Promises Lesson Learning Objectives I

Below is a complete list of the terminal learning objectives for this lesson. When you complete this lesson, you should be able to perform each of the following objectives. These objectives capture how you may be evaluated on the assessment for this lesson.

1. Instantiate a `Promise`object
2. Use `Promises`to write more maintainable asynchronous code
3. Use the `fetch`API to make `Promise`-based API calls

A Promise is a Promise: A Mostly Complete Guide to JavaScript Promises I

This article is about a JavaScript feature formally introduced into the language in 2015: the `Promise`object. The technical committee that governs the JavaScript language recognized that programmers had a hard time reasoning about and maintaining asynchronous code. They included `Promise`s as a way to encourage writing asynchronous code in a way that *appeared*synchronous.

When you finish this article, you should be able to:

- Provide examples of why `Promise`-based code is easier to maintain than traditional asynchronous callbacks;
- Recall the three states of a `Promise`, what each state means, and any associated data with that state.

A quick review of function declarations

It's important to remember about how JavaScript handles the declaration of a function. Please look at the following code.

```
function loudLog(message) {
  console.log(message.toUpperCase());
}
```

When JavaScript encounters that code, it does not run the function. You probably know that, but it's important to read again. When JavaScript encounters that code, it does not run the function.

It *does*create a `Function`object and stores that in a variable named`loudLog`. At some time later, you can run the function object in that variable with the syntax `loudLog("error occurred");`. That *runs* the function. Just declaring a function doesn't run it. Look at this following code.

```
function () {
  console.log('How did you call me?');
}
```

JavaScript will, again, create a `Function`object. However, there's no name for the function, so it doesn't get assigned to any variable, and just disappears with no way for us to use it. So, why would you declare functions without names?

The looming problem of asynchronous code with callbacks

Let's look at the documentation for how to read files in Node.js. Don't worry if you haven't used Node.js, yet. It's just like any other JavaScript.

```
readFile(path, encoding, callback)

Arguments:
  path      <string>    path to the file
  encoding  <string>    the encoding of the file
  callback  <function>  two arguments:
                        err      <error object>
                        content  <string>

Asynchronously reads the entire contents of a file.
```

The function named `readFile`accepts two arguments, a string that contains the`path` to the file and a function that `readFile`calls once it's read the content of the file. If you wanted to write out the content of the file with a header, you could write code like this.

```
function writeWithHeader(err, content) {
  console.log("YOUR FILE CONTAINS:");
  console.log(content);
}

readFile('~/Documents/todos.txt', 'utf8', writeWithHeader);
```

Recall that when JavaScript found the function declaration at the beginning of that code block, it created a `Function`object and stored it in a variable named `writeWithHeader`. Now, that variable contains the actual function that can later be run. That code passes the value of that variable, the `Function`object, into the `readFile`function so the `readFile`function can run it later.

If you're not going to use the `writeWithHeader`function anywhere else in your code, idiomatic JavaScript instructs you to get rid of the name of the function and declare it directly as the second argument of the `readFile`functions. That would turn the above code block into the following.

```
readFile('~/Documents/todos.txt', 'utf8', function (err, content) {
  console.log("YOUR FILE CONTAINS:");
  console.log(content);
});
```

Since 2015, idiomatic JavaScript would instruct you to get rid of the function keyword and just use an arrow function.

```
readFile('~/Documents/todos.txt', 'utf8', (err, content) => {
  console.log("YOUR FILE CONTAINS:");
  console.log(content);
});
```

The key to remember here is that you have only declared that function that`readFile`will call later, `readFile`is in charge of running that function.

Imagine that you have a file that has a list of other file names in it named`manifest.txt`. You want to read the file and read each of the files listed in it. Then, you want to count the characters in each of those files and print those numbers.

