Demystifying Promises

kraigb – first draft, neither edited nor reviewed.

(Note—it’s partly written as a blog post, or series of posts, and is morphing into an appendix for my book; it’s in process.)

# Introduction

For some time I’ve been wanting to go deeper into the subject of promises in WinJS, and how to understand what’s happening inside the more “interesting” pieces of code involving promises. Part of the motivation here is to understand a bit of code that I showed in Chapter 5 of [my book](http://aka.ms/BrockschmidtBook) with the multistage batching renderer of a ListView:

function createBatch(waitPeriod) {

var batchTimeout = WinJS.Promise.as();

var batchedItems = [];

function completeBatch() {

var callbacks = batchedItems;

batchedItems = [];

for (var i = 0; i < callbacks.length; i++) {

callbacks[i]();

}

}

return function () {

batchTimeout.cancel();

batchTimeout = WinJS.Promise.timeout(waitPeriod || 64).then(completeBatch);

var delayedPromise = new WinJS.Promise(function (c) {

batchedItems.push(c);

});

return function (v) {

return delayedPromise.then(function () {

return v;

});

};

};

}

Frankly, when I included this in my book I had no idea how it worked and didn’t have the time to work though it in detail, which was also true for some other code I’ve seen in which promises abound. Because I figure many of you are in the same boat, I wanted to do a series of posts here to document my journey through this subject. This also gives promises a complete treatment on their own, rather than only be discussed out of necessity in other contexts, such as using or writing asynchronous WinRT APIs.

We’ll start with the basics, outside of WinJS, to make sure that we really understand what a promise is, what it represents, and its exact nature as a programming construct. This is very important because once we start looking into code like the snippet above, we must clearly separate what’s happening within the promise construct from the other participants in the scene who are consuming the promises.

We’ll then talk a bit about the benefits of promises versus other methods that might accomplish the same ends. Next we’ll look at the full construct of promises, specifically to understand the complexities that start to arise when you get down to the gritty details—and how WinJS takes care of those complexities with the [*WinJS.Promise*](http://msdn.microsoft.com/en-us/library/windows/apps/br211867.aspx) class.

With all this we’ll finally be in a position to look at various interesting code snippets involving promises, including the one above, and understand how they work. And with that, we will have demystified promises!

# What is a Promise?

This whole notion of promises can be rather mysterious in itself, especially because promises are typically wrapped up with asynchronous functions coming out of the WinRT APIs. Indeed, when you start writing Windows Store apps, you first encounter a promise just because you’re trying to do something with WinRT, and that API happens to be asynchronous. So most of us have probably learned about promises in the context of async methods, possibly borrowing some code from a sample to get the job done without truly understanding how it all works.

Although they’re very useful for async programming, promises in and of themselves have nothing to do with async. Let’s be clear about this and do the beginner’s mind Zen thing here. Strip away anything you might already know about them and start from the beginning:

**Key Point #1:** *Promises, in themselves, are nothing more than a programming construct.*

And taking the Zen thing further, let’s go even further back and ask, what is a programming construct? It’s some combination of functions, statements, and variables that define a specific way to accomplish a task. For example, a *for* loop is a programming construct whose purpose is to iterate over a collection. A *for* loop is a way of saying, “for each item in this collection, perform these actions” (hence the creation of *forEach*!). You use such a construct anytime you need to accomplish this particular purpose, and you know it well because you’ve practiced it so often!

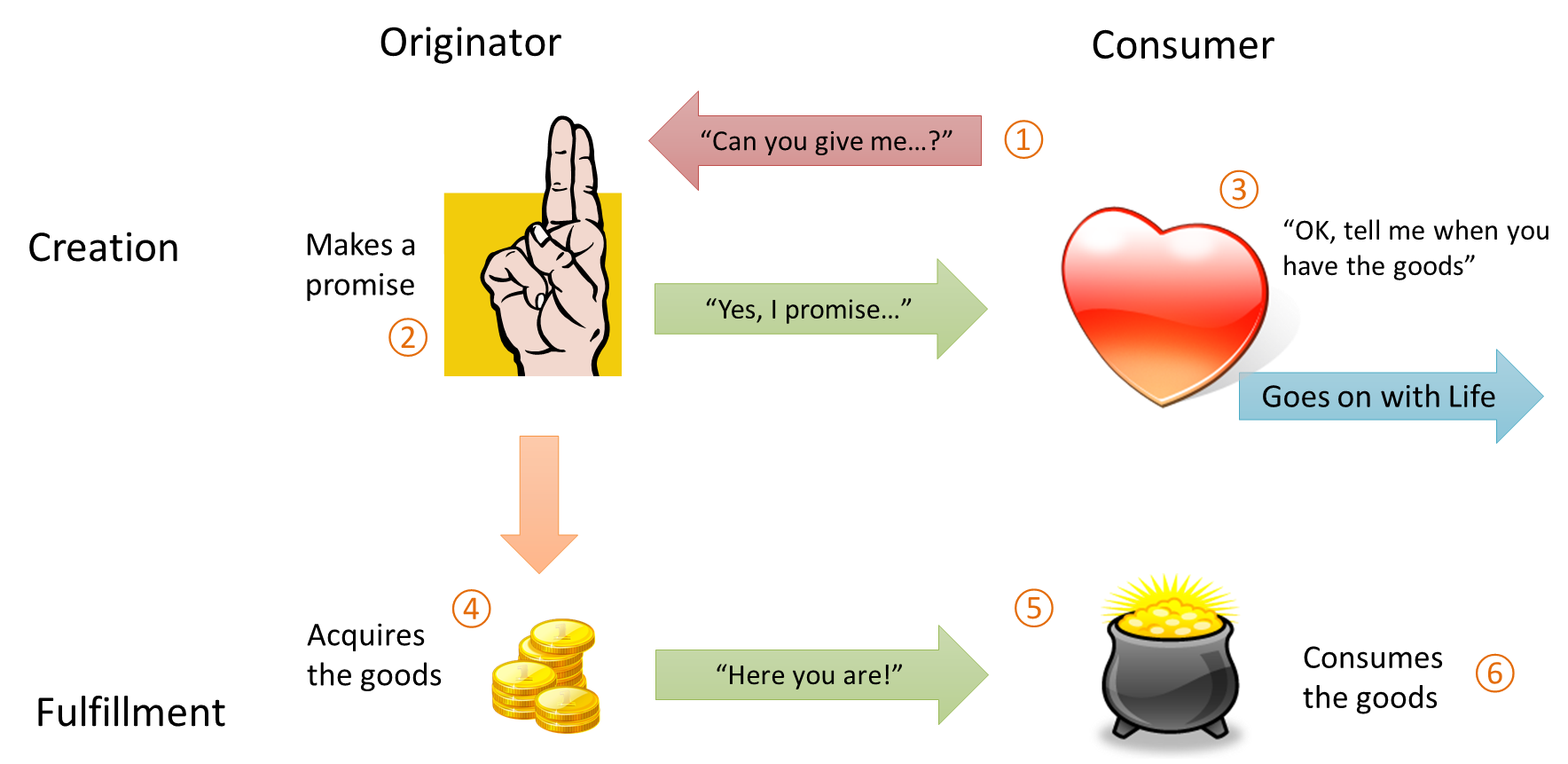
A promise is really nothing different—it’s a particular code structure for a specific purpose, namely the delivery of some value that might not yet be available future:

**Key Point #2**: *A promise represents a value that might be available at some point in the future.*

In this way, a promise is essentially the same as we use the term in human relationships. If I say to you, “I promise to deliver a dozen donuts,” then clearly I don’t have those donuts right now, but I assume that I’ll have them some time in the future, and when I do, I’ll deliver them.

# The Core Promise Relationship

A promise, then, implies a relationship between two people or, to be more generic, two *agents*. There is (a) the **originator** who makes the promise, that is, the one who has some goods to deliver, and (b) the **consumer** or recipient of that promise, who will also be the later recipient of the goods (and can do with them whatever it wants, like give them to someone else). There are also two *stages* of this relationship, **creation** and **fulfillment**. All this is illustrated in the diagram below:



Having two stages of the relationship is what bring up the asynchronous business. Let’s see how by following the flow of the relationship with the numbers in the diagram above:

1. The relationship begins when the consumer asks an originator for something, “Can you give me…?” This is what happens when an app calls some API that provides a promise rather than an immediate value.
2. The originator creates a promise for the goods in question and delivers that promise to the consumer.
3. The consumer acknowledges receipt of the promise, telling it how the promise should let the consumer know when the goods are ready. It’s like saying, “OK, call this number when you’re ready,” after which the consumer simply goes on with its life (asynchronously) instead of waiting (synchronously).
4. Meanwhile, the originator works to acquire the promised goods. Perhaps it has to manufacture the goods or acquire them from somewhere else, thus the relationship here assumes that those goods aren’t necessarily sitting around (even though they could be). This is the other place where asynchronous behavior arises, because acquisition can take an indeterminate amount of time.
5. Once the originator has the goods, it brings them to the consumer.
6. The consumer now has what it originally asked for, and can consume the goods as desired.

Now if you’re clever enough, you might have noticed that if you eliminate part of the diagram, namely the stuff around (3) and the arrow that says “Yes, I promise…,” you have a diagram of a simple synchronous delivery model. Which brings us to this:

**Key Point #3**: *Receiving a promise* *gives the consumer a chance to do something with its time (like being responsive to other requests), while it waits for the originator to get its act together and deliver the promised goods.*

And that’s the whole purpose of having asynchronous APIs in the first place! It’s like the difference between waiting in line at a some restaurant’s drive-through for a potentially very long time (the synchronous model) and calling out for pizza delivery (the asynchronous model): the latter gives you the freedom to do other things while you’re waiting for the delivery of your dinner.

# The Full Promise Relationship

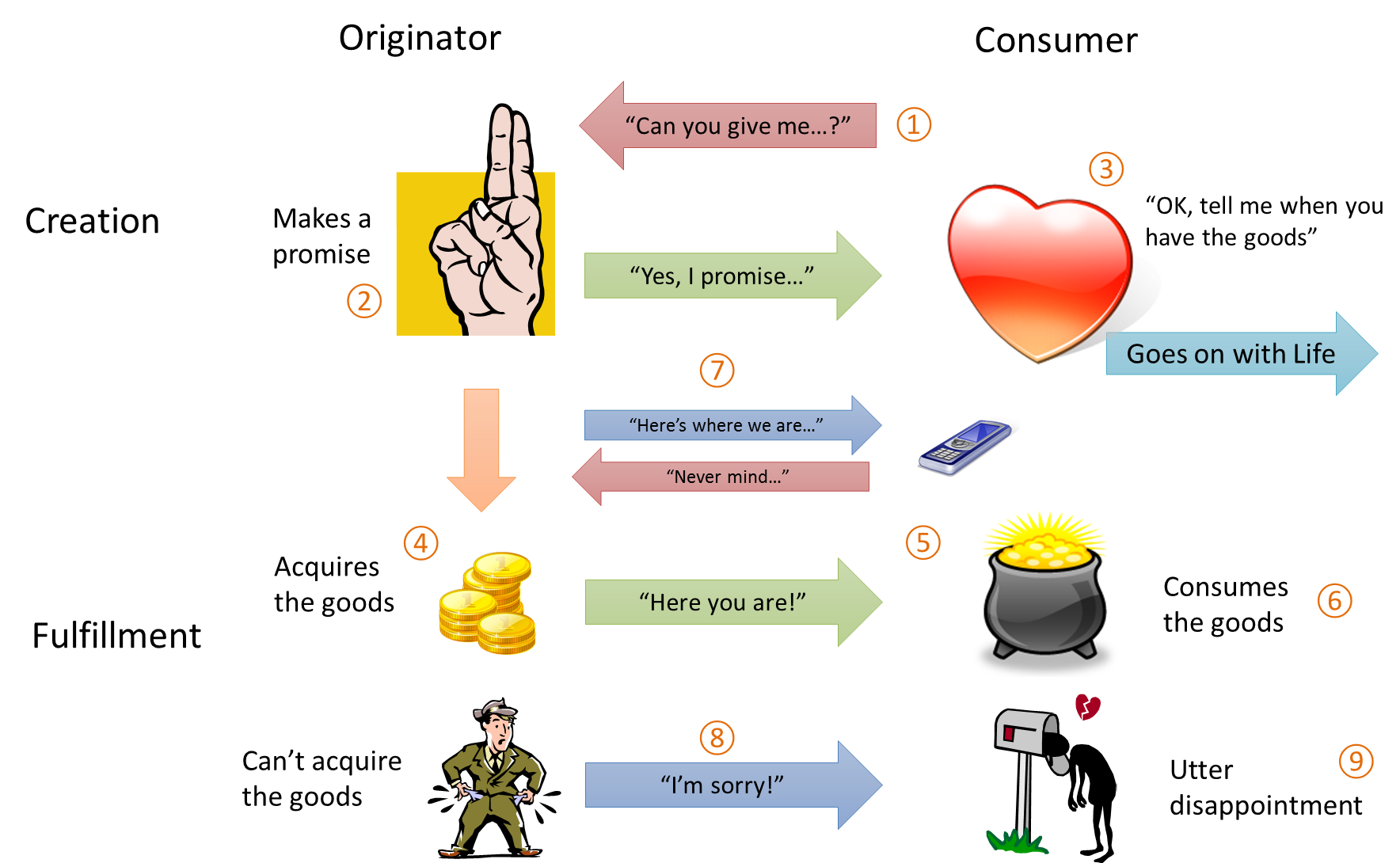
You’ve certainly made promises in your life, and have had promises made to you. Although many of those promises have been fulfilled, the reality is that many promises are broken—it is possible for the pizza delivery person to have an accident on the way to your home! Broken promises are just a fact of life, one that we have to accept, both in our personal lives and in asynchronous programming.

