

Javascript

JavaScript is the most widely used programming language on the web. We'll start with an introduction to the language though many of these concepts apply to many programming languages.

Before we start, how do we add javascript to a project?

You link to a .js file from your index.html file similarly to how you would a css file. This goes in the head:

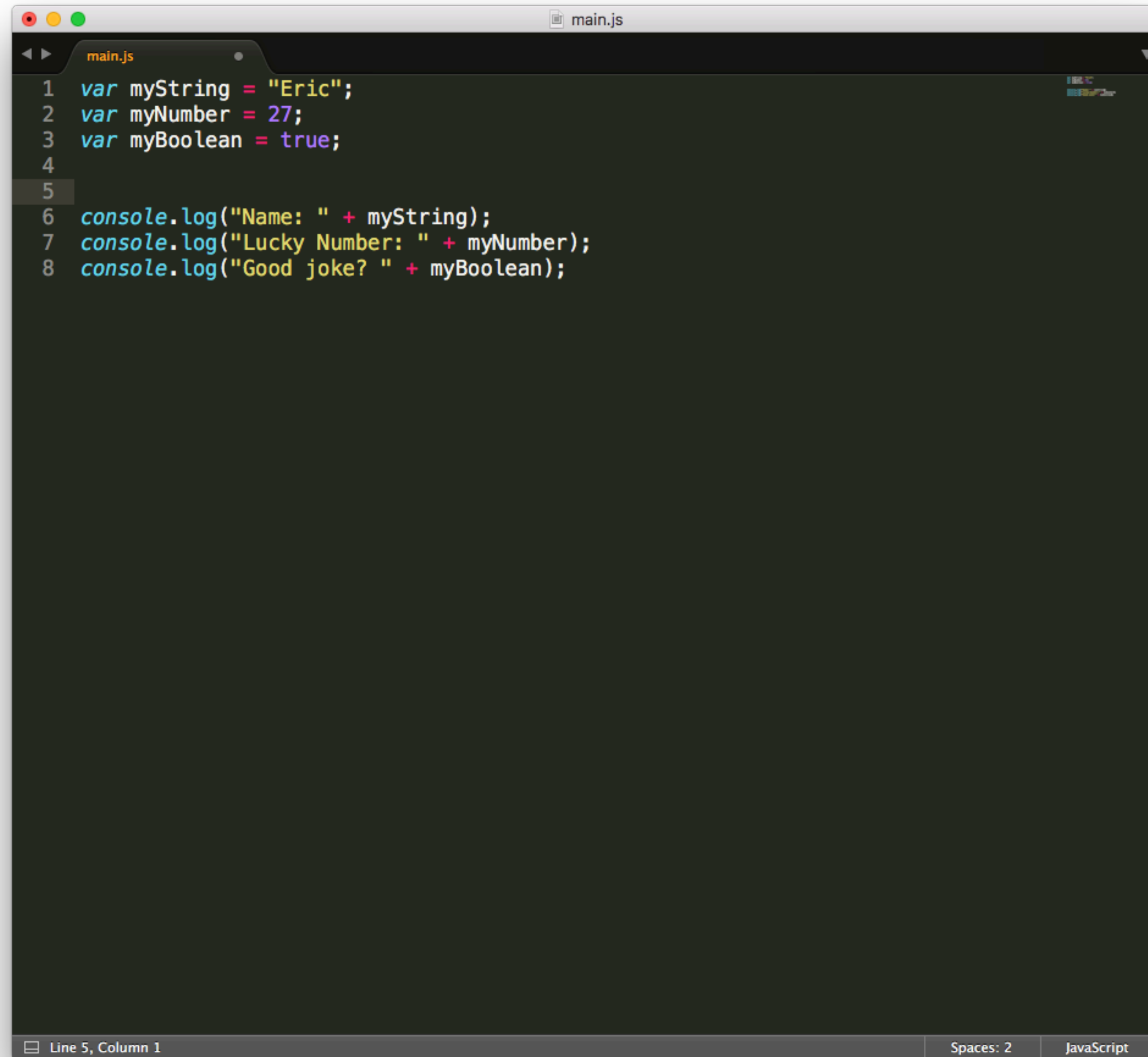
```
<script src="myscripts.js"></script>
```

There are three essential data types to know for now:

String: Any grouping of words or numbers surrounded by single quotes: '...' or double quotes "...".

Number: Any number, including numbers with decimals, without quotes: 4, 8, 1516, 3.42.

Boolean: This is always one of two words. Either true or false, with no quotations.



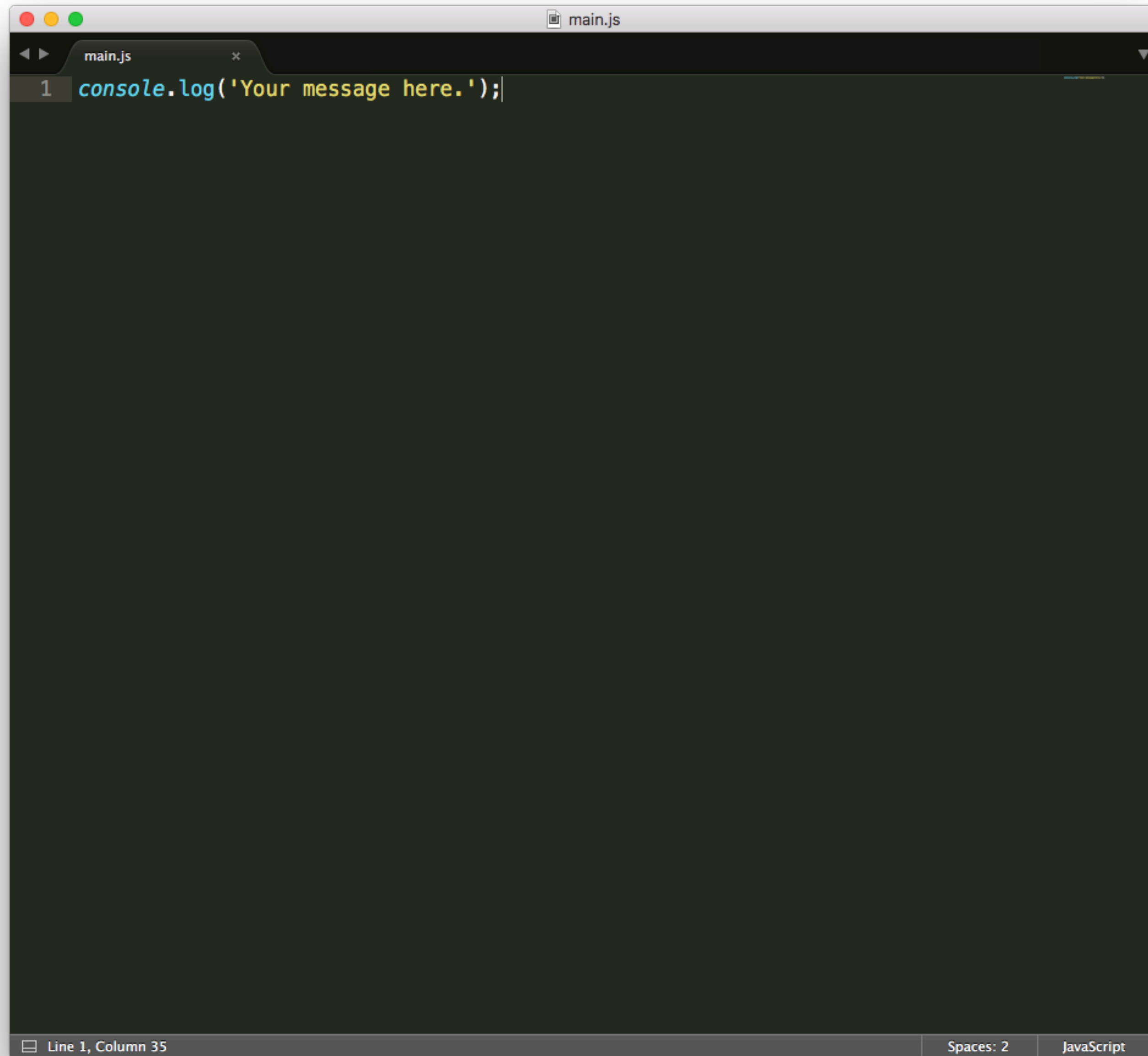
```
1 var myString = "Eric";
2 var myNumber = 27;
3 var myBoolean = true;
4
5
6 console.log("Name: " + myString);
7 console.log("Lucky Number: " + myNumber);
8 console.log("Good joke? " + myBoolean);
```

Line 5, Column 1 Spaces: 2 JavaScript

We can't do much programming with our knowledge of types right now, so let's build something simple. Let's learn how to ask JavaScript to talk to us.

To do this, we need two things:

1. A way to ask JavaScript to talk.
2. Something for JavaScript to say.



A screenshot of a code editor window. The window has a title bar with three colored buttons (red, yellow, green) and a tab labeled 'main.js'. The editor area is dark gray and contains a single line of code: `console.log('Your message here.');`. The line number '1' is visible on the left. The status bar at the bottom shows 'Line 1, Column 35', 'Spaces: 2', and 'JavaScript'.

```
1 console.log('Your message here.');
```

**In human-speak, this is saying:
"Hey console, please print/log
this thing inside the
parentheses."**

By writing this line, we've also solved the second thing we need: Something for JavaScript to say. We can put a String, Number, or Boolean (or any data type) inside the parentheses of a `console.log` statement.

Don't worry, math does not need to be your strong-suit to learn JavaScript but there are just a few operations we'll need to know to make some simple programs.

JavaScript includes the general math *operators* that you can find on a calculator:

1. Add: +
2. Subtract: -
3. Multiply: *
4. Divide: /

These all work how you might guess: 3 + 4 will equal 7, 50/10 will equal 5.

Try some math inside a `console.log`.

There is another operator. It's called the modulus operator.

So, if you divide $13 / 5$, 5 goes into 13 two times, and there will be 3 remaining. A modulus, denoted by a %, would take $13 \% 5$ and return the remainder 3.

JavaScript has built in functions, which help us do everyday things. We'll learn more about functions later, so don't worry about understanding what they are right now... but let's look at an example of a built in function.

Sometimes it's necessary to generate a random number within a program. We can do that with this code:

```
Math.random();
```


To generate a random number between 0 and 50,
we could multiply this result by 50, like so:

```
Math.random() * 50;
```

The problem with this is that the answer will most likely be a decimal. Luckily, JavaScript has our back with another built in function called **Math.floor**. **Math.floor** will take a decimal number, and round down to the nearest whole number. It is used like this:

```
Math.floor(Math.random() * 50);
```

```
Math.floor(Math.random() * 50);
```

In this case:

Math.random will generate a random number between 0 and 1.

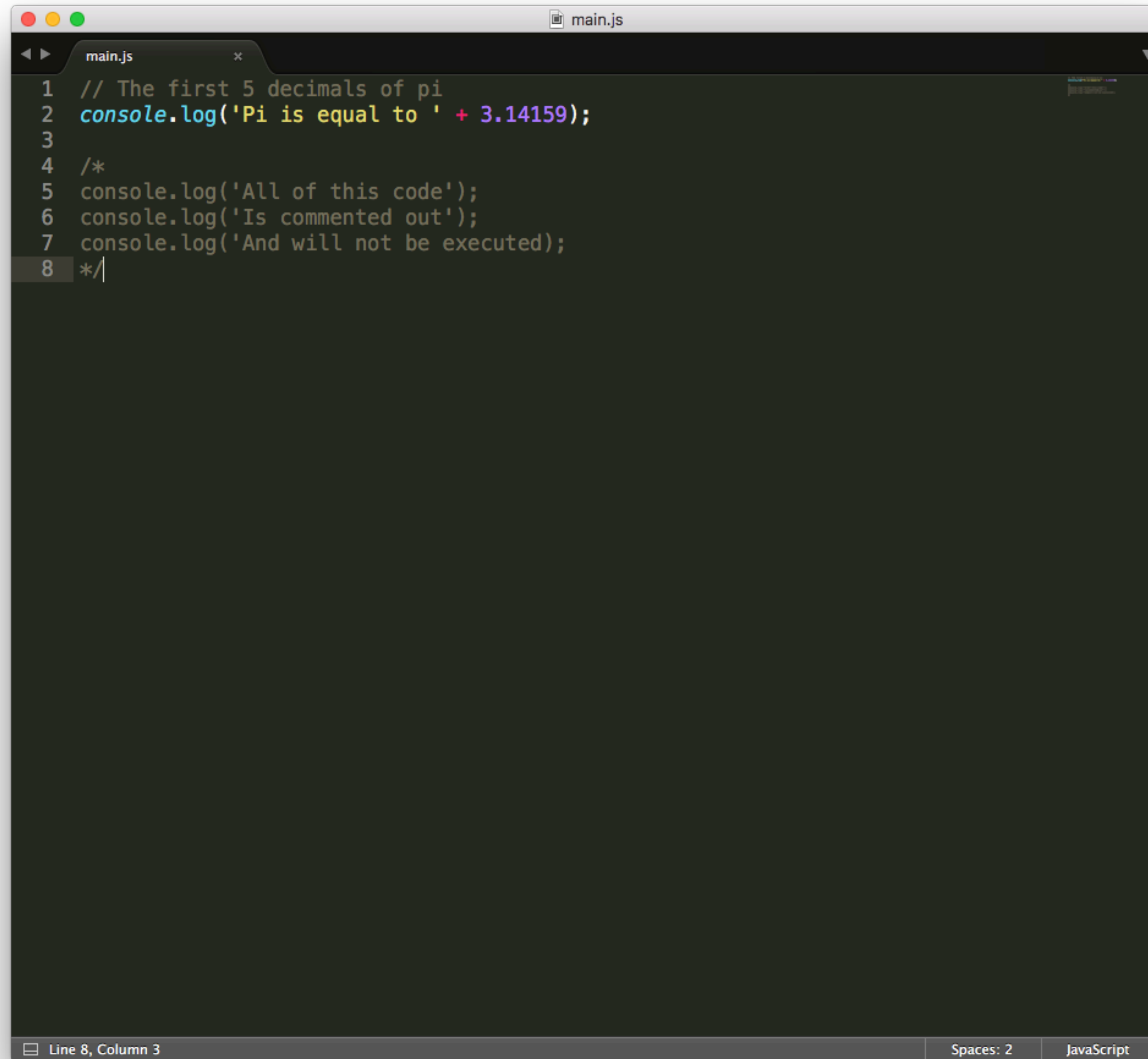
1. We then multiplied that number by **50**, so now we have a number between 0 and 50.

2. Then, **Math.floor** will round the number down to the nearest whole number.

Generating random numbers happens more often than you might think. It's a useful thing to know.

As we write JavaScript, we can create comments in our code.

Comments are lines that are not evaluated when the code runs. They exist just for human readers, in other words.



```
1 // The first 5 decimals of pi
2 console.log('Pi is equal to ' + 3.14159);
3
4 /*
5 console.log('All of this code');
6 console.log('Is commented out');
7 console.log('And will not be executed');
8 */
```

The screenshot shows a code editor window titled 'main.js'. The code contains a single-line comment on line 1, a console log statement on line 2, and a multi-line comment block from line 4 to line 8. The multi-line comment is currently open, showing the code inside it. The status bar at the bottom indicates 'Line 8, Column 3', 'Spaces: 2', and 'JavaScript'.

A single line comment will comment out a single line, and is denoted with two forward slashes **//** preceding a line of JavaScript code.

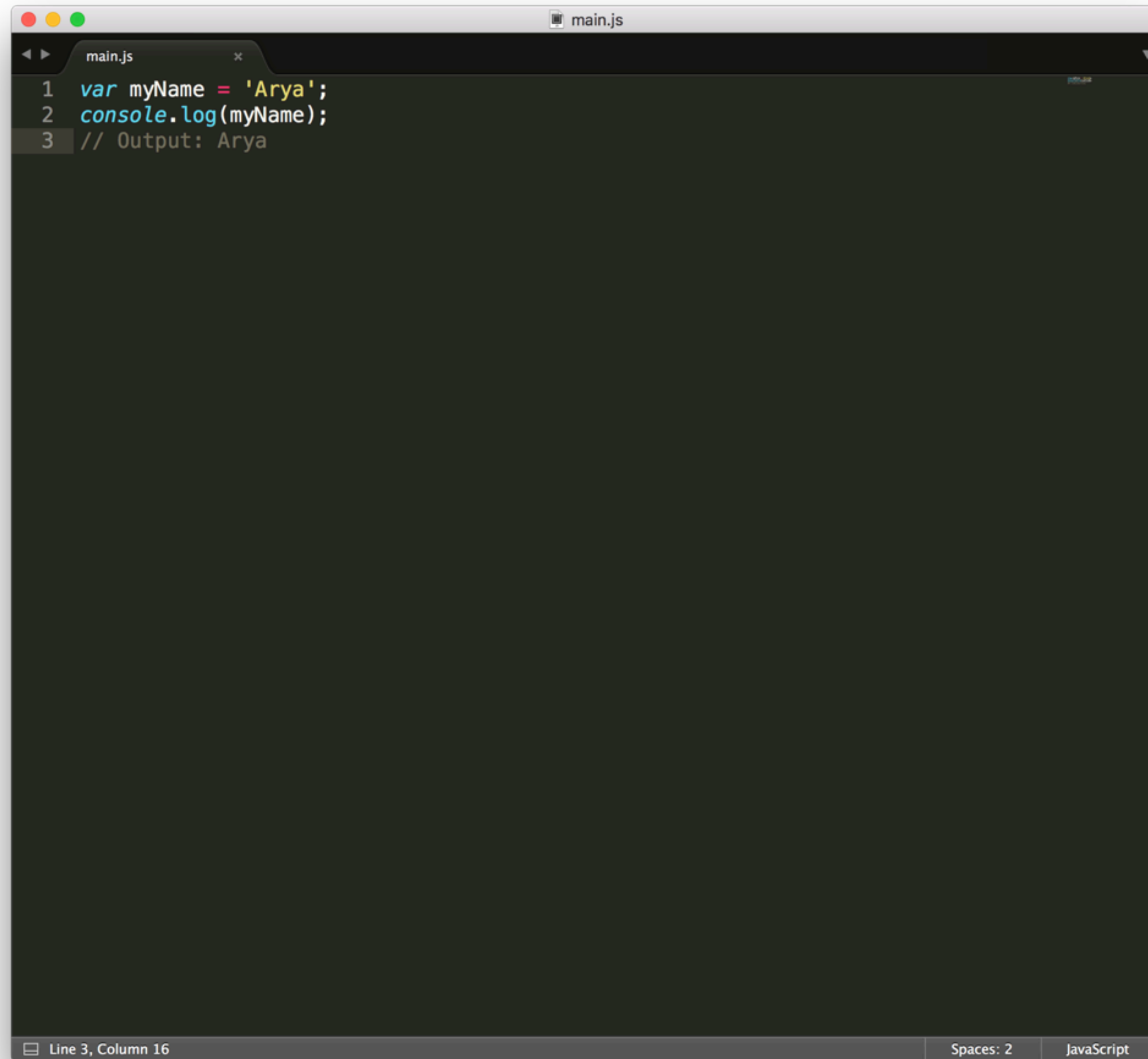
A multi-line comment will comment out multiple lines, and is denoted with **/*** to begin the comment, and ***/** to end the comment.

To write programs in JavaScript, we'll need to make our code reusable.

Part of making code reusable is removing the data we want to perform some logic on, leaving only the logic. Then we can use our logic on any data.

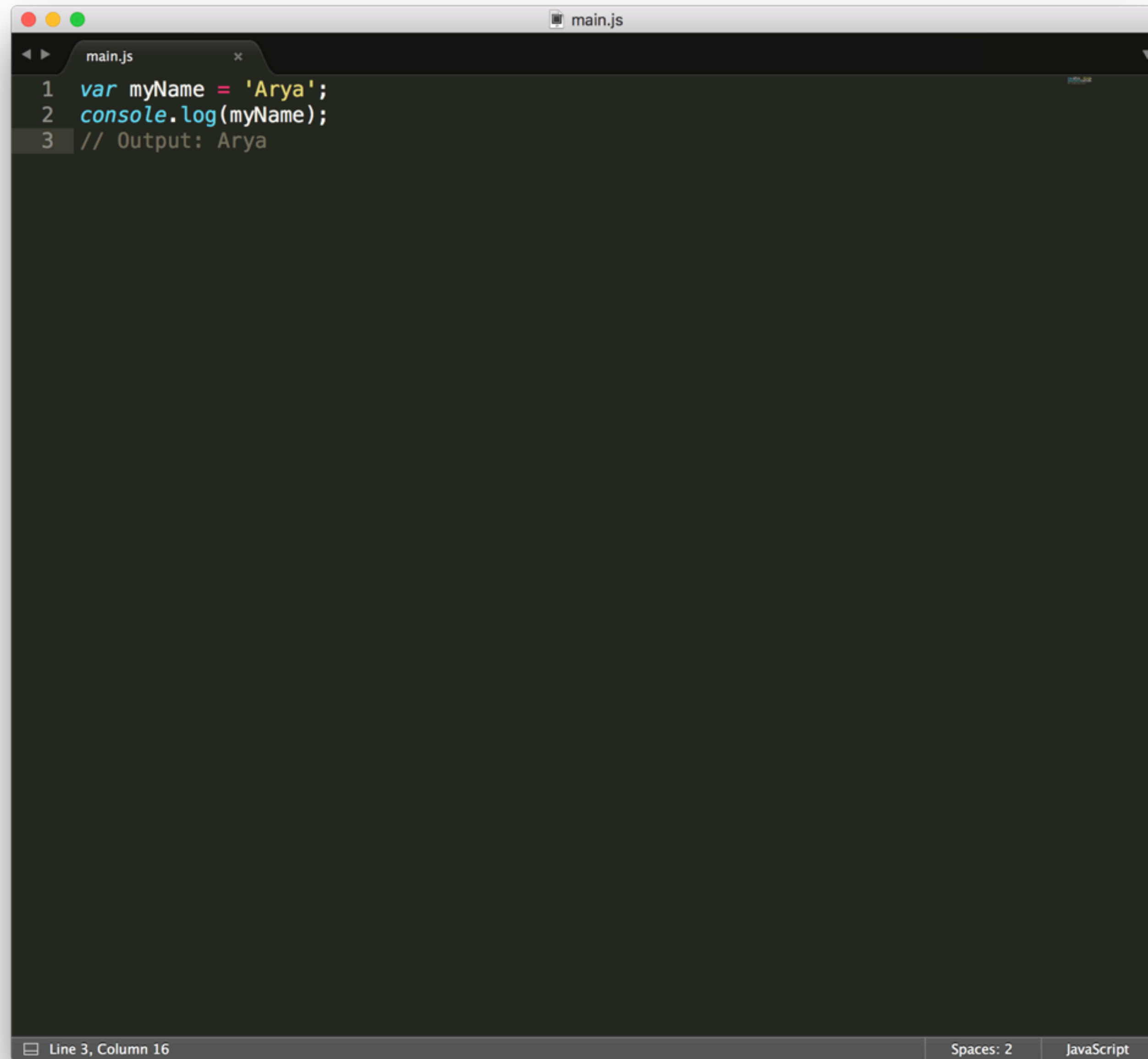
A first step toward this goal is using variables.

