ProtoPromise

Most efficient C# library for management of asynchronous operations.

This library is fully interoperable with C#'s `Task`s and Unity's Coroutines.

ProtoPromise took inspiration from ES6 Promises (javascript), RSG Promises (C#), uPromise (C#/Unity), and TPL and improved upon their short-comings. This library conforms to the Promises/A+ Spec as far as is possible with C# (using static typing instead of dynamic), and further extends it to support Cancelations and Progress.

Contents

Creating a Promise for an Async Operation

Import the namespace:

```
using Proto.Promises;
```

Create a deferred before you start the async operation:

```
var deferred = Promise.NewDeferred<string>();
```

The type of the deferred should reflect the result of the asynchronous operation.

Then initiate your async operation and return the promise to the caller.

```
return deferred.Promise;
```

Upon completion of the async op the promise is resolved via the deferred:

```
deferred.Resolve(value);
```

The promise is rejected on error/exception:

```
deferred.Reject(error);
```

To see it in context, here is an example function that downloads text from a URL. The promise is resolved when the download completes. If there is an error during download, say *unresolved domain name*, then the promise is rejected:

```
public Promise<string> Download(string url)
{
    var deferred = Promise.NewDeferred<string>();    // Create deferred.
    using (var client = new WebClient())
    {
        client.DownloadStringCompleted += (s, ev) =>  // Monitor event for download completed.
        {
            if (ev.Error != null)
            {
                deferred.Reject(ev.Error);   // Error during download, reject the promise.
            }
            else
            {
                deferred.Resolve(ev.Result); // Downloaded completed successfully, resolve the promise.
            }
        };

        client.DownloadStringAsync(new Uri(url), null); // Initiate async op.
    }

    return deferred.Promise; // Return the promise so the caller can await resolution (or error).
}
```

Creating a Promise, Alternate Method

There is another way to create a promise that replicates the JavaScript convention of passing a *resolver* function into the constructor. The only difference is a deferred is passed to the function instead of another function. The deferred is what controls the resolution or rejection of the promise, just as it was above. This allows you to express the previous example like this:

```
public Promise<string> Download(string url)
{
  return Promise.New<string>(deferred =>
  {
      using (var client = new WebClient())
      {
          client.DownloadStringCompleted += (s, ev) =>  // Monitor event for download completed.
          {
              if (ev.Error != null)
              {
                  deferred.Reject(ev.Error);   // Error during download, reject the promise.
              }
              else
              {
                  deferred.Resolve(ev.Result); // Downloaded completed successfully, resolve the promise.
              }
          };

          client.DownloadStringAsync(new Uri(url), null); // Initiate async op.
      }
  });
}
```

With this method, if the function throws an exception before the deferred is settled, the deferred/promise will be rejected with that exception.

Waiting for an Async Operation to Complete

The simplest and most common usage is to register a resolve handler to be invoked on completion of the async op:

```
Download("http://www.google.com")
    .Then(html => Console.WriteLine(html));
```

This snippet downloads the front page from Google and prints it to the console.

In this case, because the operation can fail, you will also want to register an error hander:

```
Download("http://www.google.com")
    .Then(html => Console.WriteLine(html))
    .Catch((Exception error) => Console.WriteLine("An error occured while downloading: " + error));
```

The chain of processing for a promise ends as soon as an error/exception occurs. In this case when an error occurs the *Reject* handler would be called, but not the *Resolve* handler. If there is no error, then only the *Resolve* handler is called.

Chaining Async Operations

Multiple async operations can be chained one after the other using *Then*:

```
Download("http://www.google.com")
    .Then(html => Download(ExtractFirstLink(html))) // Extract the first link and download it and wait for the download to complete.
    .Then(firstLinkHtml => Console.WriteLine(firstLinkHtml))
    .Catch((Exception error) => Console.WriteLine("An error occured while downloading: " + error));
```

Here we are chaining another download onto the end of the first download. The first link in the html is extracted and we then download that. *Then* expects the return value to be another promise. The chained promise can have a different *result type*.

