

Get Programming with JavaScript

John R. Larsen



 **manning**



Get Programming with JavaScript

by John R. Larsen

Chapter 22

Copyright 2016 Manning Publications

brief contents

PART 1 CORE CONCEPTS ON THE CONSOLE1

- 1 ■ Programming, JavaScript, and JS Bin 3
- 2 ■ Variables: storing data in your program 16
- 3 ■ Objects: grouping your data 27
- 4 ■ Functions: code on demand 40
- 5 ■ Arguments: passing data to functions 57
- 6 ■ Return values: getting data from functions 70
- 7 ■ Object arguments: functions working with objects 83
- 8 ■ Arrays: putting data into lists 104
- 9 ■ Constructors: building objects with functions 122
- 10 ■ Bracket notation: flexible property names 147

PART 2 ORGANIZING YOUR PROGRAMS169

- 11 ■ Scope: hiding information 171
- 12 ■ Conditions: choosing code to run 198
- 13 ■ Modules: breaking a program into pieces 221
- 14 ■ Models: working with data 248

- 15 ■ Views: displaying data 264
- 16 ■ Controllers: linking models and views 280

PART 3 JAVASCRIPT IN THE BROWSER.....299

- 17 ■ HTML: building web pages 301
- 18 ■ Controls: getting user input 323
- 19 ■ Templates: filling placeholders with data 343
- 20 ■ XHR: loading data 367
- 21 ■ Conclusion: get programming with JavaScript 387

- 22 ■ Node: running JavaScript outside the browser online
- 23 ■ Express: building an API online
- 24 ■ Polling: repeating requests with XHR online
- 25 ■ Socket.IO: real-time messaging online

Node: running JavaScript outside the browser

This chapter covers

- Running JavaScript with Node
- Exporting and importing code with Node modules
- Experimenting with the read-eval-print loop
- Repeating code with for loops
- Sharing functions with prototypes
- Building game-management components for *The Crypt*

JavaScript doesn't run only in browsers. Among a long list of applications, developers and hobbyists use it to program robots and other devices on the Internet of Things; as a scripting language in other applications like Photoshop and Minecraft; and to create desktop, mobile, and network applications. It's that last example, network applications, that you'll explore in these four online chapters of *Get Programming with JavaScript*. You'll write programs for server and client communication in the following chapters. Here you get started with Node.js: it provides the environment for your JavaScript creativity outside your browser.

22.1 Running JavaScript with Node.js

Node.js lets you run JavaScript programs on your computer, without running them in a browser. It encourages a modular approach to application design, letting you export interfaces from your modules for use in other modules, and it has an enormous number of third-party modules available to install. Almost all of the programs you've written so far will continue to work in Node, and it provides a console-like command-line interface you can use to try out ideas and test your work. But that's enough talk; it's time to walk and then to run.

22.1.1 Downloading Node.js

Download Node.js from <https://nodejs.org>. As shown in figure 22.1, there are two versions: a current version and a Long Term Support (LTS) version. The current version includes the latest features but the LTS version may be supported longer. Follow the LTS schedule link for more information about Node's release schedules. Either version should be fine for the examples in part 4.

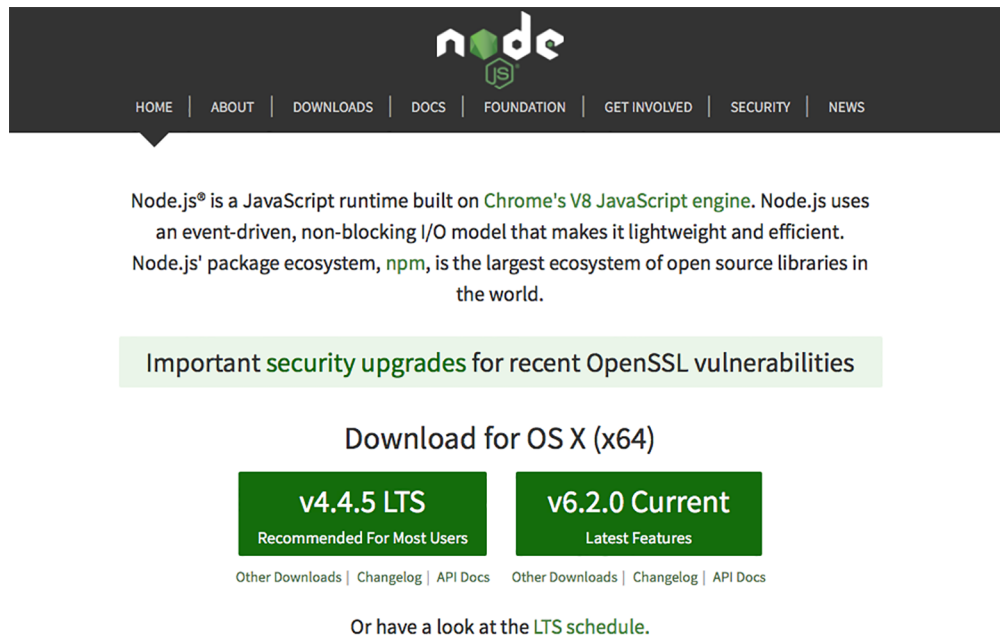


Figure 22.1 Download Node.js from nodejs.org.

22.1.2 Saying hello

Obviously, the first thing you want to accomplish with Node is a program that says hello. Figure 22.2 shows the program in Notepad and the message it produces displayed in a Command Prompt window.

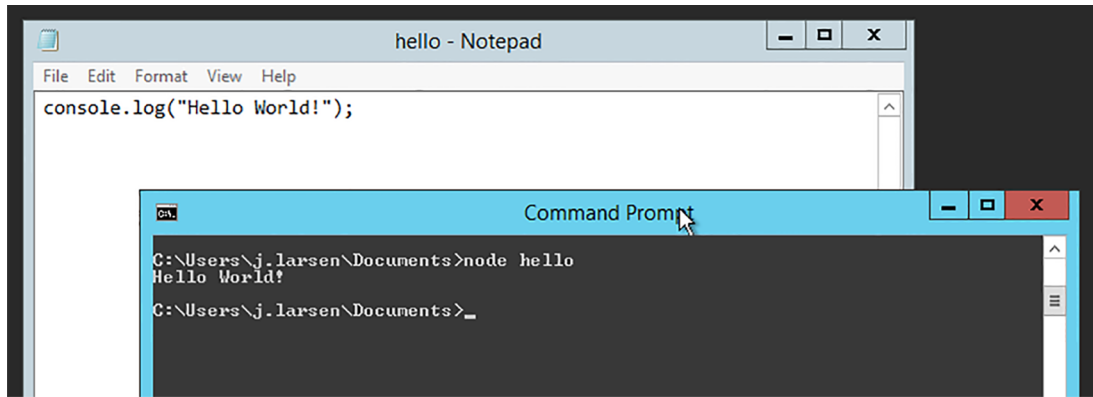


Figure 22.2 hello.js in Notepad and running it with Node at the command prompt

In a text editor, write the program shown in the following listing.

