# All About Promises

When writing Windows Store apps in JavaScript, you encounter these constructs called *promises* as soon as you do anything that involves an asynchronous API. It also doesn’t take very long before writing promise chains for sequential asynchronous operations becomes second nature.

In the course of your development work, however, you’ll probably encounter other uses of promises where it’s not entirely clear to see what’s going on. A good example of this is optimizing item rendering functions for a ListView control as demonstrated in the [HTML ListView optimizing performance sample](http://code.msdn.microsoft.com/windowsapps/ListView-performance-39fb71f0), the details of which we’ll explore in a subsequent post. Or consider this little gem that Josh Williams showed in his [Deep Dive into WinJS](http://channel9.msdn.com/Events/Build/2012/4-101) talk at //build 2012 (slightly modified):

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

var result = doOperationAsync(item);

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

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

});

})

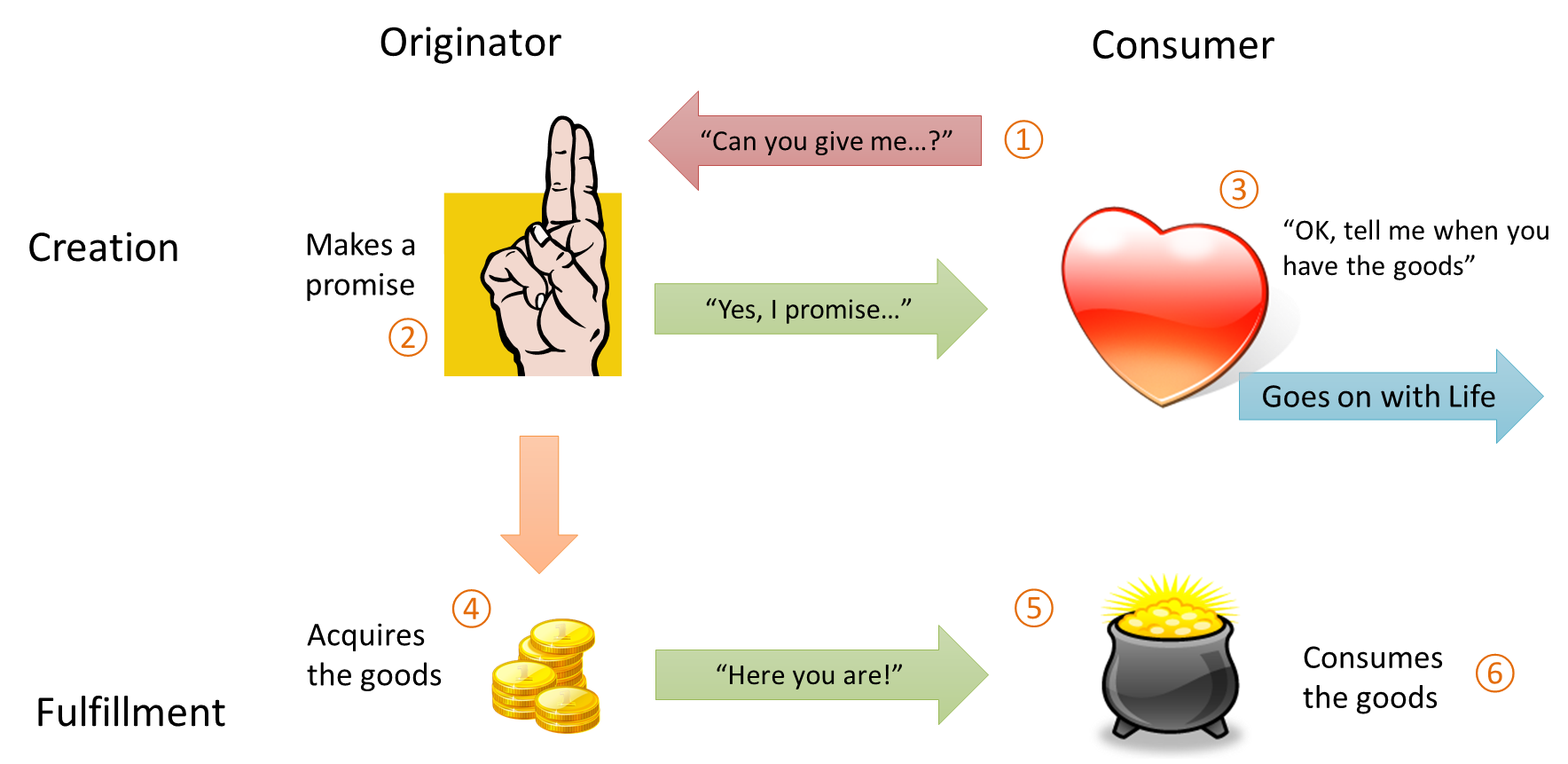
This snippet joins promises for parallel async operations and delivers their results sequentially according to the order in *list*. If you can look at this code and immediately understand what it does, then feel free to skip this post altogether! If not, let’s take a closer look at how promises really work, and their expression in WinJS—the Windows Library for JavaScript—so we can understand these kinds of patterns.

