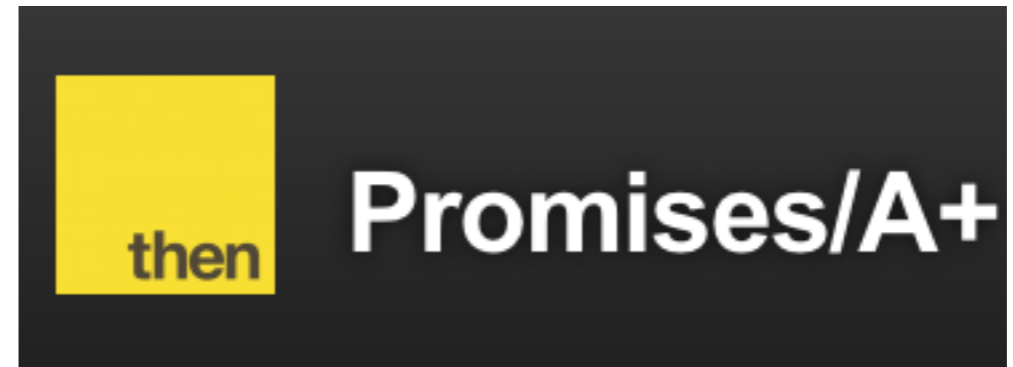
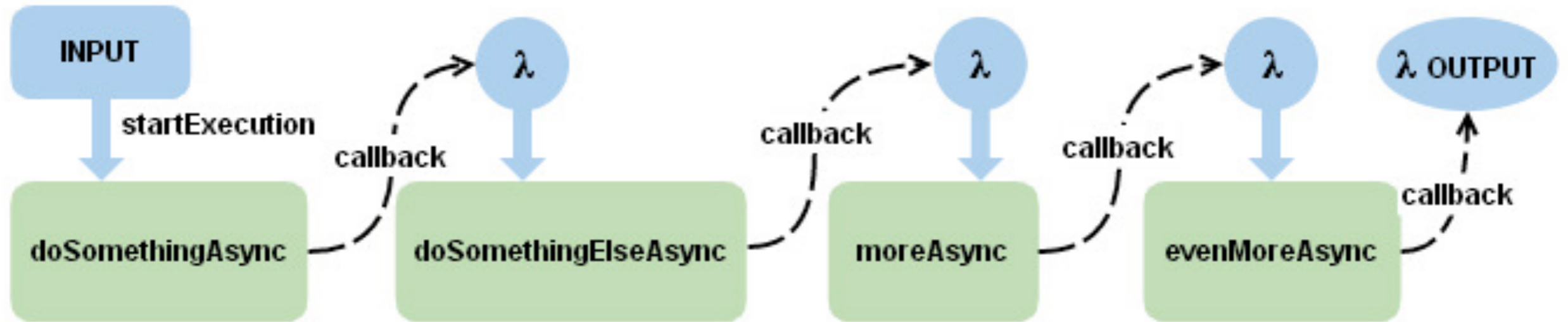


Callbacks & Promises



Agenda



- Task: read a JSON file
- A Callback Based Library (fs)
- A Promise based Library (fs-readfile-promise)

- Function Styles:

- Anonymous function
- Named function
- Function Object
- Named Arrow Function
- Anonymous Arrow Function

Task: Read a JSON File

- Read a json file into a string variable
- Parse that file into a JavaScript Object
- Print out the Javascript Object
- Deal with errors in an orderly manner:
 - File not Found
 - File not Correct JSON format

```
[
  {
    "firstName": "Homer",
    "lastName": "Simpson",
    "email": "homer@simpson.com",
    "password": "secret"
  },
  {
    "firstName": "Marge",
    "lastName": "Simpson",
    "email": "marge@simpson.com",
    "password": "secret"
  },
  {
    "firstName": "Bart",
    "lastName": "Simpson",
    "email": "bart@simpson.com",
    "password": "secret"
  }
]
```

```
obj = Array[3]
  0 = Object
    email = "homer@simpson.com"
    firstName = "Homer"
    lastName = "Simpson"
    password = "secret"
    __proto__ = Object
  1 = Object
    email = "marge@simpson.com"
    firstName = "Marge"
    lastName = "Simpson"
    password = "secret"
    __proto__ = Object
  2 = Object
    email = "bart@simpson.com"
    firstName = "Bart"
    lastName = "Simpson"
    password = "secret"
    __proto__ = Object
length = 3
```

memory

fs node module

- implicit module in node
- No need to 'npm install'
- Provides Synchronous & Asynchronous version of most functions

File System

Stability: 2 - Stable

File I/O is provided by simple wrappers around standard POSIX functions. To use this module do `require('fs')`. All the methods have asynchronous and synchronous forms.