Listing 22.1 Hello World! (hello.js)

```
console.log("Hello World!");
```

Save the program as `hello.js` in a new folder on your computer. In Command Prompt on Windows, Terminal on OS X, or the equivalent on your platform, navigate to the folder. Type the following to run the program:

```
node hello
```

22.1.3 Running existing code

The code from parts 1 and 2 of *Get Programming with JavaScript* will work just fine running in Node, although for user interaction you may want to check out Node's read-eval-print loop (REPL) discussed in section 22.3. Figure 22.3 shows some quiz code from chapter 8, saved in a file called `quiz.js`, in the same folder as `hello.js`, and executed in Node with the command

```
node quiz
```

The code for `quiz.js` follows. It's unchanged from chapter 8 and works in exactly the same way it did when run on JS Bin.

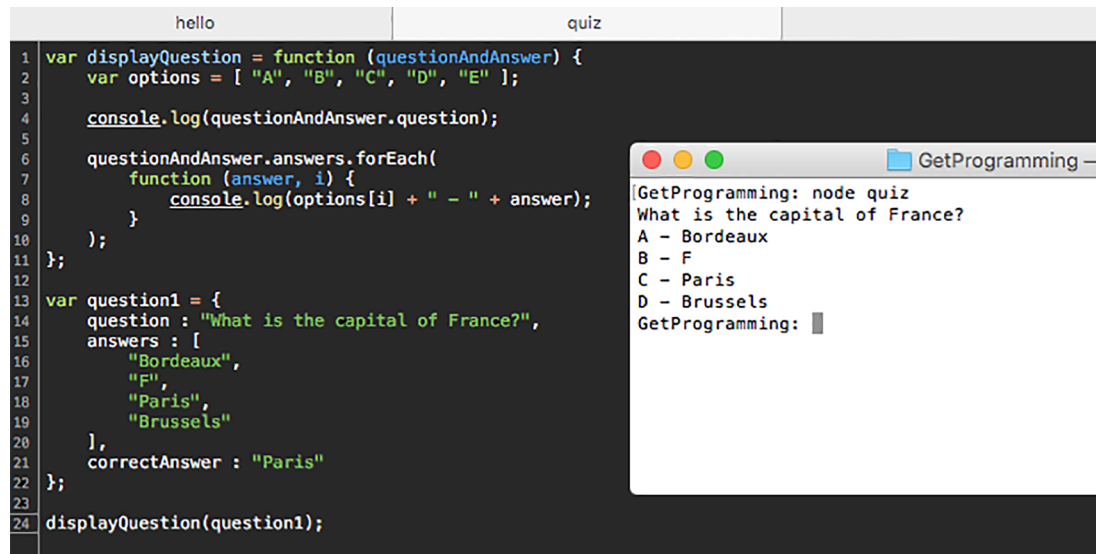


Figure 22.3 quiz.js in the Textastic text editor (left) and running it with Node in a Terminal window (right)

Listing 22.2 Displaying a multiple choice question (quiz.js)

```

var displayQuestion = function (questionAndAnswer) {
  var options = [ "A", "B", "C", "D", "E" ];

  console.log(questionAndAnswer.question);

  questionAndAnswer.answers.forEach(
    function (answer, i) {
      console.log(options[i] + " - " + answer);
    }
  );
};

var question1 = {
  question : "What is the capital of France?",
  answers : [
    "Bordeaux",
    "F",
    "Paris",
    "Brussels"
  ],
  correctAnswer : "Paris"
};

displayQuestion(question1);

```

You spent a large part of part 2 learning about modularization for the organization and reuse of code. You'll be pleased to hear that modules form a core part of Node.

22.2 Turning your code into Node modules

Your smash-hit app sensation The Fruitinator! includes 10 types of fruit to splat. In the final level, players face waves of five pieces of fruit at a time; it gets pretty intense! To choose the fruit at random, you use the `between` function, shown here:

```
var between = function (lowest, highest) {
  var range = highest - lowest + 1;
  return lowest + Math.floor(Math.random() * range);
};
```

You've used the `between` function in a number of places before, and now that you're getting used to coding outside the browser with Node you want to put the function into its own module for reuse. In chapter 13 you saw how to modularize your code without polluting the global namespace. You used immediately invoked function expressions and namespaces to create as few global variables as possible while making an interface available. Here's how you might have shared the `between` function when in a browser environment:

The diagram illustrates the use of an Immediately Invoked Function Expression (IIFE) to create a local scope and ensure a namespace exists. The code is as follows:

```
(function () {
  var between = function (lowest, highest) {
    var range = highest - lowest + 1;
    return lowest + Math.floor(Math.random() * range);
  };

  if (window.gpwj === undefined) {
    window.gpwj = {};
  }

  gpwj.between = between;
})();
```

Annotations in the diagram:

- Use an IIFE to create a local scope:** Points to the opening parenthesis of the IIFE.
- Ensure your namespace exists:** Points to the `if (window.gpwj === undefined) { window.gpwj = {}; }` block.
- Export the function by assigning it as a property of the namespace object:** Points to the `gpwj.between = between;` line.

You would then generate a random number by calling the `between` function as a property of the `gpwj` namespace, like this:

```
var num = gpwj.between(1, 10);
```

With Node, there's a better way of modularizing code.

22.2.1 Exporting your interface from a Node module

Thankfully, Node modules do away with the need for IIFEs and namespaces; the code in a module file is in its own scope and doesn't pollute the global namespace. You make your interface available by assigning it to Node's special `module.exports` property, as shown here.