You would start out by reading `manifest.txt`and splitting the content on the newline character to get the names of the files. That would look like this:

```
readFile('manifest.txt', 'utf8', (err, manifest) => {
  const fileNames = manifest.split('\n');

  // More to come
});
```

Now that you have the list of file names, you can loop over them to read each of those files. As each of those files are read, you want to count the characters in each file. Imagine that you already have the function named `countCharacters`somewhere. The looping code could look like this:

```
readFile('manifest.txt', 'utf8', (err, manifest) => {
  const fileNames = manifest.split('\n');
  const characterCounts = {};

  // Loop over each file name
  for (let fileName of fileNames) {
    // Read that file's content
    readFile(fileName, 'utf8', (err, content) => {
      // Count the characters and store it in
      // characterCounts
      countCharacters(characterCounts, content);
    });
  }
});
```

At this point, you feel pretty good. There's only one thing left to do: print out the total of all the characters in the files. So, where do you put that `console.log`statement?

This is kind of a trick question because there's no place to put it in the way the code works now.

If you put it here:

```
readFile('manifest.txt', 'utf8', (err, manifest) => {
  const fileNames = manifest.split('\n');
  const characterCounts = {};

  // Loop over each file name
  for (let fileName of fileNames) {
    // Read that file's content
    readFile(fileName, 'utf8', (err, content) => {
      // Count the characters and store it in
      // characterCounts
      countCharacters(characterCounts, content);
    });
  }

  // MY PRINT STATEMENT HERE
  console.log(characterCounts);
});
```

then you will get the output `{}`every time because the code in the inner`readFile`s doesn't run until after the `console.log`because `readFile`doesn't run the function with the arguments `(err, content)`until *after* the file is read and the current function completes.

If you put it here:

```
readFile('manifest.txt', 'utf8', (err, manifest) => {
  const fileNames = manifest.split('\n');
  const characterCounts = {};

  // Loop over each file name
  for (let fileName of fileNames) {
    // Read that file's content
    readFile(fileName, 'utf8', (err, content) => {
      // Count the characters and store it in
      // characterCounts
      countCharacters(characterCounts, content);

      // MY PRINT STATEMENT HERE
      console.log(characterCounts);
    });
  }
});
```

then it will print the number of times that your code reads a file. That's not what you want, either. To get it to work, you have to count the number of files that have been read each time one completes. Then, you only print when that number equals the total number of files to be read. The code could like this:

```
readFile('manifest.txt', 'utf8', (err, manifest) => {
  const fileNames = manifest.split('\n');
  const characterCounts = {};
  let numberOfFilesRead = 0;

  // Loop over each file name
  for (let fileName of fileNames) {
    // Read that file's content
    readFile(fileName, 'utf8', (err, content) => {
      // Count the characters and store it in
      // characterCounts
      countCharacters(characterCounts, content);

      // Increment the number of files read
      numberOfFilesRead += 1;
      // If the number of files read is equal to the
      // number of files to read, then print because
      // you're done!
      if (numberOfFilesRead === fileNames.length) {
        console.log(characterCounts);
      }
    });
  }
});
```

The asynchronous nature of this code requires you to do a lot of housekeeping just to figure out when everything is done. Imagine writing this code and going back to it in six months to add a new feature. It's not the clearest code in the world, even with code comments. That leads to a maintenance nightmare. The JavaScript community wanted a way to code better and clearer.

Designing a better solution

Look at the following code that has numbers in the order in which the`console.log`statements are run. It will print out "Q", "W", "E", "R", and "T" on separate lines.†

```
console.log('Q'); //---- 1
setTimeout(() => {
  console.log('E'); //-- 3
  setTimeout(() => {
    console.log('T'); // 5
  }, 100);
  console.log('R'); //-- 4
}, 200);
console.log('W'); //---- 2
```

What would really help is if you could get those numbers in order so that what appears in the code at least **appears**to be synchronous even though it might be asynchronous in nature. As humans, we understand things from top-to-bottom much better than in the order 1, 3, 5, 4, 2.

