Module 1

Module 1 Study Guide and Deliverables

Readings:	Brown: Chapters 1-3
Discussions:	Discussion 1 Due: Tuesday, July 10 at 6:00 AM ET
Assignments:	Assignment 1 Due: Tuesday, July 10 at 6:00 AM ET
Live Classrooms:	**Tuesday, July 3 from 8:00-10:00 PM ET**

Survey of Server-Side Languages

Objectives

By reading the lectures and textbook, participating in the discussions, and completing the assignments, you will be able to:

- Identify key Server-Side programming languages
- Understand some of the history behind server-side programming

Introduction

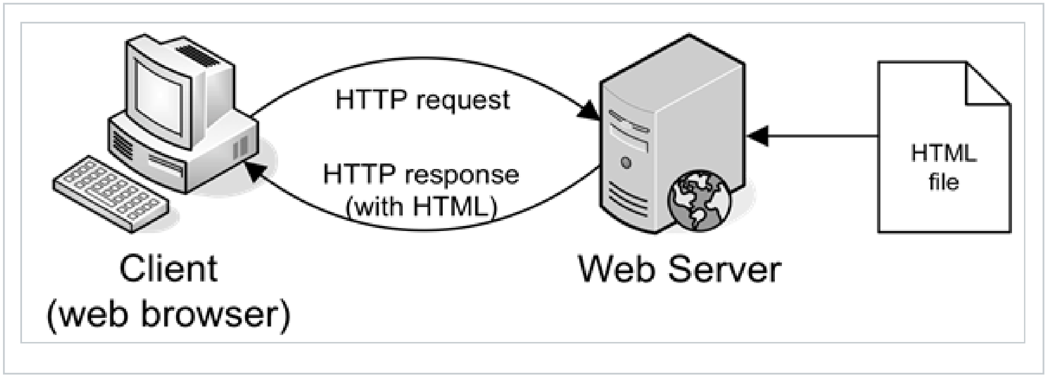
The focus of this course is on server-side web development. We will create applications that respond to the dynamic needs of each web visitor. Server-side scripts are different from client-side scripts such as JavaScript, which are run in the web browser.

Server-side programming has been around for a long time. PHP is 20 years old and other server-side scripting languages have been around for a very long time as well. Before we had languages like PHP we were able to use the Common Gateway Interface (CGI) to produce similar programs, but with limited functionality. CGI is an environment for web servers to interface with executable programs installed on the server. This allowed developers to use languages such as C, Perl, and even shell scripts in order to generate web pages in a dynamic way.

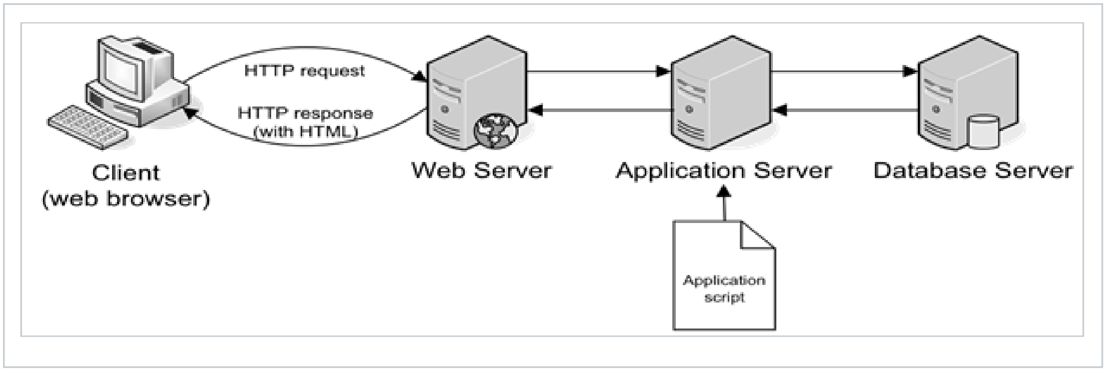
In this lecture, we will outline the request-response cycle and introduce some of the more popular server-side languages.

Request-Response Cycle

The traditional request-response cycle when dealing with client-side technologies like HTML, CSS, and JavaScript is fairly simple and doesn’t require a lot of coordinated server activities.



However, when we start using server-side languages and a database, the process becomes a bit more involved.



You can see in the second diagram, the web server is interacting with an application server (like PHP), which in turn works with a database sever to retrieve the requested data and all of this gets sent back to the web browser as HTML.

Server-Side Languages

We use server-side languages to create web applications and services. A full web application experience consists of four major components:

- Web browser
- Web server
- Server-side language(s)
- Database server(s)

There are many, many different programming languages that can be used to develop server-side applications. In fact, there are so many different ones that it can often make it difficult to choose which ones to use for development projects. In this course, we will be using PHP and Node.js to develop our own development projects and they will be our focus throughout the next six weeks of this course.

We are going to spend a little bit of time below providing an overview of today's most widely used server-side languages.

PHP

PHP was created in 1995 specifically for use in web development but it can also be used as a general purpose programming language. PHP is installed on approximately 70% of all web servers in the world and powers hundreds of millions of websites. PHP code can easily be mixed in with HTML code. The PHP code contained within web pages is parsed by the PHP interpreter residing on the server and then sends the output back as HTML to the web browser. PHP supports procedural and object oriented programing paradigms.

The PHP interpreter is powered by the Zend Engine and is free software released under it's own PHP license. It is available on nearly every operating system and platform. The current stable version of PHP is version 5.6 and version 7.0 is currently in development with a beta version released in July 2015.

You can learn more about PHP on its home page, located at http://php.net

ASP.NET

ASP.NET is an open-source server-side web application framework developed by Microsoft and designed for web development in order to produce dynamic web applications and services. Version 1.0 was released in 2002, which replaced Microsoft Active Server pages. As of July 2015, the latest version is 4.5. Web pages created in ASP.NET are known as Web Forms, which are the building blocks of a web application.

You can learn more about ASP.NET on its home page, located at http://www.asp.net

Python

Python is a general-purpose, high-level programming language. It focuses on code readability and is very popular for teaching programming to new comers. It was conceived in the late 1980's by Guido van Rossum and it is open-source software. Python can be extremely suitable for developing web applications using some of these popular frameworks:

1. Django - https://www.djangoproject.com/
2. Pyramid - http://www.pylonsproject.org/
3. Bottle - http://bottlepy.org/docs/dev/index.html
4. Tornado - http://www.tornadoweb.org/en/stable
5. Flask - http://flask.pocoo.org
6. Web2py - http://www.web2py.com

You can learn more about Python on its home page, located at https://www.python.org/

Perl

Perl is a highly capable, feature-rich programming language with over 27 years of development and over 108,000 modules available. It is currently on version 5 with version 6 under development. Perl has been used to write CGI scripts. Perl is not as popular as it used to be for web development but it is still actively used today more than most people would assume. Large projects written in Perl include cPanel, BugZilla, and Moveable Type. There are also a number of popular websites that use Perl such as Priceline, Craigslist, IMDb, Ticketmaster, Slashdot, and DuckDuckGo.

Perl is an optional component of the LAMP stack (Linux, Apache, MySQL, and PHP) where it is sometimes used instead of PHP.

You can learn more about Perl on its home page, located at https://www.perl.org

JSP

JavaServer Pages (JSP) was released by Sun Microsystems in 1999 and it allows developers to create dynamic web pages based on HTML and other document types. It is similar to PHP, but it is implemented with the Java programming language. In order to deploy and run JSP, a special web server such as Apache Tomcat is required. JSP, like Java, is maintained by Oracle today.

You can learn more about JSP on its home page, located at http://www.oracle.com/technetwork/java/javaee/jsp/index.html

Ruby on Rails

Ruby on Rails (Rails) is a web application framework that is written using the Ruby programming language. Ruby is an object oriented, general-purpose programming language developed in the 1990's by Yukihiro "Matz" Matsumoto. Rails was developed by David Heinemeier Hansson in 2004. It is a very popular web framework that has grown rapidly in the past decade, with much of that growth slowing during the last couple of years. It is a model-view-controller (MVC) framework and emphasizes the use of well-known software engineering patterns. The current version, 4.2, was released in December 2014.

You can learn more about Rails on it's home page, located at http://rubyonrails.org and more information can be found on the Ruby Programming language on its home page: https://www.ruby-lang.org/en

JavaScript

Really? JavaScript - the client-side programming language? ***Yes!***

Even though JavaScript is a very popular client-side language, it can be used to develop very powerful and efficient server-side applications. In order to do so, we need a runtime environment specifically designed to allow JavaScript to be used on the server. Node.js is that runtime environment.

Node.js is open-source and cross platform. It is event driven and non-blocking that allows for high throughput and impressive scalability. It is rapidly growing in popularity as a server-side development environment and has been used by Walmart, LinkedIn, PayPal, Microsoft, Yahoo and many other large organizations that rely on high performance web applications.

You can learn more about Node.js on its home page, located at https://nodejs.org

Summary

In this lecture, we learned some of the history of server-side web development and an overview of today's more popular server-side programming languages to include PHP, ASP.NET, Python, Perl, JSP, Ruby on Rails, and JavaScript with Node.js.

In the next module, we are going to take a deep dive into programming with PHP and use it to create dynamic web sites in conjunction with a MySQL database.

Bibliography

The PHP Group. (2015). *PHP Home Page*. Retrieved July 15, 2015, from http://php.net

Microsoft. (2015). *ASP.NET Home Page*. Retrieved July 15, 2015, from http://www.asp.net

Python Software Foundation. (2015). *Python Home Page*. Retrieved July 15, 2015, from https://www.python.org

Perl.org. (2015). *The Perl Programming Language*. Retrieved July 15, 2015, from https://www.perl.org

Oracle. (2015). *JavaServer Pages Technology*. Retrieved July 15, 2015, from http://www.oracle.com/technetwork/java/javaee/jsp/index.html

Members of the Ruby Community. (2015). *Ruby Home Page*. Retrieved July 15, 2015, from https://www.ruby-lang.org/en

David Heinemeier Hansson. (2015). *Ruby on Rails Home Page*. Retrieved July 15, 2015, from http://rubyonrails.org

Node.js Foundation. (2015). *Node.js Home Page*. Retrieved July 15, 2015, from http://nodejs.org

Core Node.js

Objectives

By reading the lectures and textbook, participating in the discussions, and completing the assignments, you will be able to:

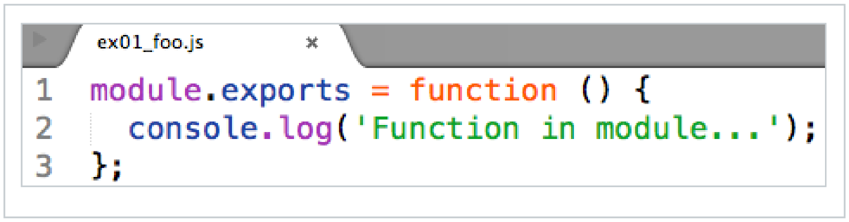
- Understand the structure of Node applications

- Develop module based applications for reusable code
- Use the core Node.js modules
- Examine how the node package manager works
- Install third-party node modules as part of your applications

Introduction

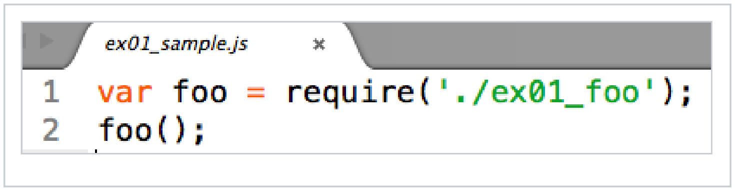
Node.js is based on the concept of modules and follows the CommonJS module specification. The JavaScript code that is executed on the server-side is organized into various modules. In the module-based approach, each JavaScript file is its own module. In each file, the module variable provides access to the current module definition. The functionality exported by the module is determined by the module.exports variable. Applications that require the module's functionality use the require function to import the module.

The following example shows a module that exports a single function.



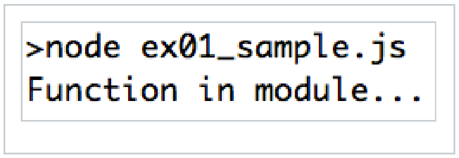
```
module.exports = function () {
  console.log('Function in module...');
};
```

In the main Node application, the module is imported into the variable, foo. Since the module is exporting a single function, the function is then invoked as shown below.



```
var foo = require('./ex01_foo');
foo();
```

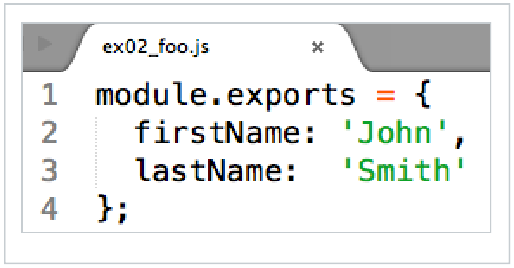
The output of the above application is shown below.



```
>node ex01_sample.js
Function in module...
```

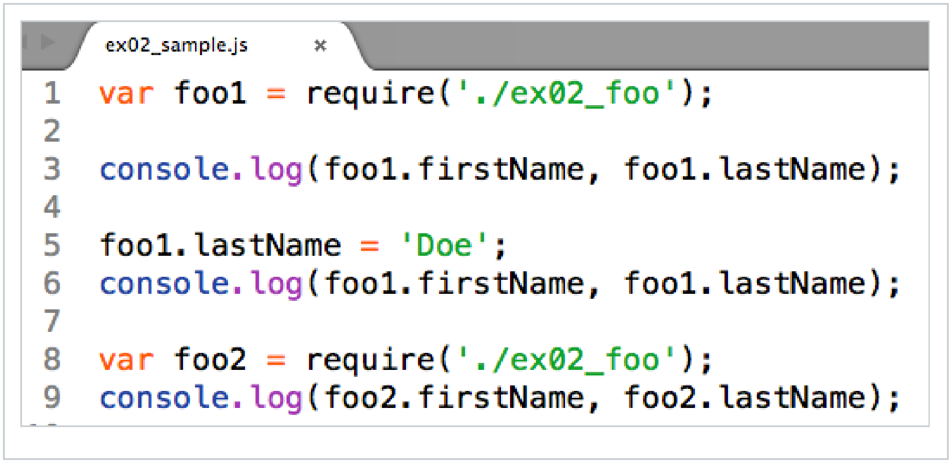
Exporting Objects and Sharing State

When a module is loaded by the require function, the module definition is cached and subsequent require calls within the application return the cached module definition. This allows the state to be shared between modules. The following example shows a JavaScript object being exported by the module.



