
    console.log(result);
  });
```

Promises provide a better way of working with callbacks: Now an asynchronous function returns a *Promise*, an object that serves as a placeholder and container for the final result. Callbacks registered via the Promise method `then()` are notified of the result:

```
asyncFunction(arg1, arg2)
.then(result => {
  console.log(result);
});
```

Compared to callbacks as continuations, Promises have the following advantages:

- No inversion of control: similarly to synchronous code, Promise-based functions return results, they don't (directly) continue – and control – execution via callbacks. That is, the caller stays in control.
- Chaining is simpler: If the callback of `then()` returns a Promise (e.g.

the result of calling another Promise-based function) then `then()` returns that Promise (how this really works is more complicated and explained later). As a consequence, you can chain `then()` method calls:

```
asyncFunction1(a, b)
.then(result1 => {
    console.log(result1);
    return asyncFunction2(x, y);
})
.then(result2 => {
    console.log(result2);
});
```

- Composing asynchronous calls (loops, mapping, etc.): is a little easier, because you have data (Promise objects) you can work with.
- Error handling: As we shall see later, error handling is simpler with Promises, because, once again, there isn't an inversion of control. Furthermore, both exceptions and asynchronous errors are managed the same way.
- Cleaner signatures: With callbacks, the parameters of a function are mixed; some are input for the function, others are responsible for delivering its output. With Promises, function signatures become cleaner; all parameters are input.
- Standardized: Prior to Promises, there were several incompatible ways of handling asynchronous results (Node.js callbacks, XMLHttpRequest, IndexedDB, etc.). With Promises, there is a clearly defined standard: ECMAScript 6. ES6 follows the standard Promises/A+ [1]. Since ES6, an increasing number of APIs is based on Promises.

25.3 A first example

Let's look at a first example, to give you a taste of what working with Promises is like.

With Node.js-style callbacks, reading a file asynchronously looks like this:

```
fs.readFile('config.json',
    function (error, text) {
        if (error) {
            console.error('Error while reading config file');
        } else {
            try {
                const obj = JSON.parse(text);
                console.log(JSON.stringify(obj, null, 4));
            } catch (e) {
                console.error('Invalid JSON in file');
            }
        }
    });
```

With Promises, the same functionality is used like this:

```
readFilePromisified('config.json')
.then(function (text) { // (A)
    const obj = JSON.parse(text);
    console.log(JSON.stringify(obj, null, 4));
})
.catch(function (error) { // (B)
    // File read error or JSON SyntaxError
    console.error('An error occurred', error);
});
```

There are still callbacks, but they are provided via methods that are invoked on the result (`then()` and `catch()`). The error callback in line B is convenient in two ways: First, it's a single style of handling errors (versus `if (error)` and `try-catch` in the previous example). Second, you can handle the errors of both `readFilePromisified()` and the callback in line A from a single location.

The code of `readFilePromisified()` is shown later.

25.4 Three ways of understanding Promises

Let's look at three ways of understanding Promises.

The following code contains a Promise-based function `asyncFunc()` and its invocation.

```
function asyncFunc() {
    return new Promise((resolve, reject) => { // (A)
        setTimeout(() => resolve('DONE'), 100); // (B)
    });
}
asyncFunc()
.then(x => console.log('Result: '+x));

// Output:
// Result: DONE
```

`asyncFunc()` returns a Promise. Once the actual result `'DONE'` of the asynchronous computation is ready, it is delivered via `resolve()` (line B), which is a parameter of the callback that starts in line A.

So what is a Promise?

- Conceptually, invoking `asyncFunc()` is a blocking function call.
- A Promise is both a container for a value and an event emitter.

25.4.1 Conceptually: calling a Promise-based function is blocking

The following code invokes `asyncFunc()` from the async function `main()`. Async functions are a feature of ECMAScript 2017.

```
async function main() {
    const x = await asyncFunc(); // (A)
    console.log('Result: '+x); // (B)

    // Same as:
    // asyncFunc()
    // .then(x => console.log('Result: '+x));
}
main();
```

The body of `main()` expresses well what's going on *conceptually*, how we usually think about asynchronous computations. Namely, `asyncFunc()` is a blocking function call:

- Line A: Wait until `asyncFunc()` is finished.
- Line B: Then log its result `x`.

Prior to ECMAScript 6 and generators, you couldn't suspend and resume code. That's why, for Promises, you put everything that happens after the code is resumed into a callback. Invoking that callback is the same as resuming the code.

25.4.2 A Promise is a container for an asynchronously delivered value

If a function returns a Promise then that Promise is like a blank into which the function will (usually) fill in its result, once it has computed it. You can simulate a simple version of this process via an Array:

```
function asyncFunc() {
   const blank = [];
   setTimeout(() => blank.push('DONE'), 100);
   return blank;
}
const blank = asyncFunc();
// Wait until the value has been filled in
setTimeout(() => {
   const x = blank[0]; // (A)
   console.log('Result: '+x);
}, 200);
```

With Promises, you don't access the eventual value via `[0]` (as in line A), you use method `then()` and a callback.

25.4.3 A Promise is an event emitter

Another way to view a Promise is as an object that emits events.

```
function asyncFunc() {
   const eventEmitter = { success: [] };
   setTimeout(() => { // (A)
      for (const handler of eventEmitter.success) {
         handler('DONE');
      }
   }, 100);
   return eventEmitter;
}
asyncFunc()
.success.push(x => console.log('Result: '+x)); // (B)
```

Registering the event listener (line B) can be done after calling `asyncFunc()`, because the callback handed to `setTimeout()` (line A) is executed asynchronously (after this piece of code is finished).

Normal event emitters specialize in delivering multiple events, starting as soon as you register.

In contrast, Promises specialize in delivering exactly one value and come with built-in protection against registering too late: the result of a Promise is cached and passed to event listeners that are registered after the Promise was settled.

25.5 Creating and using Promises

Let's look at how Promises are operated from the producer and the consumer side.

25.5.1 Producing a Promise

As a producer, you create a Promise and send a result via it:

```
const p = new Promise(
  function (resolve, reject) { // (A)
    ···
    if (···) {
      resolve(value); // success
    } else {
      reject(reason); // failure
    }
  });
```

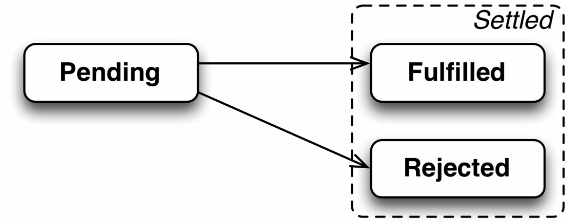
25.5.2 The states of Promises

Once a result was delivered via a Promise, the Promise stays locked in to that result. That means each Promise is always in either one of three (mutually exclusive) states:

- Pending: the result hasn't been computed, yet (the initial state of each Promise)
- Fulfilled: the result was computed successfully
- Rejected: a failure occurred during computation

A Promise is *settled* (the computation it represents has finished) if it is either fulfilled or rejected. A Promise can only be settled once and then stays settled. Subsequent attempts to settle have no effect.

The parameter of `new Promise()` (starting in line A) is called an *executor*:



- Resolving: If the computation went well, the executor sends the result via `resolve()`. That usually fulfills the Promise `p`. But it may not – resolving with a Promise `q` leads to `p` tracking `q`: If `q` is still pending then so is `p`. However `q` is settled, `p` will be settled the same way.
- Rejecting: If an error happened, the executor notifies the Promise consumer via `reject()`. That always rejects the Promise.

If an exception is thrown inside the executor, `p` is rejected with that exception.

25.5.3 Consuming a Promise

As a consumer of `promise`, you are notified of a fulfillment or a rejection via *reactions* – callbacks that you register with the methods `then()` and `catch()`:

```
promise
.then(value => { /* fulfillment */ })
.catch(error => { /* rejection */ });
```

What makes Promises so useful for asynchronous functions (with one-off results) is that once a Promise is settled, it doesn't change anymore. Furthermore, there are never any race conditions, because it doesn't matter whether you invoke `then()` or `catch()` before or after a Promise is settled:

- Reactions that are registered with a Promise before it is settled, are notified of the settlement once it happens.
- Reactions that are registered with a Promise after it is settled, receive the cached settled value "immediately" (their invocations are queued as tasks).

Note that `catch()` is simply a more convenient (and recommended) alternative to calling `then()`. That is, the following two invocations are equivalent:

```
promise.then(
    null,
    error => { /* rejection */ });

promise.catch(
    error => { /* rejection */ });
```

25.5.4 Promises are always asynchronous

A Promise library has complete control over whether results are delivered to Promise reactions synchronously (right away) or asynchronously (after the current continuation, the current piece of code, is finished). However, the Promises/A+ specification demands that the latter mode of execution be always used. It states so via the following requirement (2.2.4) for the `then()` method:

> `onFulfilled` or `onRejected` must not be called until the execution context stack contains only platform code.

That means that your code can rely on run-to-completion semantics (as explained in the previous chapter) and that chaining Promises won't starve other tasks of processing time.

Additionally, this constraint prevents you from writing functions that sometimes return results immediately, sometimes asynchronously. This is an anti-pattern, because it makes code unpredictable. For more information, consult "Designing APIs for Asynchrony" by Isaac Z. Schlueter.

25.6 Examples

Before we dig deeper into Promises, let's use what we have learned so far in a few examples.

Some of the examples in this section are available in the GitHub repository promise-examples.

25.6.1 Example: promisifying `fs.readFile()`

The following code is a Promise-based version of the built-in Node.js function fs.readFile().

```
import {readFile} from 'fs';

function readFilePromisified(filename) {
  return new Promise(
    function (resolve, reject) {
      readFile(filename, { encoding: 'utf8' },
        (error, data) => {
          if (error) {
            reject(error);
          } else {
            resolve(data);
          }
        });
    });
}
```

`readFilePromisified()` is used like this:

```
readFilePromisified(process.argv[2])
.then(text => {
  console.log(text);
})
.catch(error => {
  console.log(error);
});
```

25.6.2 Example: promisifying `XMLHttpRequest`

The following is a Promise-based function that performs an HTTP GET via the event-based XMLHttpRequest API:

```
function httpGet(url) {
    return new Promise(
        function (resolve, reject) {
            const request = new XMLHttpRequest();
            request.onload = function () {
                if (this.status === 200) {
                    // Success
                    resolve(this.response);
                } else {
                    // Something went wrong (404 etc.)
                    reject(new Error(this.statusText));
                }
            };
            request.onerror = function () {
                reject(new Error(
                    'XMLHttpRequest Error: '+this.statusText));
            };
            request.open('GET', url);
            request.send();
        });
}
```

This is how you use `httpGet()`:

```
httpGet('http://example.com/file.txt')
.then(
    function (value) {
        console.log('Contents: ' + value);
    },
    function (reason) {
        console.error('Something went wrong', reason);
    });
```

25.6.3 Example: delaying an activity

Let’s implement `setTimeout()` as the Promise-based function `delay()` (similar to Q.delay()).

```
function delay(ms) {
    return new Promise(function (resolve, reject) {
        setTimeout(resolve, ms); // (A)
    });
}

// Using delay():
delay(5000).then(function () { // (B)
    console.log('5 seconds have passed!')
});
```

Note that in line A, we are calling `resolve` with zero parameters, which is the same as calling `resolve(undefined)`. We don't need the fulfillment value in line B, either and simply ignore it. Just being notified is enough here.

25.6.4 Example: timing out a Promise

```
function timeout(ms, promise) {
    return new Promise(function (resolve, reject) {
        promise.then(resolve);
        setTimeout(function () {
            reject(new Error('Timeout after '+ms+' ms')); // (A)
        }, ms);
    });
}
```

Note that the rejection after the timeout (in line A) does not cancel the request, but it does prevent the Promise being fulfilled with its result.

Using `timeout()` looks like this:

```
timeout(5000, httpGet('http://example.com/file.txt'))
.then(function (value) {
    console.log('Contents: ' + value);
})
.catch(function (reason) {
    console.error('Error or timeout', reason);
});
```

25.7 Other ways of creating Promises

Now we are ready to dig deeper into the features of Promises. Let's first explore two more ways of creating Promises.

25.7.1 `Promise.resolve()`

`Promise.resolve(x)` works as follows:

- For most values `x`, it returns a Promise that is fulfilled with `x`:

  ```
  Promise.resolve('abc')
  .then(x => console.log(x)); // abc
  ```

- If `x` is a Promise whose constructor is the receiver (`Promise` if you call `Promise.resolve()`) then `x` is returned unchanged:

  ```
  const p = new Promise(() => null);
  console.log(Promise.resolve(p) === p); // true
  ```

- If `x` is a thenable, it is converted to a Promise: the settlement of the thenable will also become the settlement of the Promise. The following code demonstrates that. `fulfilledThenable` behaves roughly like a Promise that was fulfilled with the string `'hello'`. After converting it to the Promise `promise`, method `then()` works as expected (last line).