Variables allow us to assign data to a word, then we can use that word within our program instead of the data.



```
1 var myName = 'Arya';
2 console.log(myName);
3 // Output: Arya
```

Here is how you declare a variable. Let's dissect that statement and look at its parts.

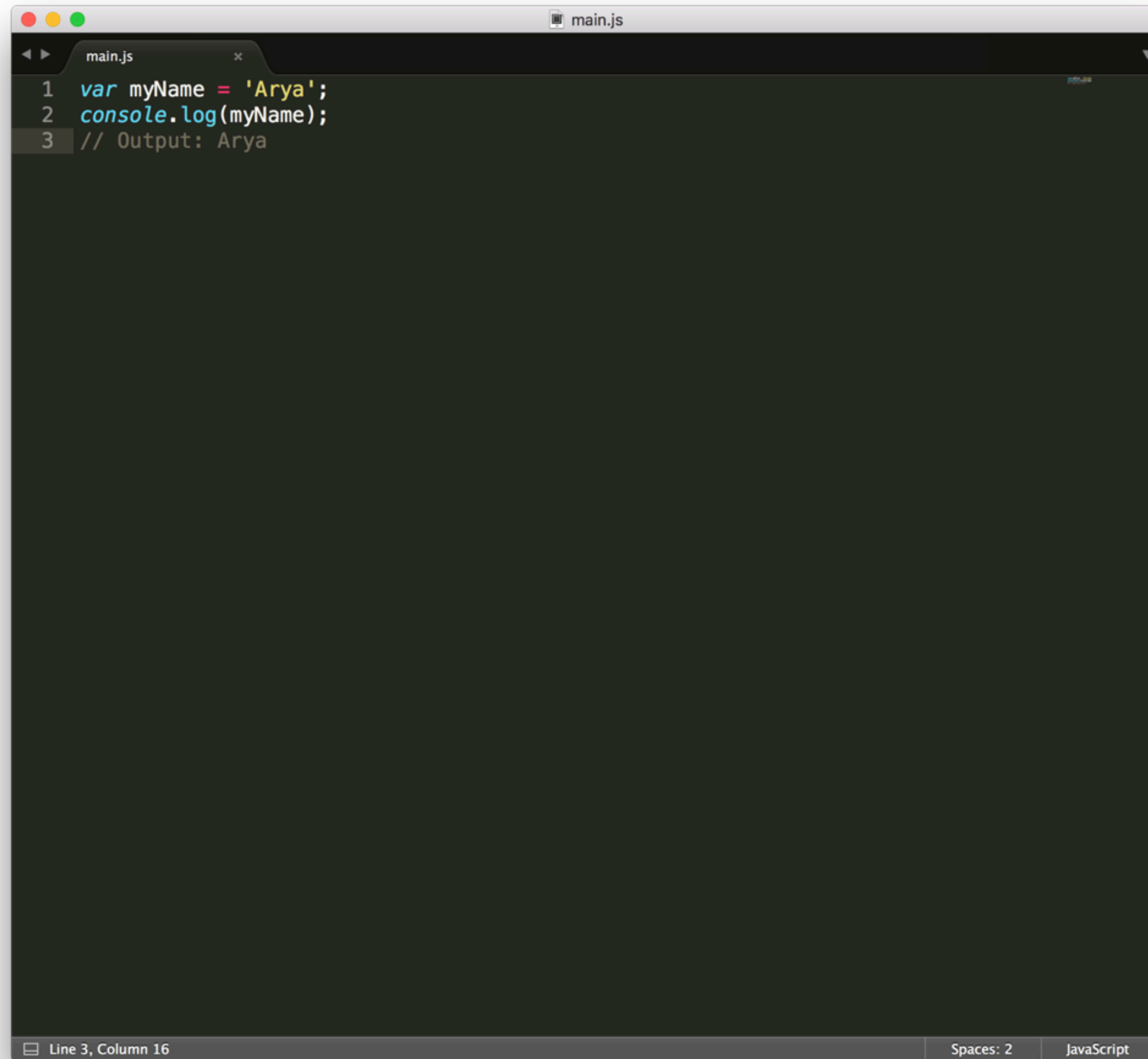
A screenshot of a code editor window with a dark theme. The window has a title bar with three colored buttons (red, yellow, green) and a tab labeled 'main.js'. The code is as follows:

```
1 var myName = 'Arya';  
2 console.log(myName);  
3 // Output: Arya
```

The status bar at the bottom shows 'Line 3, Column 16', 'Spaces: 2', and 'JavaScript'.

var, short for variable, is the JavaScript *keyword* that will create a new variable for us.

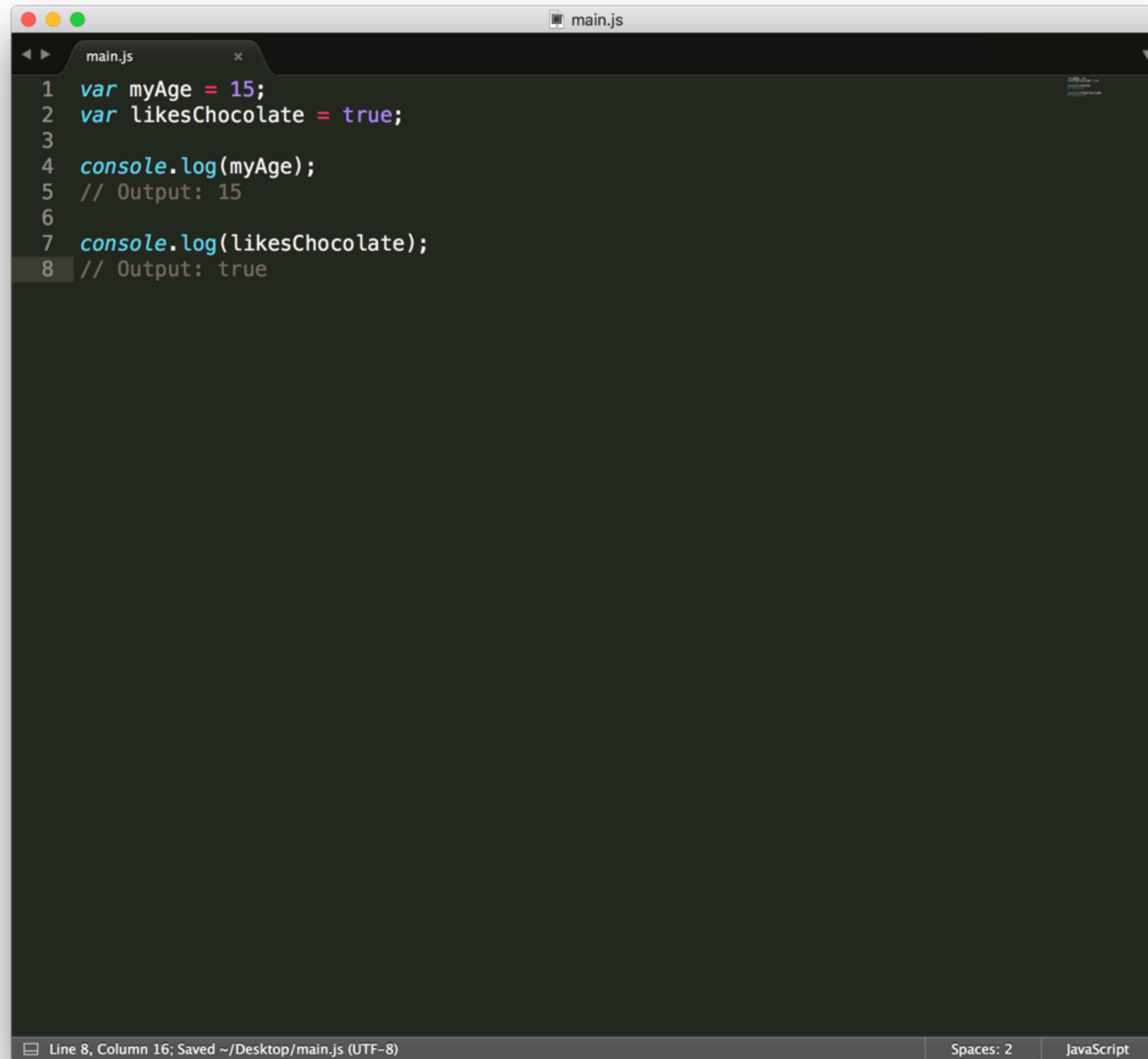
myName is chosen by a developer. Notice that the word has no spaces, and each new word is capitalized. This is a common convention in JavaScript, and is called *camelCase*.



```
1 var myName = 'Arya';
2 console.log(myName);
3 // Output: Arya
```

= means to *assign* whatever's next to the variable.

'Arya' is the *value* that the equals **=** assigns into the variable **myName**.

A screenshot of a code editor window titled 'main.js'. The editor has a dark theme and shows the following JavaScript code:

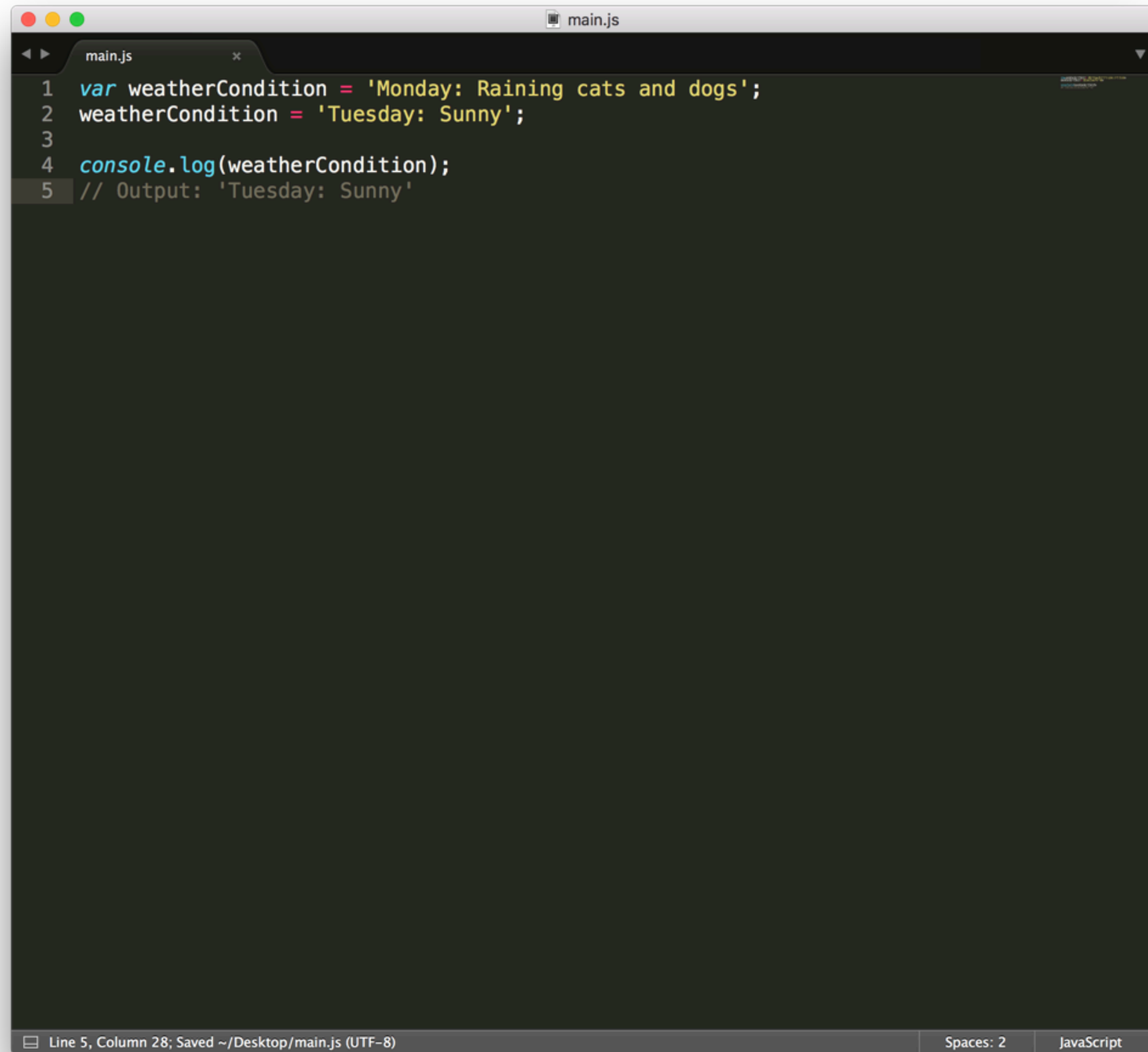
```
1 var myAge = 15;
2 var likesChocolate = true;
3
4 console.log(myAge);
5 // Output: 15
6
7 console.log(likesChocolate);
8 // Output: true
```

The code is syntax-highlighted. The status bar at the bottom indicates 'Line 8, Column 16; Saved ~/Desktop/main.js (UTF-8)', 'Spaces: 2', and 'JavaScript'.

Variables can hold any data type, like strings, numbers, and Booleans. They can also hold data types that we have not learned yet, like arrays, functions and objects (more on that later).

Variables are useful in two ways:

- 1. They allow us to use the same value over and over, without having to write a string or other data type over and over.**
- 2. More importantly, we can assign variables different values that can be read and changed by the program without altering our code.**



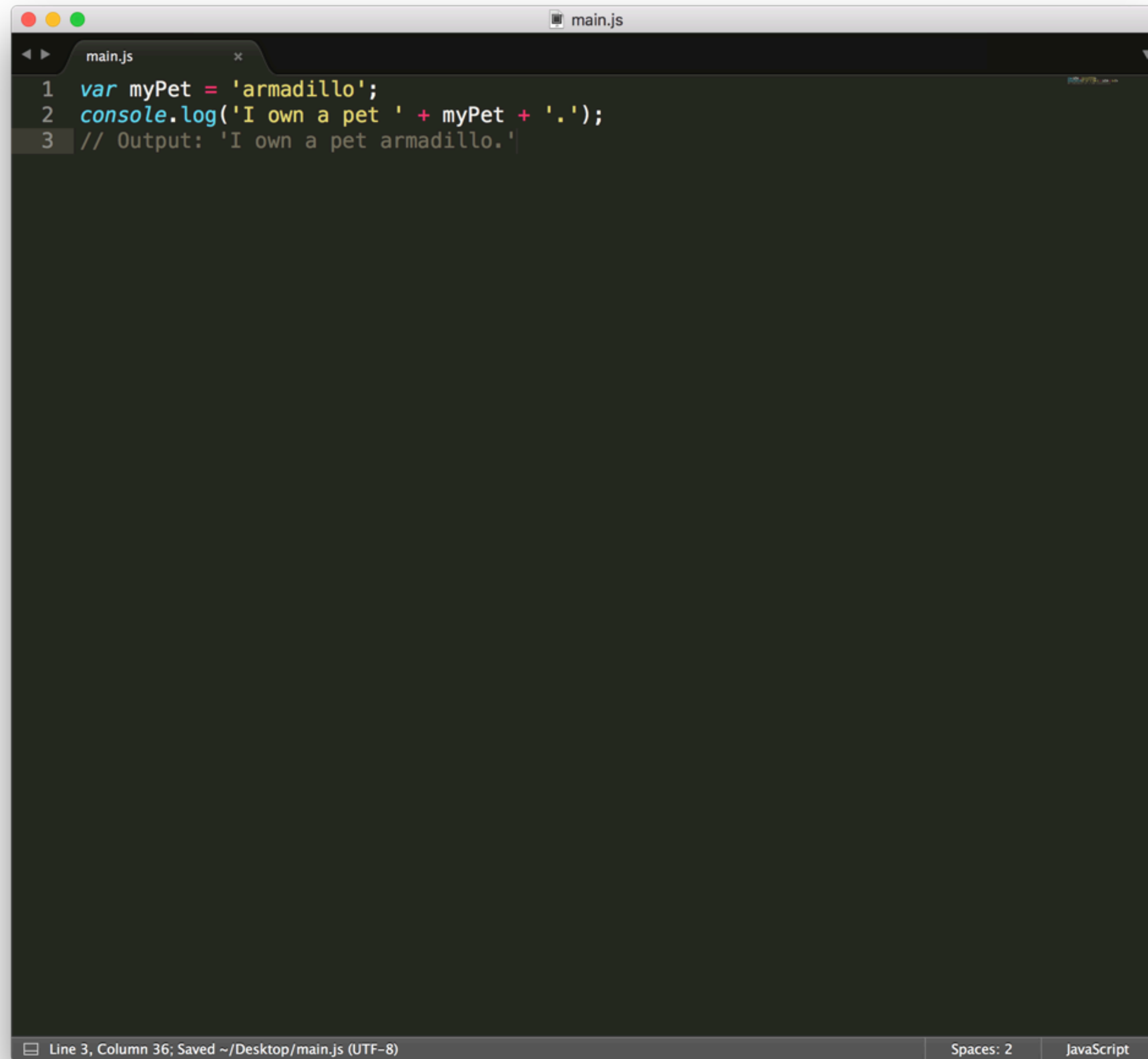
```
1 var weatherCondition = 'Monday: Raining cats and dogs';  
2 weatherCondition = 'Tuesday: Sunny';  
3  
4 console.log(weatherCondition);  
5 // Output: 'Tuesday: Sunny'
```

The screenshot shows a code editor window titled 'main.js'. The code consists of five lines: line 1 declares a variable 'weatherCondition' with the value 'Monday: Raining cats and dogs'; line 2 reassigns the variable to 'Tuesday: Sunny'; line 3 is a blank line; line 4 logs the current value of 'weatherCondition' to the console; and line 5 is a comment indicating the expected output is 'Tuesday: Sunny'. The status bar at the bottom shows 'Line 5, Column 28; Saved ~/Desktop/main.js (UTF-8)', 'Spaces: 2', and 'JavaScript'.

We can change a variable's value if we want.

In the previous example, we put strings into variables. Now, let's put a variable's value into a string.

Putting a variable in a string uses concepts we've already learned. The JavaScript term for this idea is *interpolation*.



```
1 var myPet = 'armadillo';  
2 console.log('I own a pet ' + myPet + '.');  
3 // Output: 'I own a pet armadillo.'
```

The screenshot shows a code editor window titled 'main.js'. The code consists of three lines: a variable declaration, a console log statement using string concatenation, and a comment showing the expected output. The editor has a dark theme and a status bar at the bottom indicating the current line and column, as well as the file's encoding and the language used.

We can use the **+** operator from earlier to interpolate (insert) a variable into a string.

In programming, making decisions with code is called *control flow*.

For instance, if we were making a choose-your-own-adventure game, we'd need to program a way for a user to choose which plot line they'd like to pursue. Control flow statements enable JavaScript to make those decisions by executing different code based on a condition.

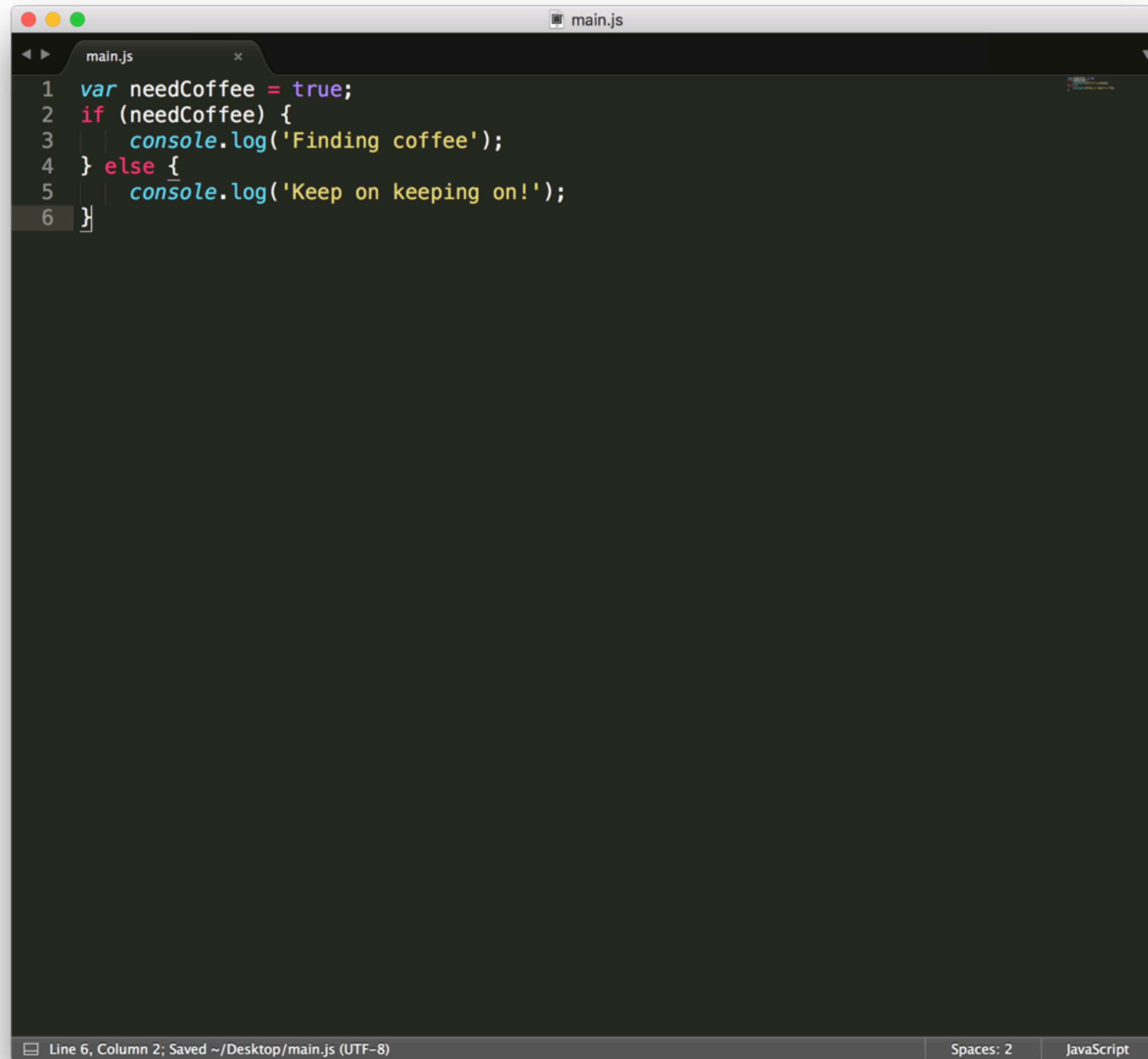
If you think about what we've been doing so far,
we've been writing instructions for computers.