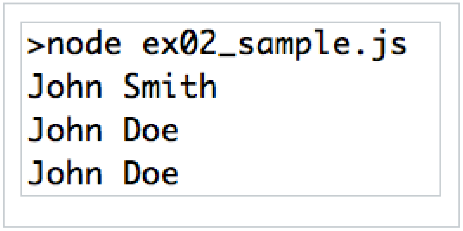
```
module.exports = {
  firstName: 'John',
  lastName:  'Smith'
};
```

In the main application, the module is first loaded into the variable, foo1. After accessing the firstName and lastName properties, the lastName value is changed. When the module is again loaded into a different variable, foo2, the changed state is visible. In a practical application, the loading of the same module will be in separate files and they will be able to share the module's state.


ex02_sample.js

```javascript
var foo1 = require('./ex02_foo');

console.log(foo1.firstName, foo1.lastName);

foo1.lastName = 'Doe';
console.log(foo1.firstName, foo1.lastName);

var foo2 = require('./ex02_foo');
console.log(foo2.firstName, foo2.lastName);
```

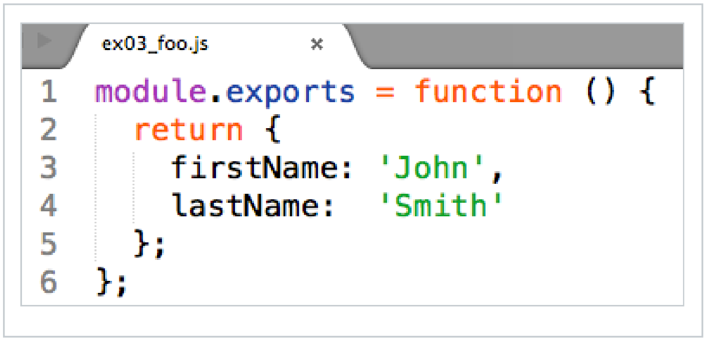
The output of the above program is shown below.



```
>node ex02_sample.js
John Smith
John Doe
John Doe
```

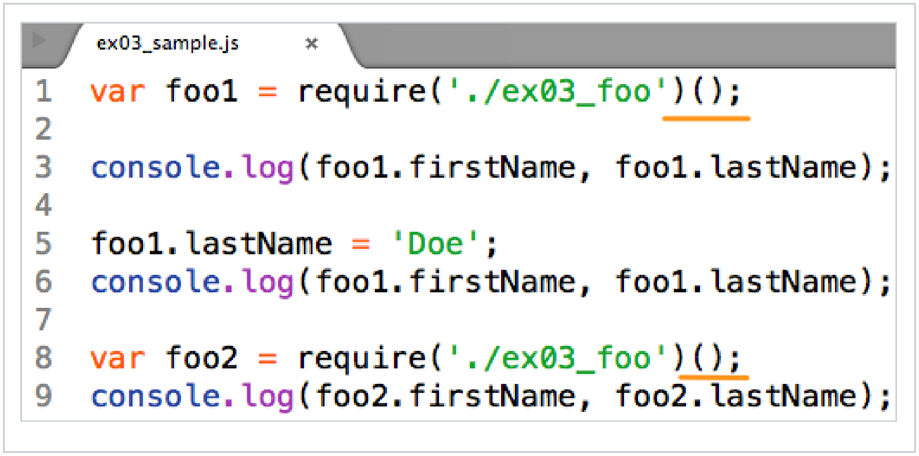
Using Object Factories

In the previous example, the same object is returned for multiple require calls. However, if a new object is required for each require call, then the module can export a function rather than the object. Invoking the exported function would then return a new object. The following example exports a single function that returns the JavaScript object, when invoked.


ex03_foo.js

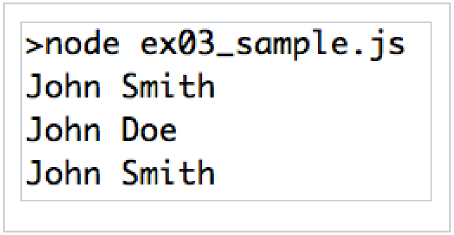
```javascript
module.exports = function () {
  return {
    firstName: 'John',
    lastName:  'Smith'
  };
};
```

In the main application, the module is loaded and then invoked to get the reference of the data into the variable, foo1. Any changes to the state of this data is only visible through the variable, foo1. If the module is loaded again and then invoked, a new object is returned.



```
var foo1 = require('./ex03_foo')();

console.log(foo1.firstName, foo1.lastName);

foo1.lastName = 'Doe';
console.log(foo1.firstName, foo1.lastName);

var foo2 = require('./ex03_foo')();
console.log(foo2.firstName, foo2.lastName);
```

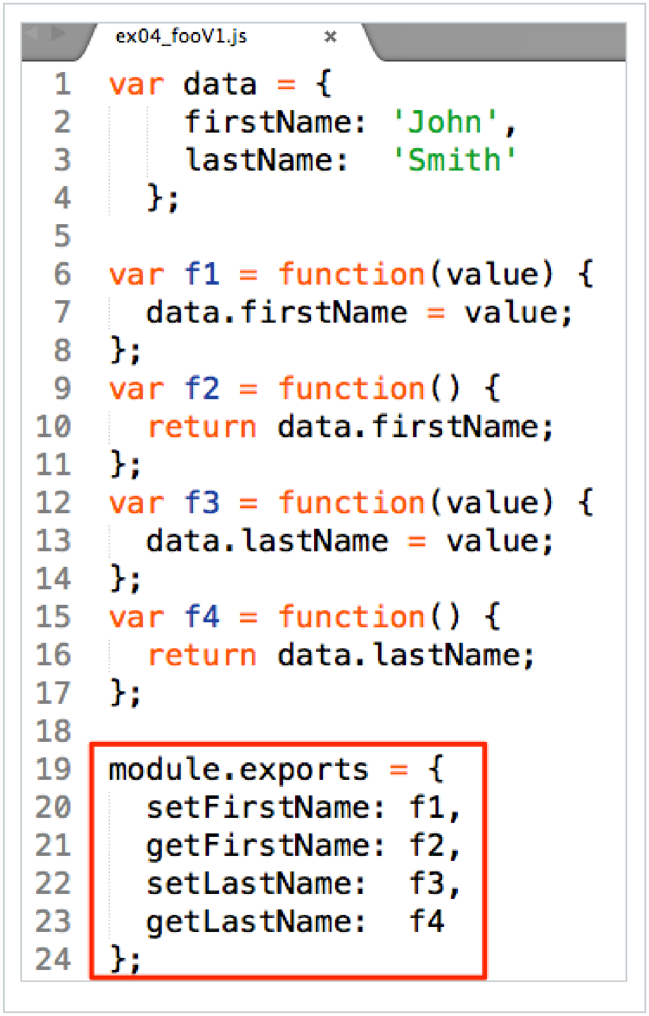
The output of the above program is shown below.



```
>node ex03_sample.js
John Smith
John Doe
John Smith
```

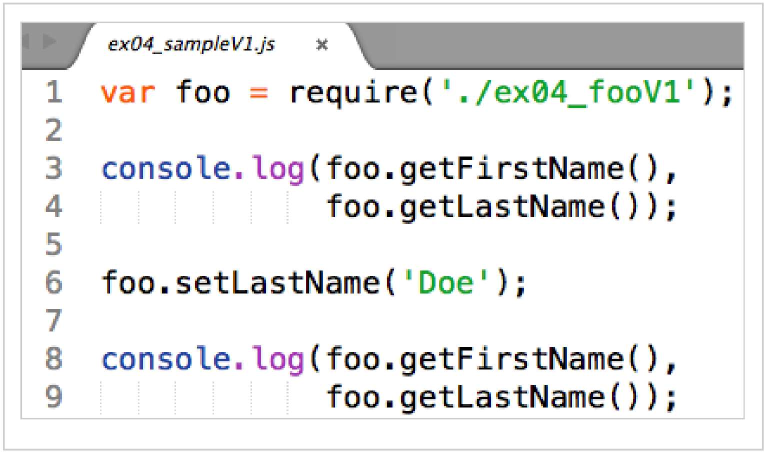
Module Exports

A typical module exports more than a single object or a single function. The following example shows the first approach of how the functionality is exported by the module. Aliases for the function definitions are used in the module exports.


ex04_fooV1.js

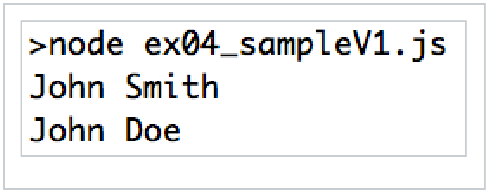
```
var data = {
    firstName: 'John',
    lastName:  'Smith'
  };

var f1 = function(value) {
  data.firstName = value;
};
var f2 = function() {
  return data.firstName;
};
var f3 = function(value) {
  data.lastName = value;
};
var f4 = function() {
  return data.lastName;
};

module.exports = {
  setFirstName: f1,
  getFirstName: f2,
  setLastName:  f3,
  getLastName:  f4
};
```

The main application uses the above module as shown below. The data from the module is not directly accessible to the application. The application can only access the data indirectly through the properties exported by the module.


ex04_sampleV1.js

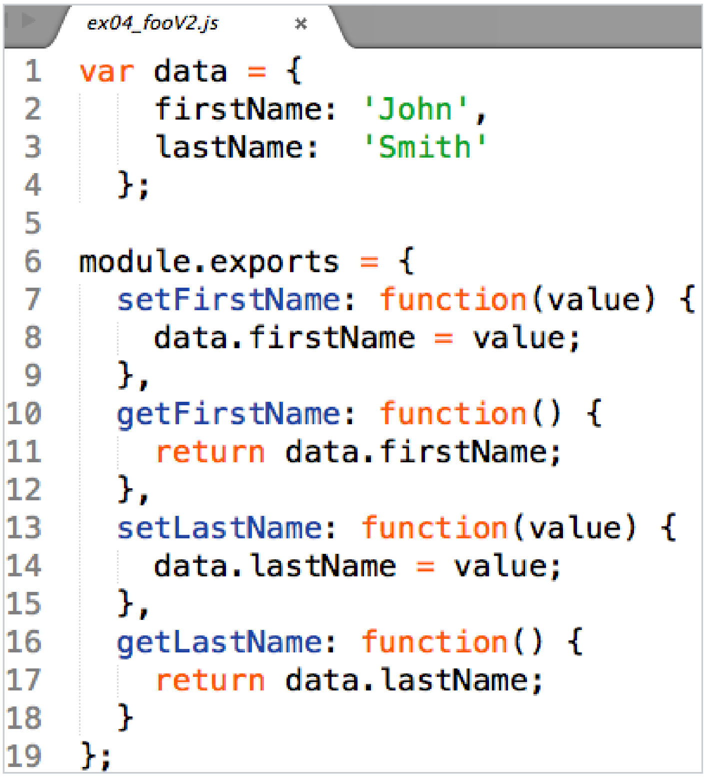
```
var foo = require('./ex04_fooV1');

console.log(foo.getFirstName(),
            foo.getLastName());

foo.setLastName('Doe');

console.log(foo.getFirstName(),
            foo.getLastName());
```

The output of the above application is shown below.



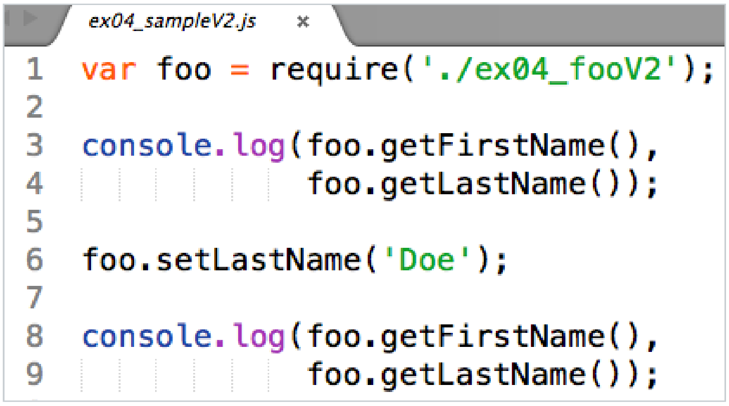
```
>node ex04_sampleV1.js
John Smith
John Doe
```

A second approach is to define the functions along with the properties of the exported JavaScript object as shown below.



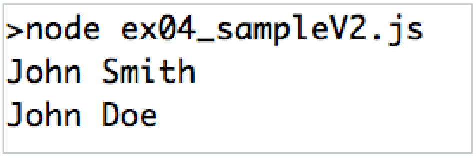
```
var data = {
    firstName: 'John',
    lastName:  'Smith'
  };

module.exports = {
  setFirstName: function(value) {
    data.firstName = value;
  },
  getFirstName: function() {
    return data.firstName;
  },
  setLastName: function(value) {
    data.lastName = value;
  },
  getLastName: function() {
    return data.lastName;
  }
};
```

The main application uses the above module as shown below. The data from the module is not directly accessible to the application. The application can only access the data indirectly through the properties exported by the module.



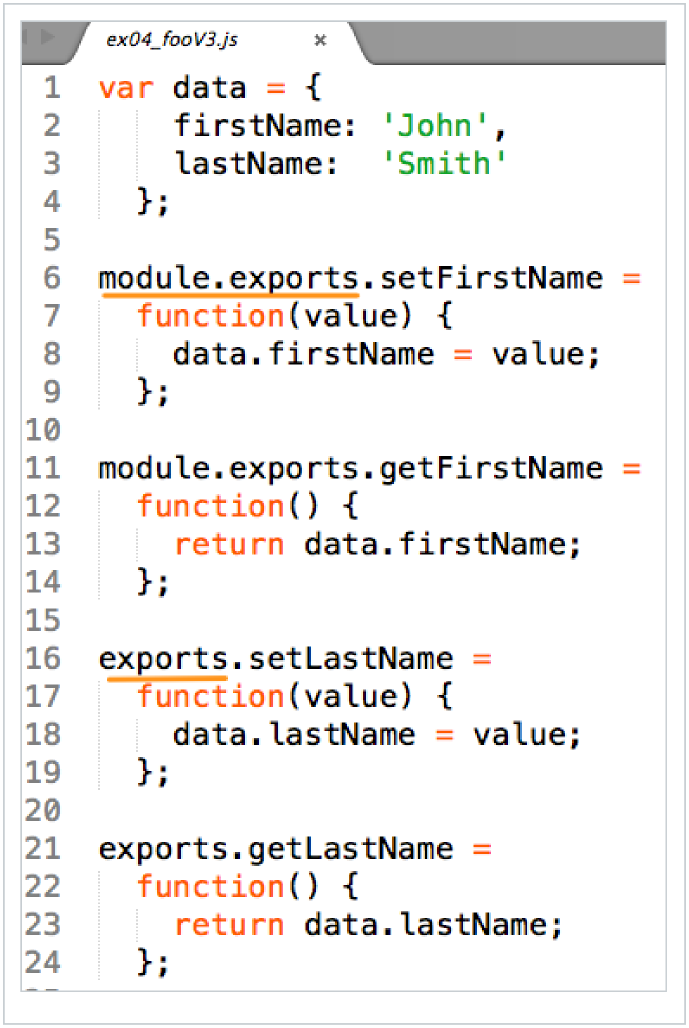
```
var foo = require('./ex04_fooV2');

console.log(foo.getFirstName(),
            foo.getLastName());

foo.setLastName('Doe');

console.log(foo.getFirstName(),
            foo.getLastName());
```

The output of the above application is shown below.



```
>node ex04_sampleV2.js
John Smith
John Doe
```

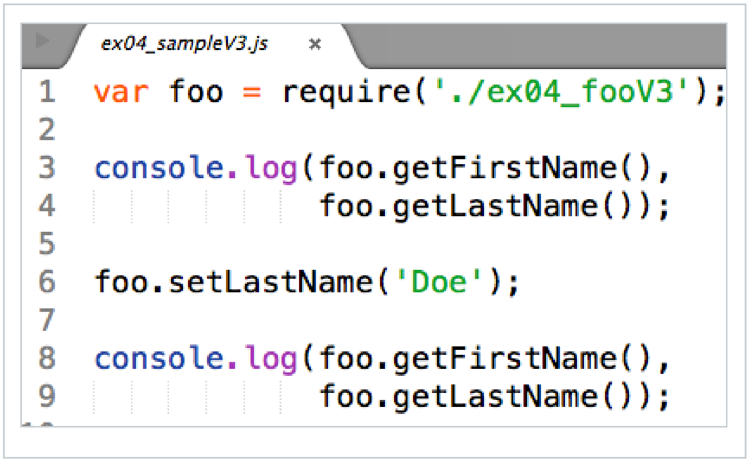
A third approach used in defining the modules is shown below. The properties are directly set using the global module.exports, or the alias for it, the global exports variable.



ex04_fooV3.js

```
var data = {
    firstName: 'John',
    lastName:  'Smith'
  };

module.exports.setFirstName =
  function(value) {
    data.firstName = value;
  };

module.exports.getFirstName =
  function() {
    return data.firstName;
  };

exports.setLastName =
  function(value) {
    data.lastName = value;
  };

exports.getLastName =
  function() {
    return data.lastName;
  };
```

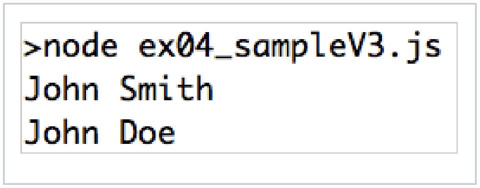
The main application uses the above module as shown below. The data from the module is not directly accessible to the application. The application can only access the data indirectly through the properties exported by the module.



ex04_sampleV3.js

```
var foo = require('./ex04_fooV3');

console.log(foo.getFirstName(),
            foo.getLastName());

foo.setLastName('Doe');

console.log(foo.getFirstName(),
            foo.getLastName());
```

The output of the above application is shown below.

