You Don't Know JS: Async & Performance

Chapter 3: Promises

In Chapter 2, we identified two major categories of deficiencies with using callbacks to express program asynchrony and manage concurrency: lack of sequentiality and lack of trustability. Now that we understand the problems more intimately, it's time we turn our attention to patterns that can address them.

The issue we want to address first is the *inversion of control*, the trust that is so fragilely held and so easily lost.

Recall that we wrap up the *continuation* of our program in a callback function, and hand that callback over to another party (potentially even external code) and just cross our fingers that it will do the right thing with the invocation of the callback.

We do this because we want to say, "here's what happens *later*, after the current step finishes."

But what if we could uninvert that *inversion of control*? What if instead of handing the continuation of our program to another party, we could expect it to return us a capability to know when its task finishes, and then our code could decide what to do next?

This paradigm is called **Promises**.

Promises are starting to take the JS world by storm, as developers and specification writers alike desperately seek to untangle the insanity of callback hell in their code/design. In fact, most new async APIs being added to JS/DOM platform are being built on Promises. So it's probably a good idea to dig in and learn them, don't you think!?

Note: The word "immediately" will be used frequently in this chapter, generally to refer to some Promise resolution action. However, in essentially all cases, "immediately" means in terms of the Job queue behavior (see Chapter 1), not in the strictly synchronous *now* sense.

What Is a Promise?

When developers decide to learn a new technology or pattern, usually their first step is "Show me the code!" It's quite natural for us to just jump in feet first and learn as we go.

But it turns out that some abstractions get lost on the APIs alone. Promises are one of those tools where it can be painfully obvious from how someone uses it whether they understand what it's for and about versus just learning and using the API.

So before I show the Promise code, I want to fully explain what a Promise really is conceptually. I hope this will then guide you better as you explore integrating Promise theory into your own async flow.

With that in mind, let's look at two different analogies for what a Promise *is*.

Future Value

Imagine this scenario: I walk up to the counter at a fast-food restaurant, and place an order for a cheeseburger. I hand the cashier $1.47. By placing my order and paying for it, I've made a request for a *value* back (the cheeseburger). I've started a transaction.

But often, the cheeseburger is not immediately available for me. The cashier hands me something in place of my cheeseburger: a receipt with an order number on it. This order number is an IOU ("I owe you") *promise* that ensures that eventually, I should receive my cheeseburger.

So I hold onto my receipt and order number. I know it represents my *future cheeseburger*, so I don't need to worry about it anymore -- aside from being hungry!

While I wait, I can do other things, like send a text message to a friend that says, "Hey, can you come join me for lunch? I'm going to eat a cheeseburger."

I am reasoning about my *future cheeseburger* already, even though I don't have it in my hands yet. My brain is able to do this because it's treating the order number as a placeholder for the cheeseburger. The placeholder essentially makes the value *time independent*. It's a **future value**.

Eventually, I hear, "Order 113!" and I gleefully walk back up to the counter with receipt in hand. I hand my receipt to the cashier, and I take my cheeseburger in return.

In other words, once my *future value* was ready, I exchanged my value-promise for the value itself.

But there's another possible outcome. They call my order number, but when I go to retrieve my cheeseburger, the cashier regretfully informs me, "I'm sorry, but we appear to be all out of cheeseburgers." Setting aside the customer frustration of this scenario for a moment, we can see an important characteristic of *future values*: they can either indicate a success or failure.

Every time I order a cheeseburger, I know that I'll either get a cheeseburger eventually, or I'll get the sad news of the cheeseburger shortage, and I'll have to figure out something else to eat for lunch.

Note: In code, things are not quite as simple, because metaphorically the order number may never be called, in which case we're left indefinitely in an unresolved state. We'll come back to dealing with that case later.

Values Now and Later

This all might sound too mentally abstract to apply to your code. So let's be more concrete.

However, before we can introduce how Promises work in this fashion, we're going to derive in code that we already understand -- callbacks! -- how to handle these *future values*.

When you write code to reason about a value, such as performing math on a `number`, whether you realize it or not, you've been assuming something very fundamental about that value, which is that it's a concrete *now* value already:

```
var x, y = 2;

console.log( x + y ); // NaN  <-- because `x` isn't set yet
```

The `x + y` operation assumes both `x` and `y` are already set. In terms we'll expound on shortly, we assume the `x` and `y` values are already *resolved*.

It would be nonsense to expect that the `+` operator by itself would somehow be magically capable of detecting and waiting around until both `x` and `y` are resolved (aka ready), only then to do the operation. That would cause chaos in the program if different statements finished *now* and others finished *later*, right?

How could you possibly reason about the relationships between two statements if either one (or both) of them might not be finished yet? If statement 2 relies on statement 1 being finished, there are just two outcomes: either statement 1 finished right *now* and everything proceeds fine, or statement 1 didn't finish yet, and thus statement 2 is going to fail.

If this sort of thing sounds familiar from Chapter 1, good!

Let's go back to our `x + y` math operation. Imagine if there was a way to say, "Add `x` and `y`, but if either of them isn't ready yet, just wait until they are. Add them as soon as you can."

Your brain might have just jumped to callbacks. OK, so...

```
function add(getX,getY,cb) {
	var x, y;
	getX( function(xVal){
		x = xVal;
		// both are ready?
		if (y != undefined) {
			cb( x + y );	// send along sum
		}
```

```
        } );
        getY( function(yVal){
                y = yVal;
                // both are ready?
                if (x != undefined) {
                        cb( x + y );    // send along sum
                }
        } );
}

// `fetchX()` and `fetchY()` are sync or async
// functions
add( fetchX, fetchY, function(sum){
        console.log( sum ); // that was easy, huh?
} );
```

Take just a moment to let the beauty (or lack thereof) of that snippet sink in (whistles patiently).

While the ugliness is undeniable, there's something very important to notice about this async pattern.

In that snippet, we treated `x` and `y` as future values, and we express an operation `add(..)` that (from the outside) does not care whether `x` or `y` or both are available right away or not. In other words, it normalizes the *now* and *later*, such that we can rely on a predictable outcome of the `add(..)` operation.

By using an `add(..)` that is temporally consistent -- it behaves the same across *now* and *later* times -- the async code is much easier to reason about.

To put it more plainly: to consistently handle both *now* and *later*, we make both of them *later*: all operations become async.

Of course, this rough callbacks-based approach leaves much to be desired. It's just a first tiny step toward realizing the benefits of reasoning about *future values* without worrying about the time aspect of when it's available or not.

Promise Value

We'll definitely go into a lot more detail about Promises later in the chapter -- so don't worry if some of this is confusing -- but let's just briefly glimpse at how we can express the `x + y` example via `Promise`s:

```
function add(xPromise,yPromise) {
        // `Promise.all([ .. ])` takes an array of promises,
        // and returns a new promise that waits on them
        // all to finish
        return Promise.all( [xPromise, yPromise] )

        // when that promise is resolved, let's take the
        // received `X` and `Y` values and add them together.
        .then( function(values){
                // `values` is an array of the messages from the
                // previously resolved promises
                return values[0] + values[1];
        } );
}

// `fetchX()` and `fetchY()` return promises for
// their respective values, which may be ready
// *now* or *later*.
add( fetchX(), fetchY() )

// we get a promise back for the sum of those
// two numbers.
// now we chain-call `then(..)` to wait for the
// resolution of that returned promise.
.then( function(sum){
        console.log( sum ); // that was easier!
} );
```

There are two layers of Promises in this snippet.

`fetchX()` and `fetchY()` are called directly, and the values they return (promises!) are passed into `add(..)`. The underlying values those promises represent may be ready *now* or *later*, but each promise normalizes the behavior to be the same regardless. We reason about `X` and `Y` values in a time-independent way. They are *future values*.

The second layer is the promise that `add(..)` creates (via `Promise.all([ .. ])`) and returns, which we wait on by calling `then(..)`. When the `add(..)` operation completes, our `sum` *future value* is ready and we can print it out. We hide inside of `add(..)` the logic for waiting on the `X` and `Y` *future values*.

Note: Inside `add(..)`, the `Promise.all([ .. ])` call creates a promise (which is waiting on `promiseX` and `promiseY` to resolve). The chained call to `.then(..)` creates another promise, which the `return values[0] + values[1]` line immediately resolves (with the result of the addition). Thus, the `then(..)` call we chain off the end of the `add(..)` call -- at the end of the snippet -- is actually operating on that second promise returned, rather than the first one created by `Promise.all([ .. ])`. Also, though we are not chaining off the end of that second `then(..)`, it too has created another promise, had we chosen to observe/use it. This Promise chaining stuff will be explained in much greater detail later in this chapter.

Just like with cheeseburger orders, it's possible that the resolution of a Promise is rejection instead of fulfillment. Unlike a fulfilled Promise, where the value is always programmatic, a rejection value -- commonly called a "rejection reason" -- can either be set directly by the program logic, or it can result implicitly from a runtime exception.

With Promises, the `then(..)` call can actually take two functions, the first for fulfillment (as shown earlier), and the second for rejection:

```
add( fetchX(), fetchY() )
.then(
	// fullfillment handler
	function(sum) {
		console.log( sum );
	},
	// rejection handler
	function(err) {
		console.error( err ); // bummer!
	}
);
```

If something went wrong getting `X` or `Y`, or something somehow failed during the addition, the promise that `add(..)` returns is rejected, and the second callback error handler passed to `then(..)` will receive the rejection value from the promise.

Because Promises encapsulate the time-dependent state -- waiting on the fulfillment or rejection of the underlying value -- from the outside, the Promise itself is time-independent, and thus Promises can be composed (combined) in predictable ways regardless of the timing or outcome underneath.

Moreover, once a Promise is resolved, it stays that way forever -- it becomes an *immutable value* at that point -- and can then be *observed* as many times as necessary.

Note: Because a Promise is externally immutable once resolved, it's now safe to pass that value around to any party and know that it cannot be modified accidentally or maliciously. This is especially true in relation to multiple parties observing the resolution of a Promise. It is not possible for one party to affect another party's ability to observe Promise resolution. Immutability may sound like an academic topic, but it's actually one of the most fundamental and important aspects of Promise design, and shouldn't be casually passed over.

That's one of the most powerful and important concepts to understand about Promises. With a fair amount of work, you could ad hoc create the same effects with nothing but ugly callback composition, but that's not really an effective strategy, especially because you have to do it over and over again.

Promises are an easily repeatable mechanism for encapsulating and composing *future values*.

Completion Event

As we just saw, an individual Promise behaves as a *future value*. But there's another way to think of the resolution of a Promise: as a flow-control mechanism -- a temporal this-then-that -- for two or more steps in an asynchronous task.

Let's imagine calling a function `foo(..)` to perform some task. We don't know about any of its details, nor do we care. It may complete the task right away, or it may take a while.

We just simply need to know when `foo(..)` finishes so that we can move on to our next task. In other words, we'd like a way to be notified of `foo(..)` 's completion so that we can *continue*.

In typical JavaScript fashion, if you need to listen for a notification, you'd likely think of that in terms of events. So we could reframe our need for notification as a need to listen for a *completion* (or *continuation*) event emitted by `foo(..)` .

Note: Whether you call it a "completion event" or a "continuation event" depends on your perspective. Is the focus more on what happens with `foo(..)` , or what happens *after* `foo(..)` finishes? Both perspectives are accurate and useful. The event notification tells us that `foo(..)` has *completed*, but also that it's OK to *continue* with the next step. Indeed, the callback you pass to be called for the event notification is itself what we've previously called a *continuation*. Because *completion event* is a bit more focused on the `foo(..)` , which more has our attention at present, we slightly favor *completion event* for the rest of this text.

With callbacks, the "notification" would be our callback invoked by the task (`foo(..)`). But with Promises, we turn the relationship around, and expect that we can listen for an event from `foo(..)` , and when notified, proceed accordingly.