Transforming the Results

Sometimes you will want to simply transform or modify the resulting value without chaining another async operation.

```
Download("http://www.google.com")
    .Then(html => ExtractAllLinks(html))   // Extract all links and return an array of strings.
    .Then(links =>                      // The input here is an array of strings.
        foreach (var link in links)
        {
            Console.WriteLine(link);
        }
    ));
```

As demonstrated, the type of the value can also be changed during transformation. In the previous snippet a `Promise<string>` is transformed to a `Promise<string[]>`.

Promise Types

Promises may either be a value promise (`Promise<T>`), or a non-value promise (`Promise`). A value promise represents an asynchronous operation that will result in a value, and a non-value promise represents an asynchronous operation that simply does something without returning anything. In games, this is useful for composing and linking animations and other effects.

```
RunAnimation("Foo")                         // RunAnimation returns a promise that
    .Then(() => RunAnimation("Bar"))        // is resolved when the animation is complete.
    .Then(() => PlaySound("AnimComplete"));
```

`Promise<T>` inherits from `Promise`, so any value promise can be used like it is a non-value promise (the onResolved delegate can ignore its value), and can be casted and passed around as such. Besides casting, you can convert any value promise to a non-value promise and vice-versa via the `Then` method. The type of delegate you pass in will determine the type of the promise that is returned:

```
public Promise<TimeSpan> DownloadTest() // Returns a promise that yields how long it took to download google.com
{
    Stopwatch watch = Stopwatch.StartNew();
    return Download("http://www.google.com")    // <---- string promise
        .Then(html =>                           // <---- non-value promise
        {
            watch.Stop(); // Download is done, stop the timer.
            Console.Log("Cool, Google works!");
            // Don't return anything
        })
        .Then(() =>                             // <---- TimeSpan promise
        {
            return watch.Elapsed;               // Return how much time elapsed since the download started.
        });
}
```

If the delegate returns nothing/void, it will become a non-value promise. If it returns an object or value, it will become a value promise of that object/value's type. Likewise, if the delegate returns another promise, the promise returned from `.Then` will adopt the type and state of that promise.

Error Handling

An error raised in a callback aborts the function and all subsequent onResolved callbacks in the chain, until an onRejected callback is encountered:

```
promise
    .Then(v => { throw new Exception(); return v; })      // <--- An error here aborts all subsequent callbacks...
    .Then(v => DoSomething())
    .Then(() => DoSomethingElse())
    .Catch((Exception e) => HandleError(e));    // <--- Until the error handler is invoked here.
```

Type Matching Error Handling

Promises can be rejected with any type of object or value, so you may decide to filter the type you want to handle:

```
rejectedPromise
    .Catch((ArgumentException e) => HandleArgumentError(e))
    .Catch((string e) => HandleStringError(e))
    .Catch((object e) => HandleAnyError(e))
    .Catch(() => HandleError());
```

A rejected reason will propagate down the chain of catches until it encounters a type that it can be assigned to. The very last `Catch` that does not accept a type or value will catch everything no matter what it is. If an earlier reject handler catches, later catches will not be ran unless another error occurs.

Note: the rejected value passed into the onRejected callback will never be null. If a null value is passed into `Deferred.Reject`, it will be transformed into a `NullReferenceException`.

Caught Error Continuation

When a promise is rejected and that rejection is caught, the next promise is resolved if the rejection handler returns without throwing any exceptions:

```
rejectedPromise
    .Catch(() => GetFallbackString())
    .Then((string s) => Console.WriteLine(s));
```

Unlike resolve handlers, which can transform the promise into any other type of promise, reject handlers can only keep the same promise type, or transform it to a non-value promise. Thus, you can also have the reject handler run another async operation and adopt its state the same way we did in Chaining Async Operations:

```
rejectedPromise
    .Catch(() => DownloadSomethingElse(fallbackUrl));
```

Unhandled Rejections

When `Catch` is omitted, or none of the filters apply, a `System.AggregateException` (which contains `UnhandledException`s that wrap the rejections) is thrown the next time Promise.Manager.HandleCompletes(AndProgress) is called, which happens automatically every frame if you're in Unity. You can optionally reroute unhandled rejections to prevent the AggregateException (This is routed to `UnityEngine.Debug.LogException` by default in Unity.)