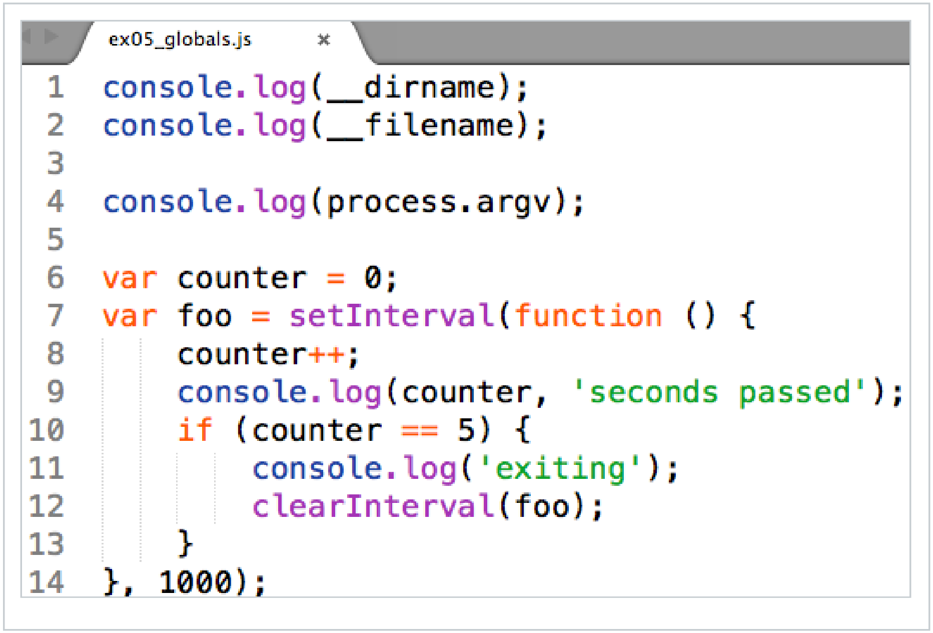


```
>node ex04_sampleV3.js
John Smith
John Doe
```

Node.js Globals

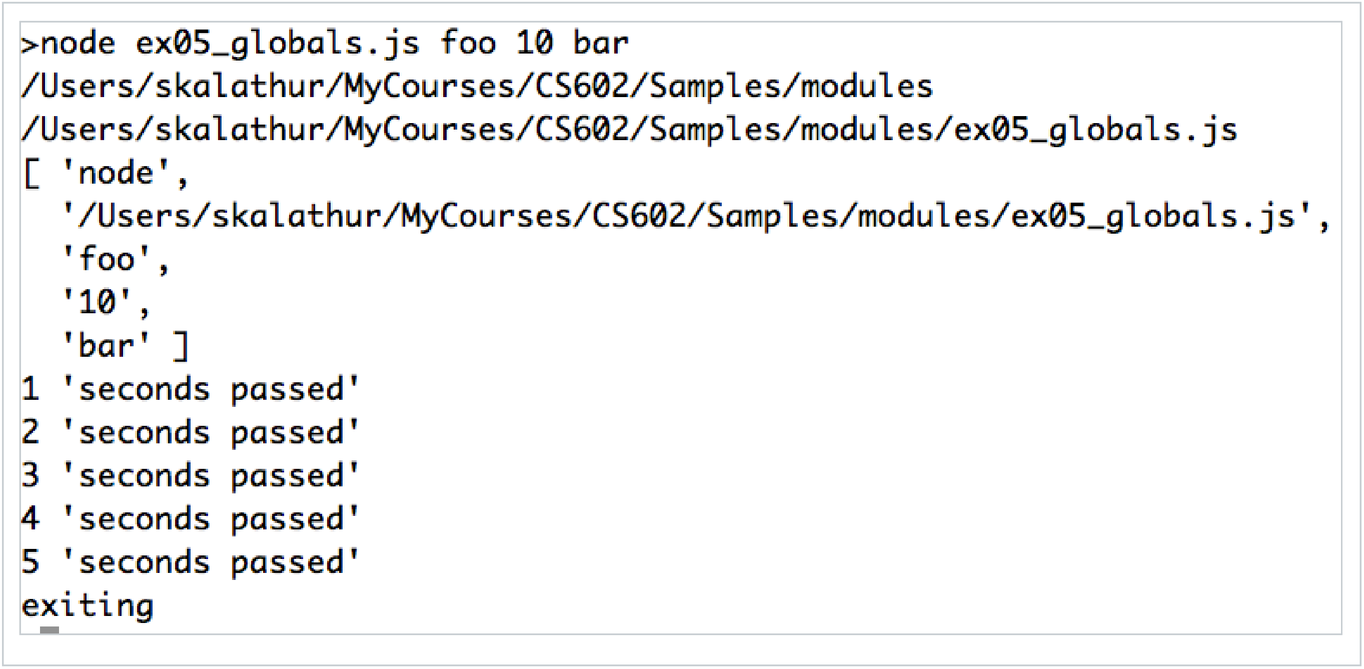
Node.js provides a good number of global variables that can be used in applications. The global variable is a JavaScript object that is first looked up for global property references. The console is one these properties.

The __dirname and __filename variables are available in each file and give the full path of the current directory and the full path of the current file. The setInterval and clearInterval properties are used to specify code that needs to executed repeatedly at the specified frequency and to stop the execution. The process property returns the process object whose argv member property can be used to access the command line arguments. The following example illustrates some of the global variables.



```
console.log(__dirname);
console.log(__filename);

console.log(process.argv);

var counter = 0;
var foo = setInterval(function () {
    counter++;
    console.log(counter, 'seconds passed');
    if (counter == 5) {
        console.log('exiting');
        clearInterval(foo);
    }
}, 1000);
```

The output of the above application invoked with the specified arguments is shown below.



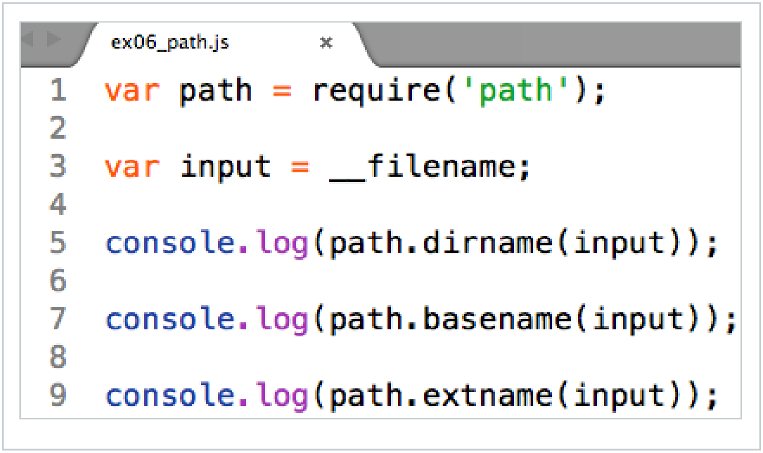
```
>node ex05_globals.js foo 10 bar
/Users/skalathur/MyCourses/CS602/Samples/modules
/Users/skalathur/MyCourses/CS602/Samples/modules/ex05_globals.js
[ 'node',
  '/Users/skalathur/MyCourses/CS602/Samples/modules/ex05_globals.js',
  'foo',
  '10',
  'bar' ]
1 'seconds passed'
2 'seconds passed'
3 'seconds passed'
4 'seconds passed'
5 'seconds passed'
exiting
```

Node.js Core Modules

Node.js provides a variety of functionality that can be reused across various applications. The name of the module is specified for the require function in order access the functionality exported by that module. The following sections show the usage of some of the core modules.

path Module

The path module provides the functions for handling and transforming file paths. The file system itself is not consulted for validity of the files. The following example shows the usage of the functions dirname, basename, and extname from the module. The dirname method returns the directory name of the specified path. The basename method returns the last portion of the specified path. The extname method returns the extension of the specified path, or the empty string if no extension is there.

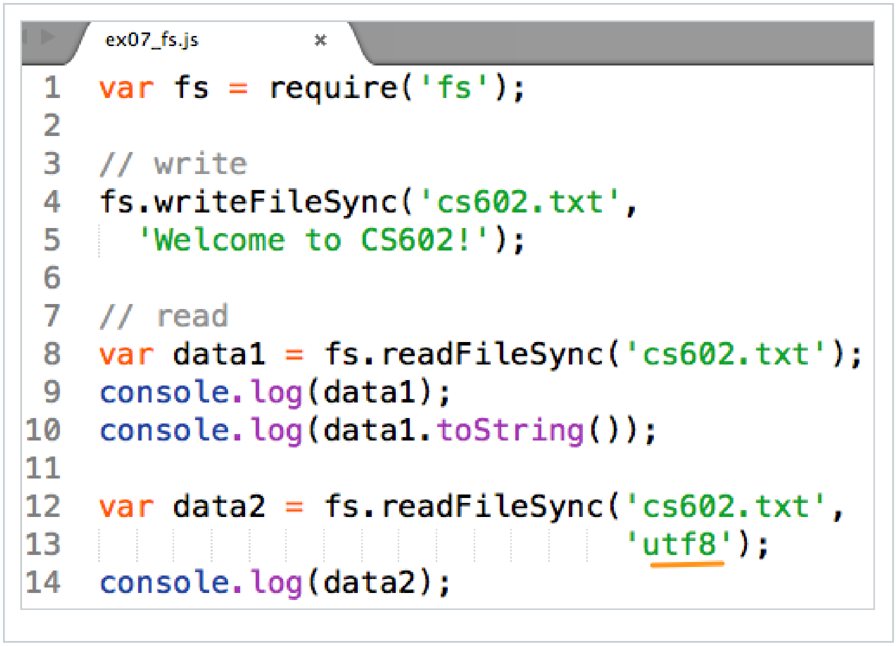


```
var path = require('path');

var input = __filename;

console.log(path.dirname(input));

console.log(path.basename(input));

console.log(path.extname(input));
```

fs module

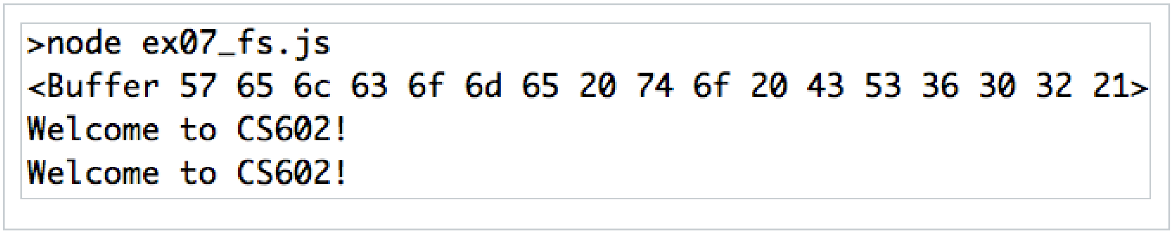
The fs module provides access to the file system. The module exports functions for reading files, writing files, deleting files, and renaming files. Most of the operations have both synchronous and asynchronous options.

The following example shows the synchronous versions of writing content to a file (writeFileSync) and reading content from the file (readFileSync). By default, the data read is returned as a Buffer object. The toString method converts the buffer to the corresponding string. The optional encoding value can also be specified to read the content directly as a string.



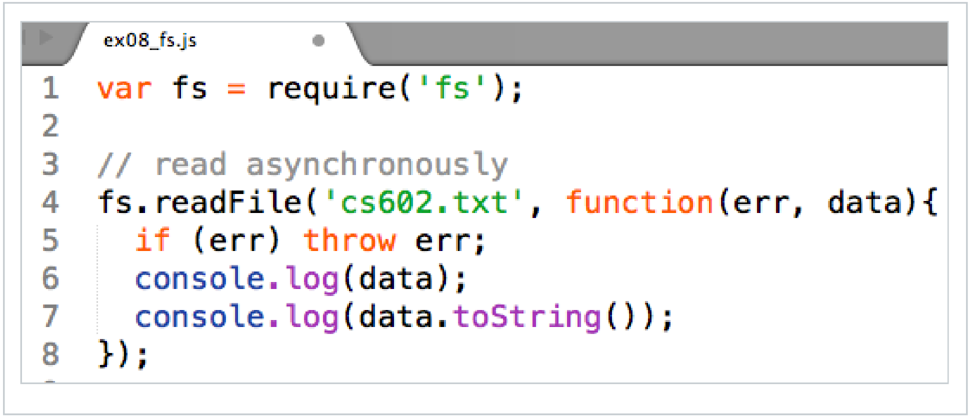
```javascript
var fs = require('fs');

// write
fs.writeFileSync('cs602.txt',
  'Welcome to CS602!');

// read
var data1 = fs.readFileSync('cs602.txt');
console.log(data1);
console.log(data1.toString());

var data2 = fs.readFileSync('cs602.txt',
                            'utf8');
console.log(data2);
```

The output of the above program is shown below.



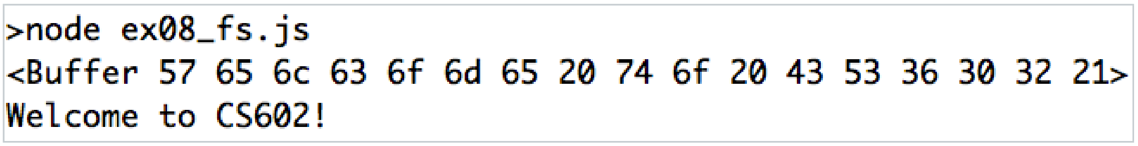
```
>node ex07_fs.js
<Buffer 57 65 6c 63 6f 6d 65 20 74 6f 20 43 53 36 30 32 21>
Welcome to CS602!
Welcome to CS602!
```

The asynchronous version of reading the file using the readFile function is shown below. The callback function is specified as the last argument of this function. When the read is complete, or if there is an error, the callback is called with the corresponding arguments. On successful read, the second argument of the callback function contains the contents read from the file.



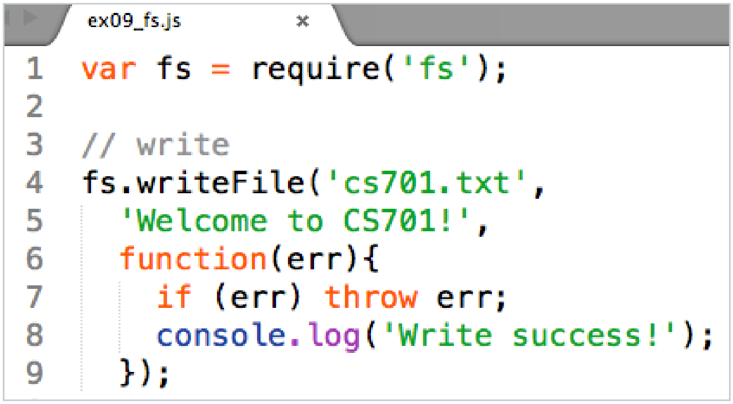
```javascript
var fs = require('fs');

// read asynchronously
fs.readFile('cs602.txt', function(err, data){
  if (err) throw err;
  console.log(data);
  console.log(data.toString());
});
```

The output of the above program is shown below.



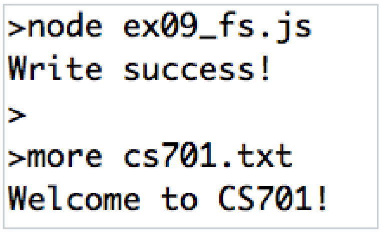
```
>node ex08_fs.js
<Buffer 57 65 6c 63 6f 6d 65 20 74 6f 20 43 53 36 30 32 21>
Welcome to CS602!
```

The asynchronous version of writing to the file using the writeFile function is shown below. The callback function is specified as the last argument of this function. When the write is complete, or if there is an error, the callback is called as shown below.



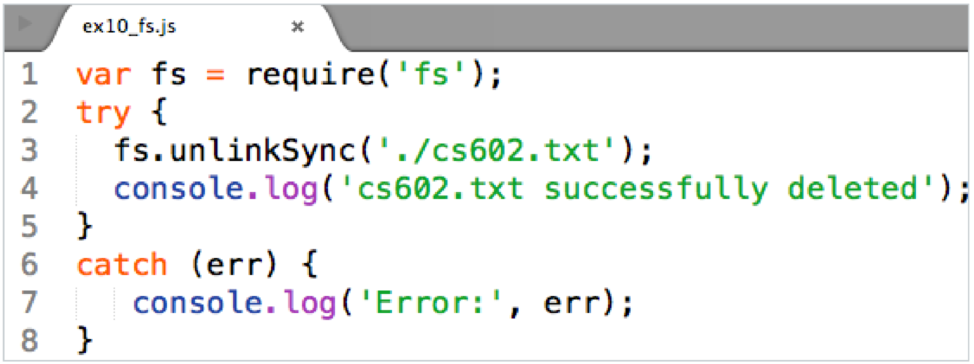
```javascript
var fs = require('fs');

// write
fs.writeFile('cs701.txt',
  'Welcome to CS701!',
  function(err){
    if (err) throw err;
    console.log('Write success!');
  });
```

The output of the above program is shown below.



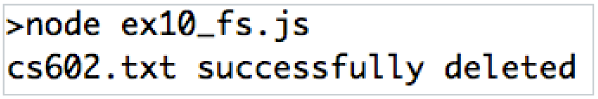
```
>node ex09_fs.js
Write success!
>
>more cs701.txt
Welcome to CS701!
```

The synchronous version of deleting the file using the unlinkSync function is shown below.



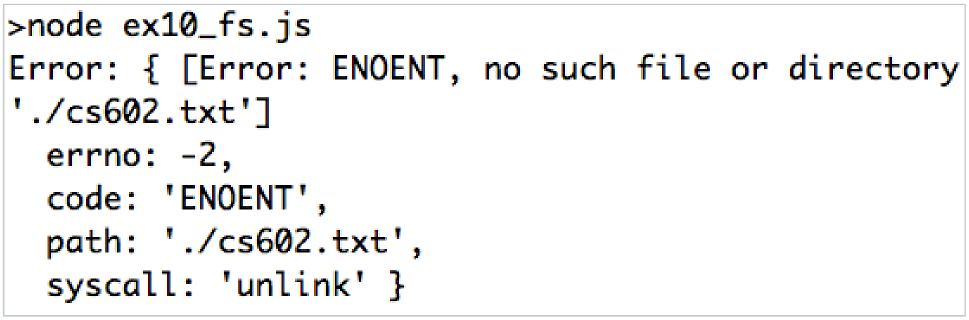
```javascript
var fs = require('fs');
try {
  fs.unlinkSync('./cs602.txt');
  console.log('cs602.txt successfully deleted');
}
catch (err) {
   console.log('Error:', err);
}
```

The output of the above program when the file exists is shown below.