Reordering the code above to reflect how it really runs, you'd get this somewhat more maintainable block.

```
console.log('Q'); //---- 1
console.log('W'); //---- 2
setTimeout(() => {
  console.log('E'); //-- 3
  console.log('R'); //-- 4
  setTimeout(() => {
    console.log('T'); // 5
  }, 100);
}, 200);
```

But, now you're stuck with those human-necessary indents to understand the function calls that occur in the code. And, to know how long the `setTimeout`s run, you have to go way to the bottom of the code blocks. The JavaScript community agreed with you and decided it'd be great if they could somehow just chain a bunch of those things together without the indentation, something like this. (The function names are completely invented for this code block.);

```
log('Q')
  .then(() => log('W'))
  .then(() => pause(200))
  .then(() => log('E'))
  .then(() => log('R'))
  .then(() => pause(100))
  .then(() => log('T'));
```

The JavaScript community realized that they'd have to use functions in the`then`blocks lest the function be immediately invoked. Remember, a function declaration is not invoked when interpreted. That means each function in each of the `then`calls is passed into the `then`function for it to run at a later time, presumably when the previous thing finishes, a previous `log` or `pause`in this example. They decided to create a new kind of abstraction in JavaScript named the "Promise".

So, what is a "Promise"?

Look at a line of code using the `readFile`method found in Node.js. Don't worry if you don't know the specifics about this function. It's the *form* of the code to which you should draw your attention.

```
readFile('manifest.txt', 'utf8', (err, manifest) => {
```

You could interpret that line of code as "Read the file named "manifest.txt" and, when done, call the method that is declared with `(err, manifest) => {`.

The important part to understand is the "when done, call the method...". That's the part that's potentially asynchronous, the part that is beyond your control. When it calls that function, it will either provide an error in the `err`parameter or a value in the `manifest`parameter. When you change it to the`then`version, you still get the same kind of guarantee: eventually, you will get an error or the value of the operation. So that's what a `Promise`is.

*A `Promise`in JavaScript is a commitment that sometime in the future, your code will get **a value**from some operation (like reading a file or getting JSON from a Web site) or your code will get **an error**from that operation (like the file doesn't exist or the Web site is down).*

Promises can exist in three states. They are:

- **Pending**: The `Promise`object has not resolved. Once it does, the state of the `Promise`object may transition to either the fulfilled or rejected state.
- **Fulfilled**: Whatever operation the `Promise`represented succeeded and your success handler will get called. Now that it's *fulfilled*, the `Promise`:
  - must not transition to any other state.
  - must have a value, which must not change.
- **Rejected**: Whatever operation the `Promise`represented failed and your error handler will get called. Now that it's *rejected*, the `Promise`:
  - must not transition to any other state.
  - must have a reason, which must not change.

`Promise`objects have the following methods available on them so that you can handle the state change from *pending*to either *fulfilled*or *rejected*.

- `then(successHandler, errorHandler)`is a way to handle a `Promise`when it leaves the *pending*state.
- `catch(errorHandler)`

The handlers mentioned in the previous list are:

- **Success Handler**is a function that has one parameter, the value that a*fulfilled*`Promise`has.
- **Error Handler**is a function that has one parameter, the reason that the`Promise`failed.

We'll elaborate on these methods in part two of this article.

What you've learned

In this reading, you learned some fancy new things that let's you turn asynchronous code into seemingly synchronous-looking code. You did that by learning that...

- A `Promise`in JavaScript is a commitment that sometime in the future, your code will get **a value**from some operation (like reading a file or getting JSON from a Web site) or your code will get **an error**from that operation (like the file doesn't exist or the Web site is down).

†: One can argue that the code following this statement is already very bad and shouldn't be written that way. I would agree. Please don't write code like that. It is *only* for demonstration purposes. However, do not be surprised if you find **someone else**wrote code like that. ;-)

A Promise is a Promise: A Mostly Complete Guide to JavaScript Promises II

This is part two of an article about classic JavaScript promises. If you have not read part one, we recommend that you navigate to the previous task to do so.