## What is a Promise, Exactly? The Promise Relationships

Let’s begin with a fundamental truth: a promise is really nothing more than a code construct. As such, promises have no inherent relationship to async operations—they just so happen to be very *useful* in that regard! A promise is simply an object that represents a value that might be available at some point in the future, or may be available already. In this way, a promise just like how we use the term in human relationships. If I say, “I promise to deliver you a dozen donuts,” then clearly I don’t need to have those donuts in my possession right now, but I certainly assume that I *will* have them some time in the future. And when I do, I’ll deliver them.

A promise thus implies a relationship between two agents: the *originator* who makes the promise to deliver some goods, and the *consumer* who is both the recipient of that promise and the goods themselves. How the originator goes about obtaining those goods is its own business. Similarly, the consumer can do whatever it wants with the promise itself and the delivered goods. It can even share the promise with other consumers.

Between originator and consumer, there are also two stages of this relationship, *creation* and *fulfillment*. All this is illustrated in the following figure.



Having two stages of the relationship is why promises work well with asynchronous delivery, as we can see when we follow the flow in the diagram:

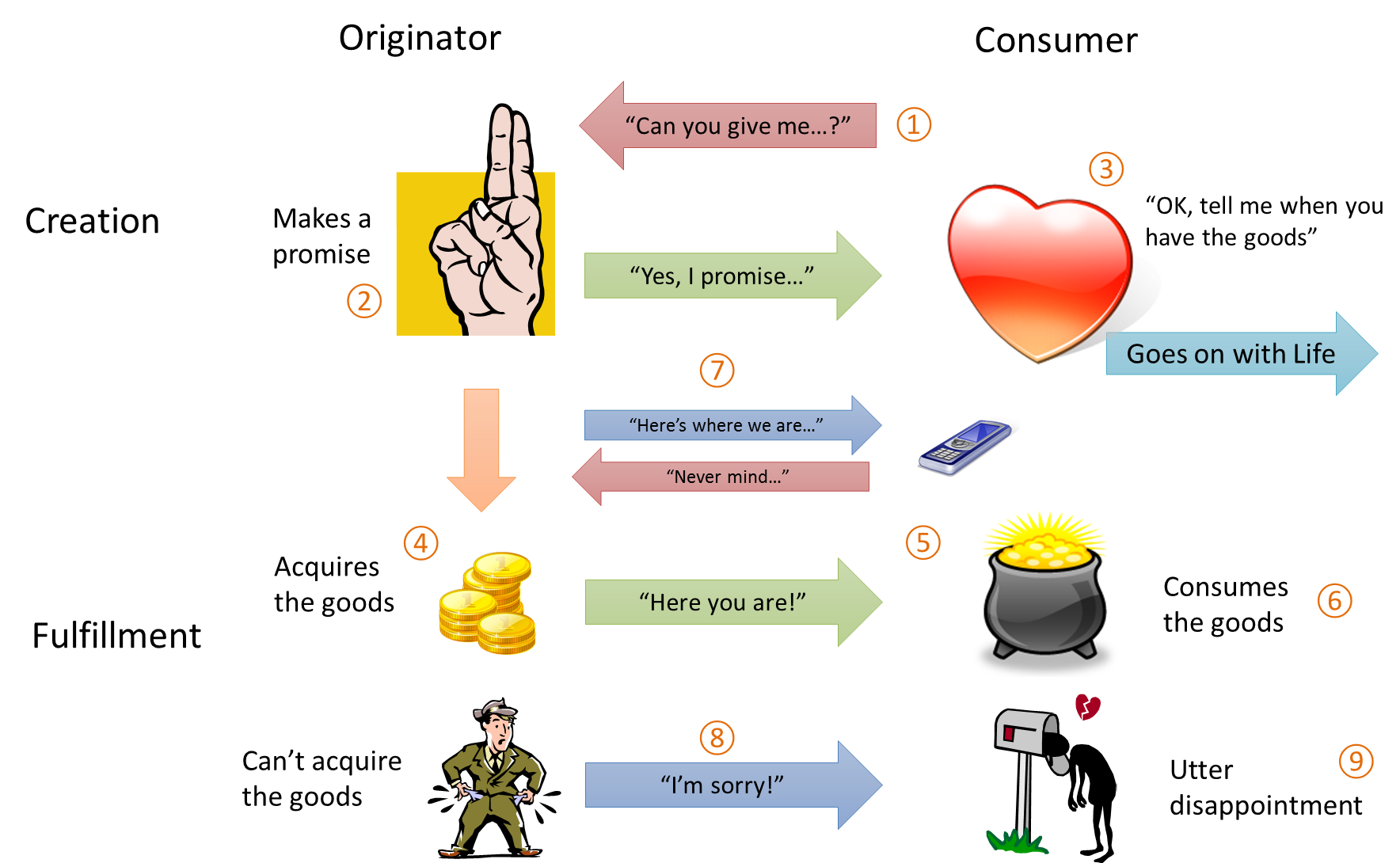
1. The relationship begins when the consumer asks an originator for something, as when an app calls some API that provides a promise rather than an immediate value.
2. The originator creates a promise for the goods in question and returns that promise to the consumer.
3. The consumer acknowledges receipt of the promise by saying in some way, “tell me when the goods are ready.” After this the consumer can go on with its life (asynchronously) instead of waiting (synchronously).
4. Meanwhile, the originator works to acquire the promised goods. Again, they might already be available, or they might take some indeterminate (asynchronous) amount of time to obtain.