Within the promise relationship, then, this means that originator of a promise needs a way to say, “Well, I’m sorry, but I can’t make good on this promise.” Likewise, the recipient needs a way to know that this is the case.

Secondly, as consumers, we can sometimes be rather impatient about promises made to us. When a shipping company makes a promise to deliver a package by a certain date, we want to be able to look up the tracking number and see where that package is! So if the originator can track its progress in fulfilling its promise, the consumer also needs a way to receive that information.

And third, the consumer can also tell the originator that it no longer needs whatever it asked for earlier. That is, the consumer needs the ability to cancel the order or request.

This extended or full relationship is illustrated in the figure below:



Here we’ve added:

1. While the originator is attempting to acquire the goods, it can let the consumer know what’s happening with periodic updates. The consumer can also let the originator know that it no longer needs the promise fulfilled (cancellation).
2. If the originator fails to acquire the goods, it has to apologize with the understanding that there’s really nothing more it could have done. (“The Internet is down, you know?”)
3. If the promise is broken, the consumer has to deal with it as best it can!

With a clear understanding of the relationships involved in making, fulfilling, and breaking a promise, and reporting progress, we can now translate all that into terms of programming constructs.

# The Promise Construct (Core Relationship)

To fulfill the core relationship of a promise between originator and consumer, we need the following:

1. A means to create a promise and attach it to whatever results are involved.
2. A means to tell the consumer when the goods are available, which means some kind of callback function into the consumer.

The first requirement generally means that the originator defines an object class of some kind that internally wraps whatever process is needed to obtain the result. An instance of such a class would be created by an asynchronous API and returned to the caller.

For the second requirement, there are two approaches we can take. One way is to have a simple property on the promise object to which the consumer assigns the callback function. The other way is to have a method on the promise to which the consumer passes its callback. Of the two, the latter (using a method) gives the originator more flexibility in how it fulfills that promise, because until a consumer assigns a callback—which is also called *subscribing* to the promises—the originator can hold off on starting the underlying work. You know how it is—there’s work you know you need to do, but you just don’t get around to it until someone actually gives you a deadline! Using a method call thus tells the originator that the consumer is now truly wanting the results.[[1]](#footnote-1) Until that time, the promise object can simply wait in stasis and potentially be passed around between different consumers.

In the definition of a promise that’s evolved within the JavaScript community, known as [Common JS/Promises A](http://wiki.commonjs.org/wiki/Promises/A) (the specification that WinJS and WinRT follow), the method for this second requirement is called **then**. In fact, this is the very definition of a promise:

**Key Point #4:** *a promise is an object that has a property named ‘then’ whose value is a function.*

That’s it. In fact, the static WinJS function [*WinJS.Promise.is*](http://msdn.microsoft.com/en-us/library/windows/apps/br211765.aspx), which tests whether a given object is a promise, is implemented as follows:

is: function Promise\_is(value) {

return value && typeof value === "object" && typeof value.then === "function";

}

Within the core relationship, **then** takes one argument: a consumer-implemented callback function known as the *completed handler*. (This is also called a fulfilled handler, but I prefer the first term.) Here’s how it fits into the core relationship diagram shown earlier (using the same number labels):

1. The consumer calls some API that returns a promise. The specific API in question typically defines the type of object being asked for. In WinRT, for example, the [*Geolocator.getGeolocationAsync*](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.geolocation.geolocator.getgeopositionasync.aspx) method return a promise whose result is a [*Geoposition*](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.geolocation.geoposition.aspx) object.
2. The originator creates a promise by instantiating an instance of whatever class it employs for its work. So long as that object has a **then** method, it can contain whatever other methods and properties it wants. Look again at Key Point #4 above: it stipulates only that a promise has a method called **then**, which neither requires nor prohibits anything else.
3. Once the consumer receives the promise and wants to know about fulfillment, it calls **then** to subscribe to the promise, passing a completed handler as the first argument. The promise must internally retain this function (unless the value is already available—see below). Note that **then** can be called multiple times, by different consumers, which means the promise must be able to maintain a list of completed handlers, not just a single one.
4. Meanwhile, the originator works to fulfill the promise. For example, the WinRT *getGeolocationAsync* API will be busy retrieving information from the device’s sensors or using an IP address-based method to approximate the user’s location.
5. When the originator has the result, it has the promise object call all its completed handlers (received through **then**) with the result.
6. Inside its completed handler, the consumer works with the data however it wants.

As you can see, a promise is again *just a programming construct* that manages the relationship between consumer and originator. Nothing more. In fact, it’s not necessary that any asynchronous work is involved: a promise can be used with results that are already known.

**Key Point #5:** *because* *promises inherently have nothing to do with asynchronous operations, a promise can immediately call any completed handler given to* ***then*** *if the result is already known.*

In such cases, the promise just adds the layer of the completed handler, which typically gets called as soon as it’s provided to the promise through **then** in step 3 rather than in step 5. While this adds overhead for known values, it allows both synchronous and asynchronous results to be treated identically, which is very beneficial with async programming in general.

To make the core promise construct clear and to also illustrate an asynchronous operation, let’s look at a few examples using a simple promise class of our own. The code for these can be found in the Promises example accompanying this post.

**Note**: as mentioned earlier, implementing a fully-functional promise class gets rather complex when you start addressing all the details, such as the need for **then** to return another promise of its own. For this reason, always use the [*WinJS.Promise*](http://msdn.microsoft.com/en-us/library/windows/apps/br211867.aspx) class with which you can easily create robust promises for your own asynchronous operations. The example I’m showing here are strictly for education purposes as they do not implement the full promise specification. Note that there are other promise implementations that follow the same Promise/A standard, so those can be used as well.

## Example #1: An Empty Promise!

Let’s say we have a function, *doSomethingForNothing*, whose results are an empty object, *{ }*, delivered through a promise:

//Originator code

function doSomethingForNothing() {

return new EmptyPromise();

}

We’ll get to the *EmptyPromise* class in a moment. First, assuming that *EmptyPromise* follows the definition and has a **then** method that accepts a completed handler, here’s how we would use the API in the consumer:

//Consumer code

var promise = doSomethingForNothing();

promise.then(function (results) {

console.log(JSON.stringify(results));

});

The output of this would be as follows:

{}

The consumer code could be shortened to just *doSomethingForNothing().then(function (results) { ... });* but we’ll keep the promise explicitly visible for clarity. Also note that you can call the argument passed to the completed handler (*results* above) whatever you want. It’s your code, and you’re in control.

Stepping through the consumer code above, we call *promise.then*, and sometime later the anonymous completed handler is called. How long that “sometime later” is, exactly, depends on the nature of the operation in question.

Let’s say that *doSomethingForNothing* already knows that it’s going to return an empty object for results. In that case, *EmptyPromise* can be implemented as follows (and please, no comments about the best way to do object-oriented JavaScript):

//Originator code

var EmptyPromise = function () {

this.\_value = {};

this.then = function (completedHandler) {

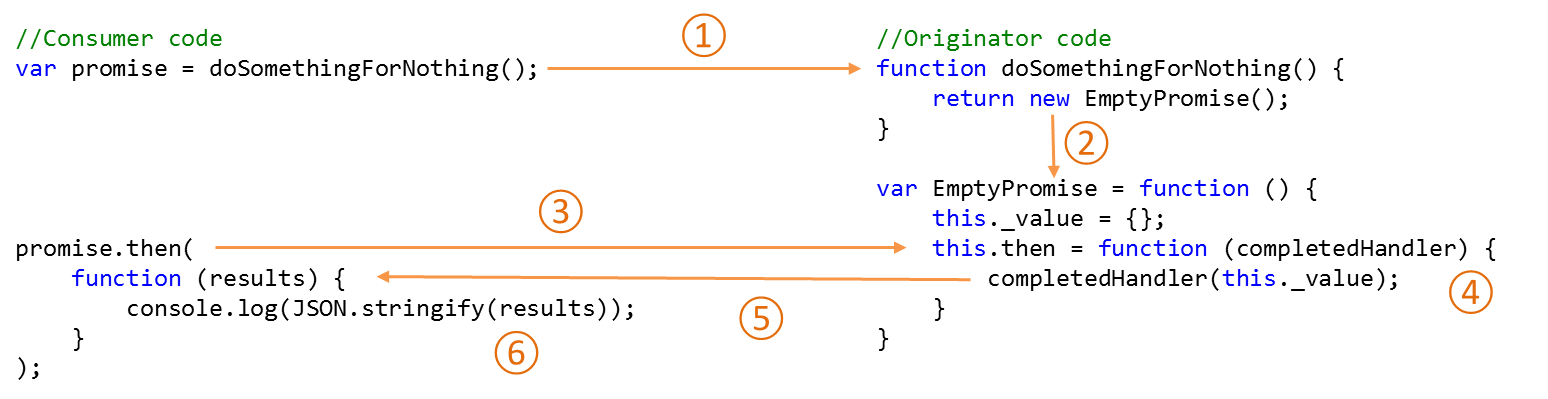
completedHandler(this.\_value);

}

}

When the originator does a *new* on this constructor, we get back an object that has a **then** method that accepts a completed handler. Because this promise already knows its result, its implementation of **then** can just turn around and call the given completed handler. This works no matter how many times **then** is called, even if the consumer passed that promise to another consumer.[[2]](#footnote-2)

Here’s how the code executes, identifying the steps in the core relationship:



Again, when a promise already knows its results, it can synchronously pass them to whatever completed handler it received through **then**. Here a promise is nothing more than an extra layer that delivers results through a callback rather than directly from a function. For pre-existing results, in other words, a promise is pure overhead. You can see this by placing another *console.log* call after *promise.then* and you’ll see that the {} result is logged before *promise.then* returns.

All this is implemented in scenario 1 of the Promises example.

## Example #2: An Empty Async Promise

While using promises with known values seems like a way to waste memory and CPU cycles for the sheer joy of it, promises become much more interesting when those values are obtained asynchronously.