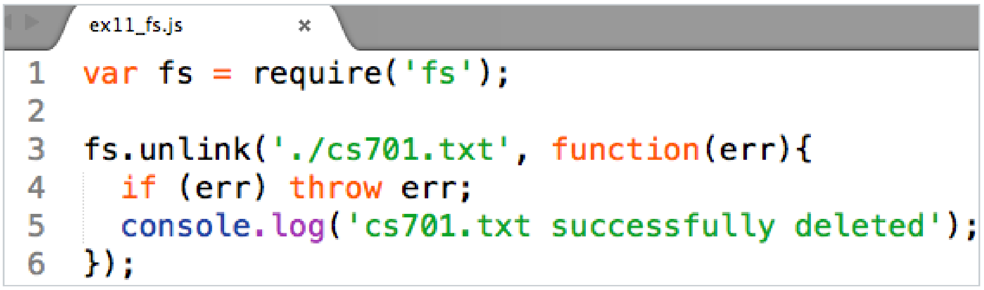
```
>node ex10_fs.js
cs602.txt successfully deleted
```

The output of the above program when the file does not exist is shown below.



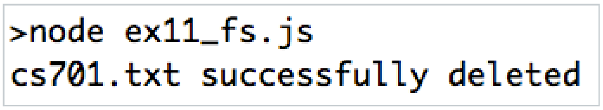
```
>node ex10_fs.js
Error: { [Error: ENOENT, no such file or directory
'./cs602.txt']
  errno: -2,
  code: 'ENOENT',
  path: './cs602.txt',
  syscall: 'unlink' }
```

The asynchronous version of deleting the file using the unlink function is shown below. The callback function is called when the delete is successful, or when an error occurs.



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

fs.unlink('./cs701.txt', function(err){
  if (err) throw err;
  console.log('cs701.txt successfully deleted');
});
```

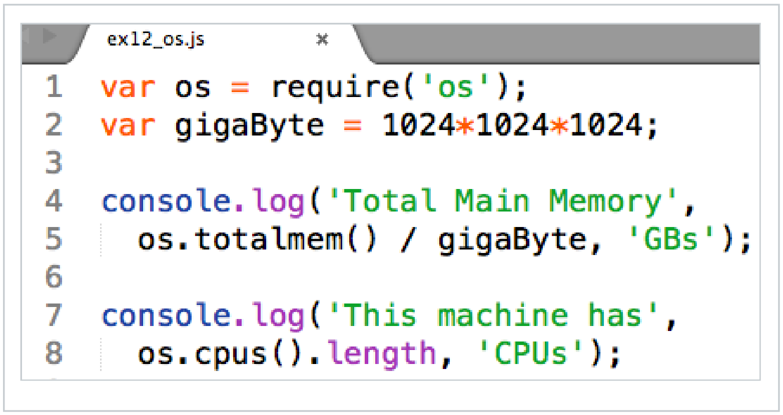
The output of the above program when the file exists is shown below.



```
>node ex11_fs.js
cs701.txt successfully deleted
```

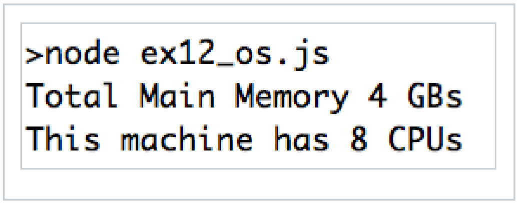
os module

The os module provides operating system related utility properties and functions. The following example shows the total memory (totalmem) and the number of CPUs (cpus) of the current system.



```
ex12_os.js
var os = require('os');
var gigaByte = 1024*1024*1024;

console.log('Total Main Memory',
  os.totalmem() / gigaByte, 'GBs');

console.log('This machine has',
  os.cpus().length, 'CPUs');
```

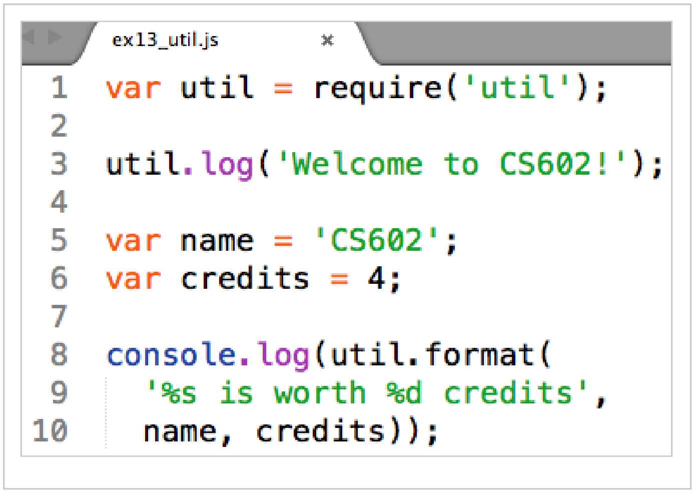
The output of the above program is shown below.



```
>node ex12_os.js
Total Main Memory 4 GBs
This machine has 8 CPUs
```

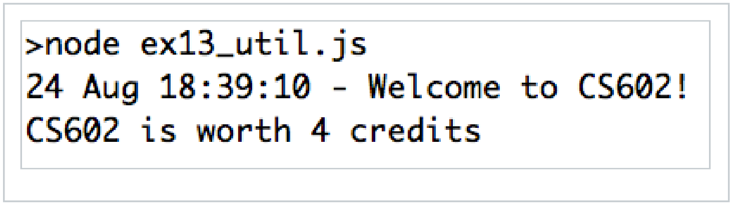
util module

The util module provides a few general purpose useful functions. The log method is used for printing the specified content to the console along with the current timestamp. The format method provides string formatting options.



```
ex13_util.js
var util = require('util');

util.log('Welcome to CS602!');

var name = 'CS602';
var credits = 4;

console.log(util.format(
  '%s is worth %d credits',
  name, credits));
```

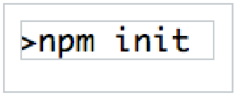
The output of the above program is shown below.



```
>node ex13_util.js
24 Aug 18:39:10 - Welcome to CS602!
CS602 is worth 4 credits
```

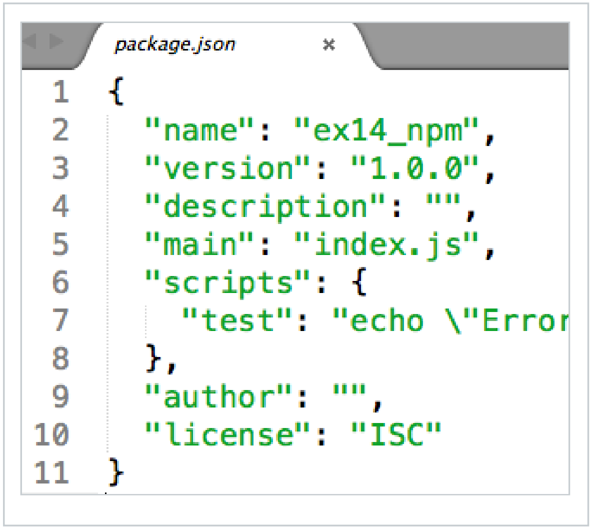
Node Package Manager (NPM)

The node package manager (npm) is part of the Node.js installation. For maintaining the Node application and its dependencies with other modules, the following command is used to initialize the application from the new application directory.



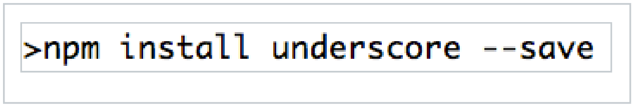
```
>npm init
```

The above command asks the user for various options. After using the defaults for those options, the following file (package.json) is created in the application directory.



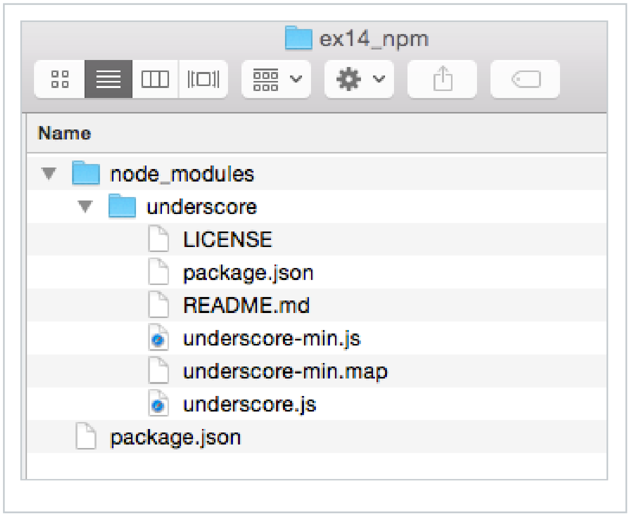
```
{
  "name": "ex14_npm",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error
  },
  "author": "",
  "license": "ISC"
}
```

Suppose the Node application requires the functionality for the node module underscore. The module is installed using the following command from the application directory.



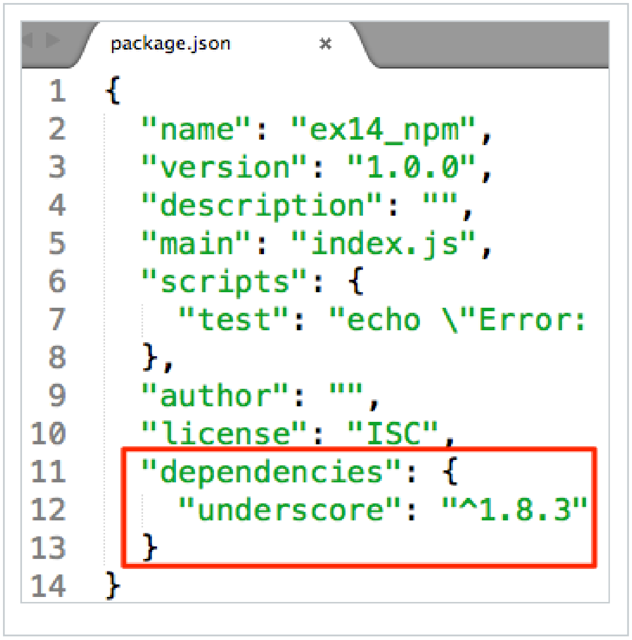
```
>npm install underscore --save
```

The save option updates the package.json file with the dependency and the version of the module that was downloaded into the node_modules directory of the application.



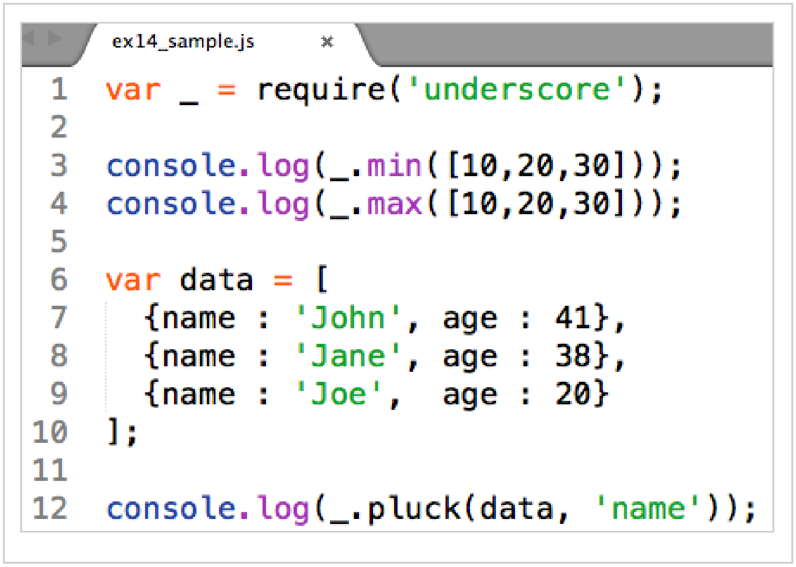
```
node_modules
    underscore
        LICENSE
        package.json
        README.md
        underscore-min.js
        underscore-min.map
        underscore.js
package.json
```

The updated package.json file is shown below.



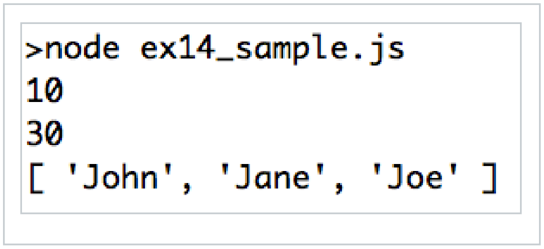
```
{
  "name": "ex14_npm",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error:
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "underscore": "^1.8.3"
  }
}
```

The Node application using the underscore module is shown below. The module provides various JavaScript functions that work on the data. The following sample uses the min, max and pluck functions from the module.



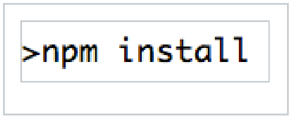
```
var _ = require('underscore');

console.log(_.min([10,20,30]));
console.log(_.max([10,20,30]));

var data = [
  {name : 'John', age : 41},
  {name : 'Jane', age : 38},
  {name : 'Joe',  age : 20}
];

console.log(_.pluck(data, 'name'));
```

The output of the above application is shown below.



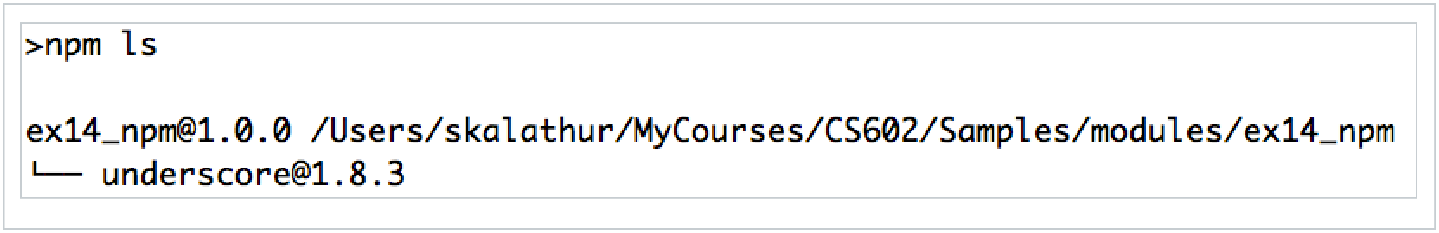
```
>node ex14_sample.js
10
30
[ 'John', 'Jane', 'Joe' ]
```

If the application is exported without the node_modules folder, the dependencies can then be automatically installed using the following command. The package.json file is consulted for the dependencies to be installed.



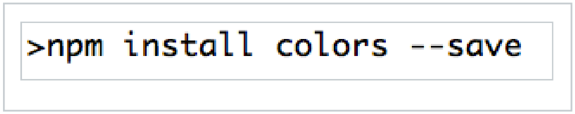
```
>npm install
```

The dependencies that were currently installed can be listed using the following command. The output shows the current application and its dependencies.



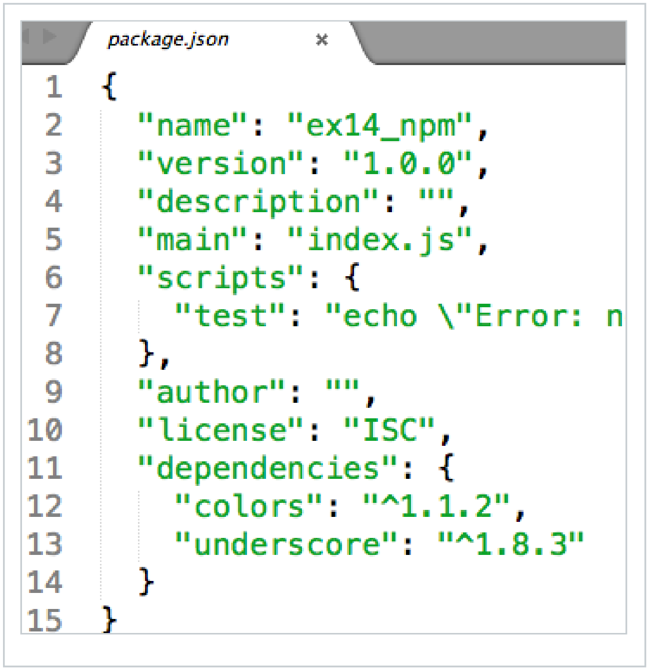
```
>npm ls