Listing 22.3 The between module (between.js)

```
var between = function (lowest, highest) {
    var range = highest - lowest + 1;
    return lowest + Math.floor(Math.random() * range);
};

module.exports = between;
```

Define a function and assign it to a variable

Assign the function to the exports property of the module object

Code that imports your module can use whatever you assign to `module.exports`.

22.2.2 Importing a module interface

To use your exported function in another module, use the `require` keyword, as shown in the following listing. The program displays a random fruit.

Listing 22.4 Using the between module (fruit.js)

```
var between = require('./between');

var fruit = ['Apple', 'Orange', 'Lime', 'Lemon', 'Strawberry',
    'Cranberry', 'Banana', 'Pineapple', 'Mango', 'Kumquat'];

var fruitIndex = between(0, 9);

console.log(fruit[fruitIndex]);
```

Import the function exported by your previous module and assign it to the between variable

Use the imported function

You pass `require` the path to the module you're importing. The path in the previous listing is relative to the current directory. Modules in the same directory as the module requiring them need an initial `./`. (The dot slash disambiguates your modules from those you've installed using `npm`, Node's package manager introduced in the next chapter.)

It's great that you can display a single fruit, but you need more fruit. You need five pieces of fruit. You need a fruit loop!

22.2.3 Fruit loops with for loops

You want to display five pieces of fruit. You've managed the display of one; can you repeat that fruity feat five times to produce these five lines?

```
Banana
Kumquat
Apple
Banana
Lime
```

It's time to meet the `for` loop. It's like a slightly more compact `while` loop. The following listing shows code you could use to display the pieces of fruit. To run the program, at the command prompt navigate to the folder containing the file and type

```
node multifruit
```

Listing 22.5 Using a `for` loop to repeat a code block (multifruit.js)

```
var between = require('./between');

var fruit = ['Apple', 'Orange', 'Lime', 'Lemon', 'Strawberry',
  'Cranberry', 'Banana', 'Pineapple', 'Mango', 'Kumquat'];

var i, fruitIndex;

for (i = 1; i < 6; i++) {
  fruitIndex = between(0, 9);
  console.log(fruit[fruitIndex]);
}
```

Use a `for` loop to execute code multiple times

Figure 22.4 shows how the loop specifies an initial expression, a condition, and a final expression, separated by semicolons, between parentheses. Remember, `++` is an increment operator; it adds 1 to the `i` variable.

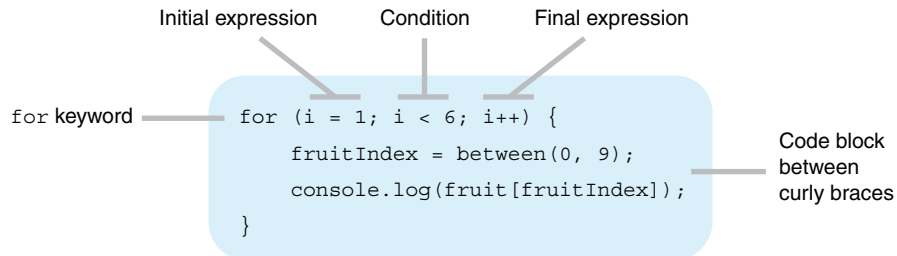
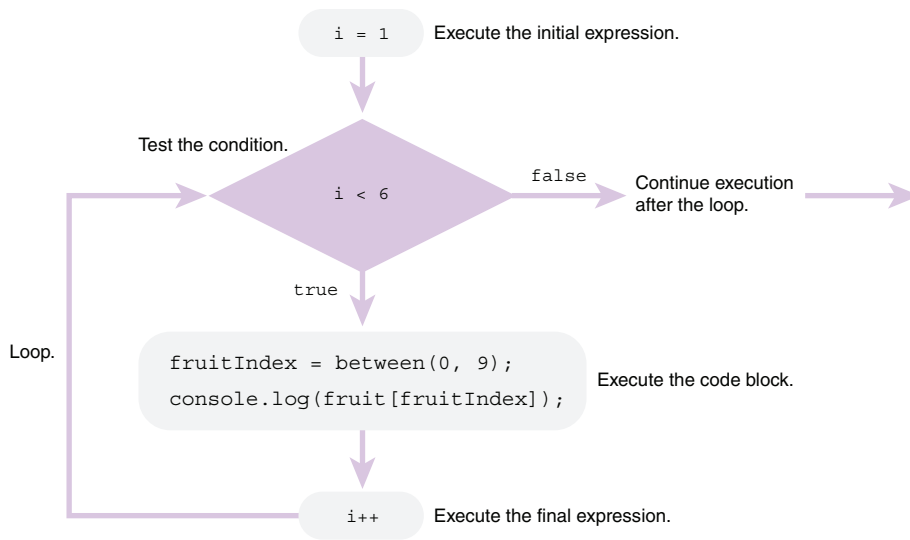


Figure 22.4 The parts of a `for` loop

First, the initial expression is executed. Then the condition is evaluated. If it evaluates to false, the code block is skipped and program execution continues after the `for` loop. If it evaluates to true, the code block is executed, followed by the final expression, and execution loops back to the condition. Figure 22.5 breaks down the process.

You don't have to use a `for` loop. You could stick with a `while` loop, as shown in the next listing. Compare the two approaches. The `for` loop is a more compact version of the `while` loop.

**Figure 22.5** The flow of a for loop**Listing 22.6** Comparing a while loop to a for loop (multifruitWhile.js)

```

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

var fruit = ['Apple', 'Orange', 'Lime', 'Lemon', 'Strawberry',
             'Cranberry', 'Banana', 'Pineapple', 'Mango', 'Kumquat'];

var i, fruitIndex;

i = 1;

while (i < 6) {
    fruitIndex = between(0, 9);
    console.log(fruit[fruitIndex]);
    i++;
}

```

Initial expression

Test the condition

Execute the code block

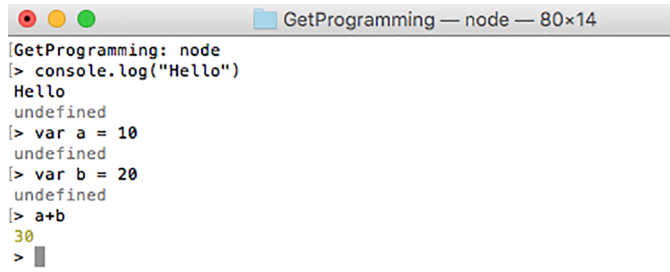
Final expression

You'll see lots of for loops in the wild; `forEach` is a more recent addition to the language, so most code uses for loops when iterating over arrays.