Let’s change the earlier *EmptyPromise* class to do this. Instead of calling the completed handler right away, we’ll do that after a timeout:

var EmptyPromise = function () {

this.\_value = { };

this.then = function (completedHandler) {

//Simulate async work with a timeout so that we return before calling completedHandler

setTimeout(completedHandler, 100, this.\_value);

}

}

With the same consumer code as before, and suitable *console.log* calls, we’ll see that *promise.then* returns before the completed handler is called. Here’s the output:

promise created

returned from promise.then

{}

Indeed, **all** the synchronous code that follows *promise.then* will execute before the completed handler is called. This is because *setTimeout* has to wait for the app to yield the UI thread, even if that’s much longer than the timeout period itself. So if I do something synchronously obnoxious in the consumer code like the following, as in scenario 2 of the Promises example:

var promise = doSomethingForNothing();

console.log("promise created");

promise.then(function (results) {

console.log(JSON.stringify(results));

});

console.log("returned from promise.then");

//Block UI thread for a longer period than the timeout

var sum = 0;

for (var i = 0; i < 500000; i++) {

sum += i;

}

console.log("calculated sum = " + sum);

The output will be:

promise created

returned from promise.then

calculated sum = 1249999750000

{}

This tells us that for async operation we don’t have to worry about completed handlers being called before the current function is done executing. At the same time, if we have multiple completed handlers for different async operations, there’s no guarantee about the order in which they’ll complete. This is where you need to either nest or chain the operations, which we’ll return to later on.

**Note**: always keep in mind that while async operations typically spawn additional threads apart from the UI thread, all those results must eventually make their way back to the UI thread. If you have a large number of async operations running in parallel, the callbacks to the completed handlers on the UI thread can cause the app to become somewhat unresponsive. If this is the case, implement strategies to stagger those operations in time, e.g. using *setTimeout* or *setInterval* to separate them into batches.

## Example #3: Retrieving Data from a URI

As a more realistic example, let’s do some asynchronous work with meaningful results from *XMLHttpRequest*, as demonstrated in scenario 3 of the Promises example:

//Originator code

function doXhrGet(uri) {

return new XhrPromise(uri);

}

var XhrPromise = function (uri) {

this.then = function (completedHandler) {

var req = new XMLHttpRequest();

req.onreadystatechange = function () {

if (req.readyState === 4) {

if (req.status >= 200 && req.status < 300) {

completedHandler(req);

}

req.onreadystatechange = function () { };

}

};

req.open("GET", uri, true);

req.responseType = "";

req.send();

}

}

//Consumer code (note that the promise isn't explicit)

doXhrGet("http://kraigbrockschmidt.com/blog/?feed=rss2").then(function (results) {

console.log(results.responseText);

});

console.log("returned from promise.then");

The key feature in this code is that the asynchronous API we’re using within the promise does not itself involve promises, just like our use of *setTimeout* in the second example. *XMLHttpRequest.send* does its work asynchronously, but reports status through its *readystatechange* event. So what we’re really doing here is wrapping that API-specific async structure inside the more generalized structure of a promise.

It should be fairly obvious that this *XhrPromise* as I’ve defined it has some limitations—a real wrapper would be much more flexible for HTTP requests. Another problem is that if **then** is called more than once, this implementation will kick off additional HTTP requests rather than sharing the results among multiple consumers. So don’t use a class like this—instead, check out the [*WinJS.xhr*](http://msdn.microsoft.com/en-us/library/windows/apps/br229787.aspx) object, from which I shamelessly plagiarized this code in the first place!

# Benefits of Promises

So why wrap async operations within promises, as we did in the previous section? Why not just use functions like *XMLHttpRequest* straight up without all the added complexity? And why would we ever want to wrap known values into a promise that ultimately acts synchronously?

There are a number of reasons. First, by wrapping async operations within promises, the consuming code no longer has to know the specific callback structure for each API. Just in the examples we’ve written so far, methods like *setImmediate*, *setTimeout*, and *setInterval* take a callback function as an argument. *XMLHttpRequest* raises an event instead. Web workers, similarly, raise a generic *message* event, and other async APIs can pretty much do what they want. By wrapping every such operation with the promise structure, the consuming code becomes much more consistent.

A second reason is that when all async operations are represented by promises—and thus match async operations in the Windows Runtime and WinJS—we can start combining and composing them in interesting ways. We can chain dependent operations sequentially, for example, join promises to create a single promise that’s fulfilled when all the others are fulfilled (logical AND), or wrap promises together into a promise that’s fulfilled when the first one is fulfilled (logical OR).

This is where wrapping potentially known or existing values within promises, because they can also be combined with other asynchronous promises. In other words, promises provide a unified way to deal with values whether they’re delivered synchronously or asynchronously.

Promises also keep everything much simpler when we start working with the full promise relationship, as described earlier. For one, promises provide a structure wherein multiple consumers can each have their own completed handlers attached to the same promise. A real promise class—unlike the simple ones we’ve been working with—internally maintains a list of completed handlers and calls all of them when the value is available.

Furthermore, when error and progress handlers enter into the picture, as well as the ability to cancel an async operation through its promise, managing the differences between various async APIs becomes exceptionally cumbersome. Promises standardize all that, including the ability to manage multiple completed/error/progress callbacks along with a consistent *cancel* method.

**Key Point #6**: For whatever complexity they might have, promises are a very helpful tool when working with asynchronous operations.

Let’s now look at a more complete promise construct, which will help us understand why we want to use *WinJS.Promise*!

# The Full Promise Construct

Supporting the full promise relationship means that whatever class we use to implement a promise must provide additional capabilities beyond what we’ve seen so far:

* Support for error and (if appropriate) progress handlers.
* Support for multiple calls to **then**, that is, the promise must maintain an arbitrary number of handlers and share results between multiple consumers.
* Support for cancelation.
* Support for the ability to chain promises.

As an example, let’s expand the *XhrPromise* class from example #3 to support error and progress handlers through its **then** method, as well as multiple calls to **then**. This implementation is also a little more robust (allowing *null* handlers), and supports a *cancel* method. It can be found in scenario 4 of the Promises example project:

var XhrPromise = function (uri) {

this.\_req = null;

//Handler lists

this.\_cList = [];

this.\_eList = [];

this.\_pList = [];

this.then = function (completedHandler, errorHandler, progressHandler) {

var firstTime = false;

var that = this;

//Only create one operation for this promise

if (!this.\_req) {

this.\_req = new XMLHttpRequest();

firstTime = true;

}

//Save handlers in their respective arrays

completedHandler && this.\_cList.push(completedHandler);

errorHandler && this.\_eList.push(errorHandler);

progressHandler && this.\_pList.push(progressHandler);

this.\_req.onreadystatechange = function () {

var req = that.\_req;

if (req.\_canceled) { return; }

if (req.readyState === 4) { //Complete

if (req.status >= 200 && req.status < 300) {

that.\_cList.forEach(function (handler) {

handler(req);

});

} else {

that.\_eList.forEach(function (handler) {

handler(req);

});

}

req.onreadystatechange = function () { };

} else {

if (req.readyState === 3) { //Some data received

that.\_pList.forEach(function (handler) {

handler(req);

});

}

}

};

//Only start the operation on the first call to then

if (firstTime) {

this.\_req.open("GET", uri, true);

this.\_req.responseType = "";

this.\_req.send();

}

};

this.cancel = function () {

if (this.\_req != null) {

this.\_req.\_canceled = true;

this.\_req.abort;

}

}

}

The consumer of this promise can now attach multiple handlers, including error and progress as desired:

//Consumer code

var promise = doXhrGet("http://kraigbrockschmidt.com/blog/?feed=rss2");

console.log("promise created");

//Listen to promise with all the handlers (as separate functions for clarity)

promise.then(completedHandler, errorHandler, progressHandler);

console.log("returned from first promise.then call");

//Listen again with a second anonymous completed handler, the same error

//handler, and a null progress handler to test then's reentrancy.

promise.then(function (results) {

console.log("second completed handler called, response length = " + results.response.length);

}, errorHandler, null);

console.log("returned from second promise.then call");

function completedHandler (results) {

console.log("operation complete, response length = " + results.response.length);

}

function errorHandler (err) {

console.log("error in request");

}

function progressHandler (partialResult) {

console.log("progress, response length = " + partialResult.response.length);

}

As you can see, the first call to **then** uses distinct functions; the second call just uses an inline anonymous complete handler and passes *null* as the progress handler.

Running this code you’ll see that we pass through all the consumer code first, and the first call to **then** starts the operation. The progress handler will be called a number of times, then the completed handlers. The resulting output is as follows:

promise created

returned from first promise.then call

returned from second promise.then call

progress, response length = 4092

progress, response length = 8188

progress, response length = 12284

progress, response length = 16380

progress, response length = 20476

progress, response length = 24572

progress, response length = 28668

progress, response length = 32764

progress, response length = 36860

progress, response length = 40956

progress, response length = 45052

progress, response length = 49148

progress, response length = 53244

progress, response length = 57340

progress, response length = 61436

progress, response length = 65532

progress, response length = 69628

progress, response length = 73724

progress, response length = 73763

operation complete, response length = 73763

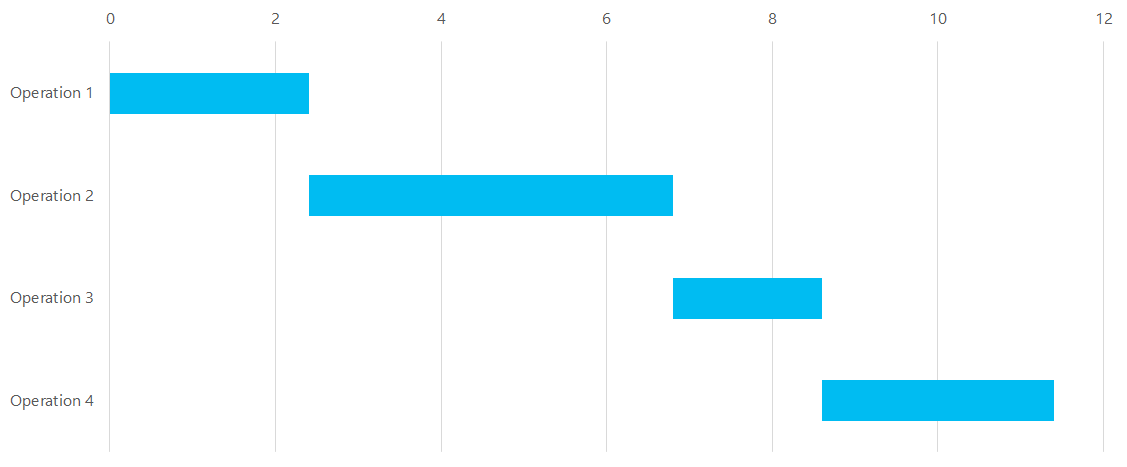
second completed handler called, response length = 73763

In the promise, you can see that we’re using three arrays to track all the handlers send to **then**, and iterate through those lists when making the necessary callbacks.

**Key Point #7**: Because there can be multiple consumers of the same promise and the same results must be delivered to each, results are considered *immutable*. That is, consumers cannot change those results.

As you can imagine, the code to handle lists of handlers is going to look pretty much the same in just about every promise class, so it makes sense to have some kind of standard implementation into which we can plug the specifics of the operation. As you probably expect by now, *WinJS.Promise* provides exactly this, as well as cancelation, as we’ll see later.

Our last concern is supporting the ability to chain promises. Although any number of asynchronous operations can run simultaneously, it’s a common need that results from one operation must be obtained before the next operation can begin, namely when those results are the inputs to the next operation. This is frequently encountered when doing file I/O in WinRT, where you need obtain a StorageFile object, open it, act on the resulting stream, then flush and close the stream, all of which are async operations. The flow of such operations can be illustrated as follows:



There are actually two ways to do this, nesting and chaining, as described in the following sections.

## Nesting Promises

One way to perform sequential async operations is to nest the calls that start each operation within the completed handler of the previous one. This actually doesn’t require anything special where the promises are concerned.