First, consider some pseudocode:

```
foo(x) {
	// start doing something that could take a while
}

foo( 42 )

on (foo "completion") {
	// now we can do the next step!
}

on (foo "error") {
	// oops, something went wrong in `foo(..)`
}
```

We call `foo(..)` and then we set up two event listeners, one for `"completion"` and one for `"error"` -- the two possible *final* outcomes of the `foo(..)` call. In essence, `foo(..)` doesn't even appear to be aware that the calling code has subscribed to these events, which makes for a very nice *separation of concerns*.

Unfortunately, such code would require some "magic" of the JS environment that doesn't exist (and would likely be a bit impractical). Here's the more natural way we could express that in JS:

```
function foo(x) {
	// start doing something that could take a while

	// make a `listener` event notification
	// capability to return

	return listener;
}

var evt = foo( 42 );

evt.on( "completion", function(){
	// now we can do the next step!
} );

evt.on( "failure", function(err){
	// oops, something went wrong in `foo(..)`
} );
```

`foo(..)` expressly creates an event subscription capability to return back, and the calling code receives and registers the two event handlers against it.

The inversion from normal callback-oriented code should be obvious, and it's intentional. Instead of passing the callbacks to `foo(..)`, it returns an event capability we call `evt`, which receives the callbacks.

But if you recall from Chapter 2, callbacks themselves represent an *inversion of control*. So inverting the callback pattern is actually an *inversion of inversion*, or an *uninversion of control* -- restoring control back to the calling code where we wanted it to be in the first place.

One important benefit is that multiple separate parts of the code can be given the event listening capability, and they can all independently be notified of when `foo(..)` completes to perform subsequent steps after its completion:

```
var evt = foo( 42 );

// let `bar(..)` listen to `foo(..)`'s completion
bar( evt );

// also, let `baz(..)` listen to `foo(..)`'s completion
baz( evt );
```

Uninversion of control enables a nicer *separation of concerns*, where `bar(..)` and `baz(..)` don't need to be involved in how `foo(..)` is called. Similarly, `foo(..)` doesn't need to know or care that `bar(..)` and `baz(..)` exist or are waiting to be notified when `foo(..)` completes.

Essentially, this `evt` object is a neutral third-party negotiation between the separate concerns.

Promise "Events"

As you may have guessed by now, the `evt` event listening capability is an analogy for a Promise.

In a Promise-based approach, the previous snippet would have `foo(..)` creating and returning a `Promise` instance, and that promise would then be passed to `bar(..)` and `baz(..)`.

Note: The Promise resolution "events" we listen for aren't strictly events (though they certainly behave like events for these purposes), and they're not typically called `"completion"` or `"error"`. Instead, we use `then(..)` to register a `"then"` event. Or perhaps more precisely, `then(..)` registers `"fulfillment"` and/or `"rejection"` event(s), though we don't see those terms used explicitly in the code.

Consider:

```
function foo(x) {
	// start doing something that could take a while

	// construct and return a promise
	return new Promise( function(resolve,reject){
		// eventually, call `resolve(..)` or `reject(..)`,
		// which are the resolution callbacks for
		// the promise.
	} );
}

var p = foo( 42 );

bar( p );

baz( p );
```

Note: The pattern shown with `new Promise( function(..){ .. } )` is generally called the "revealing constructor". The function passed in is executed immediately (not async deferred, as callbacks to `then(..)` are), and it's provided two parameters, which in this case we've named `resolve` and `reject`. These are the resolution functions for the promise. `resolve(..)` generally signals fulfillment, and `reject(..)` signals rejection.

You can probably guess what the internals of `bar(..)` and `baz(..)` might look like:

```
function bar(fooPromise) {
	// listen for `foo(..)` to complete
	fooPromise.then(
		function(){
			// `foo(..)` has now finished, so
			// do `bar(..)`'s task
		},
		function(){
			// oops, something went wrong in `foo(..)`
		}
	);
}

// ditto for `baz(..)`
```

Promise resolution doesn't necessarily need to involve sending along a message, as it did when we were examining Promises as *future values*. It can just be a flow-control signal, as used in the previous snippet.

Another way to approach this is:

```
function bar() {
	// `foo(..)` has definitely finished, so
	// do `bar(..)`'s task
}

function oopsBar() {
	// oops, something went wrong in `foo(..)`,
	// so `bar(..)` didn't run
}

// ditto for `baz()` and `oopsBaz()`

var p = foo( 42 );

p.then( bar, oopsBar );

p.then( baz, oopsBaz );
```

Note: If you've seen Promise-based coding before, you might be tempted to believe that the last two lines of that code could be written as `p.then( .. ).then( .. )`, using chaining, rather than `p.then(..); p.then(..)`. That would have an entirely different behavior, so be careful! The difference might not be clear right now, but it's actually a different async pattern than we've seen thus far: splitting/forking. Don't worry! We'll come back to this point later in this chapter.

Instead of passing the `p` promise to `bar(..)` and `baz(..)`, we use the promise to control when `bar(..)` and `baz(..)` will get executed, if ever. The primary difference is in the error handling.

In the first snippet's approach, `bar(..)` is called regardless of whether `foo(..)` succeeds or fails, and it handles its own fallback logic if it's notified that `foo(..)` failed. The same is true for `baz(..)`, obviously.

In the second snippet, `bar(..)` only gets called if `foo(..)` succeeds, and otherwise `oopsBar(..)` gets called. Ditto for `baz(..)`.

Neither approach is *correct* per se. There will be cases where one is preferred over the other.

In either case, the promise `p` that comes back from `foo(..)` is used to control what happens next.

Moreover, the fact that both snippets end up calling `then(..)` twice against the same promise `p` illustrates the point made earlier, which is that Promises (once resolved) retain their same resolution (fulfillment or rejection) forever, and can subsequently be observed as many times as necessary.

Whenever `p` is resolved, the next step will always be the same, both *now* and *later*.

Thenable Duck Typing

In Promises-land, an important detail is how to know for sure if some value is a genuine Promise or not. Or more directly, is it a value that will behave like a Promise?

Given that Promises are constructed by the `new Promise(..)` syntax, you might think that `p instanceof Promise` would be an acceptable check. But unfortunately, there are a number of reasons that's not totally sufficient.

Mainly, you can receive a Promise value from another browser window (iframe, etc.), which would have its own Promise different from the one in the current window/frame, and that check would fail to identify the Promise instance.

Moreover, a library or framework may choose to vend its own Promises and not use the native ES6 `Promise` implementation to do so. In fact, you may very well be using Promises with libraries in older browsers that have no Promise at all.

When we discuss Promise resolution processes later in this chapter, it will become more obvious why a non-genuine-but-Promise-like value would still be very important to be able to recognize and assimilate. But for now, just take my word for it that it's a critical piece of the puzzle.

As such, it was decided that the way to recognize a Promise (or something that behaves like a Promise) would be to define something called a "thenable" as any object or function which has a `then(..)` method on it. It is assumed that any such value is a Promise-conforming thenable.

The general term for "type checks" that make assumptions about a value's "type" based on its shape (what properties are present) is called "duck typing" -- "If it looks like a duck, and quacks like a duck, it must be a duck" (see the *Types & Grammar* title of this book series). So the duck typing check for a thenable would roughly be:

```
if (
	p !== null &&
	(
		typeof p === "object" ||
		typeof p === "function"
	) &&
	typeof p.then === "function"
) {
	// assume it's a thenable!
}
else {
	// not a thenable
}
```

Yuck! Setting aside the fact that this logic is a bit ugly to implement in various places, there's something deeper and more troubling going on.

If you try to fulfill a Promise with any object/function value that happens to have a `then(..)` function on it, but you weren't intending it to be treated as a Promise/thenable, you're out of luck, because it will automatically be recognized as thenable and treated with special rules (see later in the chapter).

This is even true if you didn't realize the value has a `then(..)` on it. For example:

```
var o = { then: function(){} };

// make `v` be `[[Prototype]]`-linked to `o`
var v = Object.create( o );

v.someStuff = "cool";
v.otherStuff = "not so cool";

v.hasOwnProperty( "then" );		// false
```

`v` doesn't look like a Promise or thenable at all. It's just a plain object with some properties on it. You're probably just intending to send that value around like any other object.

But unknown to you, `v` is also `[[Prototype]]`-linked (see the *this & Object Prototypes* title of this book series) to another object `o`, which happens to have a `then(..)` on it. So the thenable duck typing checks will think and assume `v` is a thenable. Uh oh.

It doesn't even need to be something as directly intentional as that:

```
Object.prototype.then = function(){};
Array.prototype.then = function(){};

var v1 = { hello: "world" };
var v2 = [ "Hello", "World" ];
```

Both `v1` and `v2` will be assumed to be thenables. You can't control or predict if any other code accidentally or maliciously adds `then(..)` to `Object.prototype`, `Array.prototype`, or any of the other native prototypes. And if what's specified is a function that doesn't call either of its parameters as callbacks, then any Promise resolved with such a value will just silently hang forever! Crazy.

Sound implausible or unlikely? Perhaps.

But keep in mind that there were several well-known non-Promise libraries preexisting in the community prior to ES6 that happened to already have a method on them called `then(..)`. Some of those libraries chose to rename their own methods to avoid collision (that sucks!). Others have simply been relegated to the unfortunate status of "incompatible with Promise-based coding" in reward for their inability to change to get out of the way.

The standards decision to hijack the previously nonreserved -- and completely general-purpose sounding -- `then` property name means that no value (or any of its delegates), either past, present, or future, can have a `then(..)` function present, either on purpose or by accident, or that value will be confused for a thenable in Promises systems, which will probably create bugs that are really hard to track down.

Warning: I do not like how we ended up with duck typing of thenables for Promise recognition. There were other options, such as "branding" or even "anti-branding"; what we got seems like a worst-case compromise. But it's not all doom and gloom. Thenable duck typing can be helpful, as we'll see later. Just beware that thenable duck typing can be hazardous if it incorrectly identifies something as a Promise that isn't.

Promise Trust

We've now seen two strong analogies that explain different aspects of what Promises can do for our async code. But if we stop there, we've missed perhaps the single most important characteristic that the Promise pattern establishes: trust.

Whereas the *future values* and *completion events* analogies play out explicitly in the code patterns we've explored, it won't be entirely obvious why or how Promises are designed to solve all of the *inversion of control* trust issues we laid out in the "Trust Issues" section of Chapter 2. But with a little digging, we can uncover some important guarantees that restore the confidence in async coding that Chapter 2 tore down!

Let's start by reviewing the trust issues with callbacks-only coding. When you pass a callback to a utility `foo(..)`, it might:

- Call the callback too early
- Call the callback too late (or never)
- Call the callback too few or too many times
- Fail to pass along any necessary environment/parameters
- Swallow any errors/exceptions that may happen

The characteristics of Promises are intentionally designed to provide useful, repeatable answers to all these concerns.

Calling Too Early

Primarily, this is a concern of whether code can introduce Zalgo-like effects (see Chapter 2), where sometimes a task finishes synchronously and sometimes asynchronously, which can lead to race conditions.

Promises by definition cannot be susceptible to this concern, because even an immediately fulfilled Promise (like `new Promise(function(resolve){ resolve(42); })`) cannot be *observed* synchronously.

That is, when you call `then(..)` on a Promise, even if that Promise was already resolved, the callback you provide to `then(..)` will **always** be called asynchronously (for more on this, refer back to "Jobs" in Chapter 1).

No more need to insert your own `setTimeout(..,0)` hacks. Promises prevent Zalgo automatically.

Calling Too Late

Similar to the previous point, a Promise's `then(..)` registered observation callbacks are automatically scheduled when either `resolve(..)` or `reject(..)` are called by the Promise creation capability. Those scheduled callbacks will predictably be fired at the next asynchronous moment (see "Jobs" in Chapter 1).

