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:

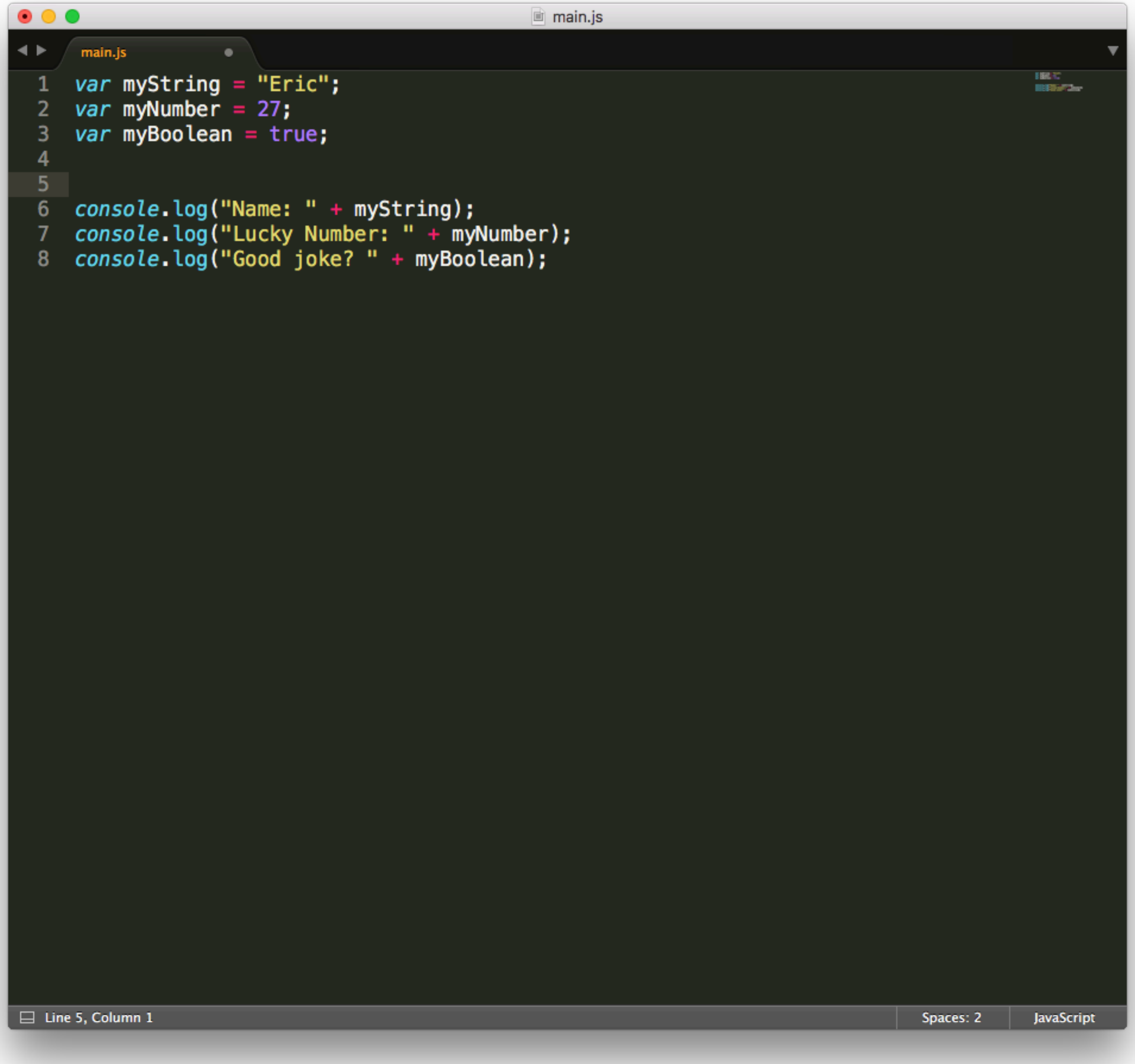
```
<script src=“myscripts.js"></script>
```

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.



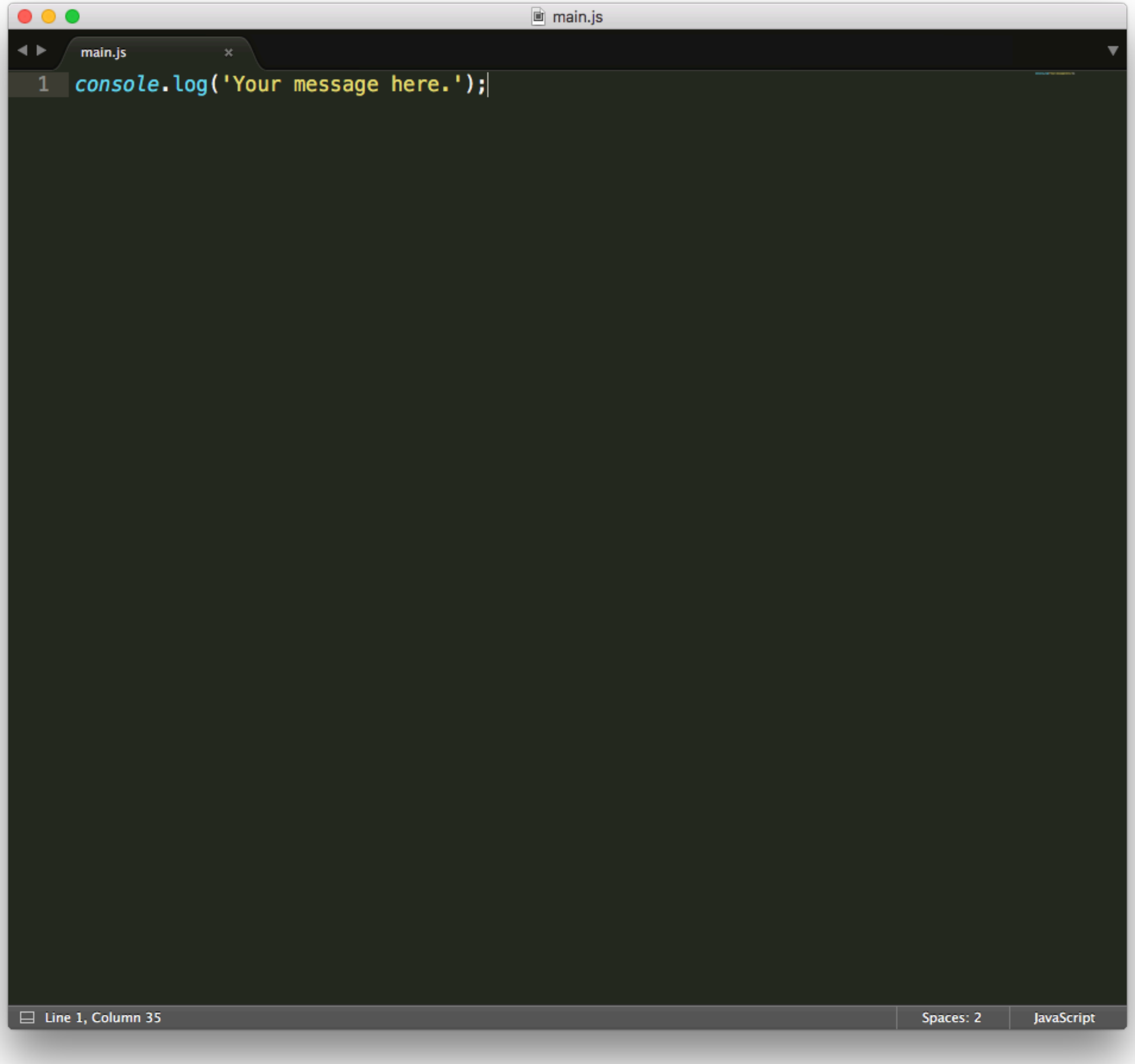
```javascript
var myString = "Eric";
var myNumber = 27;
var myBoolean = true;


console.log("Name: " + myString);
console.log("Lucky Number: " + myNumber);
console.log("Good joke? " + myBoolean);
```

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.



```
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.

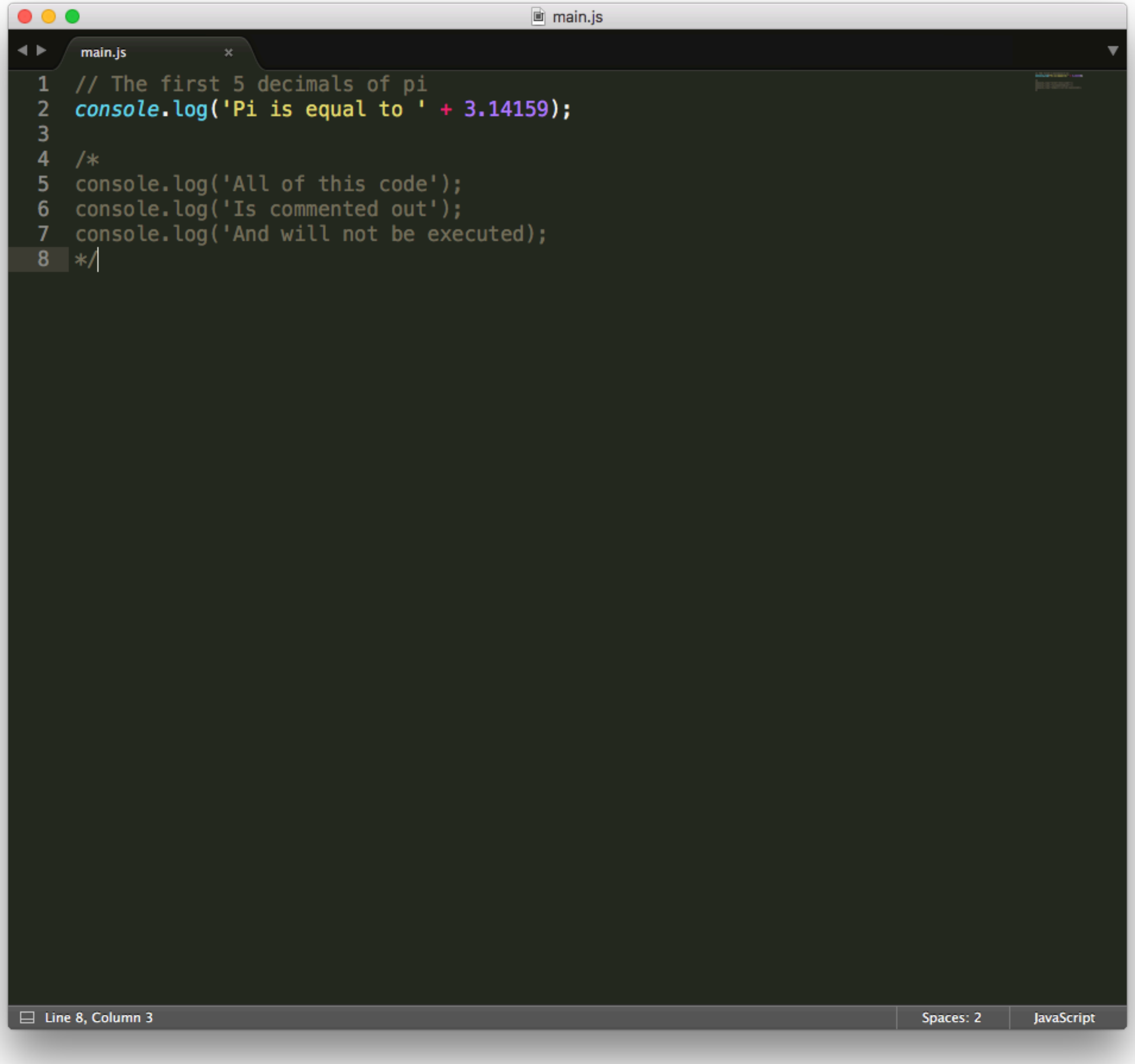
1.We then multiplied that number by 50, so now we have a number between 0 and 50.

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.



```
// The first 5 decimals of pi
console.log('Pi is equal to ' + 3.14159);

/*
console.log('All of this code');
console.log('Is commented out');
console.log('And will not be executed);
*/
```

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

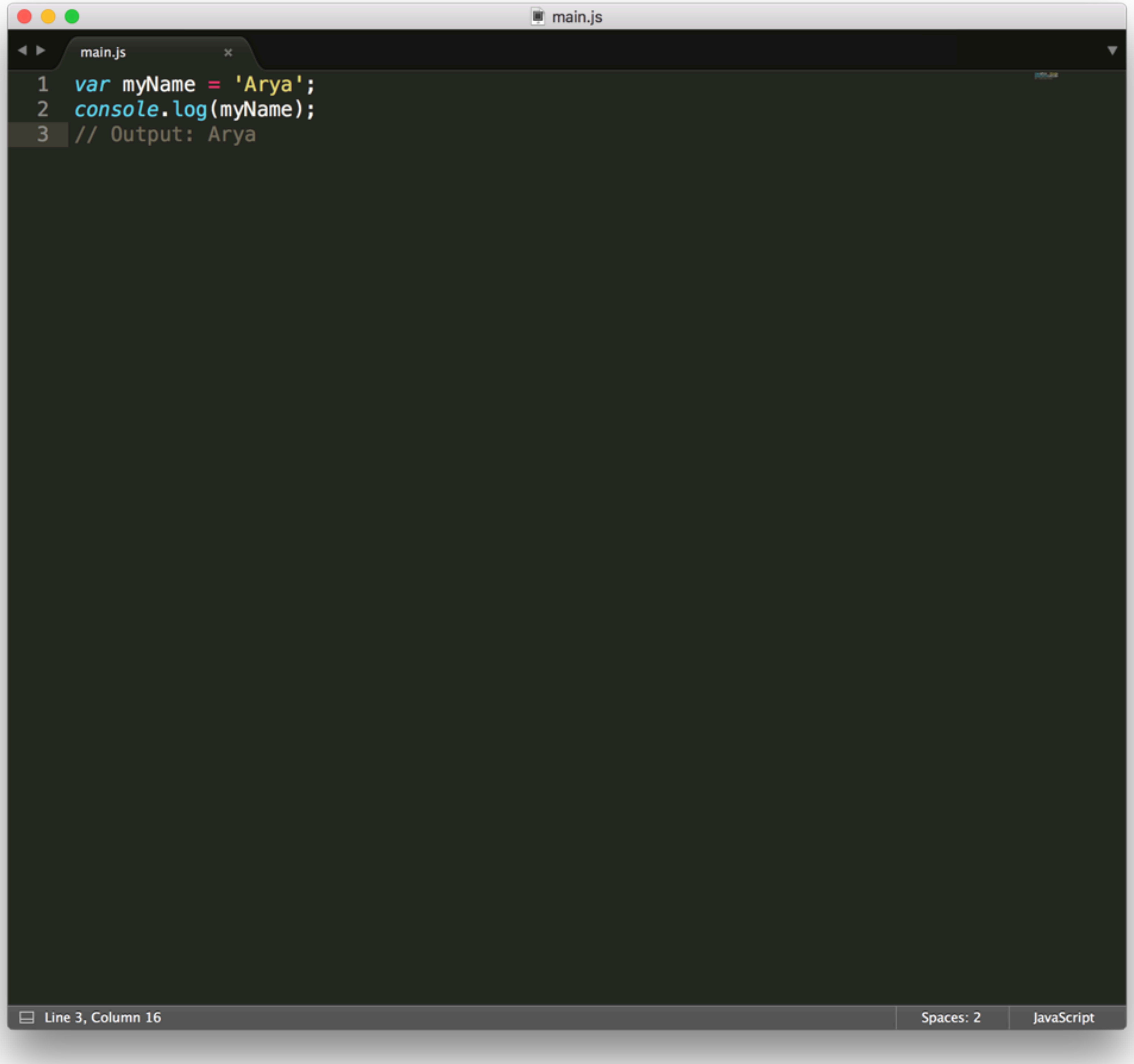
A *multi-line comment* will comment out multiple lines, and is denoted with `/*` to begin the comment, and `*/` to end the comment.

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.

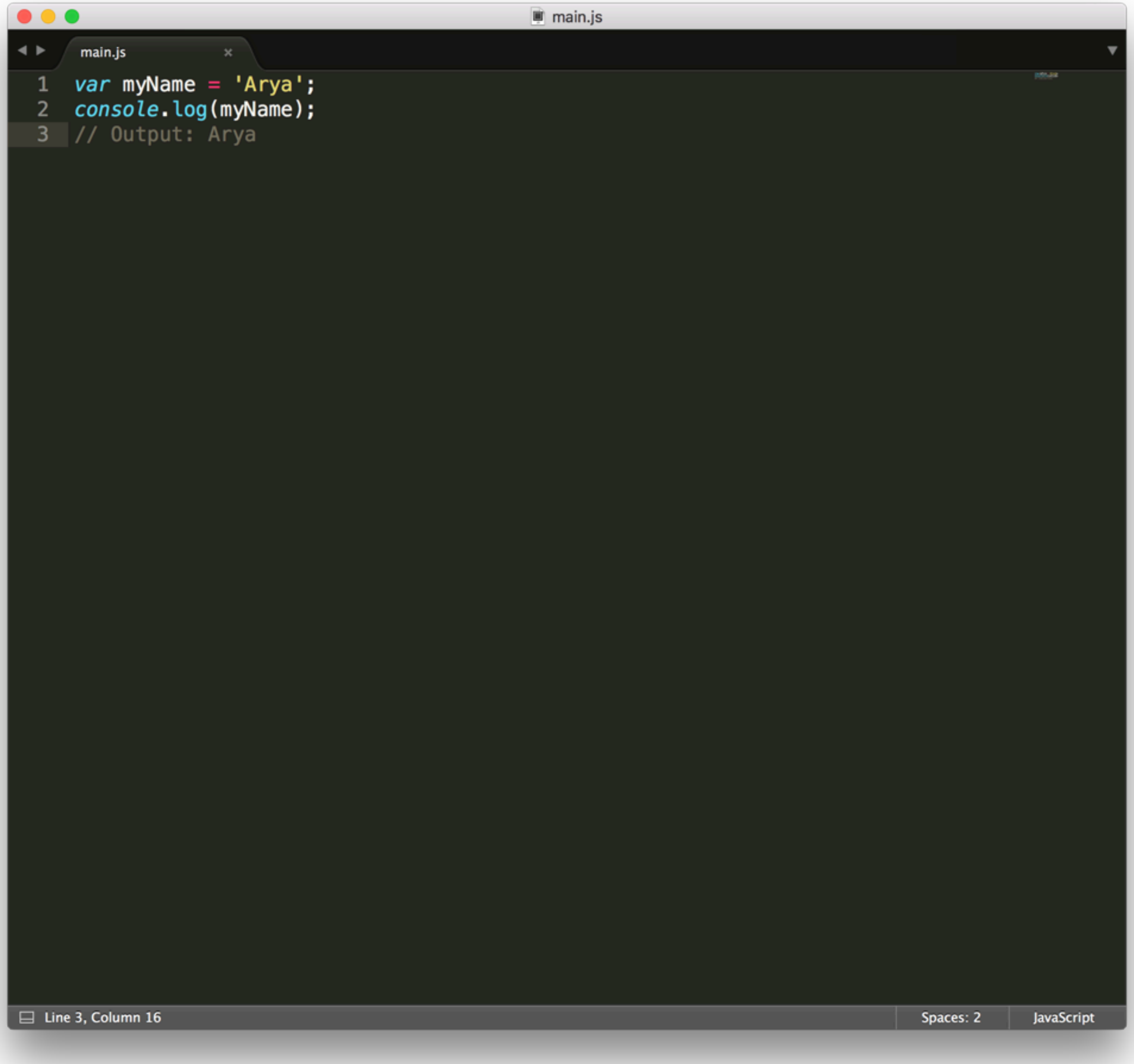
A first step toward this goal is using variables.

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



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

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

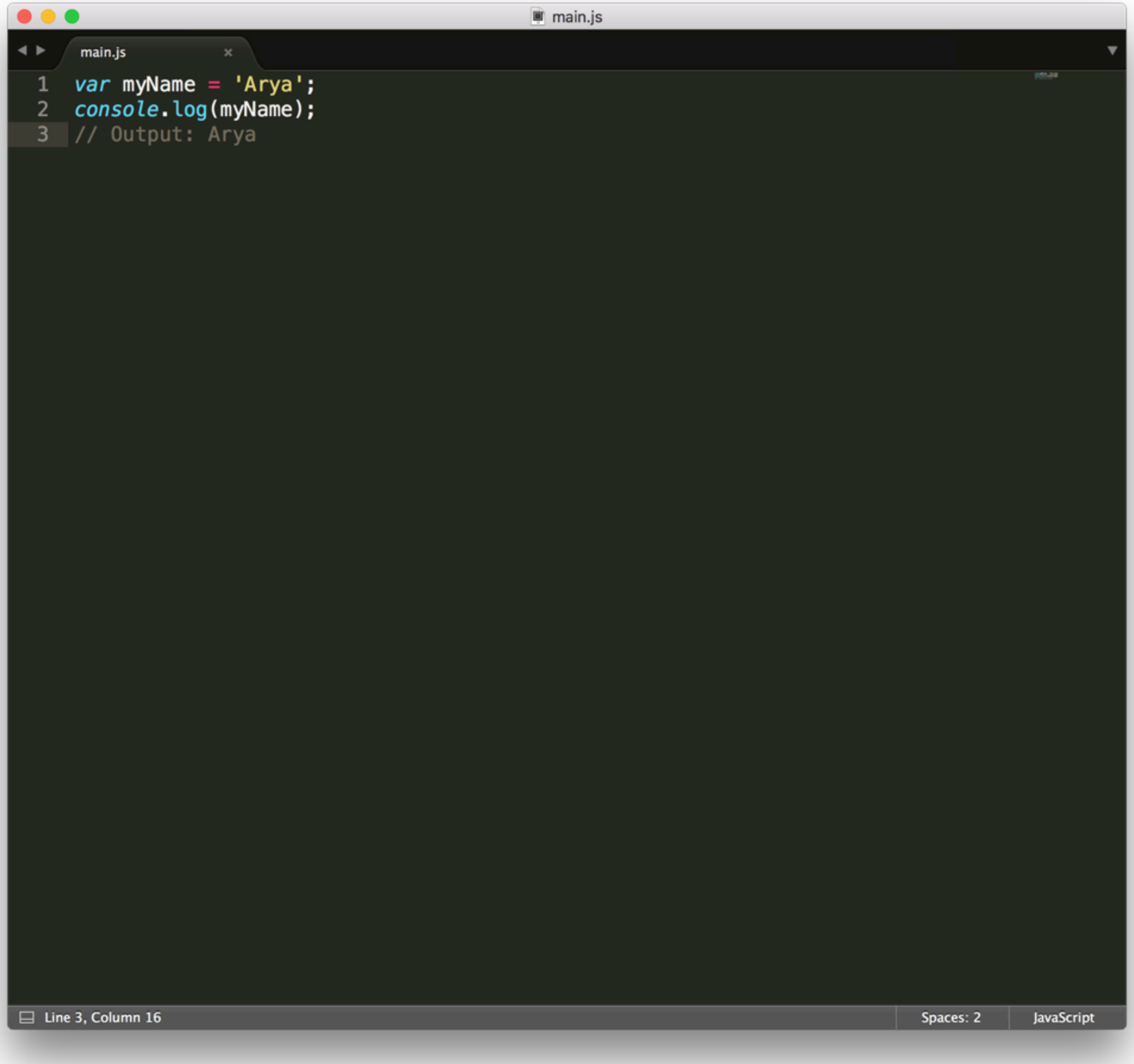