When you finish this article, you should be able to:

- Create your own `Promise`s
- Use `Promise`objects returned by language and framework libraries

Handling success with `then`

Returning to another file-reading example, consider the following block of code.

```
readFile("manifest.txt", "utf8", (err, manifest) => {
  if (err) {
    console.error("Badness happened", err);
  } else {
    const fileList = manifest.split("\n");
    console.log("Reading", fileList.length, "files");
  }
});
```

If this succeeds, then you would expect a statement like "Reading 12 files" to appear if the file contained a list of 12 files.

Now, to rewrite that using a `Promise`and printing that same statement, you would get a file-reading function that returns a `Promise`object. Later on, you'll see how to create one for yourself. At this moment, just presume that a function named `readFilePromise`exists. When you call it, it would return a promise that, when *fulfilled*, would invoke the success handler registered for the object through the `then`method. Very explicitly, you could write that code like this.

```
/* EXPLICIT CODE: NOT FOR REAL USE */

// Declare a function that will handle the content of
// the file read by readFilePromise.
function readFileSuccessHandler(manifest) {
  const fileList = manifest.split("\n");
  console.log("Reading", fileList.length, "files");
}

// Get a promise that will return the contents of the
// file.
const filePromise = readFilePromise("manifest.txt");

// Register a success handler to process the contents
// of the file. In this case, it is the function
// defined above.
filePromise.then(readFileSuccessHandler);
```

Most `Promise`-based code does **not**look like that, though. Idiomatic JavaScript instructs to not create variables that don't need to be created. You would see the above code in a real-live code base written like this, instead. Spend a moment comparing and contrasting the forms from **very explicit**to**idiomatic**.

```
readFilePromise("manifest.txt").then(manifest => {
  const fileList = manifest.split("\n");
  console.log("Reading", fileList.length, "files");
});
```

That's slightly easier to read than the weird callback thing you had above. But, you still have that nasty double indentation. The designers of the `Promise` didn't want that for you, so they allow you to chain `then`s.

Chaining `then`s.

In the above code that uses `readFilePromise`, it does not look like the ideal code that JavaScript could give us because of the success-handling function being on multiple lines that require another indent. It may be a little thing, but it still prevents you from the most readable code. Again, the Technical Committee 39 had your back. They designed "chainable thens" for you. The rules are a little complex to read.

- Each `Promise` has a `then` method that handles what happens when the `Promise` transitions out of the **pending** state.
- Each `then` method returns a `Promise` that transitions out of its **pending** state when the `then` that created it completes.
- (One more condition described below.)

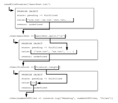
That chaining property gives you the ability to break apart the two lines of the success handler in the previous example to two one-line functions that do the same thing with less code! If you write that form explicitly, you'd have the following.

```
/* EXPLICIT CODE: NOT FOR REAL USE */

// Get a Promise that fulfills when the file is read
// with the value of the content of the file.
const filePromise = readFilePromise("manifest.txt");

// Register a success handler that takes the fulfilled
// value of the filePromise in the parameter named "manifest",
// which is the content of the file, split it on newline
// characters, and return a Promise whose fulfilled value is
// list of lines.
const fileListPromise = filePromise.then(manifest => manifest.split("\n"));

// Register a success handler to the fileListPromise that
// receives the fulfilled value in the "fileList" parameter
// and returns a Promise whose fulfilled value is the length
// of the fileList array.
const lengthPromise = fileListPromise.then(fileList => fileList.length);

// Register a success handler to the lengthPromise that
// receives the fulfilled value in the "numberOfFiles" parameter
// and uses it to print the number of files to be read.
lengthPromise.then(numberOfFiles =>
  console.log("Reading", numberOfFiles, "files")
);
```

That code block has a lot of words to describe what happens at each step of the process of using "chainable thens". In the real world, were you to find that code in a real application, it would likely look like the following.

```
readFilePromise("manifest.txt")
  .then(manifest => manifest.split("\n"))
  .then(fileList => fileList.length)
  .then(numberOfFiles => console.log("Reading", numberOfFiles, "files"));
```

Here's a diagram of what happens in the above code.