5. Once the originator has the goods, it delivers them to the consumer, *fulfilling* the promise.
6. The consumer consumes the goods as desired.

If you’re clever enough, you might have noticed that by eliminating the “Yes, I promise…” arrow and the consumer’s part around (3), we’re left with a simple synchronous function call. So it’s the fact that a consumer is given a chance to do something else with its time, like being responsive to other requests. And that is exactly the whole purpose of 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.

Of course, there’s a bit more to this relationship that we have to consider. 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’s 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 in both our personal lives and in asynchronous programming.

Within the promise relationship, then, this means that originator needs a way to say, “I’m sorry, but I can’t make good on this promise” and the consumer needs a way to know that this is the case. Secondly, as consumers, we can sometimes be rather impatient about promises made to us! 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 cancel the order and tell the originator that it no longer needs the goods.

Adding these requirements to the diagram, we can now see the complete relationship:



Here we’ve added:

1. While the originator is attempting to acquire the goods, it can periodically let the consumer know what’s happening. 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.
3. If the promise is broken, the consumer has to deal with it as best it can!

Let’s now see how these relationships manifest in code.

## The Promise Construct and Promise Chains

There are actually a number of different proposals or specifications for promises. The one used in Windows and WinJS is known as [Common JS/Promises A](http://wiki.commonjs.org/wiki/Promises/A), which says a promise—what’s returned by an originator to represent a value to deliver in the future—is an object with a function called **then**. Consumers subscribe to fulfillment of the promise by calling **then**. (Promises in Windows also support a similar function called **done** that’s used in promise chains, as we’ll see shortly.)

