JavaScript Development Series

How to Design and Create

NODE.JS MODULES

Mitch Allen

Table of Contents

Copyright

Introduction

1 - Objects as Modules

2 - Properties

3 - Methods

4 - Module Parameters

5 - Named Exports

6 - Inheritance

7 - Callbacks

8 - Promises

9 - Unit Testing

Wrapping Up

Appendix A: JavaScript Editors

Appendix B: Install jshint

About the Author

About the Editor

JavaScript Development Series

How to Design and Create Node.js Modules

Published in the United States of America

by Mitch Allen

Kindle Edition 1, January 2017

http://mitchallen.com

Introduction

The purpose of this book is to show you how to design and create node.js modules. Node.js modules are external functions or objects that you can integrate into your project. You can think of them as a library. They are written in separate JavaScript files and can be assigned to a variable in your main file with one line of code. If you know how to write JavaScript, you know to create a module.

Modules can also exist in third party packages published on the Web. For more information on how to package and publish a module, see my book **How to Create and Publish Node.js Packages**. You can find a link to the book on my home page.

Why Modules

Node.js projects can grow very quickly to the point where they will no longer fit into one *.js file. Modules are a way to break the code up into manageable and testable chunks. With browser based JavaScript, HTML script tags are one way to pull multiple source files into one place. Node.js doesn’t have that option. Instead, it relies on the built-in **require** module to pull code from one JavaScript file into another.

What this Book Covers

This book walks through the steps of designing and creating node.js modules. It starts with how to convert JavaScript objects into modules and export them for external use. Other areas that this book covers include: naming exports, constructor parameters, properties, methods, inheritance, callbacks, promises and unit testing.

Here is a more detailed breakdown of the book by chapter:

Chapter 1: Objects as Modules

The book starts with the creation of a simple module which returns a single object. A factory based module is introduced that can create multiple objects. Learn how to seal an object to protect it from changes.

Chapter 2: Properties

The next step after defining objects is to add properties. Learn how to add properties that can be enumerated and protected against invalid values.

Chapter 3: Methods

Methods are what functions are called when they are part of an object. Define methods that can accept named parameters. Learn how to chain methods together and guard against invalid parameters.

Chapter 4: Module Parameters

Modules can be passed parameters, just like a functions or methods. Learn how to pass parameters to a module or a factory function to construct objects.

Chapter 5: Named Exports

Instead of returning the whole module, only return part of it. Learn how to export named objects and functions and special techniques for naming factory related exports.

Chapter 6: Inheritance

Modules and objects can be derived from other modules and objects. Learn how to derive and combine objects and override methods in parent objects.

Chapter 7: Callbacks

When using third party libraries with callbacks, care must be taken to integrate them into a module. Learn how to create modules that wrap existing callbacks and define new callbacks.

Chapter 8: Promises

Callbacks calling callbacks calling callbacks can get out of hand. Learn how to create modules that wrap third party callbacks in promises and define new promises.

Chapter 9: Unit Testing

When making changes to a module it is handy to be able to run a suite of tests to make sure that nothing broke. It’s also helpful to make sure that new functionality works as expected. Learn how to create and run unit tests against modules.

Appendix A: JavaScript Editor

To use this book, you will need a JavaScript source code editor. Any text editor will do. But, it would be to your advantage to use an editor that can color code JavaScript. For suggestions on JavaScript editors, please see: Appendix A: JavaScript Editors.

Appendix B: Using jshint

The **jshint** tool is introduced for quickly finding syntax related bugs in your code. If you get stuck trying to find a syntax error in your code, please refer to: Appendix B: Install jshint.

Command Line

When you see a line beginning with a dollar sign (**$**) like this:

```
$ node index.js
```

That means it is something that you should type at a command line. A command line is what appears when you open up a terminal or shell window. When you see a line like the one shown above, you should just type **node index.js** and then hit the enter/return key. Do NOT include the **$**. That's just to point out that it is command line statement.

Operating Systems

If you are using Linux or OS X you should not have any trouble going through the chapters in this book.

If you are using Windows I would recommend looking at free tools like VirtualBox (https://www.virtualbox.org) to run Linux in a Virtual Machine (VM). A good version of Linux to run in a VM is Ubuntu (https://www.ubuntu.com). If you are working on a project where the production servers run a different version of Linux, then you should install that version instead.

I've worked on many projects where I had a Windows laptop, but all of my work was done on it inside of a Linux VM.

Use Strict

Most of the software examples in this book will start with this line:

“use strict”;

That statement is ignored by older versions of JavaScript. Newer versions will interpret that as a directive to be more strict while parsing. It will catch issues like misspelled variable names that could cause unseen issues in your code.

Function or Method?

In most object oriented languages, a function that is attached to an object is referred to as a method. When the word method, is used in this book it is referring to a function attached to an object.

Update Node.js

This book uses some features of ES6. It is the latest standard for writing JavaScript. To get full use out of this book, you should update your copy of node.js to the latest version.

You should have at least version 7.x. To check, type the following at the command line:

```
$ node -v
```

On a Mac and some versions of Linux, you can update your copy of node.js and npm like this:

```
$ sudo n stable
$ sudo npm cache clean -f
$ sudo npm -g install npm@next
```

There were some issues updating npm after upgrading to node.js 7.x. If you run into a problem with npm not finding modules, try this:

```
$ sudo n 6.9.1
$ sudo npm -g install npm@next
$ sudo n stable
```

1

Objects as Modules

Many programming languages require that objects are instantiated (created) from prototypes. In C++, that would be a class definition. In JavaScript, you can use a prototype; but, you don’t alway have to. You can either build objects on the fly, or create a function that returns a new object. Then, you can export them from a module.

This chapter covers:

- Creating a simple module
- Adding a function
- Refactoring object definitions
- Singletons
- Returning a function that returns an object
- Problems with exposed properties
- Sealing an object
- Sealing vs. freezing

Create a Simple Module

The simplest type of module that can be created is one that exports an empty object. This exercise shows how to export an object from a module, assign it to a variable, and add a property to it.

To setup your project, open up a terminal window and type the following at the command line:

$ cd javascript

$ mkdir mod-101

$ cd mod-101

$ touch my-obj.js index.js

This sets up a folder to work with called **javascript/mod-101**. The touch command created empty JavaScript files in the new folder.

Open up **my-obj.js** in a text or JavaScript editor, add the code below, then save it:

```
“use strict”;

module.exports = {};
```

An empty object is exported for use by any file that requires this file.

Open up **index.js** in a text or JavaScript editor, add the code below, then save it:

```
“use strict”;

var obj = require(‘./my-obj’);

obj.name = “Droid”;

console.log( “My name is”, obj.name );
```

The module is required, which will return to **obj** an empty object. Then a **name** property is tacked on to it. This differs from other languages where adding properties after an object is created would not be allowed.

Open up a command line, change to the folder where the file is and run the following:

$ node index.js

The output should be:

My name is Droid.

Add a Function

Exported objects, like any object, can also have functions added to them. This exercise shows how to add a function to an exported object and call it.

Make the changes in **bold** below to **index.js**, then save it:

```
"use strict";

var obj = require('./my-obj');

obj.name = "Droid";

// console.log( "My name is", obj.name );

obj.hello = function() {
    console.log( "Hello, my name is", this.name );
}

obj.hello();
```

The original console line is commented out. A **hello** function is tacked on to the object that encapsulates a new console line. Because the function is now part of the object, the console line can refer to the name property using **this** instead of **obj**. The new method can now be called.

At the command line, run the file again:

```
$ node index.js
```

The output should be:

```
Hello, my name is Droid.
```

Refactoring Object Definitions

Tacking properties and functions on to an object is perfectly valid. It's a neat trick that you could exploit in many ways. But, what if you want to create two types of the same object? Are you going to assign every function to every object? This exercise shows how to move name and property definitions to a module.

Edit **my-obj.js**, add the code in **bold**, then save it:

```
"use strict";

module.exports = {
    name: "unknown",
    hello: function() {
        console.log( "Hello, my name is", this.name );
    }
};
```

The **name** property and the **hello** function are now part of the object that the module will export.

Edit **index.js**, comment out the code in **bold**, then save it:

```
"use strict";

var obj = require('./my-obj');

obj.name = "Droid";

// console.log( "My name is", obj.name );

// obj.hello = function() {
//     console.log( "Hello, my name is", this.name );
// }
```

```
obj.hello();
```

The hello function is now part of the object exported by the module. It no longer needs to be tacked on to the object in index.js

Open up a command line, change to the folder where the file is and run the following:

```
$ node index.js
```

The output should be:

```
Hello, my name is Droid.
```

Singletons

A singleton occurs when only one object can be created. An example might be a logging class that only writes to one file or console. There may never be a need for more than one instance. A function can also be thought of as a singleton.

With the **require** module, care needs to be taken to avoid generating multiple objects that turn out to all be pointing to the same singleton. A common mistake is to call require twice. It may appear to return multiple objects. Instead, it keeps returning pointers to the same object as the code below will demonstrate.

Create a new file in the folder called **singleton.js**, add the code below and save it:

```
“use strict”;

var obj1 = require(‘./my-obj’);
var obj2 = require(‘./my-obj’);

obj1.name = “Droid”;
obj2.name = “Robbie”;

obj1.hello();
obj2.hello();
```

Two variables (**obj1** and **obj2**) are assigned an object exported by the module. The **name** property is set on each object. Finally, the **hello** method is called to echo to the console a message containing the value of the name property for each object.

At the command line, run the **singleton.js** file:

```
$ node singleton.js
```

You might think the output should be:

```
Hello, my name is Droid
```

Hello, my name is Robbie

But instead you get this:

Hello, my name is **Robbie**

Hello, my name is Robbie

Instead of creating two objects the **require** method is caching the first one. The second variable (**obj2**) is pointing to the same object as **obj1**. This problem can be resolved using a **factory pattern**.