You can see that each call to `then` creates a new `Promise` object that resolves to the value of the output of the previous success handler. That's what happens when everything works out. What happens when it doesn't?

Handling failure with `then`

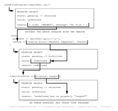
As you may recall from the sectionSo, what is a "Promise"?, you learned that the `then`method can also accept a second argument that is an error handler that takes care of things should something go wrong. Back to the file reading example from above, you add a second method to the `then`which accepts a **reason**that the error happened. For reading a file, that could be that the file doesn't exist, the current user doesn't have permissions to read it, or it ran out of memory trying to read a *huge* file.

```
readFilePromise("manifest.txt").then(
  manifest => {
    const fileList = manifest.split("\n");
    console.log("Reading", fileList.length, "files");
  },
  reason => {
    console.error("Badness happened", reason);
  }
);
```

That works, but has taken you back to the original bad multiline form of the success handler. What happens if you did it like this? How does this work?

```
readFilePromise("manifest.txt")
  .then(
    manifest => manifest.split("\n"),
    reason => console.err("Badness happened", reason)
  )
  .then(fileList => fileList.length)
  .then(numberOfFiles => console.log("Reading", numberOfFiles, "files"));
```

Here's what happens with regard to the `Promise`s in this chain of `then`s.


readFilePromise("manifest.txt")
PROMISE OBJECT
state: pending -> rejected
value: undefined
reason: { code: 'ENOENT', message: 'No file.' }
INVOKES THE ERROR HANDLER WITH THE REASON
.then(
manifest => manifest.split('\n'),
reason => console.error('Badness happened', reason)
)
PROMISE OBJECT
state: pending -> fulfilled
value: undefined
reason: undefined
.then(fileList => fileList.length)
PROMISE OBJECT
state: pending -> rejected
value: undefined
reason: 'undefined has no property "length"'
NO ERROR HANDLER! MAY CRASH YOUR PROGRAM!
.then(numberOfFiles => console.log('Reading', numberOfFiles, 'files'))

As you can see, the first `Promise`object from the `readFilePromise`function goes into the **rejected**state because, according to the error message, the file didn't exist at the time the system tried to read it. That reason is represented as an object that has a code of "ENOENT" which a Unix error code and a message that provides a human-readable explanation of the error. That error reason object gets passed to the error handler of the first `then`. It's what happens after that that is crazy neat.

The second `Promise`object is **fulfilled**! Because the first `then`doesn't have any errors, because the error handler in the first then completes without any problem (printing out the error reason), the `Promise`returned by that`then`*does not*get **rejected**. Because of that, the `Promise`resolves with the value returned by the `console.error('Badness happened', err)`call. The`console.error`method returns `undefined`, so that becomes the value passed into the next `then`handler.

Because the second `then`success handler relies on an object with a `length`property, when it runs, an exception gets raised because the `undefined`value has no `length`property. This causes the `Promise`returned by the second`then` to become **rejected**because the code threw an exception.

Because that `Promise`is in the **rejected**state, it attempts to run the error handler of the next (third) `then`. There is no error handler. In the browser, it just looks like nothing happened. In Node.js, an`UnhandledPromiseRejectionWarning`is emitted to the console. In a future version of Node.js, it will cause the process to terminate with an exit code indicating an error bringing your service to a halt.

To correctly handle the exception of no file to read and still have all of the other lines of code run properly, you should write the following code.

```
readFilePromise("manifest.txt")
  .then(manifest => manifest.split("\n"))
  .then(fileList => fileList.length)
  .then(
    numberOfFiles => console.log("Reading", numberOfFiles, "files"),
    reason => console.err("Badness happened", reason)
  );
```

Now, if an error occurs, the chain of `then`s evaluates like this:

1. First `then`: I do not have an error handler. I will pass the error on and not run the success handler.
2. Second `then`: I do not have an error handler. I will pass the error on and not run the success handler.
3. Third `then`: I have an error handler and will run it.