```
var myName = 'Arya';
console.log(myName);
// Output: Arya
```

Line 3, Column 16 Spaces: 2 JavaScript

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

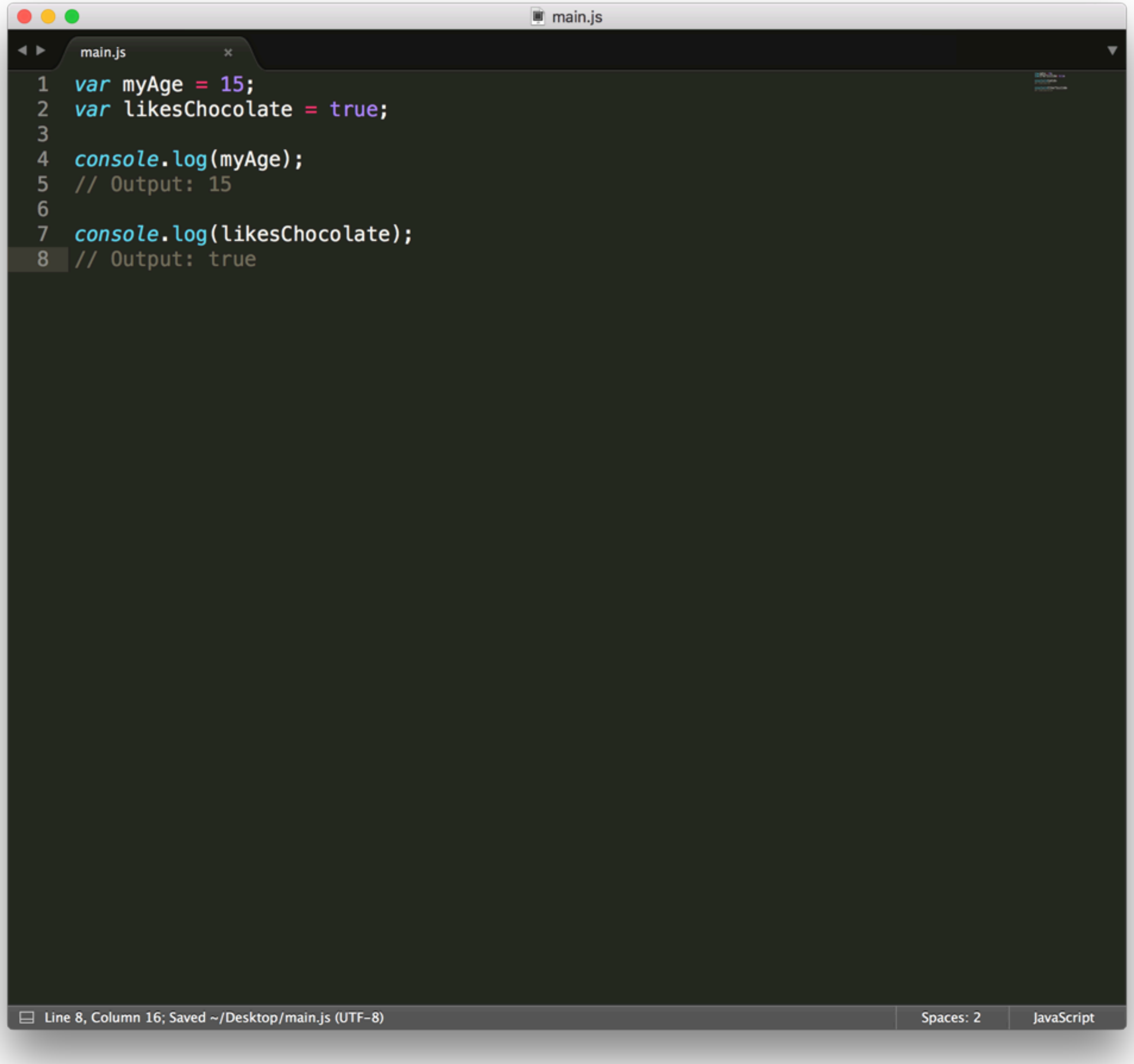
myName is chosen by a developer. Notice that the word has no spaces, and each new word is capitalized. This is a common convention in JavaScript, and is called *camelCase*.



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

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

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



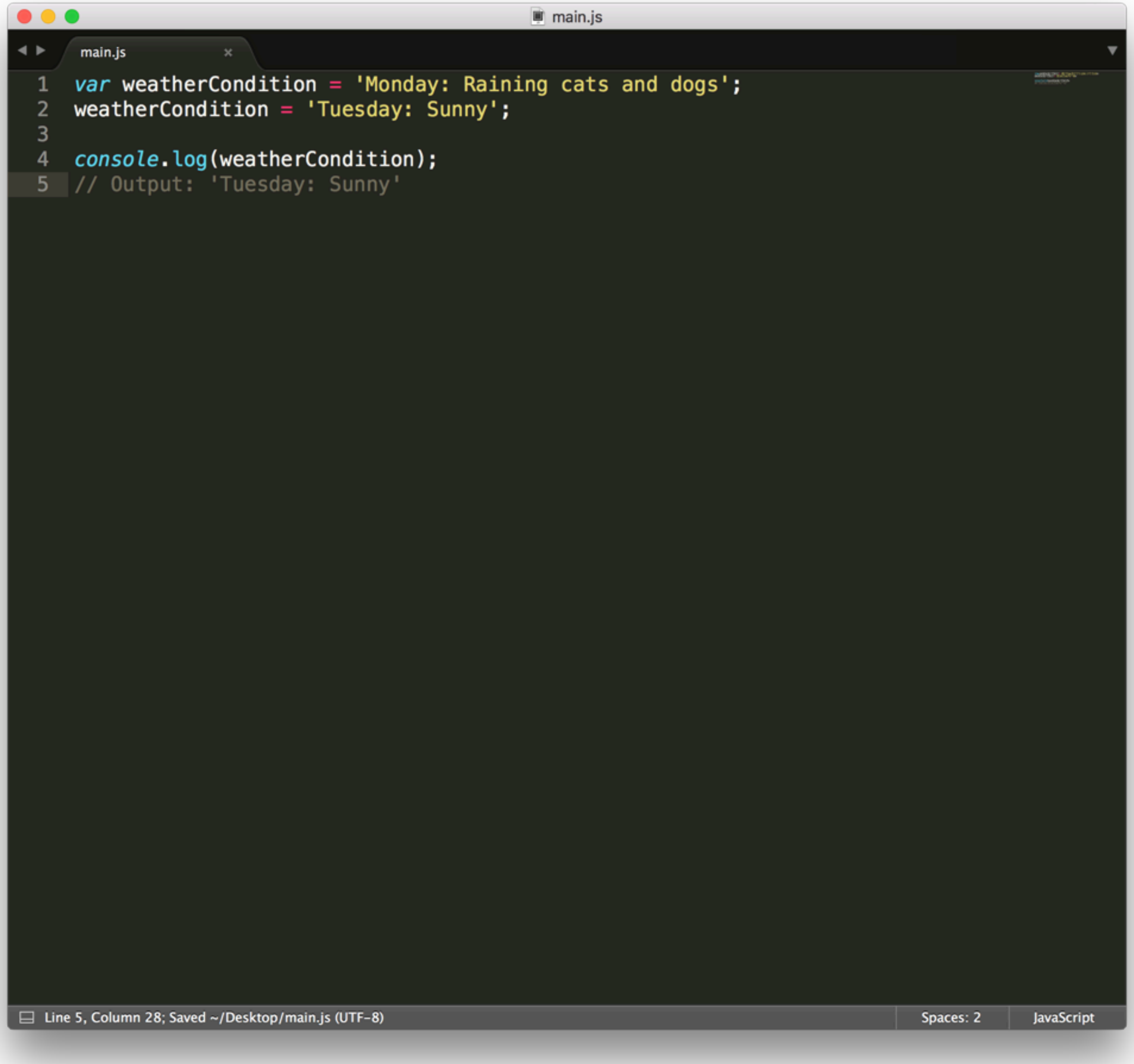
```javascript
var myAge = 15;
var likesChocolate = true;

console.log(myAge);
// Output: 15

console.log(likesChocolate);
// Output: true
```

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.

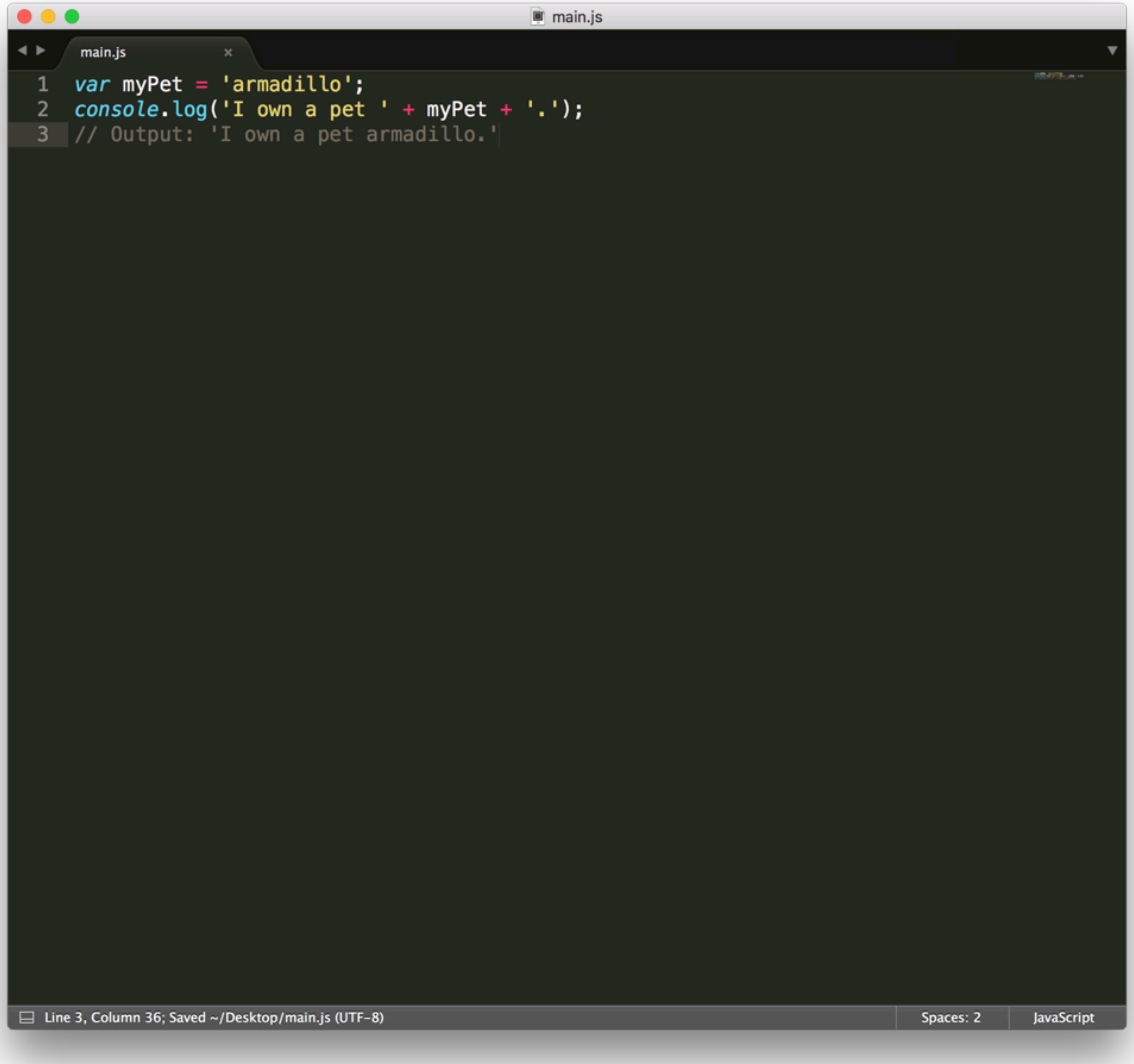


```javascript
var weatherCondition = 'Monday: Raining cats and dogs';
weatherCondition = 'Tuesday: Sunny';