ex14_npm@1.0.0 /Users/skalathur/MyCourses/CS602/Samples/modules/ex14_npm
└── underscore@1.8.3
```

The above application is extended by installing another node module called colors. The module provides color output to the console.



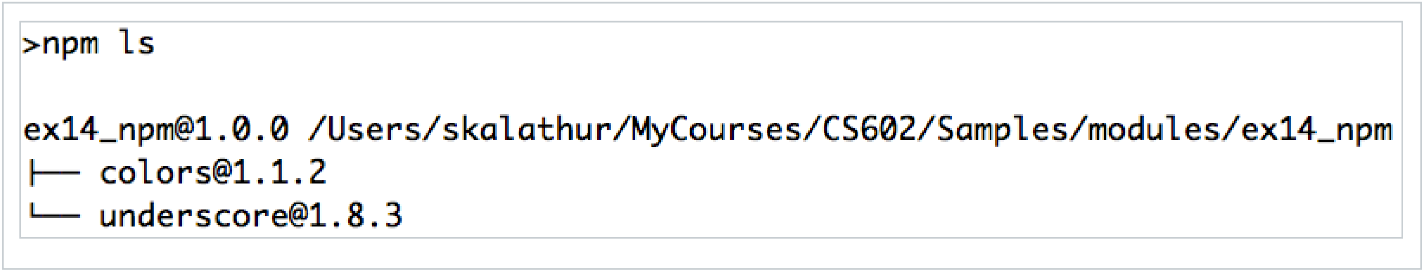
```
>npm install colors --save
```

The modified package.json file is shown below.



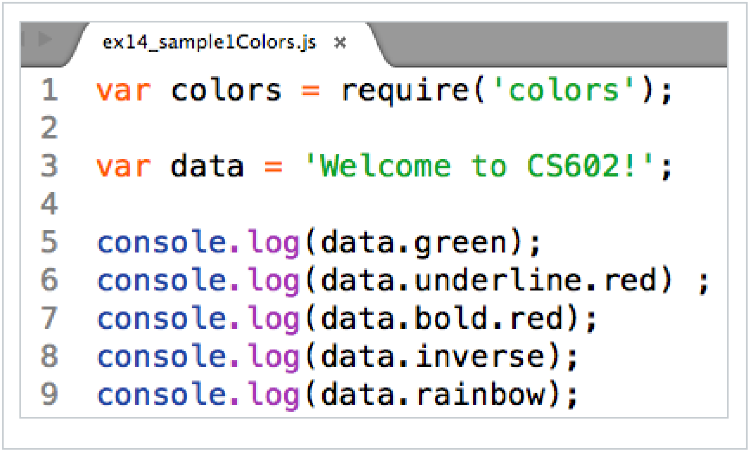
```
{
  "name": "ex14_npm",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: n
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "colors": "^1.1.2",
    "underscore": "^1.8.3"
  }
}
```

The current dependencies of the node modules is listed as shown below.



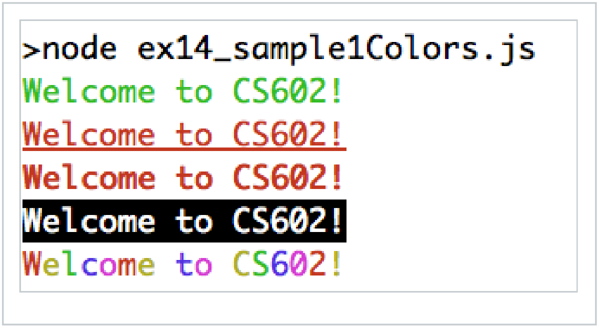
```
>npm ls

ex14_npm@1.0.0 /Users/skalathur/MyCourses/CS602/Samples/modules/ex14_npm
├── colors@1.1.2
└── underscore@1.8.3
```

The following application uses the colors module that extends the String.prototype. The string data is printed to the console with the specified color effects.



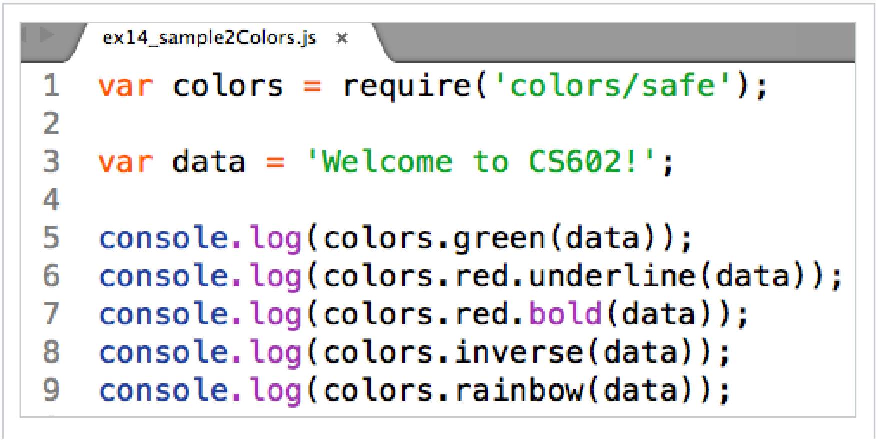
```
var colors = require('colors');

var data = 'Welcome to CS602!';

console.log(data.green);
console.log(data.underline.red) ;
console.log(data.bold.red);
console.log(data.inverse);
console.log(data.rainbow);
```

The output of the above program is shown below.



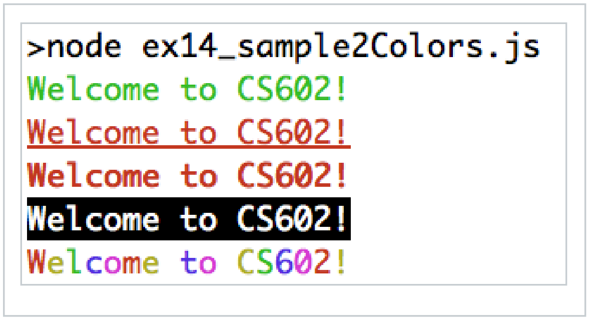
```
>node ex14_sample1Colors.js
Welcome to CS602!
Welcome to CS602!
Welcome to CS602!
Welcome to CS602!
Welcome to CS602!
```

The module also provides the safe mode where the String definition is not modified. The following example shows the same effects applied on the string data.



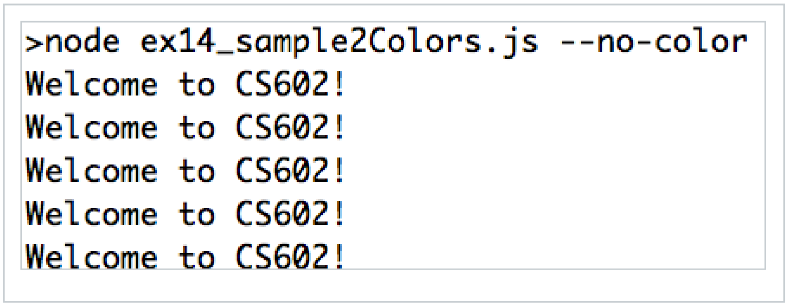
```
var colors = require('colors/safe');

var data = 'Welcome to CS602!';

console.log(colors.green(data));
console.log(colors.red.underline(data));
console.log(colors.red.bold(data));
console.log(colors.inverse(data));
console.log(colors.rainbow(data));
```

The output of the above program is shown below.



```
>node ex14_sample2Colors.js
Welcome to CS602!
Welcome to CS602!
Welcome to CS602!
Welcome to CS602!
Welcome to CS602!
```

The no-color option when running the application suppresses the effects of the colors as shown below.



```
>node ex14_sample2Colors.js --no-color
Welcome to CS602!
Welcome to CS602!
Welcome to CS602!
Welcome to CS602!
Welcome to CS602!
```

Bibliography

Node.js Manual & Documentation, https://nodejs.org/api/all.html

path Node.js Manual & Documentation, https://nodejs.org/api/path.html

fs Node.js Manual & Documentation, https://nodejs.org/api/fs.html

os Node.js Manual & Documentation, https://nodejs.org/api/os.html

Underscore.js, http://underscorejs.org

colors.js, https://github.com/marak/colors.js/

Ethan Brown, Web Development with Node and Express, O'Reilly, 2014.

Events and Streams

Objectives

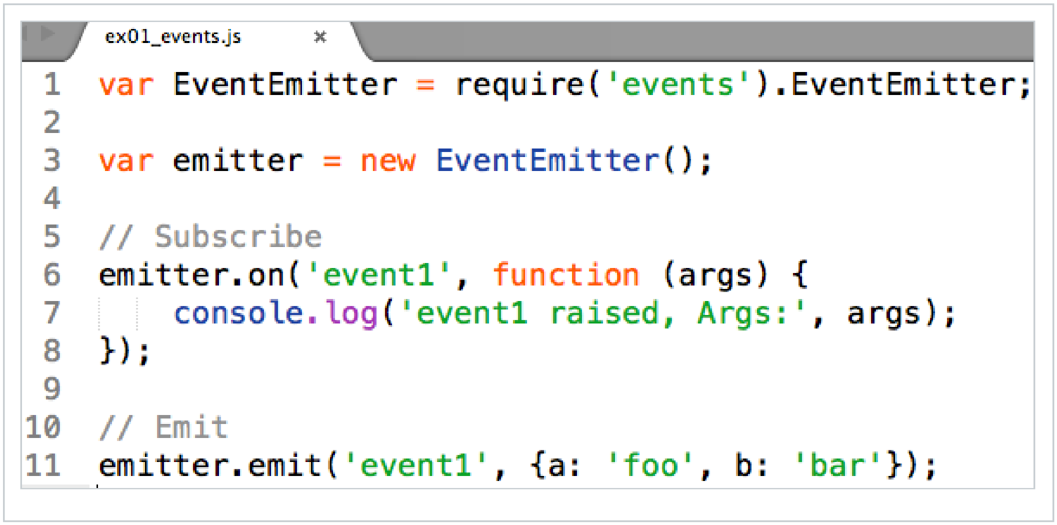
By reading the lectures and textbook, participating in the discussions, and completing the assignments, you will be able to:

- Understand core Node event handling
- Write custom classes that follows Node event pattern
- Understand how streams work
- Pipe data between streams
- Write custom stream classes

Introduction

The core events module in Node.js provides the build-in support for events and event handlers. The EventEmitter class for this module provides the interface for emitting events and binding the callback handlers for those events. Many of the core Node libraries use the events.

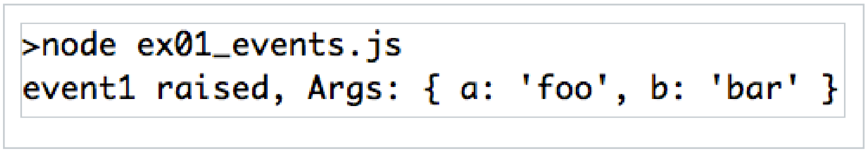
The EventEmitter class can be used for subscribing to events and triggering the events as shown in the example below. The on method is used for subscribing to events. The first argument is the name of the event and the second argument is the listener or the subscriber for this event. The emit method is used for triggering the event. The first argument is the name of the event. The optional second argument provides the arguments that are passed to the subscriber functions.



ex01_events.js

```
var EventEmitter = require('events').EventEmitter;

var emitter = new EventEmitter();

// Subscribe
emitter.on('event1', function (args) {
    console.log('event1 raised, Args:', args);
});

// Emit
emitter.emit('event1', {a: 'foo', b: 'bar'});
```

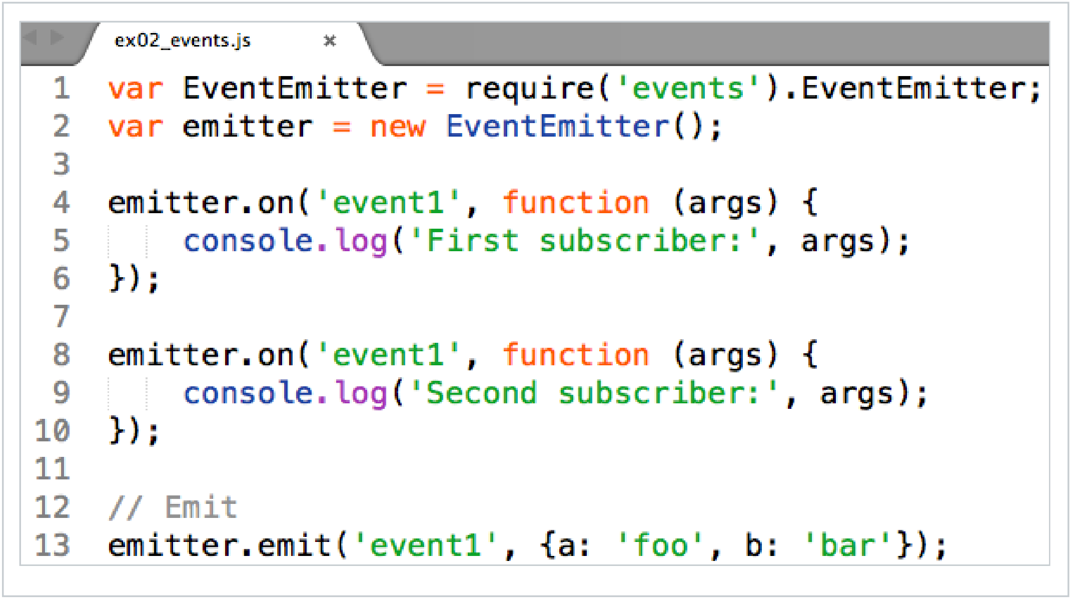
The output of the above program is shown blow. The callback function receives the arguments supplied with the emit call.



```
>node ex01_events.js
event1 raised, Args: { a: 'foo', b: 'bar' }
```

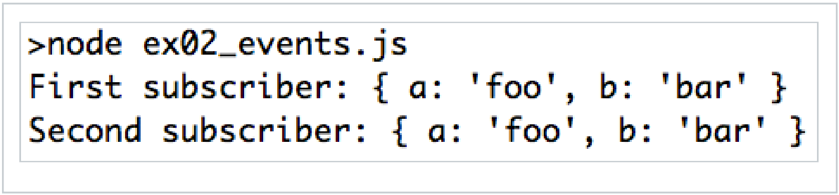
Event Handling – Multiple Subscribers

Multiple subscribers can be registered for any event using the on method. When the event is triggered, these listeners are invoked in the order they were registered. The following example shows two subscribers registered for the same event.



```javascript
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();

emitter.on('event1', function (args) {
    console.log('First subscriber:', args);
});

emitter.on('event1', function (args) {
    console.log('Second subscriber:', args);
});

// Emit
emitter.emit('event1', {a: 'foo', b: 'bar'});
```

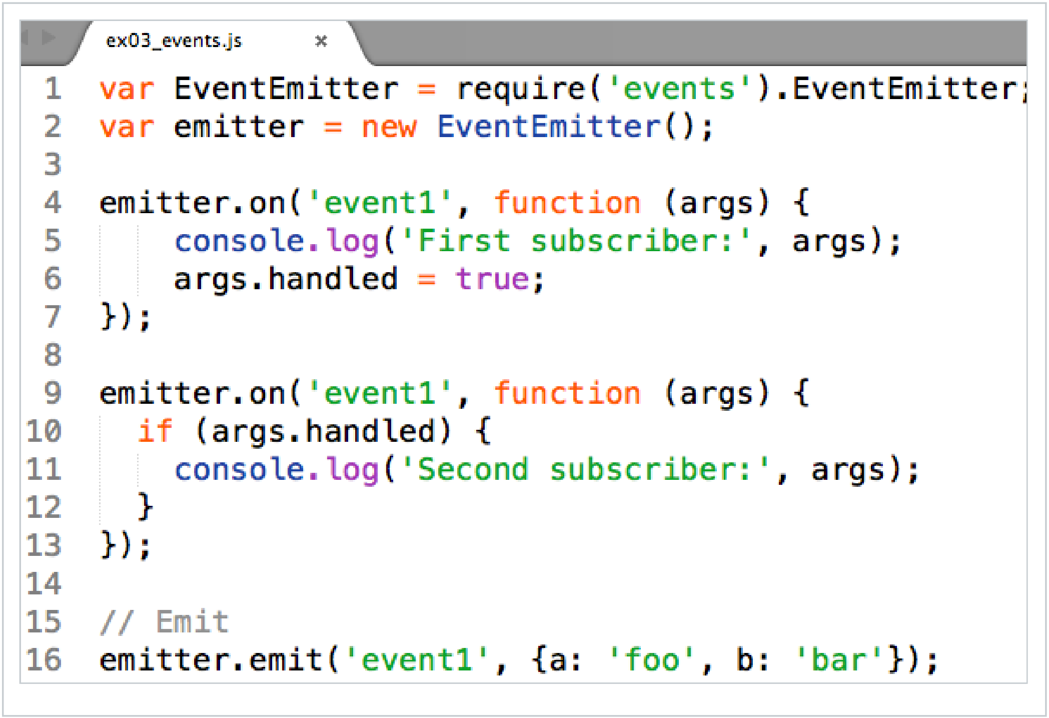
The output of the above program shows the listeners invoked in sequence.



```
>node ex02_events.js
First subscriber: { a: 'foo', b: 'bar' }
Second subscriber: { a: 'foo', b: 'bar' }
```

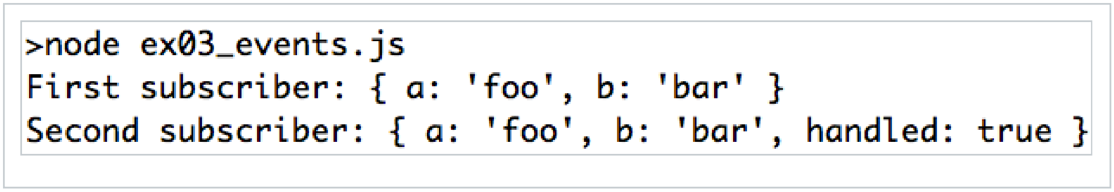
Event Handling – Sharing Data among Subscribers

When multiple subscribers are registered for an event, any arguments passed when the event is triggered are shared among the subscribers. In the following example, the first event handlers modifies the argument object by adding a new property. This property is accessed in the second handler.