It's not possible for synchronous observation, so it's not possible for a synchronous chain of tasks to run in such a way to in effect "delay" another callback from happening as expected. That is, when a Promise is resolved, all `then(..)` registered callbacks on it will be called, in order, immediately at the next asynchronous opportunity (again, see "Jobs" in Chapter 1), and nothing that happens inside of one of those callbacks can affect/delay the calling of the other callbacks.

For example:

```
p.then( function(){
	p.then( function(){
		console.log( "C" );
	} );
	console.log( "A" );
} );
p.then( function(){
	console.log( "B" );
} );
// A B C
```

Here, `"C"` cannot interrupt and precede `"B"`, by virtue of how Promises are defined to operate.

Promise Scheduling Quirks

It's important to note, though, that there are lots of nuances of scheduling where the relative ordering between callbacks chained off two separate Promises is not reliably predictable.

If two promises `p1` and `p2` are both already resolved, it should be true that `p1.then(..); p2.then(..)` would end up calling the callback(s) for `p1` before the ones for `p2`. But there are subtle cases where that might not be true, such as the following:

```
var p3 = new Promise( function(resolve,reject){
	resolve( "B" );
} );

var p1 = new Promise( function(resolve,reject){
	resolve( p3 );
} );

var p2 = new Promise( function(resolve,reject){
	resolve( "A" );
} );

p1.then( function(v){
	console.log( v );
} );

p2.then( function(v){
	console.log( v );
} );

// A B  <-- not  B A  as you might expect
```

We'll cover this more later, but as you can see, `p1` is resolved not with an immediate value, but with another promise `p3` which is itself resolved with the value `"B"`. The specified behavior is to *unwrap* `p3` into `p1`, but asynchronously, so `p1`'s callback(s) are *behind* `p2` 's callback(s) in the asynchronous Job queue (see Chapter 1).

To avoid such nuanced nightmares, you should never rely on anything about the ordering/scheduling of callbacks across Promises. In fact, a good practice is not to code in such a way where the ordering of multiple callbacks matters at all. Avoid that if you can.

Never Calling the Callback

This is a very common concern. It's addressable in several ways with Promises.

First, nothing (not even a JS error) can prevent a Promise from notifying you of its resolution (if it's resolved). If you register both fulfillment and rejection callbacks for a Promise, and the Promise gets resolved, one of the two callbacks will always be called.

Of course, if your callbacks themselves have JS errors, you may not see the outcome you expect, but the callback will in fact have been called. We'll cover later how to be notified of an error in your callback, because even those don't get swallowed.

But what if the Promise itself never gets resolved either way? Even that is a condition that Promises provide an answer for, using a higher level abstraction called a "race":

```
// a utility for timing out a Promise
function timeoutPromise(delay) {
	return new Promise( function(resolve,reject){
		setTimeout( function(){
			reject( "Timeout!" );
		}, delay );
	} );
}

// setup a timeout for `foo()`
Promise.race( [
	foo(),					// attempt `foo()`
	timeoutPromise( 3000 )	// give it 3 seconds
] )
.then(
	function(){
		// `foo(..)` fulfilled in time!
	},
	function(err){
		// either `foo()` rejected, or it just
		// didn't finish in time, so inspect
		// `err` to know which
	}
);
```

There are more details to consider with this Promise timeout pattern, but we'll come back to it later.

Importantly, we can ensure a signal as to the outcome of `foo()` , to prevent it from hanging our program indefinitely.

Calling Too Few or Too Many Times

By definition, *one* is the appropriate number of times for the callback to be called. The "too few" case would be zero calls, which is the same as the "never" case we just examined.

The "too many" case is easy to explain. Promises are defined so that they can only be resolved once. If for some reason the Promise creation code tries to call `resolve(..)` or `reject(..)` multiple times, or tries to call both, the Promise will accept only the first resolution, and will silently ignore any subsequent attempts.

Because a Promise can only be resolved once, any `then(..)` registered callbacks will only ever be called once (each).

Of course, if you register the same callback more than once, (e.g., `p.then(f); p.then(f);`), it'll be called as many times as it was registered. The guarantee that a response function is called only once does not prevent you from shooting yourself in the foot.

Failing to Pass Along Any Parameters/Environment

Promises can have, at most, one resolution value (fulfillment or rejection).

If you don't explicitly resolve with a value either way, the value is `undefined`, as is typical in JS. But whatever the value, it will always be passed to all registered (and appropriate: fulfillment or rejection) callbacks, either *now* or in the future.

Something to be aware of: If you call `resolve(..)` or `reject(..)` with multiple parameters, all subsequent parameters beyond the first will be silently ignored. Although that might seem a violation of the guarantee we just described, it's not exactly, because it constitutes an invalid usage of the Promise mechanism. Other invalid usages of the API (such as calling `resolve(..)` multiple times) are similarly *protected*, so the Promise behavior here is consistent (if not a tiny bit frustrating).

If you want to pass along multiple values, you must wrap them in another single value that you pass, such as an `array` or an `object`.

As for environment, functions in JS always retain their closure of the scope in which they're defined (see the *Scope & Closures* title of this series), so they of course would continue to have access to whatever surrounding state you provide. Of course, the same is true of callbacks-only design, so this isn't a specific augmentation of benefit from Promises -- but it's a guarantee we can rely on nonetheless.

Swallowing Any Errors/Exceptions

In the base sense, this is a restatement of the previous point. If you reject a Promise with a *reason* (aka error message), that value is passed to the rejection callback(s).

But there's something much bigger at play here. If at any point in the creation of a Promise, or in the observation of its resolution, a JS exception error occurs, such as a `TypeError` or `ReferenceError`, that exception will be caught, and it will force the Promise in question to become rejected.

For example:

```
var p = new Promise( function(resolve,reject){
	foo.bar();	// `foo` is not defined, so error!
	resolve( 42 );	// never gets here :(
} );

p.then(
	function fulfilled(){
		// never gets here :(
	},
	function rejected(err){
		// `err` will be a `TypeError` exception object
		// from the `foo.bar()` line.
	}
);
```

The JS exception that occurs from `foo.bar()` becomes a Promise rejection that you can catch and respond to.

This is an important detail, because it effectively solves another potential Zalgo moment, which is that errors could create a synchronous reaction whereas nonerrors would be asynchronous. Promises turn even JS exceptions into asynchronous behavior, thereby reducing the race condition chances greatly.

But what happens if a Promise is fulfilled, but there's a JS exception error during the observation (in a `then(..)` registered callback)? Even those aren't lost, but you may find how they're handled a bit surprising, until you dig in a little deeper:

```
var p = new Promise( function(resolve,reject){
	resolve( 42 );
} );

p.then(
	function fulfilled(msg){
		foo.bar();
		console.log( msg );	// never gets here :(
	},
	function rejected(err){
		// never gets here either :(
	}
);
```

Wait, that makes it seem like the exception from `foo.bar()` really did get swallowed. Never fear, it didn't. But something deeper is wrong, which is that we've failed to listen for it. The `p.then(..)` call itself returns another promise, and it's *that* promise that will be rejected with the `TypeError` exception.

Why couldn't it just call the error handler we have defined there? Seems like a logical behavior on the surface. But it would violate the fundamental principle that Promises are **immutable** once resolved. `p` was already fulfilled to the value `42`, so it can't later be changed to a rejection just because there's an error in observing `p`'s resolution.

Besides the principle violation, such behavior could wreak havoc, if say there were multiple `then(..)` registered callbacks on the promise `p`, because some would get called and others wouldn't, and it would be very opaque as to why.

Trustable Promise?

There's one last detail to examine to establish trust based on the Promise pattern.

You've no doubt noticed that Promises don't get rid of callbacks at all. They just change where the callback is passed to. Instead of passing a callback to `foo(..)`, we get *something* (ostensibly a genuine Promise) back from `foo(..)`, and we pass the callback to that *something* instead.

But why would this be any more trustable than just callbacks alone? How can we be sure the *something* we get back is in fact a trustable Promise? Isn't it basically all just a house of cards where we can trust only because we already trusted?

One of the most important, but often overlooked, details of Promises is that they have a solution to this issue as well. Included with the native ES6 `Promise` implementation is `Promise.resolve(..)`.

If you pass an immediate, non-Promise, non-thenable value to `Promise.resolve(..)`, you get a promise that's fulfilled with that value. In other words, these two promises `p1` and `p2` will behave basically identically:

```
var p1 = new Promise( function(resolve,reject){
	resolve( 42 );
} );

var p2 = Promise.resolve( 42 );
```

But if you pass a genuine Promise to `Promise.resolve(..)`, you just get the same promise back:

```
var p1 = Promise.resolve( 42 );

var p2 = Promise.resolve( p1 );

p1 === p2; // true
```

Even more importantly, if you pass a non-Promise thenable value to `Promise.resolve(..)`, it will attempt to unwrap that value, and the unwrapping will keep going until a concrete final non-Promise-like value is extracted.

Recall our previous discussion of thenables?

Consider:

```
var p = {
	then: function(cb) {
		cb( 42 );
	}
};
```

```
// this works OK, but only by good fortune
p
.then(
	function fulfilled(val){
		console.log( val ); // 42
	},
	function rejected(err){
		// never gets here
	}
);
```

This `p` is a thenable, but it's not a genuine Promise. Luckily, it's reasonable, as most will be. But what if you got back instead something that looked like:

```
var p = {
	then: function(cb,errcb) {
		cb( 42 );
		errcb( "evil laugh" );
	}
};

p
.then(
	function fulfilled(val){
		console.log( val ); // 42
	},
	function rejected(err){
		// oops, shouldn't have run
		console.log( err ); // evil laugh
	}
);
```

This `p` is a thenable but it's not so well behaved of a promise. Is it malicious? Or is it just ignorant of how Promises should work? It doesn't really matter, to be honest. In either case, it's not trustable as is.

Nonetheless, we can pass either of these versions of `p` to `Promise.resolve(..)`, and we'll get the normalized, safe result we'd expect:

```
Promise.resolve( p )
.then(
	function fulfilled(val){
		console.log( val ); // 42
	},
	function rejected(err){
		// never gets here
	}
);
```

`Promise.resolve(..)` will accept any thenable, and will unwrap it to its non-thenable value. But you get back from `Promise.resolve(..)` a real, genuine Promise in its place, **one that you can trust**. If what you passed in is already a genuine Promise, you just get it right back, so there's no downside at all to filtering through `Promise.resolve(..)` to gain trust.

So let's say we're calling a `foo(..)` utility and we're not sure we can trust its return value to be a well-behaving Promise, but we know it's at least a thenable. `Promise.resolve(..)` will give us a trustable Promise wrapper to chain off of:

```
// don't just do this:
foo( 42 )
.then( function(v){
	console.log( v );
} );

// instead, do this:
Promise.resolve( foo( 42 ) )
.then( function(v){
        console.log( v );
} );
```

Note: Another beneficial side effect of wrapping `Promise.resolve(..)` around any function's return value (thenable or not) is that it's an easy way to normalize that function call into a well-behaving async task. If `foo(42)` returns an immediate value sometimes, or a Promise other times, `Promise.resolve( foo(42) )` makes sure it's always a Promise result. And avoiding Zalgo makes for much better code.

Trust Built

Hopefully the previous discussion now fully "resolves" (pun intended) in your mind why the Promise is trustable, and more importantly, why that trust is so critical in building robust, maintainable software.

Can you write async code in JS without trust? Of course you can. We JS developers have been coding async with nothing but callbacks for nearly two decades.

But once you start questioning just how much you can trust the mechanisms you build upon to actually be predictable and reliable, you start to realize callbacks have a pretty shaky trust foundation.

Promises are a pattern that augments callbacks with trustable semantics, so that the behavior is more reason-able and more reliable. By uninverting the *inversion of control* of callbacks, we place the control with a trustable system (Promises) that was designed specifically to bring sanity to our async.