console.log(weatherCondition);
// Output: 'Tuesday: Sunny'
```

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*.



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

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.

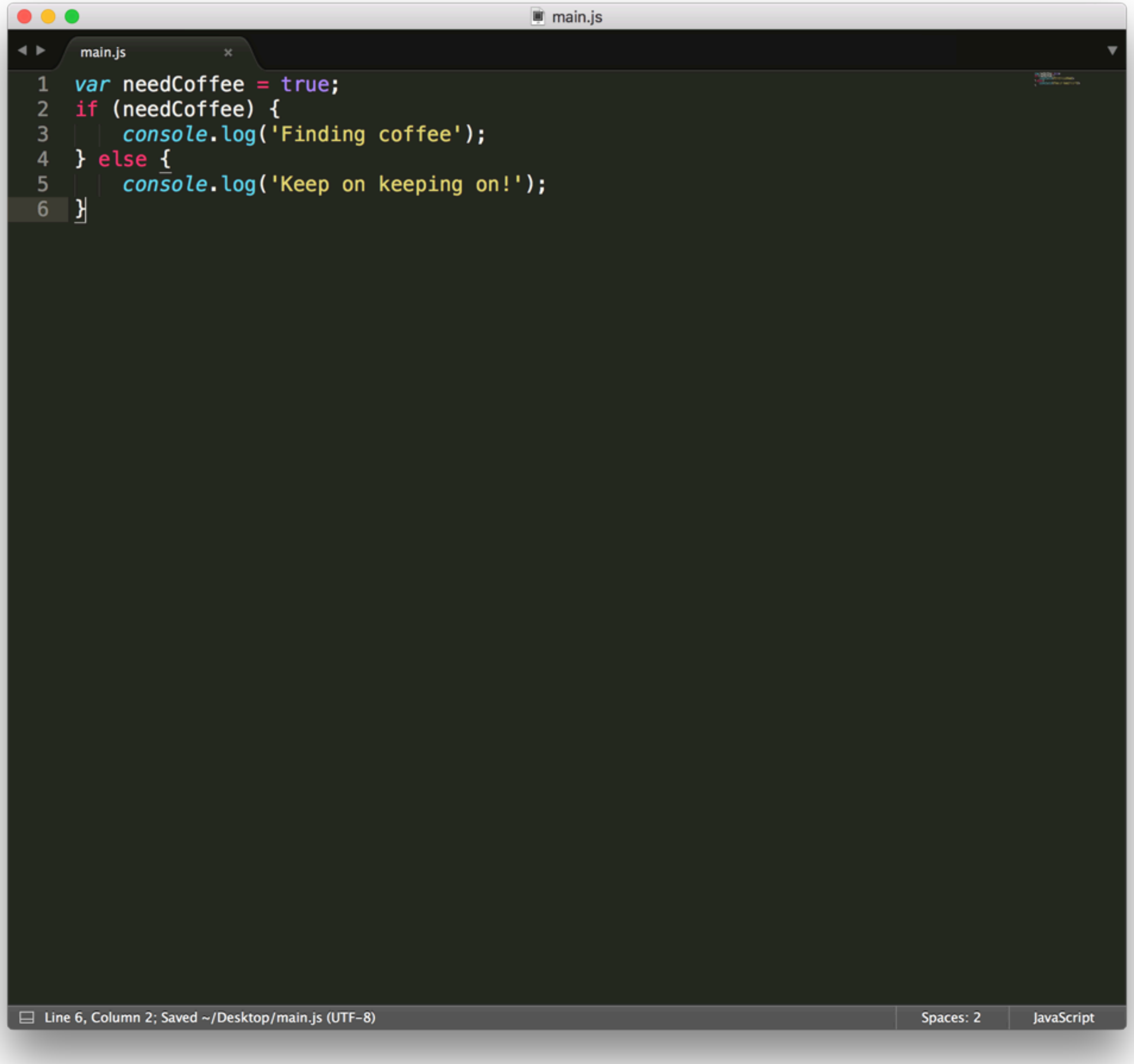
If you think about what we've been doing so far, we've been writing instructions for computers.

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*.

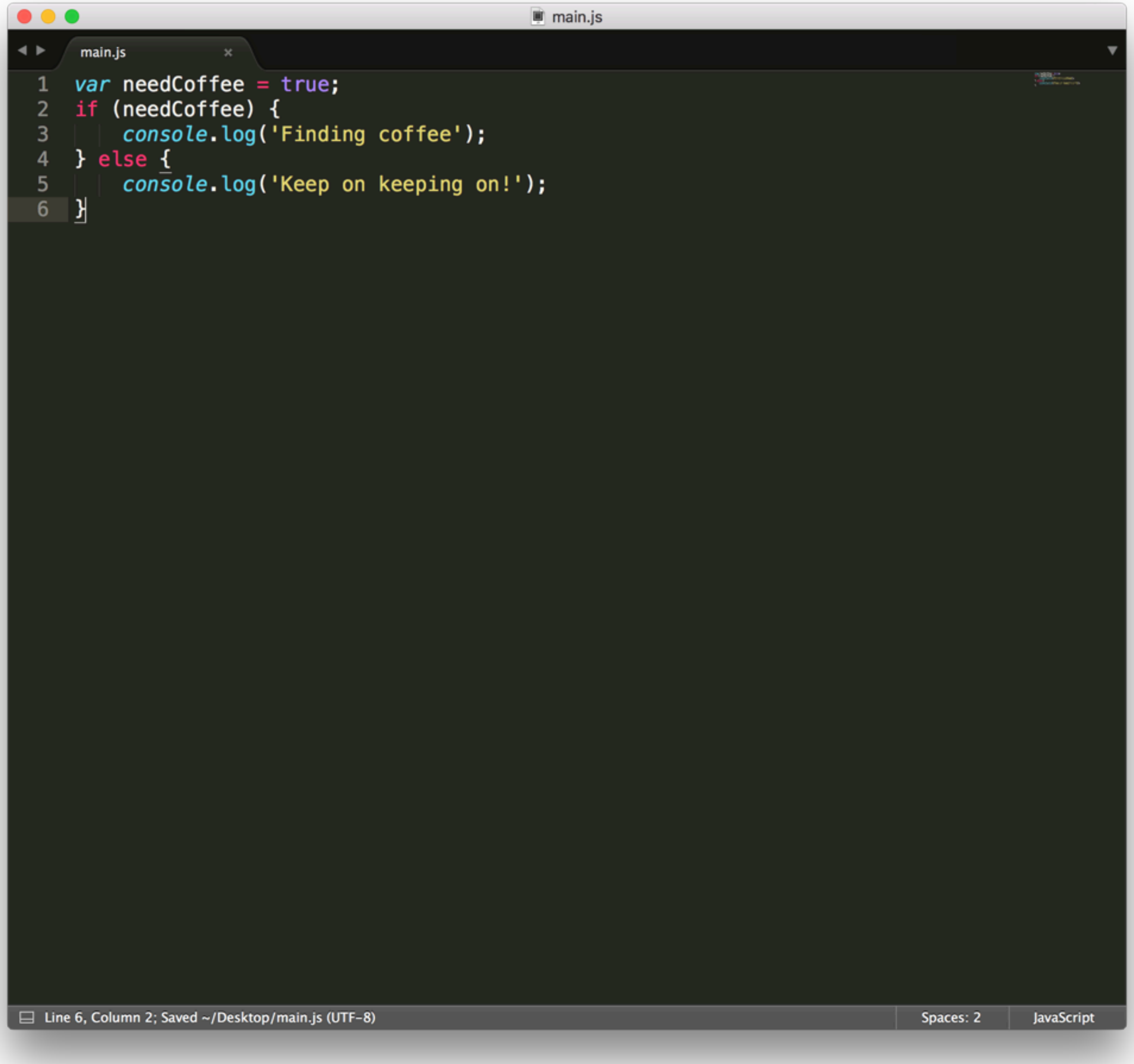
Many decisions we make everyday boil down to this sentence in some form:

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



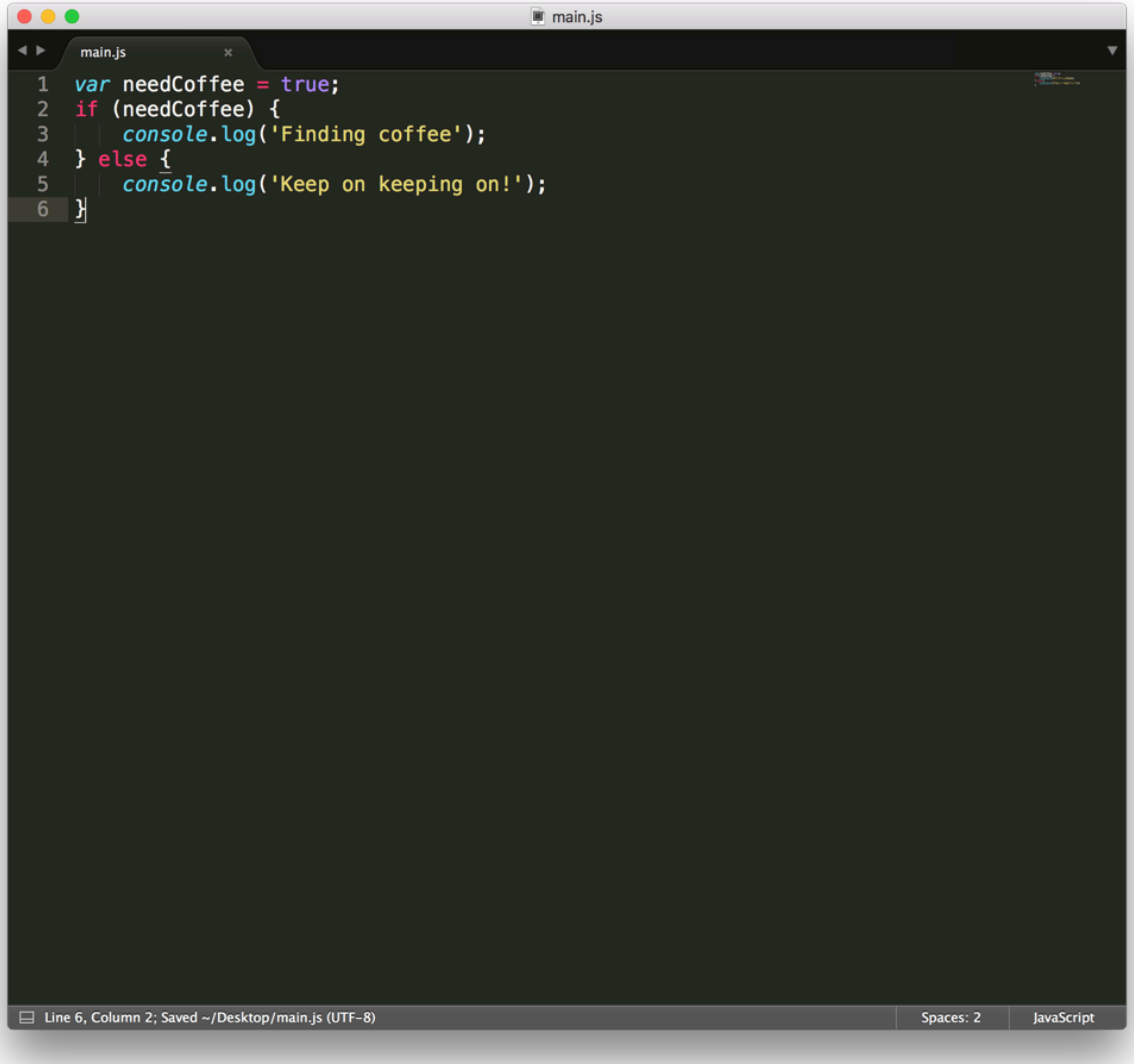
```
var needCoffee = true;
if (needCoffee) {
  console.log('Finding coffee');
} else {
  console.log('Keep on keeping on!');
}
```

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  }
```

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.



```javascript
var needCoffee = true;
if (needCoffee) {
  console.log('Finding coffee');
} else {
  console.log('Keep on keeping on!');
}
```

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: >

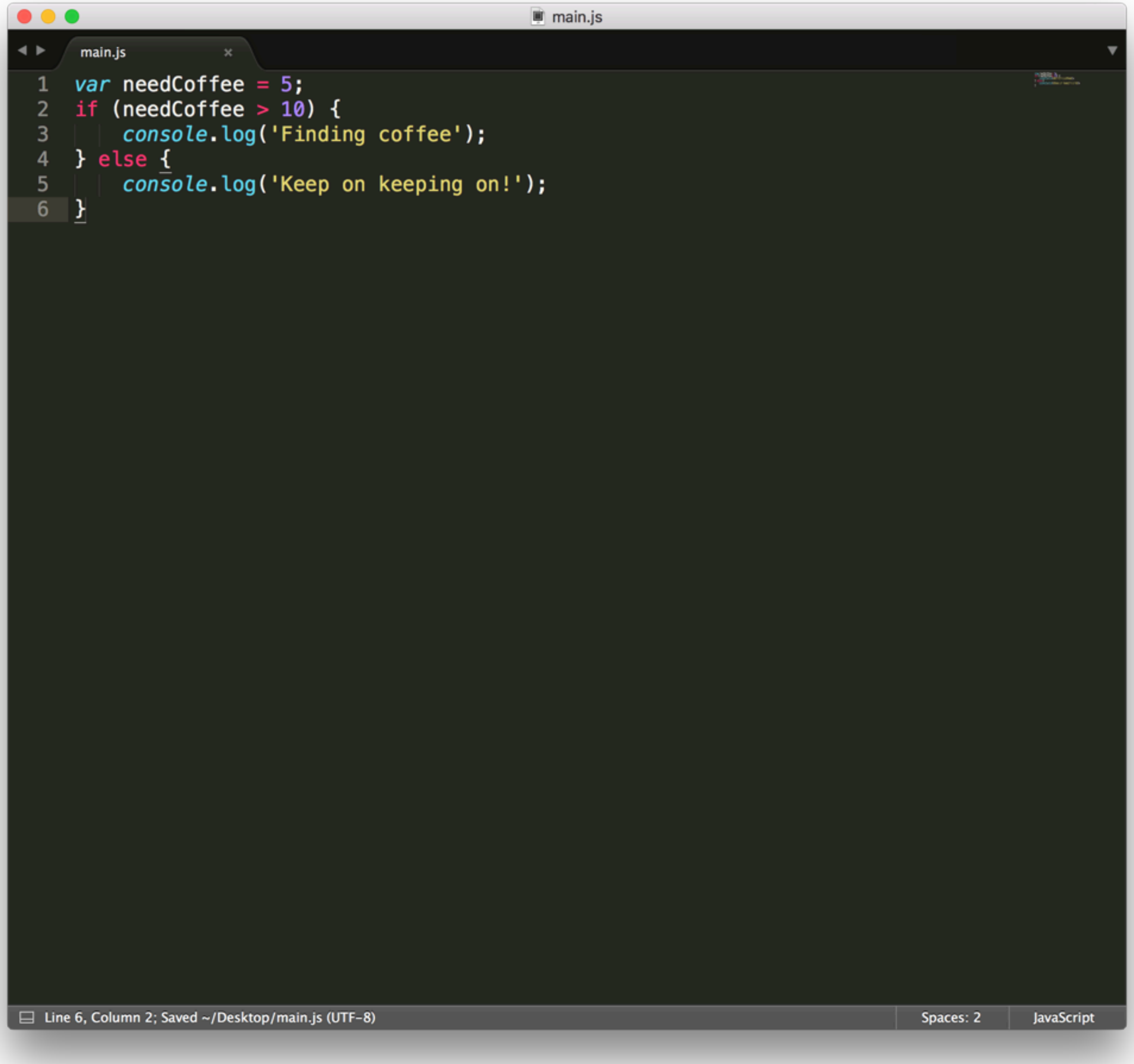
You may also recognize these:

Less than or equal to: <=

Greater than or equal to: >=

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?



```javascript
var needCoffee = 5;
if (needCoffee > 10) {
    console.log('Finding coffee');
} else {
    console.log('Keep on keeping on!');
}
```

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

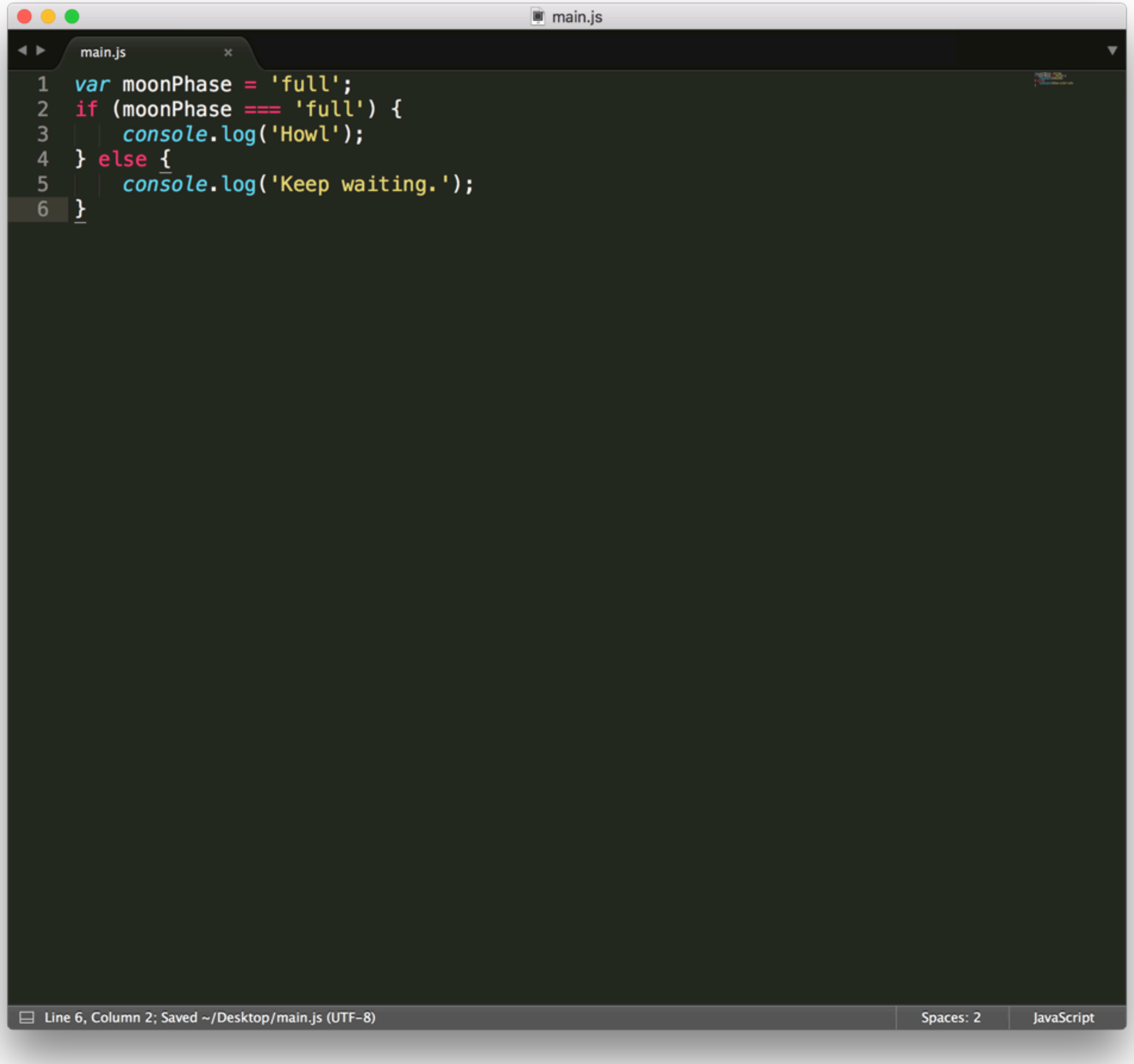
Spaces: 2

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).

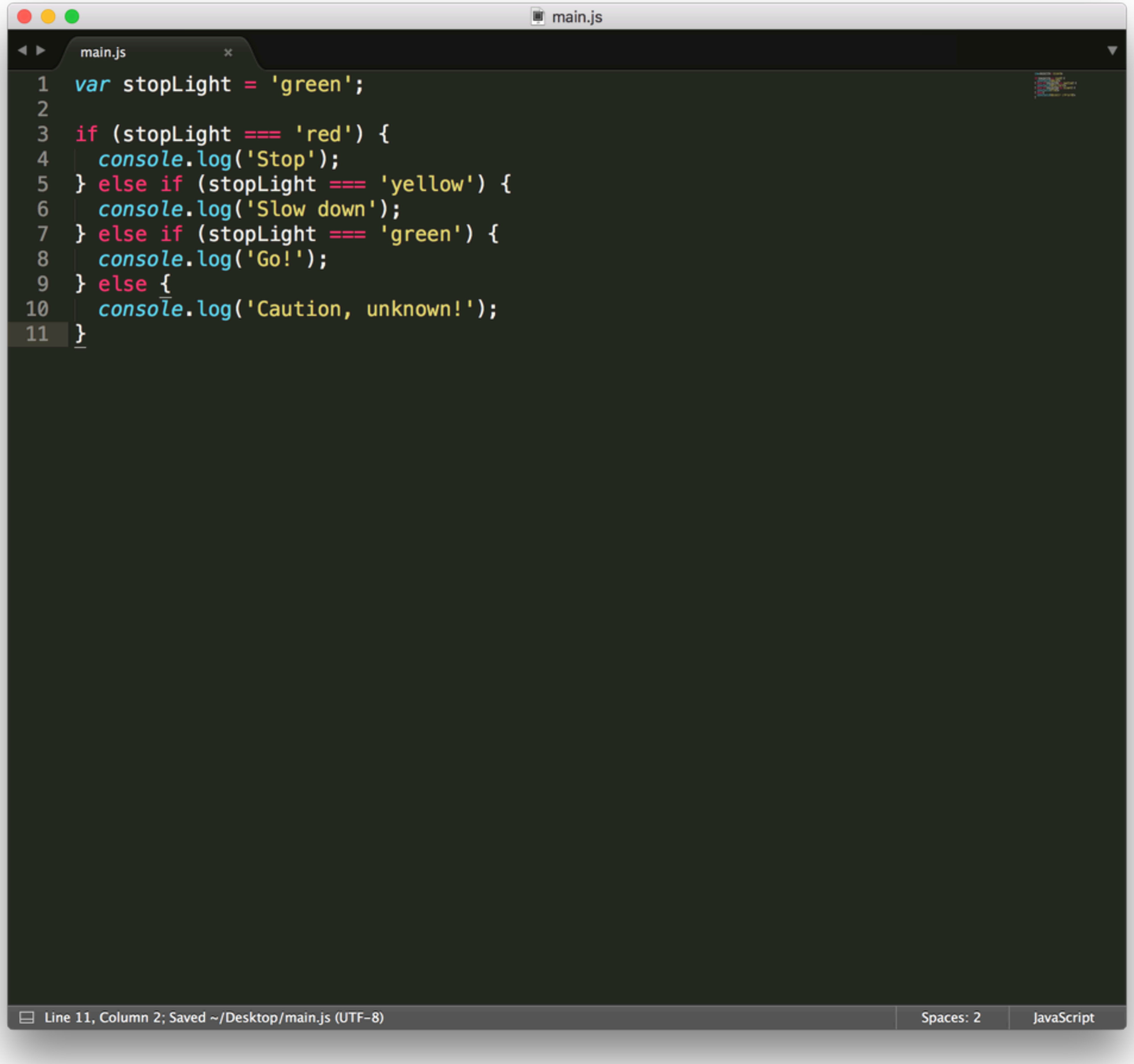
2. To check if two things *do not* equal each other, we can write `!==` (exclamation with two equals in a row).



```
var moonPhase = 'full';
if (moonPhase === 'full') {
  console.log('Howl');
} else {
  console.log('Keep waiting.');
}
```