The asynchronous form always takes a completion callback as its last argument. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be `null` or `undefined`.

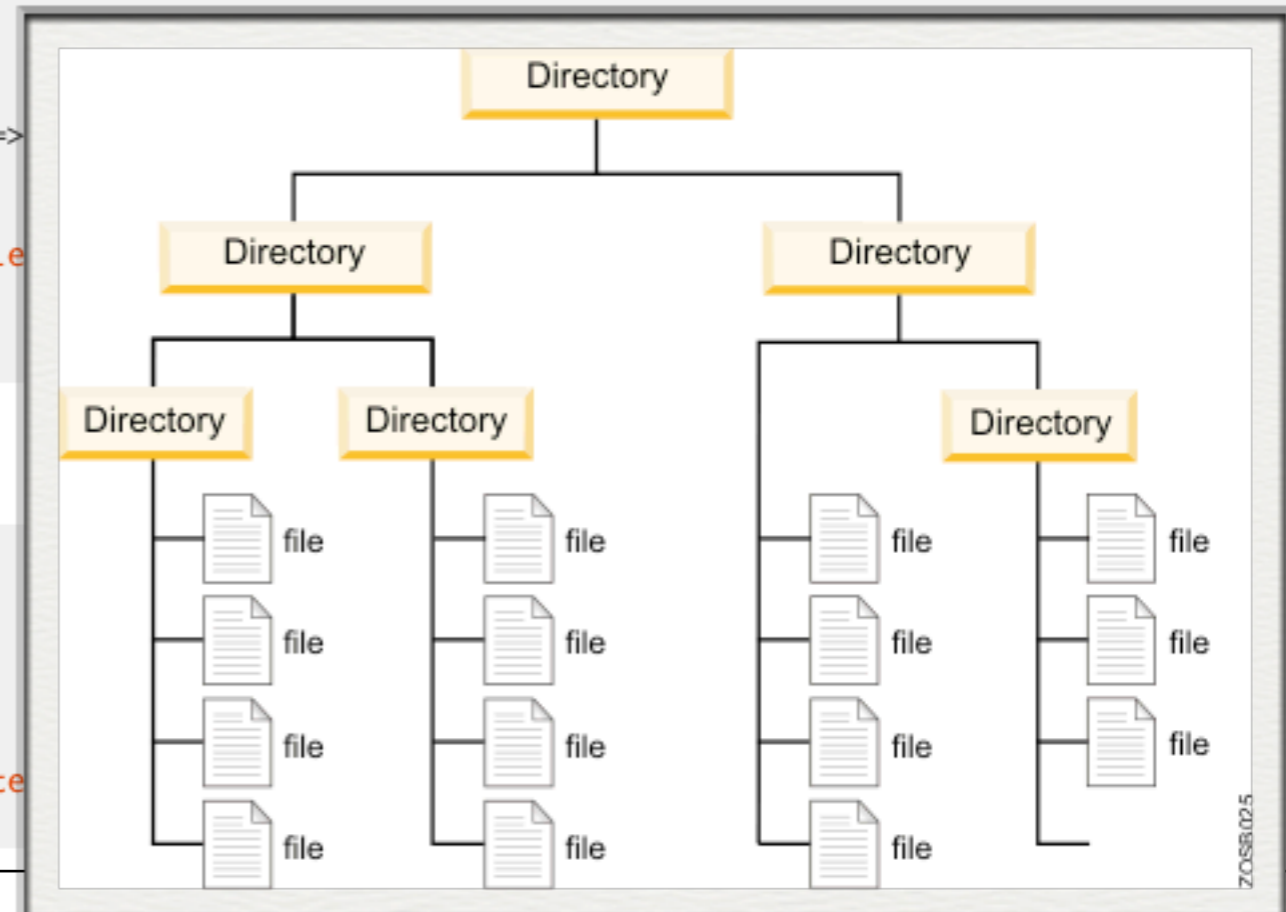
When using the synchronous form any exceptions are immediately thrown. You can use try/catch to handle exceptions or allow them to bubble up.