Chain Flow

We've hinted at this a couple of times already, but Promises are not just a mechanism for a single-step *this-then-that* sort of operation. That's the building block, of course, but it turns out we can string multiple Promises together to represent a sequence of async steps.

The key to making this work is built on two behaviors intrinsic to Promises:

- Every time you call `then(..)` on a Promise, it creates and returns a new Promise, which we can *chain* with.
- Whatever value you return from the `then(..)` call's fulfillment callback (the first parameter) is automatically set as the fulfillment of the *chained* Promise (from the first point).

Let's first illustrate what that means, and *then* we'll derive how that helps us create async sequences of flow control. Consider the following:

```
var p = Promise.resolve( 21 );

var p2 = p.then( function(v){
        console.log( v );       // 21

        // fulfill `p2` with value `42`
        return v * 2;
} );

// chain off `p2`
p2.then( function(v){
        console.log( v );       // 42
} );
```

By returning `v * 2` (i.e., `42`), we fulfill the `p2` promise that the first `then(..)` call created and returned. When `p2`'s `then(..)` call runs, it's receiving the fulfillment from the `return v * 2` statement. Of course, `p2.then(..)` creates yet another promise, which we could have stored in a `p3` variable.

But it's a little annoying to have to create an intermediate variable `p2` (or `p3`, etc.). Thankfully, we can easily just chain these together:

```
var p = Promise.resolve( 21 );

p
.then( function(v){
        console.log( v );       // 21

        // fulfill the chained promise with value `42`
        return v * 2;
} )
// here's the chained promise
.then( function(v){
        console.log( v );       // 42
} );
```

So now the first `then(..)` is the first step in an async sequence, and the second `then(..)` is the second step. This could keep going for as long as you needed it to extend. Just keep chaining off a previous `then(..)` with each automatically created Promise.

But there's something missing here. What if we want step 2 to wait for step 1 to do something asynchronous? We're using an immediate `return` statement, which immediately fulfills the chained promise.

The key to making a Promise sequence truly async capable at every step is to recall how `Promise.resolve(..)` operates when what you pass to it is a Promise or thenable instead of a final value. `Promise.resolve(..)` directly returns a received genuine Promise, or it unwraps the value of a received thenable -- and keeps going recursively while it keeps unwrapping thenables.

The same sort of unwrapping happens if you `return` a thenable or Promise from the fulfillment (or rejection) handler. Consider:

```
var p = Promise.resolve( 21 );

p.then( function(v){
        console.log( v );       // 21

        // create a promise and return it
        return new Promise( function(resolve,reject){
                // fulfill with value `42`
                resolve( v * 2 );
        } );
} )
.then( function(v){
        console.log( v );       // 42
} );
```

Even though we wrapped `42` up in a promise that we returned, it still got unwrapped and ended up as the resolution of the chained promise, such that the second `then(..)` still received `42`. If we introduce asynchrony to that wrapping promise, everything still nicely works the same:

```
var p = Promise.resolve( 21 );

p.then( function(v){
        console.log( v );       // 21

        // create a promise to return
        return new Promise( function(resolve,reject){
                // introduce asynchrony!
                setTimeout( function(){
                        // fulfill with value `42`
                        resolve( v * 2 );
                }, 100 );
        } );
} )
.then( function(v){
        // runs after the 100ms delay in the previous step
```

```
		console.log( v );		// 42
	} );
```

That's incredibly powerful! Now we can construct a sequence of however many async steps we want, and each step can delay the next step (or not!), as necessary.

Of course, the value passing from step to step in these examples is optional. If you don't return an explicit value, an implicit `undefined` is assumed, and the promises still chain together the same way. Each Promise resolution is thus just a signal to proceed to the next step.

To further the chain illustration, let's generalize a delay-Promise creation (without resolution messages) into a utility we can reuse for multiple steps:

```
function delay(time) {
	return new Promise( function(resolve,reject){
		setTimeout( resolve, time );
	} );
}

delay( 100 ) // step 1
.then( function STEP2(){
	console.log( "step 2 (after 100ms)" );
	return delay( 200 );
} )
.then( function STEP3(){
	console.log( "step 3 (after another 200ms)" );
} )
.then( function STEP4(){
	console.log( "step 4 (next Job)" );
	return delay( 50 );
} )
.then( function STEP5(){
	console.log( "step 5 (after another 50ms)" );
} )
...
```

Calling `delay(200)` creates a promise that will fulfill in 200ms, and then we return that from the first `then(..)` fulfillment callback, which causes the second `then(..)` 's promise to wait on that 200ms promise.

Note: As described, technically there are two promises in that interchange: the 200ms-delay promise and the chained promise that the second `then(..)` chains from. But you may find it easier to mentally combine these two promises together, because the Promise mechanism automatically merges their states for you. In that respect, you could think of `return delay(200)` as creating a promise that replaces the earlier-returned chained promise.

To be honest, though, sequences of delays with no message passing isn't a terribly useful example of Promise flow control. Let's look at a scenario that's a little more practical.

Instead of timers, let's consider making Ajax requests:

```
// assume an `ajax( {url}, {callback} )` utility

// Promise-aware ajax
function request(url) {
	return new Promise( function(resolve,reject){
		// the `ajax(..)` callback should be our
		// promise's `resolve(..)` function
		ajax( url, resolve );
	} );
}
```

We first define a `request(..)` utility that constructs a promise to represent the completion of the `ajax(..)` call:

```
request( "http://some.url.1/" )
.then( function(response1){
        return request( "http://some.url.2/?v=" + response1 );
} )
.then( function(response2){
        console.log( response2 );
} );
```

Note: Developers commonly encounter situations in which they want to do Promise-aware async flow control with utilities that are not themselves Promise-enabled (like `ajax(..)` here, which expects a callback). Although the native ES6 `Promise` mechanism doesn't automatically solve this pattern for us, practically all Promise libraries *do*. They usually call this process "lifting" or "promisifying" or some variation thereof. We'll come back to this technique later.

Using the Promise-returning `request(..)`, we create the first step in our chain implicitly by calling it with the first URL, and chain off that returned promise with the first `then(..)`.

Once `response1` comes back, we use that value to construct a second URL, and make a second `request(..)` call. That second `request(..)` promise is `return` ed so that the third step in our async flow control waits for that Ajax call to complete. Finally, we print `response2` once it returns.

The Promise chain we construct is not only a flow control that expresses a multistep async sequence, but it also acts as a message channel to propagate messages from step to step.

What if something went wrong in one of the steps of the Promise chain? An error/exception is on a per-Promise basis, which means it's possible to catch such an error at any point in the chain, and that catching acts to sort of "reset" the chain back to normal operation at that point:

```
// step 1:
request( "http://some.url.1/" )

// step 2:
.then( function(response1){
        foo.bar(); // undefined, error!

        // never gets here
        return request( "http://some.url.2/?v=" + response1 );
} )

// step 3:
.then(
        function fulfilled(response2){
                // never gets here
        },
        // rejection handler to catch the error
        function rejected(err){
                console.log( err );     // `TypeError` from `foo.bar()` error
                return 42;
        }
)

// step 4:
.then( function(msg){
        console.log( msg );             // 42
} );
```

When the error occurs in step 2, the rejection handler in step 3 catches it. The return value (`42` in this snippet), if any, from that rejection handler fulfills the promise for the next step (4), such that the chain is now back in a fulfillment state.

Note: As we discussed earlier, when returning a promise from a fulfillment handler, it's unwrapped and can delay the next step. That's also true for returning promises from rejection handlers, such that if the `return 42` in step 3 instead returned a promise, that promise could delay step 4. A thrown exception inside either the fulfillment or rejection handler of a `then(..)` call causes the next (chained) promise to be immediately rejected with that exception.

If you call `then(..)` on a promise, and you only pass a fulfillment handler to it, an assumed rejection handler is substituted:

```
var p = new Promise( function(resolve,reject){
	reject( "Oops" );
} );

var p2 = p.then(
	function fulfilled(){
		// never gets here
	}
	// assumed rejection handler, if omitted or
	// any other non-function value passed
	// function(err) {
	//     throw err;
	// }
);
```

As you can see, the assumed rejection handler simply rethrows the error, which ends up forcing `p2` (the chained promise) to reject with the same error reason. In essence, this allows the error to continue propagating along a Promise chain until an explicitly defined rejection handler is encountered.

Note: We'll cover more details of error handling with Promises a little later, because there are other nuanced details to be concerned about.

If a proper valid function is not passed as the fulfillment handler parameter to `then(..)`, there's also a default handler substituted:

```
var p = Promise.resolve( 42 );

p.then(
	// assumed fulfillment handler, if omitted or
	// any other non-function value passed
	// function(v) {
	//     return v;
	// }
	null,
	function rejected(err){
		// never gets here
	}
);
```

As you can see, the default fulfillment handler simply passes whatever value it receives along to the next step (Promise).

Note: The `then(null,function(err){ .. })` pattern -- only handling rejections (if any) but letting fulfillments pass through -- has a shortcut in the API: `catch(function(err){ .. })`. We'll cover `catch(..)` more fully in the next section.

Let's review briefly the intrinsic behaviors of Promises that enable chaining flow control:

- A `then(..)` call against one Promise automatically produces a new Promise to return from the call.
- Inside the fulfillment/rejection handlers, if you return a value or an exception is thrown, the new returned (chainable) Promise is resolved accordingly.
- If the fulfillment or rejection handler returns a Promise, it is unwrapped, so that whatever its resolution is will become the resolution of the chained Promise returned from the current `then(..)`.

While chaining flow control is helpful, it's probably most accurate to think of it as a side benefit of how Promises compose (combine) together, rather than the main intent. As we've discussed in detail several times already, Promises normalize asynchrony and encapsulate time-dependent value state, and *that* is what lets us chain them together in this useful way.

Certainly, the sequential expressiveness of the chain (this-then-this-then-this...) is a big improvement over the tangled mess of callbacks as we identified in Chapter 2. But there's still a fair amount of boilerplate (`then(..)` and `function(){ .. }`) to wade through. In the next chapter, we'll see a significantly nicer pattern for sequential flow control expressivity, with generators.

Terminology: Resolve, Fulfill, and Reject

There's some slight confusion around the terms "resolve," "fulfill," and "reject" that we need to clear up, before you get too much deeper into learning about Promises. Let's first consider the `Promise(..)` constructor:

```
var p = new Promise( function(X,Y){
	// X() for fulfillment
	// Y() for rejection
} );
```

As you can see, two callbacks (here labeled `X` and `Y`) are provided. The first is *usually* used to mark the Promise as fulfilled, and the second *always* marks the Promise as rejected. But what's the "usually" about, and what does that imply about accurately naming those parameters?

Ultimately, it's just your user code and the identifier names aren't interpreted by the engine to mean anything, so it doesn't *technically* matter; `foo(..)` and `bar(..)` are equally functional. But the words you use can affect not only how you are thinking about the code, but how other developers on your team will think about it. Thinking wrongly about carefully orchestrated async code is almost surely going to be worse than the spaghetti-callback alternatives.

So it actually does kind of matter what you call them.

The second parameter is easy to decide. Almost all literature uses `reject(..)` as its name, and because that's exactly (and only!) what it does, that's a very good choice for the name. I'd strongly recommend you always use `reject(..)`.

But there's a little more ambiguity around the first parameter, which in Promise literature is often labeled `resolve(..)`. That word is obviously related to "resolution," which is what's used across the literature (including this book) to describe setting a final value/state to a Promise. We've already used "resolve the Promise" several times to mean either fulfilling or rejecting the Promise.