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?



```
var stopLight = 'green';

if (stopLight === 'red') {
  console.log('Stop');
} else if (stopLight === 'yellow') {
  console.log('Slow down');
} else if (stopLight === 'green') {
  console.log('Go!');
} else {
  console.log('Caution, unknown!');
}
```

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

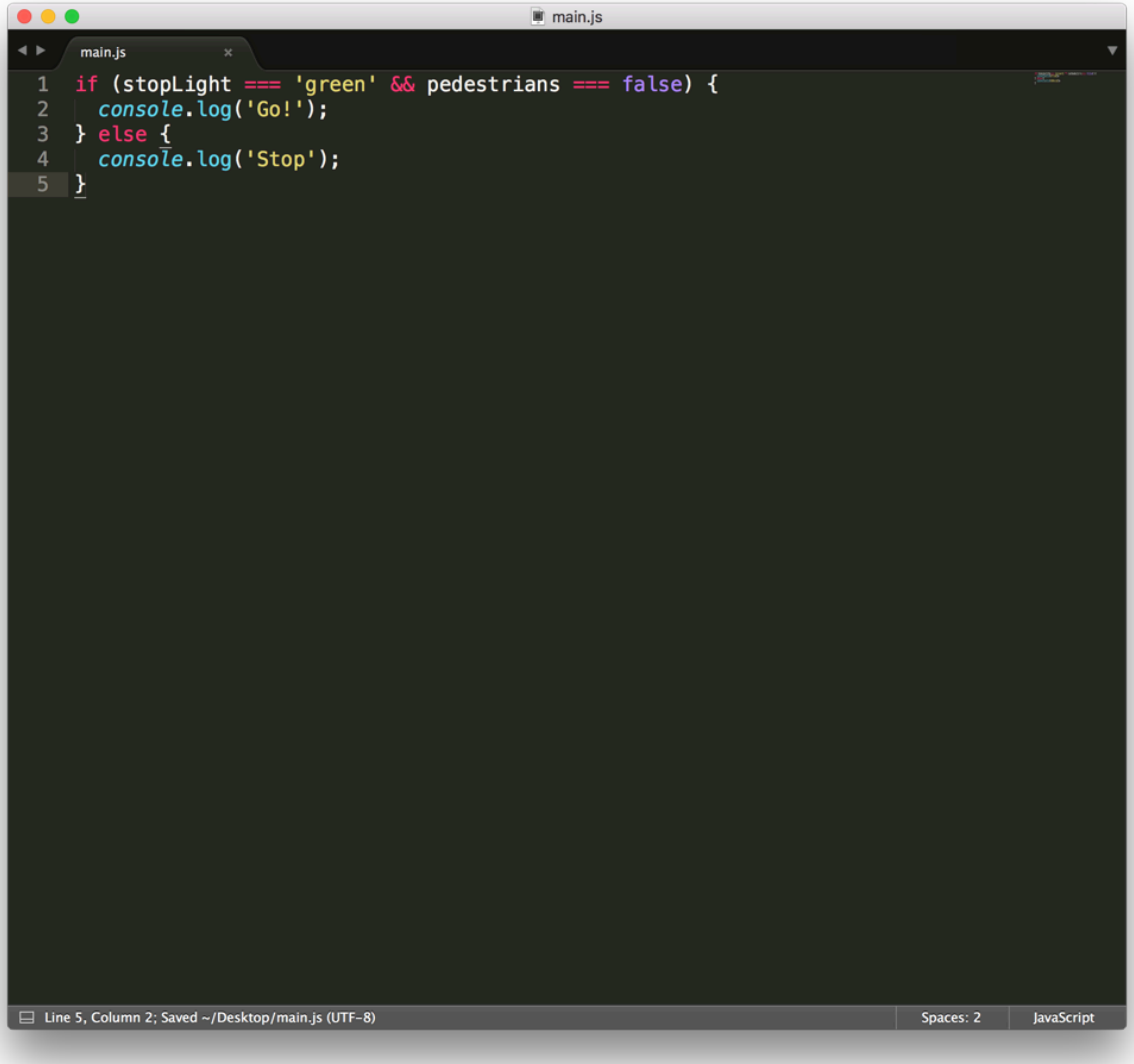
In English, sometimes we say "both of these things" or "either one of these things." Let's translate those phrases into JavaScript with some special operators called *logical operators*.

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 !==.



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

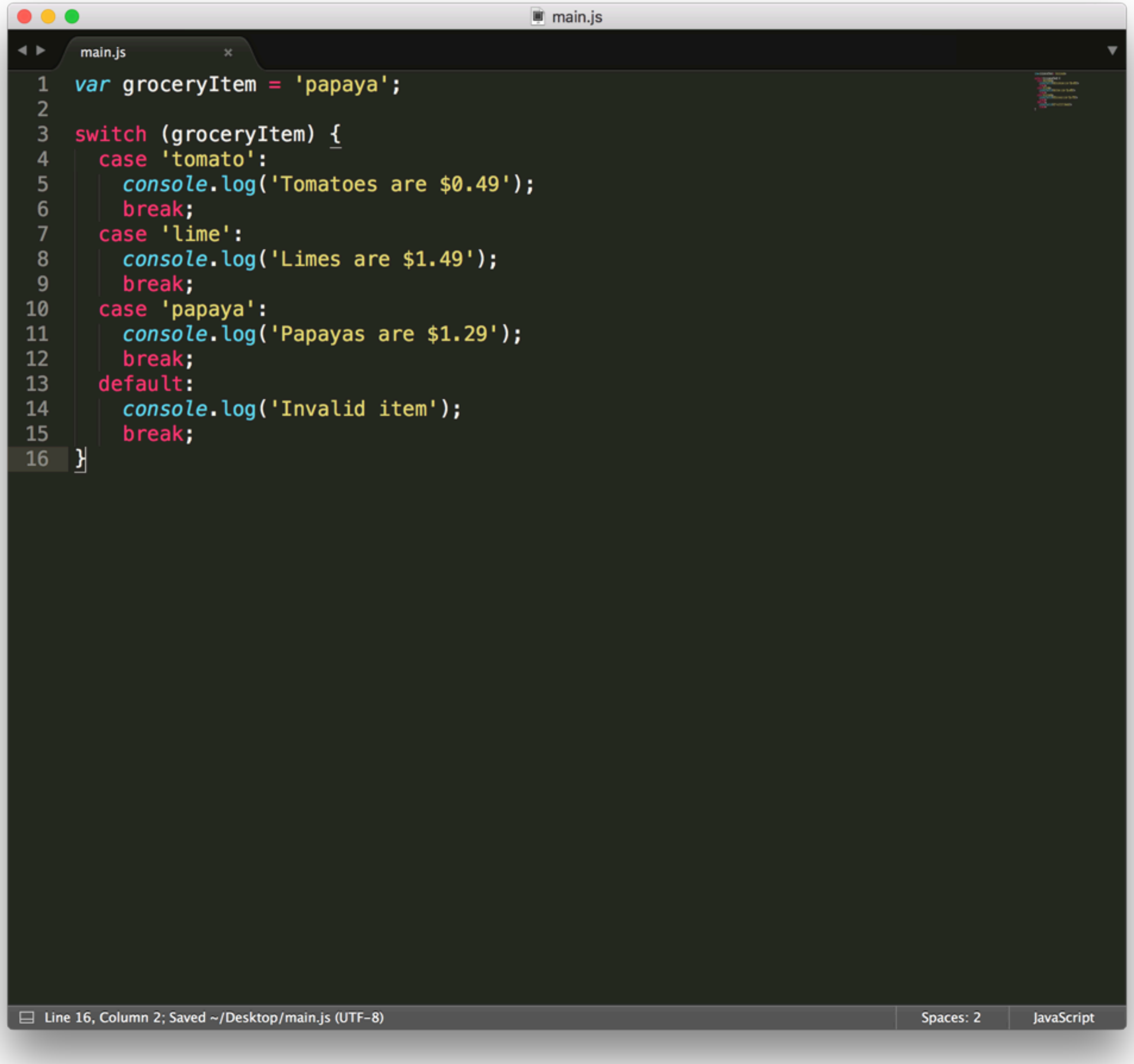
Line 5, Column 2; Saved ~/Desktop/main.js (UTF-8)

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.



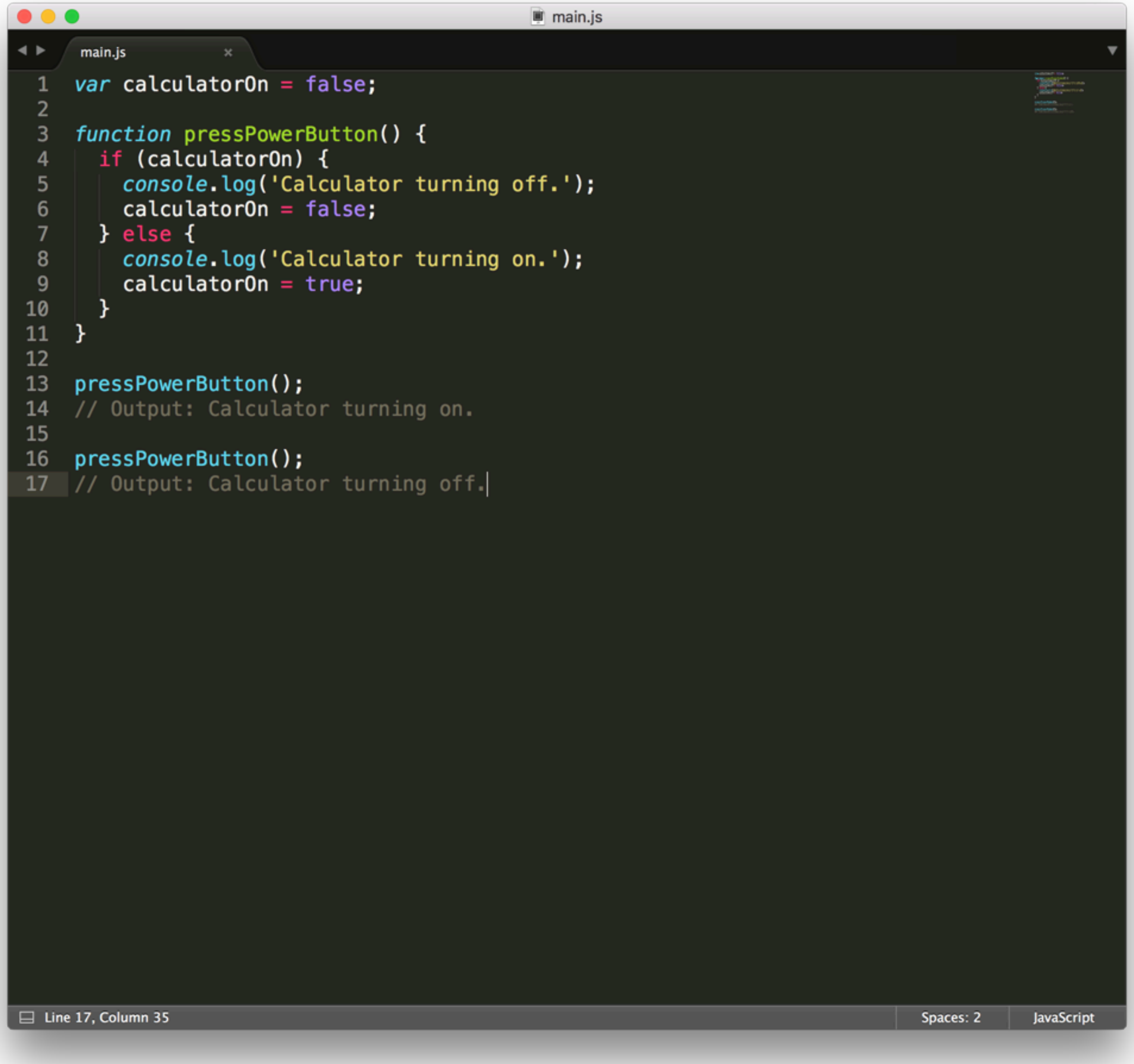
```javascript
var groceryItem = 'papaya';

switch (groceryItem) {
  case 'tomato':
    console.log('Tomatoes are $0.49');
    break;
  case 'lime':
    console.log('Limes are $1.49');
    break;
  case 'papaya':
    console.log('Papayas are $1.29');
    break;
  default:
    console.log('Invalid item');
    break;
}
```

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

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

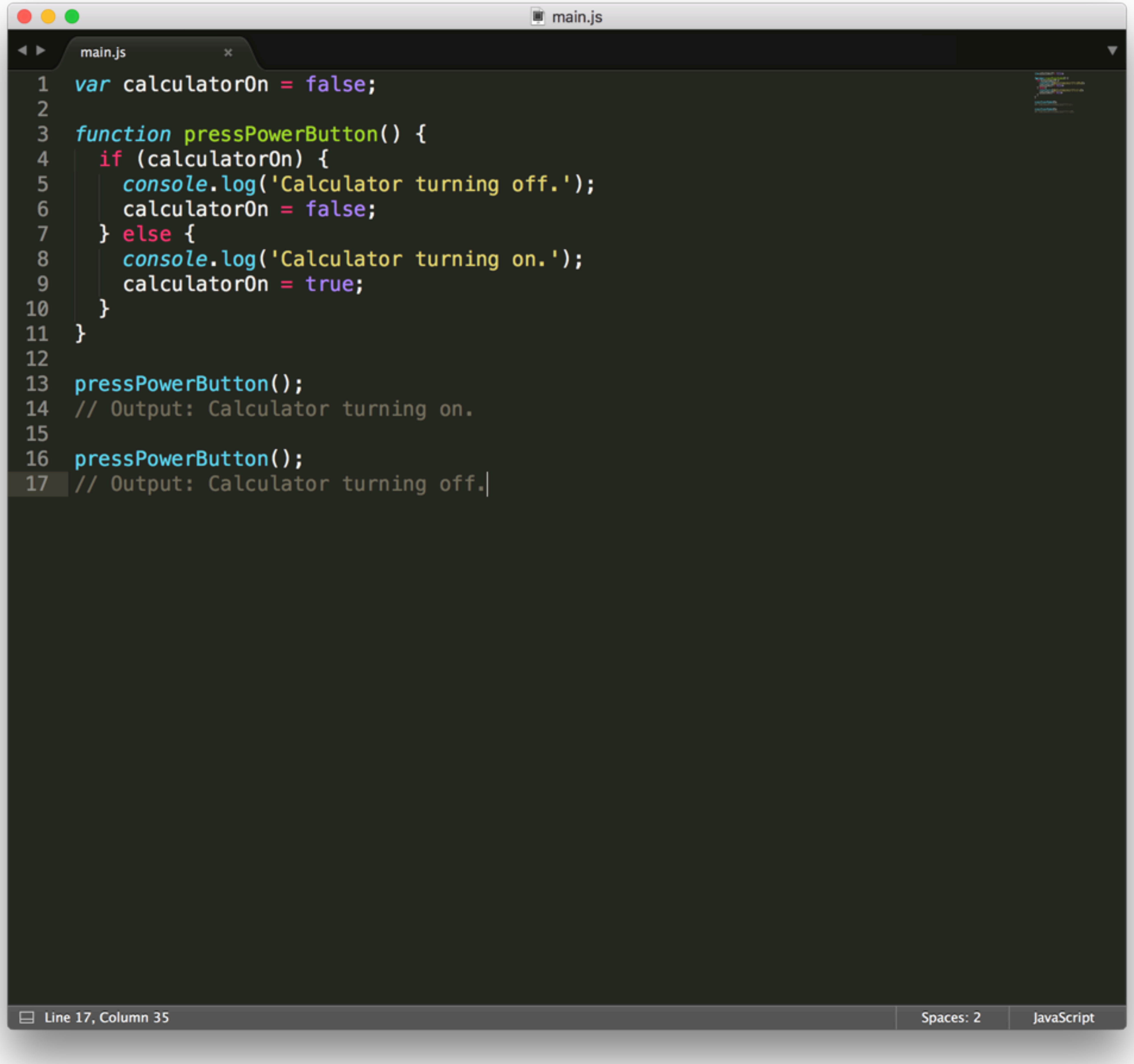
Functions are like recipes. They take data or variables, perform a set of tasks on them, and then return the result.

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.



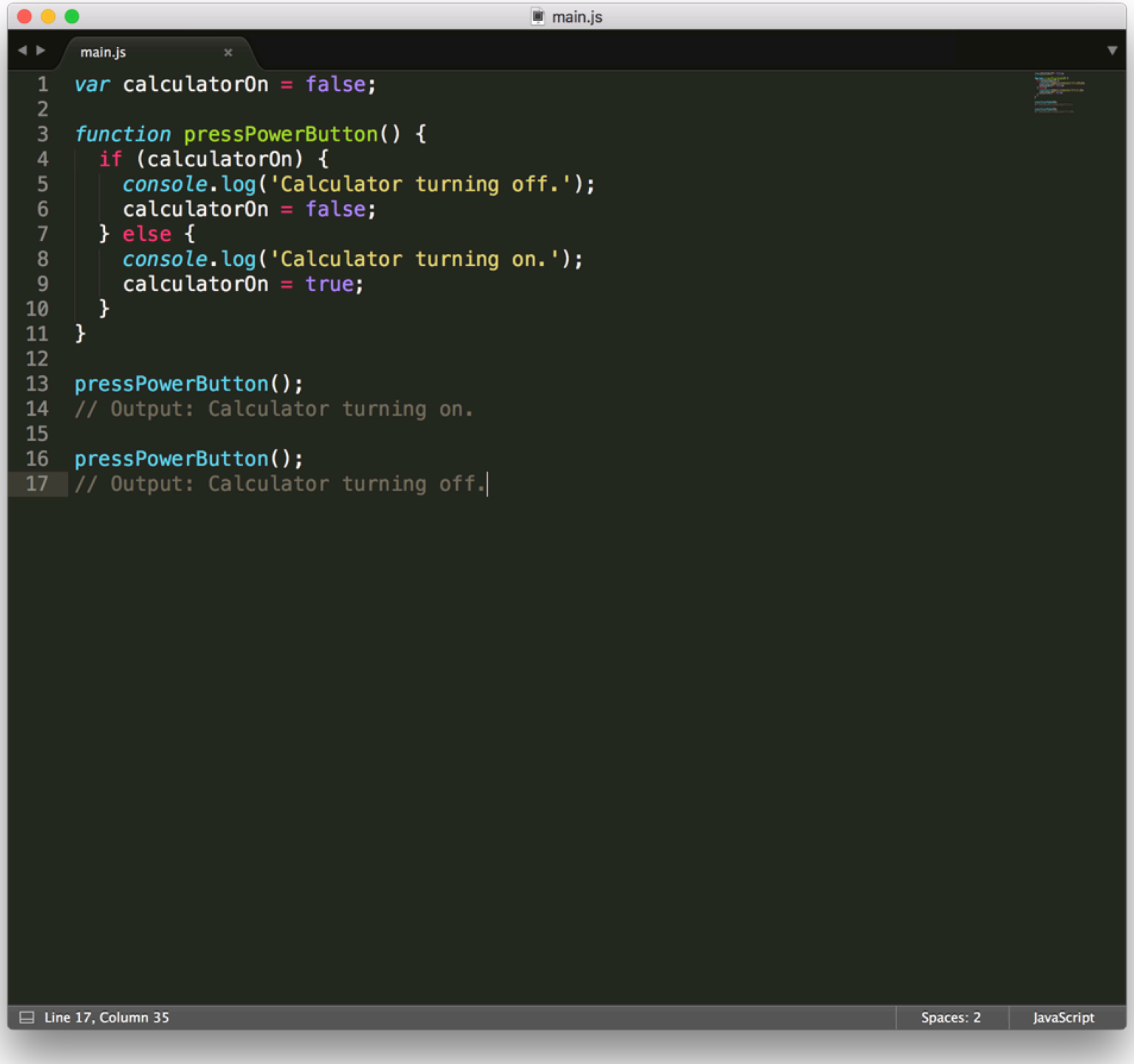
```
var calculatorOn = false;

function pressPowerButton() {
  if (calculatorOn) {
    console.log('Calculator turning off.');
    calculatorOn = false;
  } else {
    console.log('Calculator turning on.');
    calculatorOn = true;
  }
}

pressPowerButton();
// Output: Calculator turning on.

pressPowerButton();
// Output: Calculator turning off.
```

How does this code work?