Here is an example of the asynchronous version:

```
const fs = require('fs');  
  
fs.unlink('/tmp/hello', (err) =>  
  if (err) throw err;  
  console.log('successfully deleted');  
});
```

Here is the synchronous version:

```
const fs = require('fs');  
  
fs.unlinkSync('/tmp/hello');  
console.log('successfully deleted');
```



Simp

fs.readFile(file[, options], callback)

Added in: v0.1.29

- `file` `<String>` | `<Buffer>` | `<Integer>` filename or file descriptor
- `options` `<Object>` | `<String>`
 - `encoding` `<String>` | `<Null>` default = `null`
 - `flag` `<String>` default = `'r'`
- `callback` `<Function>`

Asynchronously reads the entire contents of a file. Example:

```
fs.readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

The callback is passed two arguments `(err, data)`, where `data` is the contents of the file.

If no encoding is specified, then the raw buffer is returned.

If `options` is a string, then it specifies the encoding. Example:

```
fs.readFile('/etc/passwd', 'utf8', callback);
```

Any specified file descriptor has to support reading.

Note: If a file descriptor is specified as the `file`, it will not be closed automatically.

Anonymous Function

```
fs.readFile('users.json', function (error, text) {  
  if (error) {  
    console.error(error.message);  
  } else {  
    try {  
      var obj = JSON.parse(text);  
      console.log(obj);  
    } catch (e) {  
      console.error(err.message);  
    }  
  }  
});
```

`fs.readFile(file[, options], callback)`

param 1

param 2

```
fs.readFile('users.json', function (error, text) {  
  if (error) {  
    console.error(error.message);  
  } else {  
    try {  
      var obj = JSON.parse(text);  
      console.log(obj);  
    } catch (e) {  
      console.error(err.message);  
    }  
  }  
});
```

Named Function

```
function readFileSimple(error, text) {  
  if (error) {  
    console.error(error.message);  
  } else {  
    try {  
      var obj = JSON.parse(text);  
      console.log(obj);  
    } catch (e) {  
      console.error(err.message);  
    }  
  }  
}
```

```
fs.readFile('users.json', readFileSimple);
```

↑ param 1 ↑ param 2

fs.readFile(file[, options], callback)

Function Object

```
const readFileFunc = function (error, text) {  
  if (error) {  
    console.error(error.message);  
  } else {  
    try {  
      var obj = JSON.parse(text);  
      console.log(obj);  
    } catch (err) {  
      console.error(err.message);  
    }  
  }  
};
```

```
fs.readFile('users.json', readFileFunc);
```

↑
param 1

↑
param 2

fs.readFile(file[, options], callback)

Named Arrow Function

```
const readFileArrow = (error, text) => {  
  if (error) {  
    console.error(error);  
  } else {  
    try {  
      var obj = JSON.parse(text);  
      console.log(obj);  
    } catch (err) {  
      console.error(err.message);  
    }  
  }  
};
```

```
fs.readFile('users.json', readFileArrow);
```

↑
param 1

↑
param 2

fs.readFile(file[, options], callback)

`fs.readFile(file[, options], callback)`

param 1

param 2

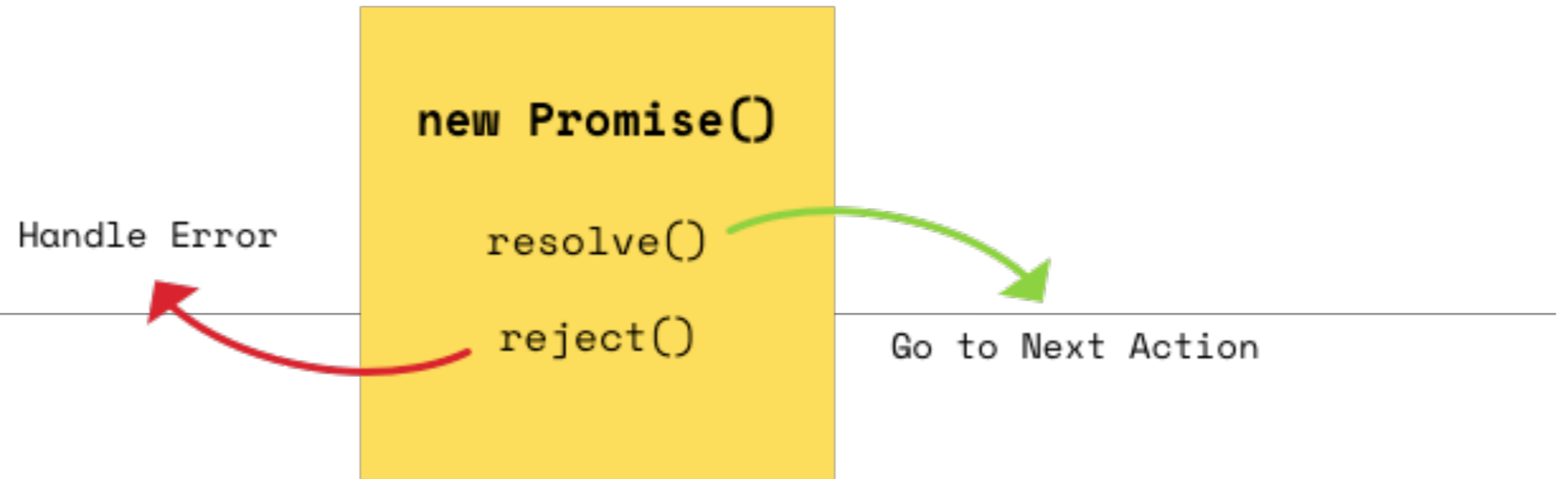
```
fs.readFile('users.json', (error, text) => {  
  if (error) {  
    console.error(error);  
  } else {  
    try {  
      var obj = JSON.parse(text);  
      console.log(obj);  
    } catch (err) {  
      console.error(err.message);  
    }  
  }  
});
```

Anonymous Arrow Function