```javascript
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();

emitter.on('event1', function (args) {
    console.log('First subscriber:', args);
    args.handled = true;
});

emitter.on('event1', function (args) {
  if (args.handled) {
    console.log('Second subscriber:', args);
  }
});

// Emit
emitter.emit('event1', {a: 'foo', b: 'bar'});
```

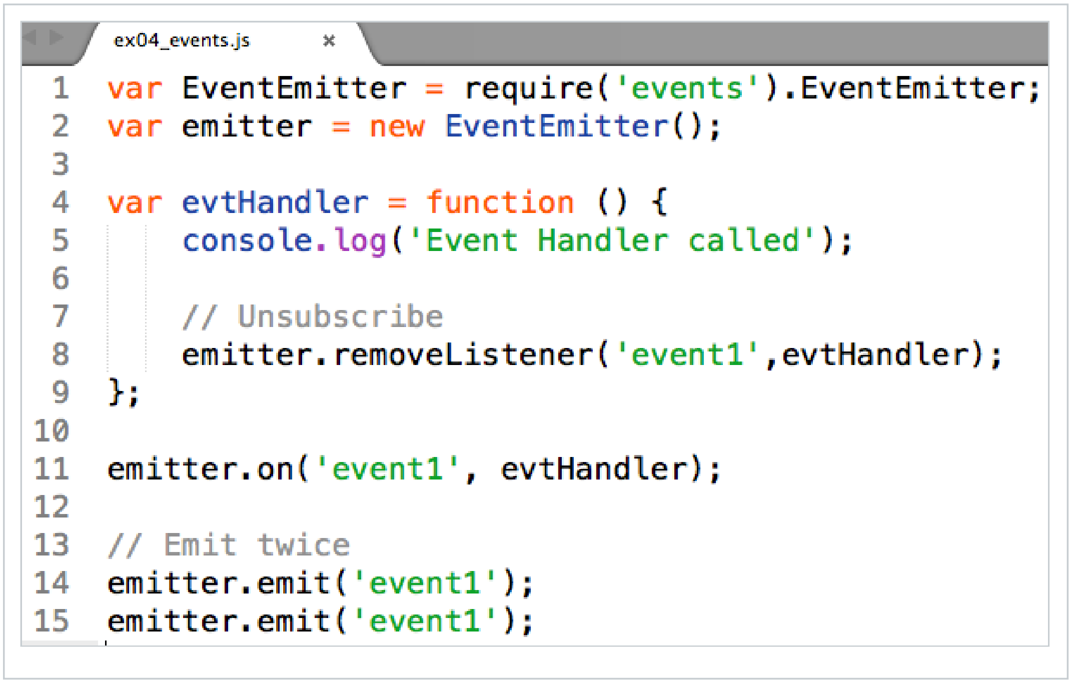
The output of the above program is shown below. The code for the second handler is only invoked if the *handled* property has been set to true by the first handler.



```
>node ex03_events.js
First subscriber: { a: 'foo', b: 'bar' }
Second subscriber: { a: 'foo', b: 'bar', handled: true }
```

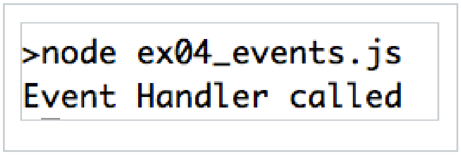
Event Handling – Unsubscribing Listeners

The removeListener method may be used for removing a registered listener. The first argument is the name of the event and the second argument is the reference to the registered function. Anonymous functions that were registered as listeners cannot be removed as a reference to the function is required. The following example removes the listener after the first triggering of the event.



```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();

var evtHandler = function () {
    console.log('Event Handler called');

    // Unsubscribe
    emitter.removeListener('event1',evtHandler);
};

emitter.on('event1', evtHandler);

// Emit twice
emitter.emit('event1');
emitter.emit('event1');
```

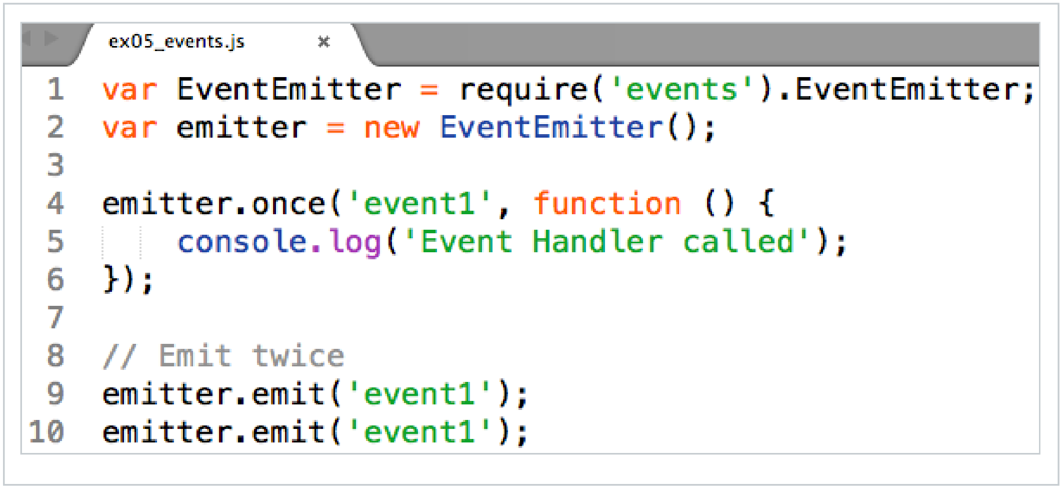
The output of the above program is shown below. The second emit call has no effect as the listener is removed by then.



```
>node ex04_events.js
Event Handler called
```

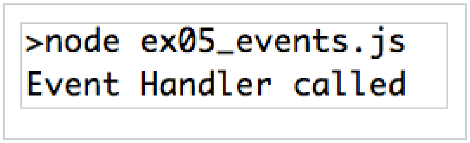
Event Handling – One-time Listeners

The once method is used for subscribing one-time only listeners. When the event is first triggered, the listener is invoked and then automatically unsubscribed for that event. The following example shows the usage for this scenario.



```javascript
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();

emitter.once('event1', function () {
    console.log('Event Handler called');
});

// Emit twice
emitter.emit('event1');
emitter.emit('event1');
```

The output of the above program is shown below. The second emit call has no effect as the listener is removed by then.

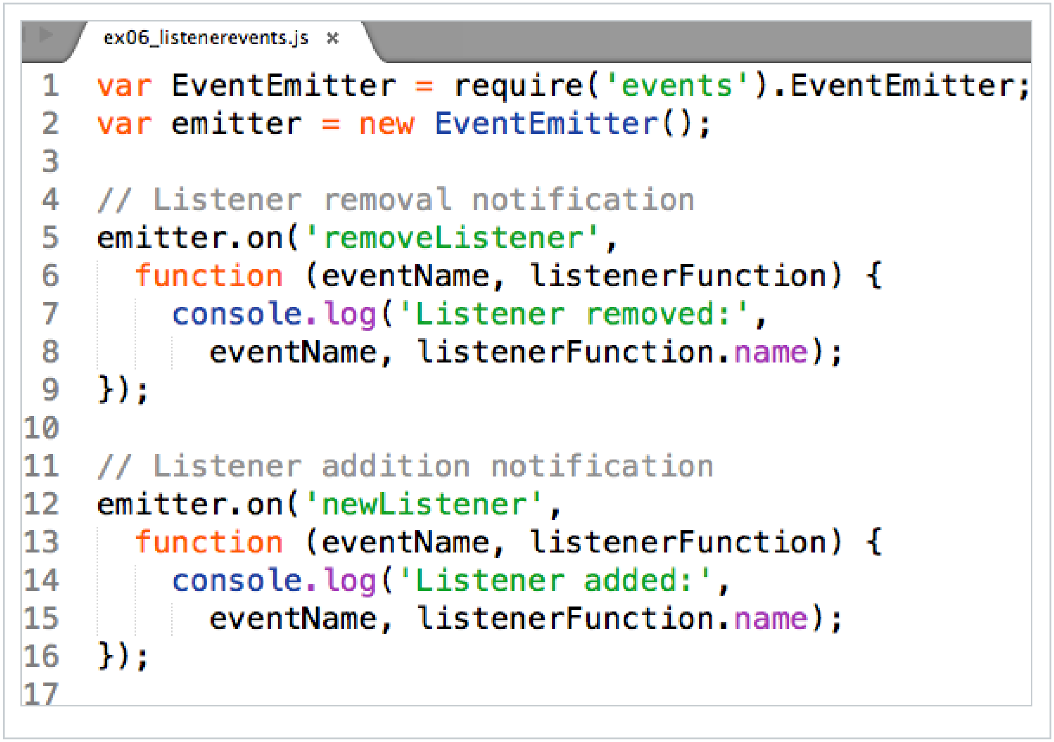


```
>node ex05_events.js
Event Handler called
```

Managing Event Listeners

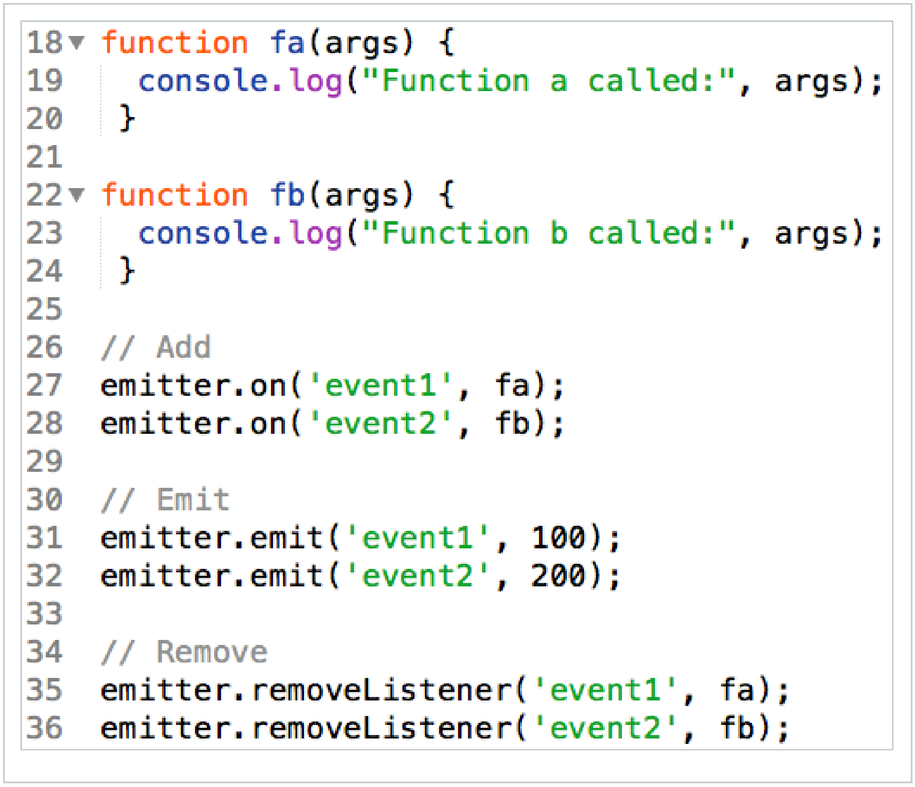
When a new listener is added to an event, the newListener event is automatically raised by the EventEmitter instances. Similarly, the removeListener event is automatically raised by the EventEmitter instances when a listener is removed for the event. The event handlers for these two events receive the event name and a reference to the corresponding function being subscribed/unsubscribed for that event.

The following example shows the callback handlers for the removeListener and the newListener events.



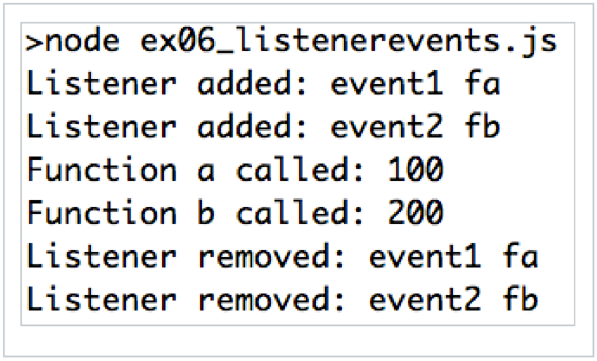
```javascript
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();

// Listener removal notification
emitter.on('removeListener',
  function (eventName, listenerFunction) {
    console.log('Listener removed:',
      eventName, listenerFunction.name);
});

// Listener addition notification
emitter.on('newListener',
  function (eventName, listenerFunction) {
    console.log('Listener added:',
      eventName, listenerFunction.name);
});

```

Two functions and two events are used to demonstrate the subscribing of the functions to the events, emitting the events, and then removing the listeners for those events.



```
function fa(args) {
  console.log("Function a called:", args);
}

function fb(args) {
  console.log("Function b called:", args);
}

// Add
emitter.on('event1', fa);
emitter.on('event2', fb);

// Emit
emitter.emit('event1', 100);
emitter.emit('event2', 200);

// Remove
emitter.removeListener('event1', fa);
emitter.removeListener('event2', fb);
```

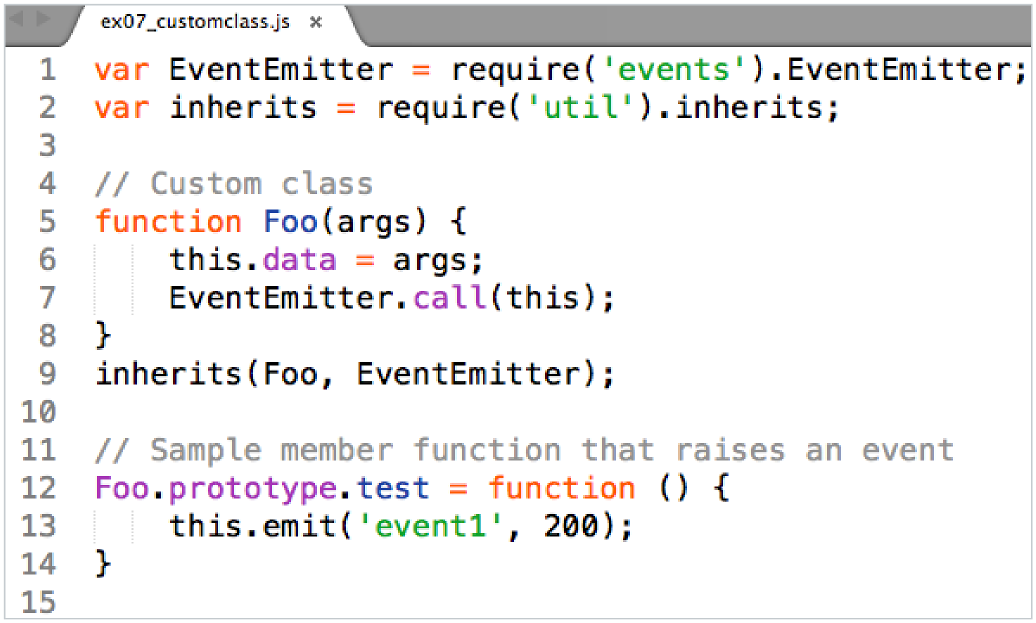
The output of the above program is shown below.



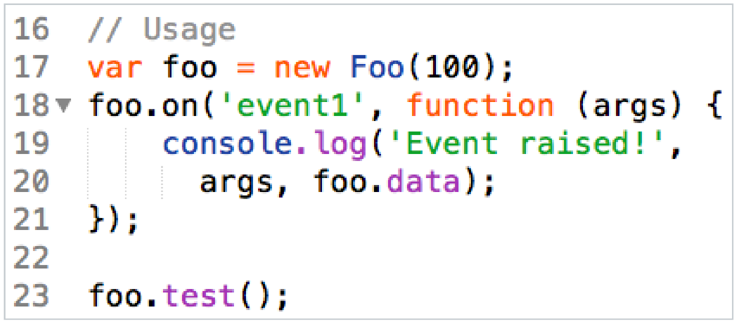
```
>node ex06_listenerevents.js
Listener added: event1 fa
Listener added: event2 fb
Function a called: 100
Function b called: 200
Listener removed: event1 fa
Listener removed: event2 fb
```

Creating Custom Event Emitters

A number of Node.js classes inherit from EventEmitter and follow the same event handling mechanism. For creating customer EventEmitter classes, the constructor for the EventEmitter class is called from the constructor of the custom class. The prototype chain is then setup using the inherits function from the util core module as shown in the following example.

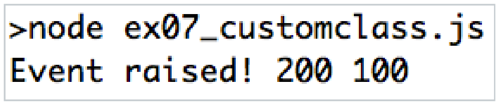