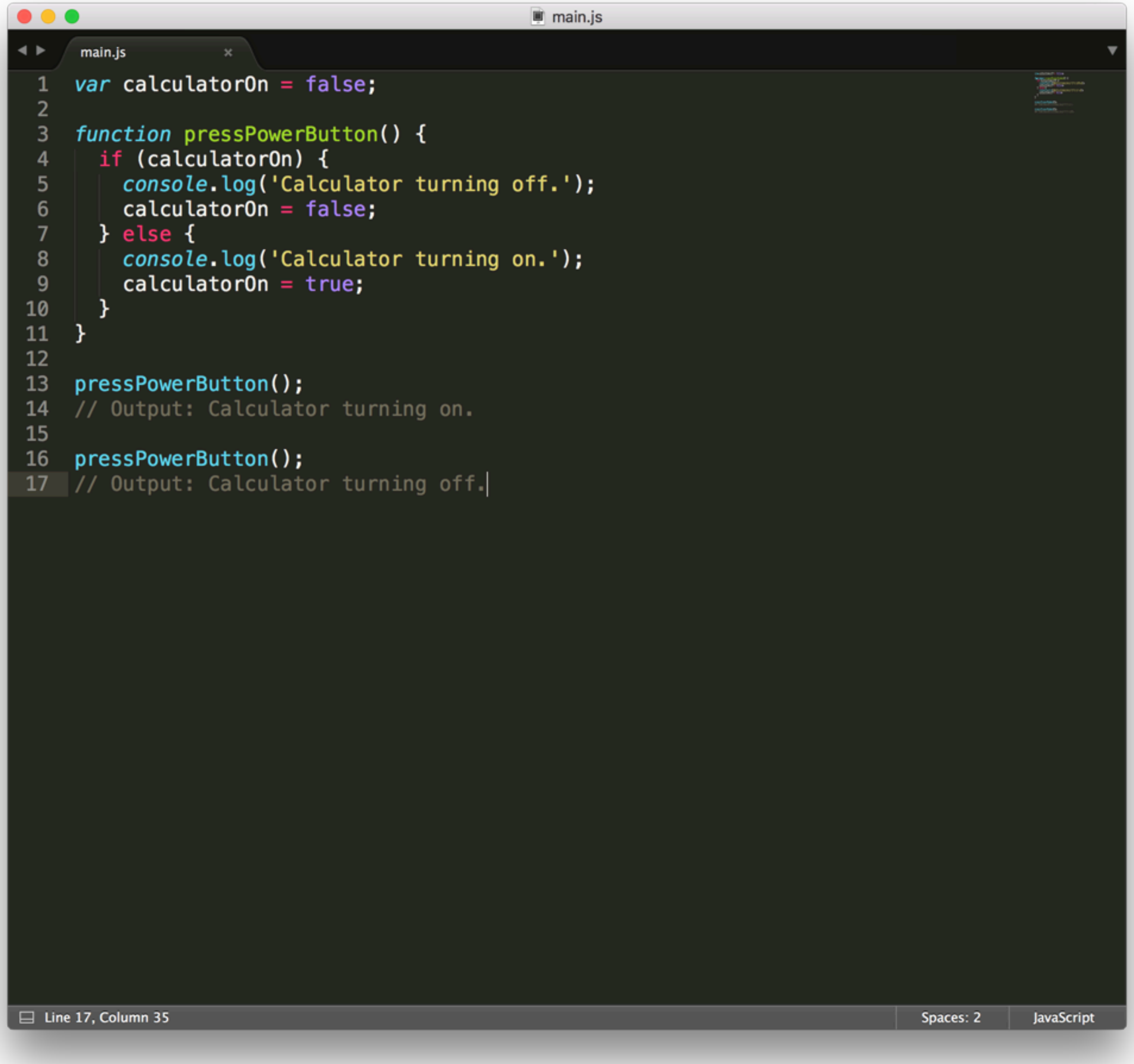
```
 1  var calculatorOn = false;
 2
 3  function pressPowerButton() {
 4    if (calculatorOn) {
 5      console.log('Calculator turning off.');
 6      calculatorOn = false;
 7    } else {
 8      console.log('Calculator turning on.');
 9      calculatorOn = true;
10    }
11  }
12
13  pressPowerButton();
14  // Output: Calculator turning on.
15
16  pressPowerButton();
17  // Output: Calculator turning off.
```

On line 3, there's a function named pressPowerButton



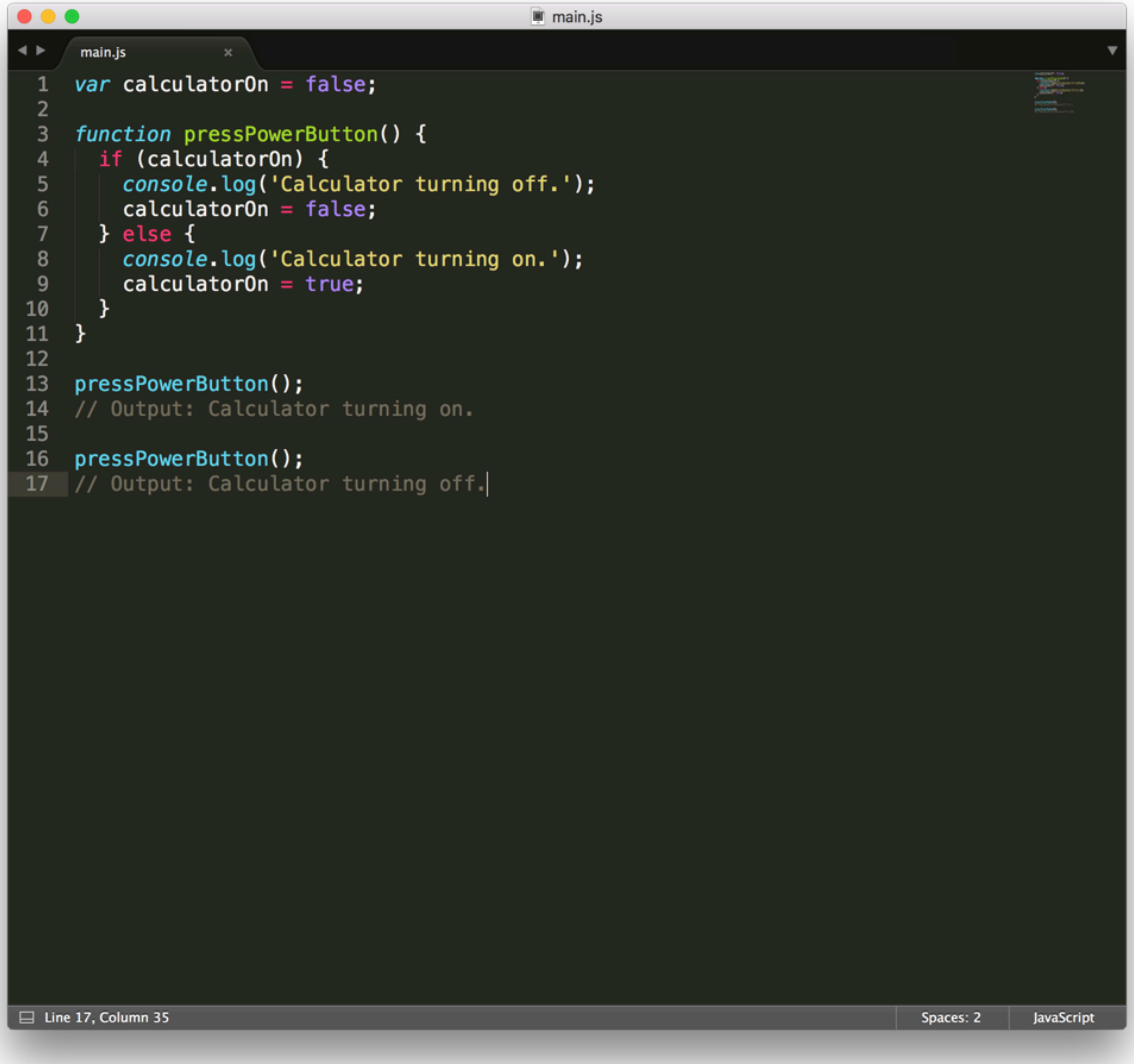
```javascript
var calculatorOn = false;

function pressPowerButton() {
  if (calculatorOn) {
    console.log('Calculator turning off.');
    calculatorOn = false;
  } else {
    console.log('Calculator turning on.');
    calculatorOn = true;
  }
}

pressPowerButton();
// Output: Calculator turning on.

pressPowerButton();
// Output: Calculator turning off.
```

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



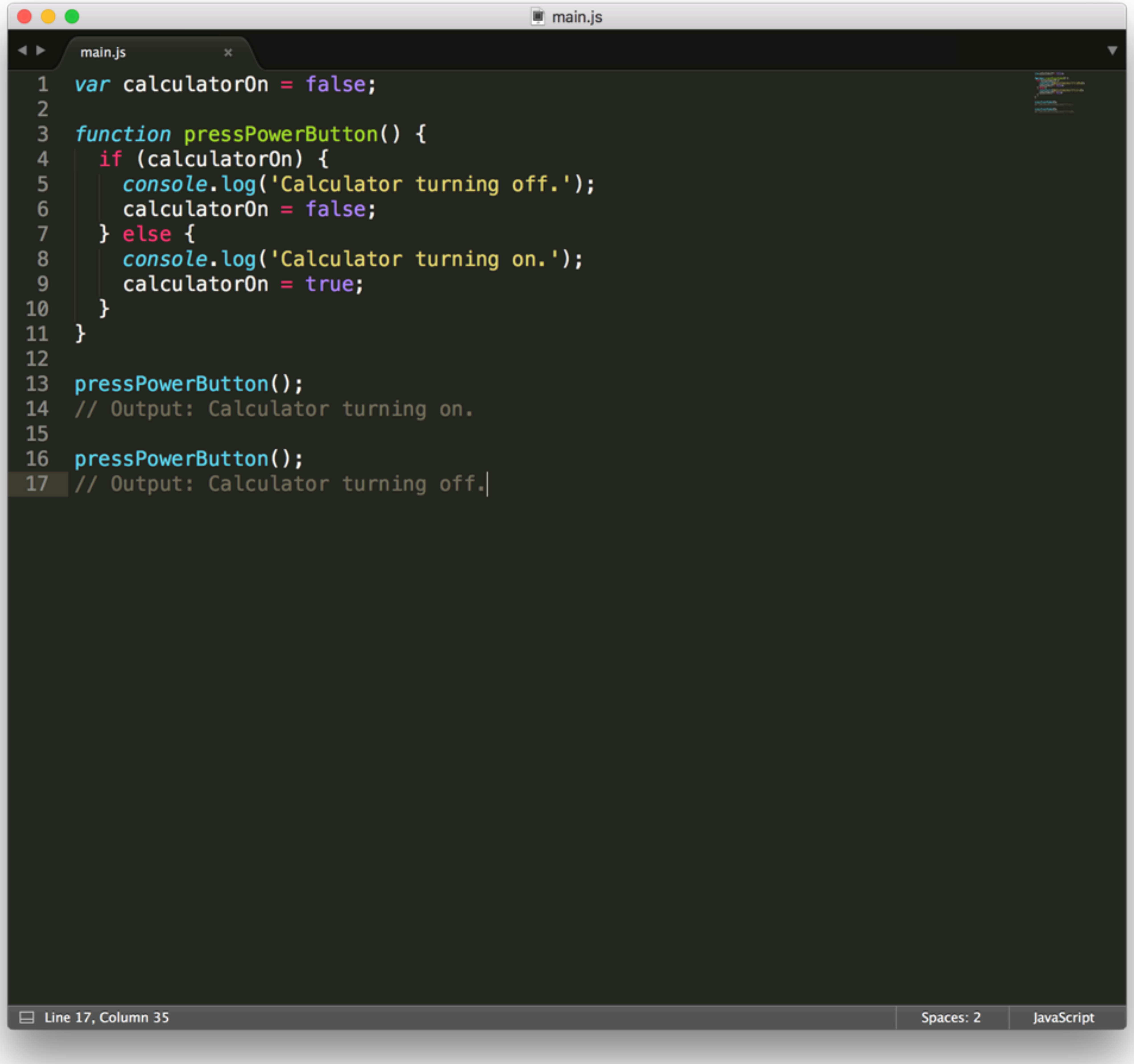
```javascript
var calculatorOn = false;

function pressPowerButton() {
  if (calculatorOn) {
    console.log('Calculator turning off.');
    calculatorOn = false;
  } else {
    console.log('Calculator turning on.');
    calculatorOn = true;
  }
}

pressPowerButton();
// Output: Calculator turning on.

pressPowerButton();
// Output: Calculator turning off.
```

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.



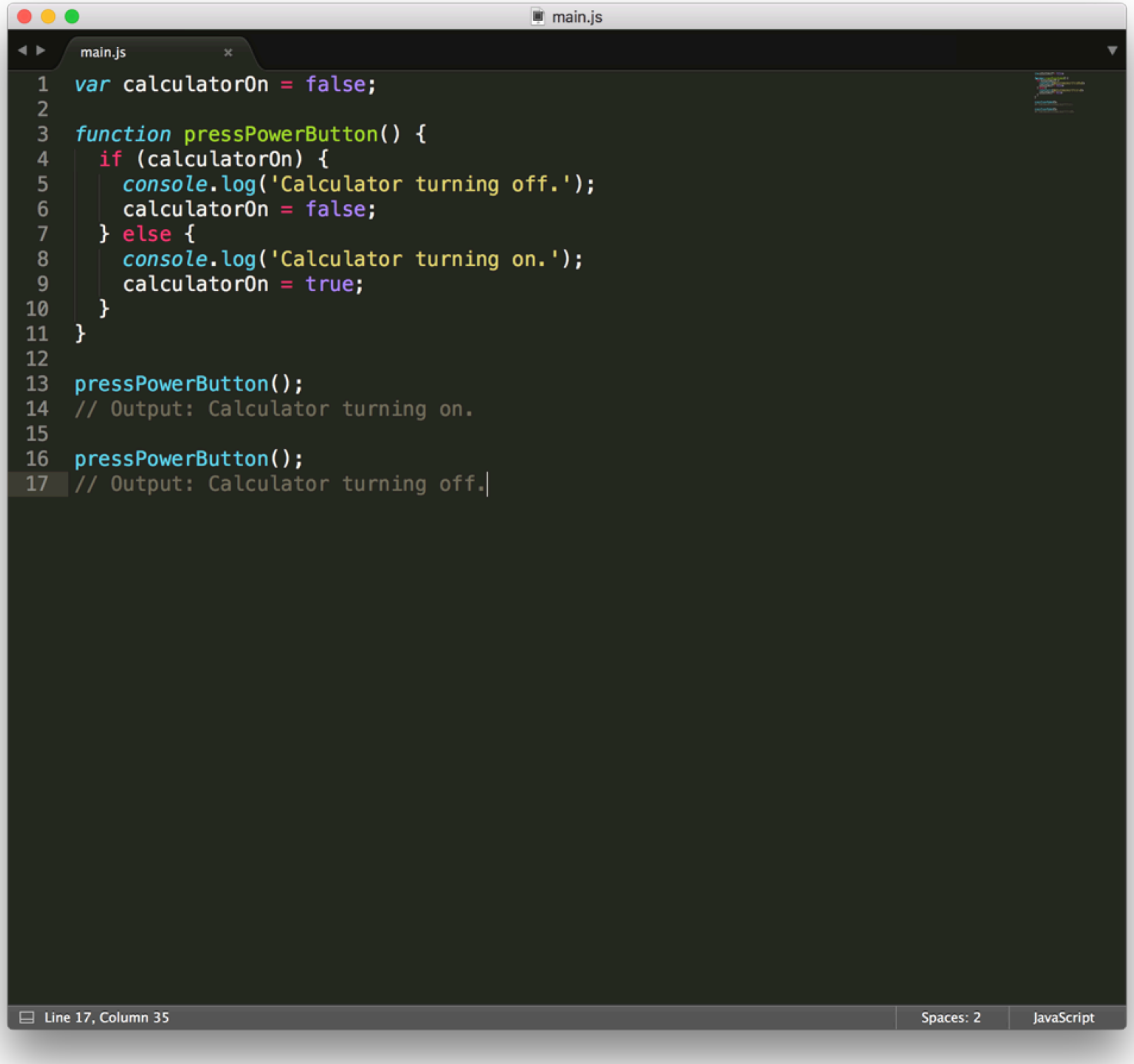
```
var calculatorOn = false;

function pressPowerButton() {
  if (calculatorOn) {
    console.log('Calculator turning off.');
    calculatorOn = false;
  } else {
    console.log('Calculator turning on.');
    calculatorOn = true;
  }
}

pressPowerButton();
// Output: Calculator turning on.

pressPowerButton();
// Output: Calculator turning off.
```

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



```javascript
var calculatorOn = false;

function pressPowerButton() {
  if (calculatorOn) {
    console.log('Calculator turning off.');
    calculatorOn = false;
  } else {
    console.log('Calculator turning on.');
    calculatorOn = true;
  }
}

pressPowerButton();
// Output: Calculator turning on.

pressPowerButton();
// 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.');
 6      calculatorOn = false;
 7    } else {
 8      console.log('Calculator turning on.');
 9      calculatorOn = true;
10    }
11  }
12
13  pressPowerButton();
14  // Output: Calculator turning on.
15
16  pressPowerButton();
17  // Output: Calculator turning off.
```

Line 17, Column 35

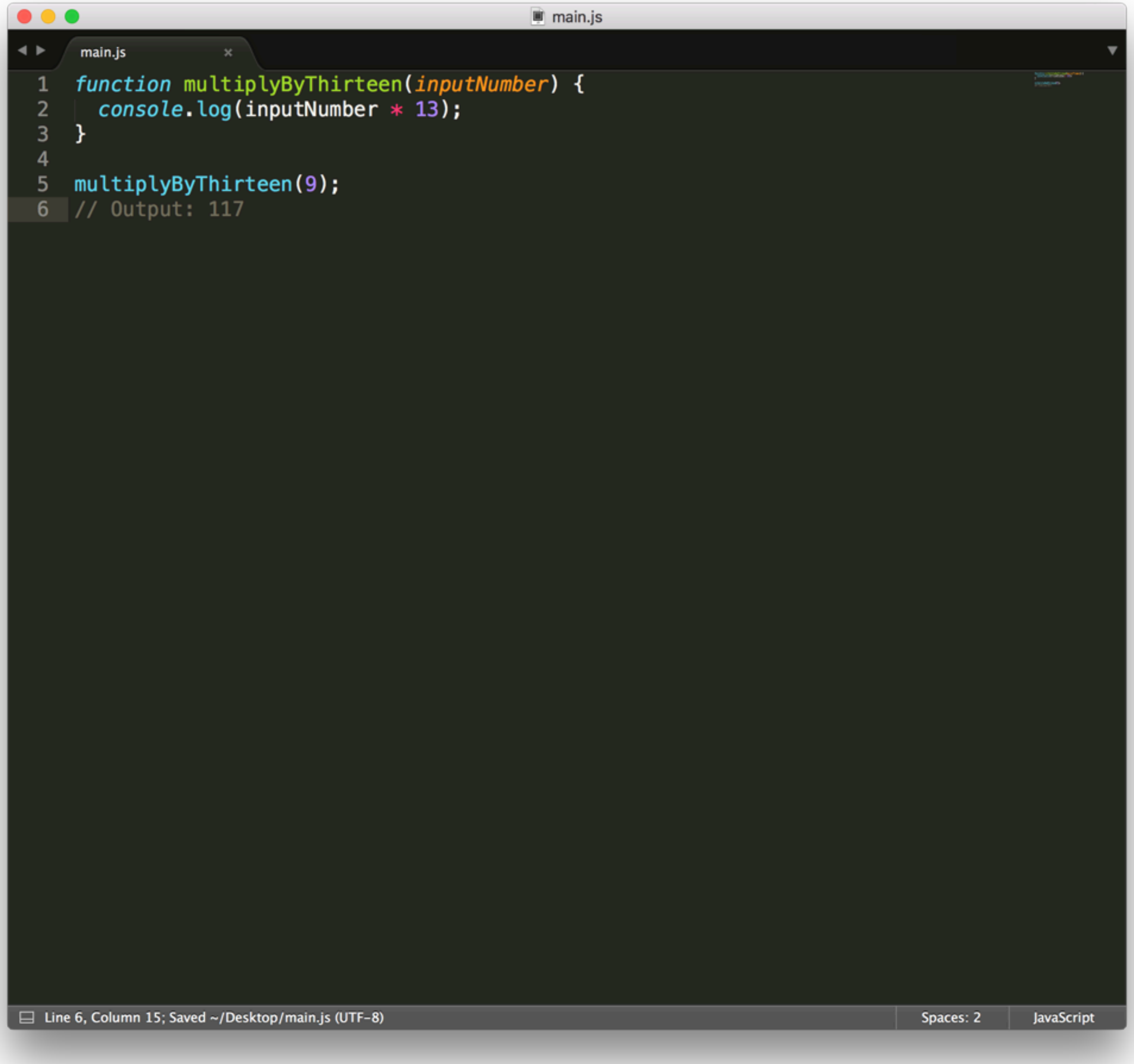
Spaces: 2

JavaScript

Lines 13 and 16 call the funtion.