Export a Function that Returns an Object

Besides objects, modules can also export functions. Any type of function can be exported. A very useful function to export is one that when called returns a new object. This avoids the problem of unintentionally generating multiple variables pointing to a singleton as the code below will demonstrate.

Modify **my-obj.js** so that it exports a function that returns an object, then save it:

```
“use strict”;

module.exports = function() {
    return {
        name: “unknown”,
        hello: function() {
            console.log( “Hello, my name is”, this.name );
        }
    }
};
```

Instead of exporting the object, the module now exports a function that will return the object.

Create a new file in the folder called **factory.js**, add the code below, then save it:

```
“use strict”;

var factory = require(‘./my-obj’);

var obj1 = factory();
var obj2 = factory();

obj1.name = “Droid”;
obj2.name = “Robbie”;
```

obj1.hello();

obj2.hello();

Instead of an object, the module now exports a function that when called will return the object. The **factory** variable is set to the function. It is important to note that the require call is returning the function and not yet calling the function.

The factory function is called twice. Each time it returns a unique object that will not interfere with other objects.

The **name** property is set on each object. Finally, the **hello** method is called on each object to echo a message containing the value of the name property to the console.

At the command line, run the **factory.js** file:

$ node factory.js

The output should be:

Hello, my name is Droid

Hello, my name is Robbie

It’s perfectly valid to treat the require statement like a function that can be called and do something like this:

var obj1 = require(‘./my-obj’)();

var obj2 = require(‘./my-obj’)();

Note the parentheses at the end. This sort of form should be avoided for a variety of reasons. For starters, despite some caching, this may end up invoking a disk or parsing operation which can slow things down. If you only need to call require once as a function, then it may be acceptable.

The Problem with Exposed Properties

Defining properties inside a function solves some issues and makes the code easier to manage. But, things can still go wrong. The code below demonstrates what happens when the wrong property name is used.

Create a new file called **bad-prop.js** in the folder, add the code below and save it:

```
“use strict”;

var factory = require(‘./my-obj’);

var obj = factory();

// Non-existent property
obj.firstName = “Droid”;

obj.hello();

console.log(obj);
```

At the command line, run the following:

```
$ node bad-prop.js
```

The output should be:

```
Hello, my name is unknown
{ name: ‘unknown’, hello: [Function: hello], firstName: ‘Droid’ }
```

That isn’t the bad part. The bad part is that the code didn’t catch an error. Not only is the wrong property used, but the invalid property has been tacked on to the object. This can be solved by sealing the returned object.

Sealing an Object

There is a way to prevent adding new properties to objects. JavaScript has a built-in object called **Object**. Object has a method called **seal**. When called, the seal method will seal an object passed to it. This will prevent new properties from being added to the object. The existing properties can still be changed. But nothing unexpected can be added.

Edit **my-obj.js**, make the changes in bold, then save it:

```
“use strict”;

module.exports = function() {

  var obj = {
      name: “unknown”,
      hello: function() {
          console.log( “Hello, my name is”, this.name );
      }
   }

   Object.seal(obj);

   return obj;
};
```

The object is now set to a local variable (**obj**). The object is then sealed and returned when the exported function is called.

Run **bad-prop.js** again:

```
$ node bad-prop.js
```

The output should contain:

…/mod-101/bad-prop.js:8

obj.firstName = “Droid”;

^

TypeError: Can’t add property firstName, object is not extensible

If you didn’t put **“use strict”;** at the top of **bad-prop.js**, the error would have been silent and nothing would have changed. But, you wouldn’t have known there was an error. To test it, comment out the top line and run it again.

Seal vs. Freeze

The **Object.seal** method prevents new properties from being added to an object.
Object.freeze prevents changes to the value of the properties.

2

Properties

Any object in JavaScript that isn’t sealed or frozen can have properties added to it. They can be added when the object is defined, or after the object has been created. The built-in **Object.defineProperty** and **Object.defineProperties** methods can be used to add properties where you can control how they are set, retrieved and accessed.

This chapter covers:

- Defining properties
- Enumerating properties
- Enabling properties
- Guarding properties

Defining Properties

Object properties can be set using the **Object.defineProperty** and **Object.defineProperties** built-in methods. The later method requires two parameter: the object where the properties will be defined, and an array of property definitions. Each definition contains a label, representing the name of the property, followed by an object. The object may contain one or more of these attributes:

- **writable** - defaults to true. If false, the property is read-only and can not be changed.
- **value** - provides a value to return if the property is read-only.
- **enumerable** - defaults to true. The property can be listed when the object is enumerated.
- **get** - a getter function usually wrapping an internal (encapsulated) variable.
- **set** - a setter function usually used to set an encapsulated variable holding the property value.
- **configurable** - defaults to false. The property definition may be changed or the property may be deleted.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir props-101
$ cd props-101
$ touch pet.js index.js
```

Edit **pet.js**, add the code below and save it:

```
“use strict”;

module.exports = () => {

    var m_name = “unknown”;

    var obj = {
        hello: function() {
            console.log( “Hello, my name is”, m_name );
```

```
    }
  };

  Object.defineProperties( obj, {
    “type”: {
      writeable: false,
      value: “pet”
    },
    “name”: {
      get: () => m_name,
      set: (newValue) => m_name = newValue
    },
  });

  Object.seal(obj);

  return obj;
};
```

A local variable is created to hold the name value (**m_name**). An object (**obj**) is created with one method to echo the name to the console (**hello**). Two properties (**type**, **name**) are added to the object using the built-in method **Object.defineProperties**. The **type** property is read-only, so it only has a get method. The **name** property can be set and retrieved. It has a **get** and **set** function that encapsulates the local **m_name** variable. The object is sealed so that no new properties can be added either intentionally or by mistake. Finally, the object is returned when the function is called.

Edit **index.js**, add the code below, then save it:

```
“use strict”;

var factory = require(‘./pet’);

var p1 = factory();
```

p1.name = “Rover”;

p1.hello();

The **require** method sets **factory** to the function exported by the module. When the factory function is called, it returns a sealed object and assigns it to **p1**. The properties of a sealed object can be modified. But, no new properties may be added. The **name** property is set to a string. The **hello** method is called on the object to echo a message with the objects name to the screen.

Open up a command line, change to the folder where the file is and run the following:

$ node index.js

The output should be:

Hello, my name is Rover

Enumerating Properties

Add the following to the bottom of **index.js** and save it:

```
var info = (obj) => {
   console.log("\nProperties:");
   Object.keys(obj).forEach( key => {
      console.log(key, ":", obj[key]);
   });
};

info(p1);
```

An **info** function is defined that will log the properties of an object. The **p1** variable is passed to the **info** function.

Run the file again:

```
$ node index.js
```

The output should be:

```
Hello, my name is Rover

Properties:
hello : function () {
          console.log( "Hello, my name is", m_name );
       }
```

The function is logged to the screen, but what about **type** and **name**?

Enabling Enumeration

Edit **pet.js**, make the changes in **bold** below to the property definitions block of code and save the file (don’t forget the commas at the end of the lines before the new lines):

```
Object.defineProperties( obj, {
    “type”: {
        writeable: false,
        value: “pet”,
        enumerable: true
    },
    “name”: {
        get: () => m_name,
        set: (newValue) => m_name = newValue,
        enumerable: true
    },
});
```

Run the file again:

```
$ node index.js
```

The output should be:

```
Hello, my name is Rover

Properties:
hello : function () {
            console.log( “Hello, my name is”, m_name );
        }
type : pet
name : Rover
```

Because the **type** and **name** properties are now enumerable, they can be enumerated by the info function.

Guarding Properties

The simplest form of the **set** function takes a potential property value and assigns it to an internal variable without any checking. The function can be expanded to add checking to see if a value is within a certain range. If the value is out of range, the set function can be written to handle it one of several ways. An exception can be thrown or the new value can be ignored or normalized.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir prop-guard
$ cd prop-guard
$ touch index.js guard.js
```

Edit **guard.js**, add the code below, then save it:

```
“use strict”;

module.exports = () => {

   var m_month = 0,
      m_day = 1,
      m_year = 1970;

   var obj = {
      info: function() {
         console.log( “My date is: “, this.expiresOn );
      }
   };

   Object.defineProperties( obj, {
      “expiresOn”: {
         get: () => new Date( m_year, m_month, m_day, 0, 0, 0, 0)
```

```
        },
        "month": {
            get: () => m_month,
            set: function(newValue) {
                if( newValue < 0 || newValue > 11 ) {
                    console.warn("Month out of range [0-11]", newValue);
                    return;
                }
                m_month = newValue;
            },
        },
        "day": {
            get: () => m_day,
            set: (newValue) => m_day = newValue,
        },
        "year": {
            get: () => m_year,
            set: (newValue) => m_year = newValue,
        },
    });

    Object.seal(obj);

    return obj;
};
```

An object (**obj**) is defined with an **info** method. The object is passed to the **Object.defineProperties** method along with a set of property definitions. The Date month parameter is zero-based, meaning that 0 stands for January and 11 stands for December. The **month set** function guards against a value outside that range. The object is sealed and returned by the function.

Edit **index.js**, add the code below, then save it:

“use strict”;

var factory = require(“./guard.js”);

var d1 = factory();

d1.info();

d1.year = 2025;
d1.month = 12;

d1.info();

The **factory** variable is assigned the object returned by the function in the module. The function is then called to return a new object and assign it to a variable (**d1**). The **info** method is called to display the current date value of the object. An attempt is made to change the values of **year** and **month**. The setter method defined in the **Object.create** call will intercept the attempt to pass a bogus value to the month parameter. The info method is called again to show that only the year was altered.