To see this, let’s expand upon that bit of UI-thread-blocking code from Example #2 that did a bunch of counting and turn it into an async operation—see scenario 5 of the Promises example:

function calculateIntegerSum(max, step) {

return new IntegerSummationPromise(max, step);

}

var IntegerSummationPromise = function (max, step) {

this.\_sum = null; //null means we haven't started the operation

this.\_cancel = false;

//Error condition

if (max < 1) {

return null;

}

//Handler lists

this.\_cList = [];

this.\_eList = [];

this.\_pList = [];

this.then = function (completedHandler, errorHandler, progressHandler) {

//Save handlers in their respective arrays

completedHandler && this.\_cList.push(completedHandler);

errorHandler && this.\_eList.push(errorHandler);

progressHandler && this.\_pList.push(progressHandler);

var that = this;

function iterate(args) {

for (var i = args.start; i < args.end; i++) {

that.\_sum += i;

};

if (i >= max) {

//Complete--dispatch results to completed handlers

that.\_cList.forEach(function (handler) {

handler(that.\_sum);

});

} else {

//Dispatch intermediate results to progress handlers

that.\_pList.forEach(function (handler) {

handler(that.\_sum);

});

//Do the next cycle

setImmediate(iterate, { start: args.end, end: Math.min(args.end + step, max) });

}

}

//Only start the operation on the first call to then

if (this.\_sum === null) {

this.\_sum = 0;

setImmediate(iterate, { start: 0, end: Math.min(step, max) });

}

};

this.cancel = function () {

this.\_cancel = true;

}

}

The *IntegerSummationPromise* class here is structurally similar to the *XhrPromise* class in scenario 4 to support multiple handlers and cancelation. Is asynchronous nature comes from using *setImmediate* to break up its computational cycles (meaning that it’s still running on the UI thread; we’d have to use a web worker to run on a separate thread).

To make sequential async operations interesting, let’s say we get our inputs for *calculateIntegerSum* from the following function (completely contrived, of course, with a promise that doesn’t support multiple handlers):

function getDesiredCount() {

return new NumberPromise();

}

var NumberPromise = function () {

this.\_value = 5000;

this.then = function (completedHandler) {

setTimeout(completedHandler, 100, this.\_value);

}

}

The calling (consumer) code looks like this, where I’ve eliminated the any intermediate variables and named functions:

getDesiredCount().then(function (count) {

console.log("getDesiredCount produced " + count);

calculateIntegerSum(count, 500).then(function (sum) {

console.log("calculated sum = " + sum);

},

null, //No error handler

//Progress handler

function (partialSum) {

console.log("partial sum = " + partialSum);

});

});

console.log("getDesiredCount.then returned");

The output of all this is as follows, where we can see that the consumer code executes through. Then the completed handler for the first promise is called, in which we start the second operation. That computation reports progress before delivering its final results:

getDesiredCount.then returned

getDesiredCount produced 5000

partial sum = 124750

partial sum = 499500

partial sum = 1124250

partial sum = 1999000

partial sum = 3123750

partial sum = 4498500

partial sum = 6123250

partial sum = 7998000

partial sum = 10122750

calculated sum = 12497500

Although the consumer code above is manageable with one nested operation, nesting gets messy when more operations are involved:

operation1().then(function (result1) {

operation2(result1).then(function (result2) {

operation3(result2).then(function (result3) {

operation4(result3).then(function (result4) {

operation5(result4).then(function (result5) {

//And so on

});

});

});

});

});

Imagine what this would look like if all the completed handlers did other work between each call! It’s very easy to get lost amongst all the braces and indents.

For this reason, real promises can also be chained in a way that sequential operations cleaner and easier to manage. When more than two operations are involved, chaining is typically the preferred approach.

## Chaining Promises

Chaining is made possible by a couple of requirements that part of the [Promises/A spec](http://wiki.commonjs.org/wiki/Promises/A) places on the **then** function:

*This function* [then] *should return a new promise that is fulfilled when the given [completedHandler] or errorHandler callback is finished. The value returned from the callback handler is the fulfillment value for the returned promise.*

Parsing this out, it means that any implementation of **then** must return a promise whose result is the return value of the completed handler given to **then**. With this characteristic, we can write consumer code in a chained manner that avoids indentation nightmares:

operation1().then(function (result1) {

return operation2(result1)

}).then(function (result2) {

return operation3(result2);

}).then(function (result3) {

return operation4(result3);

}).then(function (result4) {

return operation5(result4)

}).then(function (result5) {

//And so on

});

This structure makes it easier to read the sequence and is generally easier to work with. There’s a somewhat obvious flow from one operation to the next, where the return for each promise in the chain is essential. Applying this to the nesting example in the previous section (dropping all but the completed hander), we have the following:

getDesiredCount().then(function (count) {

return calculateIntegerSum(count, 500);

}).then(function (sum) {

console.log("calculated sum = " + sum);

});

Conceptually, when we write chained promises like this we can conveniently think of the return value from one completed handler as the promise that’s involved with the next **then**in the chain: the result from *calculateIntegerSum* shows up as theargument *sum* in the next completed handler.

However, at the point that *getDesiredCount.then* returns, we haven’t even started *calculateIntegerSum* yet. This means that whatever promise is returned from *getDesiredCount.then* is **some other intermediary promise**. This intermediary has its own **then** method and can receive its own completed, error, and progress handlers. But instead of waiting directly for some arbitrary asynchronous operation to finish, this intermediate promise is instead waiting on the results of the completed handler given to *getDesiredCount.then*. That is, the intermediate promise is a child or subsidiary of the promise that created it, such that it can manage its relationship on the parent’s completed handler.

Looking back at the code from scenario 5 in the last section, you’ll see that none of our **then** implementations don’t return anything (and are thus incomplete according to the spec). So what would it take to make it right?

Simplifying the matter by not supporting multiple calls to **then**, a promise class like *NumberPromise* would look something like this:

var NumberPromise = function () {

this.\_value = 5000;

this.then = function (completedHandler) {

setTimeout(valueAvailable, 100, this.\_value);

var that = this;

function valueAvailable(value) {

var retVal = completedHandler(value);

that.\_innerPromise.complete(retVal);

}

var retVal = new InnerPromise();

this.\_innerPromise = retVal;

return retVal;

}

}

Here, **then** creates an instance of a promise to which we pass whatever our completed handler gives us. That extra *InnerPromise.complete* method is the private communication channel through which we tell that inner promise that it can fulfill itself now, which means it calls whatever completed handlers *it* received in its own **then**.

So *InnerPromise* might start to look something like this (this is **not** complete):

var InnerPromise = function (value) {

this.\_value = value;

this.\_completedHandler = null;

var that = this;

//Internal helper

this.\_callComplete = function (value) {

that.\_completedHandler && that.\_completedHandler(value);

}

this.then = function (completedHandler) {

if (that.\_value) {

that.\_callComplete(that.\_value);

} else {

that.\_completedHandler = completedHandler;

}

};

//This tells us we have our fulfillment value

this.complete = function (value) {

that.\_value = value;

that.\_callComplete(value);

}

}

That is, we provide a **then** of our own (still incomplete), which will call its given handler if the value is already known, otherwise it saves the completed handler away (supporting only one such handler). We then assume that our parent promise calls the *complete* method when it gets a return value from whatever completed handler its holding. When that happens, this *InnerPromise* object can then fulfill itself.

So far so good. However, what happens when the parameter given to *complete* is itself a promise? That means this *InnerPromise* must wait for that other promise to finish, using yet another completed handler. Only then can it fulfill itself. Thus we need to do something like this within *InnerPromise*:

//Test if a value is a promise

function isPromise(p) {

return (p && typeof p === "object" && typeof p.then === "function");

}

//This tells us we have our fulfillment value

this.complete = function (value) {

that.\_value = value;

if (isPromise(value)) {

value.then(function (finalValue) {

that.\_callComplete(value);

})

} else {

that.\_callComplete(value);

}

}

We’re on a roll now. With this implementation, the consumer code that chains *getDesiredCount* and *calculateIntegerSum* works just fine, where the value of *sum* passed to the second completed handler is exactly what comes back from the computation.

But we still have a problem: *InnerPromise.then* does not itself return a promise, as it should, meaning that the chain dies right here. As such, we cannot chain another operation onto the sequence.

So what should *InnerPromise.then* return? Well, in the case where we already have the fulfillment value, we can just return ourselves (which is in the variable *that*). Otherwise we need to create yet another *InnerPromise* that’s wired up just as we did inside *NumberPromise*.

this.\_callComplete = function (value) {

if (that.\_completedHandler) {

var retVal = that.\_completedHandler(value);

that.\_innerPromise.complete(retVal);

}

}

this.then = function (completedHandler) {

if (that.\_value) {

var retVal = that.\_callComplete(that.\_value);

that.\_innerPromise.complete(retVal);

return that;

} else {

that.\_completedHandler = completedHandler;

//Create yet another inner promise for our return value

var retVal = new InnerPromise();

this.\_innerPromise = retVal;

return retVal;

}

};

With this in place, *InnerPromise* supports the kind of chaining we’re looking for. You can see this in scenario 6 of the Promises example. In this scenario you’ll find two buttons on the page. The first runs this condensed consumer code:

getDesiredCount().then(function (count) {

return calculateIntegerSum(count, 500);

}).then(function (sum1) {

console.log("calculated first sum = " + sum1);

return calculateIntegerSum(sum1, 500);

}).then(function (sum2) {

console.log("calculated second sum = " + sum2);

});

Where the output is:

calculated first sum = 499500  
calculated second sum = 124749875250

The second buttons runs the same consumer code but with explicit variables for the promises. It also turns on noisy logging from within the promise classes so we can see everything that’s going on. For this purpose, each promise class is tagged with an identifier so we can keep track of which is which. I won’t show the code, but the output is as follows:

p1 obtained, type = NumberPromise

InnerPromise1 created

p1.then returned, p2 obtained, type = InnerPromise1

InnerPromise1.then called

InnerPromise1.then creating new promise

InnerPromise2 created

p2.then returned, p3 obtained, type = InnerPromise2

InnerPromise2.then called

InnerPromise2.then creating new promise

InnerPromise3 created

p3.then returned (end of chain), returned promise type = InnerPromise3

NumberPromise completed.

p1 fulfilled, count = 1000

InnerPromise1.complete method called

InnerPromise1 calling IntegerSummationPromise.then

IntegerSummationPromise started

IntegerSummationPromise completed

IntegerSummationPromise fulfilled

InnerPromise1 calling completed handler

p2 fulfilled, sum1 = 499500

InnerPromise2.complete method called

InnerPromise2 calling IntegerSummationPromise.then

IntegerSummationPromise started

IntegerSummationPromise completed

IntegerSummationPromise fulfilled

InnerPromise2 calling completed handler

p3 fulfilled, sum2 = 124749875250

InnerPromise3.complete method called

InnerPromise3 calling completed handler

This log reveals what’s really going on in the chain. Because each operation in the sequence is asynchronous, we don’t have any solid values to pass to any of the completed handlers yet. But to execute the chain of **then** calls—*which happens all in a synchronous sequence*—there has to be somepromise in there to do the wiring. That’s the purpose of each *InnerPromise* instance. So in the first part of this log you can see that we’re basically creating a stack of these *InnerPromise* instances, each of which is waiting on another.

Once all the then methods return and we yield the UI thread, the async operations can start to fulfill themselves. You can see that the *NumberPromise* gets fulfilled, which means that *InnerPromise1* can be fulfilled with the return value from our first completed handler. That happens to be an *IntegerSummartionPromise*, so *InnerPromise1* attaches its own completed handler. When that handler is called, *InnerPromise1* can then call the second completed handler in the consumer code. The same thing then happens again with *InnerPromise2*, and so on, until the stack of inner promises are all fulfilled. It’s at this point that we run out of completed handlers to call, so the chain finally comes to an end.

**Key Point #8:** *Having* then *methods return another promise to allow chaining basically means that a chain of async operations builds a stack of intermediate promises to manage the connections between as-yet-unfulfilled promises and their completed handlers. As results start to come in, that stack is unwound such that the intermediate results are passed to the appropriate handler so that the next async operation can begin.*

Now let me be very clear about what we’ve done so far: the code above shows how chaining really works in the guts of promises, and yet there are still a number of unsolved problems, a few of which include:

* *InnerPromise.then* can handle only a single completed handler, and doesn’t provide for error and progress handlers.
* There’s no provision for handling exceptions in a completed handler, as specified by Promises/A.
* There’s no provision for cancellation of the chain, namely that canceling the promise produced by the chain as a whole should also cancel all the other promises involved.
* There are some repeated code structures, which beg for some kind of consolidation.
* This code really hasn’t been fully tested.