But if this parameter seems to be used to specifically fulfill the Promise, why shouldn't we call it `fulfill(..)` instead of `resolve(..)` to be more accurate? To answer that question, let's also take a look at two of the `Promise` API methods:

```
var fulfilledPr = Promise.resolve( 42 );

var rejectedPr = Promise.reject( "Oops" );
```

`Promise.resolve(..)` creates a Promise that's resolved to the value given to it. In this example, `42` is a normal, non-Promise, non-thenable value, so the fulfilled promise `fulfilledPr` is created for the value `42`. `Promise.reject("Oops")` creates the rejected promise `rejectedPr` for the reason `"Oops"`.

Let's now illustrate why the word "resolve" (such as in `Promise.resolve(..)`) is unambiguous and indeed more accurate, if used explicitly in a context that could result in either fulfillment or rejection:

```
var rejectedTh = {
	then: function(resolved,rejected) {
		rejected( "Oops" );
	}
};

var rejectedPr = Promise.resolve( rejectedTh );
```

As we discussed earlier in this chapter, `Promise.resolve(..)` will return a received genuine Promise directly, or unwrap a received thenable. If that thenable unwrapping reveals a rejected state, the Promise returned from `Promise.resolve(..)` is in fact in that same rejected state.

So `Promise.resolve(..)` is a good, accurate name for the API method, because it can actually result in either fulfillment or rejection.

The first callback parameter of the `Promise(..)` constructor will unwrap either a thenable (identically to `Promise.resolve(..)`) or a genuine Promise:

```
var rejectedPr = new Promise( function(resolve,reject){
        // resolve this promise with a rejected promise
        resolve( Promise.reject( "Oops" ) );
} );

rejectedPr.then(
        function fulfilled(){
                // never gets here
        },
        function rejected(err){
                console.log( err );     // "Oops"
        }
);
```

It should be clear now that `resolve(..)` is the appropriate name for the first callback parameter of the `Promise(..)` constructor.

Warning: The previously mentioned `reject(..)` does **not** do the unwrapping that `resolve(..)` does. If you pass a Promise/thenable value to `reject(..)` , that untouched value will be set as the rejection reason. A subsequent rejection handler would receive the actual Promise/thenable you passed to `reject(..)` , not its underlying immediate value.

But now let's turn our attention to the callbacks provided to `then(..)` . What should they be called (both in literature and in code)? I would suggest `fulfilled(..)` and `rejected(..)` :

```
function fulfilled(msg) {
        console.log( msg );
}

function rejected(err) {
        console.error( err );
}

p.then(
        fulfilled,
        rejected
);
```

In the case of the first parameter to `then(..)` , it's unambiguously always the fulfillment case, so there's no need for the duality of "resolve" terminology. As a side note, the ES6 specification uses `onFulfilled(..)` and `onRejected(..)` to label these two callbacks, so they are accurate terms.

Error Handling

We've already seen several examples of how Promise rejection -- either intentional through calling `reject(..)` or accidental through JS exceptions -- allows saner error handling in asynchronous programming. Let's circle back though and be explicit about some of the details that we glossed over.

The most natural form of error handling for most developers is the synchronous `try..catch` construct. Unfortunately, it's synchronous-only, so it fails to help in async code patterns:

```
function foo() {
        setTimeout( function(){
                baz.bar();
        }, 100 );
}

try {
        foo();
        // later throws global error from `baz.bar()`
}
catch (err) {
        // never gets here
}
```

`try..catch` would certainly be nice to have, but it doesn't work across async operations. That is, unless there's some additional environmental support, which we'll come back to with generators in Chapter 4.

In callbacks, some standards have emerged for patterned error handling, most notably the "error-first callback" style:

```
function foo(cb) {
        setTimeout( function(){
                try {
                        var x = baz.bar();
                        cb( null, x ); // success!
                }
                catch (err) {
                        cb( err );
                }
        }, 100 );
}

foo( function(err,val){
        if (err) {
                console.error( err ); // bummer :(
        }
        else {
                console.log( val );
        }
} );
```

Note: The `try..catch` here works only from the perspective that the `baz.bar()` call will either succeed or fail immediately, synchronously. If `baz.bar()` was itself its own async completing function, any async errors inside it would not be catchable.

The callback we pass to `foo(..)` expects to receive a signal of an error by the reserved first parameter `err`. If present, error is assumed. If not, success is assumed.

This sort of error handling is technically *async capable*, but it doesn't compose well at all. Multiple levels of error-first callbacks woven together with these ubiquitous `if` statement checks inevitably will lead you to the perils of callback hell (see Chapter 2).

So we come back to error handling in Promises, with the rejection handler passed to `then(..)`. Promises don't use the popular "error-first callback" design style, but instead use "split callbacks" style; there's one callback for fulfillment and one for rejection:

```
var p = Promise.reject( "Oops" );

p.then(
        function fulfilled(){
                // never gets here
        },
        function rejected(err){
                console.log( err ); // "Oops"
        }
);
```

While this pattern of error handling makes fine sense on the surface, the nuances of Promise error handling are often a fair bit more difficult to fully grasp.

Consider:

```
var p = Promise.resolve( 42 );

p.then(
        function fulfilled(msg){
                // numbers don't have string functions,
                // so will throw an error
                console.log( msg.toLowerCase() );
        },
        function rejected(err){
                // never gets here
        }
);
```

If the `msg.toLowerCase()` legitimately throws an error (it does!), why doesn't our error handler get notified? As we explained earlier, it's because *that* error handler is for the `p` promise, which has already been fulfilled with value `42`. The `p` promise is immutable, so the only promise that can be notified of the error is the one returned from `p.then(..)`, which in this case we don't capture.

That should paint a clear picture of why error handling with Promises is error-prone (pun intended). It's far too easy to have errors swallowed, as this is very rarely what you'd intend.

Warning: If you use the Promise API in an invalid way and an error occurs that prevents proper Promise construction, the result will be an immediately thrown exception, **not a rejected Promise**. Some examples of incorrect usage that fail Promise construction: `new Promise(null)`, `Promise.all()`, `Promise.race(42)`, and so on. You can't get a rejected Promise if you don't use the Promise API validly enough to actually construct a Promise in the first place!

Pit of Despair

Jeff Atwood noted years ago: programming languages are often set up in such a way that by default, developers fall into the "pit of despair" (http://blog.codinghorror.com/falling-into-the-pit-of-success/) -- where accidents are punished -- and that you have to try harder to get it right. He implored us to instead create a "pit of success," where by default you fall into expected (successful) action, and thus would have to try hard to fail.

Promise error handling is unquestionably "pit of despair" design. By default, it assumes that you want any error to be swallowed by the Promise state, and if you forget to observe that state, the error silently languishes/dies in obscurity -- usually despair.

To avoid losing an error to the silence of a forgotten/discarded Promise, some developers have claimed that a "best practice" for Promise chains is to always end your chain with a final `catch(..)`, like:

```
var p = Promise.resolve( 42 );

p.then(
        function fulfilled(msg){
                // numbers don't have string functions,
                // so will throw an error
                console.log( msg.toLowerCase() );
        }
)
.catch( handleErrors );
```

Because we didn't pass a rejection handler to the `then(..)`, the default handler was substituted, which simply propagates the error to the next promise in the chain. As such, both errors that come into `p`, and errors that come *after* `p` in its resolution (like the `msg.toLowerCase()` one) will filter down to the final `handleErrors(..)`.

Problem solved, right? Not so fast!

What happens if `handleErrors(..)` itself also has an error in it? Who catches that? There's still yet another unattended promise: the one `catch(..)` returns, which we don't capture and don't register a rejection handler for.

You can't just stick another `catch(..)` on the end of that chain, because it too could fail. The last step in any Promise chain, whatever it is, always has the possibility, even decreasingly so, of dangling with an uncaught error stuck inside an unobserved Promise.

Sound like an impossible conundrum yet?

Uncaught Handling

It's not exactly an easy problem to solve completely. There are other ways to approach it which many would say are *better*.

Some Promise libraries have added methods for registering something like a "global unhandled rejection" handler, which would be called instead of a globally thrown error. But their solution for how to identify an error as "uncaught" is to have an arbitrary-length timer, say 3 seconds, running from time of rejection. If a Promise is rejected but no error handler is registered before the timer fires, then it's assumed that you won't ever be registering a handler, so it's "uncaught."

In practice, this has worked well for many libraries, as most usage patterns don't typically call for significant delay between Promise rejection and observation of that rejection. But this pattern is troublesome because 3 seconds is so arbitrary (even if empirical), and also because there are indeed some cases where you want a Promise to hold on to its rejectedness for some indefinite period of time, and you don't really want to have your "uncaught" handler called for all those false positives (not-yet-handled "uncaught errors").

Another more common suggestion is that Promises should have a `done(..)` added to them, which essentially marks the Promise chain as "done." `done(..)` doesn't create and return a Promise, so the callbacks passed to `done(..)` are obviously not wired up to report problems to a chained Promise that doesn't exist.

So what happens instead? It's treated as you might usually expect in uncaught error conditions: any exception inside a `done(..)` rejection handler would be thrown as a global uncaught error (in the developer console, basically):

```
var p = Promise.resolve( 42 );

p.then(
	function fulfilled(msg){
		// numbers don't have string functions,
		// so will throw an error
		console.log( msg.toLowerCase() );
	}
)
.done( null, handleErrors );

// if `handleErrors(..)` caused its own exception, it would
// be thrown globally here
```

This might sound more attractive than the never-ending chain or the arbitrary timeouts. But the biggest problem is that it's not part of the ES6 standard, so no matter how good it sounds, at best it's a lot longer way off from being a reliable and ubiquitous solution.

Are we just stuck, then? Not entirely.

Browsers have a unique capability that our code does not have: they can track and know for sure when any object gets thrown away and garbage collected. So, browsers can track Promise objects, and whenever they get garbage collected, if they have a rejection in them, the browser knows for sure this was a legitimate "uncaught error," and can thus confidently know it should report it to the developer console.

Note: At the time of this writing, both Chrome and Firefox have early attempts at that sort of "uncaught rejection" capability, though support is incomplete at best.

However, if a Promise doesn't get garbage collected -- it's exceedingly easy for that to accidentally happen through lots of different coding patterns -- the browser's garbage collection sniffing won't help you know and diagnose that you have a silently rejected Promise laying around.

Is there any other alternative? Yes.

Pit of Success

The following is just theoretical, how Promises *could* be someday changed to behave. I believe it would be far superior to what we currently have. And I think this change would be possible even post-ES6 because I don't think it would break web compatibility with ES6 Promises. Moreover, it can be polyfilled/prollyfilled in, if you're careful. Let's take a look:

- Promises could default to reporting (to the developer console) any rejection, on the next Job or event loop tick, if at that exact moment no error handler has been registered for the Promise.
- For the cases where you want a rejected Promise to hold onto its rejected state for an indefinite amount of time before observing, you could call `defer()`, which suppresses automatic error reporting on that Promise.

If a Promise is rejected, it defaults to noisily reporting that fact to the developer console (instead of defaulting to silence). You can opt out of that reporting either implicitly (by registering an error handler before rejection), or explicitly (with `defer()`). In either case, *you* control the false positives.

Consider:

```
var p = Promise.reject( "Oops" ).defer();

// `foo(..)` is Promise-aware
foo( 42 )
.then(
	function fulfilled(){
		return p;
	},
	function rejected(err){
		// handle `foo(..)` error
	}
);
...
```

When we create `p`, we know we're going to wait a while to use/observe its rejection, so we call `defer()` -- thus no global reporting. `defer()` simply returns the same promise, for chaining purposes.

The promise returned from `foo(..)` gets an error handler attached *right away*, so it's implicitly opted out and no global reporting for it occurs either.

But the promise returned from the `then(..)` call has no `defer()` or error handler attached, so if it rejects (from inside either resolution handler), then *it* will be reported to the developer console as an uncaught error.

This design is a pit of success. By default, all errors are either handled or reported -- what almost all developers in almost all cases would expect. You either have to register a handler or you have to intentionally opt out, and indicate you intend to defer error handling until *later*; you're opting for the extra responsibility in just that specific case.