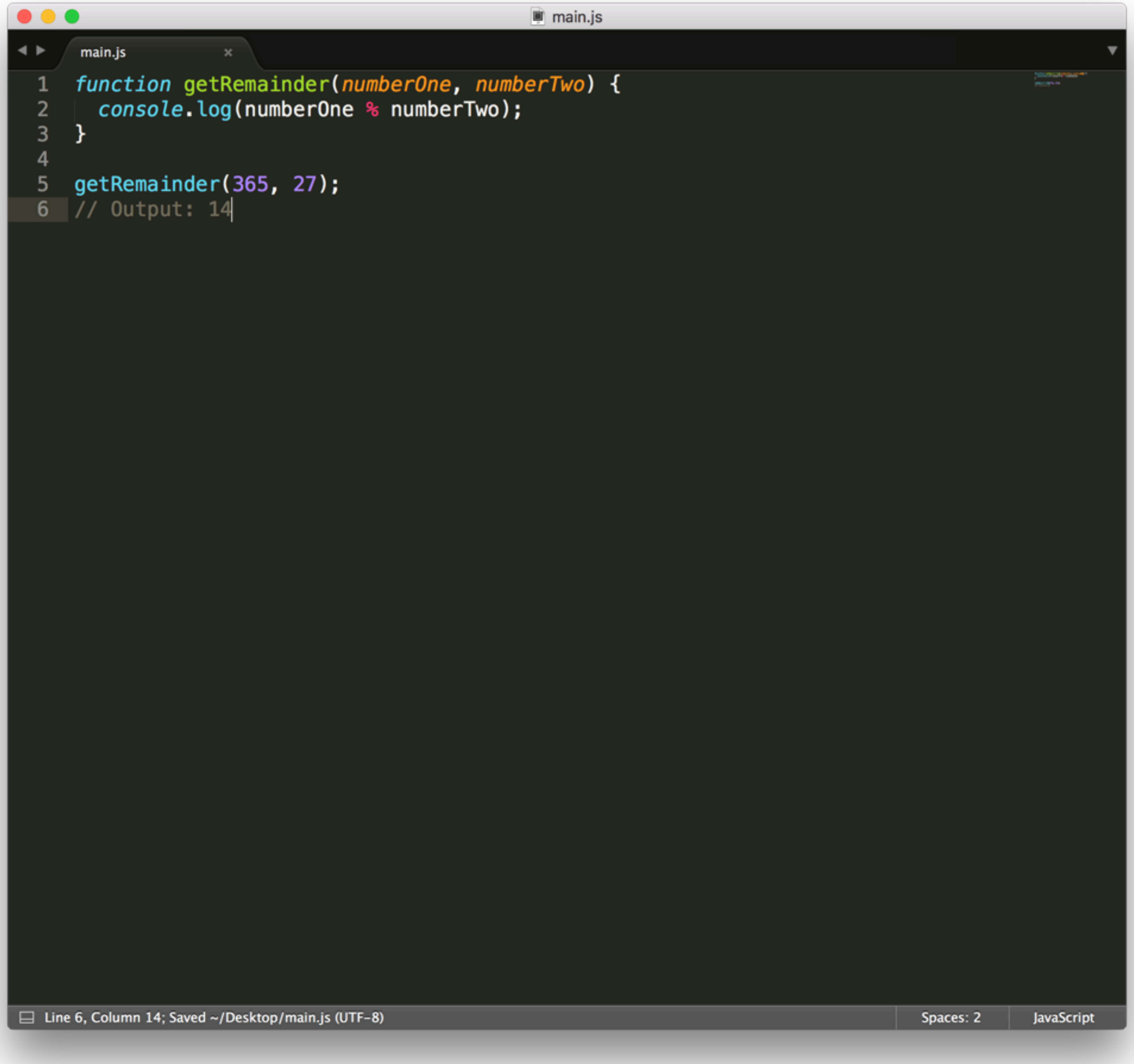
Let’s try something else. A calculator program should be able to perform a math operation on a number. We should be able to give a calculator a number, have it perform a task on it like multiplication, then print a result.

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



```
function multiplyByThirteen(inputNumber) {
  console.log(inputNumber * 13);
}

multiplyByThirteen(9);
// Output: 117
```

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

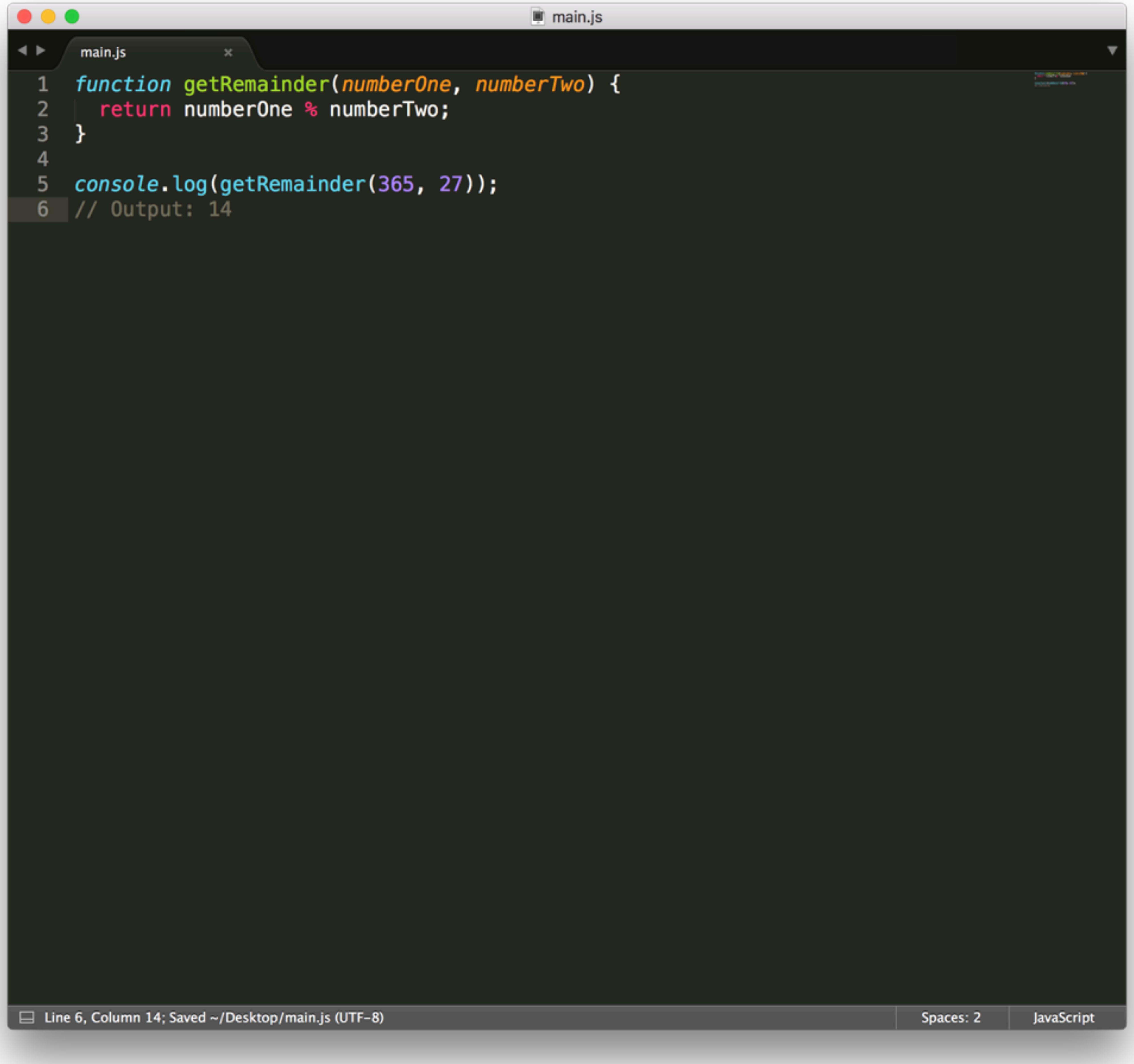


```
function getRemainder(numberOne, numberTwo) {
  console.log(numberOne % numberTwo);
}

getRemainder(365, 27);
// Output: 14
```

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.



```
function getRemainder(numberOne, numberTwo) {
  return numberOne % numberTwo;
}

console.log(getRemainder(365, 27));
// Output: 14
```

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.

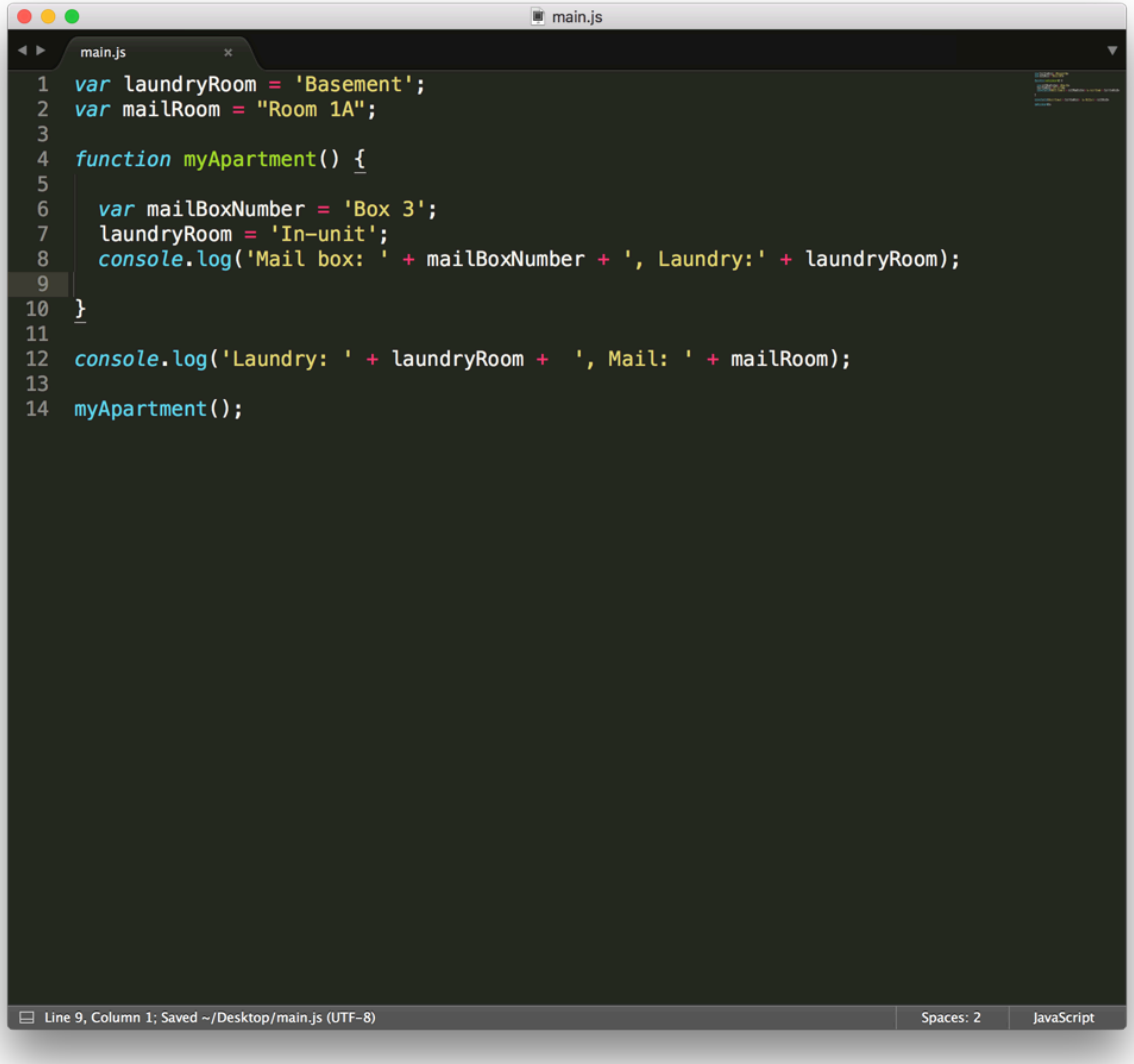
Think of it like an apartment building. Everyone who lives in the apartment building is under the *global scope* of the building and its manager. So, if there are rats in the shared laundry room, everyone has access to the laundry machines, and the rats.

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*.



```
var laundryRoom = 'Basement';
var mailRoom = "Room 1A";

function myApartment() {

  var mailBoxNumber = 'Box 3';
  laundryRoom = 'In-unit';
  console.log('Mail box: ' + mailBoxNumber + ', Laundry:' + laundryRoom);

}

console.log('Laundry: ' + laundryRoom +  ', Mail: ' + mailRoom);

myApartment();
```