`UnhandledException` s contain the full causality trace so you can more easily debug what caused an error to occur in your async functions. Only available in `DEBUG` mode, for performance reasons (See Compiler Options).

Promises that are already settled

For convenience and optimizations, there are methods to get promises that are already resolved, rejected, or canceled:

```
var resolvedNonValuePromise = Promise.Resolved();

var resolvedStringPromise = Promise.Resolved("That was fast");

var rejectedNonValuePromise = Promise.Rejected("Something went wrong!");

var rejectedIntPromise = Promise.Rejected<int, string>("Something went wrong!");

var canceledNonValuePromise = Promise.Canceled("We don't actually need this anymore.");

var canceledIntPromise = Promise.Canceled<int, string>("We don't actually need this anymore.");
```

This is useful if the operation actually completes synchronously but you still need to return a promise.

Progress reporting

Promises can additionally report their progress towards completion, allowing the implementor to give the user feedback on the asynchronous operation.

Promises report their progress as a value from `0` to `1`. You can register a progress listener like so:

```
promise
    .Progress(progress =>
    {
        progressBar.SetProgress(progress);
        progressText.SetText( ((int) (progress * 100f)).ToString() + "%" );
    };
```

Progress can be reported through the deferred, and if it is reported, progress *must* be reported between 0 and 1 inclusive:

```
Promise WaitForSeconds(float seconds)
{
    var deferred = Promise.NewDeferred();
    StartCoroutine(_Countup());
    return deferred.Promise;

    IEnumerator _Countup()
    {
        for (float current = 0f; current < seconds; current += Time.Deltatime)
        {
            yield return null;
            deferred.ReportProgress(current / seconds); // Report the progress, normalized between 0 and 1.
        }
        deferred.ReportProgress(1f);
        deferred.Resolve();
    }
}
```

Reporting progress to a deferred is entirely optional, but even if progress is never reported through the deferred, it will always be reported as `1` after the promise is resolved.

Progress will always be normalized, no matter how long the promise chain is:

```
Download("google.com")                   // <---- This will report 0.0f - 0.25f
    .Then(() => WaitForSeconds(1f))      // <---- This will report 0.25f - 0.5f
    .Then(() => Download("bing.com"))    // <---- This will report 0.5f - 0.75f
    .Then(() => WaitForSeconds(1f))      // <---- This will report 0.75f - 1.0f
    .Progress(progressBar.SetProgress)
    .Then(() => Console.Log("Downloads and extra waits complete."));
```

Combining Multiple Async Operations

All Parallel

The `All` function combines multiple async operations to run in parallel. It converts a collection of promises or a variable length parameter list of promises into a single promise that yields a collection.

Say that each promise yields a value of type T, the resulting promise then yields a collection with values of type T.

Here is an example that extracts links from multiple pages and merges the results:

```
Promise.All(Download("http://www.google.com"), Download("http://www.bing.com"))  // Download each URL.
    .Then(pages =>                          // Receives collection of downloaded pages.
        pages.SelectMany(
            page => ExtractAllLinks(page)   // Extract links from all pages then flatten to single collection of links.
        )
    )
    .Then(links =>                          // Receives the flattened collection of links from all pages at once.
    {
        foreach (var link in links)
        {
            Console.WriteLine(link);
        }
    });
```

Progress from an All promise will be normalized from all of the input promises.