To this function the consumer passes one, two, or three functions as arguments:

1. A *completed handler* (required). The originator will call this function when the promised value is available, and if that value already *is* available, the completed handler will be called immediately (synchronously) from within **then**.
2. An optional *error handler* that’s called when there’s a failure to acquire the promised value. For any given promise, the completed handler will never be called if the error handler has been called.
3. An optional *progress handler*, that’s periodically called with intermediate results if the operation supports it. (In WinRT, this means the API has a return value of **IAsync[Action | Operation]WithProgress**; those with **IAsync[Action | Operation]** do not.)

On the other side of the relationship, a consumer may subscribe as many handlers as it wants to the same promise by calling **then** multiple times. It can also share the promise with other consumers who may also call then **then** to their hearts’ content. This is entirely supported.

This means that a promise must manage lists of all the handlers it receives and invoke them at the appropriate times. Promises also need to allow for cancellation, as outlined in the full relationship.

The other requirement that comes from the Promises A specification is that the **then** method itself returns a promise. This second promise is fulfilled when the completed handler given to the first **promise.then** returns, with the return value being delivered as the result of this second promise. Consider this code snippet:

var promise1 = someOperationAsync();

var promise2 = promise1.then(function completedHandler1 (result1) { return 7103; } );

promise2.then(function completedHandler2 (result2) { });

The chain of execution here is that **someOperationAsync** gets started, returning *promise1*. While that operation is underway, we call **promise1.then** which immediately returns *promise2*. Be very clear that **completedHandler1** will not be called unless the result of the async operation is already available. Let’s assume we’re still waiting, so we go right into the call to **promise2.then**, and again, completedHandler2 will *not* be called at this time.

Sometime later, **someOperationAsync** completes with a value of, say, 14618. *promise1* is now fulfilled, so it calls **completedHandler1** with that value, so *result1* will be 14618. **completedHandler1** now executes, returning the value 7103. At this point *promise2* is fulfilled, so it calls **completedHandler2** with *result2* equal to 7103.

Now what if a completed handler returns another promise? This case is handled a little differently. Let’s say that **completedHandler1** in the code above returns a promise like so:

var promise2 = promise1.then(function completedHandler1 (result1) {

var promise2a = anotherOperationAsync();  
 return promise2a;

});

In this case, *result2* in **completedHandler2** won’t be *promise2a* itself, but the fulfillment value of *promise2a*. That is, because the completed handler returns a promise, *promise2* as returned from **promise1.then**, will be fulfilled with the results of *promise2a*.

This characteristic is precisely what makes it possible to chain together sequential async operations, where the results from each operation in the chain will feed into the next. Without intermediate variables or named handlers, you often see this pattern for promise chains:

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

});

Each completed handler, of course, will likely do something more with the results it receives, but in all chains you’ll see this core structure. What’s also true is that all the **then** methods here will execute one right after the other, as all they’re doing is saving away the given completed handler and returning another promise. So by the time we reach the end of this code, only *operation1* has started and no completed handlers have been called. But a bunch of intermediate promises from all the **then** calls have been created and wired up to one another to manage the chain as the sequential operations progress.

It’s worth noting that the same sequence can be achieved by nesting each subsequent operation within the previous completed handler, in which case you won’t have all the *return* statements. However, such nestings become an indentation nightmare, especially if you start adding progress and error handlers with each call to **then**.

Speaking of which, one of the features of promises in WinJS is that errors that occur in any part of the chain will automatically propagate to the end of the chain. This means that you should be able to simply attach a single error handler on the last call to **then** instead of having handlers at every level.

The caveat, however, is that for various subtle reasons this won’t actually work. If the last link in the chain is a call to **then**, errors within the chain just get swallowed and won’t surface in any error handler given to that **then**. This makes it difficult to debug promise chains, because exceptions within the chain will just disappear.