I will openly admit that I’m not right kind of developer to solve such problems—I’m primarily a writer! There are a number of subtle issues that start to arise when you put this kind of thing into real practice.

Fortunately, there is are software engineers who *adore* this kind of a challenge, and fortunately a number of them work in the WinJS team. As a result, they’ve done all the hard work for us already within the *WinJS.Promise* class. And we’re now ready to see—and fully appreciate!—what that library provides.

# Promises in WinJS (thank you, Microsoft!)

When writing Windows Store apps in JavaScript, promises pop up anywhere an asynchronous API is involved, and even at other times. And those promises all meet the necessary specifications, as their underlying classes as supplied by the operating system or from WinJS, which is considered a system library. From the consumer’s point of view, then, these promises can be used to their fullest extent possible—nested, chained, joined, and so forth. These promises can also be trusted to handle any number of handlers, process errors correctly, and basically handle any other subtleties that might arise.

**Key Point #9:** *Because the authors of WinJS have gone to great effort to provide highly robust and flexible promise implementations (for the full promise relationship), there is really no need to implement custom promise classes of your own. WinJS provides an extensible means to wrap any kind of async operation within a standardized and well-tested promise structure, so you can focus on the operation and not on the surrounding construct.*

Let’s then explore everything that WinJS has to offer where promises are concerned. Note that the entire source code for WinJS promises can be found in its base.js file, accessible through any app project that has a reference to WinJS (in Visual Studio’s solution explorer, expand References > Windows Library for JavaScript > js under a project and you’ll see base.js).

## The WinJS.Promise Class

Let’s start right in with the generalized promise class that WinJS provides: [*WinJS.Promise*](http://msdn.microsoft.com/en-us/library/windows/apps/br211866.aspx). In addition to supplying a few static members that we’ll see later, you create a new instance of this class whenever you want to encapsulate some operation within a promise. This allows you to focus on the nature of your operation, leaving *WinJS.Promise* to deal with the promise construct itself including implementations of **then** and *cancel* methods, management of handlers, and so forth. This includes handling complex cancellation processes involved with promise chains.

For instance, in scenario 6 of the Promises example, the *getDesiredCount* and *calculateIntegerSum* functions each create an instance of a distinct promise class to implement their async operations. And all that code got to be rather intricate, which means it will be hard to debug and maintain! With *WinJS.Promise*, we can dispense with those separate classes altogether. Instead, we just implement the operations directly within a function like *calculateIntegerSum*. This is how it now looks in scenario 7 (omitting a bit of code to handle cancelation):

function calculateIntegerSum(max, step) {

//The WinJS.Promise constructor's argument is a function that receives

//dispatchers for completed, error, and progress cases.

return new WinJS.Promise(function (completeDispatch, errorDispatch, progressDispatch) {

var sum = 0;

function iterate(args) {

for (var i = args.start; i < args.end; i++) {

sum += i;

};

if (i >= max) {

//Complete--dispatch results to completed handlers

completeDispatch(sum);

} else {

//Dispatch intermediate results to progress handlers

progressDispatch(sum);

setImmediate(iterate, { start: args.end, end: Math.min(args.end + step, max) });

}

}

setImmediate(iterate, { start: 0, end: Math.min(step, max) });

});

}

Clearly, this function still returns a promise, but it’s an instance of *WinJS.Promise* that’s essentially been configured to perform a specific operation. That “configuration,” if you will, is supplied by the function we passed to the *WinJS.Promise* constructor, referred to as the initializer. The core of this initializer function does exactly what we did within *IntegerSummationPromise.then* of scenario 6. The great thing is that we don’t need to manage all the handlers nor the details of returning another promise from **then**. That’s all taken care of for us.

All we need is a way to tell *WinJS.Promise* when to invoke the completed, error, and progress handlers it’s managing. That’s exactly what the three dispatcher arguments to the initializer function provide: calling these dispatchers loops over whatever handlers the promise received through **then**, just like we did manually in scenario 6. And again, we no longer need to worry about the structure details of creating a proper promise—we can simply concentrate on the core functionality that’s unique to our app.

A couple of other notes on *WinJS.Promise*:

* Doing *new WinJS.Promise()* (with no parameters) will throw an exception: an initializer function is required.
* If you don’t need the *errorDispatcher* and *progressDispatcher* methods, you don’t need to declare them as arguments in your function. JavaScript is nice that way!
* Any promise you get from WinJS (or WinRT for that matter) has a standard *cancel* method that cancels any pending async operation within the promise, if cancelation is supported. It has no effect on promises that contain already-known values.
* To support cancellation, the *WinJS.Promise* constructor also takes an optional second argument: a function to call if the promise’s *cancel* method is called. Here you halt whatever operation is underway. The *calculateIntegerSum* function of scenario 7, for example, has a simple function to set a *\_cancel* variable that the iteration loop checks before calling its next *setImmediate*.

## Shortcuts Functions for Creating WinJS Promises

In addition to creating a new *WinJS.Promise* from scratch, WinJS also provides a number of shortcuts to create promises for common scenarios. These are provided through static methods on the *WinJS.Promise* object: *as*, *wrap*, *timeout*, *join*, and *any*. There is also the *WinJS.xhr* class that encapsulates XmlHttpRequests. Scenario 8 of the Promises example provides a smorgasbord of short demonstrations of all of these.

### Wrapping Values as Promises: *WinJS.Promise.as* and *WinJS.Promise.wrap*

With *as* and *wrap*, both of these functions wrap a new promise around some existing value, with slight differences in behavior:

* [*WinJS.Promise.wrap*](http://msdn.microsoft.com/en-us/library/windows/apps/br229785.aspx) puts a promise wrapper around whatever non-promise value you provide. When you call its **then**, it will immediately call your completed handler with that same value. If you happen to give it a promise, the wrapped promise will be fulfilled with no results (*undefined*) when the other is fulfilled.
* [*WinJS.Promise.as*](http://msdn.microsoft.com/en-us/library/windows/apps/br211664.aspx) does the same thing as *wrap* except that if the value is already a promise, it just returns that promise as-is. In short, you use this function when the input value *might* be a promise and you want to avoid double-wrapping.

With both of these, you can just call *wrap()* or *as()* and the resulting promise will be fulfilled with *undefined*. This is useful because you can’t call *new WinJS.Promise()* to define a truly empty promise.

### Promises for *setTimeout*/*setImmediate*: *WinJS.Promise.timeout*

[*WinJS.Promise.timeout*](http://msdn.microsoft.com/en-us/library/windows/apps/br229729.aspx)is an interesting function because it has a triune nature.

* *WinJS.Promise.timeout()* creates a promise around *setImmediate,* which is to say, the shortest timeout you can get. In other words, this creates a promise that’s fulfilled as soon as the UI thread is yielded.
* *WinJS.Promise.timeout(n)* creates a promise that is fulfilled (with *undefined* results) after *n* milliseconds. This is basically a wrapper around *setTimeout(n)*.
* *WinJS.Promise.timeout(n, promise)* will return a promise that’s fulfilled with the results of *promise* if it’s fulfilled within *n* milliseconds. If *n* milliseconds elapse and *promise* is not yet fulfilled, that promise is canceled. This provides an easy way to add a timeout to some other async operation that doesn’t have one already. (In fact, the return value of this form of *timeout* is the same as the *promise* you give it; it just wires up a second timeout promise internally that will call *promise.cancel* if the period elapses.)

The implementation of these are easily seen in WinJS’s base.js file:

// Used for WinJS.Promise.timeout() and timeout(n)

function timeout(timeoutMS) {

var id;

return new WinJS.Promise(

function (c) {

if (timeoutMS) {

id = setTimeout(c, timeoutMS);

} else {

setImmediate(c);

}

},

function () {

if (id) {

clearTimeout(id);

}

}

);

}

// Used for WinJS.Promise.timeout(n, promise), where the timeout argument is

// a promise created with a call to the timeout method above.

function timeoutWithPromise(timeout, promise) {

var cancelPromise = function () { promise.cancel(); }

var cancelTimeout = function () { timeout.cancel(); }

timeout.then(cancelPromise);

promise.then(cancelTimeout, cancelTimeout);

return promise;

}

The *timeoutWithPromise* function above is a good bit of code to example more closely, as we’ll do in the last section, “Some Interesting Promise Code.”

Now you’ll notice that a promise from *timeout* itself has no fulfillment value. So how can we create a promise that will be fulfilled with some value after the timeout?

The answer is simple: just attach a completed handler to the timeout promise’s **then** that returns the value you want. The promise return from **then** is the one you want.

Remember the requirement that **then** returns a promise that’s fulfilled when the completed handler returns, and whose fulfillment value comes from that completed handler? We’re taking advantage of that here. For example, the following code creates a promise that’s fulfilled with the value 12345 after one second:

var p = WinJS.Promise.timeout(1000).then(function () { return 12345; });

In short, *WinJS.Promise.timeout(<n>).then(function () { <value> });* is a pattern for delivering a known value at some time in the future.

Indeed, you might have noticed in the previous section that I neglected to show how we’d implement the *getDesiredCount* function of scenario 6 with WinJS promises instead of our own *NumberPromise* class. If you recall, *NumberPromise* took a known value and returned it after a *setTimeout* interval of 100ms. Most of its implementation was just the overhead of the promise construct, overhead that we can wholly eliminate with *WinJS.Promise.timeout* (see scenario 7):

function getDesiredCount() {

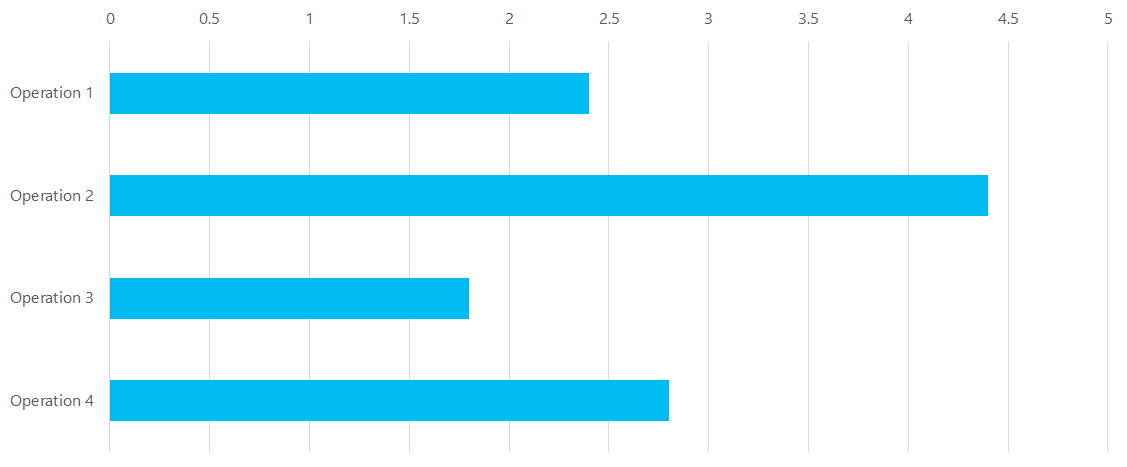
return WinJS.Promise.timeout(100).then(function () { return 1000; });

}

The output of scenario 7 now matches the (quiet) output of scenario 6, with a lot less code!

### Combining Parallel Promises: *WinJS.Promise.join* and *WinJS.Promise.any*

Apart from chaining sequential operations, there are occasions when you run multiple async operations in parallel, as illustrated below:



Taking these as a set, there are two basic things you might want to know:

1. When all the operations are complete, that is, all the promises are fulfilled. This would wait until Operation 2 completes in the diagram above.
2. When the first operation is complete, that is, the first promise is fulfilled. This would wait until Operation 3 completes in the diagram above.

WinJS provides a method for each purpose. First,

* [*WinJS.Promise.join*](http://msdn.microsoft.com/en-us/library/windows/apps/br229660.aspx)*:* given a list of values (an array or object, where the values can be promises or known values), creates a single promise that is fulfilled when all the others are fulfilled or fail with errors (a logical AND). In essence, *join* attaches completed and error handlers to all those promises, and waits until all those handlers are called before it calls whatever completed handlers it receives itself. Note that non-promise values in the list will be wrapped in a promise using WinJS.Promise.as.

