# Understanding JavaScript Promises

A promise represents the eventual result of an asynchronous operation. It is a placeholder into which the successful result value or reason for failure will materialize.

## Why Use Promises?

Promises provide a simpler alternative for executing, composing, and managing asynchronous operations when compared to traditional callback-based approaches. They also allow you to handle asynchronous errors using approaches that are similar to synchronous `try/catch`.

## Promise States

A promise can be in one of 3 states:

- Pending - the promise's outcome hasn't yet been determined, because the asynchronous operation that will produce its result hasn't completed yet.
- Fulfilled - the asynchronous operation has completed, and the promise has a value.
- Rejected - the asynchronous operation failed, and the promise will never be fulfilled. In the rejected state, a promise has a *reason* that indicates why the operation failed.

When a promise is pending, it can transition to the fulfilled or rejected state. Once a promise is fulfilled or rejected, however, it will never transition to any other state, and its value or failure reason will not change.

Promises have been implemented in many languages, and while promise APIs differ from language to language, JavaScript promises have converged to the Promises/A+ proposed standard. EcmaScript 6 is slated to provide promises as a first-class language feature, and they will be based on the Promises/A+ proposal.

## Using Promises

The primary API for a promise is its `then` method, which registers callbacks to receive either the eventual value or the reason why the promise cannot be fulfilled. Here is a simple “hello world” program that synchronously obtains and logs a greeting.

```
var greeting = sayHello();
console.log(greeting);    // 'hello world'
```

However, if `sayHello` is asynchronous and needs to look up the current greeting from a web service, it may return a promise:

```
var greetingPromise = sayHello();
greetingPromise.then(function (greeting) {
    console.log(greeting);    // 'hello world'
});
```

The same message is printed to the console, but now other code can continue while the greeting is being fetched.

As mentioned above, a promise can also represent a failure. If the network goes down and the greeting can’t be fetched from the web service, you can register to handle the failure using the second argument to the promise’s `then` method:

```
var greetingPromise = sayHello();
greetingPromise.then(function (greeting) {
    console.log(greeting);    // 'hello world'
}, function (error) {
    console.error('uh oh: ', error);   // 'uh oh: somethir
});
```

If `sayHello` succeeds, the greeting will be logged, but if it fails, then the reason, i.e. error, will be logged using `console.error`.

## Transforming Future Values

One powerful aspect of promises is allowing you to transform future values by returning a new value from callback function passed to `then`. For example:

```
var greetingPromise = sayHello();
greetingPromise.then(function (greeting) {
    return greeting + '!!!!';
}).then(function (greeting) {
    console.log(greeting);    // 'hello world!!!!'
});
```

## Sequencing Asynchronous Operations

A function passed to `then` can also return another promise. This allows asynchronous operations to be chained together, so that they are guaranteed to happen in the correct order. For example, if `addExclamation` is asynchronous (possibly needing to access another web service) and returns a promise for the new greeting:

```
var greetingPromise = sayHello();
greetingPromise.then(function (greeting) {
    return addExclamation(greeting); // addExclamation ret
}).then(function (greeting) {
    console.log(greeting);    // 'hello world!!!!'
});
```

This can be written more simply as:

```
var greetingPromise = sayHello();
greetingPromise
    .then(addExclamation)
    .then(function (greeting) {
        console.log(greeting);    // 'hello world!!!!'
    });
```

## Handling Errors

What if an error occurs while performing an asynchronous operation? For example, what if `sayHello`, or `addExclamation` fails? In synchronous code, you can use `try/catch`, and rely on exception propagation to handle errors in one spot. Here is a synchronous version of the previous example that includes `try/catch` error handling. If an error occurs in either `sayHello` or `addExclamation` (or `console.log` for that matter), the `catch` block will be executed:

```
var greeting;
try {
    greeting = sayHello();
    greeting = addExclamation(greeting);
    console.log(greeting);    // 'hello world!!!!'
} catch(error) {
    console.error('uh oh: ', error);   // 'uh oh: somethir
}
```

When dealing with asynchronous operations, synchronous `try/catch` can no longer be used. However, promises allow handling asynchronous errors in a very similar way. This allows you not only to write asynchronous code that is similar in style to synchronous code, but also to reason about asynchronous control flow and error handling similarly to synchronous code.

Here is an asynchronous version that handles errors in the same way:

```
var greetingPromise = sayHello();
greetingPromise
    .then(addExclamation)
    .then(function (greeting) {
        console.log(greeting);    // 'hello world!!!!'
    }, function(error) {
        console.error('uh oh: ', error);   // 'uh oh: some
    });
```

Notice that just like the synchronous example, you can use a single error handling block, in this case passed as the second parameter to the final call to `then`.