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

Spaces: 2

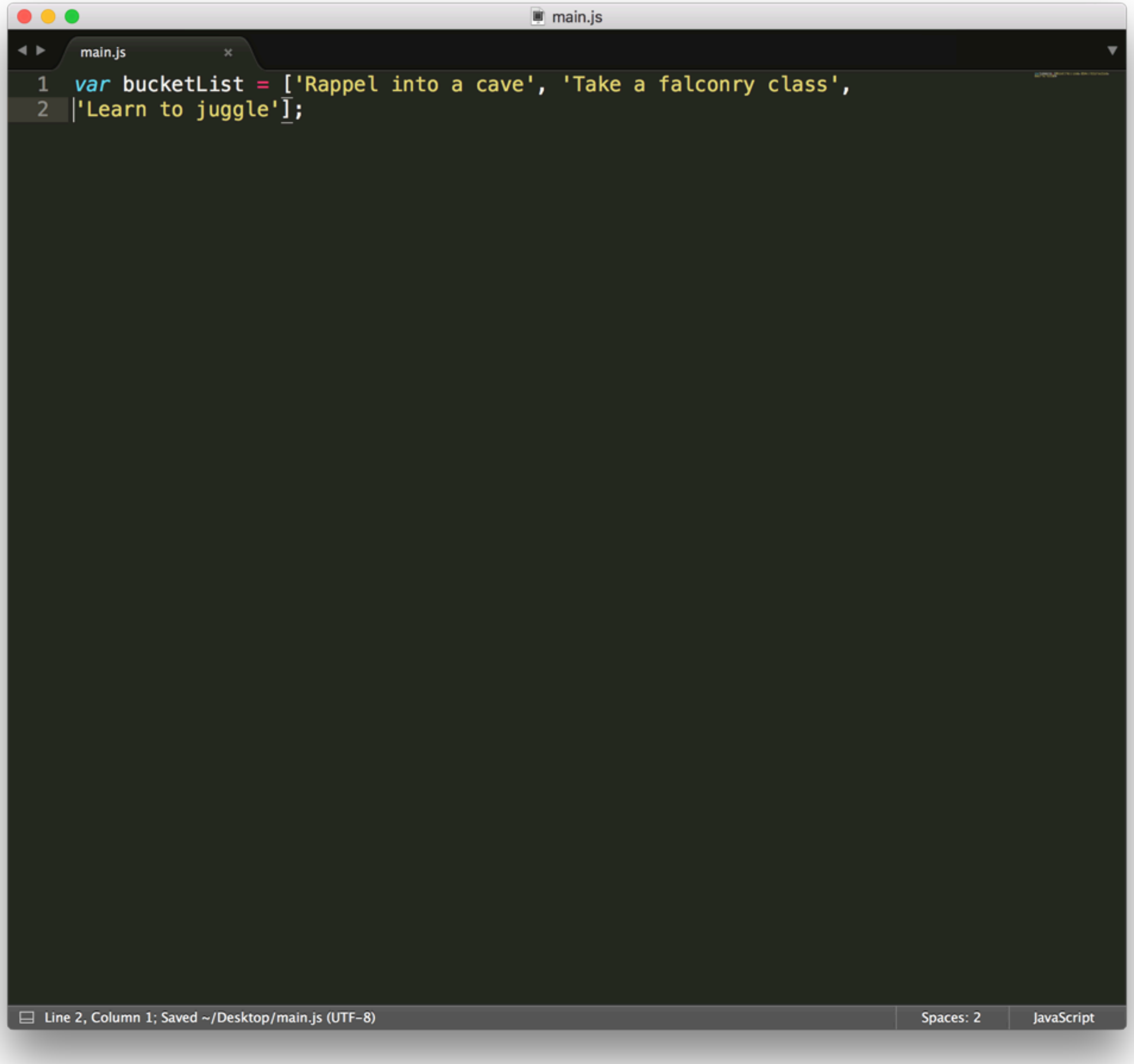
JavaScript

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*



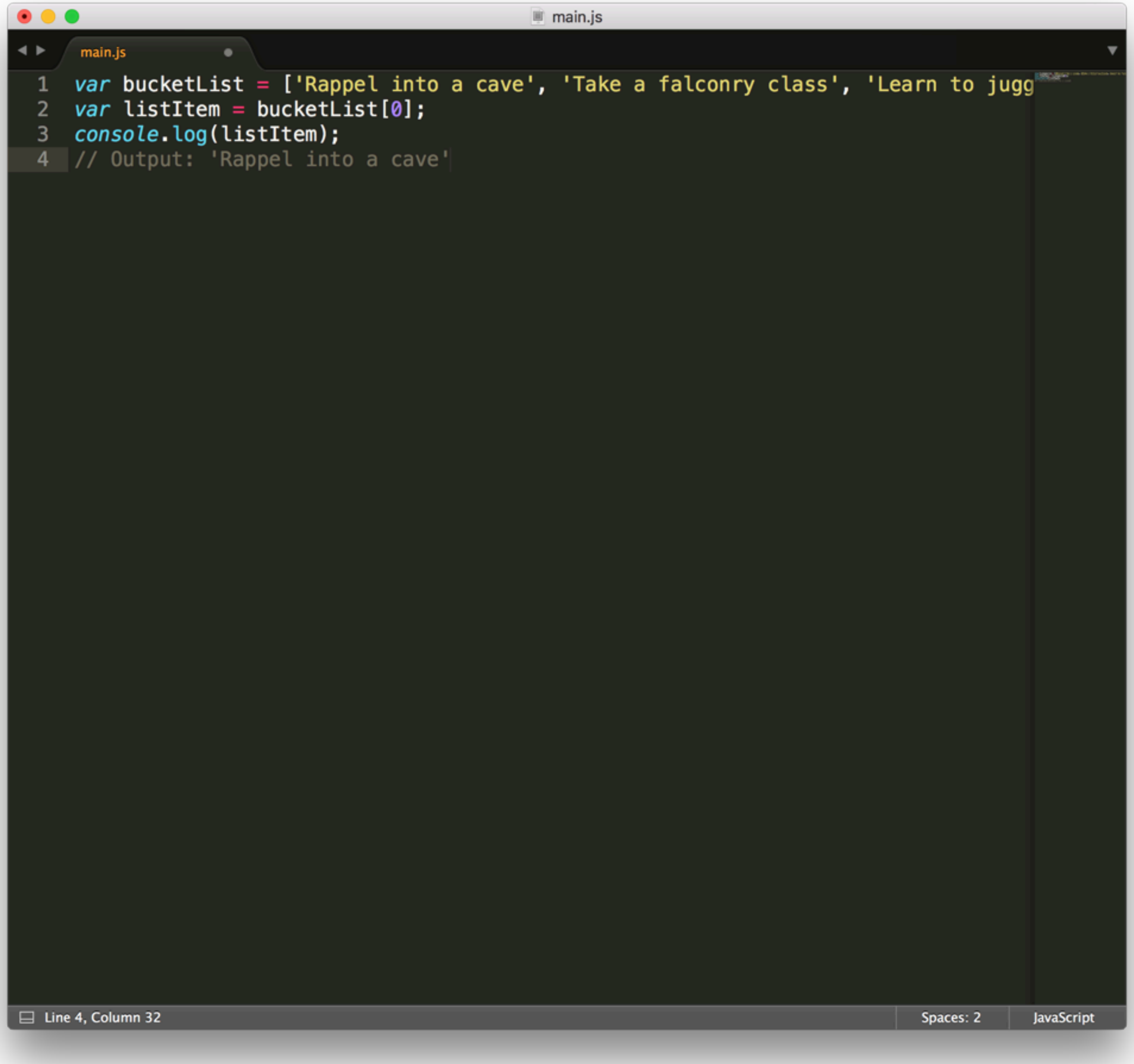
```javascript
var bucketList = ['Rappel into a cave', 'Take a falconry class',
'Learn to juggle'];
```

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

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

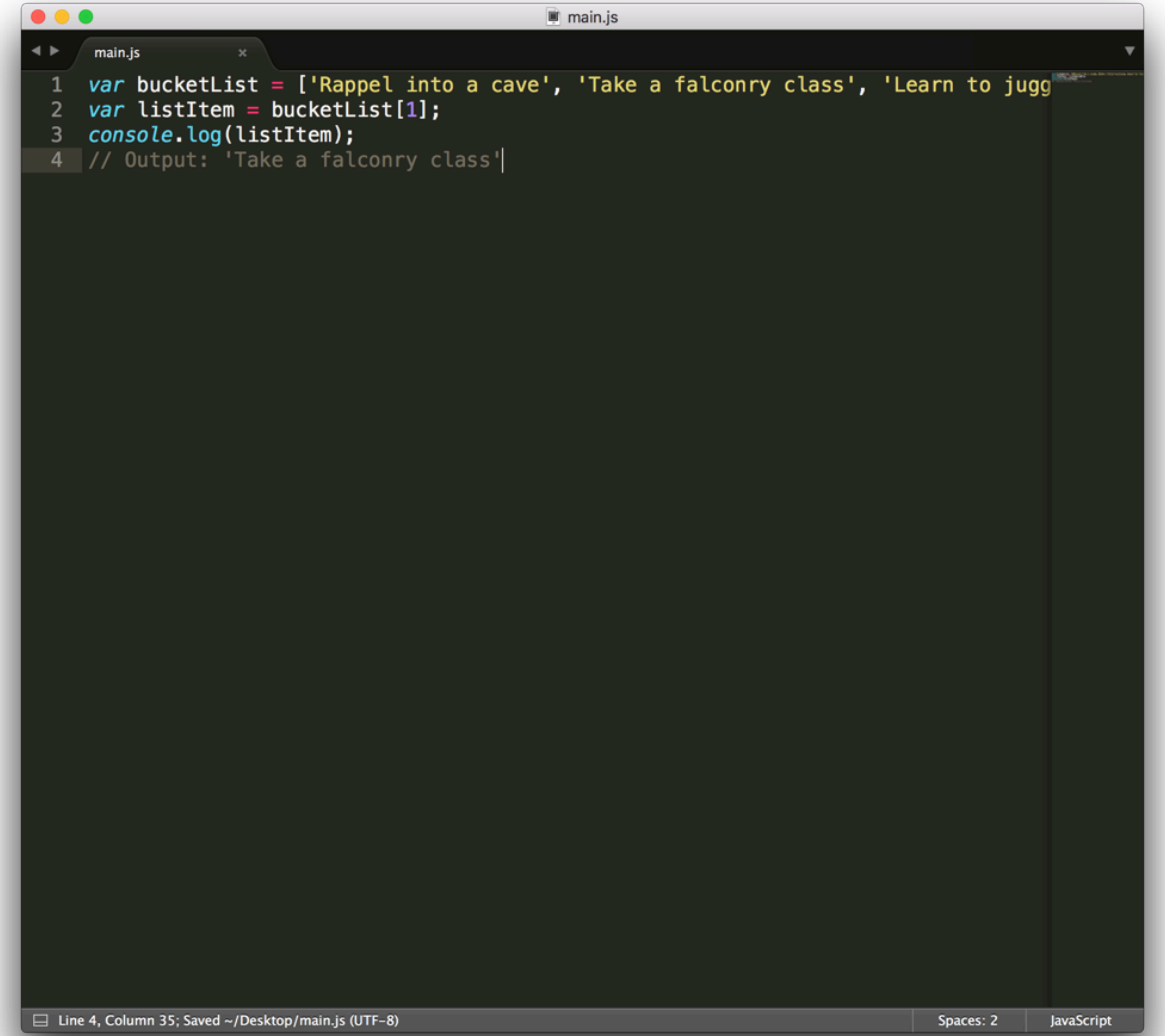
Each item in an array has a numbered position. We can access an item using its number, just like we would in an ordinary list.

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*.



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

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

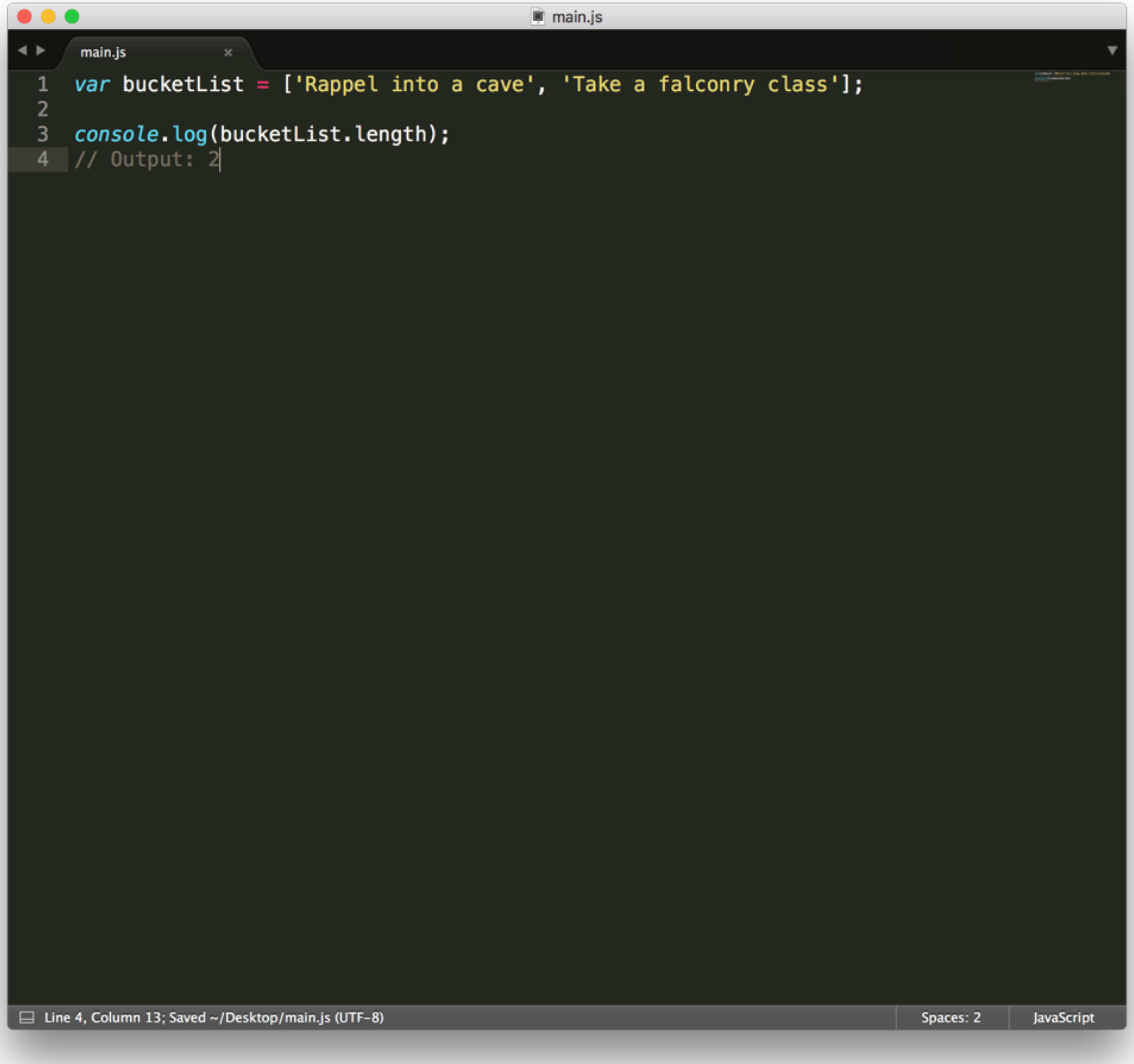


```
var bucketList = ['Rappel into a cave', 'Take a falconry class', 'Learn to jugg
var listItem = bucketList[1];
console.log(listItem);
// Output: 'Take a falconry class'
```

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.

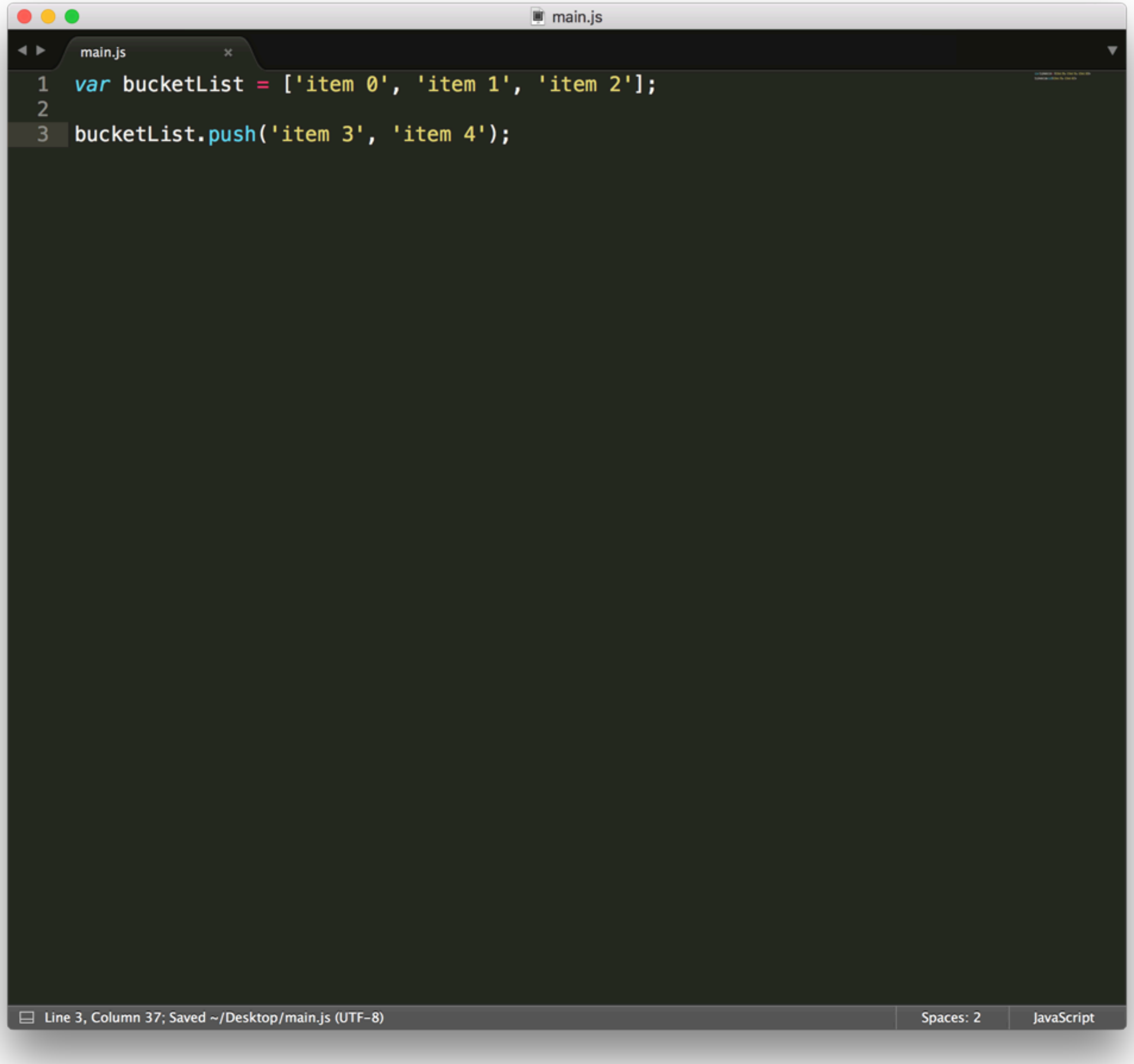


```
var bucketList = ['Rappel into a cave', 'Take a falconry class'];

console.log(bucketList.length);
// Output: 2
```

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.

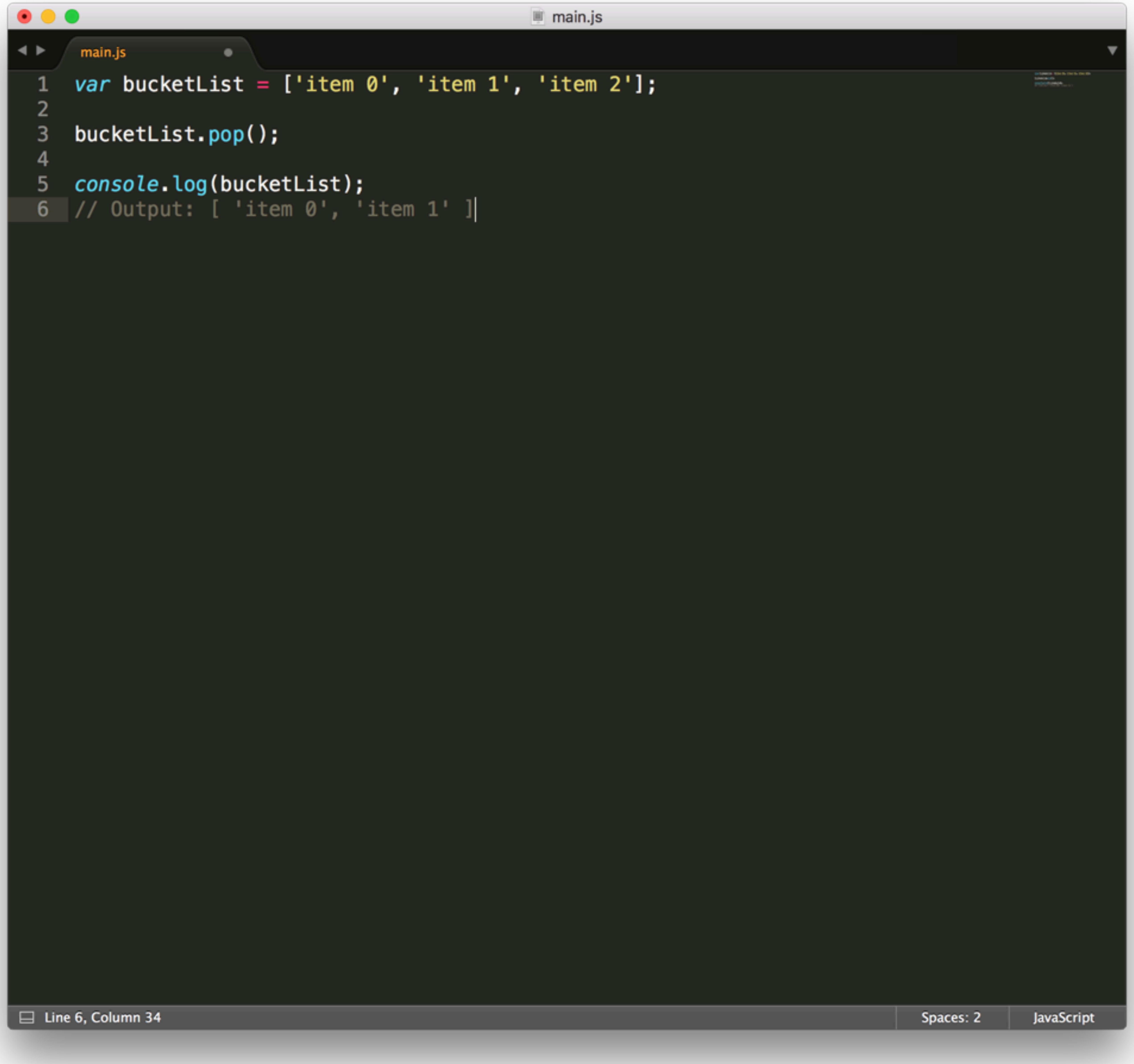


```
var bucketList = ['item 0', 'item 1', 'item 2'];

bucketList.push('item 3', 'item 4');
```

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()



```
var bucketList = ['item 0', 'item 1', 'item 2'];

bucketList.pop();

console.log(bucketList);
// Output: [ 'item 0', 'item 1' ]
```

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.