That's all programming really is: a list of instructions
for computers.

The main difficulty of being a developer is
translating our ideas in *human-speak* into ideas
in *computer-speak*.

Many decisions we make everyday boil down to this sentence in some form:

"If something is true, let's do option 1, or else, if it is false, let's do option 2."



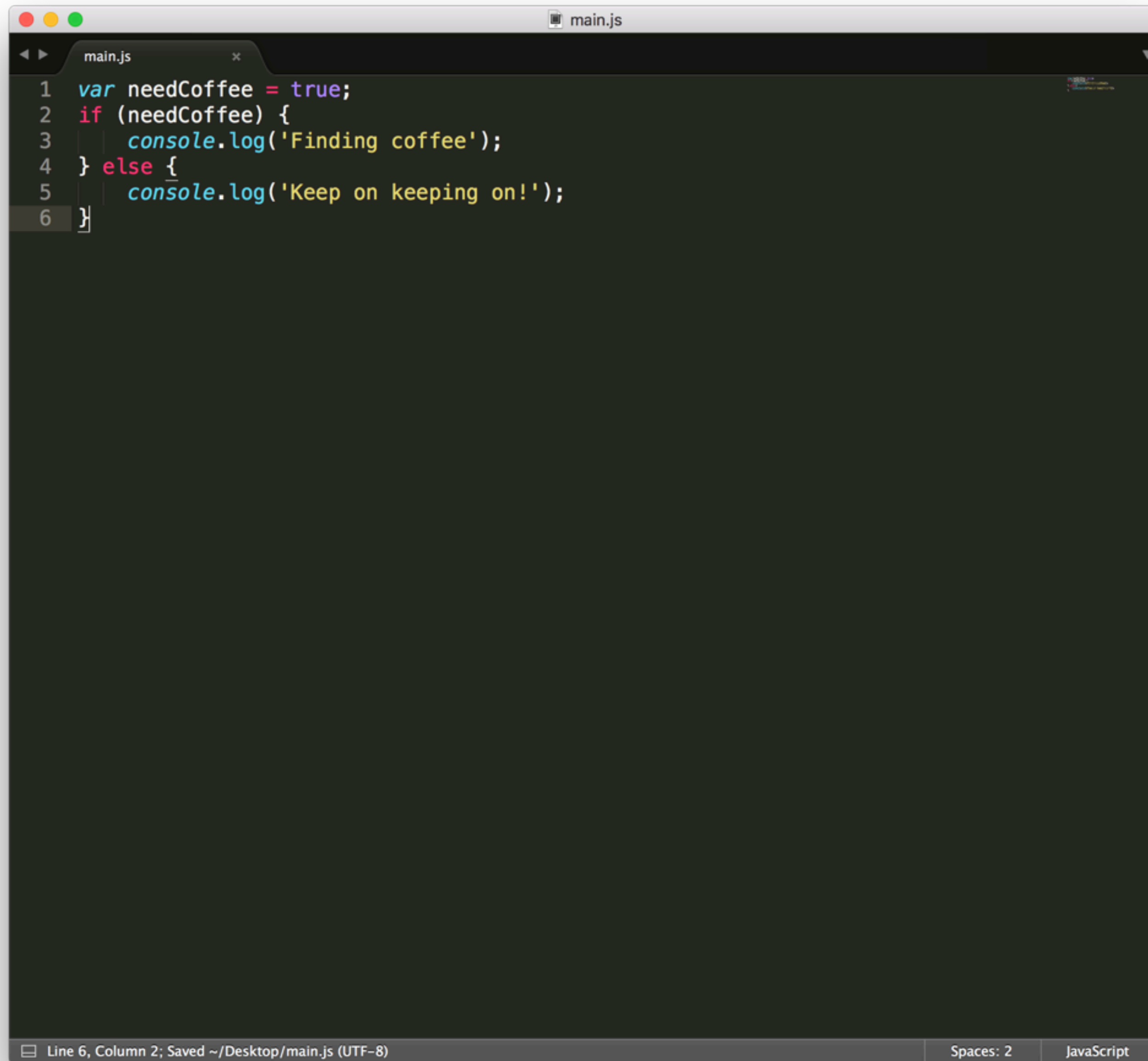
```
1 var needCoffee = true;
2 if (needCoffee) {
3   console.log('Finding coffee');
4 } else {
5   console.log('Keep on keeping on!');
6 }
```

The screenshot shows a code editor window titled 'main.js'. The code is written in JavaScript and uses an if-else statement to check the value of 'needCoffee'. The code is as follows:

```
1 var needCoffee = true;
2 if (needCoffee) {
3   console.log('Finding coffee');
4 } else {
5   console.log('Keep on keeping on!');
6 }
```

The status bar at the bottom indicates 'Line 6, Column 2; Saved ~/Desktop/main.js (UTF-8)', 'Spaces: 2', and 'JavaScript'.

1. If the variable **needCoffee** is **true**, JavaScript will run one code block, and if a variable is **false**, it will run another.



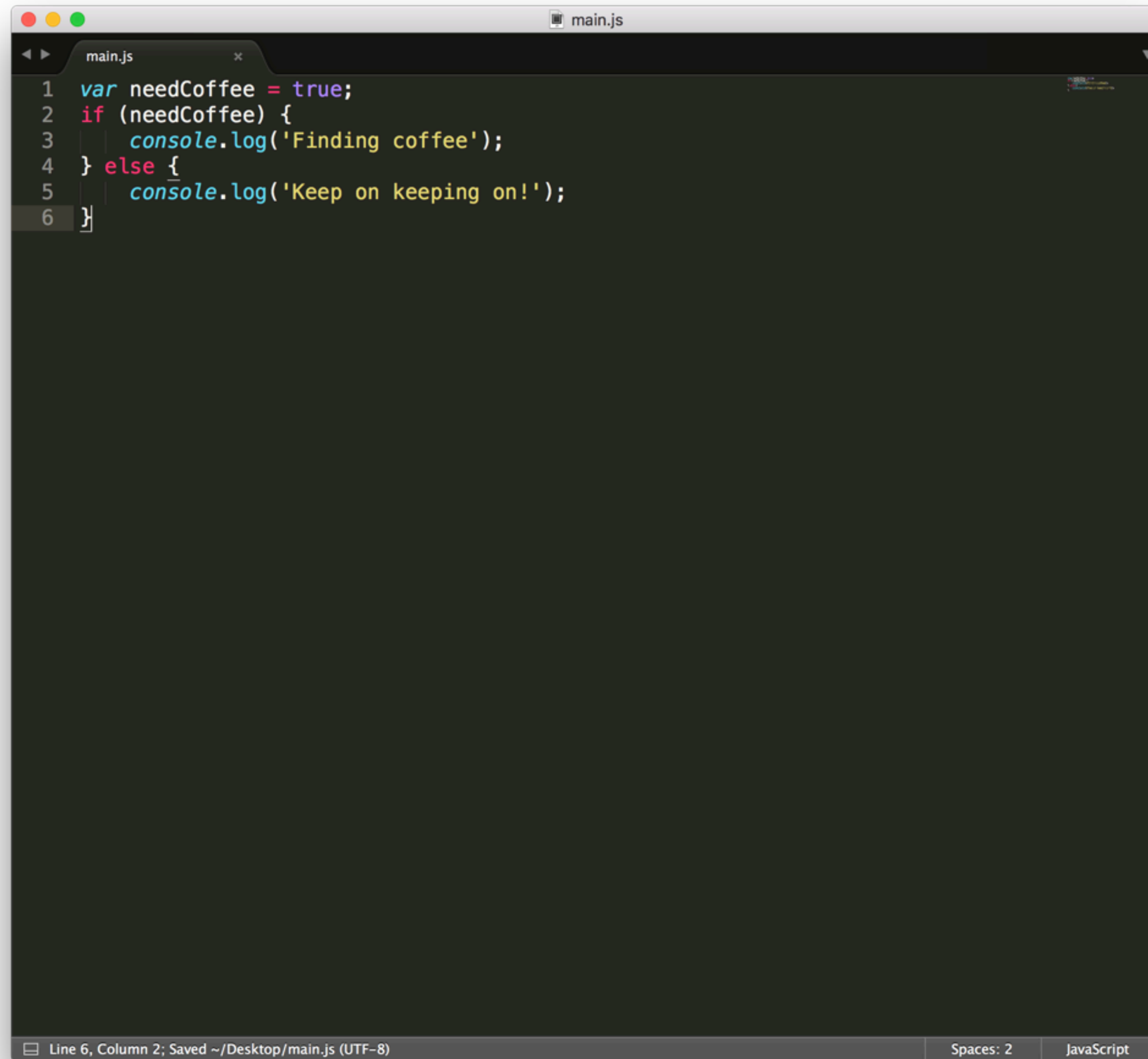
```
1 var needCoffee = true;
2 if (needCoffee) {
3   console.log('Finding coffee');
4 } else {
5   console.log('Keep on keeping on!');
6 }
```

The screenshot shows a code editor window titled 'main.js'. The code is as follows:

```
1 var needCoffee = true;
2 if (needCoffee) {
3   console.log('Finding coffee');
4 } else {
5   console.log('Keep on keeping on!');
6 }
```

The status bar at the bottom indicates 'Line 6, Column 2; Saved ~/Desktop/main.js (UTF-8)', 'Spaces: 2', and 'JavaScript'.

2. **needCoffee** is the *condition* we are checking inside the **if**'s parentheses. Since it is equal to **true**, our program will run the code between the first opening curly brace **{** (line 2) and the first closing curly brace **}** (line 4). It will completely ignore the **else { ... }** part. In this case, we'd see **'Finding coffee'** log to the console.



```
1 var needCoffee = true;
2 if (needCoffee) {
3   console.log('Finding coffee');
4 } else {
5   console.log('Keep on keeping on!');
6 }
```

Line 6, Column 2; Saved ~/Desktop/main.js (UTF-8) Spaces: 2 JavaScript

3. If we adjusted **needCoffee** to be **false**, *only* the **else's console.log** will run.

if/else statements are how we can process yes/no questions programmatically.

if/else statements are made even more powerful
with *comparison operators*.

There are two comparisons you might be familiar with:

Less than: <

Greater than: >

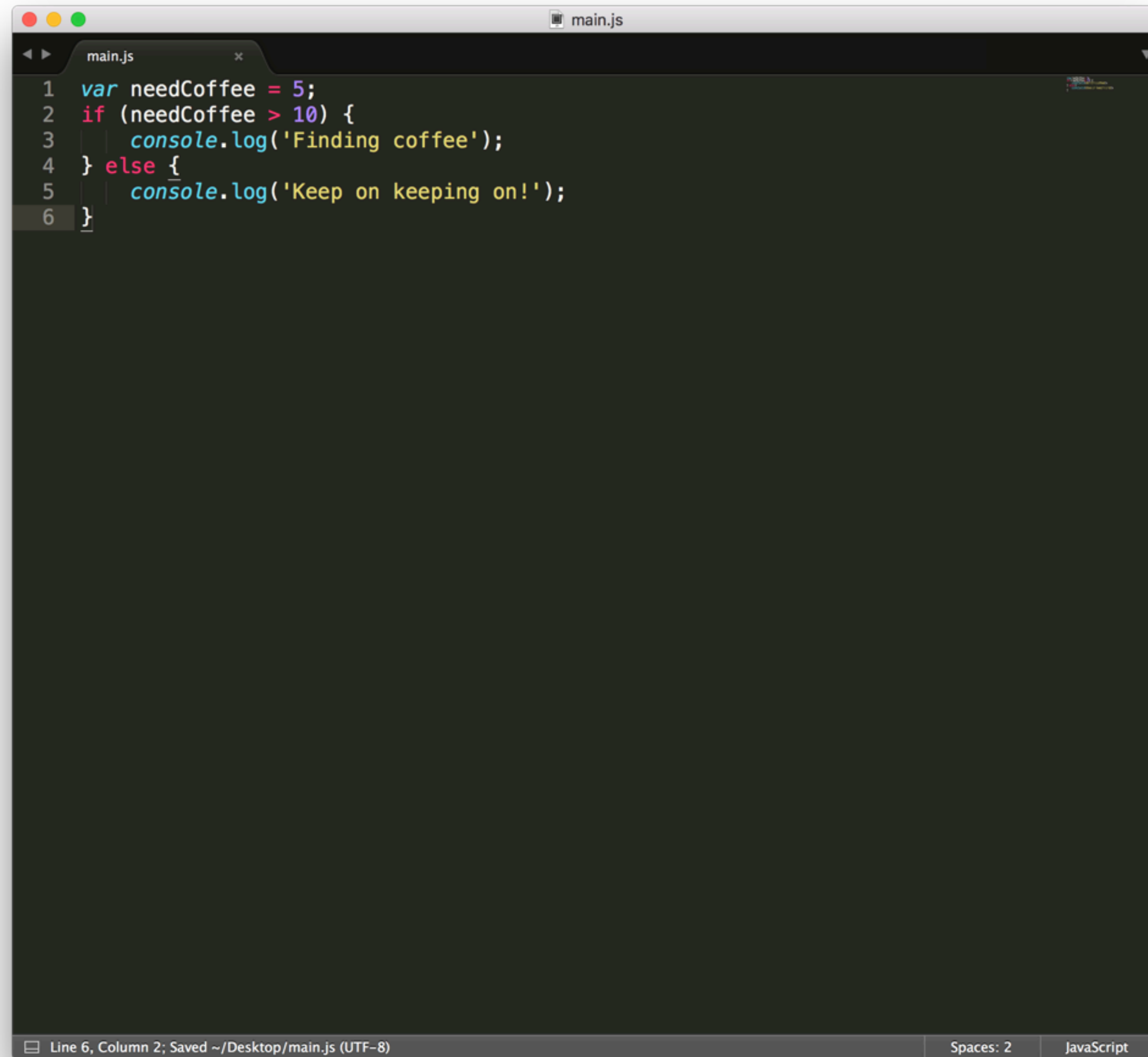
You may also recognize these:

Less than or equal to: \leq

Greater than or equal to: \geq

Comparisons need two things to compare and they will always return a boolean (**true** or **false**).

How can we use comparisons and an **if/else** statement to see if it's time to get coffee?



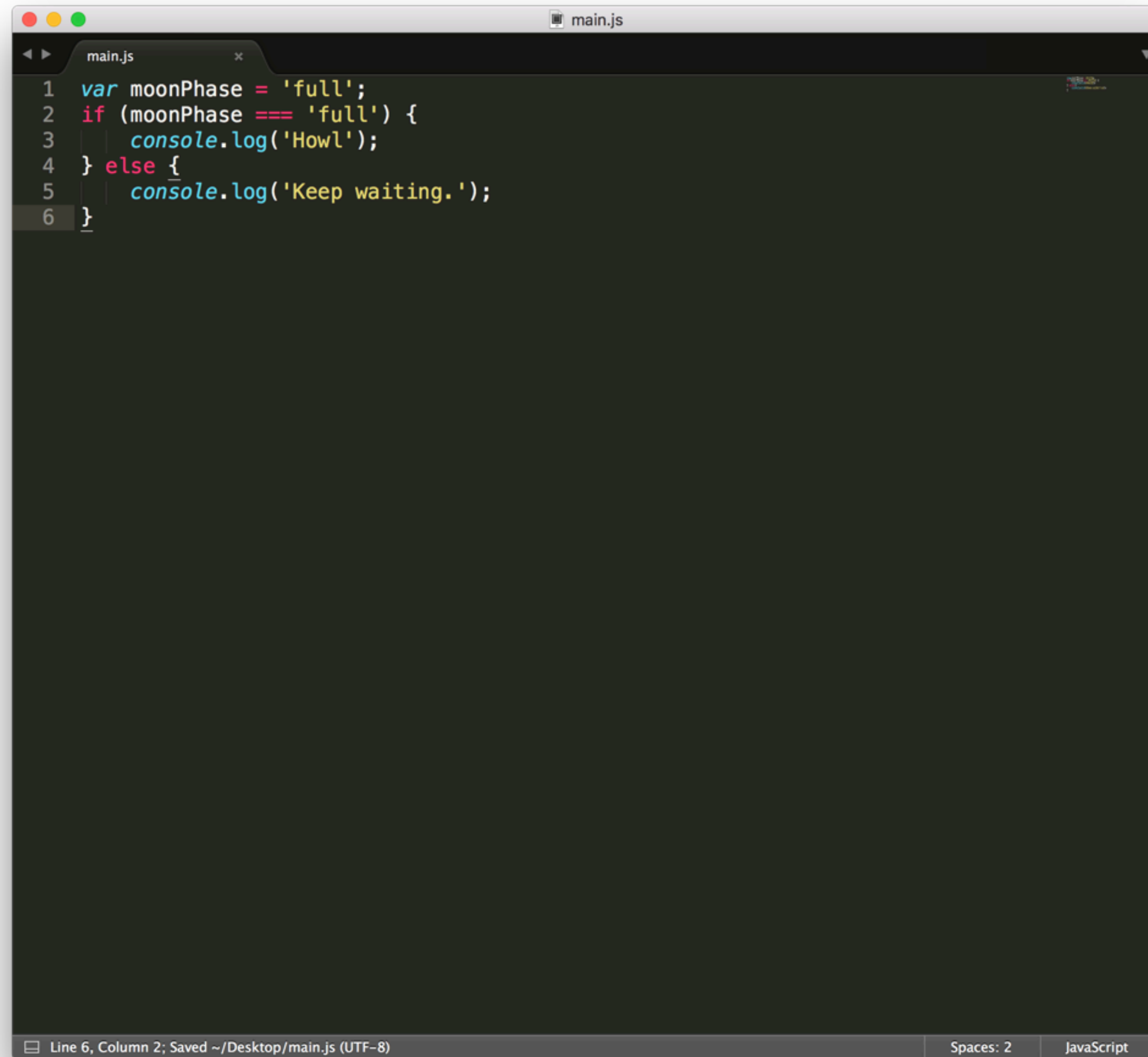
```
1 var needCoffee = 5;
2 if (needCoffee > 10) {
3 |   console.log('Finding coffee');
4 } else {
5 |   console.log('Keep on keeping on!');
6 | }
```

Line 6, Column 2; Saved ~/Desktop/main.js (UTF-8) Spaces: 2 JavaScript

There are two more extremely useful comparisons we can make. Often times, we might want to check if two things are equal, or if they are not.