Brilliant! Suitably armed with fruit loops, you can get back to your Smoothie 9mm and code that frenzied fruit fracas The Fruitinator! needs. But with no browser and no browser console, how can players interact with your programs? One possibility is the Node REPL.

22.3 Executing JavaScript with the read-eval-print loop

Logging messages to the console is fun, but you're way past that on your JavaScript adventure. You want interaction. Node provides a *read-eval-print loop* for interacting with JavaScript in a manner similar to the console in a browser. You can use it for testing, debugging, or just trying things out. You love just trying things out! To enter the REPL, type `node` at the command prompt, as shown in figure 22.6.



```
GetProgramming: node
> console.log("Hello")
Hello
undefined
> var a = 10
undefined
> var b = 20
undefined
> a+b
30
> █
```

Figure 22.6 Entering commands in the REPL

The REPL uses its own prompt, `>`. At the prompt, you can type JavaScript statements, inspect the values of variables, define functions, and require modules. When you've finished, type `.exit` or press Ctrl-C.

22.3.1 Running a quiz in the REPL

It's quiz time! In the same folder as the `between` module, create the module shown in listing 22.7 and save it as `quiz2.js`. Enter the REPL and import the quiz code using the `require` statement:

```
node
> var quiz = require('./quiz2')
undefined
```

You assign to the `quiz` variable the interface object that the quiz module exports. Take the quiz by calling the `quizMe` and `submit` methods:

```
> quiz.quizMe()
'5 x 6'
> quiz.submit("30")
'Correct!'
> quiz.quizMe()
'9 x 3'
> quiz.submit("28")
'No, the answer is 27'
```

Listing 22.7 Running a quiz (quiz2.js)

```

var between = require('./between');
var qIndex = 0;

var questions = [
  { question: "7 x 8", answer: "56" },
  { question: "12 x 12", answer: "144" },
  { question: "5 x 6", answer: "30" },
  { question: "9 x 3", answer: "27" }
];

var getQuestion = function () {
  qIndex = between(0, questions.length - 1);
  return questions[qIndex].question;
};

var checkAnswer = function (userAnswer) {
  if (userAnswer === questions[qIndex].answer) {
    return "Correct!";
  } else {
    return "No, the answer is " + questions[qIndex].answer;
  }
};

module.exports = {
  quizMe: getQuestion,
  submit: checkAnswer
};

```

← **Import the between module**

← **Use the between function to pick a random question**

Export your interface object

Only the interface methods are available at the REPL prompt; you won't be able to access the private variables from the module, like the `questions` array.

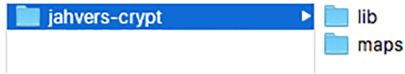
There's a lot of code from earlier in the book for you to play with at the REPL. The rest of this chapter converts code files from *The Crypt* into Node modules and uses the REPL to test them out.

22.4 The Crypt—starting a local project

Players who enter the wonderful world of *The Crypt* can explore, plunder, ponder, and expire. And yet, there's one challenge they can't overcome: loneliness. Hugging zombies and cuddling leopards are exciting at first, but those bites to the players' necks and scratches on their backs aren't true signs of a long-lasting friendship. You can help! In this extra (online only) content of *Get Programming with JavaScript*, you'll create a multiplayer environment; finally, players will be able to meet, greet, and cheat other characters sharing their adventure.

22.4.1 Setting up the project structure

In a new `jahvers-crypt` folder, create a `lib` folder and a `maps` folder, as shown in figure 22.7.



► **Figure 22.7** The initial folder structure for the multiplayer game

The lib folder will contain a library of server-side JavaScript modules used to build the game. The maps folder will contain map data JSON files. As you create modules and add them to your library, you can test out their functionality on the Node REPL.

22.4.2 Moving your Player and Place constructors into Node modules

At its core, the multiplayer version of *The Crypt* is still the same game. Much of the code is untouched and most of the rest requires only small changes. Start by converting your constructor function for player objects to a Node module, as shown here.

Listing 22.8 The Player constructor (player.js)

```
var Player = function (name, health) {
  var items = [];
  var place = null;

  this.addItem = function (item) {
    items.push(item);
  };

  this.hasItem = function (item) {
    return items.indexOf(item) !== -1;
  };

  this.removeItem = function (item) {
    var itemIndex = items.indexOf(item);
    if (itemIndex !== -1) {
      items.splice(itemIndex, 1);
    }
  };

  this.setPlace = function (destination) {
    place = destination;
  };

  this.getPlace = function () {
    return place;
  };


  this.applyDamage = function (damage) {
    health = health - damage;
  };

  this.getData = function () {
    var data = {
      "name" : name,
      "health" : health,
      "items" : items.slice()
    };
  };
};
```

```

        if (place !== null) {
            data.place = place.getData().title;
        }
        return data;
    };
};
module.exports = Player;

```


**Export the
constructor
function**

There's no time like the present, so head straight to the Node REPL to test out your new player module. On the command line, navigate to the jahvers-crypt folder and type the following:

```

node
> var Player = require('./lib/player')
> var dax = new Player('Dax', 50)
> dax.getData()
{ name: 'Dax', health: 50, items: [] }
> dax.addItem('a rusty key')
> dax.getData()
{ name: 'Dax', health: 50, items: [ 'a rusty key' ] }

```

I haven't shown responses of undefined. Try out the other player object methods; if you have one, you can use the up arrow on your keyboard to move through a history of your REPL commands—a handy, time-saving feature. One method you can't use yet is `setPlace`, because you don't have any places! Remedy that by creating the `place.js` file shown in the following listing.

Listing 22.9 The Place constructor (`place.js`)

```

var Place = function (title, description) {
    var exits = {};
    var items = [];
    var challenges = {};

    this.addItem = function (item) {
        items.push(item);
    };

    this.getLastItem = function () {
        return items.pop();
    };

    this.addExit = function (direction, exit) {
        exits[direction] = exit;
    };

    this.getExit = function (direction) {
        return exits[direction];
    };
};

```



```

    this.addChallenge = function (direction, challenge) {
        challenges[direction] = challenge;
    };

    this.getChallenge = function (direction) {
        return challenges[direction];
    };

    this.getData = function () {
        var data = {
            "title" : title,
            "description" : description,
            "items" : items.slice(),
            "exits" : Object.keys(exits)
        };

        return data;
    };
};

module.exports = Place;

```

Export the
constructor
function

Continuing at the REPL:

```

> var Place = require('./lib/place')
> var kitchen = new Place('The Kitchen', 'You are in the kitchen.')
> kitchen.getData()
{ title: 'The Kitchen',
  description: 'You are in a kitchen.',
  items: [],
  exits: [] }
> dax.setPlace(kitchen)
> dax.getData()
{ name: 'Dax',
  health: 50,
  items: [ 'a rusty key' ],
  place: 'The Kitchen' }

```

Create a second place and add items and exits; test out as much as you can.

