You Don't Know JS: Async & Performance

Appendix A: *asynquence* Library

Chapters 1 and 2 went into quite a bit of detail about typical asynchronous programming patterns and how they're commonly solved with callbacks. But we also saw why callbacks are fatally limited in capability, which led us to Chapters 3 and 4, with Promises and generators offering a much more solid, trustable, and reason-able base to build your asynchrony on.

I referenced my own asynchronous library *asynquence* (http://github.com/getify/asynquence) -- "async" + "sequence" = "asynquence" -- several times in this book, and I want to now briefly explain how it works and why its unique design is important and helpful.

In the next appendix, we'll explore some advanced async patterns, but you'll probably want a library to make those palatable enough to be useful. We'll use *asynquence* to express those patterns, so you'll want to spend a little time here getting to know the library first.

asynquence is obviously not the only option for good async coding; certainly there are many great libraries in this space. But *asynquence* provides a unique perspective by combining the best of all these patterns into a single library, and moreover is built on a single basic abstraction: the (async) sequence.

My premise is that sophisticated JS programs often need bits and pieces of various different asynchronous patterns woven together, and this is usually left entirely up to each developer to figure out. Instead of having to bring in two or more different async libraries that focus on different aspects of asynchrony, *asynquence* unifies them into variated sequence steps, with just one core library to learn and deploy.

I believe the value is strong enough with *asynquence* to make async flow control programming with Promise-style semantics super easy to accomplish, so that's why we'll exclusively focus on that library here.

To begin, I'll explain the design principles behind *asynquence*, and then we'll illustrate how its API works with code examples.

Sequences, Abstraction Design

Understanding *asynquence* begins with understanding a fundamental abstraction: any series of steps for a task, whether they separately are synchronous or asynchronous, can be collectively thought of as a "sequence". In other words, a sequence is a container that represents a task, and is comprised of individual (potentially async) steps to complete that task.

Each step in the sequence is controlled under the covers by a Promise (see Chapter 3). That is, every step you add to a sequence implicitly creates a Promise that is wired to the previous end of the sequence. Because of the semantics of Promises, every single step advancement in a sequence is asynchronous, even if you synchronously complete the step.

Moreover, a sequence will always proceed linearly from step to step, meaning that step 2 always comes after step 1 finishes, and so on.

Of course, a new sequence can be forked off an existing sequence, meaning the fork only occurs once the main sequence reaches that point in the flow. Sequences can also be combined in various ways, including having one sequence subsumed by another sequence at a particular point in the flow.

A sequence is kind of like a Promise chain. However, with Promise chains, there is no "handle" to grab that references the entire chain. Whichever Promise you have a reference to only represents the current step in the chain plus any other steps hanging off it. Essentially, you cannot hold a reference to a Promise chain unless you hold a reference to the first Promise in the chain.

There are many cases where it turns out to be quite useful to have a handle that references the entire sequence collectively. The most important of those cases is with sequence abort/cancel. As we covered extensively in Chapter 3, Promises themselves should never be able to be canceled, as this violates a fundamental design imperative: external immutability.

But sequences have no such immutability design principle, mostly because sequences are not passed around as future-value containers that need immutable value semantics. So sequences are the proper level of abstraction to handle abort/cancel behavior. *asynquence* sequences can be `abort()` ed at any time, and the sequence will stop at that point and not go for any reason.

There's plenty more reasons to prefer a sequence abstraction on top of Promise chains, for flow control purposes.

First, Promise chaining is a rather manual process -- one that can get pretty tedious once you start creating and chaining Promises across a wide swath of your programs -- and this tedium can act counterproductively to dissuade the developer from using Promises in places where they are quite appropriate.

Abstractions are meant to reduce boilerplate and tedium, so the sequence abstraction is a good solution to this problem. With Promises, your focus is on the individual step, and there's little assumption that you will keep the chain going. With sequences, the opposite approach is taken, assuming the sequence will keep having more steps added indefinitely.

This abstraction complexity reduction is especially powerful when you start thinking about higher-order Promise patterns (beyond `race([..])` and `all([..])`.

For example, in the middle of a sequence, you may want to express a step that is conceptually like a `try..catch` in that the step will always result in success, either the intended main success resolution or a positive nonerror signal for the caught error. Or, you might want to express a step that is like a retry/until loop, where it keeps trying the same step over and over until success occurs.