```
var EventEmitter = require('events').EventEmitter;
var inherits = require('util').inherits;

// Custom class
function Foo(args) {
    this.data = args;
    EventEmitter.call(this);
}
inherits(Foo, EventEmitter);

// Sample member function that raises an event
Foo.prototype.test = function () {
    this.emit('event1', 200);
}

```



```
// Usage
var foo = new Foo(100);
foo.on('event1', function (args) {
    console.log('Event raised!',
        args, foo.data);
});

foo.test();
```

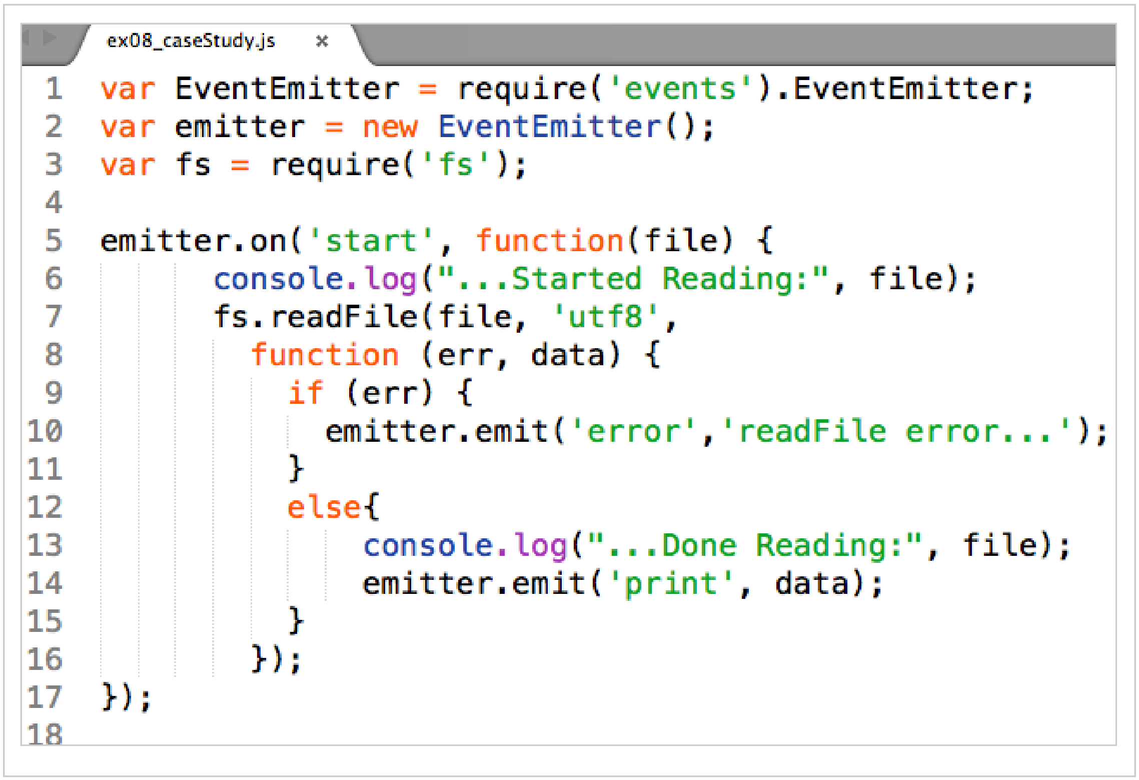
The test method of the custom class is used to trigger the event. The on method inherited from the EventEmitter causes the event listener to be invoked.



```
>node ex07_customclass.js
Event raised! 200 100
```

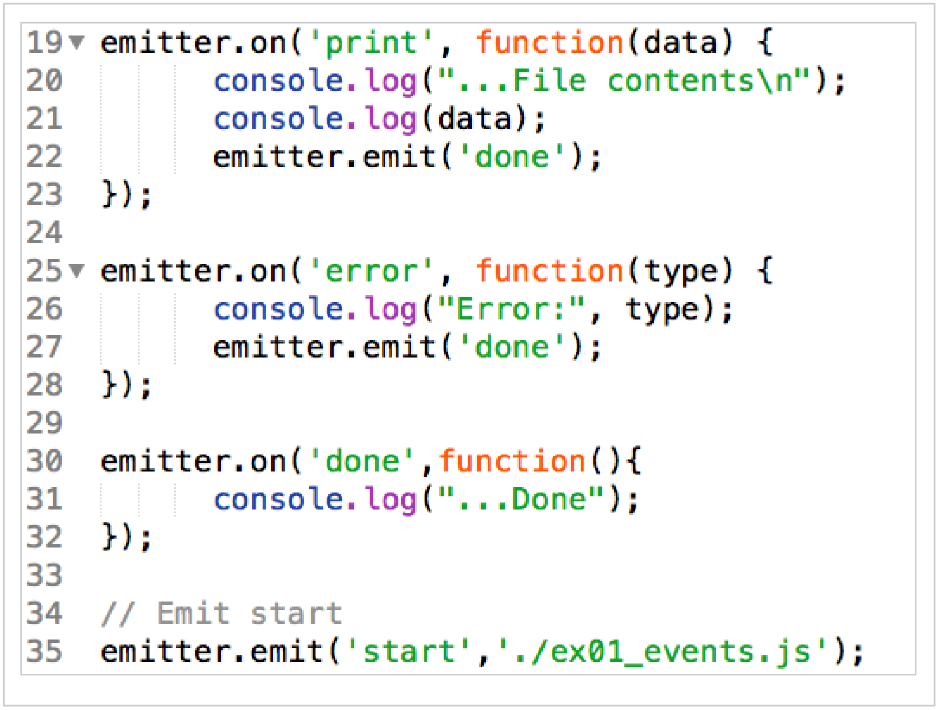
Case Study – Reading Files

The following example defines four events: *start*, *print*, *error*, and *done*. The *start* event handler starts reading the contents of the specified file passed as argument to the handler. The *error* event is triggered if there is an error reading the file. Otherwise, the *print* event is triggered on successful completion of reading the file. The fs module is used for reading the file.



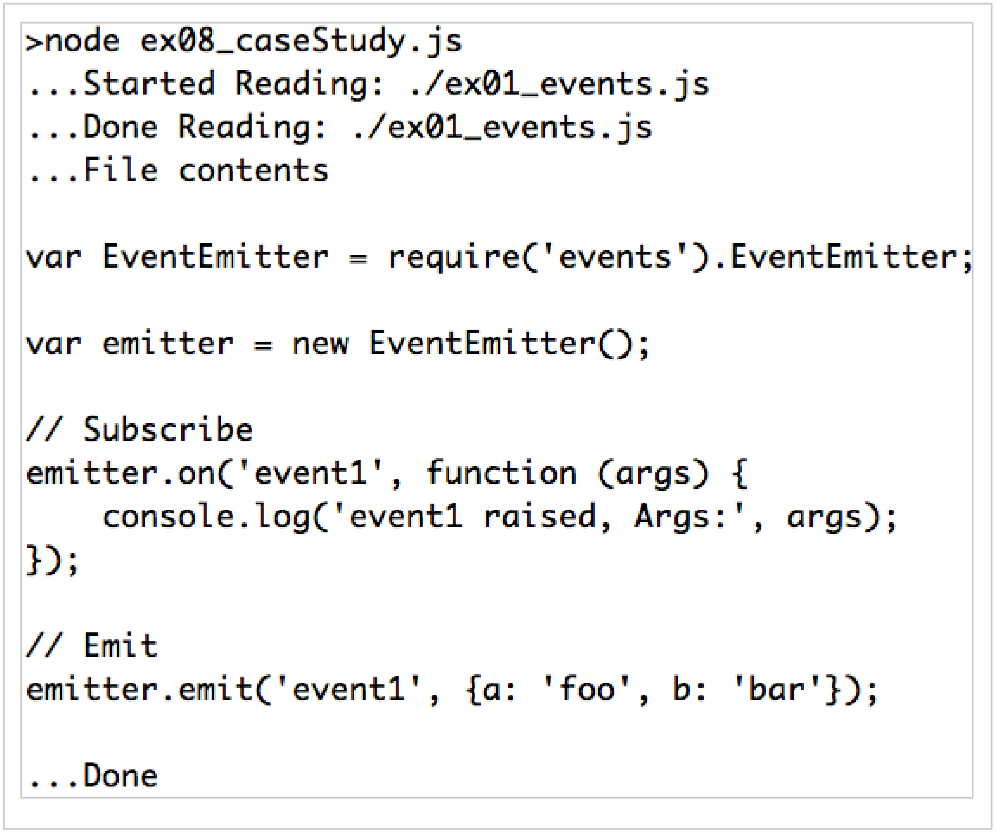
```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter();
var fs = require('fs');

emitter.on('start', function(file) {
   console.log("...Started Reading:", file);
   fs.readFile(file, 'utf8',
    function (err, data) {
     if (err) {
      emitter.emit('error','readFile error...');
     }
     else{
       console.log("...Done Reading:", file);
       emitter.emit('print', data);
     }
    });
});

```

The *print* event handler, in turn, triggers the *done* event after printing the contents to the console.



```
emitter.on('print', function(data) {
   console.log("...File contents\n");
   console.log(data);
   emitter.emit('done');
});

emitter.on('error', function(type) {
   console.log("Error:", type);
   emitter.emit('done');
});

emitter.on('done',function(){
   console.log("...Done");
});

// Emit start
emitter.emit('start','./ex01_events.js');
```

A sample output of the above program is shown below.

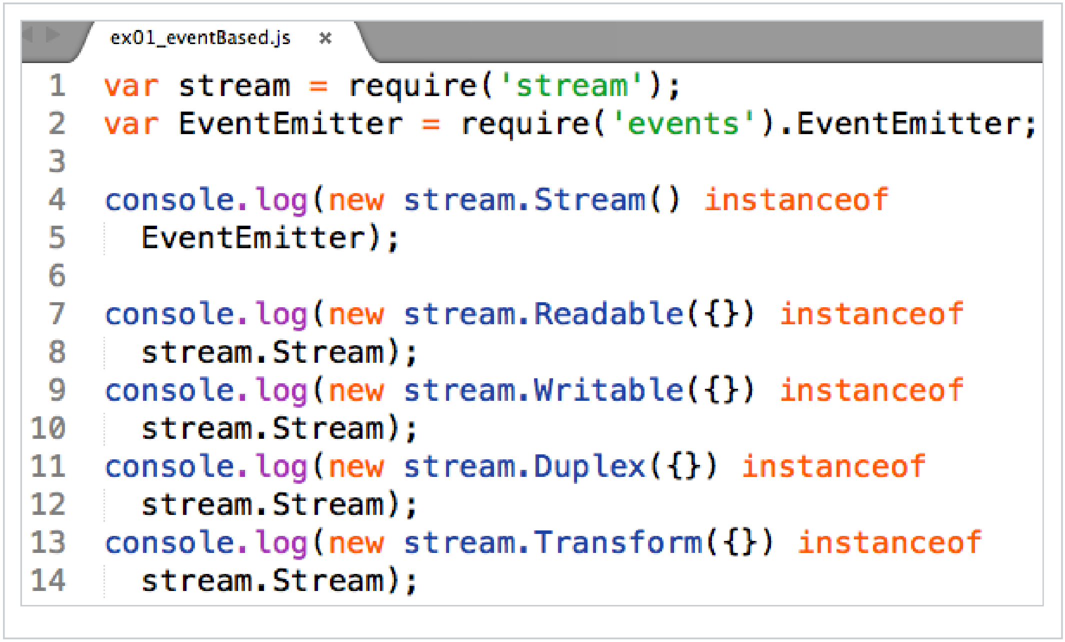


```
>node ex08_caseStudy.js
...Started Reading: ./ex01_events.js
...Done Reading: ./ex01_events.js
...File contents

var EventEmitter = require('events').EventEmitter;

var emitter = new EventEmitter();

// Subscribe
emitter.on('event1', function (args) {
    console.log('event1 raised, Args:', args);
});

// Emit
emitter.emit('event1', {a: 'foo', b: 'bar'});

...Done
```

Streams

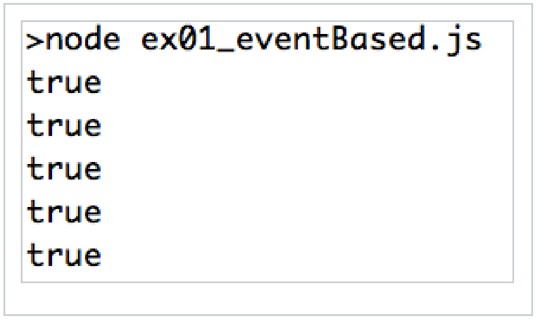
A stream is an abstract interface implements by various Node.js objects. Streams are extensively used in web applications for receiving requests and responding to them. The file system objects are based on the streams for reading and writing data. Streams can be readable, writable, or both (duplex). All streams are instances of EventEmitter and trigger various events. The stream classes from the core stream Node module are the Readable, Writable, Duplex, and Transform classes. All of these stream classes inherit for the base Stream class that in turn inherits from the EventEmitter class.

The following example demonstrates the hierarchy among the classes.



```
var stream = require('stream');
var EventEmitter = require('events').EventEmitter;

console.log(new stream.Stream() instanceof
  EventEmitter);

console.log(new stream.Readable({}) instanceof
  stream.Stream);
console.log(new stream.Writable({}) instanceof
  stream.Stream);
console.log(new stream.Duplex({}) instanceof
  stream.Stream);
console.log(new stream.Transform({}) instanceof
  stream.Stream);
```

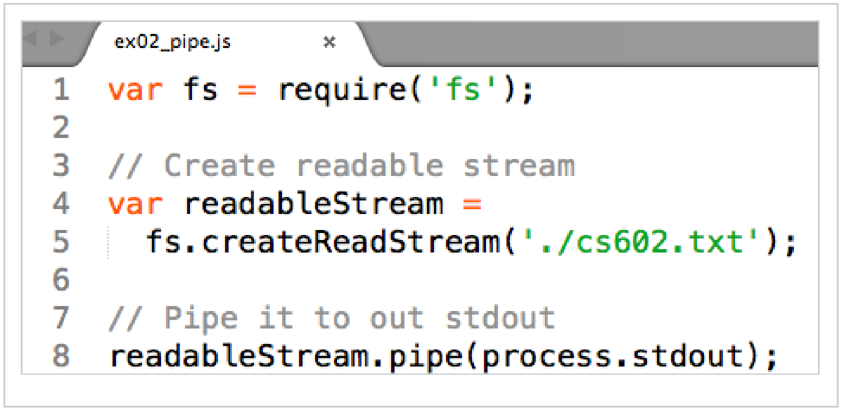
The output of the above program is shown below.



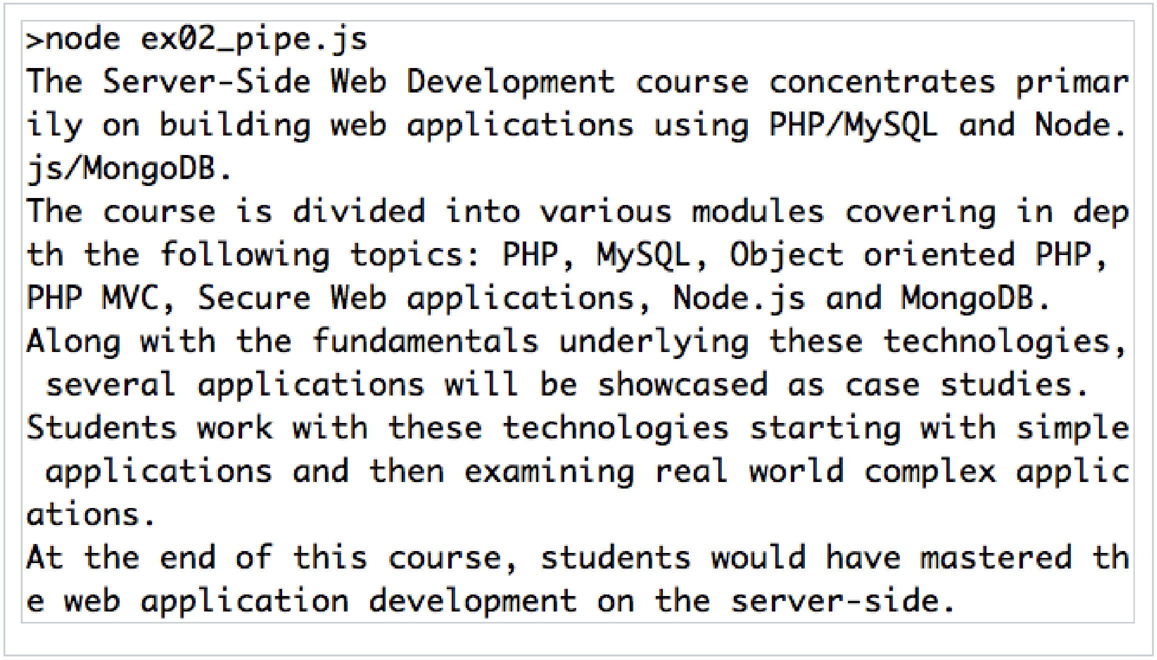
```
>node ex01_eventBased.js
true
true
true
true
true
```

Streams – Pipe Operation

All streams support a pipe operation. In the following example, the content from a Readable stream is piped to the standard output (Writable stream). The createReadStream method from the fs module is used for creating the readable stream of the specified file. The *process* global object's *stdout* member provides the Writable stream to the standard output.

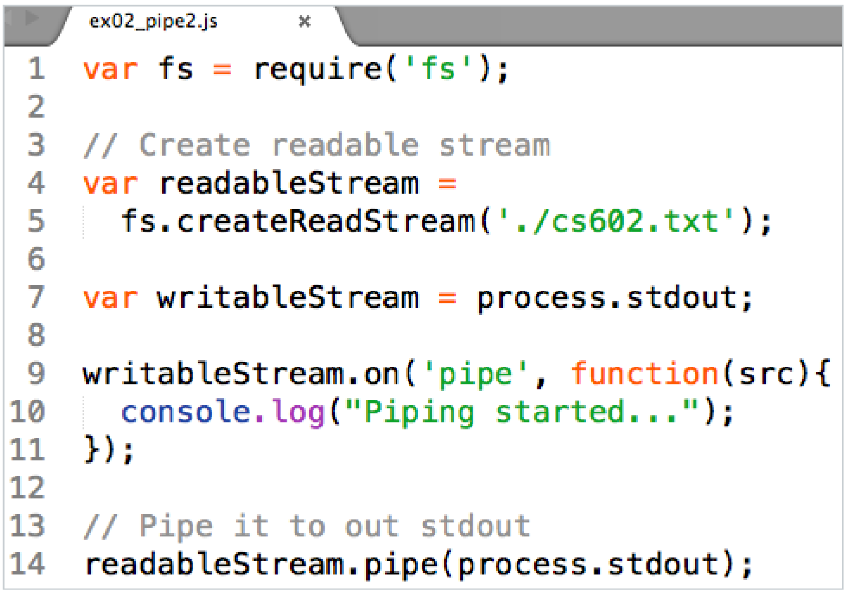