In the illustration above, the promise from *join* will be fulfilled when Operation 2 (the longest) is complete.

The result of this *join* promise isn’t clearly documented but can be understood from the source code or simple tests from the consumer side. If the promises in the join all complete, the completed handler receives a list of results from the individual promises (even if those results are *null* or *undefined*). This list is an array if join was given an array, or an object if join was given an object.

If there’s an error in the join, the error object passed to the error handler is an array that contains the individual errors.

The joined promise also reports progress to any progress handlers you provide. The intermediate result in this case will be an array of results from those individual promises that have been fulfilled so far.

Note that if you cancel the join it cancels all the other promises that are still pending.

Second,

* [*WinJS.Promise.any*](http://msdn.microsoft.com/en-us/library/windows/apps/br229660.aspx): like *join*, takes a list of other values (promises or known values that will be wrapped in a promise) and creates a promise that is fulfilled when *one* of the others is fulfilled or fails with an error (a logical OR). In essence, *any* attaches completed handlers to all those promises. As soon as one completed handler is called, it calls whatever completed handlers the *any* promise has itself received.

In the previous illustration, the promise from *any* will fulfill when Operation 3 (the shortest) completed.

The result for this *any* promise is a little odd. It’s an object whose *key* property identifies the promise that was fulfilled and whose *value* property is an object containing that promise’s state. Within that state is a *\_value* property that contains the actual result of that promise.

After the *any* promise is fulfilled (that is, after the first promise in the array is fulfilled), the other operations for promises in the list continue to run, calling any completed, error, or progress handlers are assigned to those promises individually.

If you cancel the promise from *any*, then all the promises in the array are canceled.

### *WinJS.xhr*

In Example #3 a long time ago (and scenarios 3 and 4 of the Promises example), we worked up a promise wrapper around the XMLHttpRequest API. We didn’t need to bother—[*WinJS.xhr*](http://msdn.microsoft.com/en-us/library/windows/apps/br229787.aspx) does that for us already, with support for many options including HTTP request type (not just GET), credentials, headers, response type, custom data, and a custom request initializer callback. So here’s the equivalent (and condensed) consumer code to match that of scenario 4:

var promise = WinJS.xhr("http://kraigbrockschmidt.com/blog/?feed=rss2");

promise.then(function (results) {

console.log("complete, response length = " + results.response.length);

},

function (err) {

console.log("error in request: " + JSON.stringify(err));

},

function (partialResult) {

console.log("progress, response length = " + partialResult.response.length);

});

If you need to retrieve information from multiple URIs in parallel, here’s a little snippet that gets a WinJS.xhr promise for each and joins them together:

// uris is an array of URI strings

Promise.join(

uris.map(function (uri) { return WinJS.xhr(uri); })

).then(function (results) {

results.forEach(function (result, i) {

console.log("uri: " + uris[i] + ", " + result);

});

});

## Two WinJS.Promise Utility Functions

Beyond all the functions to create promises, *WinJS.Promise* offers a couple of simple static utility methods:

* [*WinJS.Promise.is*](http://msdn.microsoft.com/en-us/library/windows/apps/br211765.aspx) : determines whether an arbitrary value is a promise, returning a Boolean. It performs the same check as the *isPromise* function shown in a previous section, simply making sure the value is an object with a function names *then*.
* [*WinJS.Promise.thenEach*](http://msdn.microsoft.com/en-us/library/windows/apps/br229727.aspx): given an array of promises along with completed, error, and progress handlers (any of which can be *null*), calls the **then** method of each promise in that array with those handlers.

With *thenEach,* the return value is the result of *WinJS.Promise.join* on all the promises in the array.

## Error Handling with WinJS Promises

The final consideration for WinJS promises is error handling, a subject we haven’t spoken of much but one that becomes very important in real practice.

Clearly, you can attach an error handler to any promise, at any point in any chain. If there are exceptions or other errors within the execution of the underlying operation, that promise will call your error handler.

When implementing an async function, there are two error different error conditions you must handle. One is obvious: you encounter something within the operation that causes it to fail, such as a network timeout, a buffer overrun, the inability to access a resource, and so on. In these cases you call the error dispatcher (the second argument given to your initialization function by the *WinJS.Promise* constructor), passing an error object that describes the problem. That error object is typically created with [*WinJS.ErrorFromName*](http://msdn.microsoft.com/en-us/library/windows/apps/br211689.aspx) (using *new*), using an error name and a message, but this is not a strict requirement. *WinJS.xhr*, for example, passes the request object directly to the error handler as that object contains much richer information already.

To contrive an example, if *calculateIntegerSum* (from scenario 7) encountered some error within its processing, it would do the following:

if (false /\* replace with error check \*/) {

errorDispatch(new WinJS.ErrorFromName("calculateIntegerSum (scenario 7)", "error occurred"));

}

The other error condition is more interesting. What happens when a function that normally returns a promise encounters a problem such that it cannot create its usual promise? It can’t just return *null*, because that would make it very difficult to chain promises together. What it needs to do instead is return a promise that already contains an error, meaning that it will immediately call any error handlers give to its **then** with that error.

For this purpose, WinJS has a special function *WinJS.Promise.wrapError* whose argument is an error object (typically a *WinJS.ErrorFromName*). *wrapError* creates a promise that has no fulfillment value and will never call a completed handler. It will only pass its error to any error handler if you call **then**. Still, this **then** function must yet return a promise itself; in this case it returns a promise whose fulfillment value is the error object.

For example, if *calculateIntegerSum* receives *max* or *step* arguments that are less than 1, it has to fail and can just return a promise from *wrapError* (see scenario 7):

if (max < 1 || step < 1) {

var err = new WinJS.ErrorFromName("calculateIntegerSum (scenario 7)"

, "max and step must be 1 or greater");

return WinJS.Promise.wrapError(err);

}

Consumer code like this (also found in scenario 7):

calculateIntegerSum(0, 1).then(function (sum) {

console.log("calculateIntegerSum(0, 1) fulfilled with " + sum);

}, function (err) {

console.log("calculateIntegerSum(0, 1) failed with error: '" + err.message +"'");

return "value returned from error handler";

}).then(function (value) {

console.log("calculateIntegerSum(0, 1).then fulfilled with: '" + value + "'");

});

Will produce this output:

calculateIntegerSum(0, 1) failed with error: 'max and step must be 1 or greater'

calculateIntegerSum(0, 1).then fulfilled with: 'value returned from error handler'

Another way that an asynchronous operation can fail is by throwing an exception rather than calling the error dispatcher directly. This is important with async WinRT APIs, as those exceptions can occur deep down in the operating system. WinJS accommodates this by just wrapping the exception itself into a promise that can then be involved in chaining. The exception just shows up in the consumer’s error handler.

Speaking of chaining, what’s the best way for a consumer to handle errors within the chain? It’s certainly possible to have error handler at every step along the way, but this clearly becomes messy, and in many cases redundant—it’s more typical that you’ll want to handle all such errors more or less the same way.

For this reason, WinJS makes sure that errors are propagated through the chain to the error handler given to the last **then** in the chain, allowing you to consolidate your handling there. This is why promises from *wrapError* are themselves fulfilled with the error value, which they send to their completed handlers instead of the error handlers.

However, due to some subtleties in the JavaScript projection layer for the WinRT APIs, exceptions thrown from async operations within a promise chain will get swallowed and will not surface in that last error handler. Mind you, this doesn’t happen with a single promise and a single call to then, nor with nested promises, but most of the time the consumer is chaining multiple operations. So how can the error come to the surface?

Enter a sibling function to **then** called **done**. This method works pretty much like **then**, accepting all three kinds of handlers, but has a number of important characteristics:

1. **done** returns *undefined*, so it’s clearly used only at the end of a chain. By allowing **done** to return nothing, it doesn’t need to concern itself with propagating errors.
2. As a result, **done** ensures that any error in the chain is propagated to its error handler.
3. If there is no error handler given to **done**, the error is thrown as an exception out to the app level it will be handled like any exception that occurs outside of promises.

Exceptions from #3 here can be handled with a *window.onerror* handler, if desired. If left unhandled, Windows terminates the app using *MSApp.terminateApp*.

What this means in practice is that you typically put **done** at the end of a chain of promises with a single error handler, rather than having error handlers all along the way (unless you want a local one). In fact, a common pattern is to have the last part of the chain be just a call to **done** with a null completed handler, for example:

operation1().then(function (result1) {

return operation2(result1)

}).then(function (result2) {

return operation3(result2);

}).then(function (result3) {

//No more operations

}).done(null, function(err) {

//Handle errors here

}

So if you do have a *window.onerror* handler, then in every promise chain—that is, within every async sequence--you can decide whether to handle an exception locally (for a specific operation), at the end of the chain (with an error handler given to **done**), or at the app level (in *window.onerror*).

## WinJS.Promise events

The final bit we need to mention for WinJS promises is the fact that promises (that is, instances of WinJS.Promise) support an *error* event, which can be assigned through a promise’s *onError* property or through the standard *addEventListener*/*removeEventListener* mechanism. Such an event listener can act in place of an error handler given to **then** or **done** on the promise itself.

For this reason, a promise also has a *dispatchEvent* method, like many other JavaScript objects, meaning that you can, if you want, dispatch a custom event and have it picked up by handlers you’ve passed to the promise’s *addEventListener* method. This is not something most developers would use; for the most part, event listeners on promises is there for WinJS itself.

# Some Interesting Promise Code

So finally, now that we’ve thoroughly explored promises both in and out of WinJS, we’re ready to dissect various pieces of code involving promises and understand exactly what they do, beyond the basics of chaining as we’ve seen.

## Delivering a value in the future: WinJS.Promise.timeout

To start with a bit of review, the simple *WinJS.Promise.timeout(<n>).then(function () { <value> });* pattern again delivers a known value at some time in the future:

var p = WinJS.Promise.timeout(1000).then(function () { return 12345; });

Of course, you can return another promise inside the first completed handler and chain more **then** calls, which is just an example of standard chaining.

## Internals of WinJS.Promise.timeout(*n*, *promise*)

As before, the *WinJS.Promise.timeout(n, promise)* form fulfills *promise* if it happens within *n* milliseconds, otherwise *promise* is canceled. Here’s the core of its implementation:

function timeoutWithPromise(timeout, promise) {

var cancelPromise = function () { promise.cancel(); }

var cancelTimeout = function () { timeout.cancel(); }

timeout.then(cancelPromise);

promise.then(cancelTimeout, cancelTimeout);

return promise;

}

Here, the *timeout* argument comes from calling *WinJS.Promise.timeout(n)*, and *promise* is the same as in the original call to *WinJS.Promise.timeout(n, promise)*. As you can see, *promise* is just returned directly However, see how the promise and the timeout are wired together. If the *timeout* promise is fulfilled first, it calls *cancelPromise* to cancel *promise.* On the flipside, if *promise* is fulfilled first or encounters an error, it calls *cancelTimeout* to cancel the timer.

## Parallel Promises with Sequential Results

We’ve already seen the use of WinJS.Promise.join and WinJS.Promise.any to work with parallel promises, that is, with parallel async operations. The promise returned by join, again, will be fulfilled when all the promises in an array are fulfilled. However, those promises can each complete in any given order. What if you have a set of operations that can execute this way, but you want to process their results in a well-defined order, namely the order that their promises appear in an array?

The trick to do this is to basically join each subsequent promise to all of those that come before it, and the following bit of code does exactly that. Here, list is an array of values of some sort that are used as arguments for some promise-producing async call which I call doOperation:

list.reduce(function callback (prev, item, i) {

var result = doOperation(item);

return WinJS.Promise.join({ prev: prev, result: result}).then(function (v) {

console.log(i + ", item: " + item+ ", " + v.result);

});

})

To understand this code we have to first understand how the array’s [reduce](http://msdn.microsoft.com/en-us/library/ie/ff679975(v=vs.94).aspx) method works, because it’s slightly tricky. For each item in the array, reduce calls the function you provide as its argument, which I’ve named callback for clarity. This callback receives four arguments (only three of which are used in the code):

* prev The value that’s returned from the *previous* call to callback. For the first item, prev is null.
* item The current value from the array.
* i The index of item in the list.
* source The original array.