These sorts of abstractions are quite nontrivial to express using only Promise primitives, and doing so in the middle of an existing Promise chain is not pretty. But if you abstract your thinking to a sequence, and consider a step as a wrapper around a Promise, that step wrapper can hide such details, freeing you to think about the flow control in the most sensible way without being bothered by the details.

Second, and perhaps more importantly, thinking of async flow control in terms of steps in a sequence allows you to abstract out the details of what types of asynchronicity are involved with each individual step. Under the covers, a Promise will always control the step, but above the covers, that step can look either like a continuation callback (the simple default), or like a real Promise, or as a run-to-completion generator, or ... Hopefully, you get the picture.

Third, sequences can more easily be twisted to adapt to different modes of thinking, such as event-, stream-, or reactive-based coding. *asynquence* provides a pattern I call "reactive sequences" (which we'll cover later) as a variation on the "reactive observable" ideas in RxJS ("Reactive Extensions"), that lets a repeatable event fire off a new sequence instance each time. Promises are one-shot-only, so it's quite awkward to express repetitious asynchrony with Promises alone.

Another alternate mode of thinking inverts the resolution/control capability in a pattern I call "iterable sequences". Instead of each individual step internally controlling its own completion (and thus advancement of the sequence), the sequence is inverted so the advancement control is through an external iterator, and each step in the *iterable sequence* just responds to the `next(..)` *iterator* control.

We'll explore all of these different variations as we go throughout the rest of this appendix, so don't worry if we ran over those bits far too quickly just now.

The takeaway is that sequences are a more powerful and sensible abstraction for complex asynchrony than just Promises (Promise chains) or just generators, and *asynquence* is designed to express that abstraction with just the right level of sugar to make async programming more understandable and more enjoyable.

asynquence API

To start off, the way you create a sequence (an *asynquence* instance) is with the `ASQ(..)` function. An `ASQ()` call with no parameters creates an empty initial sequence, whereas passing one or more values or functions to `ASQ(..)` sets up the sequence with each argument representing the initial steps of the sequence.

Note: For the purposes of all code examples here, I will use the *asynquence* top-level identifier in global browser usage: `ASQ`. If you include and use *asynquence* through a module system (browser or server), you of course can define whichever symbol you prefer, and *asynquence* won't care!

Many of the API methods discussed here are built into the core of *asynquence*, but others are provided through including the optional "contrib" plug-ins package. See the documentation for *asynquence* for whether a method is built in or defined via plug-in: http://github.com/getify/asynquence

Steps

If a function represents a normal step in the sequence, that function is invoked with the first parameter being the continuation callback, and any subsequent parameters being any messages passed on from the previous step. The step will not complete until the continuation callback is called. Once it's called, any arguments you pass to it will be sent along as messages to the next step in the sequence.

To add an additional normal step to the sequence, call `then(..)` (which has essentially the exact same semantics as the `ASQ(..)` call):

```
ASQ(
        // step 1
        function(done){
                setTimeout( function(){
                        done( "Hello" );
                }, 100 );
        },
        // step 2
        function(done,greeting) {
                setTimeout( function(){
                        done( greeting + " World" );
```

```
                }, 100 );
        }
)
// step 3
.then( function(done,msg){
        setTimeout( function(){
                done( msg.toUpperCase() );
        }, 100 );
} )
// step 4
.then( function(done,msg){
        console.log( msg );                     // HELLO WORLD
} );
```

Note: Though the name `then(..)` is identical to the native Promises API, this `then(..)` is different. You can pass as few or as many functions or values to `then(..)` as you'd like, and each is taken as a separate step. There's no two-callback fulfilled/rejected semantics involved.

Unlike with Promises, where to chain one Promise to the next you have to create and `return` that Promise from a `then(..)` fulfillment handler, with *asynquence*, all you need to do is call the continuation callback -- I always call it `done()` but you can name it whatever suits you -- and optionally pass it completion messages as arguments.