This is the purpose of the **done** method that accepts the same function arguments as **then** but does *not* return another promise (it returns *undefined*). By calling **done** you clearly indicate that the chain is complete, and if you attach an error handler to **done** then that handler will be called for any errors in the entire chain. If you don’t provide such an error handler, **done** will then throw an exception to the app level rather than making it disappear (you can catch those exceptions within the **window.onerror** of **WinJS.Application.onerror** events.) In short, all chains should ideally end in **done**.

Be mindful, though, that if you write a function whose *purpose* is to return a promise that is the result of a long chain of **then** calls, then you still want to return whatever promise comes from the last **then** in the chain. This is because you’re passing the responsibility for that promise to the caller, who might use that promise in another chain altogether.

## Creating Promises: the WinJS.Promise Class

Because promises are just a code construct at the end of the day, you might be tempted to just implement a promise class of your own. Let me tell you from direct experience, though, that this is not for the weak of heart! It’s the kind of thing that begs for a standard implementation in a library, and this is fortunately one that is delivered as part of WinJS. WinJS provides a way to easily create promises around different values and operations without having to manage the details of the originator/consumer relationships or the behavior of **then**. You can see the implementation in the base.js file of WinJS itself, and once you see the complexity involved you’ll appreciate not having to write the code yourself!

WinJS provides a robust, well-tested, and flexible promise class called [**WinJS.Promise**](http://msdn.microsoft.com/en-us/library/windows/apps/br211867.aspx). When needed, you can (and should) use **new WinJS.Promise** to create promises around both asynchronous operations and existing (synchronous) values alike. Remember that a promise is just a code construct: there’s no requirement that a promise has to wrap an async operation or async *anything*. Similarly, the mere act of wrapping some piece of code in a promise does not automatically *make* it run asynchronously*.* That’s still work you have to do yourself.

As a simple example, let’s say we want to perform a long computation—just adding up a bunch of numbers from one to some maximum—but do it asynchronously. We could invent our own callback mechanism for such a routine, but if we wrap it within a promise then we allow it to be chained or joined with other promises from other APIs. (Along these lines, the [**WinJS.xhr**](http://msdn.microsoft.com/en-us/library/windows/apps/br229787.aspx) function wraps the asynchronous **XmlHttpRequest** of JavaScript within a promise so you don’t have to deal with the latter’s particular event structure.)

We can of course use a JavaScript worker for a long computation, of course, but for the sake of illustration we’ll just keep this on the UI thread and use **setImmediate** to break the operation into steps. Here’s how we can implement it within a promise structure using **WinJS.Promise**:

function calculateIntegerSum(max, step) {  
 //The WinJS.Promise constructor's argument is a initializer 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) });  
 });  
}

When calling **new WinJS.Promise**, the single argument to its constructor is an *initializer* function (anonymous in this case). The initializer encapsulates the operation to perform, but be very clear that this function itself executes *synchronously* on the UI thread. So if we just performed a long calculation here without using **setImmediate** we would block the UI thread for all that time. Again, placing code inside a promise does automatically make it asynchronous—the initializer function needs to set that up itself.

For arguments, the initializer function receives three *dispatchers* for the completed, error, and progress cases that promises support. As you can see, we call these dispatchers at appropriate times during the operation with the appropriate arguments.

I call these functions “dispatchers” because they are not the same thing as the handlers that consumers subscribe to the promise’s **then** method (or **done**, but I won’t keep reminding you). Under the covers, WinJS is managing arrays of those handlers, which is what allows any numbers of consumers to subscribe any number of handlers. When you invoke one of these dispatchers, WinJS will iterate through its internal list and invoke all of those handlers on your behalf. **WinJS.Promise** also makes sure that its **then** returns another promise, as required for chaining.

In short, **WinJS.Promise** is providing all the surrounding details of a promise. This allows you to concentrate on the core operation that the promise represents, which is embodied in the initializer function.

## Cancellation and Generating Promise Errors

In the code we just saw, you might have noticed two deficiencies. First is that there is no way to cancel the operation once it gets started. The second is that we don’t do a very good job handling errors.