22.4.3 Working with maps and map data

Node allows you to import JSON files with the `require` statement. It automatically converts the data into a JavaScript object. In the `maps` folder, save `theDarkHouse.json` file, shown here.

Listing 22.10 The Dark House map data (`theDarkHouse.json`)

```

{
  "title" : "The Dark House",
  "firstPlace" : "1",

```

```

"places" : [
  {
    "id" : "1",
    "title" : "The Kitchen",
    "description" : "You are in a kitchen. There is a disturbing smell.",
    "items" : [ "a piece of cheese" ],
    "exits" : [
      { "direction": "south",
        "to": "2",
        "challenge" : {
          "message" : "A zombie sinks its teeth into your neck.",
          "success" : "The zombie disintegrates into a puddle of goo.",
          "failure" : "The zombie is strangely resilient.",
          "requires" : "holy water",
          "itemConsumed" : true,
          "damage" : 20
        }
      },
      { "direction": "west", "to": "3" },
      { "direction": "east", "to": "4" },
      {
        "direction": "north",
        "to": "5",
        "challenge" : {
          "message" : "The way north is blocked by a locked door.",
          "success" : "The key turns smoothly in the lock.",
          "failure" : "That just doesn't work. The door won't open.",
          "requires" : "a rusty key",
          "itemConsumed" : false
        }
      }
    ]
  },
  {
    "id" : "2",
    "title" : "The Old Library",
    "description" : "You are in a library. Dusty books line the walls.",
    "items" : [ "a rusty key" ],
    "exits" : [
      { "direction" : "north", "to" : "1" }
    ]
  },
  {
    "id" : "3",
    "title" : "The Kitchen Garden",
    "description" : "You are in a small, walled garden.",
    "items" : [ "holy water" ],
    "exits" : [
      { "direction" : "east", "to" : "1" }
    ]
  },
  {
    "id" : "4",
    "title" : "The Kitchen Cupboard",

```

```

    "description" : "You are in a cupboard. It's surprisingly roomy.",
    "items" : [ "a tin of spam" ],
    "exits" : [
      { "direction" : "west", "to" : "1" }
    ]
  },
  {
    "id" : "5",
    "title" : "Outside",
    "description" : "You are outside. Congratulations, you have finished
the game!",
    "items" : [ "a sense of pride" ]
  }
]
}

```

You can inspect the properties of the data at the REPL prompt:

```

> var mapData = require('./maps/theDarkHouse')

> mapData.title
'The Dark House'

> mapData.places[1]
{ id: '2',
  title: 'The Old Library',
  description: 'You are in a library. Dusty books line the walls.',
  items: [ 'a rusty key' ],
  exits: [ { direction: 'north', to: '1' } ] }

```

To link all of the places together into a map for the game, you need the `mapBuilder.js` code, shown in the following listing.

Listing 22.11 Map building code (`mapBuilder.js`)

```

var Place = require('./place');
var buildMap = function (mapData) {
  var placesStore = {};

  var buildPlace = function (placeData) {
    var place = new Place(placeData.title, placeData.description);

    if (placeData.items !== undefined) {
      placeData.items.forEach(place.addItem);
    }

    placesStore[placeData.id] = place;
  };

  var buildExits = function (placeData) {
    var here = placesStore[placeData.id];

```

← Import the Place
constructor module

```

    if (placeData.exits !== undefined) {
      placeData.exits.forEach(function (exit) {
        var there = placesStore[exit.to];

        var challenge;

        if (exit.challenge) {
          challenge = Object.assign({}, exit.challenge);
        }

        here.addExit(exit.direction, there);
        here.addChallenge(exit.direction, challenge);
      });
    }
  };

  // create all places
  mapData.places.forEach(buildPlace);

  // link places together
  mapData.places.forEach(buildExits);

  return placesStore[mapData.firstPlace];
};

module.exports = buildMap;

```

Return the first place on the map

Export the buildMap function

You now have all of the pieces to try out a mini-adventure at the REPL prompt:

```

> var buildMap = require('./lib/mapBuilder')

> var first = buildMap(mapData)

> dax.setPlace(first)

> dax.getData()
{ name: 'Dax',
  health: 50,
  items: [ 'a rusty key' ],
  place: 'The Kitchen' }

> dax.getPlace().getData()
{ title: 'The Kitchen',
  description: 'You are in a kitchen. There is a disturbing smell.',
  items: [ 'a piece of cheese' ],
  exits: [ 'south', 'west', 'east', 'north' ] }

> dax.setPlace(dax.getPlace().getExit('south'))

> dax.getData()
{ name: 'Dax',
  health: 50,
  items: [ 'a rusty key' ],
  place: 'The Old Library' }

```

Bravo! The old pieces still function as expected. You should now have a JSON file in the maps folder and three modules in the lib folder, as shown in figure 22.8.

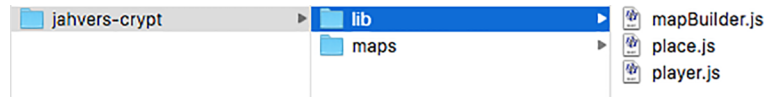


Figure 22.8 Three JavaScript files in the lib folder

It's time to get working on some new code to track multiple players as they move from place to place.

22.5 The Crypt—tracking multiple players

With multiple players wandering the labyrinths, deep space exploration ships, and lost valleys of dinosaurs accessed via *The Crypt*, you need to keep track of who's where and which items the explorers have harvested. Multiple games can be in progress at any one time, each with its own dramatis personae. Figure 22.9 shows one game with Dax and Kandra in The Kitchen Garden and Jahver in The Old Library.