Each step defined by `then(..)` is assumed to be asynchronous. If you have a step that's synchronous, you can either just call `done(..)` right away, or you can use the simpler `val(..)` step helper:

```
// step 1 (sync)
ASQ( function(done){
        done( "Hello" );        // manually synchronous
} )
// step 2 (sync)
.val( function(greeting){
        return greeting + " World";
} )
// step 3 (async)
```

```
.then( function(done,msg){
        setTimeout( function(){
                done( msg.toUpperCase() );
        }, 100 );
} )
// step 4 (sync)
.val( function(msg){
        console.log( msg );
} );
```

As you can see, `val(..)`-invoked steps don't receive a continuation callback, as that part is assumed for you -- and the parameter list is less cluttered as a result! To send a message along to the next step, you simply use `return`.

Think of `val(..)` as representing a synchronous "value-only" step, which is useful for synchronous value operations, logging, and the like.

Errors

One important difference with *asynquence* compared to Promises is with error handling.

With Promises, each individual Promise (step) in a chain can have its own independent error, and each subsequent step has the ability to handle the error or not. The main reason for this semantic comes (again) from the focus on individual Promises rather than on the chain (sequence) as a whole.

I believe that most of the time, an error in one part of a sequence is generally not recoverable, so the subsequent steps in the sequence are moot and should be skipped. So, by default, an error at any step of a sequence throws the entire sequence into error mode, and the rest of the normal steps are ignored.

If you *do* need to have a step where its error is recoverable, there are several different API methods that can accommodate, such as `try(..)` -- previously mentioned as a kind of `try..catch` step -- or `until(..)` -- a retry loop that keeps attempting the step until it succeeds or you manually `break()` the loop. *asynquence* even has `pThen(..)` and `pCatch(..)` methods, which work identically to how normal Promise `then(..)` and `catch(..)` work (see Chapter 3), so you can do localized mid-sequence error handling if you so choose.

The point is, you have both options, but the more common one in my experience is the default. With Promises, to get a chain of steps to ignore all steps once an error occurs, you have to take care not to register a rejection handler at any step; otherwise, that error gets swallowed as handled, and the sequence may continue (perhaps unexpectedly). This kind of desired behavior is a bit awkward to properly and reliably handle.

To register a sequence error notification handler, *asynquence* provides an `or(..)` sequence method, which also has an alias of `onerror(..)`. You can call this method anywhere in the sequence, and you can register as many handlers as you'd like. That makes it easy for multiple different consumers to listen in on a sequence to know if it failed or not; it's kind of like an error event handler in that respect.

Just like with Promises, all JS exceptions become sequence errors, or you can programmatically signal a sequence error:

```
var sq = ASQ( function(done){
        setTimeout( function(){
                // signal an error for the sequence
                done.fail( "Oops" );
        }, 100 );
} )
.then( function(done){
        // will never get here
} )
.or( function(err){
        console.log( err );                     // Oops
} )
.then( function(done){
        // won't get here either
} );

// later

sq.or( function(err){
        console.log( err );                     // Oops
} );
```

Another really important difference with error handling in *asynquence* compared to native Promises is the default behavior of "unhandled exceptions". As we discussed at length in Chapter 3, a rejected Promise without a registered rejection handler will just silently hold (aka swallow) the error; you have to remember to always end a chain with a final `catch(..)`.

In *asynquence*, the assumption is reversed.

If an error occurs on a sequence, and it **at that moment** has no error handlers registered, the error is reported to the `console`. In other words, unhandled rejections are by default always reported so as not to be swallowed and missed.

As soon as you register an error handler against a sequence, it opts that sequence out of such reporting, to prevent duplicate noise.

There may, in fact, be cases where you want to create a sequence that may go into the error state before you have a chance to register the handler. This isn't common, but it can happen from time to time.

In those cases, you can also **opt a sequence instance out** of error reporting by calling `defer()` on the sequence. You should only opt out of error reporting if you are sure that you're going to eventually handle such errors:

```
var sq1 = ASQ( function(done){
        doesnt.Exist();                 // will throw exception to console
} );

var sq2 = ASQ( function(done){
        doesnt.Exist();                 // will throw only a sequence error
} )
// opt-out of error reporting
.defer();

setTimeout( function(){
        sq1.or( function(err){
                console.log( err );     // ReferenceError
        } );

        sq2.or( function(err){
```

```
                console.log( err );     // ReferenceError
        } );
}, 100 );

// ReferenceError (from sq1)
```

This is better error handling behavior than Promises themselves have, because it's the Pit of Success, not the Pit of Failure (see Chapter 3).