The only real danger in this approach is if you `defer()` a Promise but then fail to actually ever observe/handle its rejection.

But you had to intentionally call `defer()` to opt into that pit of despair -- the default was the pit of success -- so there's not much else we could do to save you from your own mistakes.

I think there's still hope for Promise error handling (post-ES6). I hope the powers that be will rethink the situation and consider this alternative. In the meantime, you can implement this yourself (a challenging exercise for the reader!), or use a *smarter* Promise library that does so for you!

Note: This exact model for error handling/reporting is implemented in my *asynquence* Promise abstraction library, which will be discussed in Appendix A of this book.

Promise Patterns

We've already implicitly seen the sequence pattern with Promise chains (this-then-this-then-that flow control) but there are lots of variations on asynchronous patterns that we can build as abstractions on top of Promises. These patterns serve to simplify the expression of async flow control -- which helps make our code more reason-able and more maintainable -- even in the most complex parts of our programs.

Two such patterns are codified directly into the native ES6 `Promise` implementation, so we get them for free, to use as building blocks for other patterns.

Promise.all([..])

In an async sequence (Promise chain), only one async task is being coordinated at any given moment -- step 2 strictly follows step 1, and step 3 strictly follows step 2. But what about doing two or more steps concurrently (aka "in parallel")?

In classic programming terminology, a "gate" is a mechanism that waits on two or more parallel/concurrent tasks to complete before continuing. It doesn't matter what order they finish in, just that all of them have to complete for the gate to open and let the flow control through.

In the Promise API, we call this pattern `all([ .. ])`.

Say you wanted to make two Ajax requests at the same time, and wait for both to finish, regardless of their order, before making a third Ajax request. Consider:

```
// `request(..)` is a Promise-aware Ajax utility,
// like we defined earlier in the chapter

var p1 = request( "http://some.url.1/" );
var p2 = request( "http://some.url.2/" );

Promise.all( [p1,p2] )
.then( function(msgs){
        // both `p1` and `p2` fulfill and pass in
        // their messages here
        return request(
                "http://some.url.3/?v=" + msgs.join(",")
        );
} )
.then( function(msg){
        console.log( msg );
} );
```

`Promise.all([ .. ])` expects a single argument, an `array`, consisting generally of Promise instances. The promise returned from the `Promise.all([ .. ])` call will receive a fulfillment message (`msgs` in this snippet) that is an `array` of all the fulfillment messages from the passed in promises, in the same order as specified (regardless of fulfillment order).

Note: Technically, the `array` of values passed into `Promise.all([ .. ])` can include Promises, thenables, or even immediate values. Each value in the list is essentially passed through `Promise.resolve(..)` to make sure it's a genuine Promise to be waited on, so an immediate value will just be normalized into a Promise for that value. If the `array` is empty, the main Promise is immediately fulfilled.

The main promise returned from `Promise.all([ .. ])` will only be fulfilled if and when all its constituent promises are fulfilled. If any one of those promises instead is rejected, the main `Promise.all([ .. ])` promise is immediately rejected, discarding all results from any other promises.

Remember to always attach a rejection/error handler to every promise, even and especially the one that comes back from `Promise.all([ .. ])`.

Promise.race([..])

While `Promise.all([ .. ])` coordinates multiple Promises concurrently and assumes all are needed for fulfillment, sometimes you only want to respond to the "first Promise to cross the finish line," letting the other Promises fall away.

This pattern is classically called a "latch," but in Promises it's called a "race."

Warning: While the metaphor of "only the first across the finish line wins" fits the behavior well, unfortunately "race" is kind of a loaded term, because "race conditions" are generally taken as bugs in programs (see Chapter 1). Don't confuse `Promise.race([ .. ])` with "race condition."

`Promise.race([ .. ])` also expects a single `array` argument, containing one or more Promises, thenables, or immediate values. It doesn't make much practical sense to have a race with immediate values, because the first one listed will obviously win -- like a foot race where one runner starts at the finish line!

Similar to `Promise.all([ .. ])` , `Promise.race([ .. ])` will fulfill if and when any Promise resolution is a fulfillment, and it will reject if and when any Promise resolution is a rejection.

Warning: A "race" requires at least one "runner," so if you pass an empty `array` , instead of immediately resolving, the main `race([..])` Promise will never resolve. This is a footgun! ES6 should have specified that it either fulfills, rejects, or just throws some sort of synchronous error. Unfortunately, because of precedence in Promise libraries predating ES6 `Promise` , they had to leave this gotcha in there, so be careful never to send in an empty `array` .

Let's revisit our previous concurrent Ajax example, but in the context of a race between `p1` and `p2` :

```
// `request(..)` is a Promise-aware Ajax utility,
// like we defined earlier in the chapter

var p1 = request( "http://some.url.1/" );
var p2 = request( "http://some.url.2/" );

Promise.race( [p1,p2] )
.then( function(msg){
	// either `p1` or `p2` will win the race
	return request(
		"http://some.url.3/?v=" + msg
	);
} )
.then( function(msg){
	console.log( msg );
} );
```

Because only one promise wins, the fulfillment value is a single message, not an `array` as it was for `Promise.all([ .. ])` .

Timeout Race

We saw this example earlier, illustrating how `Promise.race([ .. ])` can be used to express the "promise timeout" pattern:

```
// `foo()` is a Promise-aware function

// `timeoutPromise(..)`, defined ealier, returns
// a Promise that rejects after a specified delay

// setup a timeout for `foo()`
Promise.race( [
	foo(),					// attempt `foo()`
	timeoutPromise( 3000 )	// give it 3 seconds
] )
.then(
	function(){
		// `foo(..)` fulfilled in time!
	},
	function(err){
		// either `foo()` rejected, or it just
		// didn't finish in time, so inspect
		// `err` to know which
	}
);
```

This timeout pattern works well in most cases. But there are some nuances to consider, and frankly they apply to both `Promise.race([ .. ])` and `Promise.all([ .. ])` equally.

"Finally"

The key question to ask is, "What happens to the promises that get discarded/ignored?" We're not asking that question from the performance perspective -- they would typically end up garbage collection eligible -- but from the behavioral perspective (side effects, etc.). Promises cannot be canceled -- and shouldn't be as that would destroy the external immutability trust discussed in the "Promise Uncancelable" section later in this chapter -- so they can only be silently ignored.

But what if `foo()` in the previous example is reserving some sort of resource for usage, but the timeout fires first and causes that promise to be ignored? Is there anything in this pattern that proactively frees the reserved resource after the timeout, or otherwise cancels any side effects it may have had? What if all you wanted was to log the fact that `foo()` timed out?

Some developers have proposed that Promises need a `finally(..)` callback registration, which is always called when a Promise resolves, and allows you to specify any cleanup that may be necessary. This doesn't exist in the specification at the moment, but it may come in ES7+. We'll have to wait and see.

It might look like:

```
var p = Promise.resolve( 42 );

p.then( something )
.finally( cleanup )
.then( another )
.finally( cleanup );
```

Note: In various Promise libraries, `finally(..)` still creates and returns a new Promise (to keep the chain going). If the `cleanup(..)` function were to return a Promise, it would be linked into the chain, which means you could still have the unhandled rejection issues we discussed earlier.

In the meantime, we could make a static helper utility that lets us observe (without interfering) the resolution of a Promise:

```
// polyfill-safe guard check
if (!Promise.observe) {
	Promise.observe = function(pr,cb) {
		// side-observe `pr`'s resolution
		pr.then(
			function fulfilled(msg){
				// schedule callback async (as Job)
				Promise.resolve( msg ).then( cb );
			},
			function rejected(err){
				// schedule callback async (as Job)
				Promise.resolve( err ).then( cb );
			}
		);

		// return original promise
		return pr;
	};
}
```

Here's how we'd use it in the timeout example from before:

```
Promise.race( [
	Promise.observe(
		foo(),					// attempt `foo()`
		function cleanup(msg){
			// clean up after `foo()`, even if it
			// didn't finish before the timeout
		}
	),
	timeoutPromise( 3000 )	// give it 3 seconds
] )
```

This `Promise.observe(..)` helper is just an illustration of how you could observe the completions of Promises without interfering with them. Other Promise libraries have their own solutions. Regardless of how you do it, you'll likely have places where you want to make sure your Promises aren't *just* silently ignored by accident.

Variations on all([..]) and race([..])

While native ES6 Promises come with built-in `Promise.all([ .. ])` and `Promise.race([ .. ])`, there are several other commonly used patterns with variations on those semantics:

- `none([ .. ])` is like `all([ .. ])`, but fulfillments and rejections are transposed. All Promises need to be rejected -- rejections become the fulfillment values and vice versa.
- `any([ .. ])` is like `all([ .. ])`, but it ignores any rejections, so only one needs to fulfill instead of *all* of them.
- `first([ .. ])` is a like a race with `any([ .. ])`, which is that it ignores any rejections and fulfills as soon as the first Promise fulfills.
- `last([ .. ])` is like `first([ .. ])`, but only the latest fulfillment wins.

Some Promise abstraction libraries provide these, but you could also define them yourself using the mechanics of Promises, `race([ .. ])` and `all([ .. ])`.

For example, here's how we could define `first([ .. ])`:

```
// polyfill-safe guard check
if (!Promise.first) {
	Promise.first = function(prs) {
		return new Promise( function(resolve,reject){
			// loop through all promises
			prs.forEach( function(pr){
				// normalize the value
				Promise.resolve( pr )
				// whichever one fulfills first wins, and
				// gets to resolve the main promise
				.then( resolve );
			} );
		} );
	};
}
```

Note: This implementation of `first(..)` does not reject if all its promises reject; it simply hangs, much like a `Promise.race([])` does. If desired, you could add additional logic to track each promise rejection and if all reject, call `reject()` on the main promise. We'll leave that as an exercise for the reader.

Concurrent Iterations

Sometimes you want to iterate over a list of Promises and perform some task against all of them, much like you can do with synchronous `array`s (e.g., `forEach(..)`, `map(..)`, `some(..)`, and `every(..)`). If the task to perform against each Promise is fundamentally synchronous, these work fine, just as we used `forEach(..)` in the previous snippet.

But if the tasks are fundamentally asynchronous, or can/should otherwise be performed concurrently, you can use async versions of these utilities as provided by many libraries.

For example, let's consider an asynchronous `map(..)` utility that takes an `array` of values (could be Promises or anything else), plus a function (task) to perform against each. `map(..)` itself returns a promise whose fulfillment value is an `array` that holds (in the same mapping order) the async fulfillment value from each task:

```
if (!Promise.map) {
	Promise.map = function(vals,cb) {
		// new promise that waits for all mapped promises
		return Promise.all(
			// note: regular array `map(..)`, turns
			// the array of values into an array of
			// promises
			vals.map( function(val){
				// replace `val` with a new promise that
				// resolves after `val` is async mapped
				return new Promise( function(resolve){
					cb( val, resolve );
				} );
			} )
```

```
                );
        };
}
```

Note: In this implementation of `map(..)`, you can't signal async rejection, but if a synchronous exception/error occurs inside of the mapping callback (`cb(..)`), the main `Promise.map(..)` returned promise would reject.

Let's illustrate using `map(..)` with a list of Promises (instead of simple values):

```
var p1 = Promise.resolve( 21 );
var p2 = Promise.resolve( 42 );
var p3 = Promise.reject( "Oops" );

// double values in list even if they're in
// Promises
Promise.map( [p1,p2,p3], function(pr,done){
        // make sure the item itself is a Promise
        Promise.resolve( pr )
        .then(
                // extract value as `v`
                function(v){
                        // map fulfillment `v` to new value
                        done( v * 2 );
                },
                // or, map to promise rejection message
                done
        );
} )
.then( function(vals){
        console.log( vals );    // [42,84,"Oops"]
} );
```

Promise API Recap

Let's review the ES6 `Promise` API that we've already seen unfold in bits and pieces throughout this chapter.