The trick in both these cases is that promise-producing functions like **calculateIntegerSum** function must always return a promise. If an operation can’t complete or never gets started in the first place, that promise is in what’s called the *error state*. This means that the promise doesn’t have and never *will* have a result that it can pass to any completed handlers: it will only ever call its error handlers.

A **WinJS.Promise** enters the error state for two reasons: the consumer calls its **cancel** method or the code within the initializer function calls the error dispatcher. When this happens, error handlers are given an instance of [**WinJS.ErrorFromName**](http://msdn.microsoft.com/en-us/library/windows/apps/br211689.aspx). This is just a JavaScript object that contains a *name* property identifying the error and a *message* property containing more information. For example, when a promise is canceled, error handlers receive an error object with both *name* and *message* name set to “Canceled”. Note also that if a consumer calls **then** on a promise that’s already in the error state, the promise will immediately (synchronously) invoke the error handler given to **then**.

The **WinJS.ErrorFromName** object is clearly what you’ll use to generate errors from within the operation whenever you need to invoke the error dispatcher. But what if you can’t even start the operation in the first place? For example, if you call **calculateIntegerSum** with bad arguments (like 0, 0) then it shouldn’t even attempt to start counting and instead return a promise in the error state. Such is the purpose of the static method [**WinJS.Promise.wrapError**](http://msdn.microsoft.com/en-us/library/windows/apps/br229784.aspx). This takes an instance of **WinJS.ErrorFromName** and returns a promise in the error state, which we would return in this case instead of a new **WinJS.Promise** instance.

The other part to all this is that although a call to the promise’s **cancel** method puts the promise itself into the error state, how do we stop the async operation that’s underway? In the previous **calculateIntegerSum** implementation, it will just keep calling **setImmediate** until the operation is complete, regardless of the state of the promise we created. In fact, if the operation calls the complete dispatcher after the promise has been canceled, the promise just ignores that completion.

What’s needed, then, is a way for the promise to tell the operation that it no longer needs to continue its work. For this reason the **WinJS.Promise** constructor takes a second function argument that’s called if the promise is canceled. In our example, a call to this function would need to prevent the next call to **setImmediate**, thereby stopping the computation. Here’s how that would look, along with proper error handling:

function calculateIntegerSum(max, step) {  
 //Return a promise in the error state for bad arguments  
 if (max < 1 || step < 1) {  
 var err = new WinJS.ErrorFromName("calculateIntegerSum", "max and step must be 1 or greater");  
 return WinJS.Promise.wrapError(err);  
 }  
  
 var \_cancel = false;  
  
 //The WinJS.Promise constructor's argument is an initializer 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 for some reason there was an error, create the error with WinJS.ErrorFromName  
 //and pass to errorDispatch  
 if (false /\* replace with any necessary error check -- we don’t have any here \*/) {  
 errorDispatch(new WinJS.ErrorFromName("calculateIntegerSum (scenario 7)", "error occurred"));  
 }  
  
 if (i >= max) {  
 //Complete--dispatch results to completed handlers  
 completeDispatch(sum);   
 } else {  
 //Dispatch intermediate results to progress handlers  
 progressDispatch(sum);  
  
 //Interrupt the operation if canceled  
 if (!\_cancel) {  
 setImmediate(iterate, { start: args.end, end: Math.min(args.end + step, max) });  
 }  
 }  
 }  
   
 setImmediate(iterate, { start: 0, end: Math.min(step, max) });  
 },  
 //Cancellation function for the WinJS.Promise constructor  
 function () {  
 \_cancel = true;  
 });  
}

Altogether, creating instances of **WinJS.Promise** has many uses. For example, if you have a library that talks to a web service through some other async method, you can wrap those operations within promises. You might also use a new promise to combine multiple async operations (or other promises!) from different sources into a single promise where you want control over all the relationships involved. Within the code of an initializer for **WinJS.Promise**, you can certainly have you own handlers for other async operations and their promises, such that you can encapsulate automatic retry mechanisms for network timeouts and such, hook into a generic progress updater UI, or add under-the-covers logging or analytics. In all these ways the rest of your code never needs to know about the details and can just deal with promises from the consumer side.