Merge Parallel

The `Merge` function behaves just like the `All` function, except that it can be used to combine multiple types, and instead of yielding an `IList<T>`, it yields a `ValueTuple<>` that contains the types of the promises provided to the function.

```
Promise.Merge(Download("http://www.google.com"), DownloadImage("http://www.example.com/image.jpg"))  // Download HTML and image.
    .Then(values =>                         // Receives ValueTuple<string, Texture>.
    {
        Console.WriteLine(values.Item1);    // Print the HTML.
        image.SetTexture(values.Item2);     // Assign the texture to an image object.
    });
```

Race Parallel

The `Race` function is similar to the `All` function, but it is the first async operation that settles that wins the race and the promise adopts its state.

```
Promise.Race(Download("http://www.google.com"), Download("http://www.bing.com"))  // Download each URL.
    .Then(html => Console.Log(html));                       // Both pages are downloaded, but only
                                                            // log the first one downloaded.
```

Progress from a Race promise will be the maximum of those reported by all the input promises.

First Parallel

The `First` function is almost idential to `Race` except that if a promise is rejected or canceled, the first promise will remain pending until one of the input promises is resolved or they are all rejected/canceled.

Sequence

The `Sequence` function builds a single promise that wraps multiple sequential operations that will be invoked one after the other.

Multiple promise-yielding functions are provided as input, these are chained one after the other and wrapped in a single promise that is resolved once the sequence has completed.

```
var sequencePromise = Promise.Sequence(
    () => RunAnimation("Foo"),
    () => RunAnimation("Bar"),
    () => PlaySound("AnimComplete")
);
```

Configuration

You can change whether or not objects will be pooled via `Promise.Config.ObjectPooling`. Enabling pooling reduces GC pressure.

If you are in DEBUG mode, you can configure when additional stacktraces will be generated via `Promise.Config.DebugCausalityTracer`.

`Promise.Config.UncaughtRejectionHandler` allows you to route unhandled rejections through a delegate instead of being thrown.

`Promise.Config.WarningHandler` allows you to route warnings.

Compiler Options

Progress can be disabled if you don't intend to use them and want to save a little memory/cpu cycles. You can disable progress by adding `PROTO_PROMISE_PROGRESS_DISABLE` to your compiler symbols.

By default, debug options are tied to the `DEBUG` compiler symbol, which is defined by default in the Unity Editor and not defined in release builds. You can override that by defining `PROTO_PROMISE_DEBUG_ENABLE` to force debugging on in release builds, or `PROTO_PROMISE_DEBUG_DISABLE` to force debugging off in debug builds (or in the Unity Editor). If both symbols are defined, `ENABLE` takes precedence.

Advanced

Cancelations

Cancelation tokens are primarily used to cancel promises, but can be used to cancel anything. They come in 3 parts: `CancelationSource`, `CancelationToken`, and `CancelationRegistration`.

Cancelation Source

A `CancelationSource` is what is used to actually cancel a token. When a consumer wants to cancel a producer's operation, it creates a `CancelationSource` via `CancelationSource.New()` and caches it somewhere (usually in a private field). When it determines it no longer needs the result of the operation, it calls `CancelationSource.Cancel()`, optionally providing a cancelation reason.

When you are sure that the operation has been fully cleaned up, you must dispose of the source: `CancelationSource.Dispose()`. This usually makes most sense to do it in a promise's Finally callback.

You can get the token to pass to the producer from the `CancelationSource.Token` property.

Cancelation Token

A `CancelationToken` is what is passed around to listen for a cancelation event. Tokens are read-only, meaning it cannot be canceled without the source. You can use the token to pass into functions (like `Promise.Then`) without worrying about it being canceled from within those functions.

You can register a callback to the token that will be invoked when the source is canceled:

```
public void Func(CancelationToken token)
{
    token.Register(reasonContainer => Console.Log("token was canceled with reason: " + reasonContainer.Value));
}
```

If the source is disposed without being canceled, the callback will not be invoked.