```
fs.readdir('/Users', function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err);
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename);
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err);
        } else {
          console.log(filename + ' : ' + values);
          aspect = (values.width / values.height);
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect);
            console.log('resizing ' + filename + ' to ' + height + 'x' + height);
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function (err) {
              if (err) console.log('Error writing file: ' + err);
            });
          });
        }
      });
    });
  }
});
```

- callbacks within callbacks within callbacks....
- unreadable and unmaintainable
- telltale 'pyramid' shape

Promises

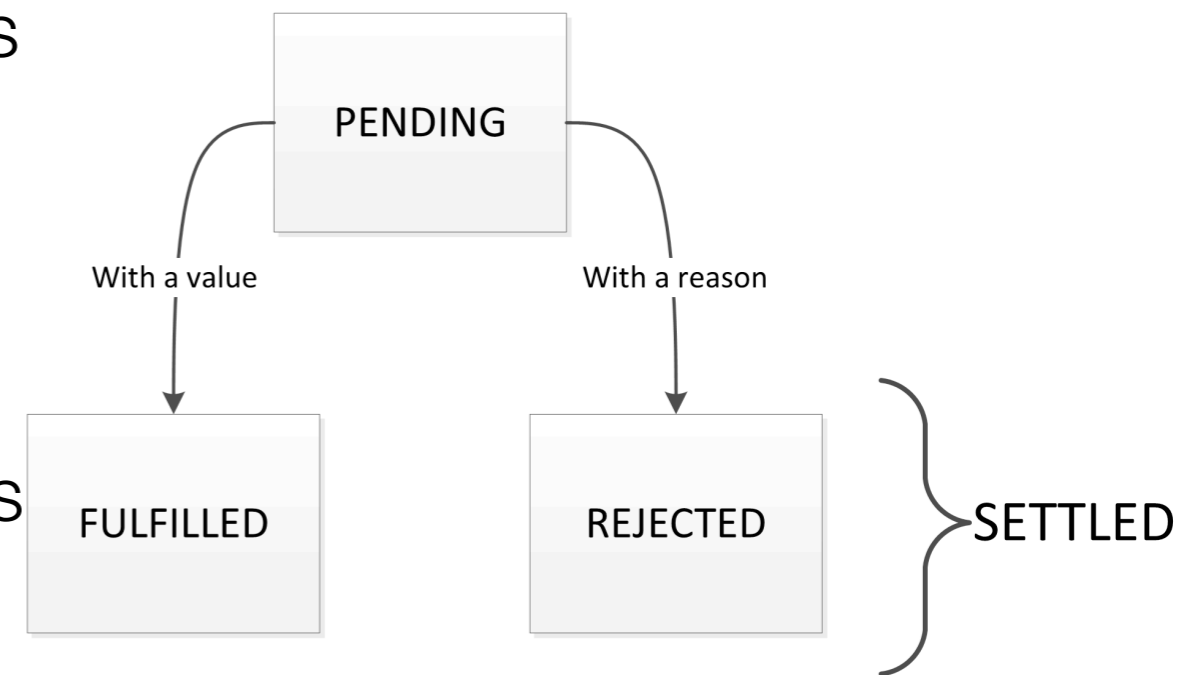


- Promises provide a simpler alternative for executing, composing, and managing asynchronous operations when compared to traditional callback-based approaches.
- Gradually replacing Callbacks in many libraries and applications
- Implements a simpler and more robust pattern for asynchronous programming
- Is seen as one solution to 'Callback Hell' problem

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.

Promise States



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 is regarded as **Settled**

Promise Examples

- Library must support promises
- Or must be converted to use promises in some fashion (promisify techniques)



find packages

★ fs-readfile-promise public

Promise version of fs.readFile

npm v3.0.0 build passing build passing coverage 100% dependencies up to date
devDependencies out of date

Promises/A+ version of `fs.readFile`