Now, the code looks almost like you'd imagined back in theDesigning a better solutionsection. There's still that annoying last double handler code that makes us use indentation and passing in two function objects to a `then`which looks kind of yucky. The Technical Committee gave you a solution for that, too.

*`then`can handle both success and failures. The success handler is called with the value of the operation of the `Promise`when the `Promise`object transitions to the **fulfilled**state. If an error condition occurs, them the error handler of the `then` is called.*

*If a `Promise`object transitions to the **rejected**state and no error handler exists for the `then`, then that `then` is skipped altogether.*

*If an error handler is called and does not raise an exception, then the next`Promise`object transitions to the **fulfilled**state and the next success handler is called.*

Handling failure with `catch`

Rather than using a `then`with a success and error handler, you can use the similar `catch`method that takes just an error handler. By doing that, the code from the last section ends up looking like this.

```
readFilePromise("manifest.txt")
  .then(manifest => manifest.split("\n"))
  .then(fileList => fileList.length)
  .then(numberOfFiles => console.log("Reading", numberOfFiles, "files"))
  .catch(reason => console.err("Badness happened", reason));
```

That is exactly what the design expressed. The `catch`acts just like an error handler in the last `then`. If the `catch`doesn't throw an exception, then it returns a `Promise`in a fulfilled state with whatever the return value is, just like the error handler of a `then`.

`catch`is a convenient way to do error handling in a `then`chain that looks kind of like part of a try/catch block.

Using `Promise.all`for many future values

You're almost to the place where you can read the manifest file, get the list, and then count the characters in each of the files, and print out the result. You need to learn about two more features of JavaScript `Promise`s.

Imagine that you have three files that you want to read with the`readFilePromise`method. You want to wait until all three are done, but let them read files simultaneously. How do you manage all three `Promise`s as one`Promise`? That's what the `Promise.all`method allows you to do.

For example, imagine you have the following array.

```
const values = [
  readFilePromise("file-boop.txt"), // this is a Promise object: pending
  readFilePromise("file-doop.txt"), // this is a Promise object: pending
  readFilePromise("file-goop.txt"), // this is a Promise object: pending
];
```

When you pass that array into `Promise.all`, it returns a `Promise`object that manages all of the `Promise`s in the array!

```
const superPromise = Promise.all(values);
// superPromise is a Promise object in the pending state.
//
// Inside superPromise is an array of Promise objects
// that look like this:
//
// 1. file reading promise in pending state, same as the one passed in
// 2. file reading promise in pending state, same as the one passed in
// 3. file reading promise in pending state, same as the one passed in
```

When all of the `Promise`objects in the super `Promise`transition out of the pending state, then the super `Promise`will also transition out of the pending state. If any one of the `Promise`objects in the array transition to the**rejected**state, then the super `Promise`will immediately transition to the**rejected**state with the same reason as the inner `Promise`failed with. If*all* of the internal `Promise`objects transition to the **fulfilled**state, then the super `Promise`will transition to the **fulfilled**state and its value will be an array of *all* of the resolved values of the original array.

With that in mind, you could continue the above code with a `then`and a `catch`that would demonstrate what happens.

```
superPromise
  .then(values => console.log(values))
  .catch(reason => console.error(reason));
```

```
// If the function successfully reads the file, the values passed
// to the then come from the values that were in the superPromise
//
// 1. the content of file-boop.txt
// 2. the content of file-doop.txt
// 3. the content of file-goop.txt

// If something goes wrong with reading the file, then the `catch`
// gets called with the error reason from the Promise object that
// first failed.
```

*`Promise.all`accepts an array of values and returns a new `Promise`object in the **pending**state colloquially called a "super promise". It converts all non-`Promise`values into `Promise`objects that are immediately in the**fulfilled**state. Then,*

- *If any one of the `Promise`s in the array transitions to the **rejected**state, then the "super promise" transitions to the **rejected**state with the same reason that the inner `Promise`object failed.*
- *If all of the inner `Promise`objects in the array transition to the**fulfilled**state, then the "super promise" transitions to the**fulfilled**state with a value of an array populated, in order, of the resolved values of the original array.*