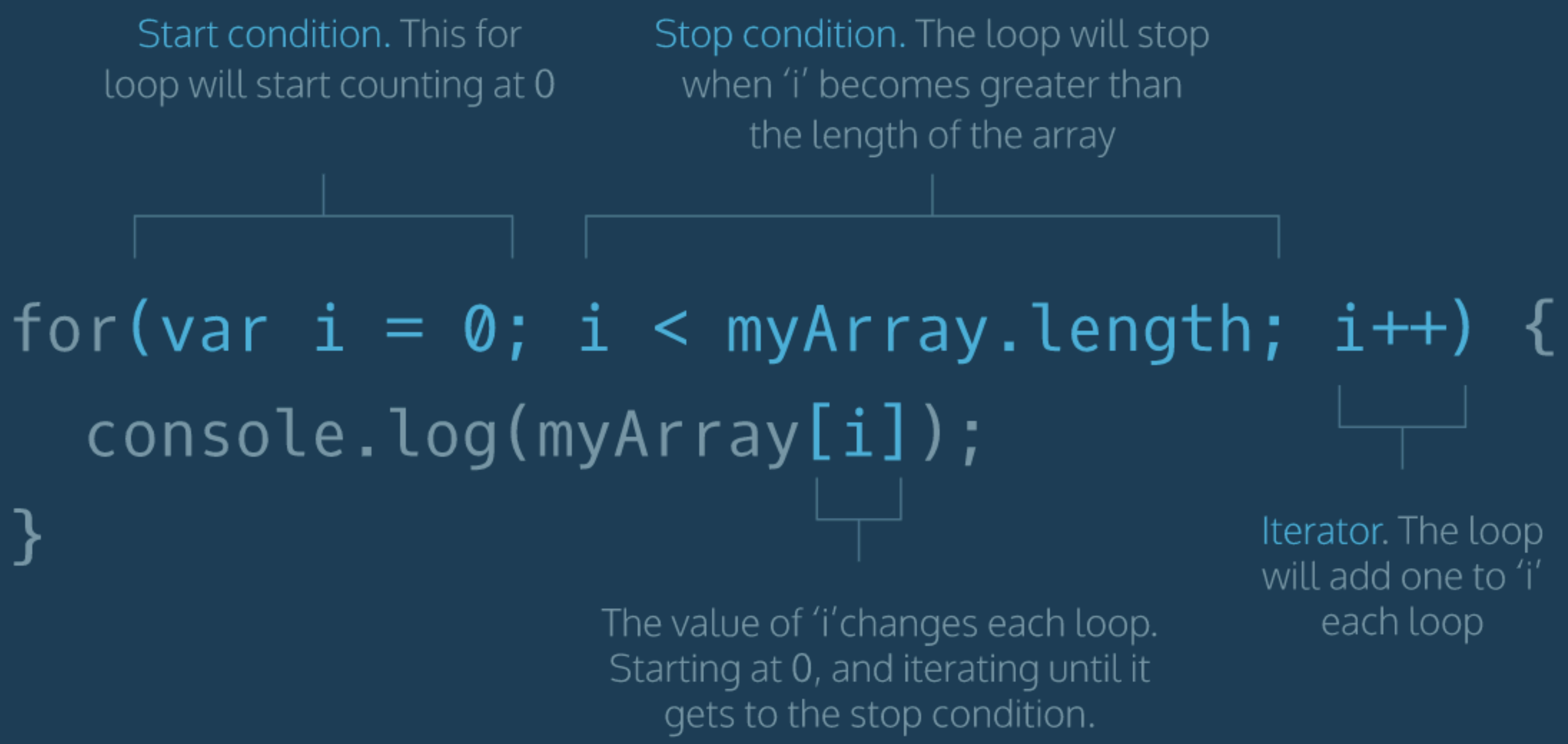
Loops are especially helpful when we have an array where we'd like to do something to each item in the array, like logging each item to the console.

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.



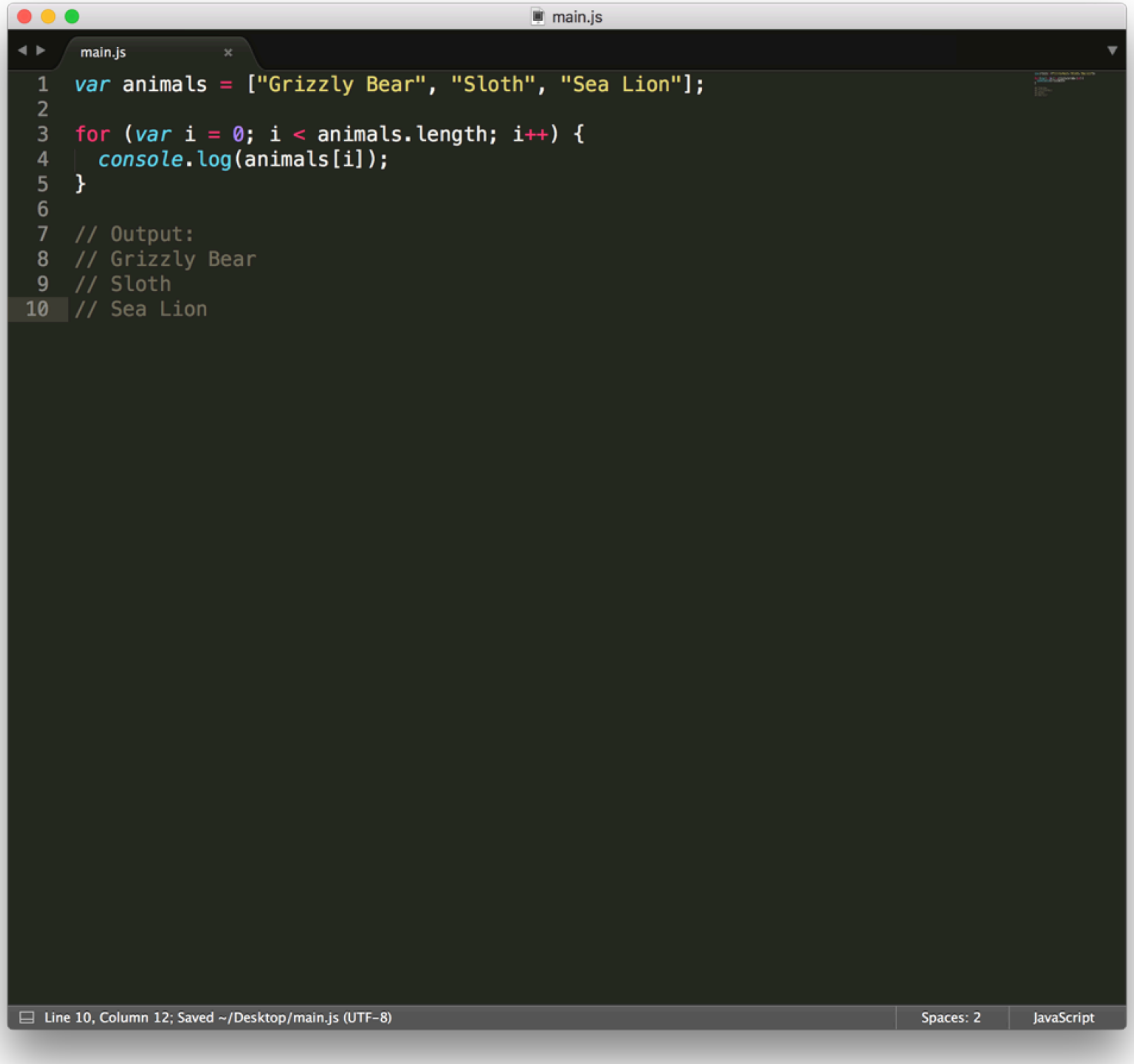
Start condition. This for loop will start counting at 0

Stop condition. The loop will stop when ‘i’ becomes greater than the length of the array

```
for(var i = 0; i < myArray.length; i++) {
  console.log(myArray[i]);
}
```

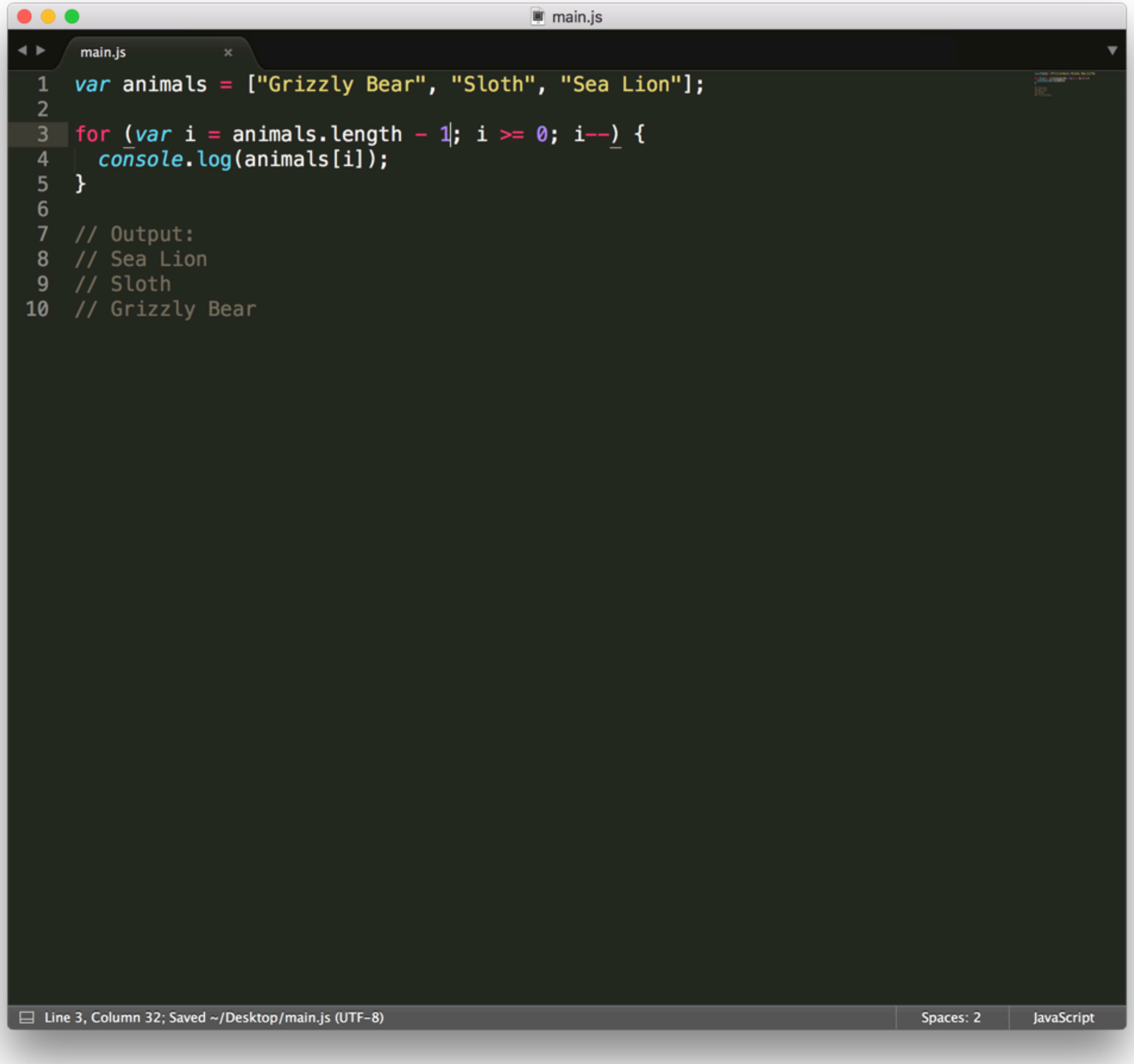
Iterator. The loop will add one to ‘i’ each loop

The value of ‘i’changes each loop. Starting at 0, and iterating until it gets to the stop condition.



```
var animals = ["Grizzly Bear", "Sloth", "Sea Lion"];

for (var i = 0; i < animals.length; i++) {
  console.log(animals[i]);
}

// Output:
// Grizzly Bear
// Sloth
// Sea Lion
```

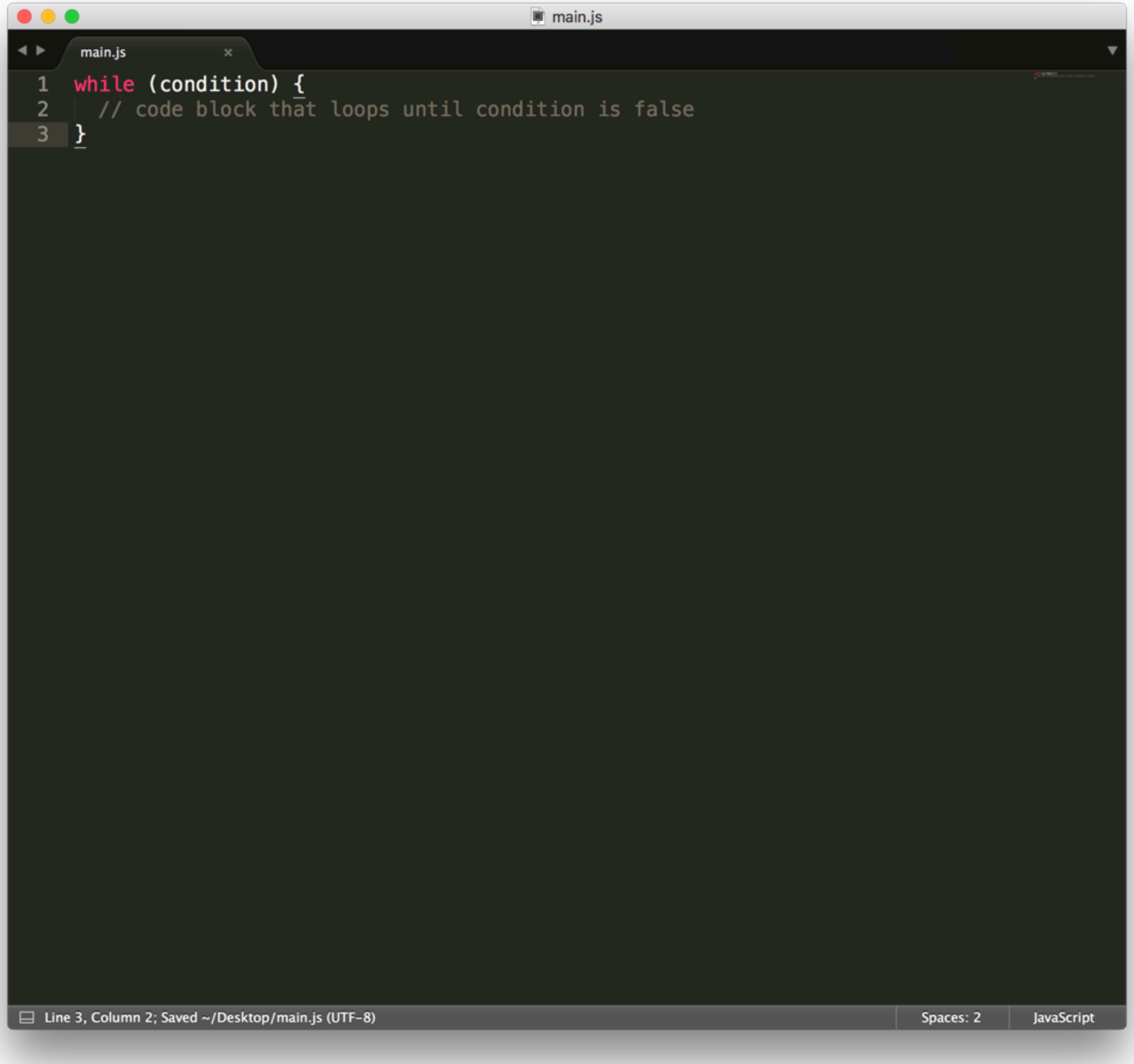
We can make out loop run backwards by modifying the start, stop, and iterator conditions.



```javascript
var animals = ["Grizzly Bear", "Sloth", "Sea Lion"];

for (var i = animals.length - 1; i >= 0; i--) {
  console.log(animals[i]);
}

// Output:
// Sea Lion
// Sloth
// Grizzly Bear
```

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.



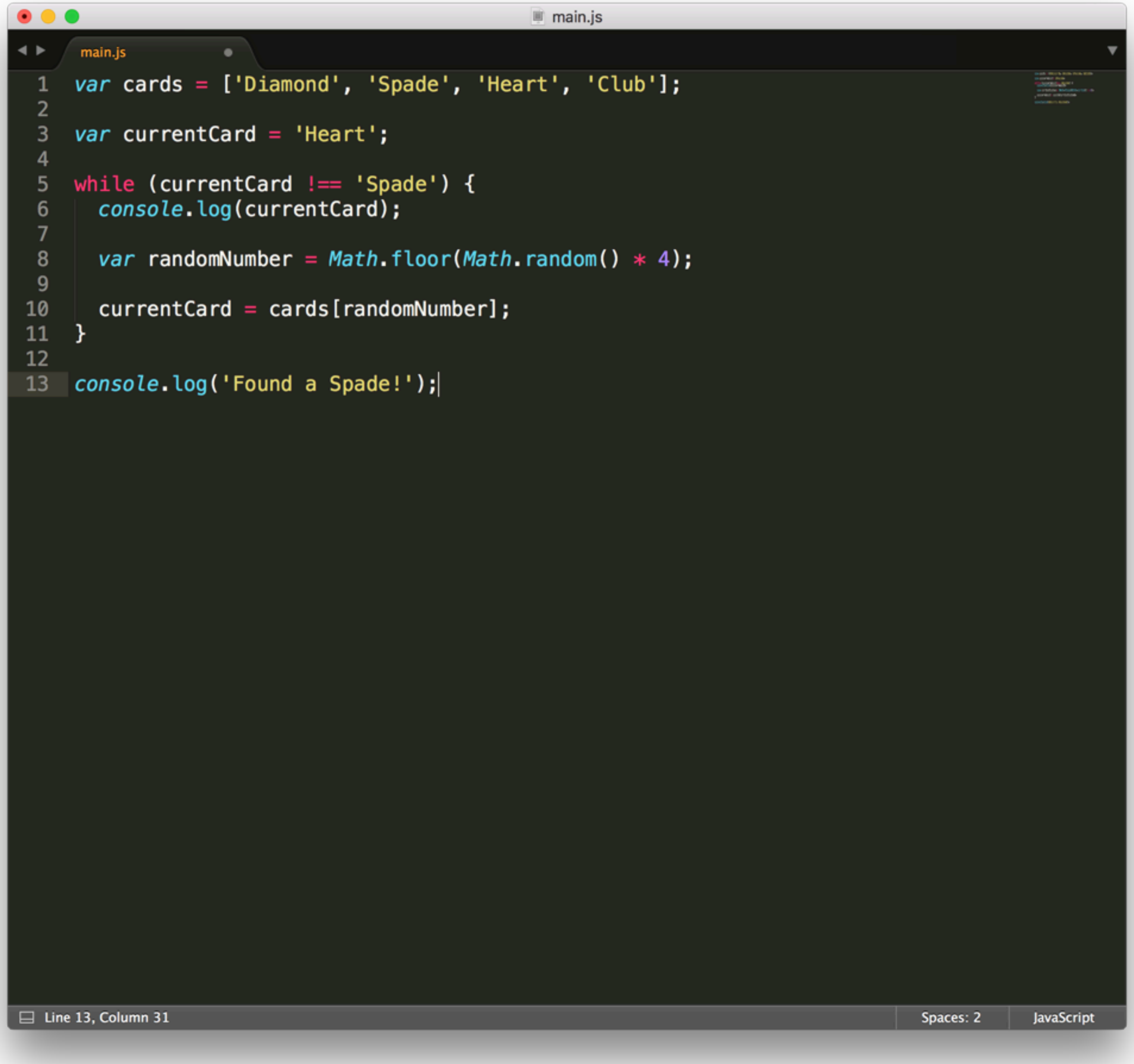
```
while (condition) {
  // code block that loops until condition is false
}
```

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

Spaces: 2

JavaScript

Here’s another example…



```javascript
var cards = ['Diamond', 'Spade', 'Heart', 'Club'];

var currentCard = 'Heart';

while (currentCard !== 'Spade') {
  console.log(currentCard);

  var randomNumber = Math.floor(Math.random() * 4);

  currentCard = cards[randomNumber];
}

console.log('Found a Spade!');
```

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