Open up a command line, change to the folder where the file is and run the following:

$ node index.js

The output should be:

My date is: 1970-01-01T07:00:00.000Z
Month out of range [0-11] 12
My date is: 2025-01-01T07:00:00.000Z

The original date value of the object is displayed. The attempt to set the value of the month to an invalid range is logged to the console. The date is displayed again to show that only the year was changed.

Object.create

Besides **Object.defineProperty** and **Object.defineProperties**, properties can be defined when an object is first created using **Object.create**. There is one important difference. The define methods add properties to the existing object. The create method uses the object as a prototype to create and return a new object.

This changes the object (**obj**) passed to it:

```
Object.defineProperties( obj, { … } );
```

This does NOT change the object (**obj**) passed to it:

```
Object.create( obj, { … } );
```

To add properties to the current object, assign it to the return value from the create method:

```
obj = Object.create( obj, { … } );
```

Or, to make a distinction, call the original object something else and assign the result to a new object:

```
var proto = { … };

var obj = Object.create( proto, { … } );
```

To start with a fresh object that has nothing but properties, assign an empty object to the method.

```
var obj = Object.create( {}, { … } );
```

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
```

```
$ mkdir obj-create
$ cd obj-create
$ touch index.js obj-create.js
```

Edit **obj-create.js**, add the code below, then save it:

```
"use strict";

module.exports = () => {

    var m_month = 0,
        m_day = 1,
        m_year = 1970,
        m_name = "foo";

    var proto = {
        info: function() {
            console.log( "My date is: ", this.expiresOn );
        }
    };

    var obj = Object.create( proto, {
        "expiresOn": {
            get: () => new Date( m_year, m_month, m_day, 0, 0, 0, 0)
        },
        "month": {
            get: () => m_month,
            set: function(newValue) {
                if( newValue < 0 || newValue > 11 ) {
                    console.warn("Month out of range [0-11]", newValue);
                    return;
                }
```

```
            m_month = newValue;
        },
    },
    "day": {
        get: () => m_day,
        set: (newValue) => m_day = newValue,
    },
    "year": {
        get: () => m_year,
        set: (newValue) => m_year = newValue,
    },
});

Object.seal(obj);

return obj;
};
```

An object that will act as a prototype is passed to the **Object.create** method along with a set of property definitions. The definitions are defined the same way that they are defined for **Object.defineProperties**. The difference being that the create method returns a new object instead of altering the existing one. The new object is sealed and returned by the function.

Edit **index.js**, add the code below, then save it:

```
"use strict";

var factory = require("./obj-create.js");

var d1 = factory();

d1.info();
```

d1.year = 2025;

d1.month = 12;

d1.info();

The **factory** variable is assigned the object returned by the function in the module. The function is then called to create a new object and assign it to a variable (**d1**). The **info** method is called to display the current date value of the object. An attempt is made to change the values of **year** and **month**. The setter method defined in the **Object.create** call will intercept the attempt to pass a bogus value to the month parameter. The info method is called again to show that only the year was altered.

Open up a command line, change to the folder where the file is and run the following:

$ node index.js

The output should be:

My date is: 1970-01-01T07:00:00.000Z

Month out of range [0-11] 12

My date is: 2025-01-01T07:00:00.000Z

The original date value of the object is displayed. The attempt to set the value of the month to an invalid range is logged to the console. The date is displayed again to show that only the year was changed.

3

Methods

Methods are what functions are called when they are part of an object.

This chapter covers:

- Assigning methods to an object
- Named method parameters
- Guarding methods
- Method chaining

Assigning Methods

Methods can be assigned to an object when it is initially defined. If an object isn't sealed, new methods can be added to it later. Another way is to use the built-in **Object.assign** method to assign new methods to an object.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir obj-assign
$ cd obj-assign
$ touch index.js obj-assign.js
```

Edit **obj-assign.js**, add the code below, then save it:

```
"use strict";

module.exports = () => {

  var m_month = 0,
    m_day = 1,
    m_year = 1970;

  var obj = Object.create( {}, {
    "expiresOn": {
      get: () => new Date( m_year, m_month, m_day, 0, 0, 0, 0)
    },
    "month": {
      get: () => m_month,
      set: function(newValue) {
        if( newValue < 0 || newValue > 11 ) {
          console.warn("Month out of range [0-11]", newValue);
          return;
        }
```

```
            m_month = newValue;
        },
    },
    "day": {
        get: () => m_day,
        set: (newValue) => m_day = newValue,
    },
    "year": {
        get: () => m_year,
        set: (newValue) => m_year = newValue,
    },
});

obj = Object.assign( obj, {
    info: function() {
        console.log( "My date is: ", this.expiresOn );
    },
    hello: function() {
        console.log( "Hello World!");
    },
});

Object.seal(obj);

return obj;
};
```

The built-in **Object.create** method is called to create a new object with properties. The builtin **Object.assign** method is called to assign functions to the new object. The object is sealed and returned by the function.

According to the documentation, a new object is returned by the method, implying that the original is not altered. But, I've found that when I pass in an existing object as the first

parameter, the original object is assigned the new methods.

Edit **index.js**, add the code below, then save it:

```
"use strict";

var factory = require("./obj-assign.js");

var d1 = factory();

d1.hello();
d1.info();
```

The **factory** variable is assigned the function returned by the module. The function is called to assign a new object to a variable (**d1**). The two methods (**hello**, **info**) added by **Object.assign** are called.

At the command line, run the file:

```
$ node index.js
```

The output should be:

```
Hello World!
My date is:  1970-01-01T07:00:00.000Z
```

Named Method Parameters

When passing parameters to a constructor, function or method, things can get out of hand. Consider the JavaScript date object:

```
var d = new Date( 2018, 1, 3, 0, 0, 0, 0 );
```

It would be easy to get the parameters wrong. Is the second parameter the month or the day? What about all those zeros representing parts of the time value? Which parameter is which and what if the time doesn't matter? What if only the date is needed? How would you clean something like that up?

A common practice is to pass in just one object that names parameters, and only the parameters whose defaults need to be changed. For example:

```
var options = {
   year: 2018,
   month: 1,
   day: 2
};
var d = date( options );
```

If the signature of the date parameters changes or a new parameter is added, existing code doesn't need to be changed. Parameter order doesn't matter and changes to the module method won't break existing code.

Internally, the date method can assign defaults for any missing parameters:

```
module.exports = (spec) => {
   spec = spec || {}
   var m_year = spec.name || 1970,
      m_month = spec.month || 1,
      …
      m_milliseconds = spec.milliseconds || 0;
```

If later a timezone parameter were added, it wouldn't break existing code.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir method-params
$ cd method-params
$ touch index.js logger.js
```

Edit **logger.js**, add the code below and save it:

```
"use strict";

module.exports = {

    log: function(options) {

        options = options || {};

        var message = options.message || "",
            category = options.category || "";

        console.log(
            new Date()
            + ": [" + category + "] "
            + message );
    }
};
```

A singleton is defined that contains one method (**log**). The log method has one parameter (**options**). If options is null then the value defaults to an empty object (**{}**). If **options.message** or **options.category** are defined, their values are assigned to one of two variables (**message** or **category**). Otherwise, those values default to empty strings. Finally,

the values are used with a data stamp to log a message to the console.

Edit **index.js**, add the code below and save it:

```
"use strict";

var logger = require('./logger.js');

logger.log( {
   message: "Hello World"
} );

logger.log( {
   message: "Danger World",
   category: "WARNING"
} );

logger.log();
```

The module is assigned to a variable (**logger**). The **log** method is called with an object that only contains one named parameter (**message**). This will result in a line with no category being echoed to the console. Next, the log method is called with both **message** and **category** parameters. Finally, the log method is called with no parameters. This will result in logging an empty message with no category.

Open up a command line, change to the folder where the file is and run the following:

```
$ node index.js
```

The output should be:

(date-stamp)**: [] Hello World**
(date-stamp)**: [WARNING] Danger World**

(date-stamp)**: []**

Guarding Methods

Property values can be guarded using a **set** function. Method parameters can also be guarded by checking the parameters passed to them. When an invalid value is passed to a method, it can be handled in several ways. The method can throw an exception, ignore the value or normalize it. It can also return a boolean or null value as another way to highlight a problem.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir method-guard
$ cd method-guard
$ touch index.js guard.js
```

Edit **guard.js**, add the code below and save it:

```
“use strict”;

module.exports = () => {

  var limit = 10,
    m_position = 0;

  var obj = {
    canMove: (loc) => (loc >= 0 && loc <= limit),
    moveTo: function( options ) {

      if( options === undefined ) {
        throw new Error(“moveTo requires parameters”);
      }

      if( typeof options !== ‘object’ ) {
        throw new Error(“moveTo requires object with named parameters”);
```

```
        }

        var pos = options.position;

        if(!pos) {
           return false;
        }

        if(!this.canMove(pos)) {
           return false;
        }

        m_position = pos;

        console.log("Moving to: ", this.position );

        return true;
     }
  };

  Object.defineProperties( obj, {
     "position": {
        get: () => m_position
     }
  });

  Object.seal(obj);

  return obj;
};
```

A **canMove** method is created that returns true or false if a move can occur to a new location (loc).

A **moveTo** method is defined that guards against several conditions:

- If no option are passed to the method, throw an exception.
- If the parameter passed to the method is not an object, throw an exception.
- If no position was passed in as a named parameter
- If canMove returns false for the position, return false.

If none of the error conditions are met, then **moveTo** should set the new position and return true.