Along these lines, it’s fairly straightforward to wrap a JavaScript worker into a promise such that it looks and behaves like other asynchronous operations in WinRT. Workers, as you might know, deliver their results through a **postMessage** call that raises a **message** event on the worker object in the app. The following code, then, links that event to a promise that’s fulfilled with whatever results are delivered in that message:

// This is the function variable we're wiring up.  
var workerCompleteDispatch = null;  
  
var promiseJS = new WinJS.Promise(function (completeDispatch, errorDispatch, progressDispatch) {  
 workerCompleteDispatch = completeDispatch;  
});  
  
// Worker would be created here and stored in the 'worker' variable  
  
// Listen for worker events  
worker.onmessage = function (e) {  
 if (workerCompleteDispatch != null) {  
 workerCompleteDispatch(e.data.results); /\* event args depends on the worker \*/  
 }  
}  
  
promiseJS.done(function (result) {  
 // Output for JS worker  
});

To expand this code to handle errors from the worker, you’d save the error dispatcher in another variable, have the **message** event handler check for error information in its event args, and the call the error dispatcher instead of the complete dispatcher as appropriate.

## Helpers for Creating Promises

Besides encapsulating an arbitrary asynchronous operation with **new WinJS.Promise**, WinJS provides a few other helpers for common scenarios. One example is **WinJS.Promise.wrapError** that, as we’ve already seen, creates a promise that’s in the error state from the start.

One of the most basic helpers is the static method [**WinJS.Promise.as**](http://msdn.microsoft.com/en-us/library/windows/apps/br211664.aspx), which wraps *any* value in a promise. Such a wrapper on an already existing value will just turn right around and call any completed handler passed to **then**. This specifically allows you to treat arbitrary known values as promises, such that you can compose them (through joining or chaining) with other promises. In other words, **WinJS.Promise.as** makes it easy to intermix synchronous and asynchronous values by treating them all as promises. **WinJS.Promise.as** is also intelligent enough to not double-wrap a promise; that is, if you give it a promise it just returns that same promise. (Note that you should use **as** instead of an older and deprecated function **WinJS.Promise.wrap**.)

It is allowable to call **WinJS.Promise.as()** if you just want a promise that’s fulfilled with *undefined*. This is a shortcut for creating what’s essentially an empty promise (calling **new WinJS.Promise()**without an initializer function will fail).

The other static helper function is [**WinJS.Promise.timeout**](http://msdn.microsoft.com/en-us/library/windows/apps/br229729.aspx), which provides a convenient wrapper around **setTimeout** and **setImmediate**. This function has constructors:

* **WinJS.Promise.timeout()** creates a promise around **setImmediate**, so the promise is fulfilled (with the result of *undefined*) after the UI thread is yielded. This makes is easy to chain another async operation with **setImmediate**.
* **WinJS.Promise.timeout(<n>)** creates a promise that is fulfilled (with *undefined* again) 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 **cancel** if the period elapses.)

I won’t show the code here, but the implementation of these are easily seen in WinJS’s base.js file (search for “timeout”) and can be an instructive study. A more relevant question is “If the first two forms of **timeout** just produce *undefined*, can we use them to return some other result after the timeout?” The answer is yes: we take advantage of the fact that **then** returns another promise that’s fulfilled with the return value of the completed handler. This line of code, for example:

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

creates the promise *p* that will be fulfilled with the value 12345 after one second. In other words, **WinJS.Promise.timeout(…).then(function () { return <value>} )** is a pattern to deliver *<value>* after the given timeout. And if *<value>* itself is another promise, then it means to deliver the fulfillment value from that promise at some point after the timeout.

## Joining Parallel Promises and Other Useful Functions

As promises are often used to wrap asynchronous operations, it’s certainly possible that you can have multiple operations going on in parallel. In these cases you might want to know when either one promise in a group is fulfilled, or when all the promises in the group are fulfilled. The static functions [**WinJS.Promise.any**](http://msdn.microsoft.com/en-us/library/windows/apps/br229660.aspx) and [**WinJS.Promise.join**](http://msdn.microsoft.com/en-us/library/windows/apps/br211774.aspx) provide for this.