1. To check if two things equal each other, we can use `==` (three equals in a row).

2. To check if two things *do not* equal each other, we can write `!=` (exclamation with two equals in a row).

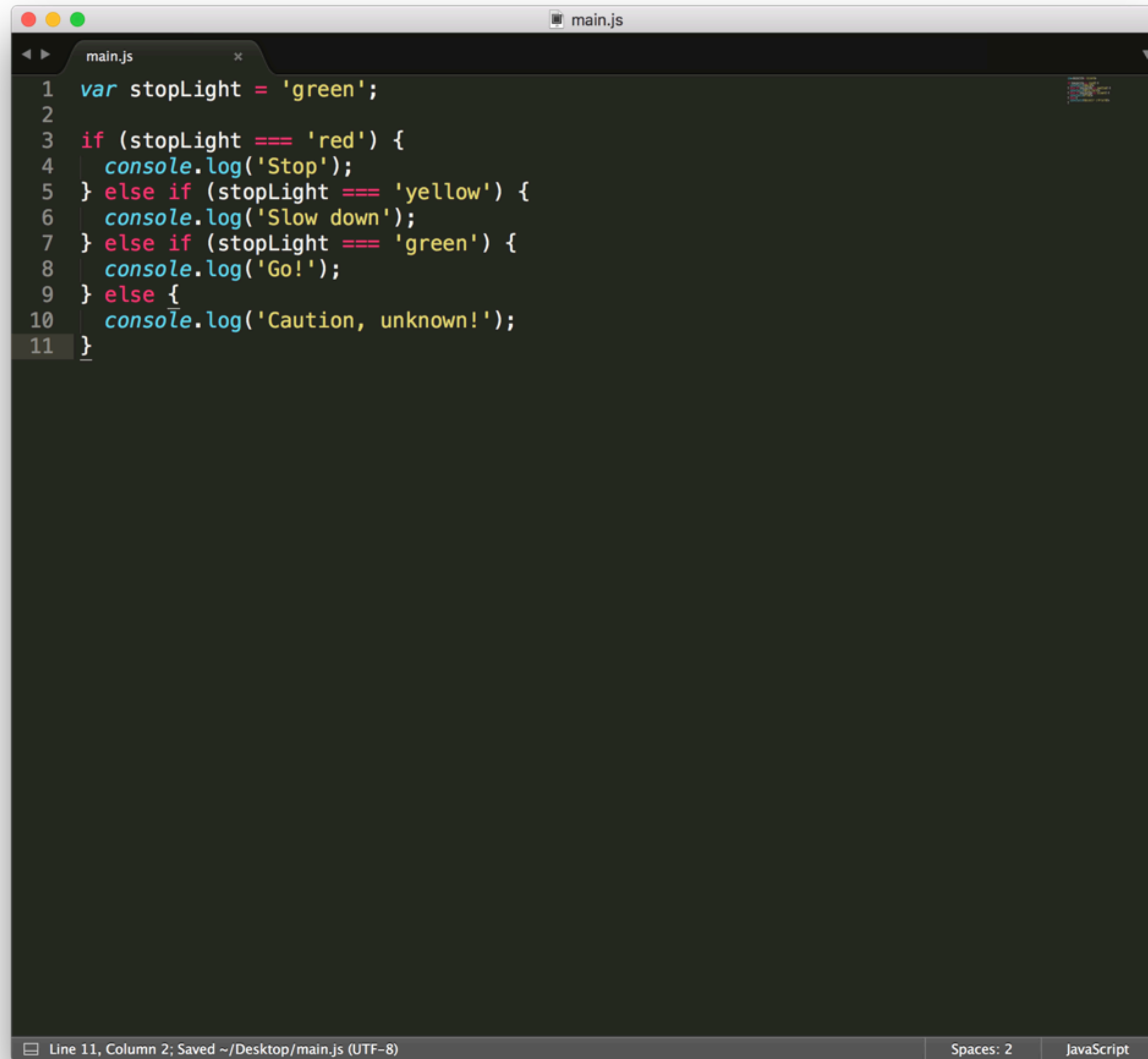


```
1 var moonPhase = 'full';
2 if (moonPhase === 'full') {
3     console.log('Howl');
4 } else {
5     console.log('Keep waiting. ');
6 }
```

Line 6, Column 2; Saved ~/Desktop/main.js (UTF-8) Spaces: 2 JavaScript

if/else statements are either this or that for us right now. They answer questions that are either yes or no.

What can we do if we have a question that has multiple yes conditions, or multiple no conditions?



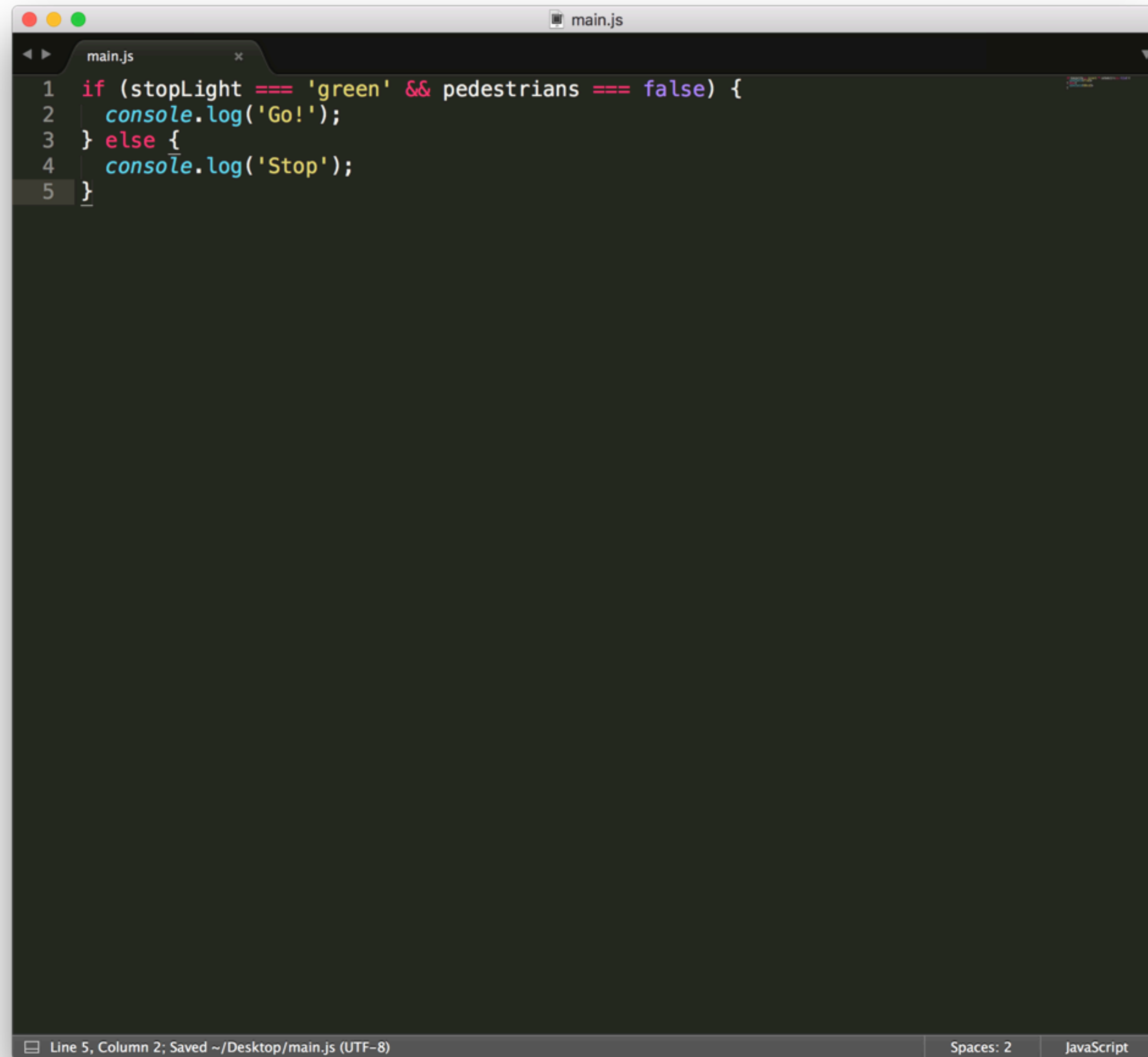
```
1 var stopLight = 'green';
2
3 if (stopLight === 'red') {
4   console.log('Stop');
5 } else if (stopLight === 'yellow') {
6   console.log('Slow down');
7 } else if (stopLight === 'green') {
8   console.log('Go!');
9 } else {
10  console.log('Caution, unknown!');
11 }
```

The screenshot shows a code editor window titled 'main.js'. The code is written in JavaScript and implements a traffic light simulation. It starts by declaring a variable 'stopLight' with the value 'green'. Then, it uses a series of 'if', 'else if', and 'else' statements to log different messages based on the value of 'stopLight'. The messages are 'Stop' for red, 'Slow down' for yellow, 'Go!' for green, and 'Caution, unknown!' for any other value. The code is formatted with syntax highlighting and line numbers.

We can add more conditions to our **if/else** statement with: **else if**

In English, sometimes we say "both of these things" or "either one of these things." Let's translate those phrases into JavaScript with some special operators called *logical operators*.

1. To say "both must be true," we can use **&&**.
2. To say "either can be true," we can use **||**.
3. To say "I want to make sure this is the opposite of what it really is," we can use **!**.
4. To say "these should not be equal to each other," we can use **!=**.



```
1 if (stopLight === 'green' && pedestrians === false) {
2   console.log('Go!');
3 } else {
4   console.log('Stop');
5 }
```

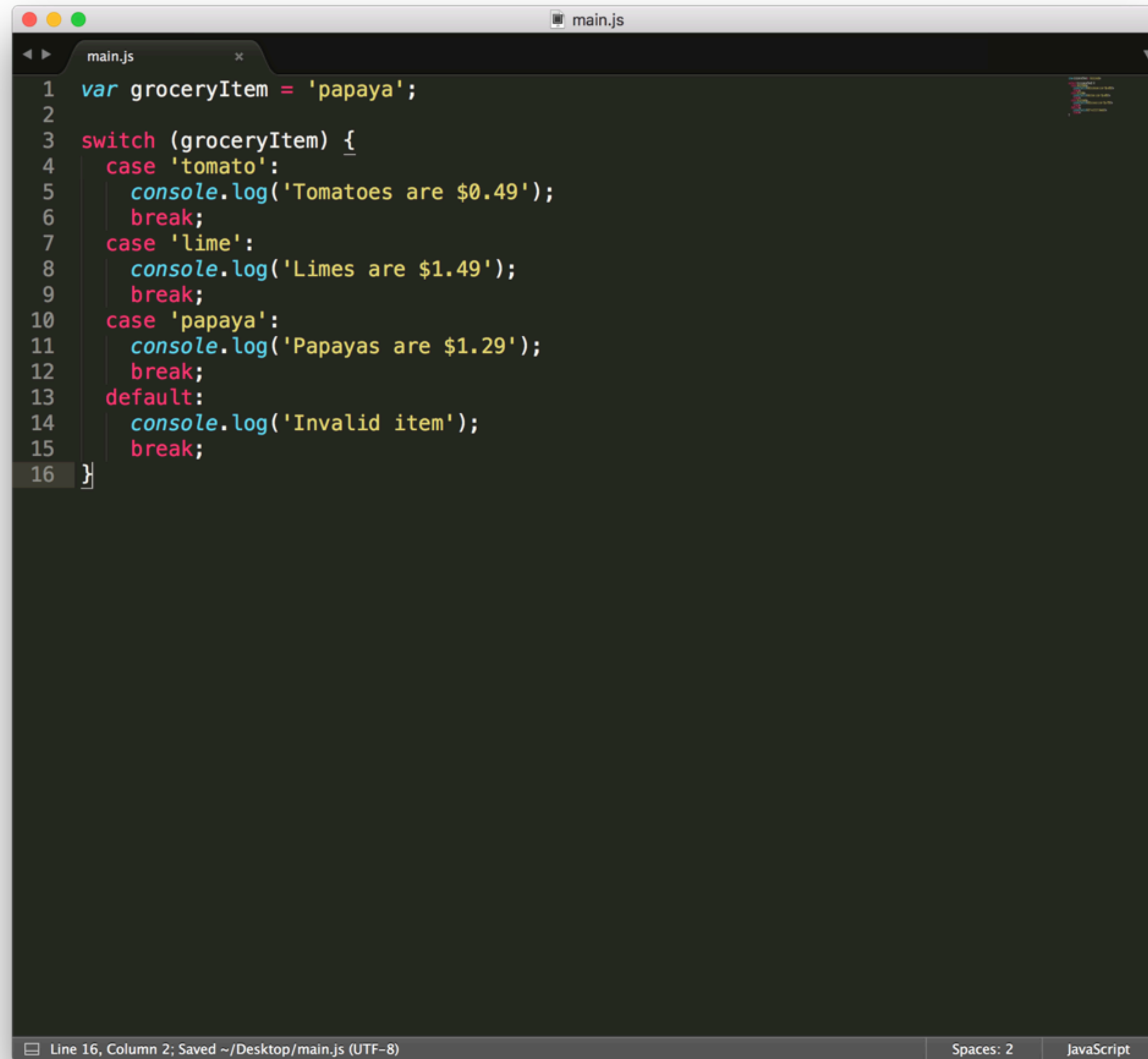
Line 5, Column 2; Saved ~/Desktop/main.js (UTF-8) Spaces: 2 JavaScript

Before we move on, let's circle back to **else if** statements.

Using **else if** is a great tool for when we have a few different conditions we'd like to consider.

else if is limited however. If we want to write a program with 25 different conditions, like a JavaScript cash register, we'd have to write *a lot* of code, and it can be difficult to read and understand.

To deal with times when you need many **else if** conditions, we can turn to a **switch** statement to write more concise and readable code.



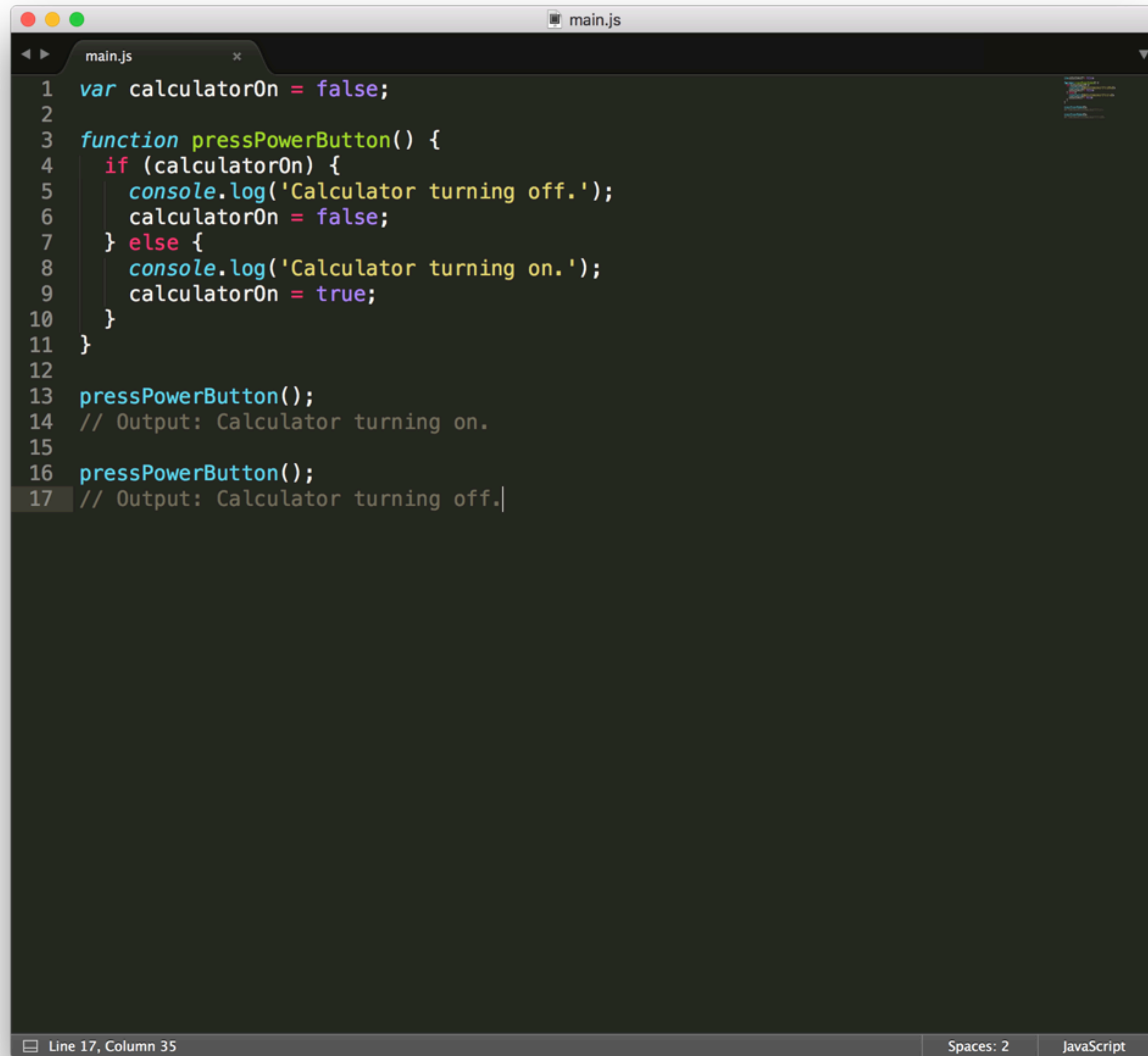
```
1 var groceryItem = 'papaya';
2
3 switch (groceryItem) {
4   case 'tomato':
5     console.log('Tomatoes are $0.49');
6     break;
7   case 'lime':
8     console.log('Limes are $1.49');
9     break;
10  case 'papaya':
11    console.log('Papayas are $1.29');
12    break;
13  default:
14    console.log('Invalid item');
15    break;
16 }
```

Line 16, Column 2; Saved ~/Desktop/main.js (UTF-8) Spaces: 2 JavaScript

A function is a block of code designed to perform a task.

Functions are like recipes. They take data or variables, perform a set of tasks on them, and then return the result.

The beauty of functions is that they allow us to write a chunk of code once, then we can reuse it over and over without writing the same code over and over.



```
1 var calculatorOn = false;
2
3 function pressPowerButton() {
4   if (calculatorOn) {
5     console.log('Calculator turning off.');
```

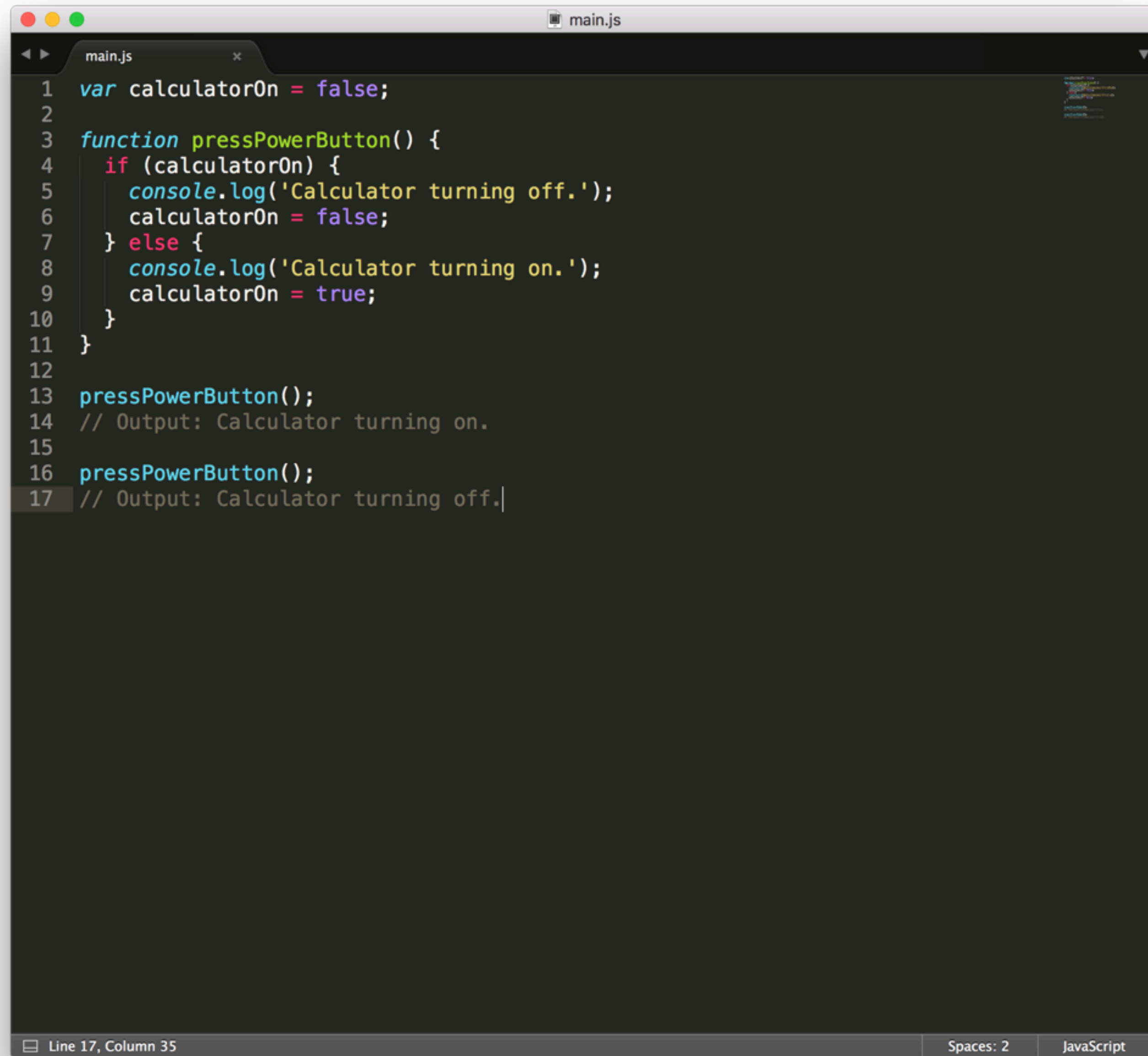
The image shows a code editor window titled 'main.js'. The code is written in JavaScript and defines a function 'pressPowerButton()' that toggles a boolean variable 'calculatorOn'. The function uses an 'if' statement to check the current state of 'calculatorOn'. If it is 'true', it logs 'Calculator turning off.' and sets 'calculatorOn' to 'false'. If it is 'false', it logs 'Calculator turning on.' and sets 'calculatorOn' to 'true'. The code is executed twice: first, 'pressPowerButton()' is called, resulting in the output 'Calculator turning on.' (commented on line 14). Then, 'pressPowerButton()' is called again, resulting in the output 'Calculator turning off.' (commented on line 17). The editor interface includes a tab for 'main.js', a line and column indicator at the bottom left ('Line 17, Column 35'), and settings for 'Spaces: 2' and 'JavaScript' at the bottom right.

```
6     calculatorOn = false;
7   } else {
8     console.log('Calculator turning on.');
```

The image shows a code editor window titled 'main.js'. The code is written in JavaScript and defines a function 'pressPowerButton()' that toggles a boolean variable 'calculatorOn'. The function uses an 'if' statement to check the current state of 'calculatorOn'. If it is 'true', it logs 'Calculator turning off.' and sets 'calculatorOn' to 'false'. If it is 'false', it logs 'Calculator turning on.' and sets 'calculatorOn' to 'true'. The code is executed twice: first, 'pressPowerButton()' is called, resulting in the output 'Calculator turning on.' (commented on line 14). Then, 'pressPowerButton()' is called again, resulting in the output 'Calculator turning off.' (commented on line 17). The editor interface includes a tab for 'main.js', a line and column indicator at the bottom left ('Line 17, Column 35'), and settings for 'Spaces: 2' and 'JavaScript' at the bottom right.

```
9     calculatorOn = true;
10  }
11 }
12
13 pressPowerButton();
14 // Output: Calculator turning on.
15
16 pressPowerButton();
17 // Output: Calculator turning off. |
```