You can check whether the token is already canceled:

```
public IEnumerator FuncEnumerator(CancelationToken token)
{
    while (!token.IsCancelationRequested)
    {
        Console.Log("Doing something");
        yield return null;
    }
    Console.Log("token was canceled");
}
```

Cancelation Registration

When you register a callback to a token, it returns a `CancelationRegistration` which can be used to unregister the callback.

```
CancelationRegistration registration = token.Register(reasonContainer => Console.Log("This won't get called."));

// ... later, before the source is canceled
registration.Unregister();
```

If the registration is unregistered before the source is canceled, the callback will not be invoked. Once a registration has been unregistered, it cannot be re-registered. You must register a new callback to the token if you wish to do so.

Canceling Promises

Promise implementations usually do not allow cancelations, but I thought it would be an invaluable addition to this library.

Promises can be canceled 2 ways: passing a `CancelationToken` into `Promise.{Then, Catch, ContinueWith}` or `Promise.NewDeferred`, or by throwing a Cancelation Exception. When a promise is canceled, all promises that have been chained from it will be canceled with the same reason.

```
CancelationSource cancelationSource = CancelationSource.New();

Download("http://www.google.com");                              // <---- This will run to completion if no errors occur.
    .Then(html => Console.Log(html), cancelationSource.Token).      // <---- This will be canceled before the download completes and
    .Then(() => Download("http://www.bing.com"))                // <---- This will also be canceled and will not run.
    .Then(html => Console.Log(html))                            // <---- This will also be canceled and will not run.
    .Finally(cancelationSource.Dispose);                        // Remember to always dispose of the cancelation source when it'

// ... later, before the first download is complete
cancelationSource.Cancel("Page no longer needed.");             // <---- This will stop the callbacks from being ran, but will n
```

Cancelations can be caught, similar to how rejections are caught, except you cannot filter on the type. Instead, a `Promise.ReasonContainer` is passed into the onCanceled delegate which you can use to access the cancelation reason:

```
cancelablePromise
    .CatchCancelation((Promise.ReasonContainer reason) =>
    {
        Console.Log("Download was canceled! Reason: " + reason.Value);
    });
```

Another difference is that `CatchCancelation` returns the same promise instead of a new promise. Also, unlike catching rejections, the cancelation does not stop when it is caught. The delegate is run, then the cancelation continues down the promise chain:

```
cancelablePromise
    .CatchCancelation((Promise.ReasonContainer reason) =>
    {
        Console.Log("Download was canceled! Reason: " + reason.Value);      // <--- This will run first
    })
    .Then(html => Console.Log(html))                                        // <--- This will not run
    .CatchCancelation((Promise.ReasonContainer reason) =>
    {
        Console.Log("Download was canceled for some reason...");            // <--- Then this will run second
    });
```

Cancelations usually cancel the entire promise chain from the promise that was canceled. The only way you can continue a promise chain after a cancelation is with ContinueWith.

Cancelations always propagate downwards, and never upwards:

```
CancelationSource cancelationSource = CancelationSource.New();

Download("http://www.google.com")                                       // <---- This will *not* be canceled and will run to com
    .Then(html => Console.Log(html))                                    // <---- This will also *not* be canceled and will run
    .Then(() => Download("http://www.bing.com"), cancelationSource.Token)   // <---- This will be canceled before the download start
    .Then(html => Console.Log(html))                                    // <---- This will also be canceled and will not run.
    .Finally(cancelationSource.Dispose);                                // Remember to always dispose of the cancelation source

// ... later, before the first download is complete
cancelationSource.Cancel("Bing no longer needed.");
```

Unhandled Cancelations

Unlike rejections, cancelations are considered part of normal program flow, and will not be thrown. Therefore, catching cancelations is entirely optional.