Note: The following API is native only as of ES6, but there are specification-compliant polyfills (not just extended Promise libraries) which can define `Promise` and all its associated behavior so that you can use native Promises even in pre-ES6 browsers. One such polyfill is "Native Promise Only" (http://github.com/getify/native-promise-only), which I wrote!

new Promise(..) Constructor

The *revealing constructor* `Promise(..)` must be used with `new`, and must be provided a function callback that is synchronously/immediately called. This function is passed two function callbacks that act as resolution capabilities for the promise. We commonly label these `resolve(..)` and `reject(..)`:

```
var p = new Promise( function(resolve,reject){
        // `resolve(..)` to resolve/fulfill the promise
        // `reject(..)` to reject the promise
} );
```

`reject(..)` simply rejects the promise, but `resolve(..)` can either fulfill the promise or reject it, depending on what it's passed. If `resolve(..)` is passed an immediate, non-Promise, non-thenable value, then the promise is fulfilled with that value.

But if `resolve(..)` is passed a genuine Promise or thenable value, that value is unwrapped recursively, and whatever its final resolution/state is will be adopted by the promise.

Promise.resolve(..) and Promise.reject(..)

A shortcut for creating an already-rejected Promise is `Promise.reject(..)`, so these two promises are equivalent:

```
var p1 = new Promise( function(resolve,reject){
        reject( "Oops" );
} );

var p2 = Promise.reject( "Oops" );
```

`Promise.resolve(..)` is usually used to create an already-fulfilled Promise in a similar way to `Promise.reject(..)`. However, `Promise.resolve(..)` also unwraps thenable values (as discussed several times already). In that case, the Promise returned adopts the final resolution of the thenable you passed in, which could either be fulfillment or rejection:

```
var fulfilledTh = {
        then: function(cb) { cb( 42 ); }
};
var rejectedTh = {
        then: function(cb,errCb) {
                errCb( "Oops" );
        }
};

var p1 = Promise.resolve( fulfilledTh );
var p2 = Promise.resolve( rejectedTh );

// `p1` will be a fulfilled promise
// `p2` will be a rejected promise
```

And remember, `Promise.resolve(..)` doesn't do anything if what you pass is already a genuine Promise; it just returns the value directly. So there's no overhead to calling `Promise.resolve(..)` on values that you don't know the nature of, if one happens to already be a genuine Promise.

then(..) and catch(..)

Each Promise instance (**not** the `Promise` API namespace) has `then(..)` and `catch(..)` methods, which allow registering of fulfillment and rejection handlers for the Promise. Once the Promise is resolved, one or the other of these handlers will be called, but not both, and it will always be called asynchronously (see "Jobs" in Chapter 1).

`then(..)` takes one or two parameters, the first for the fulfillment callback, and the second for the rejection callback. If either is omitted or is otherwise passed as a non-function value, a default callback is substituted respectively. The default fulfillment callback simply passes the message along, while the default rejection callback simply rethrows (propagates) the error reason it receives.

`catch(..)` takes only the rejection callback as a parameter, and automatically substitutes the default fulfillment callback, as just discussed. In other words, it's equivalent to `then(null,..)`:

```
p.then( fulfilled );

p.then( fulfilled, rejected );

p.catch( rejected ); // or `p.then( null, rejected )`
```

`then(..)` and `catch(..)` also create and return a new promise, which can be used to express Promise chain flow control. If the fulfillment or rejection callbacks have an exception thrown, the returned promise is rejected. If either callback returns an immediate, non-Promise, non-thenable value, that value is set as the fulfillment for the returned promise. If the fulfillment handler specifically returns a promise or thenable value, that value is unwrapped and becomes the resolution of the returned promise.

Promise.all([..]) and Promise.race([..])

The static helpers `Promise.all([ .. ])` and `Promise.race([ .. ])` on the ES6 `Promise` API both create a Promise as their return value. The resolution of that promise is controlled entirely by the array of promises that you pass in.

For `Promise.all([ .. ])`, all the promises you pass in must fulfill for the returned promise to fulfill. If any promise is rejected, the main returned promise is immediately rejected, too (discarding the results of any of the other promises). For fulfillment, you receive an `array` of all the passed in promises' fulfillment values. For rejection, you receive just the first promise rejection reason value. This pattern is classically called a "gate": all must arrive before the gate opens.

For `Promise.race([ .. ])`, only the first promise to resolve (fulfillment or rejection) "wins," and whatever that resolution is becomes the resolution of the returned promise. This pattern is classically called a "latch": first one to open the latch gets through. Consider:

```
var p1 = Promise.resolve( 42 );
var p2 = Promise.resolve( "Hello World" );
var p3 = Promise.reject( "Oops" );

Promise.race( [p1,p2,p3] )
.then( function(msg){
        console.log( msg );             // 42
} );

Promise.all( [p1,p2,p3] )
.catch( function(err){
        console.error( err );   // "Oops"
} );

Promise.all( [p1,p2] )
.then( function(msgs){
        console.log( msgs );    // [42,"Hello World"]
} );
```

Warning: Be careful! If an empty `array` is passed to `Promise.all([ .. ])`, it will fulfill immediately, but `Promise.race([ .. ])` will hang forever and never resolve.

The ES6 `Promise` API is pretty simple and straightforward. It's at least good enough to serve the most basic of async cases, and is a good place to start when rearranging your code from callback hell to something better.

But there's a whole lot of async sophistication that apps often demand which Promises themselves will be limited in addressing. In the next section, we'll dive into those limitations as motivations for the benefit of Promise libraries.

Promise Limitations

Many of the details we'll discuss in this section have already been alluded to in this chapter, but we'll just make sure to review these limitations specifically.

Sequence Error Handling

We covered Promise-flavored error handling in detail earlier in this chapter. The limitations of how Promises are designed -- how they chain, specifically -- creates a very easy pitfall where an error in a Promise chain can be silently ignored accidentally.

But there's something else to consider with Promise errors. Because a Promise chain is nothing more than its constituent Promises wired together, there's no entity to refer to the entire chain as a single *thing*, which means there's no external way to observe any errors that may occur.

If you construct a Promise chain that has no error handling in it, any error anywhere in the chain will propagate indefinitely down the chain, until observed (by registering a rejection handler at some step). So, in that specific case, having a reference to the *last* promise in the chain is enough (`p` in the following snippet), because you can register a rejection handler there, and it will be notified of any propagated errors:

```
// `foo(..)`, `STEP2(..)` and `STEP3(..)` are
// all promise-aware utilities

var p = foo( 42 )
```

```
.then( STEP2 )
.then( STEP3 );
```

Although it may seem sneakily confusing, `p` here doesn't point to the first promise in the chain (the one from the `foo(42)` call), but instead from the last promise, the one that comes from the `then(STEP3)` call.

Also, no step in the promise chain is observably doing its own error handling. That means that you could then register a rejection error handler on `p`, and it would be notified if any errors occur anywhere in the chain:

```
p.catch( handleErrors );
```

But if any step of the chain in fact does its own error handling (perhaps hidden/abstracted away from what you can see), your `handleErrors(..)` won't be notified. This may be what you want -- it was, after all, a "handled rejection" -- but it also may *not* be what you want. The complete lack of ability to be notified (of "already handled" rejection errors) is a limitation that restricts capabilities in some use cases.

It's basically the same limitation that exists with a `try..catch` that can catch an exception and simply swallow it. So this isn't a limitation **unique to Promises**, but it *is* something we might wish to have a workaround for.

Unfortunately, many times there is no reference kept for the intermediate steps in a Promise-chain sequence, so without such references, you cannot attach error handlers to reliably observe the errors.

Single Value

Promises by definition only have a single fulfillment value or a single rejection reason. In simple examples, this isn't that big of a deal, but in more sophisticated scenarios, you may find this limiting.

The typical advice is to construct a values wrapper (such as an `object` or `array`) to contain these multiple messages. This solution works, but it can be quite awkward and tedious to wrap and unwrap your messages with every single step of your Promise chain.

Splitting Values

Sometimes you can take this as a signal that you could/should decompose the problem into two or more Promises.

Imagine you have a utility `foo(..)` that produces two values (`x` and `y`) asynchronously:

```
function getY(x) {
	return new Promise( function(resolve,reject){
		setTimeout( function(){
			resolve( (3 * x) - 1 );
		}, 100 );
	} );
}

function foo(bar,baz) {
	var x = bar * baz;

	return getY( x )
	.then( function(y){
		// wrap both values into container
		return [x,y];
	} );
}

foo( 10, 20 )
.then( function(msgs){
	var x = msgs[0];
	var y = msgs[1];

	console.log( x, y );	// 200 599
} );
```

First, let's rearrange what `foo(..)` returns so that we don't have to wrap `x` and `y` into a single `array` value to transport through one Promise. Instead, we can wrap each value into its own promise:

```
function foo(bar,baz) {
	var x = bar * baz;

	// return both promises
	return [
		Promise.resolve( x ),
		getY( x )
	];
}

Promise.all(
	foo( 10, 20 )
)
.then( function(msgs){
	var x = msgs[0];
	var y = msgs[1];

	console.log( x, y );
} );
```

Is an `array` of promises really better than an `array` of values passed through a single promise? Syntactically, it's not much of an improvement.

But this approach more closely embraces the Promise design theory. It's now easier in the future to refactor to split the calculation of `x` and `y` into separate functions. It's cleaner and more flexible to let the calling code decide how to orchestrate the two promises -- using `Promise.all([ .. ])` here, but certainly not the only option -- rather than to abstract such details away inside of `foo(..)`.

Unwrap/Spread Arguments

The `var x = ..` and `var y = ..` assignments are still awkward overhead. We can employ some functional trickery (hat tip to Reginald Braithwaite, @raganwald on Twitter) in a helper utility:

```
function spread(fn) {
	return Function.apply.bind( fn, null );
}

Promise.all(
	foo( 10, 20 )
)
.then(
	spread( function(x,y){
		console.log( x, y );	// 200 599
	} )
)
```

That's a bit nicer! Of course, you could inline the functional magic to avoid the extra helper:

```
Promise.all(
	foo( 10, 20 )
)
.then( Function.apply.bind(
	function(x,y){
		console.log( x, y );	// 200 599
	},
	null
) );
```

These tricks may be neat, but ES6 has an even better answer for us: destructuring. The array destructuring assignment form looks like this:

```
Promise.all(
	foo( 10, 20 )
)
.then( function(msgs){
	var [x,y] = msgs;

	console.log( x, y );	// 200 599
} );
```

But best of all, ES6 offers the array parameter destructuring form:

```
Promise.all(
	foo( 10, 20 )
)
.then( function([x,y]){
	console.log( x, y );	// 200 599
} );
```

We've now embraced the one-value-per-Promise mantra, but kept our supporting boilerplate to a minimum!

Note: For more information on ES6 destructuring forms, see the *ES6 & Beyond* title of this series.

Single Resolution

One of the most intrinsic behaviors of Promises is that a Promise can only be resolved once (fulfillment or rejection). For many async use cases, you're only retrieving a value once, so this works fine.

But there's also a lot of async cases that fit into a different model -- one that's more akin to events and/or streams of data. It's not clear on the surface how well Promises can fit into such use cases, if at all. Without a significant abstraction on top of Promises, they will completely fall short for handling multiple value resolution.

Imagine a scenario where you might want to fire off a sequence of async steps in response to a stimulus (like an event) that can in fact happen multiple times, like a button click.

This probably won't work the way you want:

```
// `click(..)` binds the `"click"` event to a DOM element
// `request(..)` is the previously defined Promise-aware Ajax

var p = new Promise( function(resolve,reject){
	click( "#mybtn", resolve );
} );

p.then( function(evt){
	var btnID = evt.currentTarget.id;
	return request( "http://some.url.1/?id=" + btnID );
} )
.then( function(text){
	console.log( text );
} );
```

The behavior here only works if your application calls for the button to be clicked just once. If the button is clicked a second time, the `p` promise has already been resolved, so the second `resolve(..)` call would be ignored.