How does this code work?



```
1 var calculatorOn = false;
2
3 function pressPowerButton() {
4   if (calculatorOn) {
5     console.log('Calculator turning off.');
```

The image shows a code editor window titled 'main.js'. The code is as follows:

```
1 var calculatorOn = false;
2
3 function pressPowerButton() {
4   if (calculatorOn) {
5     console.log('Calculator turning off.');
```

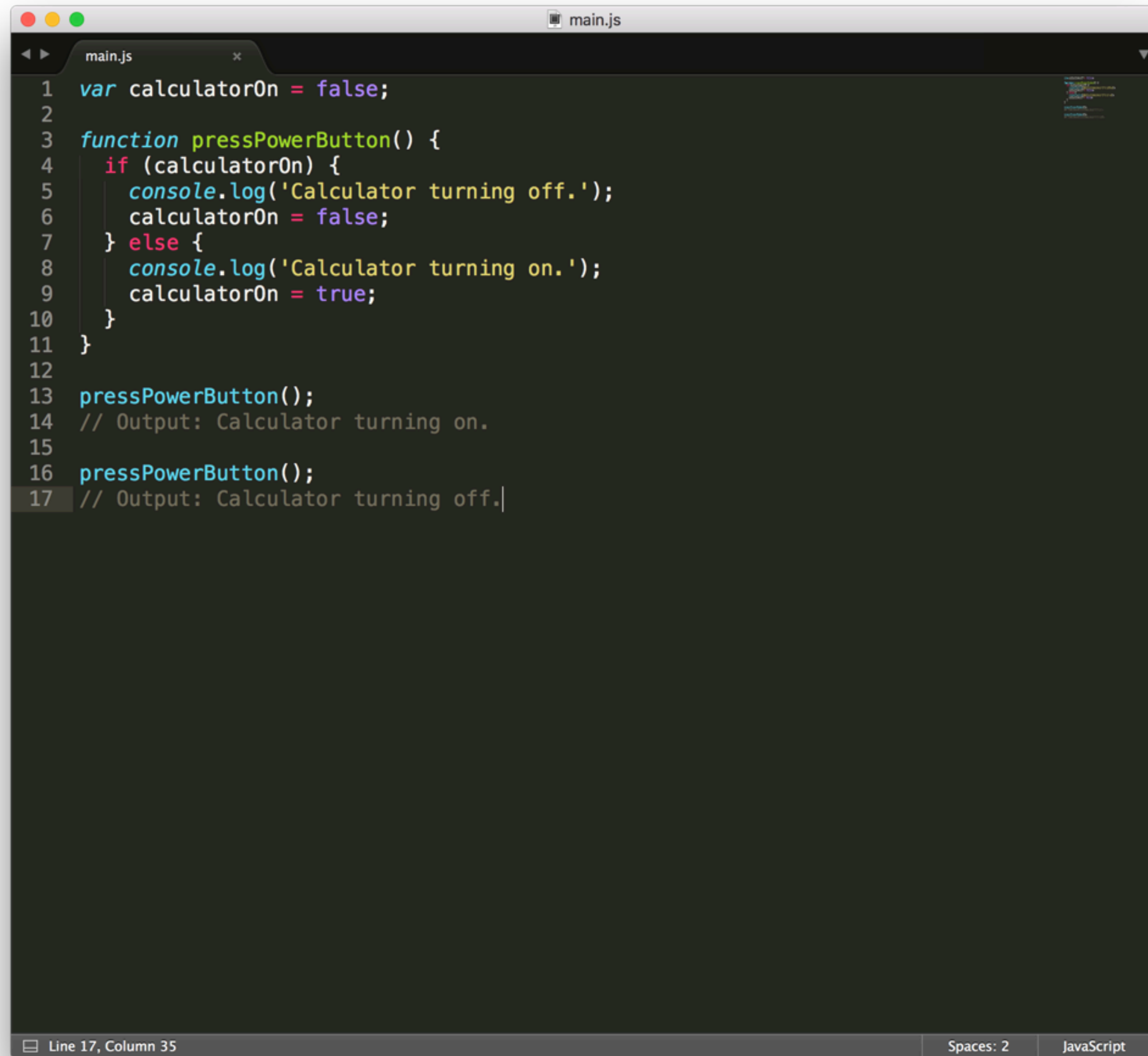
The code defines a variable `calculatorOn` set to `false`. It then defines a function `pressPowerButton` which checks the state of `calculatorOn`. If it's `true`, it logs 'Calculator turning off.' and sets `calculatorOn` to `false`. If it's `false`, it logs 'Calculator turning on.' and sets `calculatorOn` to `true`. The function is called twice: first, it logs 'Calculator turning on.' (commented), and second, it logs 'Calculator turning off.' (commented).

```
6     calculatorOn = false;
7   } else {
8     console.log('Calculator turning on.');
```

The status bar at the bottom indicates 'Line 17, Column 35', 'Spaces: 2', and 'JavaScript'.

```
9     calculatorOn = true;
10  }
11 }
12
13 pressPowerButton();
14 // Output: Calculator turning on.
15
16 pressPowerButton();
17 // Output: Calculator turning off. |
```

On line 3, there's a function
named **pressPowerButton**



```
1 var calculatorOn = false;
2
3 function pressPowerButton() {
4   if (calculatorOn) {
5     console.log('Calculator turning off.');
```

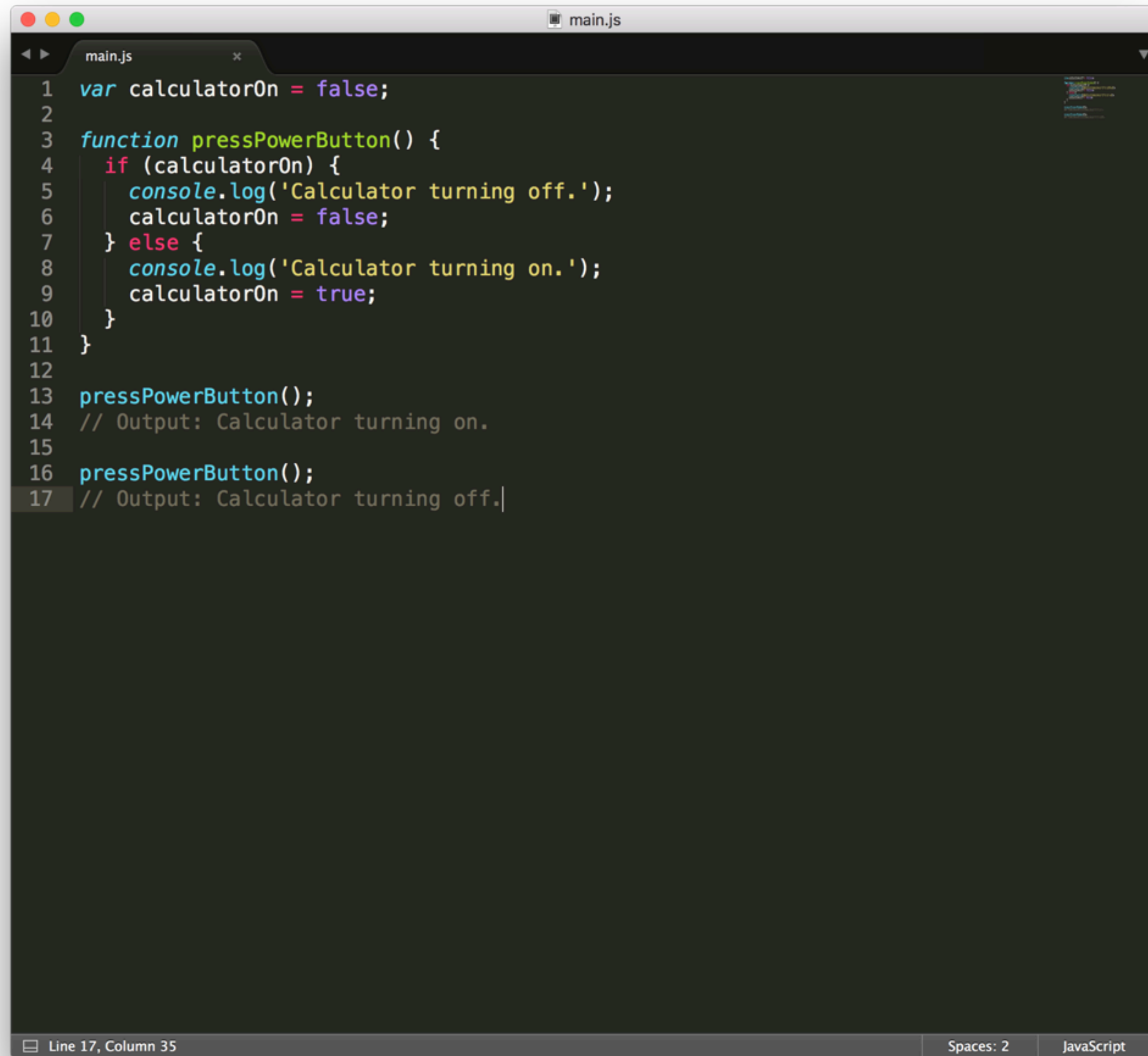
The image shows a code editor window titled 'main.js'. The code defines a variable 'calculatorOn' set to false, a function 'pressPowerButton' that toggles the state and logs messages, and two calls to the function. The status bar at the bottom indicates 'Line 17, Column 35', 'Spaces: 2', and 'JavaScript'.

```
6     calculatorOn = false;
7   } else {
8     console.log('Calculator turning on.');
```

The code continues with the function closing brace, two calls to 'pressPowerButton()', and a comment. The status bar at the bottom shows 'Line 17, Column 35', 'Spaces: 2', and 'JavaScript'.

```
9     calculatorOn = true;
10  }
11 }
12
13 pressPowerButton();
14 // Output: Calculator turning on.
15
16 pressPowerButton();
17 // Output: Calculator turning off. |
```

Functions begin with the
JavaScript keyword **function**.



```
1 var calculatorOn = false;
2
3 function pressPowerButton() {
4   if (calculatorOn) {
5     console.log('Calculator turning off.');
```

The image shows a code editor window titled 'main.js'. The code defines a variable 'calculatorOn' set to false, a function 'pressPowerButton' that toggles the state and logs messages, and two calls to the function with corresponding comments. The editor has a dark theme and a sidebar on the right.

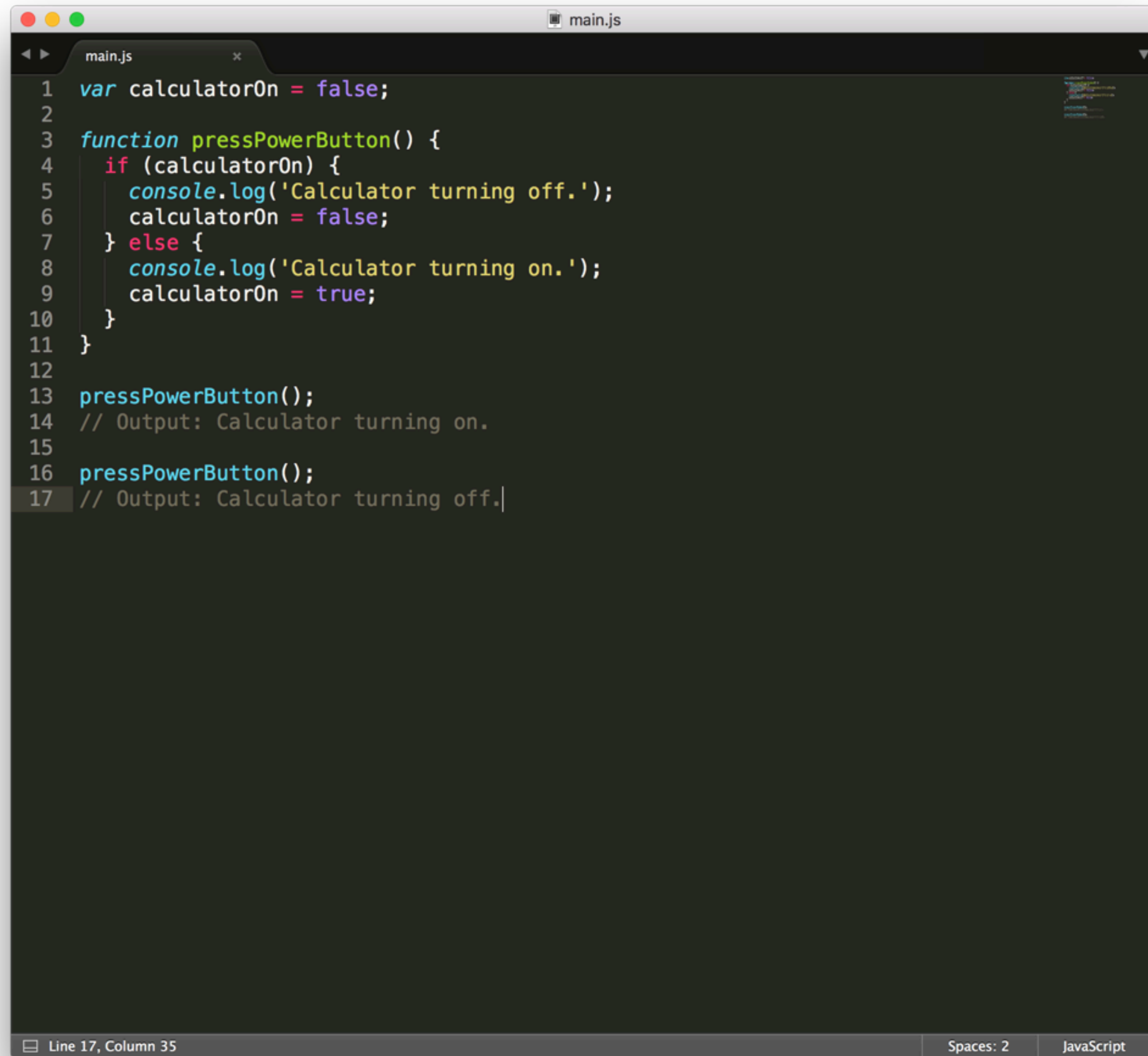
```
6     calculatorOn = false;
7   } else {
8     console.log('Calculator turning on.');
```

The code continues with the function closing brace, followed by two function calls and their comments.

```
9     calculatorOn = true;
10  }
11 }
12
13 pressPowerButton();
14 // Output: Calculator turning on.
15
16 pressPowerButton();
17 // Output: Calculator turning off.
```

The status bar at the bottom indicates 'Line 17, Column 35', 'Spaces: 2', and 'JavaScript'.

After **function** comes the name of the function. **pressPowerButton** is the name of the function. Notice there are no spaces in the name and each new word is capitalized.



```
1 var calculatorOn = false;
2
3 function pressPowerButton() {
4   if (calculatorOn) {
5     console.log('Calculator turning off.');
```

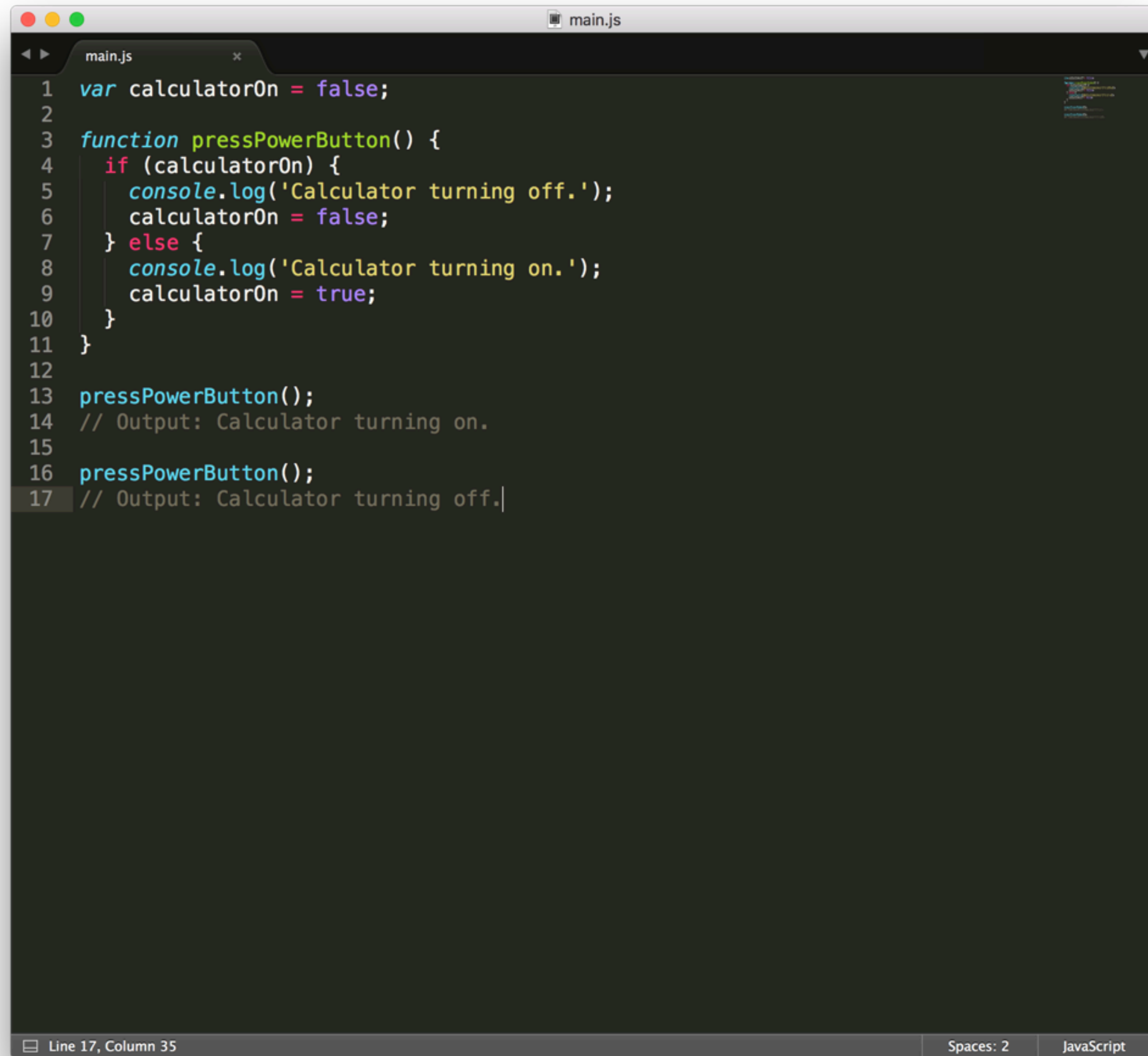
The image shows a code editor window titled 'main.js'. The code defines a variable 'calculatorOn' set to false, a function 'pressPowerButton' that toggles the state and logs messages, and two calls to the function. The status bar at the bottom indicates 'Line 17, Column 35', 'Spaces: 2', and 'JavaScript'.

```
6     calculatorOn = false;
7   } else {
8     console.log('Calculator turning on.');
```

The code continues with the function closing brace, followed by two calls to 'pressPowerButton()' with comments indicating the expected console output.

```
9     calculatorOn = true;
10  }
11 }
12
13 pressPowerButton();
14 // Output: Calculator turning on.
15
16 pressPowerButton();
17 // Output: Calculator turning off. |
```

After the function's name, comes parentheses **()**. We'll learn about these soon.



```
1 var calculatorOn = false;
2
3 function pressPowerButton() {
4   if (calculatorOn) {
5     console.log('Calculator turning off.');
```

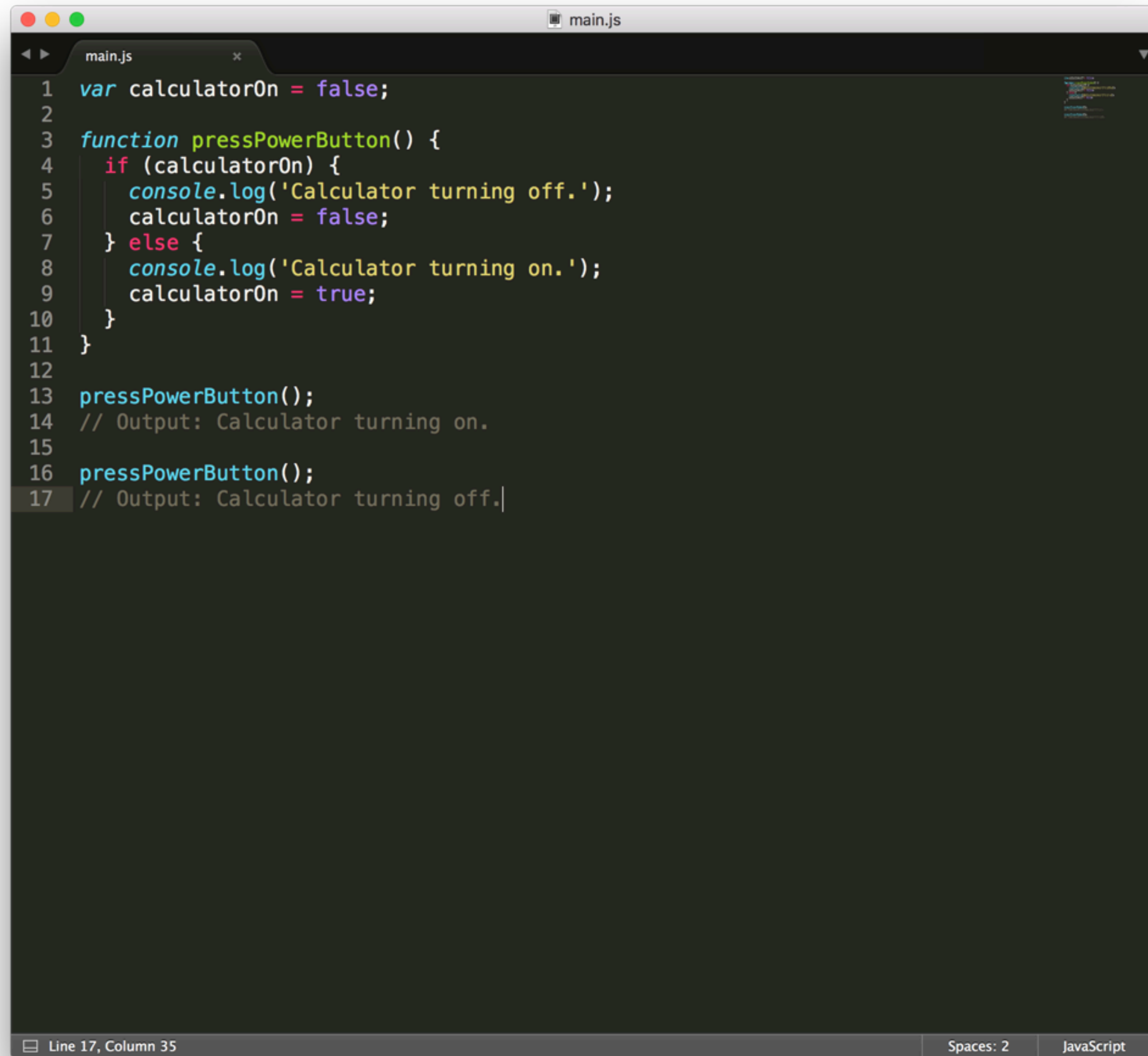
The screenshot shows a code editor window titled 'main.js'. The code defines a variable 'calculatorOn' set to false, a function 'pressPowerButton' with an 'if' statement to toggle the state and log messages, and two calls to the function. The status bar at the bottom indicates 'Line 17, Column 35', 'Spaces: 2', and 'JavaScript'.

```
6     calculatorOn = false;
7   } else {
8     console.log('Calculator turning on.');
```