Edit **index.js**, add the code below and save it:

```
"use strict";

var factory = require("./guard.js");

var tool = factory();

console.log( tool.canMove(5) );
console.log( tool.canMove(20) );

console.log( tool.moveTo( { position:  5 } ) );
console.log( tool.moveTo( { position: 10 } ) );
console.log( tool.moveTo( { position: 20 } ) );
console.log( tool.moveTo( {} ) );

console.log( "POSITION:", tool.position );
```

Open up a command line, change to the folder where the file is and run the following:

```
$ node index.js
```

The output should be:

true

false

Moving to: 5

true

Moving to: 10

true

false

false

POSITION: 10

Add this line, save the file and run the command again:

tool.moveTo();

The output should contain:

throw new Error(“moveTo requires parameters”);

^

Error: moveTo requires parameters

Comment out the line you just added, add this line to the bottom of the file, save it and run the command again:

// tool.moveTo();

tool.moveTo(5);

The output should contain:

throw new Error(“moveTo requires object with named parameters”);

^

Error: moveTo requires object with named parameters

Method Chaining

Method chaining is a way to reduce code complexity, making it easier to write and read.

Which looks simpler and is easier to write and read?

```
calc.clear()
calc.add(500);
calc.add(600);
calc.add(200);
var result = calc.value();
```

Or:

```
var result =
    calc.clear()
        .add(500)
        .add(600)
        .add(200)
        .value();
```

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir chaining
$ cd chaining
$ touch index.js calc.js
```

Edit **calc.js**, add the code below, then save it:

```
“use strict”;

module.exports = () => {
```

```
    var m_acc = 0;

    return {
        value: () => m_acc,
        clear: function() {
            m_acc = 0;
            return this;
        },
        add: function(n) {
            m_acc += n;
            return this;
        }
    };
};
```

The factory returns an object with three methods: **value**, **add** and **clear**. The **add** and **clear** methods return **this**. The **this** value refers to the current object. Because those two method calls return the object, you can call methods directly from them.

Edit **index.js**, add the code below, then save it:

```
"use strict";

var factory = require('./calc.js');

var calc = factory();

var result =
    calc.clear()
        .add(500)
        .add(600)
        .add(200)
```

```
    .value();

console.log( "Result: ", result );
```

You can't call a method from value because that returns a value and not the object. But you can place it on the end of the chain. Because it is at the end of the chain, the value it returns is the return value for the whole chain. That is why in the code the chain can be used to set the result variable. It will contain the value that was created by the clear and add methods.

At the command line, run the file:

```
$ node index.js
```

The output should be:

```
Result:  1300
```

4

Module Parameters

Values can be passed to a module using parameters. The module can use the parameters to set the initial values for objects and functions returned by the module.

This chapter covers:

- Module parameters
- Adding parameters
- Guarding against bad factory parameters

Module Parameters

When requiring a module, it can be useful to pass values to it. The values can be used internally or to initialize an object that will be returned by a factory function. The values passed in could be thought of as parameters to a constructor.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir params-101
$ cd params-101
$ touch pet.js index.js
```

Edit **pet.js**, add the code below, then save it:

```
“use strict”;

module.exports = (spec) => {

    spec = spec || {};
    var m_name = spec.name || “unknown”;

    var obj = {
        hello: function() {
            console.log(
                “\nHello, my name is”, m_name,
                “\nI am a”, this.type
            );
        }
    };

    Object.defineProperties( obj, {
        “type”: {
            writable: false,
```

```
            value: “pet”,
            enumerable: true
        },
        “name”: {
            get: () => m_name,
            set: newValue => m_name = newValue,
            enumerable: true
        }
    });

    Object.seal(obj);

    return obj;
};
```

The exported function takes one parameter (**spec**). If the spec parameter contains a **name** property, it will be used to set a local variable (**m_name**). A **hello** method echoes a message to the console containing the value of the local variable and the type property. Two public properties are defined (**type** and **name**). The **type** property is read-only and will return the string “pet.” The **name** property encapsulates the local **m_name** variable. The object is sealed. The exported function will return a new instance of the object when it is called.

Edit **index.js**, add the code below, then save it:

```
“use strict”;

var factory = require(‘./pet’);

var p1 = factory({
    name: “Rover”
});

p1.hello();
```

The **factory** variable is assigned the function exported by the **pet** module. When the factory function is called with a named parameter (**name**), it will return an object with the name property set to that value. The returned object will be assigned to the variable **p1**. When the **hello** method is called on the **p1** object, it should echo a message with the new name value to the screen.

At the command line, run the file:

$ node index.js

The output should be:

Hello, my name is Rover

I am a pet

Adding Parameters

Because a single object with named parameters is being used, adding new parameters won't break existing code.

Edit **pet.js**, make the changes in **bold** and save the file (don't forget the commas at the end of the line above the new lines).

```
"use strict";

module.exports = (spec) => {

    spec = spec || {};
    var m_name = spec.name || "unknown",
        m_sound = spec.sound || "unknown";

    var obj = {
        hello: function() {
            console.log(
                "\nHello, my name is", m_name,
                "\nI am a", this.type,
                "\nI say", m_sound
            );
        }
    };

    Object.defineProperties( obj, {
        "type": {
            writable: false,
            value: "pet",
            enumerable: true
        },
        "name": {
            get: () => m_name,
```

```
            set: newValue => m_name = newValue,
            enumerable: true
        },
        "sound": {
            get: () => m_sound,
            set: (newValue) => m_sound = newValue,
            enumerable: true
        }
    });

    Object.seal(obj);

    return obj;
};
```

A new local variable (**m_sound**) is defined. It can optionally be set by a new named parameter (**sound**) if it is passed to the module via the **spec** parameter. Otherwise, it will be set to the default string "unknown." The **hello** method is updated to include a message with the **m_sound** value that will be echoed to the screen. A publicly available property (**sound**) is defined to encapsulate the local variable **m_sound**.

Edit **index.js**, make the changes in bold and save the file (remember to add the comma above the new line):

```
"use strict";

var factory = require('./pet');

var p1 = factory({
    name: "Rover",
    sound: "woof"
});

p1.hello();
```

Run **index.js** again:

```
$ node index.js
```

The output should be:

```
Hello, my name is Rover
I am a pet
I say woof
```

Guarding Factory Parameters

This book has already covered guarding against bad parameters passed to properties or methods. What if a bad parameter is sent to a factory object before the object is even created?

As with properties and methods, there are a number of ways to guard against bad factory function parameters. The most popular being:

- Throw an exception
- Return false
- Return a null

A factory function is designed to return a valid object. Returning a null object can be a way to indicate an error.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir factory-guard
$ cd factory-guard
$ touch index.js guard-factory.js
```

Edit **guard-factory.js**, add the code below, then save it:

```
“use strict”;

module.exports = (spec) => {

  spec = spec || {};
  var month = spec.month || 1,
    day = spec.day || 1,
    year = spec.year || 1970;
```

```
    if(month < 0 || month > 11) {
        console.error("Month value is out of range [0-11]:", month );
        return null;
    }

    var expires = new Date( year, month, day, 0, 0, 0, 0);

    return {
        expiresOn: () => {
            console.log( "Expires on", expires );
        }
    };
};
```

A factory function is defined that will return null if the month parameter is out of range. Recall that the Date month parameter is zero-based (0 stands for January, 11 stands for December).

Edit **index.js**, add the code below, then save it:

```
"use strict";

var factory = require('./guard-factory.js');

var d = factory( { month: 12, day: 1, year: 2017 } );

if(!d) {
    console.error("Couldn't set date");
} else {
    d.expiresOn();
}
```

An attempt is made to call the factory function with an invalid month parameter. An **if**

statement will echo a message to the console if the factory returns a null.

At the command line, run the file:

```
$ node index.js
```

The output should be:

```
Month value is out of range [0-11]: 12
Couldn’t set date
```

5

Named Exports

Named exports are a way to label sections of a module to be exported. A module can export several objects and/or functions by name. A factory module can have an exported function named **create** for specifically creating and returning new objects.

This chapter covers:

- Creating a named export
- Exporting named functions
- Exporting a create method
- Exporting an object method

Create a Named Export

Named exports are a convenient way to breakup a module and refer to individual components.

This is one way to export an object with a property called **database** and another one called **server**:

```
module.exports = {
    database: {
        hostname: “db.example.com”
    },
    server: {
        hostname: “web.example.com”
    }
};
```

Another way to export an object with multiple properties is like this:

```
module.exports.database = {
    hostname: “db.example.com”
};

module.exports.server = {
    hostname: “web.example.com”
};
```

When written the second way, the individual items are referred to as “named exports.” Named exports, written either way, can be referred to in the code by their name:

```
var config = require(“./config”);

var db = config.database;
var server = config.server;
```

Any property in the exported object can be accessed just like any other JavaScript property. For example:

```
var dbHostname = config.database.hostname;
```

To setup a project to demonstrate named exports, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir named
$ cd named
$ touch index.js config.js
```

Edit **config.js**, add the code below, then save it:

```
module.exports.database = {
    hostname: “db.example.com”
};

module.exports.server = {
    hostname: “web.example.com”
};
```

The **module.exports** value is assigned an object containing two named objects (**database** and **server**). The named objects will be exported and returned when the file is called using the **require** function.

Edit **index.js**, add the code below, then save it:

```
var config = require(“./config”);

var db = config.database;
```

```
var server = config.server;

console.log( config );
console.log( db );
console.log( server );

console.log( “Database:”, db.hostname );
console.log( “  Server:”, server.hostname );
```

The **require** call will assign the exported object from **config.js** to the **config** variable. Two variables (**db** and **server**) are assigned the named objects. The objects are echoed to the console for demo purposes. The **hostname** values for **db** and **server** are echoed to the console.