Note: If a sequence is piped into (aka subsumed by) another sequence -- see "Combining Sequences" for a complete description -- then the source sequence is opted out of error reporting, but now the target sequence's error reporting or lack thereof must be considered.

Parallel Steps

Not all steps in your sequences will have just a single (async) task to perform; some will need to perform multiple steps "in parallel" (concurrently). A step in a sequence in which multiple substeps are processing concurrently is called a `gate(..)` -- there's an `all(..)` alias if you prefer -- and is directly symmetric to native `Promise.all([..])`.

If all the steps in the `gate(..)` complete successfully, all success messages will be passed to the next sequence step. If any of them generate errors, the whole sequence immediately goes into an error state.

Consider:

```
ASQ( function(done){
        setTimeout( done, 100 );
} )
.gate(
        function(done){
                setTimeout( function(){
                        done( "Hello" );
                }, 100 );
        },
```

```
        function(done){
                setTimeout( function(){
                        done( "World", "!" );
                }, 100 );
        }
)
.val( function(msg1,msg2){
        console.log( msg1 );    // Hello
        console.log( msg2 );    // [ "World", "!" ]
} );
```

For illustration, let's compare that example to native Promises:

```
new Promise( function(resolve,reject){
        setTimeout( resolve, 100 );
} )
.then( function(){
        return Promise.all( [
                new Promise( function(resolve,reject){
                        setTimeout( function(){
                                resolve( "Hello" );
                        }, 100 );
                } ),
                new Promise( function(resolve,reject){
                        setTimeout( function(){
                                // note: we need a [ ] array here
                                resolve( [ "World", "!" ] );
                        }, 100 );
                } )
        ] );
} )
.then( function(msgs){
        console.log( msgs[0] ); // Hello
        console.log( msgs[1] ); // [ "World", "!" ]
} );
```

Yuck. Promises require a lot more boilerplate overhead to express the same asynchronous flow control. That's a great illustration of why the *asynquence* API and abstraction make dealing with Promise steps a lot nicer. The improvement only goes higher the more complex your asynchrony is.

Step Variations

There are several variations in the contrib plug-ins on *asynquence*'s `gate(..)` step type that can be quite helpful:

- `any(..)` is like `gate(..)`, except just one segment has to eventually succeed to proceed on the main sequence.
- `first(..)` is like `any(..)`, except as soon as any segment succeeds, the main sequence proceeds (ignoring subsequent results from other segments).
- `race(..)` (symmetric with `Promise.race([..])`) is like `first(..)`, except the main sequence proceeds as soon as any segment completes (either success or failure).
- `last(..)` is like `any(..)`, except only the latest segment to complete successfully sends its message(s) along to the main sequence.
- `none(..)` is the inverse of `gate(..)`: the main sequence proceeds only if all the segments fail (with all segment error message(s) transposed as success message(s) and vice versa).

Let's first define some helpers to make illustration cleaner:

```
function success1(done) {
        setTimeout( function(){
                done( 1 );
        }, 100 );
}

function success2(done) {
        setTimeout( function(){
                done( 2 );
        }, 100 );
}
```

```
function failure3(done) {
        setTimeout( function(){
                done.fail( 3 );
        }, 100 );
}

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

Now, let's demonstrate these `gate(..)` step variations:

```
ASQ().race(
        failure3,
        success1
)
.or( output );          // 3


ASQ().any(
        success1,
        failure3,
        success2
)
.val( function(){
        var args = [].slice.call( arguments );
        console.log(
                args            // [ 1, undefined, 2 ]
        );
} );


ASQ().first(
        failure3,
        success1,
        success2
```

```
)
.val( output );         // 1


ASQ().last(
        failure3,
        success1,
        success2
)
.val( output );         // 2

ASQ().none(
        failure3
)
.val( output )          // 3
.none(
        failure3
        success1
)
.or( output );          // 1
```

Another step variation is `map(..)`, which lets you asynchronously map elements of an array to different values, and the step doesn't proceed until all the mappings are complete. `map(..)` is very similar to `gate(..)`, except it gets the initial values from an array instead of from separately specified functions, and also because you define a single function callback to operate on each value:

```
function double(x,done) {
        setTimeout( function(){
                done( x * 2 );
        }, 100 );
}

ASQ().map( [1,2,3], double )
.val( output );                                 // [2,4,6]
```

Also, `map(..)` can receive either of its parameters (the array or the callback) from messages passed from the previous step:

```
function plusOne(x,done) {
        setTimeout( function(){
                done( x + 1 );
        }, 100 );
}

ASQ( [1,2,3] )
.map( double )                  // message `[1,2,3]` comes in
.map( plusOne )                 // message `[2,4,6]` comes in
.val( output );                 // [3,5,7]
```

Another variation is `waterfall(..)`, which is kind of like a mixture between `gate(..)`'s message collection behavior but `then(..)`'s sequential processing.

Step 1 is first executed, then the success message from step 1 is given to step 2, and then both success messages go to step 3, and then all three success messages go to step 4, and so on, such that the messages sort of collect and cascade down the "waterfall".

Consider:

```
function double(done) {
        var args = [].slice.call( arguments, 1 );
        console.log( args );

        setTimeout( function(){
                done( args[args.length - 1] * 2 );
        }, 100 );
}

ASQ( 3 )
.waterfall(
        double,                                 // [ 3 ]
```

```
        double,                                 // [ 6 ]
        double,                                 // [ 6, 12 ]
        double                                  // [ 6, 12, 24 ]
)
.val( function(){
        var args = [].slice.call( arguments );
        console.log( args );    // [ 6, 12, 24, 48 ]
} );
```

If at any point in the "waterfall" an error occurs, the whole sequence immediately goes into an error state.

Error Tolerance

Sometimes you want to manage errors at the step level and not let them necessarily send the whole sequence into the error state. *asynquence* offers two step variations for that purpose.

`try(..)` attempts a step, and if it succeeds, the sequence proceeds as normal, but if the step fails, the failure is turned into a success message formated as `{ catch: .. }` with the error message(s) filled in:

```
ASQ()
.try( success1 )
.val( output )                  // 1
.try( failure3 )
.val( output )                  // { catch: 3 }
.or( function(err){
        // never gets here
} );
```

You could instead set up a retry loop using `until(..)`, which tries the step and if it fails, retries the step again on the next event loop tick, and so on.

This retry loop can continue indefinitely, but if you want to break out of the loop, you can call the `break()` flag on the completion trigger, which sends the main sequence into an error state:

```
var count = 0;

ASQ( 3 )
.until( double )
.val( output )                                  // 6
.until( function(done){
        count++;

        setTimeout( function(){
                if (count < 5) {
                        done.fail();
                }
                else {
                        // break out of the `until(..)` retry loop
                        done.break( "Oops" );
                }
        }, 100 );
} )
.or( output );                                  // Oops
```

Promise-Style Steps

If you would prefer to have, inline in your sequence, Promise-style semantics like Promises' `then(..)` and `catch(..)` (see Chapter 3), you can use the `pThen` and `pCatch` plug-ins:

```
ASQ( 21 )
.pThen( function(msg){
        return msg * 2;
} )
.pThen( output )                                // 42
.pThen( function(){
        // throw an exception
        doesnt.Exist();
} )
```

```
.pCatch( function(err){
        // caught the exception (rejection)
        console.log( err );                     // ReferenceError
} )
.val( function(){
        // main sequence is back in a
        // success state because previous
        // exception was caught by
        // `pCatch(..)`
} );
```

`pThen(..)` and `pCatch(..)` are designed to run in the sequence, but behave as if it was a normal Promise chain. As such, you can either resolve genuine Promises or *asynquence* sequences from the "fulfillment" handler passed to `pThen(..)` (see Chapter 3).

Forking Sequences

One feature that can be quite useful about Promises is that you can attach multiple `then(..)` handler registrations to the same promise, effectively "forking" the flow-control at that promise:

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

// fork 1 (from `p`)
p.then( function(msg){
        return msg * 2;
} )
.then( function(msg){
        console.log( msg );             // 42
} )

// fork 2 (from `p`)
p.then( function(msg){
        console.log( msg );             // 21
} );
```

The same "forking" is easy in *asynquence* with `fork()`:

```
var sq = ASQ(..).then(..).then(..);

var sq2 = sq.fork();

// fork 1
sq.then(..)..;

// fork 2
sq2.then(..)..;
```