The code continues with the function closing brace, followed by two calls to 'pressPowerButton()' with corresponding comments. The cursor is at the end of the second comment on line 17.

```
9     calculatorOn = true;
10  }
11 }
12
13 pressPowerButton();
14 // Output: Calculator turning on.
15
16 pressPowerButton();
17 // Output: Calculator turning off. |
```

Finally, the function has a block of code it executes between the curly braces `{ }`.



```
1 var calculatorOn = false;
2
3 function pressPowerButton() {
4   if (calculatorOn) {
5     console.log('Calculator turning off.');
```

The image shows a code editor window titled 'main.js'. The code is written in JavaScript and defines a function 'pressPowerButton()' which toggles a 'calculatorOn' boolean variable. The function uses an 'if' statement to check the current state of 'calculatorOn' and logs a message to the console before toggling the value. Lines 13 and 16 show the function being called. The editor has a dark theme and a status bar at the bottom indicating 'Line 17, Column 35', 'Spaces: 2', and 'JavaScript'.

```
6     calculatorOn = false;
7   } else {
8     console.log('Calculator turning on.');
```

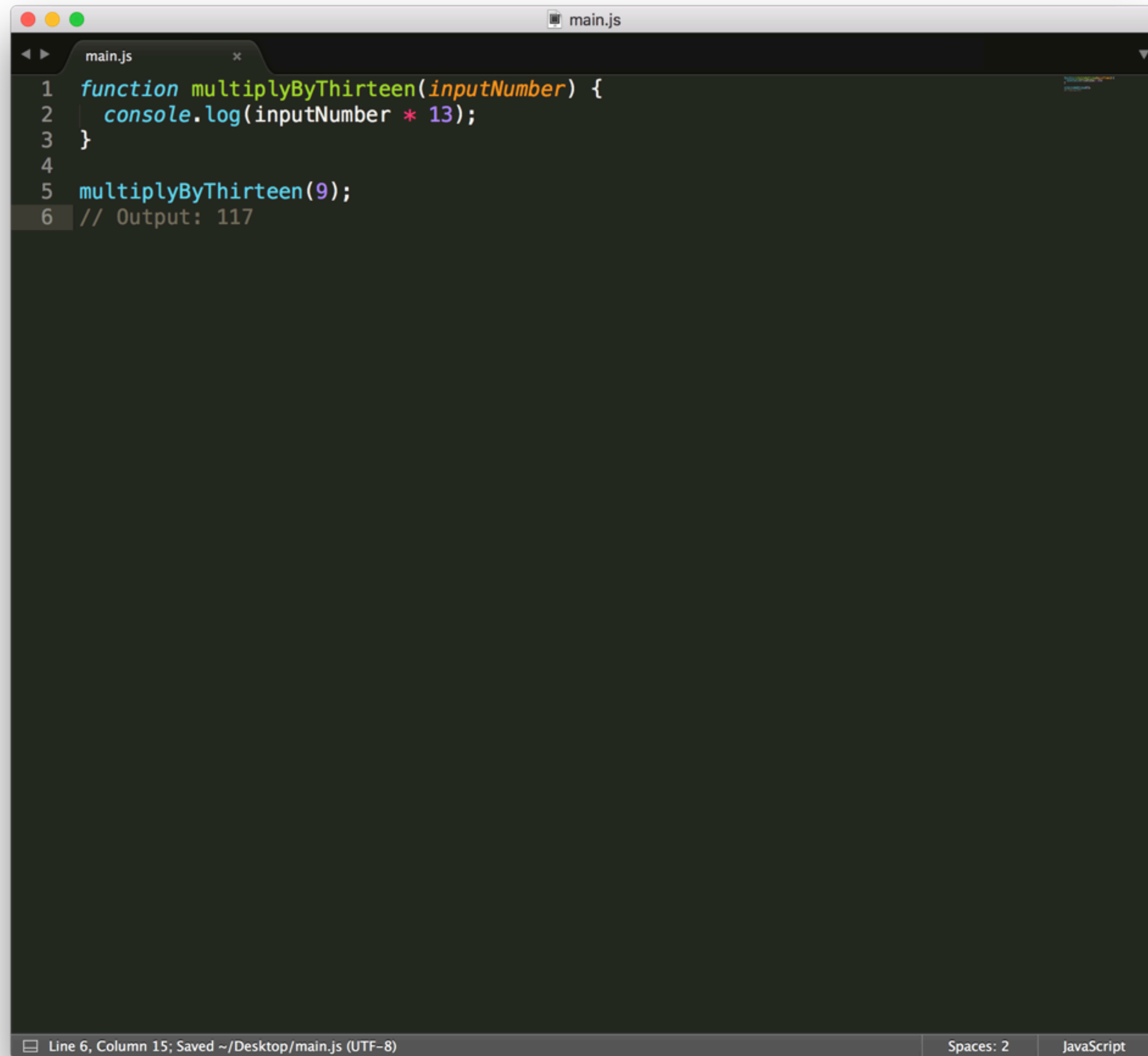
The code continues with the 'else' block, which logs a message and sets 'calculatorOn' to 'true'. The function definition ends with closing braces. Lines 13 and 16 show the function being called. Line 14 has a comment '// Output: Calculator turning on.' and line 17 has a comment '// Output: Calculator turning off.'.

```
9     calculatorOn = true;
10  }
11 }
12
13 pressPowerButton();
14 // Output: Calculator turning on.
15
16 pressPowerButton();
17 // Output: Calculator turning off.|
```

Lines 13 and 16 call the funtion.

Let's try something else. A calculator program should be able to perform a math operation on a number. We should be able to give a calculator a number, have it perform a task on it like multiplication, then print a result.

Currently, we have no way to give a function a number. To do this, we can use *parameters*.

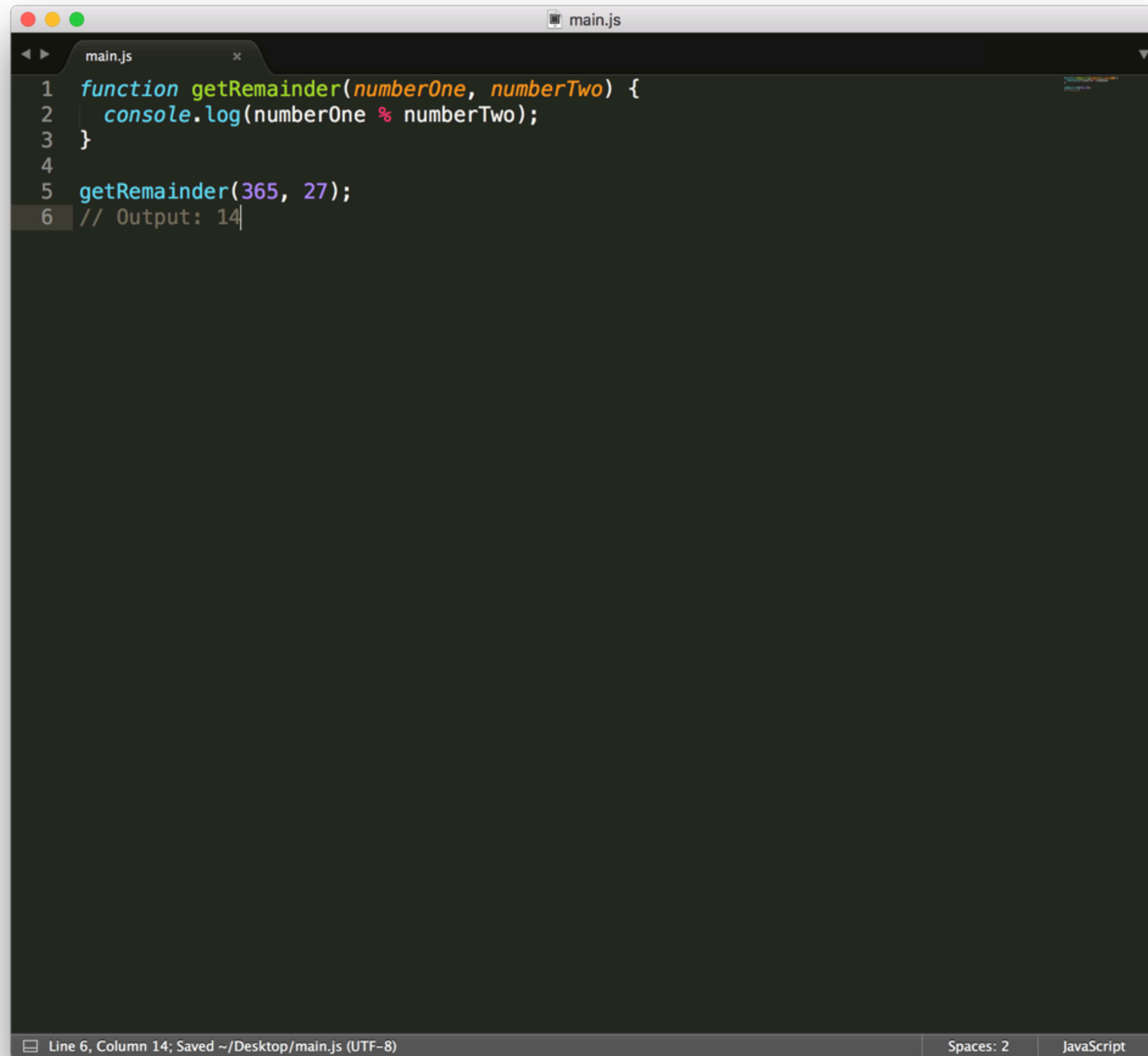


The image shows a code editor window with a dark theme. The title bar at the top indicates the file is named 'main.js'. The editor contains the following JavaScript code:

```
1 function multiplyByThirteen(inputNumber) {  
2   console.log(inputNumber * 13);  
3 }  
4  
5 multiplyByThirteen(9);  
6 // Output: 117
```

The code is syntax-highlighted: 'function' is blue, 'multiplyByThirteen' is green, 'inputNumber' is orange, 'console.log' is purple, and '13' is pink. The line numbers 1 through 6 are on the left. The status bar at the bottom shows 'Line 6, Column 15; Saved ~/Desktop/main.js (UTF-8)', 'Spaces: 2', and 'JavaScript'.

Parameters are variables that we can set when we call the function.



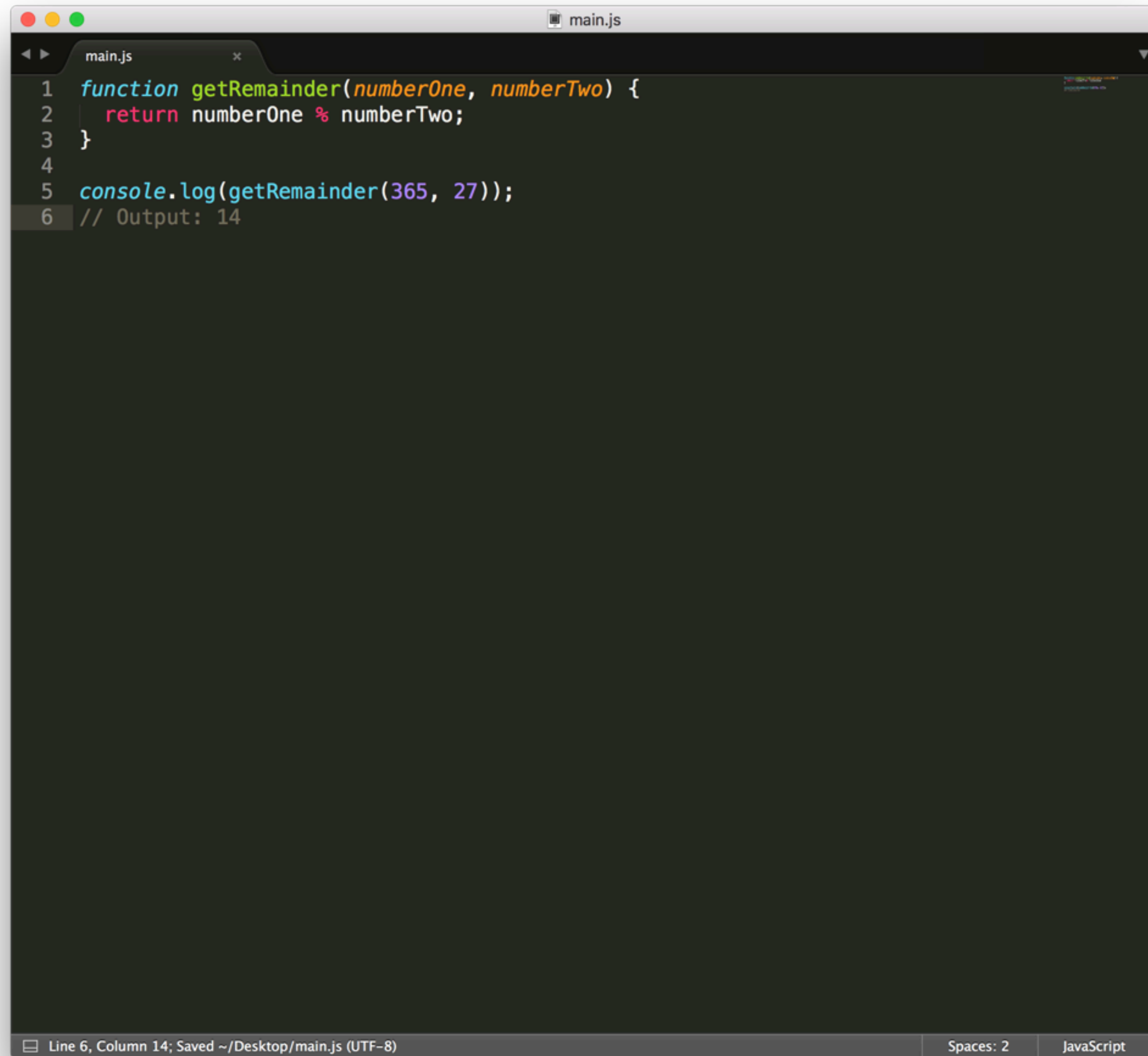
```
1 function getRemainder(numberOne, numberTwo) {  
2   console.log(numberOne % numberTwo);  
3 }  
4  
5 getRemainder(365, 27);  
6 // Output: 14
```

The screenshot shows a code editor window titled 'main.js'. The code defines a function 'getRemainder' with two parameters, 'numberOne' and 'numberTwo'. The function body contains a single line 'console.log(numberOne % numberTwo);'. Below the function definition, the function is called with arguments '365' and '27'. A comment on the final line indicates the expected output is '14'. The editor's status bar at the bottom shows 'Line 6, Column 14; Saved ~/Desktop/main.js (UTF-8)', 'Spaces: 2', and 'JavaScript'.

We can set as many parameters as we'd like by adding them when we declare the function, separated by commas

The purpose of a function is to take some input, perform some task on that input, then return a result.

To return a result, we can use the **return** keyword.



```
main.js
1 function getRemainder(numberOne, numberTwo) {
2   return numberOne % numberTwo;
3 }
4
5 console.log(getRemainder(365, 27));
6 // Output: 14
```

Line 6, Column 14; Saved ~/Desktop/main.js (UTF-8) Spaces: 2 JavaScript

Now our code is better. Why? If we wanted to use the **getRemainder** function in another place in our program, we could without printing the result to the console. Using **return** is generally a best practice when writing functions, as it makes your code more maintainable and flexible.

Scope is a big idea in programming, so let's start at a high level.

Scope refers to where in a program a variable can be accessed. The idea is that some variables are unable to be accessed everywhere within a program.

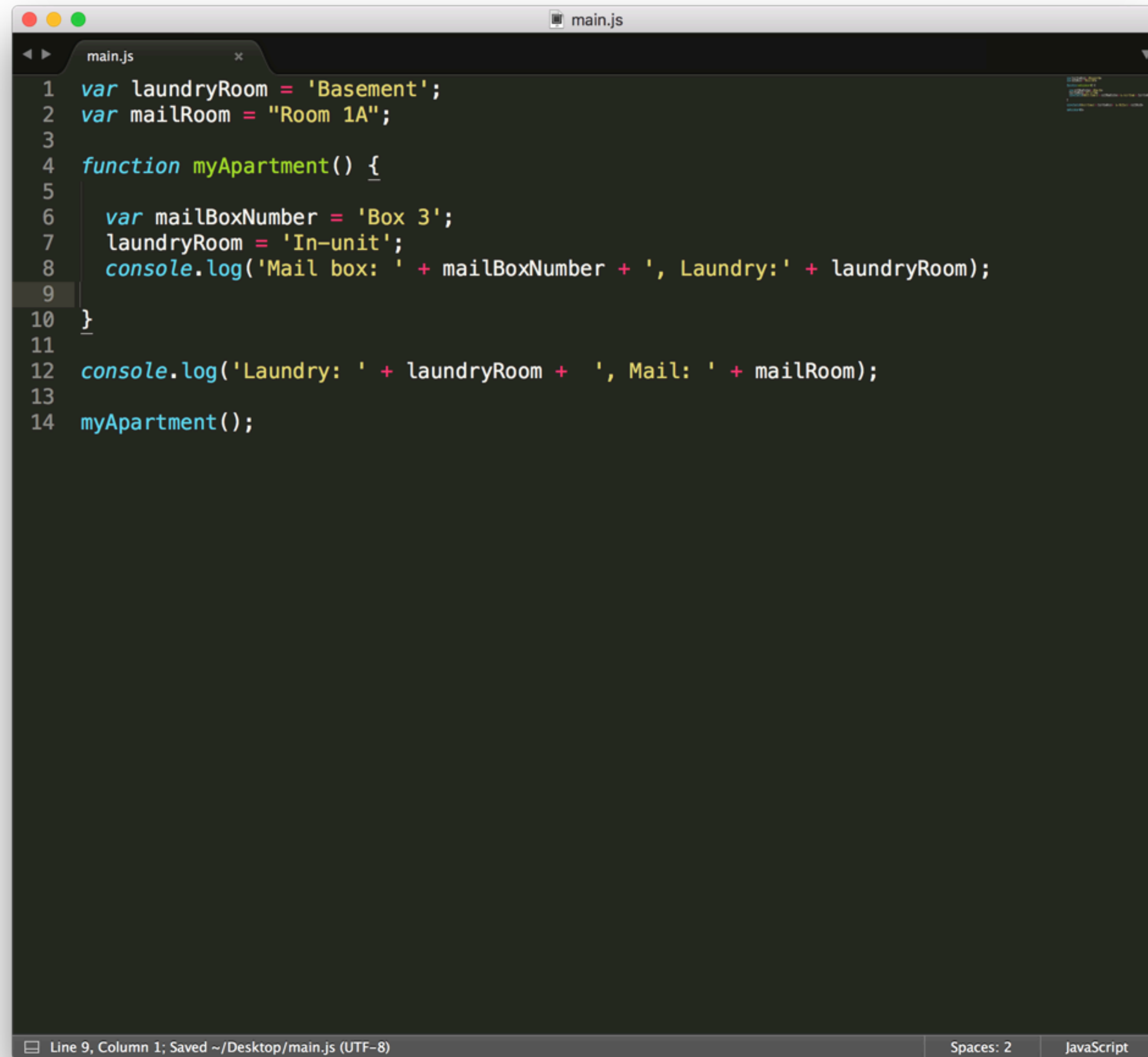
Think of it like an apartment building. Everyone who lives in the apartment building is under the *global scope* of the building and its manager. So, if there are rats in the shared laundry room, everyone has access to the laundry machines, and the rats.

If you write a variable outside of a function in JavaScript, it's in the *global scope* and can be used by any other part of the program, just like the laundry room can be used by everyone in an apartment.

In our theoretical apartment building, you have your own apartment. It has stuff in it that is yours. Other people in the building can't access it.

This is like *functional scope*. You have access to your stuff inside your apartment, and in the building – but not anyone else's apartment.