It’s also important to remember that WinJS.Promise.join can accept a list in the form of an object, as shown here, as well as an array (it uses Object.keys(list).forEach to iterate).

To make this code clearer, it helps to write it out with explicit promises:

list.reduce(function callback (prev, item, i) {

var opPromise = doOperation(item);

var join = WinJS.Promise.join({ prev: prev, result: opPromise});

return join.then(function completed (v) {

console.log(i + ", item: " + item+ ", " + v.result);

});

})

By tracing through this code for a few items in list, we’ll see how we build the sequential dependencies.

For the first item in the list, we get its opPromise and then join it with whatever is contained in prev. For this first item prev is null, so we’re essentially joining, to express it in terms of an array, [WinJS.Promise.as(null), opPromise]. But notice that we’re not returning join itself. Instead, we’re attaching a completed handler (which I’ve called completed) to that join and returning the promise from its then.

Remember that the promise returned from then will be fulfilled when the completed handler returns. This means that what we’re returning from callback is a promise that’s not completed until the first item’s completed handler has processed the results from opPromise. And if you look back at the result of a join, its fulfilled with an object that contains the results from the promises in the original list. That means that the fulfillment value v will contain both a prev property and a result property, the values of which will be the values from prev (which is null) and opPromise. Therefore v.result is the result of opPromise.

Now see what happens for the next item in list. When callback is invoked this time, prev will now contain the promise from the previous join.then. So in the second pass through callback we create a new join of opPromise2 and opPromise1.then. As a result, this join will not complete until both opPromise2 is fullfilled *and* the completed handler for opPromise1 returns. So voila! The completed2 handler we now attach to this join will not be called until after completed1 has returned.

In short, the same dependencies continue to be built up for each item in list—the promise from join.then for item *n* will not be fulfilled until completed*n* returns, and we’ve guaranteed that the completed handlers will be called in the same sequence as list.

## PageControlNavigator.\_navigated (page control rendering)

The next piece of code comes from the navigator.js file that’s included with the Visual Studio templates that employ WinJS page controls. This is an event handler for the WinJS.Navigation.onnavigated event, and performs the actual loading of the target page (using WinJS.UI.Pages.render to load it into a newly created div, which is then appended to the DOM) and unloading of the current page (by removing it from the DOM):

\_navigated: function (args) {  
 var that = this;  
 var newElement = that.\_createPageElement();  
 var parentedComplete;  
 var parented = new WinJS.Promise(function (c) { parentedComplete = c; });  
  
 args.detail.setPromise(  
 WinJS.Promise.timeout().then(function () {  
 if (that.pageElement.winControl && that.pageElement.winControl.unload) {  
 that.pageElement.winControl.unload();  
 }  
 return WinJS.UI.Pages.render(args.detail.location, newElement,   
 args.detail.state, parented);  
 }).then(function parentElement(control) {  
 that.element.appendChild(newElement);  
 that.element.removeChild(that.pageElement);  
 that.navigated();  
 parentedComplete();  
 })  
 );  
},

First of all, the args.detail.setPromise method is the WinJS deferral mechanism that’s used in a number of places. It tells WinJS.Navigation.onnavigated to defer its default process until the given promise is fulfilled. In this case, WinJS doesn’t actually have any other processing, but it’s reserving the possibility that it might in the future.

Anyway, the promise in question here is what’s produced by the remaining code, which is basically a WinJS.Promise.timeout().then().then() sequence.

The initial use of timeout means that the process of rendering a page control first yields the UI thread via setImmediate, allowing other work to complete before we start the rendering process.

After such yielding, we then enter into the first completed handler. If we already have a page control loaded (that.pageElement.winControl), we call it’s unload method, if available. After this we start rendering the new page control into newElementwith WinJS.UI.Pages.render. Rendering is an async operation itself (it involves a file loading operation, for one thing), so render returns a promise. Note that at this point, newElement is an orphan—it’s not yet part of the DOM, just an object in memory, so all this rendering is just a matter of loading up the page control’s contents and building that standalone chunk of DOM.

When render completes, the next completed handler in the chain, which is actually named parentElement (“parent” in this case being a verb), receives the newly-loaded page control object. This code doesn’t make use of this argument, however, because it knows that it’s the contents of newElement (newElement.winControl, to be precise). So we now attach that new page’s contents to the DOM and remove the previous page’s contents, meaning that the new page contents will appears in place of the old the next time the rendering engine gets a chance to do its thing.

Finally we fire off a navigated event, and then call this function parentedComplete. This last bit is really a wiring job so that WinJS will not invoke the new page’s ready method until it’s been actually added to the DOM. This means that we need a way for WinJS to hold off making that call until we here tell it that parenting has finished.

Earlier in \_navigated, we created a parentedPromise variable, which was then given as the fourth parameter to WinJS.UI.Pages.render. This parentedPromise is very simple: we’re just calling new WinJS.Promise and doing nothing more than saving its completed dispatcher in the parentedComplete variable, which is what we call at the end of the process.

For this to serve any purpose, of course, someone needs to call parentedPromise.then and attach a completed handler. A WinJS page control does this, and all its completed handler does is call ready. Here’s not it looks in base.js:

this.renderComplete.then(function () {

return parentedPromise;

}).then(function Pages\_ready() {

that.ready(element, options);

})

In the end, this whole \_navigated code is just saying, “After yielding the UI thread, asynchronously load up the new page’s HTML, add it to the DOM, remove the old page from the DOM, then tell WinJS that it can call the new page’s ready method, because we’re not calling it directly ourselves.”

## ListView Item Rendering, Part 1: The Basic Renderer Structure

The thorough understanding of promises lets us see exactly what’s going on in the different stages of optimization that are demonstrated in scenario 1 of the [HTML ListView optimizing performance sample](http://code.msdn.microsoft.com/windowsapps/ListView-performance-39fb71f0). We’ll start with the basic structure, then move into the different stages of optimization to finally come back to the code with which we started this appendix!

The first stage is just using a rendering function to create a ListView item (its elements) instead of a declarative template (in an html file). This allows the function to create different elements on the fly depending on the item’s contents. The ListView, for its part, will call this function for each item in the data source.

So you might think that such a rendering function would look something like this:

function NOTaRealRenderer(item) {  
 var element = document.createElement("div");  
 element.className = "itemTempl";  
 element.innerHTML = "<img src='" + item.data.thumbnail +  
 "' alt='Databound image' /><div class='content'>" + item.data.title + "</div>";  
 return element;  
}

However, there are some additional considerations. First, the item data itself might be loaded asynchronously, so it makes sense for the element creation to be tied to the availability of that data. The ListView could, of course, attach its own completed handler to every promise involved, where that handler would then call the rendering function. Alternately, the ListView could just give the promise directly to the rendering function and have it attach a completed handler of its own, where that handler produces the item rendering.

So an item renderer could look like this:

function stillNotARealRenderer(itemPromise) {  
 itemPromise.then(function (item) {  
 var element = document.createElement("div");  
 element.className = "itemTempl";  
 element.innerHTML = "<img src='" + item.data.thumbnail +  
 "' alt='Databound image' /><div class='content'>" + item.data.title + "</div>";  
 return element;  
 });  
}

This would work just fine if item rendering always happened synchronously, but what if you need to do some additional async work, like load images or read additional data from a config file? This is exactly what we want to accommodate with the different stages of optimization, allowing an arbitrary amount of async work between the request for the item and its fulfillment. So instead of having the rendering function return the item’s element, such a function must return a promise for that element. This is a simple matter with the code above: because the return value of itemPromise.then is a promise that’s fulfilled with the result of the completed handler (element), we can just return the return value from then. So the following is now a *real* item rendering function, as found in the sample:

function simpleRenderer(itemPromise) {  
 return itemPromise.then(function (item) {  
 var element = document.createElement("div");  
 element.className = "itemTempl";  
 element.innerHTML = "<img src='" + item.data.thumbnail +  
 "' alt='Databound image' /><div class='content'>" + item.data.title + "</div>";  
 return element;  
 });  
}

With this, the ListView can also join such item rendering promises together if it needs to wait until a whole page of items has been built up. It does this, in fact, to intelligently manage how it builds up different pages in the list. It first builds the page of items that are currently visible, then builds up two pages forward and back (where most users will pan to next), and afterwards can build up more distant pages. In addition, having all these promises in place means that the ListView can effectively cancel the rendering of unfinished items if the user pans away.

And again, having the rendering function return a promise means that it can also do other async work involving other promises, and this is what other stages of optimization rely upon.

## ListView Item Rendering, Part 2: Placeholders and Multistage Rendering

The next stage of ListView optimization involves what’s called a *placeholder renderer*, which separates building up the element into two stages. Here, the renderer returns an object that contains two properties:

* element  The top-level element in the item’s structure that’s enough to define its size and shape and is not dependent on the item data. This allows the ListView to quickly determine how those items are laid out in the list without having to build up all the child elements.
* renderComplete A promise that’s fulfilled when the remainder of the element’s contents are constructed, that is, returns the promise from itemPromise.then(>completedHandler>) where <completedHandler> builds whatever parts of the item weren’t built for element. The completed handler in this case, however, need not return a promise if no async work is involved.

All we’re doing here is structuring the item construction process to allow the ListView to ask for the outer placeholder separately from the full item. The ListView is smart enough to check whether your renderer returns a promise (the simple case) or an object with element and renderComplete properties (more advanced cases). Thus the equivalent placeholder renderer for the one we saw earlier (also in the sample, js/scenario1.js) is as follows:

function placeholderRenderer(itemPromise) {  
 // create a basic template for the item which doesn't depend on the data  
 var element = document.createElement("div");  
 element.className = "itemTempl";  
 element.innerHTML = "<div class='content'>...</div>";  
  
 // return the element as the placeholder, and a callback to update it when data is available  
 return {  
 element: element,  
  
 // specifies a promise that will be completed when rendering is complete  
 // itemPromise will complete when the data is available  
 renderComplete: itemPromise.then(function (item) {  
 // mutate the element to include the data  
 element.querySelector(".content").innerText = item.data.title;  
 element.insertAdjacentHTML("afterBegin", "<img src='" +  
 item.data.thumbnail + "' alt='Databound image' />");  
 })  
 };  
}

Note that in the sample, the element.innerHTML assignment can even be moved inside renderComplete because the *itemTempl* class in css/scenario1.css specifies the width and height of the item directly. The reason why it’s included in the element property is because it provides the default “…” text in the placeholder. You could just as easily use an img element that refers to a small in-package resource that’s shared across all the items (and thus renders quickly).

(For another example, the added Scenario 8 of the FlipView example in the companion content for Chapter 6 has commented code that implements this approach.)

The next stage now, the *recycling placeholder* renderer, doesn’t add anything new where promises are concerned. It simply adds awareness of a second parameter called recycled that the ListView (but not the FlipView) can provide to your rendering function when the ListView’s loadingBehavior is set to "randomaccess". If recycled is given, you can just clean out the element, return it as the placeholder, and then fill in the data values within the renderComplete promise as before. If it’s not provided (as when the ListView is first created or when loadingBehavior is "incremental"), you’ll create the element anew. Here the code from the samples (js/scenario1.js):

function recyclingPlaceholderRenderer(itemPromise, recycled) {  
 var element, img, label;  
 if (!recycled) {  
 // create a basic template for the item which doesn't depend on the data  
 element = document.createElement("div");  
 element.className = "itemTempl";  
 element.innerHTML = "<img alt='Databound image' style='visibility:hidden;'/>" +  
 "<div class='content'>...</div>";  
 }  
 else {  
 // clean up the recycled element so that we can re-use it   
 element = recycled;  
 label = element.querySelector(".content");  
 label.innerHTML = "...";  
 img = element.querySelector("img");  
 img.style.visibility = "hidden";  
 }  
 return {  
 element: element,  
 renderComplete: itemPromise.then(function (item) {  
 // mutate the element to include the data  
 if (!label) {  
 label = element.querySelector(".content");  
 img = element.querySelector("img");  
 }  
 label.innerText = item.data.title;  
 img.src = item.data.thumbnail;  
 img.style.visibility = "visible";  
 })  
 };  
}

In renderComplete, be sure to check for the existence of elements that you don’t create for a new placeholder, such as label, and create them here if needed.