Instead, you'd probably need to invert the paradigm, creating a whole new Promise chain for each event firing:

```
click( "#mybtn", function(evt){
	var btnID = evt.currentTarget.id;

	request( "http://some.url.1/?id=" + btnID )
	.then( function(text){
		console.log( text );
```

```
        } );
} );
```

This approach will *work* in that a whole new Promise sequence will be fired off for each `"click"` event on the button.

But beyond just the ugliness of having to define the entire Promise chain inside the event handler, this design in some respects violates the idea of separation of concerns/capabilities (SoC). You might very well want to define your event handler in a different place in your code from where you define the *response* to the event (the Promise chain). That's pretty awkward to do in this pattern, without helper mechanisms.

Note: Another way of articulating this limitation is that it'd be nice if we could construct some sort of "observable" that we can subscribe a Promise chain to. There are libraries that have created these abstractions (such as RxJS -- http://rxjs.codeplex.com/), but the abstractions can seem so heavy that you can't even see the nature of Promises anymore. Such heavy abstraction brings important questions to mind such as whether (sans Promises) these mechanisms are as *trustable* as Promises themselves have been designed to be. We'll revisit the "Observable" pattern in Appendix B.

Inertia

One concrete barrier to starting to use Promises in your own code is all the code that currently exists which is not already Promise-aware. If you have lots of callback-based code, it's far easier to just keep coding in that same style.

"A code base in motion (with callbacks) will remain in motion (with callbacks) unless acted upon by a smart, Promises-aware developer."

Promises offer a different paradigm, and as such, the approach to the code can be anywhere from just a little different to, in some cases, radically different. You have to be intentional about it, because Promises will not just naturally shake out from the same ol' ways of doing code that have served you well thus far.

Consider a callback-based scenario like the following:

```
function foo(x,y,cb) {
        ajax(
                "http://some.url.1/?x=" + x + "&y=" + y,
                cb
        );
}

foo( 11, 31, function(err,text) {
        if (err) {
                console.error( err );
        }
        else {
                console.log( text );
        }
} );
```

Is it immediately obvious what the first steps are to convert this callback-based code to Promise-aware code? Depends on your experience. The more practice you have with it, the more natural it will feel. But certainly, Promises don't just advertise on the label exactly how to do it -- there's no one-size-fits-all answer -- so the responsibility is up to you.

As we've covered before, we definitely need an Ajax utility that is Promise-aware instead of callback-based, which we could call `request(..)`. You can make your own, as we have already. But the overhead of having to manually define Promise-aware wrappers for every callback-based utility makes it less likely you'll choose to refactor to Promise-aware coding at all.

Promises offer no direct answer to that limitation. Most Promise libraries do offer a helper, however. But even without a library, imagine a helper like this:

```
// polyfill-safe guard check
if (!Promise.wrap) {
        Promise.wrap = function(fn) {
                return function() {
                        var args = [].slice.call( arguments );
                        return new Promise( function(resolve,reject){
                                fn.apply(
                                        null,
                                        args.concat( function(err,v){
                                                if (err) {
                                                        reject( err );
                                                }
                                                else {
                                                        resolve( v );
                                                }
                                        } )
                                );
                        } );
                };
        };
}
```

OK, that's more than just a tiny trivial utility. However, although it may look a bit intimidating, it's not as bad as you'd think. It takes a function that expects an error-first style callback as its last parameter, and returns a new one that automatically creates a Promise to return, and substitutes the callback for you, wired up to the Promise fulfillment/rejection.

Rather than waste too much time talking about *how* this `Promise.wrap(..)` helper works, let's just look at how we use it:

```
var request = Promise.wrap( ajax );

request( "http://some.url.1/" )
.then( .. )
..
```

Wow, that was pretty easy!

`Promise.wrap(..)` does **not** produce a Promise. It produces a function that will produce Promises. In a sense, a Promise-producing function could be seen as a "Promise factory." I propose "promisory" as the name for such a thing ("Promise" + "factory").

The act of wrapping a callback-expecting function to be a Promise-aware function is sometimes referred to as "lifting" or "promisifying". But there doesn't seem to be a standard term for what to call the resultant function other than a "lifted function", so I like "promisory" better as I think it's more descriptive.

Note: Promisory isn't a made-up term. It's a real word, and its definition means to contain or convey a promise. That's exactly what these functions are doing, so it turns out to be a pretty perfect terminology match!

So, `Promise.wrap(ajax)` produces an `ajax(..)` promisory we call `request(..)`, and that promisory produces Promises for Ajax responses.

If all functions were already promisories, we wouldn't need to make them ourselves, so the extra step is a tad bit of a shame. But at least the wrapping pattern is (usually) repeatable so we can put it into a `Promise.wrap(..)` helper as shown to aid our promise coding.

So back to our earlier example, we need a promisory for both `ajax(..)` and `foo(..)`:

```
// make a promisory for `ajax(..)`
var request = Promise.wrap( ajax );

// refactor `foo(..)`, but keep it externally
// callback-based for compatibility with other
// parts of the code for now -- only use
// `request(..)`'s promise internally.
function foo(x,y,cb) {
        request(
                "http://some.url.1/?x=" + x + "&y=" + y
        )
        .then(
				function fulfilled(text){
					cb( null, text );
				},
				cb
			);
	}

	// now, for this code's purposes, make a
	// promisory for `foo(..)`
	var betterFoo = Promise.wrap( foo );

	// and use the promisory
	betterFoo( 11, 31 )
	.then(
		function fulfilled(text){
			console.log( text );
		},
		function rejected(err){
			console.error( err );
		}
	);
```

Of course, while we're refactoring `foo(..)` to use our new `request(..)` promisory, we could just make `foo(..)` a promisory itself, instead of remaining callback-based and needing to make and use the subsequent `betterFoo(..)` promisory. This decision just depends on whether `foo(..)` needs to stay callback-based compatible with other parts of the code base or not.

Consider:

```
	// `foo(..)` is now also a promisory because it
	// delegates to the `request(..)` promisory
	function foo(x,y) {
		return request(
			"http://some.url.1/?x=" + x + "&y=" + y
		);
	}

	foo( 11, 31 )
	.then( .. )
	..
```

While ES6 Promises don't natively ship with helpers for such promisory wrapping, most libraries provide them, or you can make your own. Either way, this particular limitation of Promises is addressable without too much pain (certainly compared to the pain of callback hell!).

Promise Uncancelable

Once you create a Promise and register a fulfillment and/or rejection handler for it, there's nothing external you can do to stop that progression if something else happens to make that task moot.

Note: Many Promise abstraction libraries provide facilities to cancel Promises, but this is a terrible idea! Many developers wish Promises had natively been designed with external cancelation capability, but the problem is that it would let one consumer/observer of a Promise affect some other consumer's ability to observe that same Promise. This violates the future-value's trustability (external immutability), but moreover is the embodiment of the "action at a distance" anti-pattern (http://en.wikipedia.org/wiki/Action_at_a_distance_%28computer_programming%29). Regardless of how useful it seems, it will actually lead you straight back into the same nightmares as callbacks.

Consider our Promise timeout scenario from earlier:

```
	var p = foo( 42 );

	Promise.race( [
		p,
		timeoutPromise( 3000 )
	] )
```

```
.then(
        doSomething,
        handleError
);

p.then( function(){
        // still happens even in the timeout case :(
} );
```

The "timeout" was external to the promise `p`, so `p` itself keeps going, which we probably don't want.

One option is to invasively define your resolution callbacks:

```
var OK = true;

var p = foo( 42 );

Promise.race( [
        p,
        timeoutPromise( 3000 )
        .catch( function(err){
                OK = false;
                throw err;
        } )
] )
.then(
        doSomething,
        handleError
);

p.then( function(){
        if (OK) {
                // only happens if no timeout! :)
        }
} );
```

This is ugly. It works, but it's far from ideal. Generally, you should try to avoid such scenarios.

But if you can't, the ugliness of this solution should be a clue that *cancelation* is a functionality that belongs at a higher level of abstraction on top of Promises. I'd recommend you look to Promise abstraction libraries for assistance rather than hacking it yourself.

Note: My *asynquence* Promise abstraction library provides just such an abstraction and an `abort()` capability for the sequence, all of which will be discussed in Appendix A.

A single Promise is not really a flow-control mechanism (at least not in a very meaningful sense), which is exactly what *cancelation* refers to; that's why Promise cancelation would feel awkward.

By contrast, a chain of Promises taken collectively together -- what I like to call a "sequence" -- *is* a flow control expression, and thus it's appropriate for cancelation to be defined at that level of abstraction.

No individual Promise should be cancelable, but it's sensible for a *sequence* to be cancelable, because you don't pass around a sequence as a single immutable value like you do with a Promise.

Promise Performance

This particular limitation is both simple and complex.

Comparing how many pieces are moving with a basic callback-based async task chain versus a Promise chain, it's clear Promises have a fair bit more going on, which means they are naturally at least a tiny bit slower. Think back to just the simple list of trust guarantees that Promises offer, as compared to the ad hoc solution code you'd have to layer on top of callbacks to achieve the same protections.

More work to do, more guards to protect, means that Promises *are* slower as compared to naked, untrustable callbacks. That much is obvious, and probably simple to wrap your brain around.

But how much slower? Well... that's actually proving to be an incredibly difficult question to answer absolutely, across the board.

Frankly, it's kind of an apples-to-oranges comparison, so it's probably the wrong question to ask. You should actually compare whether an ad-hoc callback system with all the same protections manually layered in is faster than a Promise implementation.

If Promises have a legitimate performance limitation, it's more that they don't really offer a line-item choice as to which trustability protections you want/need or not -- you get them all, always.

Nevertheless, if we grant that a Promise is generally a *little bit slower* than its non-Promise, non-trustable callback equivalent -- assuming there are places where you feel you can justify the lack of trustability -- does that mean that Promises should be avoided across the board, as if your entire application is driven by nothing but must-be-utterly-the-fastest code possible?

Sanity check: if your code is legitimately like that, **is JavaScript even the right language for such tasks?** JavaScript can be optimized to run applications very performantly (see Chapter 5 and Chapter 6). But is obsessing over tiny performance tradeoffs with Promises, in light of all the benefits they offer, *really* appropriate?

Another subtle issue is that Promises make *everything* async, which means that some immediately (synchronously) complete steps still defer advancement of the next step to a Job (see Chapter 1). That means that it's possible that a sequence of Promise tasks could complete ever-so-slightly slower than the same sequence wired up with callbacks.

Of course, the question here is this: are these potential slips in tiny fractions of performance *worth* all the other articulated benefits of Promises we've laid out across this chapter?

My take is that in virtually all cases where you might think Promise performance is slow enough to be concerned, it's actually an anti-pattern to optimize away the benefits of Promise trustability and composability by avoiding them altogether.

Instead, you should default to using them across the code base, and then profile and analyze your application's hot (critical) paths. Are Promises *really* a bottleneck, or are they just a theoretical slowdown? Only *then*, armed with actual valid benchmarks (see Chapter 6) is it responsible and prudent to factor out the Promises in just those identified critical areas.

Promises are a little slower, but in exchange you're getting a lot of trustability, non-Zalgo predictability, and composability built in. Maybe the limitation is not actually their performance, but your lack of perception of their benefits?

Review

Promises are awesome. Use them. They solve the *inversion of control* issues that plague us with callbacks-only code.

They don't get rid of callbacks, they just redirect the orchestration of those callbacks to a trustable intermediary mechanism that sits between us and another utility.

Promise chains also begin to address (though certainly not perfectly) a better way of expressing async flow in sequential fashion, which helps our brains plan and maintain async JS code better. We'll see an even better solution to *that* problem in the next chapter!