Special Exceptions

Normally, an `Exception` thrown in an `onResolved` or `onRejected` callback will reject the promise with that exception. There are, however, a few special exceptions that can be thrown to produce different behaviour:

Rethrow

`throw Promise.Rethrow` can be used if you want to do something if a rejection occurs, but not suppress that rejection. Throwing `Promise.Rethrow` will rethrow that rejection, preserving its stacktrace (if applicable). This works just like `throw;` in synchronous catch clauses. This is only valid when used in `onRejected` callbacks. If accessed in other contexts, it will throw an `InvalidOperationException`.

RejectException

`throw Promise.RejectException(reason)` can be used to reject the promise with a reason that is not an `Exception`. If reason is an `Exception`, you may want to just throw it directly.

CancelException

`throw Promise.CancelException(reason)` can be used to cancel the promise with any reason, or `throw Promise.CancelException()` to cancel the promise without a reason. You can also throw an `OperationCanceledException`, which is equivalent to `Promise.CancelException()`.

Error Retries

What I especially love above this system is you can implement retries through a technique I call "Asynchronous Recursion":

```
public Promise<string> Download(string url, int maxRetries = 0)
{
    return Download(url)
        .Catch(() =>
        {
            if (maxRetries <= 0)
            {
                throw Promise.Rethrow; // Rethrow the rejection without processing it so that the caller can catch it.
            }
            Console.Log($"There was an error downloading {url}, retrying...");
            return Download(url, maxRetries - 1);
        };
}
```

Even though the recursion can go extremely deep or shallow, the promise's progress will still be normalized between 0 and 1. Though, a caveat to this is if the first attempt succeeds, the progress will go up to 0.5, then immediately jump to 1. Otherwise you might notice it behave like 0.5, 0.75, 0.875, 0.9375, ...

Async recursion is just as powerful as regular recursion, but it is also just as dangerous, if not more. If you mess up on regular recursion, your program will immediately crash from a `StackOverflowException`. Async recursion with this library will never crash from a stack overflow due to the iterative implementation, however if you don't do it right, it will eventually crash from an `OutOfMemoryException` due to each call waiting for the next and creating a new promise each time, consuming your heap space. Because promises can remain pending for an indeterminate amount of time, this error can potentially take a long time to show itself and be difficult to track down. So be very careful when implementing async recursion, and remember to always have a base case!

Multiple-Consumer

Multiple callbacks can be added to a single promise object which will be invoked in the order that they are added.

Capture Values

The C# compiler allows capturing variables inside delegates, known as closures. This involves creating a new object and a new delegate for every closure. These objects will eventually need to be garbage collected when the delegate is no longer reachable.

To solve this issue, I added capture values to the library. Every method that accepts a delegate can optionally take any value as a parameter, and pass that value as the first argument to the delegate. To capture multiple values, you should pass a `System.ValueTuple<>` that contains the values you wish to capture. The error retry example can be rewritten to reduce allocations:

```
public Promise<string> Download(string url, int maxRetries = 0)
{
    return Download(url)
        .Catch((url, maxRetries), cv => // Capture url and maxRetries in a System.ValueTuple<string, int>
        {
            var (_url, retryCount) = cv; // Deconstruct the value tuple (C# 7 feature)
            if (retryCount <= 0)
            {
                throw Promise.Rethrow; // Rethrow the rejection without processing it so that the caller can catch it.
            }
            Console.Log($"There was an error downloading {_url}, retrying...");
            return Download(_url, retryCount - 1);
        });
}
```

When the C# compiler sees a lamda expression that does not capture/close any variables, it will cache the delegate statically, so there is only one instance in the program. If the lambda only captures `this`, it's not quite as bad as capturing local variables, as the compiler will generate a cached delegate in the class. This means there is one delegate per instance. We can reduce that to one delegate in the program by passing `this` as the capture value.

Note: Visual Studio will tell you what variables are captured/closed if you hover the `=>`. You can use that information to optimize your delegates.

See Understanding Then for information on all the different ways you can capture values with the `Then` overloads.

Promise Retention

This is not recommended. See Promises that are already settled for a safer option.