Combining Sequences

The reverse of `fork()`ing, you can combine two sequences by subsuming one into another, using the `seq(..)` instance method:

```
var sq = ASQ( function(done){
        setTimeout( function(){
                done( "Hello World" );
        }, 200 );
} );

ASQ( function(done){
        setTimeout( done, 100 );
} )
// subsume `sq` sequence into this sequence
.seq( sq )
.val( function(msg){
        console.log( msg );             // Hello World
} )
```

`seq(..)` can either accept a sequence itself, as shown here, or a function. If a function, it's expected that the function when called will return a sequence, so the preceding code could have been done with:

```
// ..
.seq( function(){
        return sq;
} )
// ..
```

Also, that step could instead have been accomplished with a `pipe(..)`:

```
// ..
.then( function(done){
        // pipe `sq` into the `done` continuation callback
        sq.pipe( done );
} )
// ..
```

When a sequence is subsumed, both its success message stream and its error stream are piped in.

Note: As mentioned in an earlier note, piping (manually with `pipe(..)` or automatically with `seq(..)`) opts the source sequence out of error-reporting, but doesn't affect the error reporting status of the target sequence.

Value and Error Sequences

If any step of a sequence is just a normal value, that value is just mapped to that step's completion message:

```
var sq = ASQ( 42 );

sq.val( function(msg){
```

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

If you want to make a sequence that's automatically errored:

```
var sq = ASQ.failed( "Oops" );

ASQ()
.seq( sq )
.val( function(msg){
        // won't get here
} )
.or( function(err){
        console.log( err );             // Oops
} );
```

You also may want to automatically create a delayed-value or a delayed-error sequence. Using the `after` and `failAfter` contrib plug-ins, this is easy:

```
var sq1 = ASQ.after( 100, "Hello", "World" );
var sq2 = ASQ.failAfter( 100, "Oops" );

sq1.val( function(msg1,msg2){
        console.log( msg1, msg2 );              // Hello World
} );

sq2.or( function(err){
        console.log( err );                             // Oops
} );
```

You can also insert a delay in the middle of a sequence using `after(..)`:

```
ASQ( 42 )
// insert a delay into the sequence
.after( 100 )
.val( function(msg){
        console.log( msg );             // 42
} );
```

Promises and Callbacks

I think *asynquence* sequences provide a lot of value on top of native Promises, and for the most part you'll find it more pleasant and more powerful to work at that level of abstraction. However, integrating *asynquence* with other non-*asynquence* code will be a reality.

You can easily subsume a promise (e.g., thenable -- see Chapter 3) into a sequence using the `promise(..)` instance method:

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

ASQ()
.promise( p )                   // could also: `function(){ return p; }`
.val( function(msg){
        console.log( msg );     // 42
} );
```

And to go the opposite direction and fork/vend a promise from a sequence at a certain step, use the `toPromise` contrib plug-in:

```
var sq = ASQ.after( 100, "Hello World" );

sq.toPromise()
// this is a standard promise chain now
.then( function(msg){
```

```
        return msg.toUpperCase();
} )
.then( function(msg){
        console.log( msg );             // HELLO WORLD
} );
```

To adapt *asynquence* to systems using callbacks, there are several helper facilities. To automatically generate an "error-first style" callback from your sequence to wire into a callback-oriented utility, use `errfcb`:

```
var sq = ASQ( function(done){
        // note: expecting "error-first style" callback
        someAsyncFuncWithCB( 1, 2, done.errfcb )
} )
.val( function(msg){
        // ..
} )
.or( function(err){
        // ..
} );

// note: expecting "error-first style" callback
anotherAsyncFuncWithCB( 1, 2, sq.errfcb() );
```

You also may want to create a sequence-wrapped version of a utility -- compare to "promisory" in Chapter 3 and "thunkory" in Chapter 4 -- and *asynquence* provides `ASQ.wrap(..)` for that purpose:

```
var coolUtility = ASQ.wrap( someAsyncFuncWithCB );

coolUtility( 1, 2 )
.val( function(msg){
        // ..
} )
.or( function(err){
```