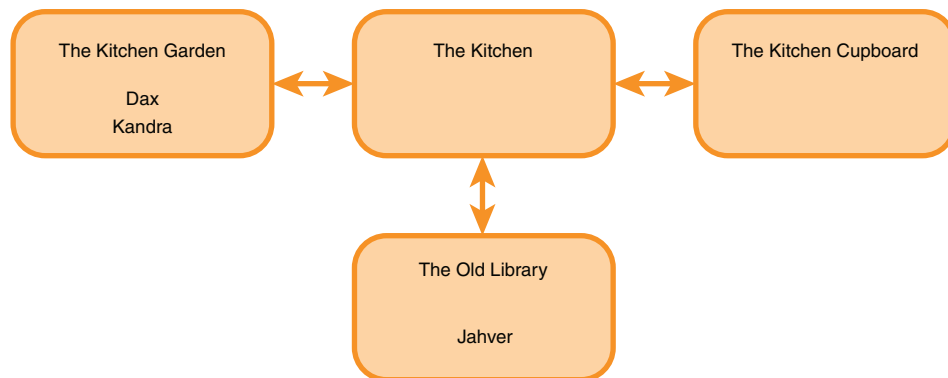


Figure 22.9 A game in progress with three players

Figure 22.10 shows what the browser would show for Dax or Kandra. Both players appear in The Kitchen Garden. It looks like Dax couldn't resist the cheese from The Kitchen, and Kandra has been hugging the zombie again.

Your first new module will be a Game constructor function. You'll be able to use Game to create multiple game objects. You need to be able to pass the constructor an



Figure 22.10 Dax and Kandra are in The Kitchen Garden.

ID for the game and map data imported from a JSON file. Here's a REPL session that shows what you expect, initially, from a game object:

```
node
> var mapData = require('./maps/theDarkHouse')

> var Game = require('./lib/game')

> var game = new Game(1, mapData)

> game.id
1

> game._firstPlace.getData()
{ title: 'The Kitchen',
  description: 'You are in a kitchen. There is a disturbing smell.',
  items: [ 'a piece of cheese' ],
  exits: [ 'south', 'west', 'east', 'north' ] }
```

The following listing shows the constructor function module, before you include methods for adding and listing players.

Listing 22.12 The Game constructor (in game.js)

```
var buildMap = require('./mapBuilder');

function Game (id, mapData) {
  this.id = id;
  this._players = [];
```

```

    this._firstPlace = buildMap(mapData);
}

module.exports = Game;

```

You’ve given two of the properties, `_players` and `_firstPlace`, names with leading underscores. These are properties you’d like others to treat as private, even though you’ve assigned them to `this`, making them public. The underscore means nothing to JavaScript; it’s just a convention used by some programmers. You’ll see why the properties have been made public in the next section, when you’ll meet a new way of specifying methods for objects.

The next listing includes the `addPlayer` method, made public by assigning it to `this`. You won’t use this code because there’s a better place to define the method. (All the other listings relate to files saved by the reader or on Github. This one doesn’t.)

Listing 22.13 `this.addPlayer`

```

var buildMap = require('./mapBuilder');

function Game (id, mapData) {
  this.id = id;
  this._players = [];

  this._firstPlace = buildMap(mapData);

  this.addPlayer = function (player) {
    this._players.push(player);
    player.setPlace(this._firstPlace);
  };
}

module.exports = Game;

```

Assigning methods to `this` has worked for you up until now. So, what’s the problem? It’s just a matter of efficiency. When you create objects with the `Game` constructor in listing 22.13, each object you create has its own copy of the `addPlayer` function. Figure 22.11 shows three game objects, each with its own `addPlayer` function. Although each game’s ID is different, all of the `addPlayer` functions are the same.

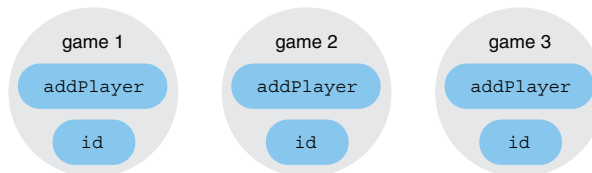


Figure 22.11 Each object created with the `Game` constructor has a copy of the `addPlayer` function.

There’s a more efficient way of assigning methods to objects created with a constructor function; you can add methods to the prototype.

22.5.1 Adding methods to the prototype

Rather than every object you create with a constructor including its own copy of a method, it's more efficient for the objects to share a single copy. JavaScript provides the *prototype* object for that purpose. Figure 22.12 shows three improved game objects sharing a single `addPlayer` method. You assign the method to the prototype object to which all game objects have access.

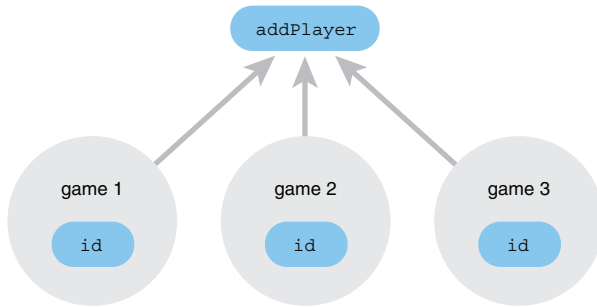


Figure 22.12 Objects created with the same constructor share functions assigned to the prototype.

To share a method, you can assign it to a property of the prototype:

```
Game.prototype.addPlayer = function (player) { ... };
```

Or you can assign an object with your methods to the prototype property:

```
Game.prototype = {
  addPlayer: function (player) { ... }
};
```

The next listing shows the code in listing 22.13 rewritten to use the prototype.

Listing 22.14 Game.prototype.addPlayer (in game.js)

```
var buildMap = require('./mapBuilder');

function Game (id, mapData) {
  this.id = id;
  this._players = [];

  this._firstPlace = buildMap(mapData);
}

Game.prototype = {
  addPlayer: function (player) {
    this._players.push(player);
    player.setPlace(this._firstPlace);
  }
};

module.exports = Game;
```

← Assign the function as a property of the prototype object

The `addPlayer` method needs to access the array of players and the first place in the game. That's why you assign them to this. But they're not part of the interface, so you begin their names with an underscore: `_players` and `_firstPlace`. It's just to give other programmers a better sense of your intentions.