At the command prompt run the following:

```
$ node index.js
```

The output should be:

```
{ database: { hostname: ‘db.example.com’ },
  server: { hostname: ‘web.example.com’ } }
{ hostname: ‘db.example.com’ }
{ hostname: ‘web.example.com’ }
Database: db.example.com
  Server: web.example.com
```

Export Named Functions

Just as named exports can be objects, they can also be functions.

A module exporting two functions can be written like this:

```
module.exports = {
    hello: () => {
        console.log(“Hello World”);
    },
    howdy: () => {
        console.log(“Howdy World”);
    }
};
```

Or, it can be written like this:

```
module.exports.hello = () => {
    console.log(“Hello World”);
};

module.exports.howdy = () => {
    console.log(“Howdy World”);
};
```

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir named-func
$ cd named-func
$ touch index.js greet.js
```

Edit **greet.js**, add the code below, then save it:

```
"use strict";

module.exports.hello = () => {
    console.log("Hello World");
};

module.exports.howdy = () => {
    console.log("Howdy World");
};
```

The **module.exports** value is assigned an object containing two named functions. The named functions will be exported and returned when the file is called using the **require** function.

Edit **index.js**, add the code below, then save it:

```
"use strict";

var greet = require('./greet');

var hello = greet.hello;
var howdy = greet.howdy;

console.log(greet);

hello();
howdy();
```

The **require** call will assign the exported object from **greet.js** to the **greet** variable. Two variables (**hello** and **howdy**) are assigned the named functions. The **greet** object is echoed to the console for demo purposes. The two functions are called to echo greetings to the console.

At the command prompted run the following:

$ node index.js

The output should be:

{ hello: [Function], howdy: [Function] }
Hello World
Howdy World

You could also call the functions like this:

require('./greet').hello();

If the function returned anything, the **require** statement could be used to set a variable (**var x = require …**). This should be avoided unless the function and require are only called once.

Export a Create Method

Which is more intuitive?

```
var obj = factory();
```

Or:

```
var obj = factory.create();
```

Either form will work. But using a named export may make the code easier to read.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir create-factory
$ cd create-factory
$ touch index.js obj-factory.js
```

Edit **obj-factory.js**, add the code below, then save it:

```
“use strict”;

module.exports.create = function() {

  var obj = {
      name: “unknown”,
      hello: function() {
          console.log( “Hello, my name is”, this.name );
      }
   }

    Object.seal(obj);
```

```
    return obj;
};
```

The **module.exports** value is assigned an object containing a named function (**create**). The named function will be exported. The **create** method will return an object that contains a **name** property and a **hello** function.

Edit **index.js**, add the code below, then save it:

```
var factory = require('./obj-factory')

var p1 = factory.create();
var p2 = factory.create();

p1.name = "Jack";
p2.name = "Jill";

p1.hello();
p2.hello();
```

The **require** call will assign the exported object to the **factory** variable. The **create** method is called twice to create two objects (**p1** and **p2**). The **name** function on each person object is set. The **hello** function is called to echo a message with the object's name to the console.

Because the code is using a factory object that returns two unique objects, there should be no cache issues. Each object can be set without interfering with the other one.

At the command prompt, run the following:

```
$ node index.js
```

The output should be:

Hello, my name is Jack

Hello, my name is Jill

Export an Object Method

Naming a method “create” is one way to name an export. Another way is to name it after the object that it creates. This is most convenient when a factory can return more than one type of object.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir named-obj
$ cd named-obj
$ touch index.js dog.js
```

Edit **dog.js**, add the content below and save it:

```
“use strict”;

module.exports.Dog = (spec) => {

    spec = spec || {};
    var m_name = spec.name || “default”;

    var obj = {
        info: function() {
            console.log( “Info: “, this.name, “is a”, this.type );
        }
    };

    Object.defineProperties( obj, {
        “type”: {
            get: () => “dog”,
            enumerable: true
        },
        “name”: {
```

```
            get: () => { return m_name; },
            set: (newValue) => { m_name = newValue; },
            enumerable: true
        }
    });

    Object.seal(obj);

    return obj;
};
```

Edit **index.js**, add the content below and save it:

```
"use strict";

var factory = require('./dog.js');

var dogs = [
    factory.Dog( { name: "Zeus" } ),
    factory.Dog( { name: "Rover" } ),
    factory.Dog( { name: "Spot" } )
];

dogs.forEach( dog => {
    dog.info();
});
```

At the command line, run the file:

```
$ node index.js
```

The output should be:

Info: Zeus is a dog

Info: Rover is a dog

Info: Spot is a dog

6

Inheritance

Inheritance is when one object inherits the properties and function of another object. The derived object can redefine or add new properties and functions.

This chapter covers:

- Deriving objects
- Combining objects
- Overriding methods

Deriving Objects

When an object is derived from another object, it inherits all of its functions and properties. Some of those functions and properties can be overridden and some of them can't be. For example, a base class with a read-only type parameter that returns the string "pet" can be overridden in a child class to return "dog" instead.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir derive
$ cd derive
$ touch index.js pet.js dog.js cat.js
```

Edit **pet.js**, add the content below and save it:

```
"use strict";

module.exports = function (spec) {
    spec = spec || {};
    var m_name = spec.name || "default";

    var obj = {
        info: function() {
            console.log( "Info: ", this.name, "is a", this.type );
        }
    };

    Object.defineProperties( obj, {
        "type": {
            writable: false,
            value: "pet",
            enumerable: true,
            configurable: true  // So it can be redefined
```

```
    },
    "name": {
      get: () => m_name,
      set: (newValue) => m_name = newValue,
      enumerable: true
    }
  });

  // Can't seal or else can't be extended
  // Object.seal(obj);

  return obj;
};
```

A pet factory function is defined that takes one named parameter (**name**). When called, the function will return an object with one method (**info**) and two properties (**type** and **name**). The type property is hard coded to return the string "pet." The type property has it's attribute **configurable** set to **true** so that it can be redefined in a derived object. Normally, an object could be marked as sealed. But, if that is done, the object can't be extended. To illustrate that point, the code is provided but commented out.

Edit **dog.js**, add the content below and save it:

```
"use strict";

var petFactory = require('./pet.js');

module.exports = (spec) => {

  var obj = petFactory(spec);

  Object.defineProperties( obj, {
    "type": {
      writable: false,
```

```
        value: “dog”,
        enumerable: true
      }
    });

    Object.seal(obj);

    return obj;
  };
```

A dog factory function is defined that takes one named parameter (**name**). When called, the function will return an object that inherits the **info** method and **name** property from a **pet** object. The **type** property is overridden to return the hard-coded string “dog.” The object is sealed before it is returned.

Edit **cat.js**, add the content below and save it:

```
  “use strict”;

  var petFactory = require(‘./pet.js’);

  module.exports = (spec) => {

    var obj = petFactory(spec);

    Object.defineProperties( obj, {
      “type”: {
        writabe: false,
        value: “cat”,
        enumerable: true
      }
    });
```

```
    Object.seal(obj);

    return obj;
};
```

A cat factory function is defined that takes one named parameter (**name**). When called, the function will return an object that inherits the **info** method and **name** property from a **pet** object. The **type** property is overridden to return the hard-coded string “cat.” The object is sealed before it is returned.

Edit **index.js**, add the content below and save it:

```
“use strict”;

var petFactory = require(‘./pet.js’),
    dogFactory = require(‘./dog.js’),
    catFactory = require(‘./cat.js’);

var pets = [
    petFactory( { name: “Zeus” }),
    dogFactory( { name: “Rover” }),
    dogFactory( { name: “Spot” }),
    catFactory( { name: “Kitty” })
];

pets.forEach( pet => {
    pet.info();
});
```

Pet, dog and cat factory functions are initialized by modules. Through a **spec** object, they each take one named parameter which is the **name** of the pet.

When the **info** method is called on the dog or cat objects, that method will use the overridden value to print the **type** of the pet.

At the command line, run the file:

$ node index.js

The output should be:

Info: Zeus is a pet
Info: Rover is a dog
Info: Spot is a dog
Info: Kitty is a cat

Combining Objects

Factories can return not just one type of object but several types of objects. The factory can hide the implementation details of creating each object. It's convenient from a programmer's perspective because only one module needs to be required instead of several. Instead of calling a generic create method, a method named for the object can be called. For example:

```
var d1 = factory.Dog( { name: "Rover" } );
var d2 = factory.Dog( { name: "Spot" } );
var c1 = factory.Cat( { name: "Kitty" } );
```

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir combine
$ cd combine
$ touch index.js pet.js
```

Edit **pet.js**, add the content below and save it:

```
"use strict";

var petFactory = (spec) => {

  spec = spec || {};
  var m_name = spec.name || "default";

  var obj = {
    info: function() {
      console.log( "Info: ", this.name, "is a", this.type );
    }
  };
```

```
    Object.defineProperties( obj, {
        “type”: {
            writable: false,
            value: “pet”,
            enumerable: true,
            configurable: true  // So it can be redefined
        },
        “name”: {
            get: () => { return m_name; },
            set: (newValue) => { m_name = newValue; },
            enumerable: true
        }
    });

    // Can’t seal or else can’t be extended
    // Object.seal(obj);

    return obj;
};

var dogFactory = (spec) => {

    var obj = petFactory(spec);

    Object.defineProperties( obj, {
        “type”: {
            writable: false,
            value: “dog”,
            enumerable: true
        }
    });
```

```
    Object.seal(obj);

    return obj;
};

var catFactory = (spec) => {

    var obj = petFactory(spec);

    Object.defineProperties( obj, {
        “type”: {
            writable: false,
            value: “cat”,
            enumerable: true
        }
    });