  ```
  const fulfilledThenable = {
    then(reaction) {
      reaction('hello');
    }
  };
  const promise = Promise.resolve(fulfilledThenable);
  console.log(promise instanceof Promise); // true
  promise.then(x => console.log(x)); // hello
  ```

That means that you can use `Promise.resolve()` to convert any value (Promise, thenable or other) to a Promise. In fact, it is used by `Promise.all()` and `Promise.race()` to convert Arrays of arbitrary values to Arrays of Promises.

25.7.2 `Promise.reject()`

`Promise.reject(err)` returns a Promise that is rejected with `err`:

```
const myError = new Error('Problem!');
Promise.reject(myError)
.catch(err => console.log(err === myError)); // true
```

25.8 Chaining Promises

In this section, we take a closer look at how Promises can be chained. The result of the method call:

```
P.then(onFulfilled, onRejected)
```

is a new Promise Q. That means that you can keep the Promise-based control flow going by invoking `then()` on Q:

- Q is resolved with what is returned by either `onFulfilled` or `onRejected`.
- Q is rejected if either `onFulfilled` or `onRejected` throw an exception.

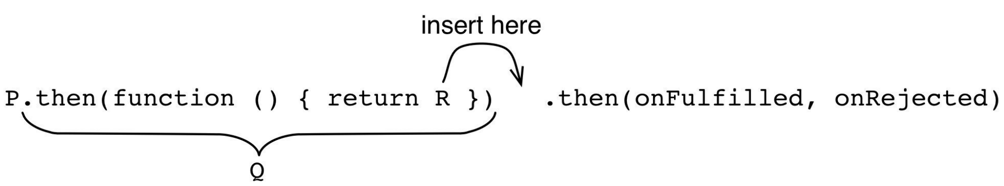
25.8.1 Resolving Q with a normal value

If you resolve the Promise Q returned by `then()` with a normal value, you can pick up that value via a subsequent `then()`:

```
asyncFunc()
.then(function (value1) {
    return 123;
})
.then(function (value2) {
    console.log(value2); // 123
});
```

25.8.2 Resolving Q with a thenable

You can also resolve the Promise Q returned by `then()` with a *thenable* R. A thenable is any object that has a method `then()` that works like `Promise.prototype.then()`. Thus, Promises are thenables. Resolving with R (e.g. by returning it from `onFulfilled`) means that it is inserted “after” Q: R’s settlement is forwarded to Q’s `onFulfilled` and `onRejected` callbacks. In a way, Q becomes R.



The main use for this mechanism is to flatten nested `then()` calls, like in the following example:

```
asyncFunc1()
.then(function (value1) {
    asyncFunc2()
    .then(function (value2) {
        ···
    });
})
```

The flat version looks like this:

```
asyncFunc1()
.then(function (value1) {
    return asyncFunc2();
})
.then(function (value2) {
    ···
})
```

25.8.3 Resolving Q from `onRejected`

Whatever you return in an error handler becomes a fulfillment value (not rejection value!). That allows you to specify default values that are used in case of failure:

```
retrieveFileName()
.catch(function () {
    // Something went wrong, use a default value
    return 'Untitled.txt';
})
.then(function (fileName) {
    ···
});
```

25.8.4 Rejecting Q by throwing an exception

Exceptions that are thrown in the callbacks of `then()` and `catch()` are passed on to the next error handler, as rejections:

```
asyncFunc()
.then(function (value) {
    throw new Error();
})
.catch(function (reason) {
    // Handle error here
});
```

25.8.5 Chaining and errors

There can be one or more `then()` method calls that don't have error handlers. Then the error is passed on until there is an error handler.

```
asyncFunc1()
.then(asyncFunc2)
.then(asyncFunc3)
.catch(function (reason) {
    // Something went wrong above
});
```

25.9 Common Promise chaining mistakes

25.9.1 Mistake: losing the tail of a Promise chain

In the following code, a chain of two Promises is built, but only the first part of it is returned. As a consequence, the tail of the chain is lost.

```
// Don't do this
function foo() {
    const promise = asyncFunc();
    promise.then(result => {
        ...
    });

    return promise;
}
```

This can be fixed by returning the tail of the chain:

```
function foo() {
    const promise = asyncFunc();
    return promise.then(result => {
        ...
    });
}
```

If you don't need the variable `promise`, you can simplify this code further:

```
function foo() {
  return asyncFunc()
  .then(result => {
    ...
  });
}
```

25.9.2 Mistake: nesting Promises

In the following code, the invocation of `asyncFunc2()` is nested:

```
// Don't do this
asyncFunc1()
.then(result1 => {
  asyncFunc2()
  .then(result2 => {
    ...
  });
});
```

The fix is to un-nest this code by returning the second Promise from the first `then()` and handling it via a second, chained, `then()`:

```
asyncFunc1()
.then(result1 => {
  return asyncFunc2();
})
.then(result2 => {
  ...
});
```

25.9.3 Mistake: creating Promises instead of chaining

In the following code, method `insertInto()` creates a new Promise for its result (line A):

```
// Don’t do this
class Model {
  insertInto(db) {
    return new Promise((resolve, reject) => { // (A)
      db.insert(this.fields) // (B)
      .then(resultCode => {
        this.notifyObservers({event: 'created', model: this});
        resolve(resultCode); // (C)
      }).catch(err => {
        reject(err); // (D)
      })
    });
  }
  ···
}
```

If you look closely, you can see that the result Promise is mainly used to forward the fulfillment (line C) and the rejection (line D) of the asynchronous method call `db.insert()` (line B).

The fix is to not create a Promise, by relying on `then()` and chaining:

```
class Model {
  insertInto(db) {
    return db.insert(this.fields) // (A)
    .then(resultCode => {
      this.notifyObservers({event: 'created', model: this});
      return resultCode; // (B)
    });
  }
  ···
}
```

Explanations:

- We return `resultCode` (line B) and let `then()` create the Promise for us.
- We return the Promise chain (line A) and `then()` will pass on any rejection produced by `db.insert()`.

25.9.4 Mistake: using `then()` for error handling

In principle, `catch(cb)` is an abbreviation for `then(null, cb)`. But using both parameters of `then()` at the same time can cause problems:

```
// Don’t do this
asyncFunc1()
.then(
  value => { // (A)
    doSomething(); // (B)
    return asyncFunc2(); // (C)
  },
  error => { // (D)
    ...
  });
```

The rejection callback (line D) receives all rejections of `asyncFunc1()`, but it does not receive rejections created by the fulfillment callback (line A). For example, the synchronous function call in line B may throw an exception or the asynchronous function call in line C may produce a rejection.

Therefore, it is better to move the rejection callback to a chained `catch()`:

```
asyncFunc1()
.then(value => {
  doSomething();
  return asyncFunc2();
})
.catch(error => {
  ...
});
```

25.10 Tips for error handling

25.10.1 Operational errors versus programmer errors

In programs, there are two kinds of errors:

- *Operational errors* happen when a correct program encounters an exceptional situation that requires deviating from the “normal” algorithm. For example, a storage device may run out of memory while the program is writing data to it. This kind of error is expected.
- *Programmer errors* happen when code does something wrong. For example, a function may require a parameter to be a string, but receives a number. This kind of error is unexpected.

25.10.1.1 Operational errors: don’t mix rejections and exceptions

For operational errors, each function should support exactly one way of signaling errors. For Promise-based functions that means not mixing rejections and exceptions, which is the same as saying that they shouldn't throw exceptions.

25.10.1.2 Programmer errors: fail quickly

For programmer errors, it can make sense to fail as quickly as possible, by throwing an exception:

```
function downloadFile(url) {
    if (typeof url !== 'string') {
        throw new Error('Illegal argument: ' + url);
    }
    return new Promise(···).
}
```

If you do this, you must make sure that your asynchronous code can handle exceptions. I find throwing exceptions acceptable for assertions and similar things that could, in theory, be checked statically (e.g. via a linter that analyzes the source code).

25.10.2 Handling exceptions in Promise-based functions

If exceptions are thrown inside the callbacks of `then()` and `catch()` then that's not a problem, because these two methods convert them to rejections.

However, things are different if you start your async function by doing something synchronous:

```
function asyncFunc() {
    doSomethingSync(); // (A)
    return doSomethingAsync()
    .then(result => {
        ···
    });
}
```

If an exception is thrown in line A then the whole function throws an exception. There are two solutions to this problem.

25.10.2.1 Solution 1: returning a rejected Promise

You can catch exceptions and return them as rejected Promises:

```
function asyncFunc() {
  try {
    doSomethingSync();
    return doSomethingAsync()
    .then(result => {
      ···
    });
  } catch (err) {
    return Promise.reject(err);
  }
}
```

25.10.2.2 Solution 2: executing the sync code inside a callback

You can also start a chain of `then()` method calls via `Promise.resolve()` and execute the synchronous code inside a callback:

```
function asyncFunc() {
  return Promise.resolve()
  .then(() => {
    doSomethingSync();
    return doSomethingAsync();
  })
  .then(result => {
    ···
  });
}
```

An alternative is to start the Promise chain via the Promise constructor:

```
function asyncFunc() {
  return new Promise((resolve, reject) => {
    doSomethingSync();
    resolve(doSomethingAsync());
  })
  .then(result => {
    ···
  });
}
```

This approach saves you a tick (the synchronous code is executed right away), but it makes your code less regular.

25.10.3 Further reading

Sources of this section:

- Chaining:
    "Promise Anti-patterns" on Tao of Code.
- Error handling:
    - "Error Handling in Node.js" by Joyent
    - A post by user Mörre Noseshine in the "Exploring ES6" Google Group
    - Feedback to asking whether it is OK to throw exceptions from Promise-based functions.

25.11 Composing Promises

Composing means creating new things out of existing pieces. We have already encountered sequential composition of Promises: Given two Promises P and Q, the following code produces a new Promise that executes Q after P is fulfilled.

```
P.then(() => Q)
```

Note that this is similar to the semicolon for synchronous code: Sequential composition of the synchronous operations `f()` and `g()` looks as follows.

```
f(); g()
```

This section describes additional ways of composing Promises.

25.11.1 Manually forking and joining computations

Let's assume you want to perform two asynchronous computations, `asyncFunc1()` and `asyncFunc2()` in parallel:

```
// Don’t do this
asyncFunc1()
.then(result1 => {
   handleSuccess({result1});
});
.catch(handleError);

asyncFunc2()
.then(result2 => {
   handleSuccess({result2});
})
.catch(handleError);

const results = {};
function handleSuccess(props) {
   Object.assign(results, props);
   if (Object.keys(results).length === 2) {
      const {result1, result2} = results;
      ...
   }
}
let errorCounter = 0;
function handleError(err) {
   errorCounter++;
   if (errorCounter === 1) {
      // One error means that everything failed,
      // only react to first error
      ...
   }
}
```

The two function calls `asyncFunc1()` and `asyncFunc2()` are made without `then()` chaining. As a consequence, they are both executed immediately and more or less in parallel. Execution is now forked; each function call spawned a separate “thread”. Once both threads are finished (with a result or an error), execution is joined into a single thread in either `handleSuccess()` or `handleError()`.

The problem with this approach is that it involves too much manual and error-prone work. The fix is to not do this yourself, by relying on the built-in method `Promise.all()`.

25.11.2 Forking and joining computations via `Promise.all()`

`Promise.all(iterable)` takes an iterable over Promises (thenables and other values are converted to Promises via `Promise.resolve()`). Once all of them are fulfilled, it fulfills with an Array of their values. If `iterable` is empty, the Promise returned by `all()` is fulfilled immediately.

```
Promise.all([
    asyncFunc1(),
    asyncFunc2(),
])
.then(([result1, result2]) => {
    ···
})
.catch(err => {
    // Receives first rejection among the Promises
    ···
});
```

25.11.3 `map()` via `Promise.all()`

One nice thing about Promises is that many synchronous tools still work, because Promise-based functions return results. For example, you can use the Array method `map()`:

```
const fileUrls = [
    'http://example.com/file1.txt',
    'http://example.com/file2.txt',
];
const promisedTexts = fileUrls.map(httpGet);
```

`promisedTexts` is an Array of Promises. We can use `Promise.all()`, which we have already encountered in the previous section, to convert that Array to a Promise that fulfills with an Array of results.

```
Promise.all(promisedTexts)
.then(texts => {
    for (const text of texts) {
        console.log(text);
    }
})
.catch(reason => {
    // Receives first rejection among the Promises
});
```

25.11.4 Timing out via `Promise.race()`

`Promise.race(iterable)` takes an iterable over Promises (thenables and other values are converted to Promises via `Promise.resolve()`) and returns a Promise P. The first of the input Promises that is settled passes its settlement on to the output Promise. If `iterable` is empty then the Promise returned by `race()` is never settled.

As an example, let's use `Promise.race()` to implement a timeout:

```
Promise.race([
    httpGet('http://example.com/file.txt'),
    delay(5000).then(function () {
        throw new Error('Timed out')
    });
])
.then(function (text) { ··· })
.catch(function (reason) { ··· });
```

25.12 Two useful additional Promise methods

This section describes two useful methods for Promises that many Promise libraries provide. They are only shown to further demonstrate Promises, you should not add them to `Promise.prototype` (this kind of patching should only be done by polyfills).

25.12.1 `done()`

When you chain several Promise method calls, you risk silently discarding errors. For example:

```
function doSomething() {
    asyncFunc()
    .then(f1)
    .catch(r1)
    .then(f2); // (A)
}
```

If `then()` in line A produces a rejection, it will never be handled anywhere. The Promise library Q provides a method `done()`, to be used as the last element in a chain of method calls. It either replaces the last `then()` (and has one to two arguments):

```
function doSomething() {
  asyncFunc()
  .then(f1)
  .catch(r1)
  .done(f2);
}
```

Or it is inserted after the last `then()` (and has zero arguments):

```
function doSomething() {
  asyncFunc()
  .then(f1)
  .catch(r1)
  .then(f2)
  .done();
}
```

Quoting the Q documentation:

> The Golden Rule of `done` versus `then` usage is: either return your promise to someone else, or if the chain ends with you, call `done` to terminate it. Terminating with `catch` is not sufficient because the catch handler may itself throw an error.

This is how you would implement `done()` in ECMAScript 6:

```
Promise.prototype.done = function (onFulfilled, onRejected) {
  this.then(onFulfilled, onRejected)
  .catch(function (reason) {
    // Throw an exception globally
    setTimeout(() => { throw reason }, 0);
  });
};
```

While `done`'s functionality is clearly useful, it has not been added to ECMAScript 6. The idea was to first explore how much engines can detect automatically. Depending on how well that works, it may to be necessary to introduce `done()`.

25.12.2 `finally()`

Sometimes you want to perform an action independently of whether an error happened or not. For example, to clean up after you are done with a resource. That's what the Promise method `finally()` is for, which works much like the finally clause in exception handling. Its callback receives no arguments, but is notified of either a resolution or a rejection.

```
createResource(···)
.then(function (value1) {
    // Use resource
})
.then(function (value2) {
    // Use resource
})
.finally(function () {
    // Clean up
});
```

This is how Domenic Denicola proposes to implement finally():

```
Promise.prototype.finally = function (callback) {
    const P = this.constructor;
    // We don't invoke the callback in here,
    // because we want then() to handle its exceptions
    return this.then(
        // Callback fulfills => continue with receiver's fulfillment or rejec\
tion
        // Callback rejects => pass on that rejection (then() has no 2nd para\
meter!)
        value  => P.resolve(callback()).then(() => value),
        reason => P.resolve(callback()).then(() => { throw reason })
    );
};
```

The callback determines how the settlement of the receiver (this) is handled:

- If the callback throws an exception or returns a rejected Promise then that becomes/contributes the rejection value.
- Otherwise, the settlement (fulfillment or rejection) of the receiver becomes the settlement of the Promise returned by finally() . In a way, we take finally() out of the chain of methods.

Example 1 (by Jake Archibald): using finally() to hide a spinner. Simplified version:

```
showSpinner();
fetchGalleryData()
.then(data => updateGallery(data))
.catch(showNoDataError)
.finally(hideSpinner);
```

Example 2 (by Kris Kowal): using `finally()` to tear down a test.

```
const HTTP = require("q-io/http");
const server = HTTP.Server(app);
return server.listen(0)
.then(function () {
    // run test
})
.finally(server.stop);
```

25.13 Node.js: using callback-based sync functions with Promises

The Promise library Q has tool functions for interfacing with Node.js-style `(err, result)` callback APIs. For example, `denodeify` converts a callback-based function to a Promise-based one:

```
const readFile = Q.denodeify(FS.readFile);

readFile('foo.txt', 'utf-8')
.then(function (text) {
    ...
});
```

denodify is a micro-library that only provides the functionality of `Q.denodeify()` and complies with the ECMAScript 6 Promise API.

25.14 ES6-compatible Promise libraries

There are many Promise libraries out there. The following ones conform to the ECMAScript 6 API, which means that you can use them now and easily migrate to native ES6 later.

Minimal polyfills:

- "ES6-Promises" by Jake Archibald extracts just the ES6 API out of RSVP.js.
- "Native Promise Only (NPO)" by Kyle Simpson is "a polyfill for native ES6 promises, as close as possible (no extensions) to the strict spec

definitions”.
- “Lie” by Calvin Metcalf is “a small, performant, promise library implementing the Promises/A+ spec”.

Larger Promise libraries:

- “RSVP.js” by Stefan Penner is a superset of the ES6 Promise API.
- “Bluebird” by Petka Antonov is a popular Promises library that passes the ES2015 tests (Test262) and is thus an alternative to ES6 Promises.
- Q.Promise by Kris Kowal implements the ES6 API.

ES6 standard library polyfills:

- “ES6 Shim” by Paul Millr includes `Promise`.
- “core-js” by Denis Pushkarev, the ES6+ polyfill used by Babel, includes `Promise`.

25.15 Next step: using Promises via generators

Implementing asynchronous functions via Promises is more convenient than via events or callbacks, but it’s still not ideal:

- Asynchronous code and synchronous code work completely differently. As a consequence, mixing those execution styles and switching between them for a function or method is cumbersome.
- Conceptually, invoking an asynchronous function is a blocking call: The code making the call is suspended during the asynchronous computation and resumed once the result is in. However, the code does not reflect this as much as it could.

The solution is to bring blocking calls to JavaScript. Generators let us do that, via libraries: In the following code, I use the control flow library co to asynchronously retrieve two JSON files.

```
co(function* () {
    try {
        const [croftStr, bondStr] = yield Promise.all([  // (A)
            getFile('http://localhost:8000/croft.json'),
            getFile('http://localhost:8000/bond.json'),
        ]);
        const croftJson = JSON.parse(croftStr);
        const bondJson = JSON.parse(bondStr);

        console.log(croftJson);
        console.log(bondJson);
    } catch (e) {
        console.log('Failure to read: ' + e);
    }
});
```

In line A, execution blocks (waits) via `yield` until the result of `Promise.all()` is ready. That means that the code looks synchronous while performing asynchronous operations.

Details are explained in the chapter on generators.

25.16 Promises in depth: a simple implementation

In this section, we will approach Promises from a different angle: Instead of learning how to use the API, we will look at a simple implementation of it. This different angle helped me greatly with making sense of Promises.

The Promise implementation is called `DemoPromise`. In order to be easier to understand, it doesn't completely match the API. But it is close enough to still give you much insight into the challenges that actual implementations face.

`DemoPromise` is available on GitHub, in the repository demo_promise.

`DemoPromise` is a class with three prototype methods:

- `DemoPromise.prototype.resolve(value)`
- `DemoPromise.prototype.reject(reason)`
- `DemoPromise.prototype.then(onFulfilled, onRejected)`

That is, `resolve` and `reject` are methods (versus functions handed to a callback parameter of the constructor).

25.16.1 A stand-alone Promise

Our first implementation is a stand-alone Promise with minimal functionality:

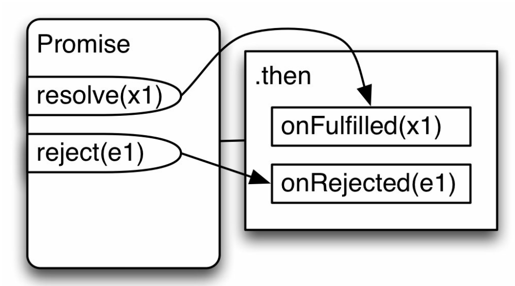
- You can create a Promise.
- You can resolve or reject a Promise and you can only do it once.
- You can register *reactions* (callbacks) via `then()`. It must work independently of whether the Promise has already been settled or not.
  - This method does not support chaining, yet – it does not return anything.

This is how this first implementation is used:

```
const dp = new DemoPromise();
dp.resolve('abc');
dp.then(function (value) {
    console.log(value); // abc
});
```

The following diagram illustrates how our first `DemoPromise` works:

25.16.1.1 `DemoPromise.prototype.then()`



Let's examine `then()` first. It has to handle two cases:

- If the Promise is still pending, it queues invocations of `onFulfilled` and `onRejected`, to be used when the Promise is settled.
- If the Promise is already fulfilled or rejected, `onFulfilled` or `onRejected` can be invoked right away.

```
then(onFulfilled, onRejected) {
  const self = this;
  const fulfilledTask = function () {
    onFulfilled(self.promiseResult);
  };
  const rejectedTask = function () {
    onRejected(self.promiseResult);
  };
  switch (this.promiseState) {
    case 'pending':
      this.fulfillReactions.push(fulfilledTask);
      this.rejectReactions.push(rejectedTask);
      break;
    case 'fulfilled':
      addToTaskQueue(fulfilledTask);
      break;
    case 'rejected':
      addToTaskQueue(rejectedTask);
      break;
  }
}
```

The previous code snippet uses the following helper function:

```
function addToTaskQueue(task) {
  setTimeout(task, 0);
}
```

25.16.1.2 `DemoPromise.prototype.resolve()`

`resolve()` works as follows: If the Promise is already settled, it does nothing (ensuring that a Promise can only be settled once). Otherwise, the state of the Promise changes to `'fulfilled'` and the result is cached in `this.promiseResult`. Next, all fulfillment reactions, that have been enqueued so far, are be triggered.

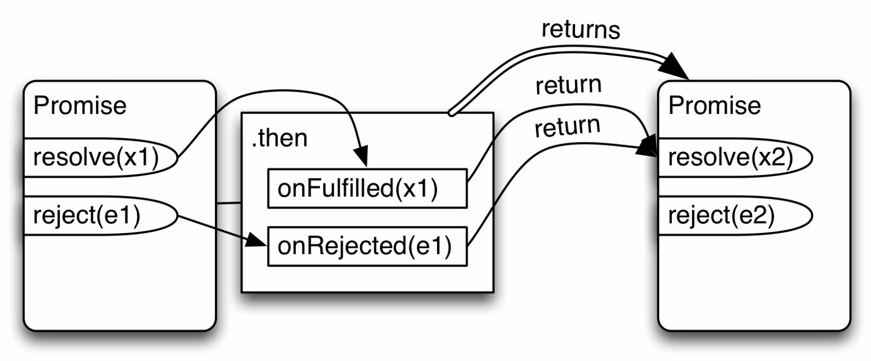
```
resolve(value) {
    if (this.promiseState !== 'pending') return;
    this.promiseState = 'fulfilled';
    this.promiseResult = value;
    this._clearAndEnqueueReactions(this.fulfillReactions);
    return this; // enable chaining
}
_clearAndEnqueueReactions(reactions) {
    this.fulfillReactions = undefined;
    this.rejectReactions = undefined;
    reactions.map(addToTaskQueue);
}
```

`reject()` is similar to `resolve()`.

25.16.2 Chaining

The next feature we implement is chaining:

- `then()` returns a Promise that is resolved with what either `onFulfilled` or `onRejected` return.
- If `onFulfilled` or `onRejected` are missing, whatever they would have received is passed on to the Promise returned by `then()`.



Obviously, only `then()` changes:

```js
then(onFulfilled, onRejected) {
  const returnValue = new Promise(); // (A)
  const self = this;

  let fulfilledTask;
  if (typeof onFulfilled === 'function') {
    fulfilledTask = function () {
      const r = onFulfilled(self.promiseResult);
      returnValue.resolve(r); // (B)
    };
  } else {
    fulfilledTask = function () {
      returnValue.resolve(self.promiseResult); // (C)
    };
  }

  let rejectedTask;
  if (typeof onRejected === 'function') {
    rejectedTask = function () {
      const r = onRejected(self.promiseResult);
      returnValue.resolve(r); // (D)
    };
  } else {
    rejectedTask = function () {
      // `onRejected` has not been provided
      // => we must pass on the rejection
      returnValue.reject(self.promiseResult); // (E)
    };
  }
  ···
  return returnValue; // (F)
}
```

`then()` creates and returns a new Promise (lines A and F). Additionally, `fulfilledTask` and `rejectedTask` are set up differently: After a settlement…

- The result of `onFulfilled` is used to resolve `returnValue` (line B).
  - If `onFulfilled` is missing, we use the fulfillment value to resolve `returnValue` (line C).
- The result of `onRejected` is used to resolve (not reject!) `returnValue` (line D).
  - If `onRejected` is missing, we use pass on the rejection value to `returnValue` (line E).

25.16.3 Flattening

Flattening is mostly about making chaining more convenient: Normally, returning a value from a reaction passes it on to the next `then()`. If we return a Promise, it would be nice if it could be “unwrapped” for us, like in the following example:

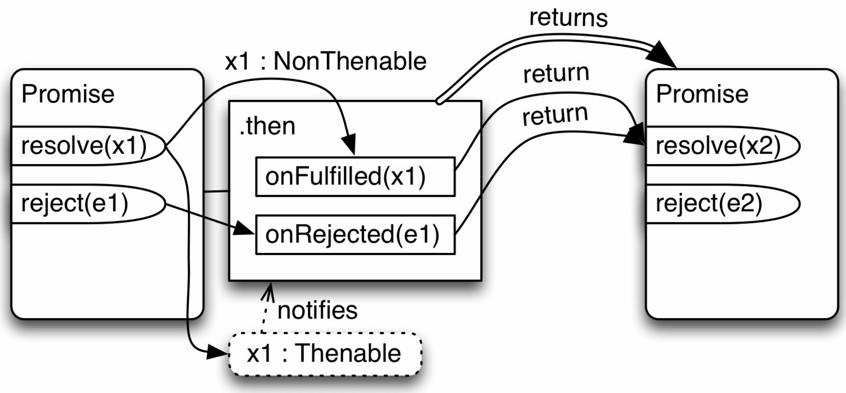
```
asyncFunc1()
.then(function (value1) {
    return asyncFunc2(); // (A)
})
.then(function (value2) {
    // value2 is fulfillment value of asyncFunc2() Promise
    console.log(value2);
});
```

We returned a Promise in line A and didn’t have to nest a call to `then()` inside the current method, we could invoke `then()` on the method’s result. Thus: no nested `then()`, everything remains flat.

We implement this by letting the `resolve()` method do the flattening:

- Resolving a Promise P with a Promise Q means that Q’s settlement is forwarded to P’s reactions.
- P becomes “locked in” on Q: it can’t be resolved (incl. rejected), anymore. And its state and result are always the same as Q’s.

We can make flattening more generic if we allow Q to be a thenable (instead of only a Promise).



To implement locking-in, we introduce a new boolean flag `this.alreadyResolved`. Once it is true, `this` is locked and can't be resolved anymore. Note that `this` may still be pending, because its state is now the same as the Promise it is locked in on.