Flattening `Promises`

The last thing you need to learn about `Promise`s is the coolest feature of them all. If you return a `Promise`object from either a success or error handler, the next step doesn't get run until that `Promise`object resolves! Here's what happens when you type the following code. It's step 4 that is the amazing part.

```
readFilePromise("manifest.txt")
  .then(manifestContent => manifestContent.split("\n"))
  .then(manifestList => manifestList[0])
  .then(fileName => readFilePromise(fileName))
  .then(otherFileContent => console.log(otherFileContent));

// Interpreted as:
// 1. Read the file of the manifest.txt file and pass the
//    content to the first then.
// 2. Split the content from manifest.txt on newline chars
//    to get the full list of files.
// 3. Return just the first entry in the list of files.
// 4. RETURN A PROMISE THAT WILL READ THE FILE NAMED ON THE
//    FIRST LINE OF THE manifest.txt! The next then method
//    doesn't get called until this Promise object completes!
// 5. Get the content of the file just read and print it.
```

Again, here's the rule.

If you return a `Promise`from a success or error handler, the next handler isn't called until that `Promise`completes.

Putting it all together

You can now use all of this knowledge to use `Promise`s to read a manifest file, read each of the files in the manifest files, and count all of the characters in those files with code that reads much better than this.

```
readFile("manifest.txt", "utf8", (err, manifest) => {
  const fileNames = manifest.split("\n");
  const characterCounts = {};
  let numberOfFilesRead = 0;

  // Loop over each file name
  for (let fileName of fileNames) {
    // Read that file's content
    readFile(fileName, "utf8", (err, content) => {
      // Count the characters and store it in
      // characterCounts
      countCharacters(characterCounts, content);

      // Increment the number of files read
      numberOfFilesRead += 1;

      // If the number of files read is equal to the
      // number of files to read, then print because
      // we're done!
      if (numberOfFilesRead === fileNames.length) {
        console.log(characterCounts);
      }
    });
  }
});
```

Remember that you've created a `countCharacters`methods elsewhere that does the grunt work of counting characters. So, now, if you were to list out the steps that you'd like to have the code perform, you should be able to write a`Promise`-based chain of `then`s that does that work.

1. Read `manifest.txt`.
2. Split the content into a list of files.
3. Read the contents of each file.
4. If all of them succeed, then
    - count the characters in each file and
    - print the character counts.
5. If anything fails, print the error.

So, in code, that you would translate that to the following.

```
const characterCounts = {};
readFilePromise('manifest.txt')
  .then(fileContent => fileContent.split('\n'))
  .then(fileList => fileList.map(fileName => readFilePromise(fileName)))
  .then(lotsOfReadFilePromises => Promise.all(lotsOfReadFilePromises))
  .then(contentsArray => contentsArray.forEach(c => countCharacters(characterCou
  .then(() => console.log(characterCounts))
  .catch(reason => console.error(reason));
```

Through the magic of `Promise`s, you have now been able to do lots of asynchronous work but make it look synchronous!

Creating your own `Promise`s

Early on, you designed the way `Promise`s should work to look something like this.

```
log("Q")
  .then(() => log("W"))
  .then(() => pause(2))
  .then(() => log("E"))
  .then(() => log("R"))
  .then(() => pause(1))
  .then(() => log("T"));
```

That code uses two functions that you can define:

- a `log`function that takes a value to print and returns a `Promise`object that is in the **fulfilled**state; and,
- a `pause`function that takes a number and returns a `Promise`object that, after the indicated number of seconds, transitions to the **fulfilled**state.

Here is a way that you could create those functions.

```
function log(message) {
  console.log(message);
  return Promise.resolve();
}
```

The above function logs the message passed to it and, then creates a `Promise`object already transitioned to the **fulfilled**state. If you provide a value to the resolve method, then that becomes the value of the `Promise`object.†

The `pause`method is a little more difficult. You have to create a new`Promise`object from scratch to pause and then continue. To do that, you will use the `Promise`constructor.