```
var fs = require('fs');

// Create readable stream
var readableStream =
  fs.createReadStream('./cs602.txt');

// Pipe it to out stdout
readableStream.pipe(process.stdout);
```



```
>node ex02_pipe.js
The Server-Side Web Development course concentrates primar
ily on building web applications using PHP/MySQL and Node.
js/MongoDB.
The course is divided into various modules covering in dep
th the following topics: PHP, MySQL, Object oriented PHP,
PHP MVC, Secure Web applications, Node.js and MongoDB.
Along with the fundamentals underlying these technologies,
 several applications will be showcased as case studies.
Students work with these technologies starting with simple
 applications and then examining real world complex applic
ations.
At the end of this course, students would have mastered th
e web application development on the server-side.
```

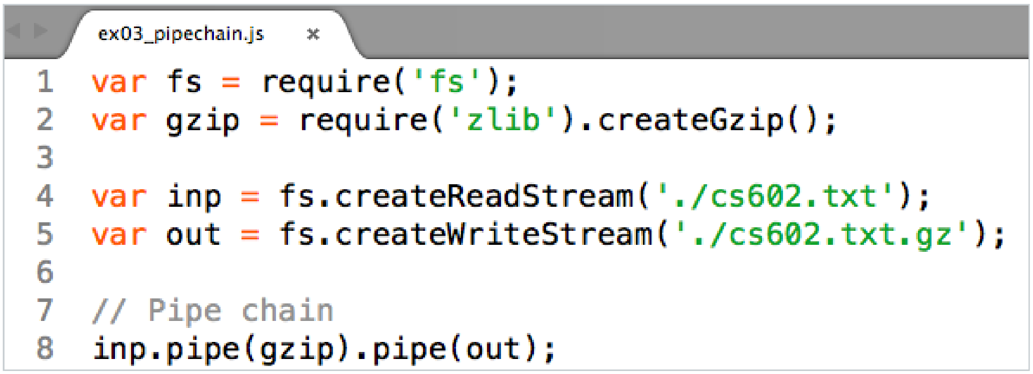
The *pipe* event is emitted whenever the pipe() method is called on the readable stream as shown in the following sample.



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

// Create readable stream
var readableStream =
  fs.createReadStream('./cs602.txt');

var writableStream = process.stdout;

writableStream.on('pipe', function(src){
  console.log("Piping started...");
});

// Pipe it to out stdout
readableStream.pipe(process.stdout);
```

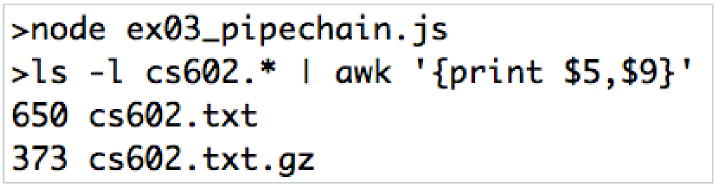
Chaining Multiple Streams

Multiple streams can also be chained using the *pipe* method. In the following example, the contents of a readable stream are piped through a zip transform stream, and then piped to a writable stream. The readable and writable streams are created using the methods from the fs module. The zlib core module provides the method for creating the compressed transform stream using *gzip*.



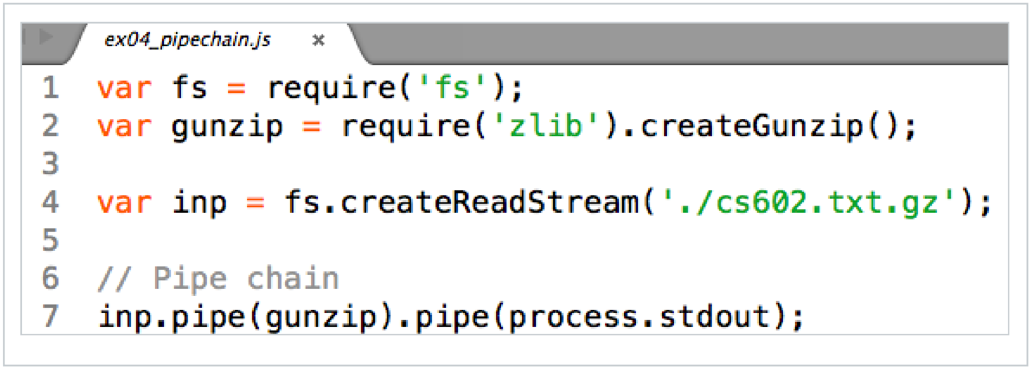
```
var fs = require('fs');
var gzip = require('zlib').createGzip();

var inp = fs.createReadStream('./cs602.txt');
var out = fs.createWriteStream('./cs602.txt.gz');

// Pipe chain
inp.pipe(gzip).pipe(out);
```

The contents of the input file are compresses and written to the specified output file.
The following command shows the sizes of the original file and the compresses file.



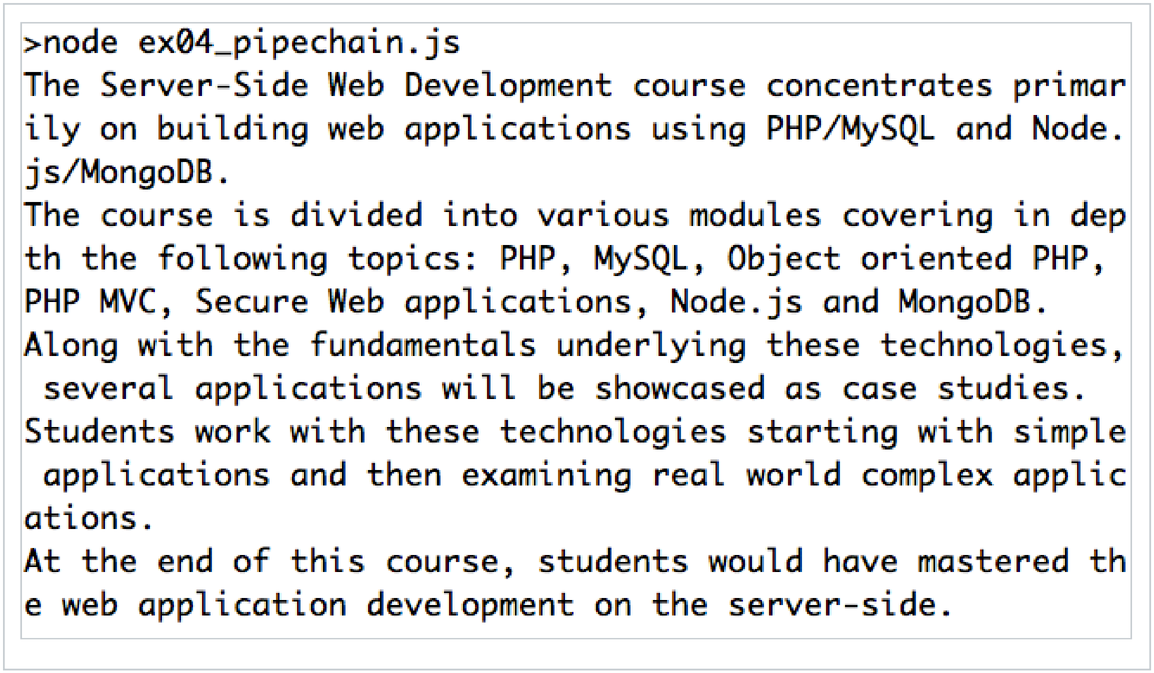
```
>node ex03_pipechain.js
>ls -l cs602.* | awk '{print $5,$9}'
650 cs602.txt
373 cs602.txt.gz
```

The contents of a compressed file can be read and piped to the uncompressed transform stream. The content is then piped to the standard output as shown below.



```javascript
var fs = require('fs');
var gunzip = require('zlib').createGunzip();

var inp = fs.createReadStream('./cs602.txt.gz');

// Pipe chain
inp.pipe(gunzip).pipe(process.stdout);
```

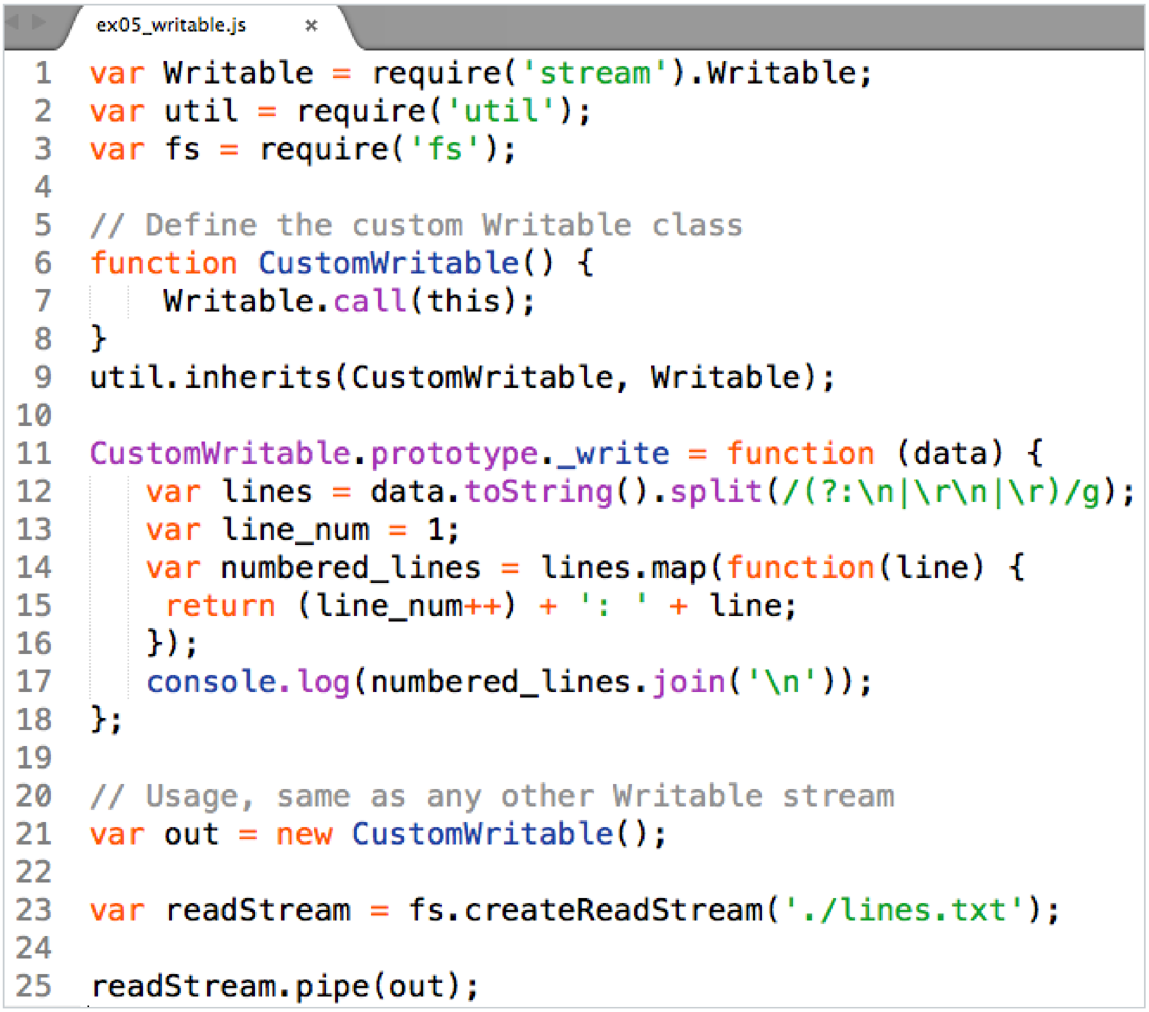
The output of the program is shown below. The uncompressed version of the file is written to the standard output.



```
>node ex04_pipechain.js
The Server-Side Web Development course concentrates primar
ily on building web applications using PHP/MySQL and Node.
js/MongoDB.
The course is divided into various modules covering in dep
th the following topics: PHP, MySQL, Object oriented PHP,
PHP MVC, Secure Web applications, Node.js and MongoDB.
Along with the fundamentals underlying these technologies,
 several applications will be showcased as case studies.
Students work with these technologies starting with simple
 applications and then examining real world complex applic
ations.
At the end of this course, students would have mastered th
e web application development on the server-side.
```

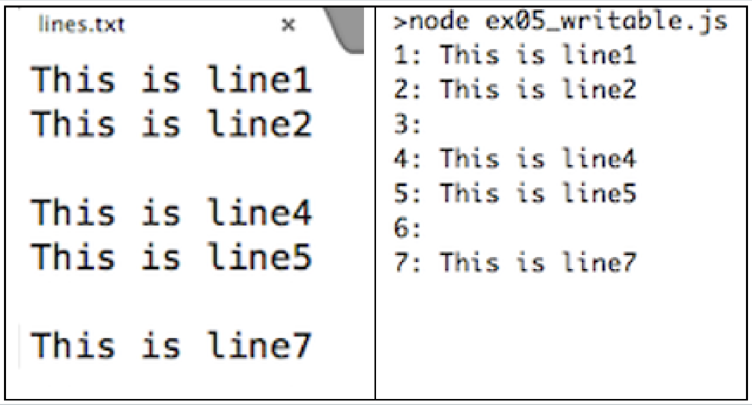
Creating Custom Writable Streams

Custom writable streams can easily be creating by inheriting from the Writable class and overriding the functionality of the _write method. The following example defines the constructor function for the CustomWritable class. The data passed to the _write method is split on the newline (or carriage return) characters and prefixed with the corresponding line number. The readable stream data is piped to this custom writable class.



```javascript
var Writable = require('stream').Writable;
var util = require('util');
var fs = require('fs');

// Define the custom Writable class
function CustomWritable() {
    Writable.call(this);
}
util.inherits(CustomWritable, Writable);

CustomWritable.prototype._write = function (data) {
    var lines = data.toString().split(/(?:\n|\r\n|\r)/g);
    var line_num = 1;
    var numbered_lines = lines.map(function(line) {
     return (line_num++) + ': ' + line;
    });
    console.log(numbered_lines.join('\n'));
};

// Usage, same as any other Writable stream
var out = new CustomWritable();

var readStream = fs.createReadStream('./lines.txt');

readStream.pipe(out);
```

The sample input file and the corresponding output is shown below.



```
This is line1
This is line2

This is line4
This is line5

This is line7
```

```
>node ex05_writable.js
1: This is line1
2: This is line2
3:
4: This is line4
5: This is line5
6:
7: This is line7
```

Bibliography

event Node.js Manual & Documentation, https://nodejs.org/api/events.html

util Node.js Manual & Documentation, https://nodejs.org/api/util.html

stream Node.js Manual & Documentation, https://nodejs.org/api/stream.html zlib Node.js Manual & Documentation, https://nodejs.org/api/zlib.html

fs Node.js Manual & Documentation, https://nodejs.org/api/fs.html

Ethan Brown, Web Development with Node and Express, O'Reilly, 2014.