```
resolve(value) {
  if (this.alreadyResolved) return;
  this.alreadyResolved = true;
  this._doResolve(value);
  return this; // enable chaining
}
```

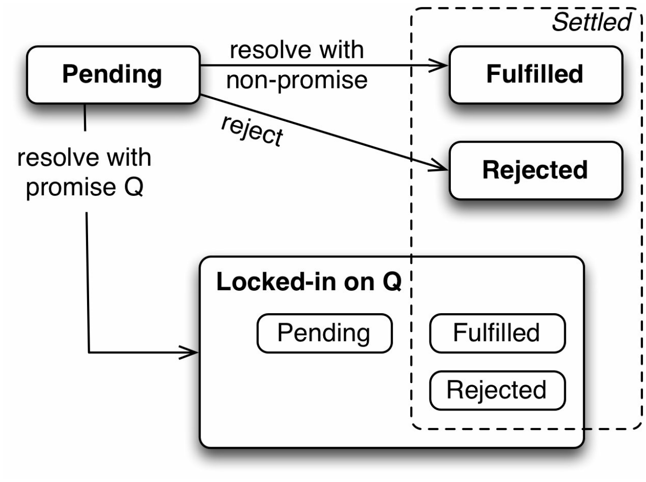
The actual resolution now happens in the private method `_doResolve()`:

```
_doResolve(value) {
  const self = this;
  // Is `value` a thenable?
  if (typeof value === 'object' && value !== null && 'then' in value) {
    // Forward fulfillments and rejections from `value` to `this`.
    // Added as a task (versus done immediately) to preserve async semant\
ics.
    addToTaskQueue(function () { // (A)
      value.then(
        function onFulfilled(result) {
          self._doResolve(result);
        },
        function onRejected(error) {
          self._doReject(error);
        });
    });
  } else {
    this.promiseState = 'fulfilled';
    this.promiseResult = value;
    this._clearAndEnqueueReactions(this.fulfillReactions);
  }
}
```

The flattening is performed in line A: If `value` is fulfilled, we want `self` to be fulfilled and if `value` is rejected, we want `self` to be rejected. The forwarding happens via the private methods `_doResolve` and `_doReject`, to get around the protection via `alreadyResolved`.

25.16.4 Promise states in more detail

With chaining, the states of Promises become more complex (as covered by Sect. 25.4 of the ECMAScript 6 specification):



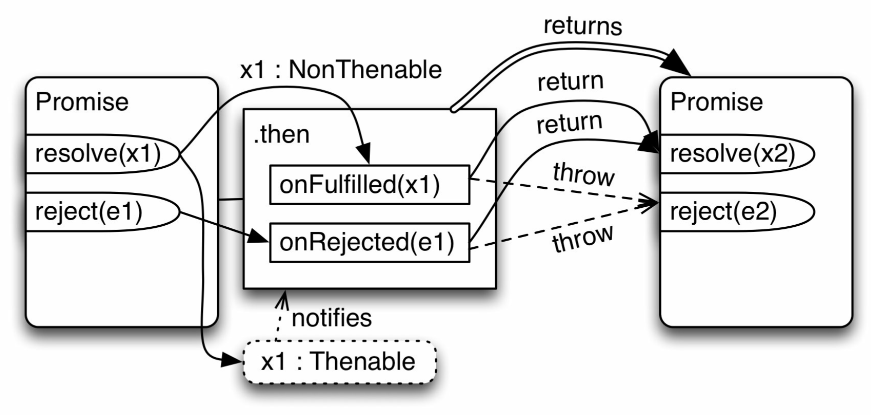
If you are only *using* Promises, you can normally adopt a simplified worldview and ignore locking-in. The most important state-related concept remains "settledness": a Promise is settled if it is either fulfilled or rejected. After a Promise is settled, it doesn't change, anymore (state and fulfillment or rejection value).

If you want to *implement* Promises then "resolving" matters, too and is now harder to understand:

- Intuitively, "resolved" means "can't be (directly) resolved anymore". A Promise is resolved if it is either settled or locked in. Quoting the spec: "An unresolved Promise is always in the pending state. A resolved Promise may be pending, fulfilled or rejected."
- Resolving does not necessarily lead to settling: you can resolve a Promise with another one that is always pending.
- Resolving now includes rejecting (i.e., it is more general): you can reject a Promise by resolving it with a rejected Promise.

25.16.5 Exceptions

As our final feature, we'd like our Promises to handle exceptions in user code as rejections. For now, "user code" means the two callback parameters of `then()`.

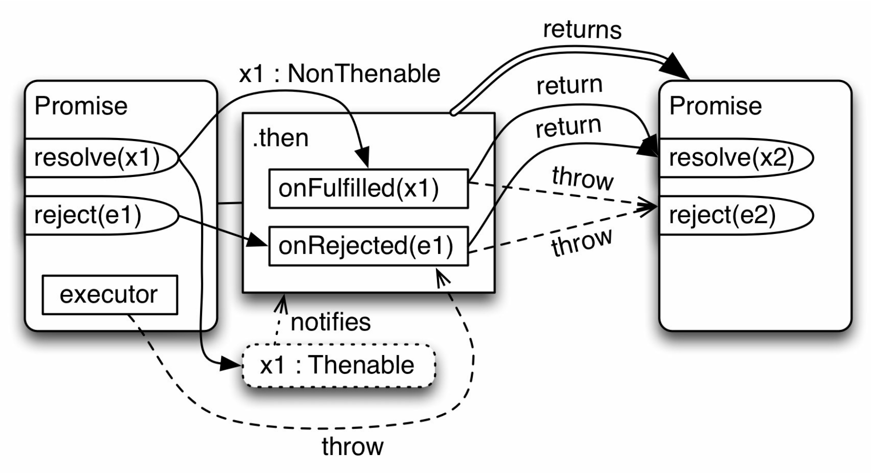


The following excerpt shows how we turn exceptions inside `onFulfilled` into rejections – by wrapping a `try-catch` around its invocation in line A.