If you’d like to clean out recycled items, you can also provide a function to the ListView’s resetItem property that would contain the same code as shown above for that case. The same is true for the resetGroupHeader property, because you can use template functions for group headers as well as items. We haven’t spoken of these as much because group headers are far fewer and don’t typically have the same performance implications. Nevertheless, the capability is there.

This brings us to the penultimate optimization, the *multistage renderer*. This extends the recycling placeholder renderer to delay-load images and other media until the rest of the item is wholly present in the DOM. It also delays effects like animations until the item is actually on-screen. All of this involves more async operations, so you can expect there are more promises involved.

The hooks for this are provided as members on the item result from itemPromise: a property called ready (a promise) and two methods and isOnScreen , both of which return promises. That is, you’ll find these members on the item that’s passed to your first completed handler:

renderComplete: itemPromise.then(function (item) {  
 // item has properties names ready, loadImage, and isOnScreen  
 })

Here’s how you use them:

* ready Return this promise from the first completed handler in your chain. This promise is fulfilled when the full structure of the element has been rendered and is visible. This means you can chain another then with a completed handler in which you do other post-visibility work like loading images.
* loadImage Downloads an image from a URI and displays it in the given img element, returning a promise that’s fulfilled with . You attach a completed handler to this promise, which itself returns the promise from isOnScreen.
* isOnScreen Returns a promise whose fulfillment value is a Boolean indicating whether the item is visible or not. In present implementations, this is a known value so the promise is fulfilled synchronously. By wrapping it in a promise, though, it can be used in an overall chain.

We see all this in the sample’s multistageRenderer function (js/scenario1.js, other parts omitted), where completion of the image load is used to start a fade-in animation:

renderComplete: itemPromise.then(function (item) {  
 // mutate the element to update only the title  
 if (!label) { label = element.querySelector(".content"); }  
 label.innerText = item.data.title;  
  
 // use the item.ready promise to delay the more expensive work  
 return item.ready;  
 // use the ability to chain promises, to enable work to be cancelled  
 }).then(function (item) {  
 //use the image loader to queue the loading of the image  
 if (!img) { img = element.querySelector("img"); }  
 return item.loadImage(item.data.thumbnail, img).then(function () {  
 //once loaded check if the item is visible  
 return item.isOnScreen();  
 });  
 }).then(function (onscreen) {  
 if (!onscreen) {  
 //if the item is not visible, then don't animate its opacity  
 img.style.opacity = 1;  
 } else {  
 //if the item is visible then animate the opacity of the image  
 WinJS.UI.Animation.fadeIn(img);  
 }  
 })

Even though there’s a lot going on, we still just have a basic promise chain here. The first async operation in the renderer updates simple parts of the item element, such as text. It then returns the promise in item.ready. When that promise is fulfilled—or, more accurately, if that promise is fulfilled—you can use the item’s async loadImage method to kick off an image download, returning the item.isOnScreen promise from that completed handler, such that the onscreen visibility flag gets to the final completed handler in the chain. When and if that isOnScreen promise is fulfilled, you can perform those operations that are relevant only to a visible item.

I emphasize the if part of all this because it’s very likely that the user will be panning around within the ListView while all this is happening. Having all these promises chained together again makes it possible for the ListView to cancel the async operations any time these items are scrolled out of view and/or off any buffered pages. Suffice it to say that the ListView control has gone through a *lot* of performance testing!

It’s also important to remind ourselves that we’re using then throughout all these chains because we’re still returning a promise from the rendering function within the renderComplete property. We’re never the end of the chain in these renderers, so we’ll never use done at the end.

## ListView Item Rendering, Part 3: Thumbnail Batching

And finally, the *coup de grace*, the createBatch function for thumbnails in a ListView control, with which we started this conversation. To understand this, we first need to understand what we’re trying to accomplish.

If we just had a ListView with a single item, various loading optimizations wouldn’t be noticeable. But ListViews typically have many items, and the rendering function is called for each one. In the multistageRenderer of the previous section, the rendering of each item kicks off an async item.loadImage operation to download its thumbnail from an arbitrary URI, and each operation can take any amount of time. So for the list as a whole, we have a bunch of simultaneous loadImage calls going on, with the rendering of each item waiting on the completion of its particular thumbnail. This is not a problem in and of itself.

An important characteristic that’s not visible in multistageRenderer, however, is that the img element for the thumbnail is *already* in the DOM, and the loadImage function will set that image’s src attribute as soon as the download has finished. This triggers an update in the rendering engine as soon as we return from the rest of the promise chain, which is essentially synchronous after this point.

It’s possible, then, to see a bunch of thumbnails coming back within a short amount of time. This will cause excess churn in the rendering engine that results in poor visual performance. To improve this, we want to add those img elements to the DOM in batches, thereby combining them together in a single rendering pass. The code in createBatch achieves this with some promise magic, structured such that it’s easily transportable to other projects.

Let’s look first at how createBatch is used, then we’ll see how it works. createBatch is called just once for the whole app, and its result—another function—is stored in a variable named thumbnailBatch:

var thumbnailBatch;  
thumbnailBatch = createBatch();

A call to this thumbnailBatch function, as I’ll refer to it from here on, is then inserted into the promise chain of the updated multistage renderer (batchRenderer in the sample) between the item.loadImage call and the item.isOnScreen check—notice the added line with then(thumbnailBatch()):

renderComplete: itemPromise.then(function (i) {  
 item = i;  
 // ...  
 return item.ready;  
 }).then(function () {  
 return item.loadImage(item.data.thumbnail);  
 }).then(thumbnailBatch()  
 ).then(function (newimg) {  
 img = newimg;  
 element.insertBefore(img, element.firstElementChild);  
 return item.isOnScreen();  
 }).then(function (onscreen) {  
 //...  
 })

This simply insertion, given the nature of the batching code, very clearly collects the loaded images into batches, releasing them for further processing at suitable intervals. Clearly, whatever thumbnailBatch() returns it itself a completed handler that must return another promise, and that promise delivers the new img element that can then be added to the DOM. By adding those images to the DOM after the batching has taken place, we combine that whole group into the same rendering pass.

There’s a subtle difference between this batchRenderer code and the previous multistageRenderer: in the latter, then thumbnail’s img element already exists in the DOM and is passed to loadImage as the second parameters. Thus when loadImage sets the image’s src attribute, a rendering update is triggered. Within batchRenderer, however, that img element hasn’t been created yet. This instead happens within loadImage, where src is also set, but the img is *not* yet in the DOM. It’s only added to the DOM after the thumbnailBatch step completes, which means it’s done as part of a group within that single layout pass.

So now let’s see how that batching works. For reference, here’s that function again in its entirety:

function createBatch(waitPeriod) {

var batchTimeout = WinJS.Promise.as();

var batchedItems = [];

function completeBatch() {

var callbacks = batchedItems;

batchedItems = [];

for (var i = 0; i < callbacks.length; i++) {

callbacks[i]();

}

}

return function () {

batchTimeout.cancel();

batchTimeout = WinJS.Promise.timeout(waitPeriod || 64).then(completeBatch);

var delayedPromise = new WinJS.Promise(function (c) {

batchedItems.push(c);

});

return function (v) {

return delayedPromise.then(function () {

return v;

});

};

};

}

Again, createBatch is called just *once* and thumbnailBatch is called for *every item in the list*, returning a completed handler that itself returns a promise.

Such a completed handler might just as easily have been inserted directly into the rendering function, but what we’re trying to do here is coordinate activities *across multiple items* rather than just on a per-item basis. This coordination is achieved through the two variables created and initialized at the beginning of createBatch: batchedTimeout, initialized as an empty promise, and batchedItems, an array of functions that’s initially empty. createBatch also declares a function that simply empties batchedItems, calling each function in the array:

function completeBatch() {

//Copy and clear the array so that the next batch can start to accumulate  
 //while we're processing the previous one.  
 var callbacks = batchedItems;

batchedItems = [];

for (var i = 0; i < callbacks.length; i++) {

callbacks[i]();

}

}

Now let’s see what happens within thumbnailBatch, which is again called for each item being rendered. First, we *cancel* any existing batchedTimeout and immediately recreate it.

batchTimeout.cancel();

batchTimeout = WinJS.Promise.timeout(waitPeriod || 64).then(completeBatch);

The second line shows the future delivery/fulfillment pattern as discussed earlier: it says to call completeBatch after a delay of waitPeriod milliseconds (with a default of 64ms). This means that so long as thumbnailBatch is being called within waitPeriod of a previous call, batchTimeout will be reset to another waitPeriod. And because thumbnailBatch is only called *after* an item.loadImage call completes, we’re effectively saying that any loadImage operations that complete within waitPeriod of the previous one will be included in the same batch. When there’s a gap longer then waitPeriod the batch is processed (images are added to the DOM) and the next batch begins.

After handling this timeout business, thumbnailBatch creates a new promise that simply pushes the complete dispatcher function into batchedItems:

var delayedPromise = new WinJS.Promise(function (c) {

batchedItems.push(c);

});

Remember from early on that a promise is just a code construct, and that’s all we have here. The newly created promise has no async behavior in and of itself: we’re just adding the complete dispatcher function, c, to batchedItems. But of course, we don’t do anything with batchedItems until batchedTimeout completed (asynchronously), so there is an async relationship here: when the timeout happens and we clear the batch (inside completeBatch), we’ll invoke any completed handlers given elsewhere to delayedPromise.then.

This brings us to the last line of code in createBatch, which is the function that thumbnailBatch returns. This function is exactly the completed handler that gets inserted into the renderer’s whole promise chain:

return function (v) {

return delayedPromise.then(function () {

return v;

});

};

In fact, let’s put this piece of code directly into the promise chain so we can see the resulting relationships:

return item.loadImage(item.data.thumbnail);  
 }).then(function (v) {  
 return delayedPromise.then(function () {  
 return v;  
 });  
 ).then(function (newimg) {

Now we can see that the argument v is the result of item.loadImage, which is the img element it creates for us. If we didn’t want to do batching, we could just say return WinJS.Promise.as(v) and the whole chain would still work: v would then be passed on synchronously and show up as newimg in the next step.

Instead, though, we’re returning a promise from delayedPromise.then, which won’t be fulfilled—with v—until the current batchedTimeout is fulfilled. At that time—when again there’s a gap of waitPeriod between loadImage completions—those img elements are then delivered to the next step in the chain where they’re added to the DOM.

And that’s it!

# In Closing

We’ve now explored promises in great depths, and in the process we can really see how powerful the promise construct is to define complex relationships in very little code. It can, of course, be a challenge to understand exactly what’s happening, but having looked at all the relationships and patterns in detail—especially how the WinJS.Promise APIs work—we’ve been able to tease apart even complex promise code like handling parallel promises with sequential results and the createBatch function used for ListView optimization.

1. The method could be a *property* *setter*, of course; the point here is that a method of some kind is necessary for the object to have a trigger for additional action, something that a passive (non-setter) property lacks.

   In this context I’ll also share a trick that Chris Sells, who was my manager at Microsoft for a short time, used on me repeatedly. For some given deliverable I owed him, he’d ask, “When will you have it done?” If I said I didn’t know, he’d ask, “When will you know when you’ll have it done?” If I still couldn’t answer that, he’d ask, “When will you know when you’ll know when you’ll have it done?” *ad infinitum* until he extracted some kind of solid commitment from me! [↑](#footnote-ref-1)
2. The specification for **then**, however, stipulates that its return value is a promise that’s fulfilled when the completed handler returns, which is one of the bits that makes the implementation of a promise quite complicated. Clearly, we’re ignoring that part of the definition here! [↑](#footnote-ref-2)