The game objects also need to be able to list players in a specified place. Listing 22.15 includes the `getPlayersInPlace` method, for that purpose. Using it at the REPL looks like this:

```
> var Game = require('./lib/game')
> var mapData = require('./maps/theDarkHouse')
> var Player = require('./lib/player')

> var game = new Game(1, mapData)

> var dax = new Player('Dax', 50)
> game.addPlayer(dax)
> game.addPlayer(new Player('Kandra', 100))
> game.addPlayer(new Player('Jahver', 70))

> game.getPlayersInPlace('The Kitchen')
[ { name: 'Dax', health: 50, items: [], place: 'The Kitchen' },
  { name: 'Kandra', health: 100, items: [], place: 'The Kitchen' },
  { name: 'Jahver', health: 70, items: [], place: 'The Kitchen' } ]

> game.getPlayersInPlace('The Kitchen', dax)
[ { name: 'Kandra', health: 100, items: [], place: 'The Kitchen' },
  { name: 'Jahver', health: 70, items: [], place: 'The Kitchen' } ]
```

The last command shows how to omit a player from the list of players in a place. You'll use that to see who remains when a player exits a place.

Listing 22.15 `Game.prototype.getPlayersInPlace` (game.js)

```
var buildMap = require('./mapBuilder');

function Game (id, mapData) {
  this.id = id;
  this._players = [];

  this._firstPlace = buildMap(mapData);
}

Game.prototype = {
  getPlayersInPlace: function (placeTitle, playerToOmit) {
    var playersInPlace = [];

    this._players.forEach(function (player) {
      if (player !== playerToOmit) {
        var playerPlace = player.getPlayer().getData();

        if (playerPlace.title === placeTitle) {
          playersInPlace.push(player.getData());
        }
      }
    })
  }
}
```

Iterate over all
of the players
in the game

If you don't want to
omit the player,
execute the code block.

If the player is in the
target location, add
them to the array.

```

    });

    return playersInPlace;
  },

  addPlayer: function (player) {
    this._players.push(player);
    player.setPlace(this._firstPlace);
  }
};

module.exports = Game;

```

And with that, you now have a way to track multiple players in multiple games. But the players are individuals and have their own needs; you need a way for players to move around the game independently.

22.6 The Crypt—controlling players

Dax wants that cheese. Kandra wants to travel to the south even though she knows the zombie's a bit bitey. Jahver's trying to open the door to the north with a tin of Spam—good luck with that! However desperate their motivations may seem, each player can choose their destiny; each player can get, go, and use.

Previous versions of *The Crypt* employed a single controller module to manage player actions and game state. You now need similar functionality for each player. You want to be able to issue commands along the lines of the following:

```

daxGame.get();
kandraGame.go('south');
jahverGame.use('a tin of spam', 'north');

```

And you need to be able to retrieve the state of any game:

```

> daxGame.getData()
{ gameId: 1,
  players:
    [ { name: 'Dax', health: 30, items: [Object], place: 'The Kitchen' },
      { name: 'Kandra', health: 100, items: [], place: 'The Kitchen' },
      { name: 'Jahver', health: 70, items: [], place: 'The Kitchen' } ],
  place:
    { title: 'The Kitchen',
      description: 'You are in a kitchen. There is a disturbing smell.',
      items: [],
      exits: [ 'south', 'west', 'east', 'north' ] },
  inPlay: true,
  messages: [] }

```

You've seen most of the methods in the next listing before. But they're now attached to the prototype of a `PlayerGame` constructor. Each `PlayerGame` object will represent a single player in a specified game.

Listing 22.16 The PlayerGame constructor (playerGame.js)

```

var Player = require('./player');

function PlayerGame (playerName, game) {
  this._player = new Player(playerName, 50);
  game.addPlayer(this._player);
  this._game = game;
  this._messages = [];
  this._inPlay = true;
  this._playerName = playerName;
}

PlayerGame.prototype = {
  clearMessages: function () {
    this._messages = [];
  },

  addMessage: function (message) {
    this._messages.push(message);
  },

  getData: function (onlyOtherPlayers) {
    var game = this._game;
    var placeData = this._player.getPlace().getData();

    var playerToOmit = null;
    if (onlyOtherPlayers) {
      playerToOmit = this._player;
    }

    return {
      gameId: game.id,
      players: game.getPlayersInPlace(
        placeData.title, playerToOmit),
      place: placeData,
      inPlay: this._inPlay,
      messages: this._messages
    };
  },

  _checkStatus: function () {
    if (this._player.getData().health <= 0) {
      this._inPlay = false;
      this.addMessage("Overcome by your wounds...");
      this.addMessage("...you fall to the ground.");
      this.addMessage("- Your adventure is over -");
    }
  },

  get: function () {
    var player = this._player;
    var place = player.getPlace();
  }
};

```

Pass a game object when calling the constructor

Create a player

Add the player to the game

Use the game object to retrieve its ID and a list of players

Call addMessage to store a message for a player

```

var item = place.getLastItem();

if (item !== undefined) {
    player.addItem(item);
} else {
    this.addMessage("There is no item to get");
}

return this.getData();
},

```

← **Return the latest game state**

```

use: function (item, direction) {
    var player = this._player;
    var place = player.getPlace();
    var challenge = place.getChallenge(direction);

    if (challenge === undefined || challenge.complete === true) {
        this.addMessage("You don't need to use that there");
    } else if (player.hasItem(item)) {

        if (item === challenge.requires) {
            this.addMessage(challenge.success);
            challenge.complete = true;
            if (challenge.itemConsumed) {
                player.removeItem(item);
            }
        } else {
            this.addMessage(challenge.failure);
        }
    } else {
        this.addMessage("You don't have that item");
    }

    return this.getData();
},

go: function (direction) {
    var player = this._player;
    var place = player.getPlace();
    var destination = place.getExit(direction);
    var challenge = place.getChallenge(direction);

    if (destination === undefined) {
        this.addMessage("There is no exit in that direction");
    } else {
        if ((challenge === undefined) || challenge.complete) {
            player.setPlace(destination);
        } else {
            if (challenge.damage) {
                player.applyDamage(challenge.damage);
            }
            this.addMessage(challenge.message);
            this._checkStatus();
        }
    }
}

```

```

        return this.getData();
    }
};

module.exports = PlayerGame;

```

The methods no longer render messages directly to a display; they now add messages to an array. The `getData` method returns the messages along with the rest of the game's state. The `get`, `go`, and `use` methods return the latest game state after they've executed their action's code.