Both functions accept an array of value or an object with value properties. Those values can be promises, and any non-promise values are wrapped with **WinJS.Promise.as**, such that the whole array or object is composed of promises.

Here are the characteristics of **any**:

* **any** creates a single 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, and as soon as one completed handler is called, it calls whatever completed handlers the **any**promise has itself received.
* 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 list is fulfilled), the other operations in the list continue to run, calling whatever completed, error, or progress handlers are assigned to those promises individually.
* If you cancel the promise from **any**, then all the promises in the list are canceled.

As for **join**:

* **join** 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.
* 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 an array of results from the individual promises (even if those results are null or undefined). 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 **join** 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.
* If you cancel promise from **join** it cancels all the other promises that are still pending.

Beyond **any** and **join**, there are two other static **WinJS.Promise** methods to know about, which can come in handy:

* [**is**](http://msdn.microsoft.com/en-us/library/windows/apps/br211765.aspx) determines whether an arbitrary value is a promise, returning a Boolean. It basically makes sure it’s an object with a function named “then”; it does not test for “done”.
* [**theneach**](http://msdn.microsoft.com/en-us/library/windows/apps/br229727.aspx)applies completed, error, and progress handlers to a group of promises (using **then**), returning the results as another group of values inside a promise. Any of the handlers can be *null*.

## Parallel Promises with Sequential Results

With **WinJS.Promise.join** and **WinJS.Promise.any** we have the ability 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 will likely complete in a random 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?

To do this you need to join each subsequent promise to the join of all those that came before, and the bit of code with which we began this post does exactly that. Here’s that code again, but rewritten to make the promises explicit. (Assume *list*is an array of values of some sort that are used as arguments for a hypothetical promise-producing async call **doOperationAsync**):

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

var opPromise = doOperationAsync(item);

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

return join.then(function completed (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. For each item in the array, **reduce** calls the function argument, here named **callback**, which receives four arguments (only three of which are used in the code):

* *prev* The value that was returned from the *previous* call to **callback** (this is *null* for the first item).
* *item* The current value from the array.
* *i* The index of item in the list.
* *source* The original array.

For the first item in the list, we get promise which I’ll call **opPromise1**. As **prev** is *null*, we’re joining **[WinJS.Promise.as(null), opPromise1]**. 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 **opPromise1**. And if you look back at the result of a join, it’s 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, where the latter is the result of **opPromise1**.

With the next item in *list*, **callback** receives a *prev* that contains the promise from the previous **join.then**. We’ll then create a new join of **opPromise1.then** and **opPromise2**. As a result, this join will not complete until both **opPromise2** is fullfilled *and* the completed handler for **opPromise1** returns. Voila! The **completed2** handler we now attach to this join will not be called until after **completed1** has returned.

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. This guarantees that the completed handlers will be called in the same sequence as *list*.

## In Closing

In this post we’ve seen that promises in themselves are just a code construct—albeit a powerful one—to represent a specific relationship between an originator, who has values to deliver at an arbitrarily later time, and a consumer that wants to know when those values are available. As such, promises work very well to represent results from asynchronous operations, and are used extensively within Windows Store apps written in JavaScript. The specification for promises also allows sequential async operations to be chained together, with each intermediate result flowing from one link to the next.

The Windows Library for JavaScript, WinJS, provides a robust implementation of promises that you can use to wrap any kind of operation of your own. It also provides helpers for common scenarios such as joining together promises for parallel operations. With these, WinJS makes it possible to work with asynchronous operations very efficiently and effectively.

Kraig Brockschmidt  
Program Manager, Windows Ecosystem Team  
Author, [*Programming Windows 8 Apps in HTML, CSS, and JavaScript*](http://blogs.msdn.com/b/microsoft_press/archive/2012/10/29/free-ebook-programming-windows-8-apps-with-html-css-and-javascript.aspx)