The `Promise`constructor accepts a function that has two parameters. Each of those parameters will be functions, themselves. The first parameter is the so-called **resolve**parameter which, when called, transitions the `Promise`object to the **fulfilled**state. The second parameter is the so-called**reject**parameter which, when called, transitions the `Promise`object to the**rejected**state.

```
function pause(numberOfSeconds) {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(), numberOfSeconds * 1000);
  });
}
```

As you can see from the above code, the `new Promise`gets a single argument, a two-parameter function that does some asynchronous thing. The two parameters are the **resolve**and the **reject**functions that you can use to transition the state of the `Promise`object being constructed. In this case, after a certain amount of time, the `resolve()`method is invoked which transitions the `Promise` object to the **fulfilled** state. The value is `undefined` because you've passed no value into the `resolve()` function invocation. If you wanted the `Promise` to have the value of 6.28, then you would invoke it like this `resolve(6.28)`. You can pass any one value into the `resolve` function, be it a number, a boolean, an array, an object, or whatever.

With that knowledge, think about how you would write a function using the `readFile` function that would return a `Promise` object that would resolve to the contents of the file on success and reject the `Promise` if an error occurred. Take a moment to scratch that out into an editor or something.

If you wrote something similar to the following, then you did a great job! If you didn't, work through the following in a Node.js JavaScript environment to figure out how it works. You can use it like in any of the above examples.

```
const { readFile } = require("fs"); // This is just the way to get
// the readFile method into the
// current file. If you don't
// understand it, that's ok.

function readFilePromise(path) {
  return new Promise((resolve, reject) => {
    readFile(path, "utf8", (err, content) => {
      if (err) {
        reject(err);
      } else {
        resolve(content);
      }
    });
  });
}
```

What you've learned

In this reading, you learned some fancy new things that let's you turn asynchronous code into seemingly synchronous-looking code. You did that by learning that...

- `then` can handle both success and failures. The success handler is called with the value of the operation of the `Promise` when the `Promise` object transitions to the **fulfilled** state. If an error condition occurs, them the error handler of the `then` is called.
- If a `Promise` object transitions to the **rejected** state and no error handler exists for the `then`, then that `then` is skipped altogether.
- If an error handler is called and does not raise an exception, then the next `Promise` object transitions to the **fulfilled** state and the next success handler is called.
- `catch` is a convenient way to do error handling in a `then` chain that looks kind of like part of a try/catch block.
- `Promise.all` accepts an array of values and returns a new `Promise` object in the **pending** state colloquially called a "super promise". It converts all non-`Promise` values into `Promise` objects that are immediately in the **fulfilled** state. Then,
    - If any one of the `Promise`s in the array transitions to the **rejected** state, then the "super promise" transitions to the **rejected** state with the same reason that the inner `Promise` object failed.
    - If all of the inner `Promise` objects in the array transition to the **fulfilled** state, then the "super promise" transitions to the **fulfilled** state with a value of an array populated, in order, of the resolved values of the original array.
- If you return a `Promise` from a success or error handler, the next handler isn't called until that `Promise` completes.
- You can create a **fulfilled** `Promise` object by using the `Promise.resolve(value)` method.
- You can create your own `Promise` objects from scratch by using the `Promise` constructor with the form

  ```
  new Promise((resolve, reject) => {
    // do some async stuff
    // call resolve(value) to make the Promise succeed
  ```

```
  // call reject(reason) to make the Promise fail
});
```

See also

- Section: Promises, ECMAScript® 2015 Language Specificationis the minimum standard for how JavaScript Promises should act in **all**JavaScript environments. Language standards are dense and hard to read. You may want to give it a shot. The more you grow in your knowledge of how JavaScript works, the clearer it should become.
- The Promises/A+ Specificationhas a very nice terse description of how Promises work. It is mostly the standard that was adopted by the Technical Committee 39 when including the `Promise`object into JavaScript.

†: There's a corresponding `Promise.reject(reason)`method that creates a`Promise`object immediately in the **rejected**state.