```
        // ..
} );
```

Note: For the sake of clarity (and for fun!), let's coin yet another term, for a sequence-producing function that comes from `ASQ.wrap(..)`, like `coolUtility` here. I propose "sequory" ("sequence" + "factory").

Iterable Sequences

The normal paradigm for a sequence is that each step is responsible for completing itself, which is what advances the sequence. Promises work the same way.

The unfortunate part is that sometimes you need external control over a Promise/step, which leads to awkward "capability extraction".

Consider this Promises example:

```
var domready = new Promise( function(resolve,reject){
        // don't want to put this here, because
        // it belongs logically in another part
        // of the code
        document.addEventListener( "DOMContentLoaded", resolve );
} );

// ..

domready.then( function(){
        // DOM is ready!
} );
```

The "capability extraction" anti-pattern with Promises looks like this:

```
var ready;

var domready = new Promise( function(resolve,reject){
        // extract the `resolve()` capability
        ready = resolve;
} );

// ..

domready.then( function(){
        // DOM is ready!
} );

// ..

document.addEventListener( "DOMContentLoaded", ready );
```

Note: This anti-pattern is an awkward code smell, in my opinion, but some developers like it, for reasons I can't grasp.

asynquence offers an inverted sequence type I call "iterable sequences", which externalizes the control capability (it's quite useful in use cases like the `domready`):

```
// note: `domready` here is an *iterator* that
// controls the sequence
var domready = ASQ.iterable();

// ..

domready.val( function(){
        // DOM is ready
} );

// ..
```

```
document.addEventListener( "DOMContentLoaded", domready.next );
```

There's more to iterable sequences than what we see in this scenario. We'll come back to them in Appendix B.

Running Generators

In Chapter 4, we derived a utility called `run(..)` which can run generators to completion, listening for `yield` ed Promises and using them to async resume the generator. *asynquence* has just such a utility built in, called `runner(..)`.

Let's first set up some helpers for illustration:

```
function doublePr(x) {
        return new Promise( function(resolve,reject){
                setTimeout( function(){
                        resolve( x * 2 );
                }, 100 );
        } );
}

function doubleSeq(x) {
        return ASQ( function(done){
                setTimeout( function(){
                        done( x * 2)
                }, 100 );
        } );
}
```

Now, we can use `runner(..)` as a step in the middle of a sequence:

```
ASQ( 10, 11 )
.runner( function*(token){
```

```
        var x = token.messages[0] + token.messages[1];

        // yield a real promise
        x = yield doublePr( x );

        // yield a sequence
        x = yield doubleSeq( x );

        return x;
} )
.val( function(msg){
        console.log( msg );                     // 84
} );
```

Wrapped Generators

You can also create a self-packaged generator -- that is, a normal function that runs your specified generator and returns a sequence for its completion -- by `ASQ.wrap(..)` ing it:

```
var foo = ASQ.wrap( function*(token){
        var x = token.messages[0] + token.messages[1];

        // yield a real promise
        x = yield doublePr( x );

        // yield a sequence
        x = yield doubleSeq( x );

        return x;
}, { gen: true } );

// ..

foo( 8, 9 )
.val( function(msg){
```

```
        console.log( msg );                     // 68
} );
```

There's a lot more awesome that `runner(..)` is capable of, but we'll come back to that in Appendix B.

Review

asynquence is a simple abstraction -- a sequence is a series of (async) steps -- on top of Promises, aimed at making working with various asynchronous patterns much easier, without any compromise in capability.

There are other goodies in the *asynquence* core API and its contrib plug-ins beyond what we saw in this appendix, but we'll leave that as an exercise for the reader to go check the rest of the capabilities out.

You've now seen the essence and spirit of *asynquence*. The key take away is that a sequence is comprised of steps, and those steps can be any of dozens of different variations on Promises, or they can be a generator-run, or... The choice is up to you, you have all the freedom to weave together whatever async flow control logic is appropriate for your tasks. No more library switching to catch different async patterns.

If these *asynquence* snippets have made sense to you, you're now pretty well up to speed on the library; it doesn't take that much to learn, actually!

If you're still a little fuzzy on how it works (or why!), you'll want to spend a little more time examining the previous examples and playing around with *asynquence* yourself, before going on to the next appendix. Appendix B will push *asynquence* into several more advanced and powerful async patterns.