When we write variables inside a function, only that function has access to its own variables. Therefore, they are in the *functional scope*.



```
1 var laundryRoom = 'Basement';
2 var mailRoom = "Room 1A";
3
4 function myApartment() {
5
6     var mailBoxNumber = 'Box 3';
7     laundryRoom = 'In-unit';
8     console.log('Mail box: ' + mailBoxNumber + ', Laundry: ' + laundryRoom);
9
10 }
11
12 console.log('Laundry: ' + laundryRoom + ', Mail: ' + mailRoom);
13
14 myApartment();
```

Line 9, Column 1; Saved ~/Desktop/main.js (UTF-8) Spaces: 2 JavaScript

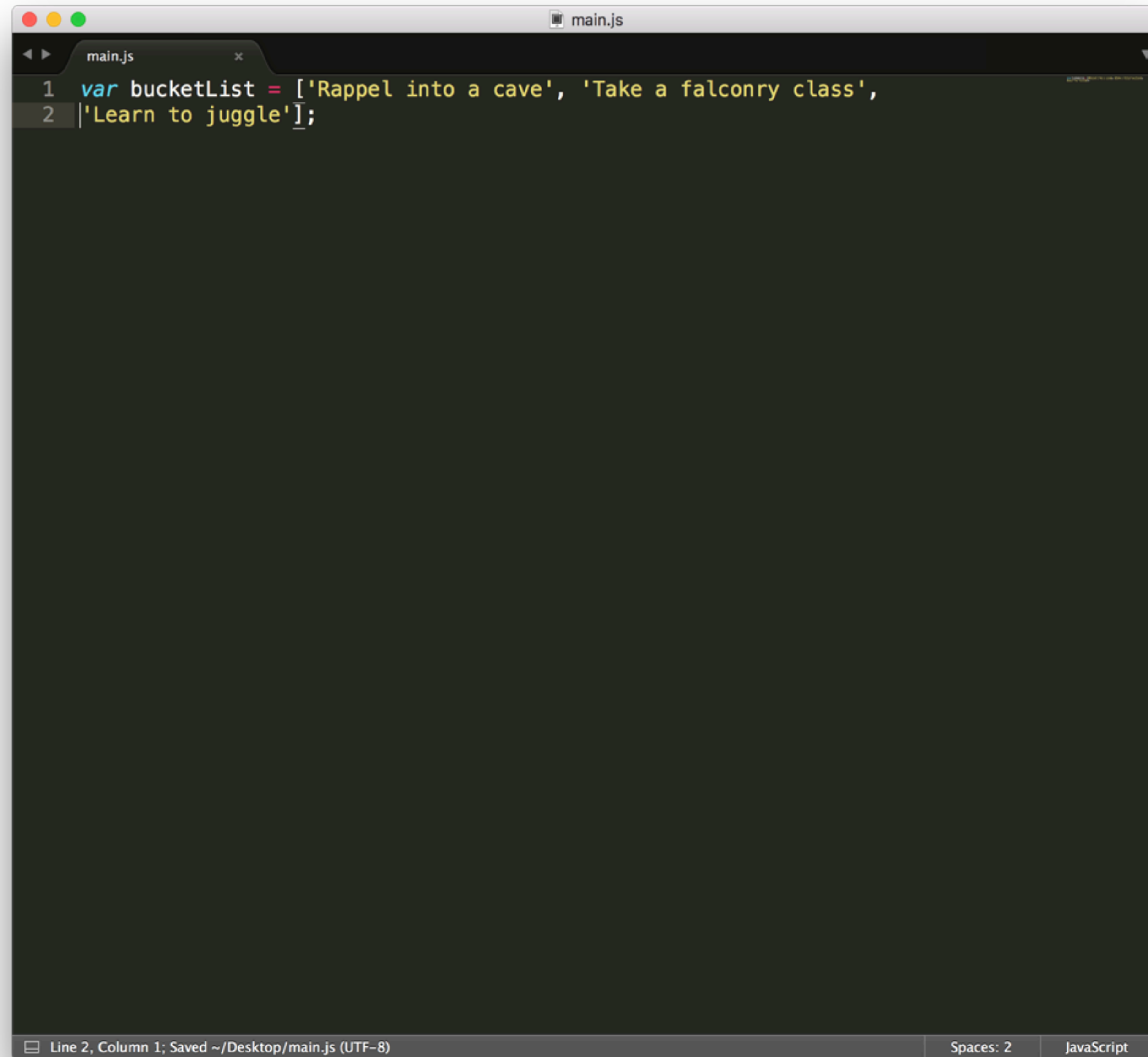
**One thing that we haven't learned yet is how to
organize and store data.**

**One way we organize data in real life is to make lists.
Let's make one here:**

Bucket List:

- 0. Rappel into a cave**
- 1. Take a falconry class**
- 2. Learn to juggle**

Let's now write this list in JavaScript, as an *array*



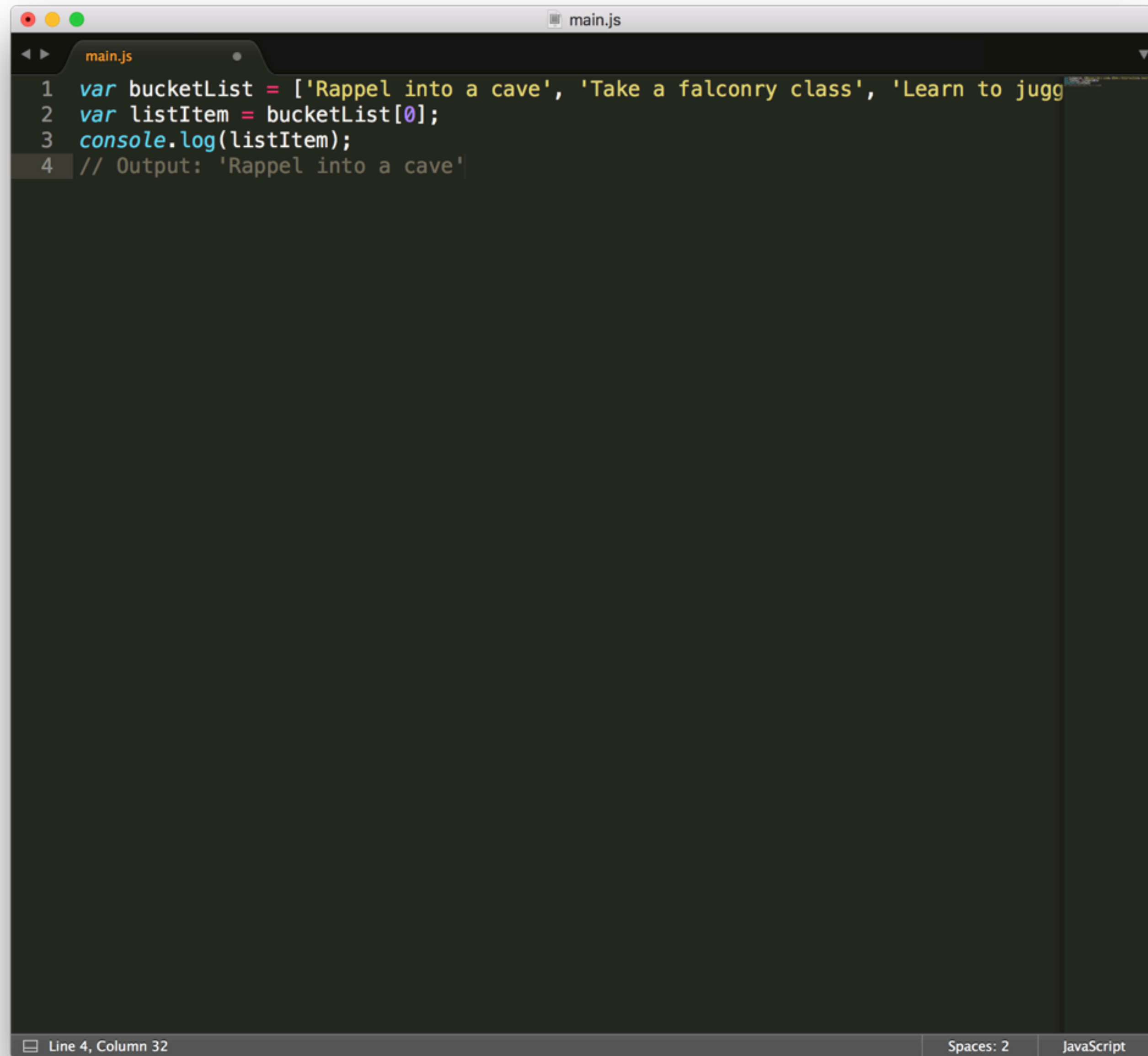
```
1 var bucketList = ['Rappel into a cave', 'Take a falconry class',  
2 'Learn to juggle'];
```

Line 2, Column 1; Saved ~/Desktop/main.js (UTF-8) Spaces: 2 JavaScript

Now, what if we want to select one item from an array?

Each item in an array has a numbered position. We can access an item using its number, just like we would in an ordinary list.

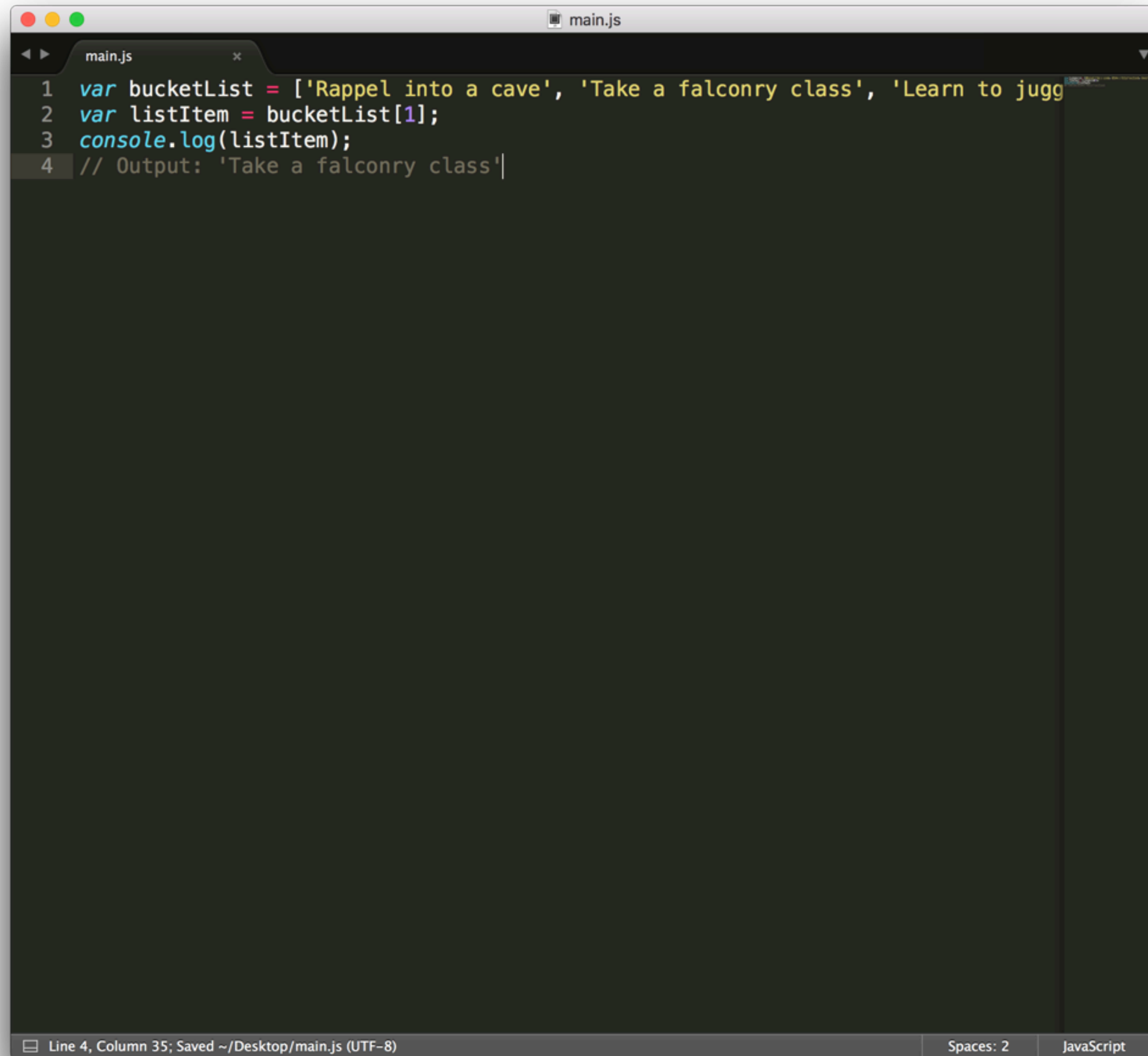
JavaScript counts starting from **0**, not **1**, so the first item in an array will be at position **0**. This is because JavaScript is *zero-indexed*.

A screenshot of a code editor window with a dark theme. The window title is 'main.js'. The code is as follows:

```
1 var bucketList = ['Rappel into a cave', 'Take a falconry class', 'Learn to jugg
2 var listItem = bucketList[0];
3 console.log(listItem);
4 // Output: 'Rappel into a cave'
```

The code is syntax-highlighted. The status bar at the bottom shows 'Line 4, Column 32', 'Spaces: 2', and 'JavaScript'.

We can select the first item in an array like this.

A screenshot of a code editor window with a dark theme. The window has a title bar with three colored buttons (red, yellow, green) and a tab labeled 'main.js'. The code is as follows:

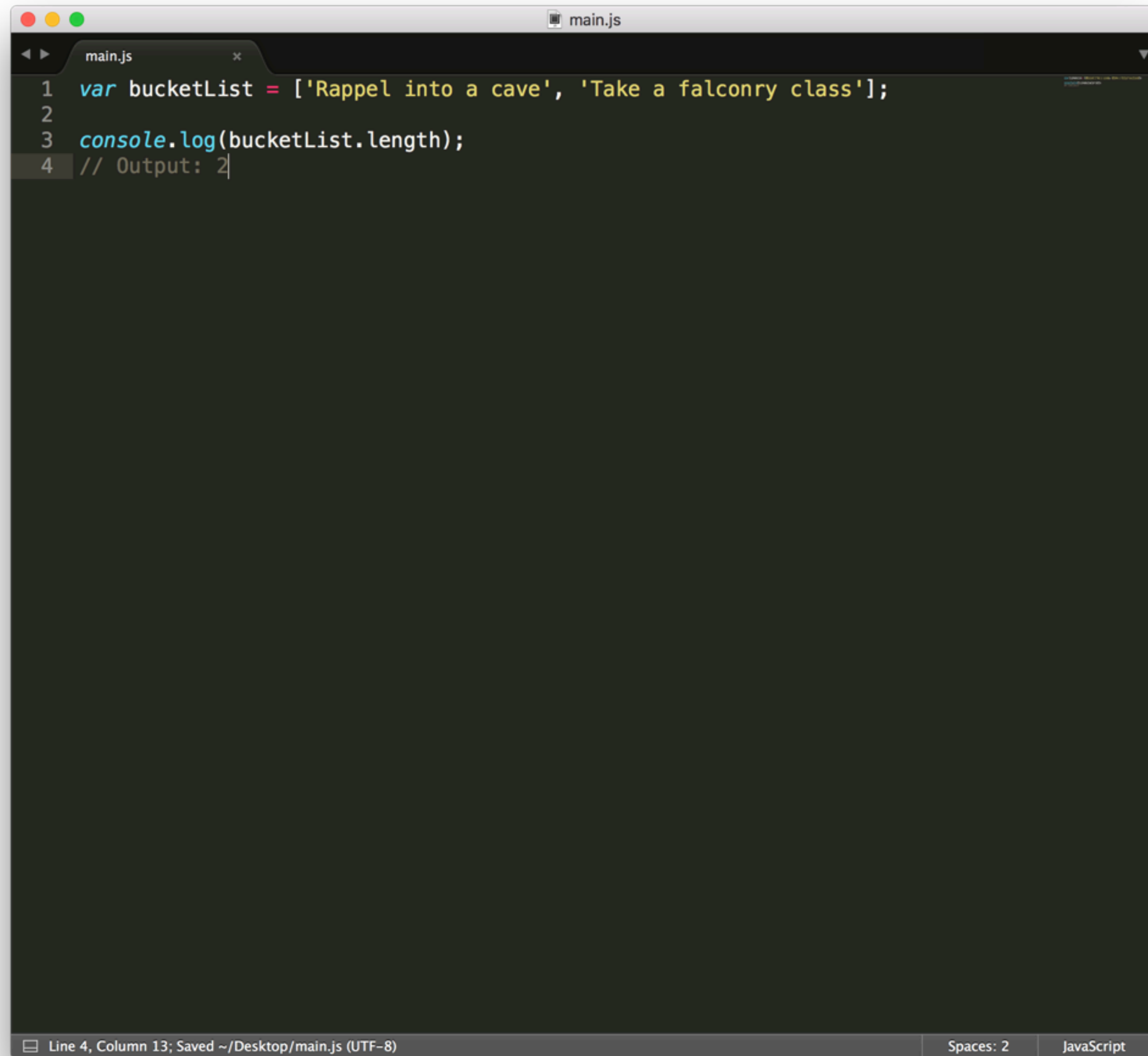
```
1 var bucketList = ['Rappel into a cave', 'Take a falconry class', 'Learn to jugg
2 var listItem = bucketList[1];
3 console.log(listItem);
4 // Output: 'Take a falconry class'|
```

The cursor is at the end of line 4. The status bar at the bottom shows 'Line 4, Column 35; Saved ~/Desktop/main.js (UTF-8)', 'Spaces: 2', and 'JavaScript'.

If we wanted the second item, we'd write this.

It is often convenient to know how many items are inside of an array.

We can find this out by using one of an array's built in *properties*, called **.length**. We can attach this to any variable holding an array and it will return the number of items inside.

A screenshot of a code editor window with a dark theme. The window has a title bar with three colored buttons (red, yellow, green) and a tab labeled 'main.js'. The code is as follows:

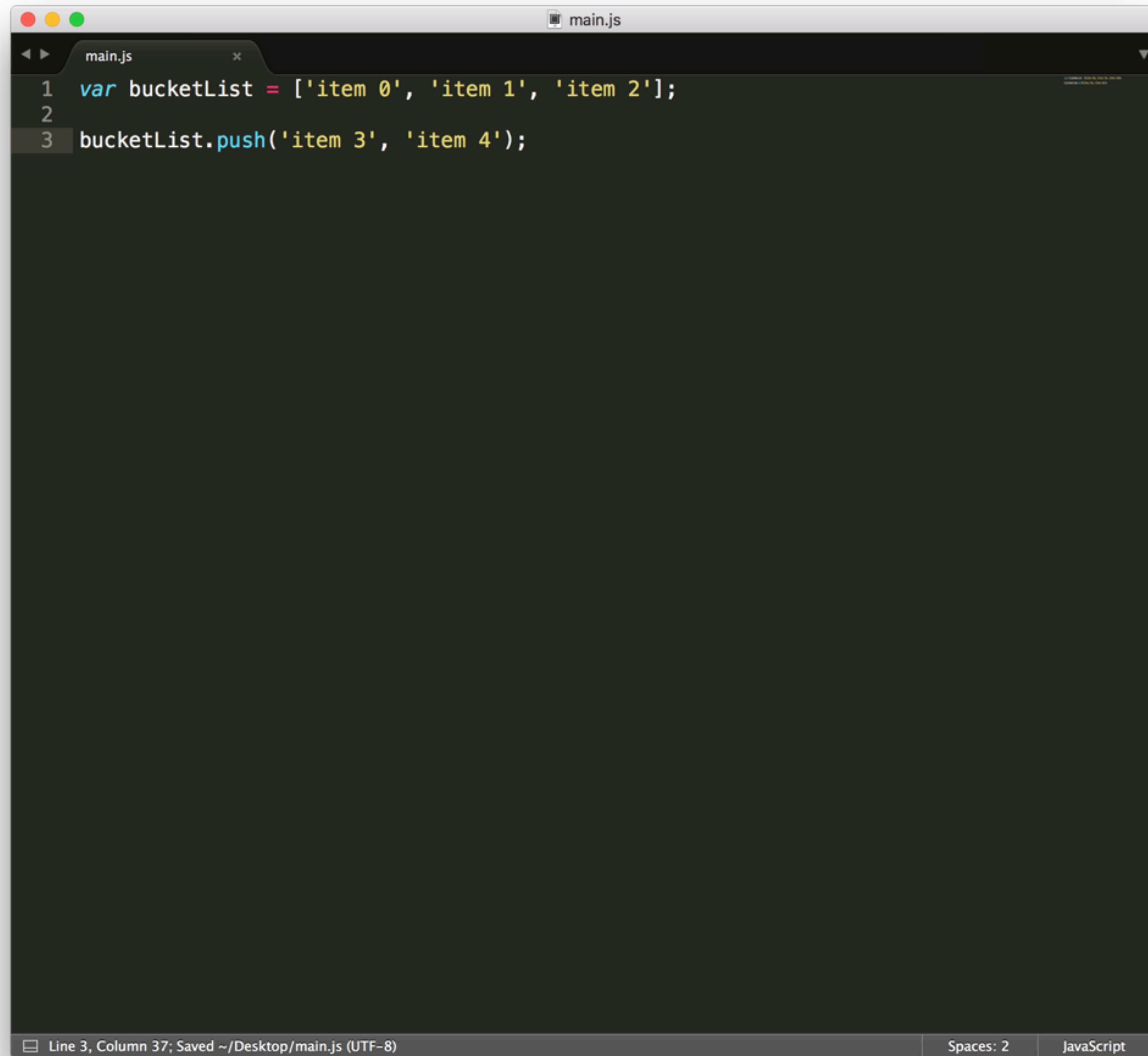
```
1 var bucketList = ['Rappel into a cave', 'Take a falconry class'];
2
3 console.log(bucketList.length);
4 // Output: 2
```

The cursor is at the end of line 4. The status bar at the bottom shows 'Line 4, Column 13; Saved ~/Desktop/main.js (UTF-8)', 'Spaces: 2', and 'JavaScript'.

If we wanted the second item, we'd write this.

JavaScript has a surprise for us: it has built in functions for arrays that help us do common tasks. Let's learn two of them.

First, **push()** allows us to add items to the end of an array.