22.6.1 Playing a multiplayer game at the REPL prompt

With the files you have (figure 22.13), you can now embark on a multiplayer adventure at the REPL prompt. Okay, the display won't be the prettiest, but the mechanics are sound.

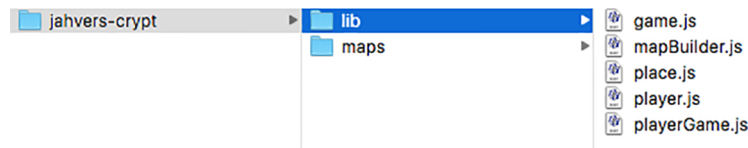


Figure 22.13 Five JavaScript files in the lib folder

Fortunately, you don't have to manually import all of the files or call the constructors; the REPL has a `.load` command. If you save the code in the following listing into a `replTest` folder in the root of the project, you can run the game like this:

```

node
> .load ./replTest/playerGameTest.js

```

Listing 22.17 Testing PlayerGame (playerGameTest.js)

```

var Game = require('../lib/game');
var PlayerGame = require('../lib/playerGame');
var mapData = require('../maps/theDarkHouse');

var game = new Game(1, mapData);

var dax = new PlayerGame('Dax', game);
var kandra = new PlayerGame('Kandra', game);
var jahver = new PlayerGame('Jahver', game);

```

You'll see the REPL execute the commands in the file and then you'll be good to go:

```

> dax.go('west')
{ gameID: 1,
  players:

```


```


    [ { name: 'Dax',
        health: 50,
        items: [],
        place: 'The Kitchen Garden' } ],
    place:
      { title: 'The Kitchen Garden',
        description: 'You are in a small, walled garden.',
        items: [ 'holy water' ],
        exits: [ 'east' ] },
    inPlay: true,
    messages: [] }
> dax.get()
{ gameId: 1,
  players:
    [ { name: 'Dax',
        health: 50,
        items: [Object],
        place: 'The Kitchen Garden' } ],
  place:
    { title: 'The Kitchen Garden',
      description: 'You are in a small, walled garden.',
      items: [],
      exits: [ 'east' ] },
  inPlay: true,
  messages: [] }
> dax.go('east')
{ gameId: 1,
  players:
    [ { name: 'Dax', health: 50, items: [Object], place: 'The Kitchen' },
      { name: 'Kandra', health: 50, items: [], place: 'The Kitchen' },
      { name: 'Jahver', health: 50, items: [], place: 'The Kitchen' } ],
  place:
    { title: 'The Kitchen',
      description: 'You are in a kitchen. There is a disturbing smell.',
      items: [ 'a piece of cheese' ],
      exits: [ 'south', 'west', 'east', 'north' ] },
  inPlay: true,
  messages: [] }
> dax.use('holy water', 'south')
{ gameId: 1,
  players:
    [ { name: 'Dax', health: 50, items: [], place: 'The Kitchen' },
      { name: 'Kandra', health: 50, items: [], place: 'The Kitchen' },
      { name: 'Jahver', health: 50, items: [], place: 'The Kitchen' } ],
  place:
    { title: 'The Kitchen',
      description: 'You are in a kitchen. There is a disturbing smell.',
      items: [ 'a piece of cheese' ],
      exits: [ 'south', 'west', 'east', 'north' ] },
  inPlay: true,
  messages: [
    'The zombie disintegrates into a puddle of goo.'
  ]
}

```


You pick up an item


The place no longer has the item


There are 3 players here


A message has been added

You can run multiple games and each game can include multiple players. The players can move from place to place, pick up items, and use those items to overcome challenges. But you don't want players shackled to the REPL; you want a beautiful browser interface.

In chapter 23, you'll use a package called Express that makes communication between a browser and a server easy to implement. You'll put the modules you created in this chapter to work as key components in a new game server for *The Crypt*.

22.7 Summary

- Run JavaScript programs on your computer, outside a browser, with Node.js.
- Use Node modules to hide implementations and export interfaces.
- Export an interface from a Node module by assigning it to the `module.exports` property.
- Export a function:

```
function sayHello () {  
    console.log('Hello World!');  
}  
  
module.exports = sayHello;
```

- Export an interface object:

```
module.exports = {  
    go: myGoFunction,  
    get: myGetFunction  
};
```

- Use the `require` function to import modules into other modules:

```
var sayHi = require('./hello');  
  
sayHi();
```

- Use Node's read-eval-print loop for testing, debugging, or just trying things out. To start the REPL, type `node`.
- Load files into the REPL with the `.load` command:

```
.load ./lib/myGroovyModule.js
```

- Assign properties to the prototype object to share them among objects created with a constructor function:

```
function Planet (name) {  
    this.name = name;  
}
```

```
Planet.prototype = {  
  show: function () {  
    console.log('Planet: ' + this.name);  
  }  
};  
  
var planet1 = new Planet('Mercury');  
var planet2 = new Planet('Venus');  
  
planet1.show();    // Mercury  
planet2.show();    // Venus
```


Get Programming with JavaScript

John R. Larsen Foreword by Remy Sharp

Are you ready to start writing your own web apps, games, and programs? You're in the right place! **Get Programming with JavaScript** is a hands-on introduction to programming for readers who have never written a line of code.

Since you're just getting started, this friendly book offers you lots of examples backed by careful explanations. As you go along, you'll find exercises to check your understanding and plenty of opportunities to practice your new skills. You don't need anything special to follow the examples—just the text editor and web browser already installed on your computer. We even give you links to working online code so you can see how everything should look live on your screen.

WHAT'S INSIDE

- All the basics—objects, functions, responding to users, and more
- Think like a coder and design your own programs
- Create a text-based adventure game
- Enhance web pages with JavaScript
- Run your programs in a web browser

No experience required! All you need is a web browser and an internet connection.

John Larsen is a web developer and professional teacher in the UK who has many years of experience working with students of all levels, helping them to successfully write their first lines of code.



"Provides the guidance you need to get started ..., the support to keep practicing, and the encouragement to enjoy the adventure."

—From the Foreword by Remy Sharp
Founder of JS Bin

"A great book for the new programmer who wants to learn JavaScript."

—Alvin Raj, Oracle

"An approachable and interactive way of learning JavaScript."

—Giselle Stidston, Breville Pty Ltd

"Great interactive code examples! Building a computer game was my favorite part of the book."

—Ivan Rubelj, Vipnet

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/get-programming-with-javascript