    Object.seal(obj);

    return obj;
};

module.exports = {
    Pet: petFactory,
    Dog: dogFactory,
    Cat: catFactory
};
```

A pet factory function is defined that when called will return an object that contains an **info** method, a **type** property and a **name** property. The pet object type property is a read-only property that returns the string “pet.”

A dog factory and a cat factory are derived from the pet factory. Each overrides the type

property to return a hard-coded string for their type (“dog” or “cat”).

The three factory functions are exported from the modules as named exports (**Pet**, **Dog**, **Cat**).

Edit **index.js**, add the content below and save it:

```
“use strict”;

var factory = require(‘./pet.js’);

var pets = [
   factory.Pet( { name: “Zeus” }),
   factory.Dog( { name: “Rover” }),
   factory.Dog( { name: “Spot” }),
   factory.Cat( { name: “Kitty” })
];

pets.forEach( pet => {
   pet.info();
});
```

The require module is called to assign a pet module to a factory variable. An array is initialized by calling the Pet, Dog and Cat named exports on the pet factory.

The array is enumerated to call the **info** method on each object.

At the command line, run the file:

```
$ node index.js
```

The output should be:

Info: Zeus is a pet

Info: Rover is a dog

Info: Spot is a dog

Info: Kitty is a cat

Overriding Methods

Besides properties, methods can also be overidden.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir override
$ cd override
$ touch index.js hello.js yo.js
```

Edit **hello.js**, add the content below and save it:

```
“use strict”;

module.exports.create = () => {

    var obj = Object.assign({}, {
        hello: () => {
            console.log(“Hello World!”);
        }
    });

    Object.seal(obj);

    return obj;
};
```

A module is defined with a factory create method. The method returns a new object that has one method assigned to it (hello). Even though the object is sealed, that won’t affect the ability to override an existing method. That would be a problem if there was an attempt to add a new method that doesn’t exist.

Edit **yo.js**, add the content below and save it:

```
"use strict";

var helloFactory = require('./hello');

module.exports = () => {

    var obj = helloFactory.create();

    obj = Object.assign(obj, {
        hello: () => {
            console.log("Yo world.");
        }
    });

    Object.seal(obj);

    return obj;
};
```

The hello factory method is assigned to the variable (**helloFactory**). The hello factory create method is called to assign a new object to the variable (**obj**). The new object is assigned a new definition of the hello method, overriding the method in the base class. Even though the base object is sealed, existing methods can be redefined. An attempt to add a new method would result in an error.

Edit **index.js**, add the content below and save it:

```
"use strict";

var factory = require("./yo.js");

var yo = factory();
```

yo.hello();

The derived module (**yo**) is required and assigned to a factory variable. The factory function is called to assign a new object to a new variable (**yo**). The hello method is called on the new object to demonstrate that the overridden method is called.

At the command line, run the file:

$ node index.js

The output should be:

Yo world.

7

Callbacks

Callbacks provide a way for node.js to operate asynchronously. In other words, callbacks are a way to write functions that don’t block the rest of the program. For example, reading a file can be slow. Instead of waiting for the disk operation, the main program can continue on. When the file operation is finished, the callback can run.

If your module accesses packages with callbacks, you should integrate those callbacks into your module.

This chapter covers:

- Wrapping callbacks
- Defining callbacks

Wrapping Callbacks

When a method wraps a function with a callback, it should define a callback that is passed to the method.

For example:

```
read: function( param, callback ) {
    …
    fs.readFile( file, options, callback );
}
```

If a function needs to return an error, it should use the callback and return the error as the first and only parameter. The callback should be called within a function called by **process.nextTick** . For example:

```
if(!file) {
    process.nextTick( function() {
        callback( new Error('A file parameter is required') );
    });
} else {
    fs.readFile( file, options, callback );
}
```

To demonstrate, create a module that wraps a call to **fs.readFile**.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir callback-101
$ cd callback-101
$ touch index.js call-wrap.js
```

Edit **call-wrap.js**, add the code below, then save it:

```
"use strict";

var fs = require('fs');

module.exports.create = () => {

    obj = {};

    Object.assign(obj, {
        read: function( param, callback ) {
            param = param || {};
            var file = param.file,
                options = param.options || "utf8";
            if(!file) {
                process.nextTick( function() {
                    callback( new Error('A file parameter is required') );
                });
            } else {
                fs.readFile( file, options, callback );
            }
        }
    });

    return obj;
};
```

The built-in package **fs** is required and assigned to a variable of the same name. An empty object (**obj**) is declared. This is what will be expanded and returned from the **create** method.

The **create** method defines an object with one method named **read**. The read method is a wrapper for **fs.readFile** which is an asynchronous method to read a file. The read method

takes two parameters. The first is a **param** object that should contain named parameters (**file** and optionally, **options**). The second is the callback.

If the file parameter is not set, the callback will be called by **process.nextTick** with a new error. The nextTick call keeps the call from blocking. If you've ever used **setTimer**, it has similar behavior. Otherwise, the **fs.readFile** method is called with the file, options and callback parameters.

Finally, the **create** method will return the new object when called.

Edit **index.js**, add the code below, then save it:

```
"use strict";

var factory = require('./call-wrap'),
    wrapper = factory.create();

var params = {
    file: './foo.txt'
};

wrapper.read( params, function( err, contents ) {
    if( err ) {
        console.error(err);
    } else {
        console.log( "\n—\n" + contents + "\n==" );
    }
});

console.log('*** Program Ended ***');
```

The **factory** variable is assigned with an instance of the **call-wrap** module. The **wrapper** variable is assigned the object returned by the factory (module) **create** method.

The params object is defined with one parameter named **file**.

The returned object's (**wrapper's**) **read** method is called. It takes two arguments, the **params** object and an error-first callback. The callback is called and if **err** is set, the error is echoed to the screen. Otherwise, the contents returned by the callback is echoed to the screen.

At the end of the program, a message is logged to the screen.

You may notice that **foo.txt** has not been created yet. That's so an error condition can be demonstrated first.

At the command line, run the file:

$ node index.js

The output should be:

***** Program Ended *****

{ [Error: ENOENT: no such file or directory, open './foo.txt'] errno: -2, code: 'ENOENT', syscall: 'open', path: './foo.txt' }

Notice that the message at the end of the source file was called before that error message from the callback. This demonstrates that the code was non-blocking and used the callback effectively.

The error message was logged because foo.txt has not been created yet. To create the file, type the following at the command line:

$ echo "One man's magic is another man's engineering." > foo.txt

The quote is from Robert A. Heinlein.

At the command line, run the file again:

$ node index.js

The output should now be:

***** Program Ended *****

—

One man’s magic is another man’s engineering.

==

Notice again that the message at the end of the program was called before the callback echoed the contents of the file.

There is one more thing to test: the error handler for when the file parameter is not passed to the read method.

To comment out the **file** parameter: modify **index.js**, make the change below in **bold** and save the file.

```
var params = {
   // file: ‘./foo.txt’
};
```

Now run the command again.

$ node index.js

The output should now be:

***** Program Ended *****

[Error: A file parameter is required]

The error message was echoed and in the proper, non-blocking sequence. If **process.nextTick** had not been used, the error would have been echoed before the end of the program message.

Defining Callbacks

When a custom callback is defined, it should use the following rules:

- Use an error-first callback.
- On success, the first callback parameter (error) will be null.
- On failure, the first callback parameter will contain an error object.
- On success or failure, the callback and any slow process should be called within a function called by **process.nextTick**. That is so the callback will be non-blocking.

To setup your project, open up a terminal window and do the following:

$ cd javascript

$ mkdir callback-102

$ cd callback-102

$ touch index.js call-def.js

Edit **call-def.js**, add the code below, then save it:

```
“use strict”;

var obj = {};

module.exports.create = () => {

   Object.assign(obj, {
      calc: function( param, callback ) {
         param = param || {};
         var limit = param.limit,
            err = null;
         if(limit === undefined) {
            err = new Error(‘A limit parameter is required’);
         } else if(typeof limit != “number”) {
            console.log(typeof limit);
```

```
            err = new Error('A limit parameter must be a number');
        }
        if( err ) {
            process.nextTick( function() {
                callback( err );
            });
        } else {
            process.nextTick( function() {
                function fibonacci(num) {
                    if (num <= 1) return 1;
                    return fibonacci(num - 1) + fibonacci(num - 2);
                }
                var result = {
                    Fn: limit,
                    fibonacci: fibonacci(limit),
                };
                callback( null, result );
            });
        }
    }
});

return obj;
};
```

A factory object is defined whose create method returns an object. The returned object contains a **calc** method. The calc method requires two parameters. The first is an object containing a named param (**limit**). The second is an error-first callback. On success, the callback will return an object that contains two values: **Fn** which is the fibonacci number, and **fibonacci**, which is the result for the number.

If an error occurs, the error is passed back to the calling function via the callback.

Edit **index.js**, add the code below, then save it:

```
"use strict";

var factory = require('./call-def'),
    calculator = factory.create();

for(var i = 0; i < 10; i++ ) {

    var params = {
        limit: i
    };

    calculator.calc( params, function( err, result ) {
        if( err ) {
            console.error(err);
        } else {
            console.log( result );
        }
    });
}

console.log('*** Program Ended ***');
```

The **factory** variable is assigned an instance of the object exported from the **call-def** module. The **calculator** variable is assigned the object returned from the **create** call.

A **for** loop calls the **calculator.calc** method for the values 0 through 9. The method takes two parameters. The first is an object containing the named parameter (**limit**) and the second is an **error first callback**. The callback will either be called with an error (**err**) or a null for the error (indicating no error) and a **result** object.

The result object will contain two properties: **Fn**, representing the value passed in (**limit**) and the **fibonacci** result for that value. On success, the result object will be echoed to the

screen.

At the command line, run the file:

```
$ node index.js
```

The output should be:

```
*** Program Ended ***
{ Fn: 0, fibonacci: 1 }
{ Fn: 1, fibonacci: 1 }
{ Fn: 2, fibonacci: 2 }
{ Fn: 3, fibonacci: 3 }
{ Fn: 4, fibonacci: 5 }
{ Fn: 5, fibonacci: 8 }
{ Fn: 6, fibonacci: 13 }
{ Fn: 7, fibonacci: 21 }
{ Fn: 8, fibonacci: 34 }
{ Fn: 9, fibonacci: 55 }
```

8

Promises

When you have callbacks calling callbacks calling callbacks things can get out of hand. The code becomes unreadable and maybe even unmaintainable. Promises are an attempt to offer a cleaner alternative.

This chapter covers:

- Wrapping promises
- Defining promises

Wrapping Promises

Callbacks can be wrapped within promises. The callback error can be passed back in a Promise **reject** call. On success any returned data can be passed back in a Promise **resolve** call. For example:

```
return new Promise( (resolve, reject) => {
   fs.readFile( file, options, (err, content) => {
      if(err) {
         return reject(err);
      }
      resolve(content);
   });
});
```

The Promise will then use the results to manage how **then** and **catch** methods will be called. For example:

```
.then((content) => console.log(content))
.catch((err) => console.error(err));
```

To demonstrate, create a project that wraps **fs.readFile** in a Promise.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir promise-101
$ cd promise-101
$ touch index.js promise-wrap.js
```

Edit **promise-wrap.js**, add the code below, then save it:

```
“use strict”;
```

```
var fs = require('fs'),
    obj = {};

module.exports.create = () => {

    Object.assign(obj, {

        read: function(param) {
            param = param || {};
            var file = param.file,
                options = param.options || "utf8";

            return new Promise( (resolve, reject) => {
                fs.readFile( file, options, (err, content) => {
                    if(err) {
                        return reject(err);
                    }
                    resolve(content);
                });
            });
        }
    });

    return obj;
};
```

A factory object is defined with a **create** method. The create method returns an object that has one method (**read**) that returns a **Promise** object. The Promise wraps the call to **fs.readFile** and manages the callback to convert it into a promise.

Edit **index.js**, add the code below, then save it:

```
"use strict";
```

```
var factory = require('./promise-wrap'),
    wrapper = factory.create();

Promise.all([
    wrapper.read( { file: './foo1.txt' } ),
    wrapper.read( { file: './foo2.txt' } )
])
.then((content) => console.log(content))
.catch((err) => console.error(err));

console.log('*** Program Ended ***');
```

The **factory** variable is assigned with an instance of the **promise-wrap** module. The **wrapper** variable is assigned the object returned by the factory (module) **create** method.

The **read** method requires a **params** object with one parameter named **file**.

The **wrapper's read** method is called twice within a **Promise.all** array. The read method only takes one argument, the **params** object and no callback. On success of all the operations, the **then** method is called. If any call to **wrapper.read** results in an error, the **catch** method is called instead.

At the end of the program a message is logged to the screen.

You may notice that **foo1.txt** and **foo2.txt** have not been created yet. That's so an error condition can be demonstrated first.

At the command line, run the file:

```
$ node index.js
```

The output should be:

***** Program Ended *****

{ [Error: ENOENT: no such file or directory, open ‘./foo1.txt’] errno: -2, code: ‘ENOENT’, syscall: ‘open’, path: ‘./foo1.txt’ }

The error message was logged because foo1.txt has not been created yet. To create the file, type the following at the command line:

$ echo “Life is trying things to see if they work.” > foo1.txt

The quote above is from Ray Bradbury.

At the command line, run the file again:

$ node index.js

The output will still contain an error for foo2.txt instead of foo1.txt:

***** Program Ended *****

{ [Error: ENOENT: no such file or directory, open ‘./foo2.txt’] errno: -2, code: ‘ENOENT’, syscall: ‘open’, path: ‘./foo2.txt’ }

Create foo2.txt with this quote from Pablo Picasso:

$ echo “Everything you can imagine is real.” > foo2.txt

At the command line, run the file again:

$ node index.js

The output should be:

***** Program Ended *****

[‘Life is trying things to see if they work.\n’,

‘Everything you can imagine is real\n’]

Notice that the output is not two individual operations. It is an array.

To test the error condition for the missing named parameter, modify the **Promise.all** array by adding the line in **bold** and saving the file:

```
Promise.all([
  wrapper.read(),
  wrapper.read( { file: ‘./foo1.txt’ } ),
  wrapper.read( { file: ‘./foo2.txt’ } )
])
```

At the command line, run the file again:

$ node index.js

The output should be:

***** Program Ended *****

[TypeError: path must be a string]

Once you’ve demonstrated the error, you can remove or comment that new line out.

Defining Promises

A custom promise can be defined that operates asynchronously just like a callback. On error, call the Promise **reject** method with a new Error object. On success, call the Promise **resolve** method passing in a result object.

To setup your project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir promise-102
$ cd promise-102
$ touch index.js
$ touch promise-def.js
```

Edit **promise-def.js**, add the code below, then save it:

```
“use strict”;

module.exports.create = () => {

   var obj = {};

   Object.assign(obj, {
      calc: function( param ) {

         param = param || {};
         var limit = param.limit,
            err = null;

         return new Promise( (resolve, reject) => {

            if(limit === undefined) {
               err = new Error(‘A limit parameter is required’);
            } else if(typeof limit != “number”) {
```

```
            console.log(typeof limit);
            err = new Error('A limit parameter must be a number');
        }

        if(err) {
            return reject(err);
        }

        function fibonacci(num) {
            if (num <= 1) return 1;
            return fibonacci(num - 1) + fibonacci(num - 2);
        }

        var result = {
            Fn: limit,
            fibonacci: fibonacci(limit),
        };

        resolve(result);
    });
  }
});

return obj;
};
```

A factory object is defined with a **create** method. The create method returns an object that has one method (**calc**) that returns a **Promise** object. The Promise wraps the code to calculate a fibonacci number. On error, the **Promise reject** method will be called and passed the error. Otherwise, the result we be passed to the **resolve** function.

Edit **index.js**, add the code below, then save it:

```
“use strict”;

var factory = require(‘./promise-def’),
   calculator = factory.create();

Promise.all([
   calculator.calc( { limit: 1 } ),
   calculator.calc( { limit: 2 } ),
   calculator.calc( { limit: 3 } ),
   calculator.calc( { limit: 4 } ),
   calculator.calc( { limit: 5 } ),
   calculator.calc( { limit: 6 } ),
   calculator.calc( { limit: 7 } ),
   calculator.calc( { limit: 8 } ),
   calculator.calc( { limit: 9 } ),
])
.then((content) => console.log(content))
.catch((err) => console.error(err));

console.log(‘*** Program Ended ***’);
```

The **factory** variable is assigned an instance of the object exported from the **promise-def** module. The **calculator** variable is assigned the object returned from the **create** call.

The **calculator’s calc** method is called several times within a **Promise.all** array. The calc method only takes one argument, the **params** object and no callback. On success of all the operations, the **then** method is called. If any call to **calculator.calc** results in an error, the **catch** method is called instead.

At the command line, run the file:

```
$ node index.js
```

The output should be:

```
*** Program Ended ***
[ { Fn: 1, fibonacci: 1 },
  { Fn: 2, fibonacci: 2 },
  { Fn: 3, fibonacci: 3 },
  { Fn: 4, fibonacci: 5 },
  { Fn: 5, fibonacci: 8 },
  { Fn: 6, fibonacci: 13 },
  { Fn: 7, fibonacci: 21 },
  { Fn: 8, fibonacci: 34 },
  { Fn: 9, fibonacci: 55 } ]
```

Notice that the result is an array.

To test the error condition for the missing named parameter, modify the **Promise.all** array by adding the line in **bold** and saving the file:

```
Promise.all([
  calculator.calc(),
  calculator.calc( { limit: 1 } ),
```

At the command line, run the file again:

```
$ node index.js
```

The output should be:

```
*** Program Ended ***
[Error: A limit parameter is required]
```

Once you've demonstrated the error, you can remove or comment that new line out.

9

Unit Testing

When developing modules, you should unit test them. This chapter gives a quick overview of automating unit tests. For more details on unit testing, see my book **How to Create and Publish Node.js Packages**. You can find a link on my home page.

This chapter covers:

- The mocha test framework (mochajs)
- Creating a test project
- Initializing the project with npm
- The should assertion module (shouldjs)
- Creating a test folder
- Creating a smoke test file
- Running tests
- Testing a method
- Testing a property
- Testing exceptions

mocha

Mocha (**mochajs**) is a popular test framework. Install it through npm using a global flag (-g). That will make it available from any folder.