If for some reason you wish to hold onto a promise reference and re-use it even after it has settled, you may call `promise.Retain();`. Then when you are finished with it, you must call `promise.Release();` and clear your reference `promise = null;`. All retain calls must come before release calls, and they must be made in pairs.

Async/Await

Async

Starting with C# 7.0, we are now able to make task-like types which can be returned from an `async` function. Promises are task-like and thus can be returned from `async` functions. So you can declare `async Promise` or `async Promise<T>` and then you can `await` any awaitable within that function, and the returned promise will resolve when all of those awaitables have completed. If an exception is thrown, the promise will be rejected with that exception unless it is a Special Exception. If an `OperationCanceledException` is thrown, the promise will be canceled.

Await

Promises are awaitable. This means you can use the `await` keyword to wait for the promise to complete in any `async` function, even if that function does not return a promise. If the promise is a value promise (`Promise<T>`), you can get the value from it like `T value = await promise`. If the promise is rejected, an `UnhandledException` is thrown. If the promise is canceled, a `CanceledException` is thrown.

For example, the error retry method can be re-written using async/await like this:

```
public async Promise<string> Download(string url, int maxRetries = 0)
{
    try
    {
        return await Download(url);
    }
    catch
    {
        if (maxRetries <= 0)
        {
            throw; // Rethrow the rejection without processing it so that the caller can catch it.
        }
        Console.Log($"There was an error downloading {url}, retrying...");
        return await Download(url, maxRetries - 1);
    };
}
```

Async/Await pitfalls

There are some pitfalls to using the async and await features.

If you've used Tasks, you are probably used to them throwing the exception that actually occurred instead of an exception wrapper. Promises throw a wrapper exception (`UnhandledException`) because the promise can be rejected with *any* value, not just an exception. It also contains the full causality trace. At least you can still catch `OperationCanceledException` to catch cancelations the same way as Tasks (and, unlike with the normal `Promise.Then` API where catching cancelations is optional, you probably *should* catch cancelation exceptions in an async function).

The other thing, and this is more of an issue than exception handling, is that you can't report progress to the promise returned from the async function. This is probably why the designers of Tasks chose to use `Progress<T>` passed into functions instead of implementing it directly into Tasks. Therefore, if you use async Promise functions, I recommend you disable promise progress (See Compiler Options).

Additional Information

Almost all promise methods are asynchronous. This means that calling `promise.Then` will never invoke the delegate before the method returns, even if the promise is already settled. The same goes for `deferred.Resolve/Reject/Cancel/ReportProgress` . Invoking those methods will not call attached progress or resolve/reject/cancel listeners before the method returns. Callbacks will be invoked later, the next time `Promise.Manager.HandleCompletes(AndProgress)` is called, which happens automatically every frame if you're in Unity.

The only exception to this is `Promise.New` , which invokes the delegate synchronously.

Understanding Then

There are 144 overloads for the `Then` method (72 for `Promise` and another 72 for `Promise<T>`). Rather than trying to remember all 144 overloads, it's easier to remember these rules:

- `Then` must always be given at least 1 delegate.
- The first delegate is `onResolved` .
- `onResolved` will be invoked if the promise is resolved.
- If the promise provides a value (`Promise<T>`), onResolved may take that value as an argument.
- If a capture value is provided to `onResolved` , the capture value must be the first argument to `Then` and the first argument to `onResolved`
- A second delegate is optional. If it is provided, it is `onRejected` .
- If `onRejected` does not accept any arguments, it will be invoked if the promise is rejected for any reason.
- If `onRejected` accepts an argument without a capture value, it will be invoked if the promise is rejected with a reason that is convertible to that argument's type.
- If a capture value is provided to `onRejected` , it must come after `onResolved` and before `onRejected` in the `Then` arguments, and it must be the first argument to `onRejected` .
- If a capture value is provided to `onRejected` and that is the only argument `onRejected` accepts, it will be invoked if the promise is rejected for any reason.
- If a capture value is provided to `onRejected` and `onRejected` accepts another argument, it will be invoked if the promise is rejected with a reason that is convertible to the second argument's type.
- If `onResolved` does not return a value, or it returns a non-value `Promise` :
    - the returned promise will be a non-value `Promise` .
    - `onRejected` must not return a value, or it must return a non-value `Promise` .
- If `onResolved` returns a value, or it returns a `Promise<T>` :
    - the returned promise will be a `Promise<T>` of the type of that value (or the same type of promise).
    - `onRejected` must return a value of the same type, or a `Promise<T>` of the same type.
- If either `onResolved` or `onRejected` return a promise, the promise returned from `Then` will adopt the state of that promise (waits until it completes).
- If either `onResolved` or `onRejected` throws an `Exception` , the returned promise will be rejected with that exception, unless that exception is one of the Special Exceptions.
- You may optionally provide a `CancelationToken` as the last parameter.
    - If the token is canceled while the promise is pending, the callback(s) will not be invoked, and the returned promise will be canceled with the token's reason.

You may realize that `Catch(onRejected)` also works just like `onRejected` in `Then` . There is, however, one key difference: with `Then(onResolved, onRejected)` , only one of the callbacks will ever be invoked. With `Then(onResolved).Catch(onRejected)` , both callbacks can be invoked if `onResolved` throws an exception.

Finally

`Finally` adds an `onFinally` delegate that will be invoked when the promise is resolved, rejected, or canceled. If the promise is rejected, that rejection will *not* be handled by the finally callback. That way it works just like finally clauses in normal synchronous code. `Finally` , therefore, should be used to clean up resources, like `IDisposable` s.

ContinueWith

`ContinueWith` adds an `onContinue` delegate that will be invoked when the promise is resolved, rejected, or canceled. A `Promise.ResultContainer` or `Promise<T>.ResultContainer` will be passed into the delegate that can be used to check the promise's state and result or reject/cancel reason. The promise returned from `ContinueWith` will be resolved/rejected/canceled with the same rules as `Then` in Understanding Then. `Promise.Rethrow` is an invalid operation during an `onContinue` invocation, instead you can use `resultContainer.RethrowIfRejected()` and `resultContainer.RethrowIfCanceled()`

Task Interoperability

Promises can easily interoperate with Tasks simply by calling the `Promise.ToTask()` or `Task.ToPromise()` extension methods.

Unity Yield Instructions and Coroutines Interoperability

If you are using coroutines, you can easily convert a promise to a yield instruction via `promise.ToYieldInstruction()` which you can yield return to wait until the promise has settled. You can also convert any yield instruction (including coroutines themselves) to a promise via `PromiseYielder.WaitFor(yieldInstruction)`. This will wait until the yieldInstruction has completed and provide the same instruction to an onResolved callback.

```
public Promise<Texture2D> DownloadTexture(string url)
{
    var www = UnityWebRequestTexture.GetTexture(url);
    return PromiseYielder.WaitFor(www.SendWebRequest())
        .Then(asyncOperation =>
        {
            if (asyncOperation.webRequest.isHttpError || asyncOperation.webRequest.isNetworkError)
            {
                throw Promise.RejectException(asyncOperation.webRequest.error);
            }
            return ((DownloadHandlerTexture) asyncOperation.webRequest.downloadHandler).texture;
        })
        .Finally(www.Dispose);
}
```

```
IEnumerator GetAndAssignTexture(Image image, string url)
{
    using (var textureYieldInstruction = DownloadTexture(url).ToYieldInstruction())
    {
        yield return textureYieldInstruction;
        Texture2D texture = textureYieldInstruction.GetResult();
        Sprite sprite = Sprite.Create(texture, new Rect(0, 0, texture.width, texture.height), new Vector2(0.5f, 0.5f));
        image.sprite = sprite;
    }
}
```