A screenshot of a code editor window titled 'main.js'. The editor has a dark background with light-colored text. The code is as follows:

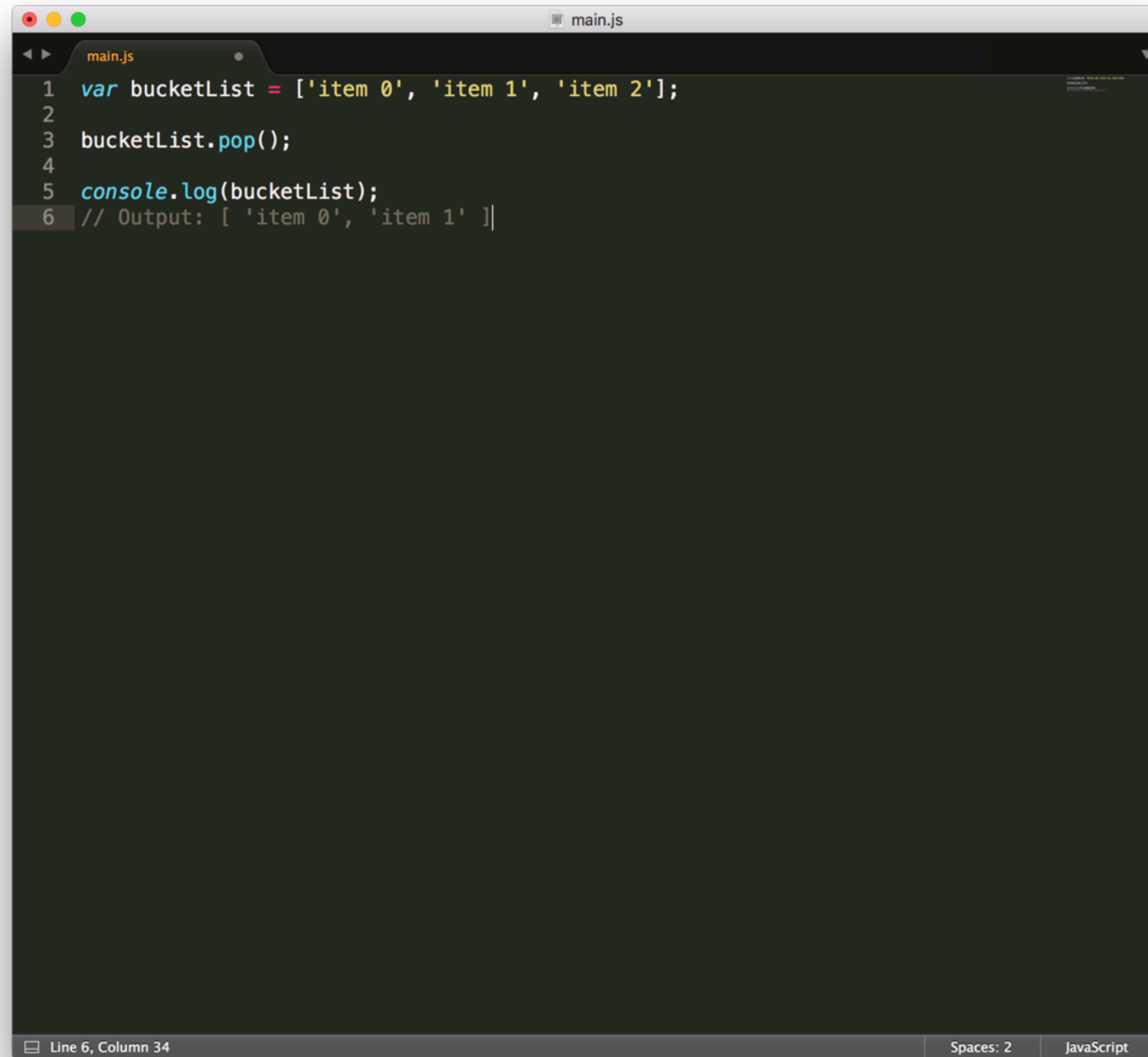
```
1 var bucketList = ['item 0', 'item 1', 'item 2'];  
2  
3 bucketList.push('item 3', 'item 4');
```

The third line is highlighted. The status bar at the bottom shows 'Line 3, Column 37; Saved ~/Desktop/main.js (UTF-8)', 'Spaces: 2', and 'JavaScript'.

The method **push()** would make the **bucketList** array look like:

['item 0' 'item 1' 'item 2' 'item 3' 'item 4'];

Now that we can `push()` items into an array, let's pop one off, using `pop()`



```
1 var bucketList = ['item 0', 'item 1', 'item 2'];
2
3 bucketList.pop();
4
5 console.log(bucketList);
6 // Output: [ 'item 0', 'item 1' ]|
```

Line 6, Column 34 Spaces: 2 JavaScript

One of a computer's greatest abilities is to repeat a task over and over so we don't have to. Loops let us tell the computer to loop over a block of code so that we don't have to write out the same process over and over.

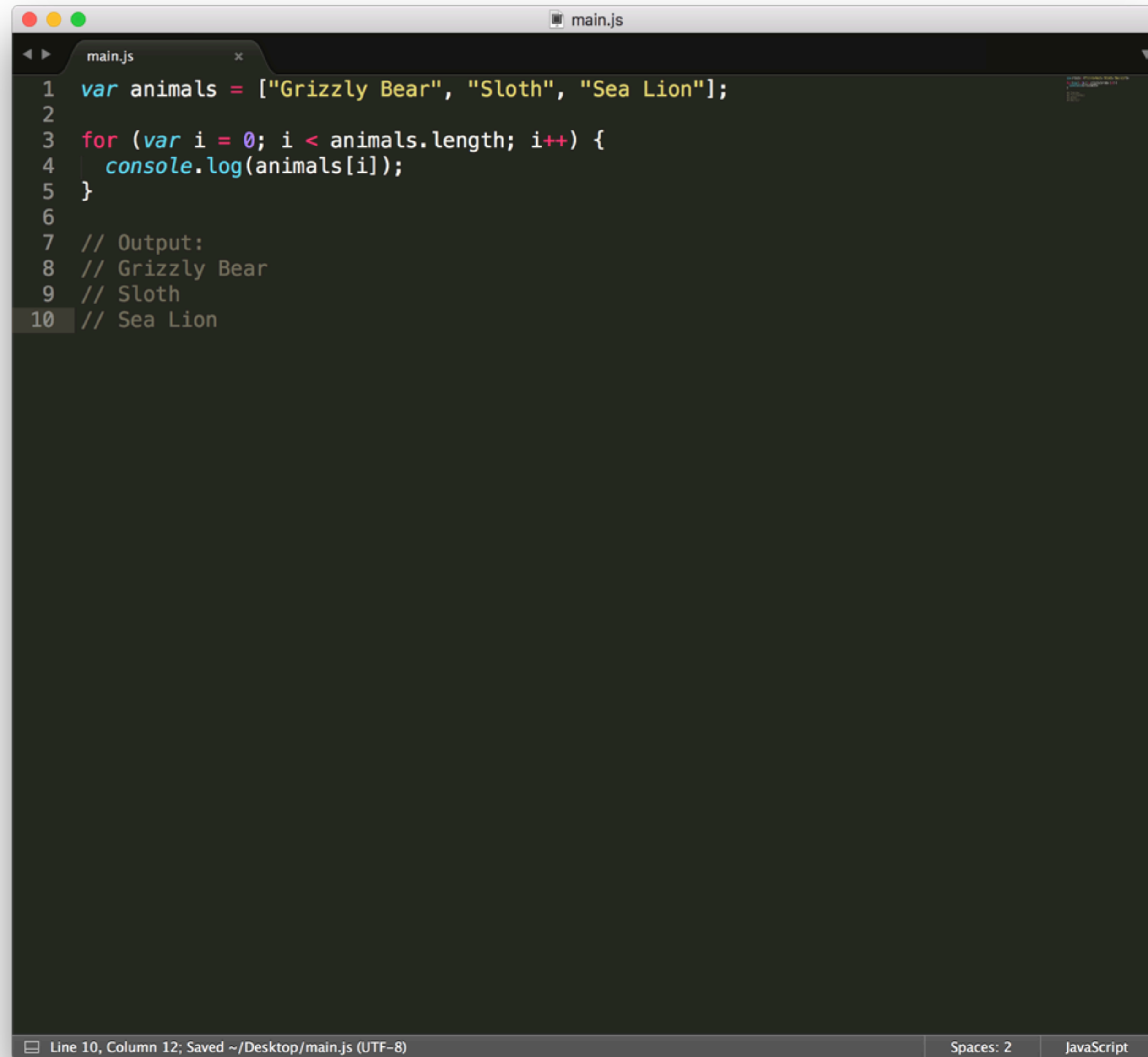
Loops are especially helpful when we have an array where we'd like to do something to each item in the array, like logging each item to the console.

There are two kinds of loops we will learn:

for loops, which let us loop a block of code a known amount of times.

while loops, which let us loop a block of code an unknown amount of times.

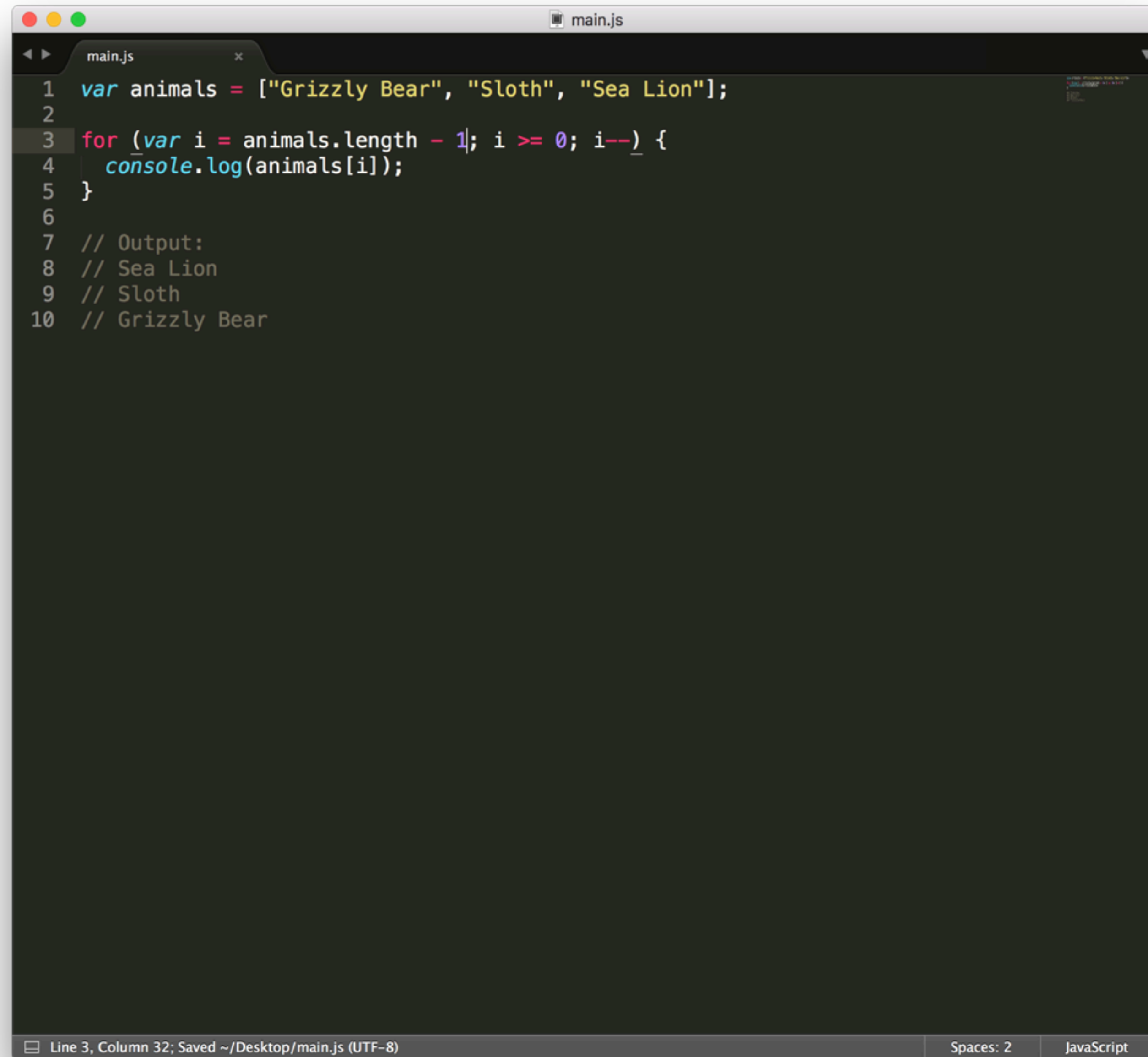
let's make the computer loop through our array for us. We can do this with **for** loops.



```
1 var animals = ["Grizzly Bear", "Sloth", "Sea Lion"];
2
3 for (var i = 0; i < animals.length; i++) {
4   console.log(animals[i]);
5 }
6
7 // Output:
8 // Grizzly Bear
9 // Sloth
10 // Sea Lion
```

Line 10, Column 12; Saved ~/Desktop/main.js (UTF-8) Spaces: 2 JavaScript

We can make our loop run backwards by modifying the start, stop, and iterator conditions.

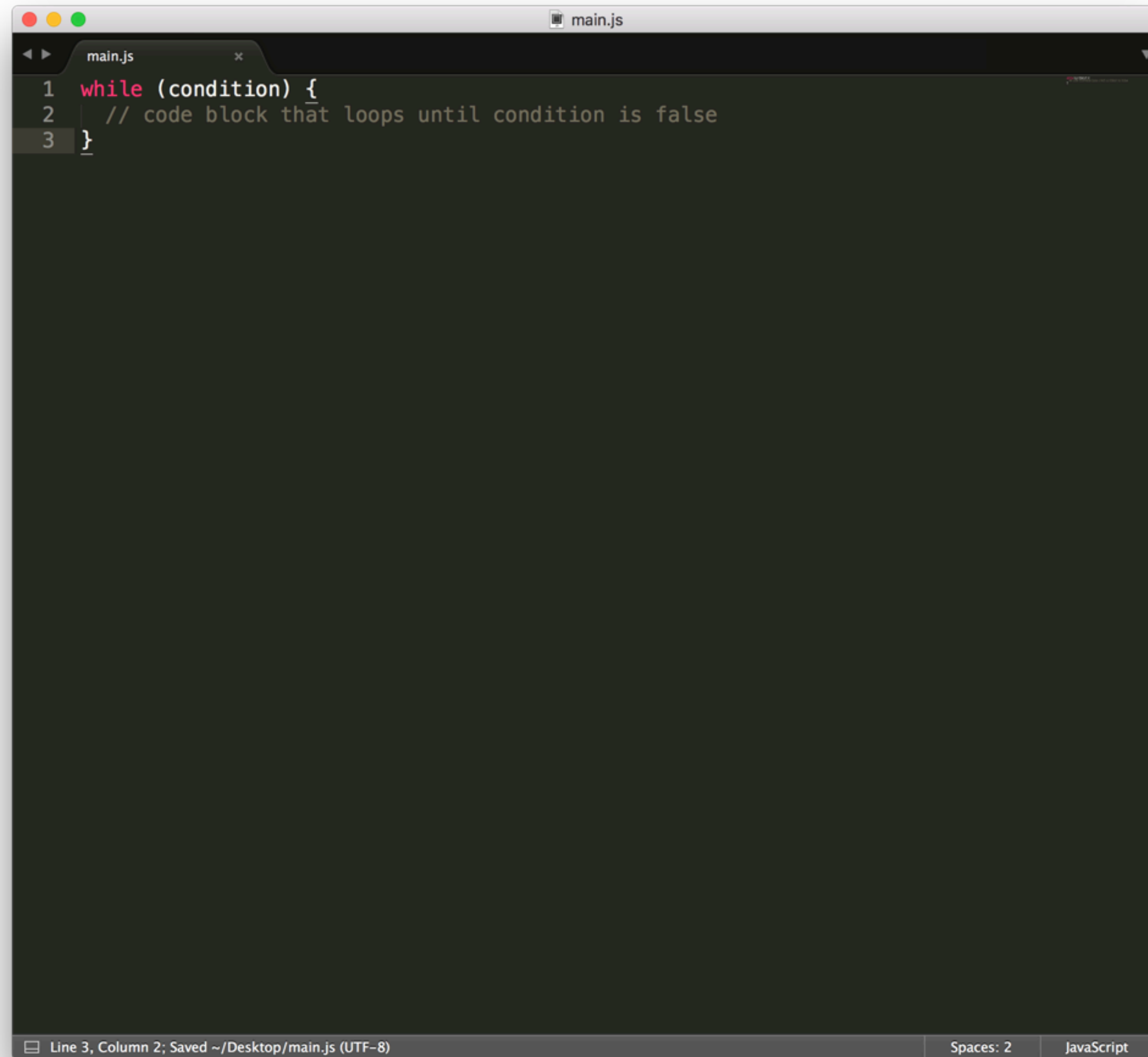


```
1 var animals = ["Grizzly Bear", "Sloth", "Sea Lion"];
2
3 for (var i = animals.length - 1; i >= 0; i--) {
4     console.log(animals[i]);
5 }
6
7 // Output:
8 // Sea Lion
9 // Sloth
10 // Grizzly Bear
```

Line 3, Column 32; Saved ~/Desktop/main.js (UTF-8) Spaces: 2 JavaScript

for loops are great, but they have a limitation: you have to know how many times you want the loop to run. What if you want a loop to run an unknown or variable number of times instead?

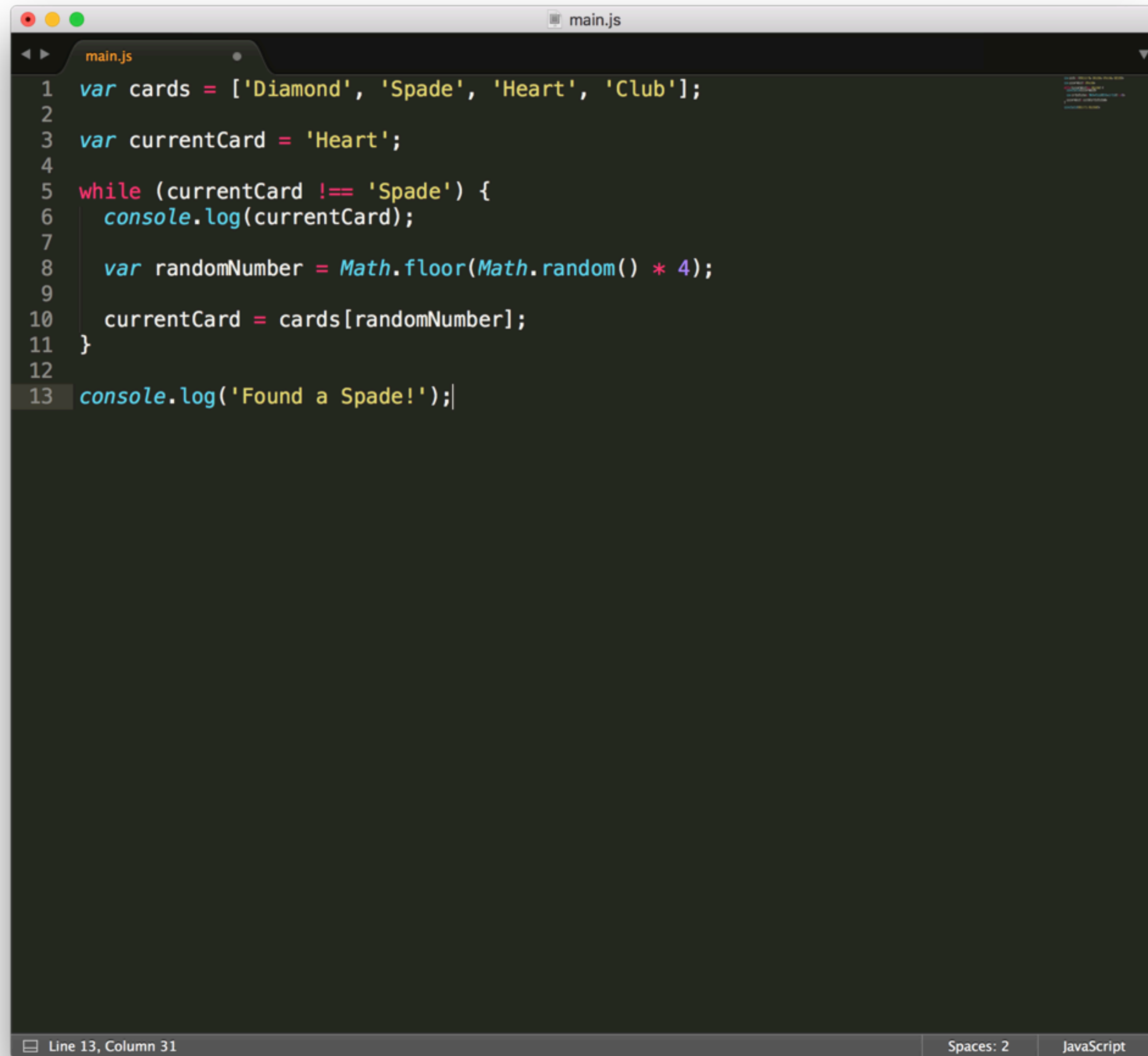
That's the purpose of the **while** loop. It looks like similar to a **for** loop.



```
1 while (condition) {  
2   // code block that loops until condition is false  
3 }
```

Line 3, Column 2; Saved ~/Desktop/main.js (UTF-8) Spaces: 2 JavaScript

Here's another example...



```
main.js
1 var cards = ['Diamond', 'Spade', 'Heart', 'Club'];
2
3 var currentCard = 'Heart';
4
5 while (currentCard !== 'Spade') {
6   console.log(currentCard);
7
8   var randomNumber = Math.floor(Math.random() * 4);
9
10  currentCard = cards[randomNumber];
11 }
12
13 console.log('Found a Spade!');
```

Line 13, Column 31 Spaces: 2 JavaScript

Here's a program that flips cards until we get a 'Spade.'