```
then(onFulfilled, onRejected) {
    ···
    let fulfilledTask;
    if (typeof onFulfilled === 'function') {
        fulfilledTask = function () {
            try {
                const r = onFulfilled(self.promiseResult); // (A)
                returnValue.resolve(r);
            } catch (e) {
                returnValue.reject(e);
            }
        };
    } else {
        fulfilledTask = function () {
            returnValue.resolve(self.promiseResult);
        };
    }
    ···
}
```

25.16.6 Revealing constructor pattern

If we wanted to turn `DemoPromise` into an actual Promise implementation, we'd still need to implement the revealing constructor pattern [2]: ES6 Promises are not resolved and rejected via methods, but via functions that are handed to the *executor*, the callback parameter of the constructor.



If the executor throws an exception then “its” Promise must be rejected.

25.17 Advantages and limitations of Promises

25.17.1 Advantages of Promises

25.17.1.1 Unifying asynchronous APIs

One important advantage of Promises is that they will increasingly be used by asynchronous browser APIs and unify currently diverse and incompatible patterns and conventions. Let’s look at two upcoming Promise-based APIs.

The fetch API is a Promise-based alternative to XMLHttpRequest:

```
fetch(url)
.then(request => request.text())
.then(str => ···)
```

`fetch()` returns a Promise for the actual request, `text()` returns a Promise for the content as a string.

The ECMAScript 6 API for programmatically importing modules is based on Promises, too:

```
System.import('some_module.js')
.then(some_module => {
    ···
})
```

25.17.1.2 Promises versus events

Compared to events, Promises are better for handling one-off results. It doesn't matter whether you register for a result before or after it has been computed, you will get it. This advantage of Promises is fundamental in nature. On the flip side, you can't use them for handling recurring events. Chaining is another advantage of Promises, but one that could be added to event handling.

25.17.1.3 Promises versus callbacks

Compared to callbacks, Promises have cleaner function (or method) signatures. With callbacks, parameters are used for input and output:

```
fs.readFile(name, opts?, (err, string | Buffer) => void)
```

With Promises, all parameters are used for input:

```
readFilePromisified(name, opts?) : Promise<string | Buffer>
```

Additional Promise advantages include:

- Unified handling of both asynchronous errors and normal exceptions.
- Easier composition, because you can reuse synchronous tools such as `Array.prototype.map()`.
- Chaining of `then()` and `catch()`.
- Guarding against notifying callbacks more than once. Some development environments also warn about rejections that are never handled.

25.17.2 Promises are not always the best choice

Promises work well for for single asynchronous results. They are not suited for:

- Recurring events: If you are interested in those, take a look at reactive programming, which add a clever way of chaining to normal event handling.
- Streams of data: A standard for supporting those is currently in development.

ECMAScript 6 Promises lack two features that are sometimes useful:

- You can't cancel them.
- You can't query them for how far along they are (e.g. to display a progress bar in a client-side user interface).

The Q Promise library has support for the latter and there are plans to add both capabilities to Promises/A+.

25.18 Reference: the ECMAScript 6 Promise API

This section gives an overview of the ECMAScript 6 Promise API, as described in the specification.

25.18.1 `Promise` constructor

The constructor for Promises is invoked as follows:

```
const p = new Promise(function (resolve, reject) { ··· });
```

The callback of this constructor is called an *executor*. The executor can use its parameters to resolve or reject the new Promise `p`:

- `resolve(x)` resolves `p` with `x`:
    - If `x` is thenable, its settlement is forwarded to `p` (which includes triggering reactions registered via `then()`).
    - Otherwise, `p` is fulfilled with `x`.
- `reject(e)` rejects `p` with the value `e` (often an instance of Error).

25.18.2 Static `Promise` methods

25.18.2.1 Creating Promises

The following two static methods create new instances of their receivers:

- `Promise.resolve(x)`: converts arbitrary values to Promises, with an awareness of Promises.
    - If the constructor of `x` is the receiver, `x` is returned unchanged.
    - Otherwise, return a new instance of the receiver that is fulfilled with `x`.
- `Promise.reject(reason)`: creates a new instance of the receiver that is rejected with the value `reason`.

25.18.2.2 Composing Promises

Intuitively, the static methods `Promise.all()` and `Promise.race()` compose iterables of Promises to a single Promise. That is:

- They take an iterable. The elements of the iterable are converted to Promises via `this.resolve()`.
- They return a new Promise. That Promise is a new instance of the receiver.

The methods are:

- `Promise.all(iterable)`: returns a Promise that...
    - is fulfilled if all elements in `iterable` are fulfilled.
      Fulfillment value: Array with fulfillment values.
    - is rejected if any of the elements are rejected.
      Rejection value: first rejection value.
- `Promise.race(iterable)`: the first element of `iterable` that is settled is used to settle the returned Promise.

25.18.3 `Promise.prototype` methods

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

- The callbacks `onFulfilled` and `onRejected` are called *reactions*.
- `onFulfilled` is called immediately if the Promise is already fulfilled or as soon as it becomes fulfilled. Similarly, `onRejected` is informed of rejections.
- `then()` returns a new Promise Q (created via the species of the constructor of the receiver):
    - If either of the reactions returns a value, Q is resolved with it.
    - If either of the reactions throws an exception, Q is rejected with it.
- Omitted reactions:
    - If `onFulfilled` has been omitted, a fulfillment of the receiver is forwarded to the result of `then()`.
    - If `onRejected` has been omitted, a rejection of the receiver is forwarded to the result of `then()`.

Default values for omitted reactions could be implemented like this:

```
function defaultOnFulfilled(x) {
    return x;
}
function defaultOnRejected(e) {
    throw e;
}
```

25.18.3.2 `Promise.prototype.catch(onRejected)`

`p.catch(onRejected)` is the same as `p.then(null, onRejected)`.

25.19 Further reading

[1] "Promises/A+", edited by Brian Cavalier and Domenic Denicola (the de-facto standard for JavaScript Promises)

[2] "The Revealing Constructor Pattern" by Domenic Denicola (this pattern is used by the `Promise` constructor)

Next: VI Miscellaneous