```
const readFile = require('fs-readfile-promise');

readFile('path/to/file')
  .then(buffer => console.log(buffer.toString()))
  .catch(err => console.error(err.message));
```

Based on the principle of *modular programming*, this module has only one functionality `readFile`, unlike other promise-based file system modules. If you want to use a bunch of other `fs` methods in the promises' way, choose other modules such as `q-io` and `fs-promise`.

Installation

Use npm.

```
npm install fs-readfile-promise
```

```
const readFile = require('fs-readfile-promise');
```

Callback

```
var fs = require('fs');

fs.readFile('users.json', function (error, text) {
  if (error) {
    console.error(error.message);
  } else {
    try {
      var obj = JSON.parse(text);
      console.log(obj);
    } catch (e) {
      console.error(err.message);
    }
  }
});
```

Promise

```
const readFile = require('fs-readfile-promise');

readFile('users.json')
  .then(text => {
    try {
      var obj = JSON.parse(text);
      console.log(obj);
    } catch (err) {
      console.error(err.message);
    }
  })
  .catch(err => {
    console.error(err.message);
  });
```

- In this small example, no major advantage to using promises
- However, as soon as callbacks become nested, then promises quickly become cleaner and simpler approach
- e.g: interacting with a database to lookup multiple objects, modify them and then save updates if no errors have occurred.

Promises - Function Objects/Promise Object

Standard Function
Objects readSuccess
& readFail

readFile returns a promise object

install success and fail methods
into promise

```
const readSuccess = function (text) {  
  try {  
    var obj = JSON.parse(text);  
    console.log(obj);  
  } catch (err) {  
    console.error(err.message);  
  }  
};  
  
const readFail = function (err) {  
  console.error(err.message);  
};  
  
const promise = readFile('users.json');  
  
promise.then(readSuccess);  
promise.catch(readFail);
```

Promises - Arrow Function/Promise Object

Standard Function
Objects readSuccess
& readFail

readFile returns a promise object

install success and fail methods
into promise

```
const readSuccessArrow = text => {  
  try {  
    var obj = JSON.parse(text);  
    console.log(obj);  
  } catch (err) {  
    console.error(err.message);  
  }  
};  
  
const readFailArrow = err => {  
  console.error(err.message);  
};  
  
const promise = readFile('users.json');  
  
promise.then(readSuccess);  
promise.catch(readFail);
```

Promises - Function Objects/Chaining

Standard Function
Objects readSuccess
& readFail

install success and fail methods
into promise directly (chaining)

```
const readSuccess = function (text) {  
  try {  
    var obj = JSON.parse(text);  
    console.log(obj);  
  } catch (err) {  
    console.error(err.message);  
  }  
};  
  
const readFail = function (err) {  
  console.error(err.message);  
};  
  
readFile('users.json')  
  .then(readSuccess)  
  .catch(readFail);
```

Promises - Arrow Functions/Chaining

Standard Function
Objects readSuccess
& readFail

install success and fail methods
into promise directly (chaining)

```
const readSuccessArrow = text => {  
  try {  
    var obj = JSON.parse(text);  
    console.log(obj);  
  } catch (err) {  
    console.error(err.message);  
  }  
};  
  
const readFailArrow = err => {  
  console.error(err.message);  
};  
  
readFile('users.json')  
  .then(readSuccess)  
  .catch(readFail);
```

Promises - Anonymous Arrow Functions

```
const readFile = require('fs-readfile-promise');

readFile('users.json')
  .then(text => {
    try {
      var obj = JSON.parse(text);
      console.log(obj);
    } catch (err) {
      console.error(err.message);
    }
  })
  .catch(err => {
    console.error(err.message);
  });
```

Update 2017: Node 8: Promises are now supported by the core

util.promisify(original)

Added in: v8.0.0

- `original` `<Function>`

Takes a function following the common Node.js callback style, i.e. taking a `(err, value) => ...` callback as the last argument, and returns a version that returns promises.

For example:

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);
stat('.').then((stats) => {
  // Do something with `stats`
}).catch((error) => {
  // Handle the error.
});
```