```
$ npm install -g mocha
```

You can find more info on mocha at http://mochajs.org.

Create a Test Project

To setup a test project, open up a terminal window and do the following:

```
$ cd javascript
$ mkdir xunit-101
$ cd xunit-101
$ touch index.js
```

Open up **index.js** in a text or JavaScript editor, add the code below, then save it:

```
“use strict”;

module.exports.create = (spec) => {

    var nameLimit = 10;

    spec = spec || {};

    var m_name = “unknown”;

    var obj = {
        hello: function() {
            console.log(
                “\nHello, my name is”, m_name,
                “\nI am a”, this.type
            );
        }
    };

    Object.defineProperties( obj, {
        “type”: {
            get: () => “pet”,
```

```
            enumerable: true
        },
        “name”: {
            get: () => m_name,
            set: function(newValue) {
                if(newValue.length > nameLimit ) {
                    throw new Error( “Name string limit:”, nameLimit );
                }
                m_name = newValue;
            },
            enumerable: true
        }
    });

    Object.seal(obj);

    obj.name = spec.name || obj.name;

    return obj;
};
```

The **module.exports** value is assigned an object containing a named function (**create**). The named function will be exported and returned when the file is called using the **require** function. The create method will return an object that contains a function called **name**. If the name function is called with no parameter, it will return the current value of **m_name**. If it is called with a value, that value will be assigned to **m_name**.

Initializing the Project with npm

To keep things simple, initializing npm has been left out of the discussion until this chapter. That's because the example projects have not required installing any third party packages locally. This project is going to require the installation of a development dependency. It can be done without initializing npm for the project. But, it will generate an error. To keep things clean, initialize npm by doing the following in the projects root folder:

$ npm init —yes

Normally, **npm init** would ask for a series of inputs to generate a **package.json** file in the root of your project. Using the **—yes** flag (that's two hyphens before the flag) bypasses the questions.

should

If you've ever done any form of automated testing, then you are probably familiar with assert statements. Your test will assert whether a condition is true of false. For example, you would want to assert that a return value is not null, otherwise fail the test.

Node.js has an assert module that you can include using **require('assert')**. But, the documentation states that it is intended for internal use only and should not be used as a general purpose library. For more information on node.js assert, visit:

https://nodejs.org/api/assert.html

The assertion module that I like to use is **should** (**shouldjs**). With that library you can do things like this:

```
var obj = factory.create({ x: 5, y: 6 });
should.exist(obj);
```

Inside a test framework, if the create method above returns a null object, the test will fail.

Install the **should** package now so that it will be part of the project. Type the following at the command line (note that the flag has two dashes at the beginning):

```
$ npm install should —save-dev
```

You may see a couple of warnings that you can ignore. This will create a new subfolder called **node_modules** under your project where the third party package is stored.

You can find more information about the **should** package on npm here:

https://www.npmjs.com/package/should

and more documentation here:

http://shouldjs.github.io

Create a Test Folder

The mocha framework expects a subfolder called **test** to be under whatever folder you are running mocha from. So create it now:

```
$ mkdir test
```

Create a Smoke Test File

Create a smoke-test file in the test folder:

```
$ touch test/smoke-test.js
```

Edit **test/smoke-test.js** and add the code below, then save it:

```
“use strict”;

var should = require(‘should’),
    modulePath = “../index”;

describe(‘module’, () => {

    var factory = null;

    beforeEach( done => {
        // Call before all tests
        delete require.cache[require.resolve(modulePath)];
        factory = require(modulePath);
        done();
    });

    it(‘should exist’, done => {
        should.exist(factory);
        done();
    });
});
```

The **should** package is required. There is no need to include a path. Node.js will look in the **node_modules** folder automatically.

The **modulePath** value is a path pointing back to the **index.js** file in the root of the project.

Think of **describe** as a wrapper for holding a suite of tests. The text parameter should identify what you are testing. The function parameter holds all the tests for the suite and supporting code.

A default **factory** variable is defined to hold a pointer to the factory function between tests.

The **beforeEach** function is an optional function that mocha will look for. It is code that is run before each test. The **done** parameter is so that the test framework can pass in a function to indicate to mocha that the **beforeEach** function is finished.

Notice at the end of the function that done is called. If it isn't called, the test suite will hang waiting for a signal that never arrives. Besides **beforeEach**, there is also **afterEach** to run cleanup code after each test. There is also **before** and **after** to run code before and after all the tests have run. If the functions aren't needed, they don't need to be included.

The **delete** call clears the **require cache**. Normally, **require** caches a pointer to a module. That's one reason why **require** called several times just points back to the same object. For testing, this should be cleared out to start fresh between each test.

The **factory** variable is set using the **require** call. The call references the **index.js** file in the root of the project.

The **it** function contains a test. The text parameter combined with the describe string should form a phrase to define what is being tested. For example: 'module should exist'. In this case, the **should** module is used to test that the factory variable has been set (should.exist(factory)). That factory variable should be set before each test in the beforeEach function.

As with **beforeEach**, the **done** function must be called at the end of the test or the test will hang.

A **describe** function can contain several **it** functions to define a test suite.

Run the Test

From the command line, run the test:

```
$ mocha
```

You should see the result for one test and it should pass:

```
module
  ✓ should exist


1 passing (9ms)
```

Test the Create Method

Now that a framework is place, additional tests can be added.

Here are some ideas for testing the create method:

- Calling create with no parameters should still return an object
- Calling create with a short name parameter should still return an object

Those are positive test cases. Negative test cases will be covered later.

Based on the list above, add these tests cases to the test file:

```
it('create with no parameters should return object', done => {
    should.exist(factory);
    var obj = factory.create();
    should.exist(obj);
    done();
});

it('create with short name parameter should return object', done => {
    should.exist(factory);
    var obj = factory.create( { name: "Foo" } );
    should.exist(obj);
    done();
});
```

At the command line, run **mocha** and the tests should pass.

Test the Name Property

Here are some ideas for testing the name property:

- The name property should be the default if it was never set in the create method call
- If the name property was the default, setting it to a new value should successfully change it to that new value
- If the name property was set with the create call, setting the name property to a new value should cause it to take on that new value

Based on the list above, add these tests cases to the test file:

```
it('name should return default value if none set', done => {
    should.exist(factory);
    var defaultName = "unknown";
    var obj = factory.create();
    obj.name.should.eql(defaultName);
    done();
});

it('name should set new value if was default', done => {
    should.exist(factory);
    var newName = "Jill";
    var obj = factory.create();
    obj.name = newName;
    obj.name.should.eql(newName);
    done();
});

it('name should set new value', done => {
    should.exist(factory);
    var testName = "John";
    var newName = "Jill";
    var obj = factory.create( { name: testName } );
```

```
    obj.name = newName;
    obj.name.should.eql(newName);
    done();
  });
```

At the command line, run **mocha** and the tests should pass.

Testing for Exceptions

To test for exceptions, they need to be wrapped in an anonymous function that is chained to **should.throw**, like this:

```
(function() {
    obj.newProperty = “test”;
}).should.throw();
```

The exception will be trapped. Even though the call will cause an exception to be thrown, the test will pass because the exception was expected.

Here are some ideas for testing exceptions:

- The object is sealed and it should not be possible to add new properties
- The create method should throw an exception if the name parameter is too long
- Setting the name should throw an exception if the new value is too long

Based on the list above, add these tests cases to the test file:

```
it(‘object is sealed’, done => {
    should.exist(factory);
    var obj = factory.create();
    (function() {
        obj.newProperty = “test”;
    }).should.throw();
    done();
});

it(‘create method should fail if name too long’, done => {
    should.exist(factory);
    var longName = “this is a very long name”;
    (function() {
        factory.create( { name: longName } )}
```

```
    ).should.throw();
    done();
});

it('name set should fail if name too long', done => {
    should.exist(factory);
    var obj = factory.create();
    var longName = "this is a very long name";
    (function() {
        obj.name = longName;
    }).should.throw();
    done();
});
```

At the command, line run **mocha** and the tests should pass.

Breaking Up Tests

When the smoke-test file becomes too large, tests can be moved to new files using the same framework. Mocha will pick up all of the ***test.js** files that it finds in the **test** folder. If subfolders under **test** folder are added, be sure to use the **—recursive** flag (two hyphens).

Wrapping Up

In this book, you learned how to break your code up into manageable modules. You should now know how to create factory functions that create useful objects. You should also know how to define those objects giving them properties and methods.

The next step will be to take those modules and roll them into a package. For more information on creating packages see my book **How to Create and Publish Node.js Packages**. You can find a link on my home page.

Please Review This Book

As an independent author, I rely on feedback and reviews to figure out how to improve my work. If you purchased this book online, please write a review and let me know what you think. I use reviews and ratings to decide if a book should be updated and what book to write next. So, if you would like to see more, please let me know!

Appendix A: JavaScript Editors

To edit JavaScript, you can use an IDE (Integrated Development Environment) such as Eclipse, which can be downloaded here: http://www.eclipse.org.

Another option is to use a popular code editor such as Sublime. It can be downloaded here: https://www.sublimetext.com.

Eclipse is free. Sublime is what's known as "nagware." It's free to use, but it will nag you to pay for a license.

Disclaimer: I have no association with either tool and won't receive any compensation if you pay for a Sublime license.

Appendix B: Install jshint

If you run into a syntax error and can’t figure out why your code is failing, run a lint checker against your code. There are many JavaScript lint checkers out there. One of the ones I use is called jshint. It can be installed globally using npm:

```
$ sudo npm install -g jshint
```

To use it with node.js and ES6, put this at the very top of your source files:

```
/*jshint node: true */
/*jshint esversion: 6 */
```

Those lines must be there or jshint will report errors for any ES6 code (like arrow functions).

At the command line, run **jshint** against the file that you are having trouble with. For example :

```
$ jshint index.js
```

Fix any errors that the tool may report.

About the Author

Mitch Allen is and independent developer, maker and tech writer. You can visit his Web site at http://mitchallen.com or follow him on Instagram as user **virtualmitch**. You can find his apps on the iTunes App Store (search for “Mitchell Allen”).

About the Editor

Priscilla Grogan is a technical editor with over 35 years experience in the software industry. She has edited many books and tested countless products throughout her career.

JavaScript Development Series

This is the second in a series of books on software development using JavaScript.

See also:

- **How to Create and Publish Node.js Packages**.

For links and information on current or future publications in the **JavaScript Development Series**, please visit:

http://mitchallen.com