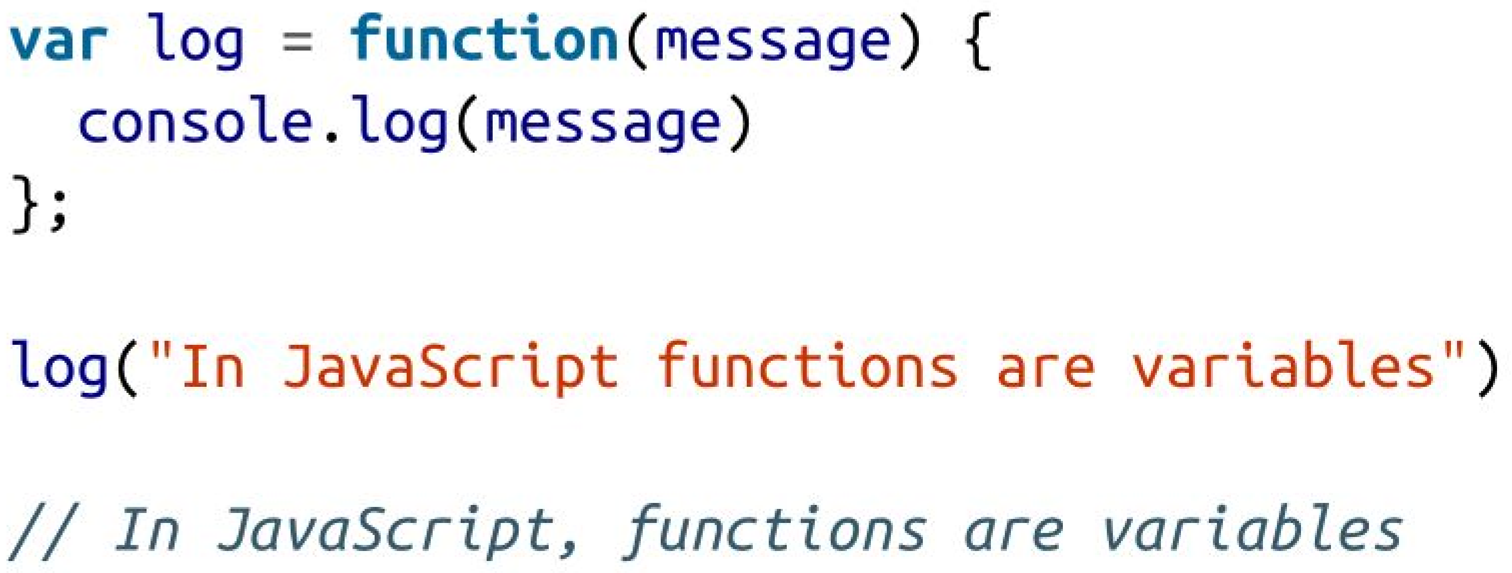
SESSION 1 - FOUNDATION

Part 3 - Javascript Basics

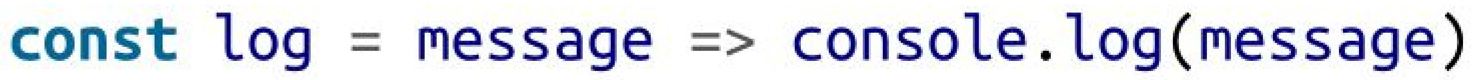
Part 3 - Functional Javascript
First Class Functions
Higher Order Functions
Currying
Declarative vs Imperative
Immutability
Pure Functions
Data Transformations - Map, Reduce, Filter
Composition
This, Call, Apply, Bind
Closure
Callback, Promises, async/await
Mixins

First Class Functions

This means that functions can do the same things that variables can do.



```
var log = function(message) {
  console.log(message)
};

log("In JavaScript functions are variables")

// In JavaScript, functions are variables
```



```
const log = message => console.log(message)
```

Since functions are variables, we can add them to objects:

```
const obj = {
    message: "They can be added to objects like variables",
    log(message) {
        console.log(message)
    }
}

obj.log(obj.message)

// They can be added to objects like variables
```

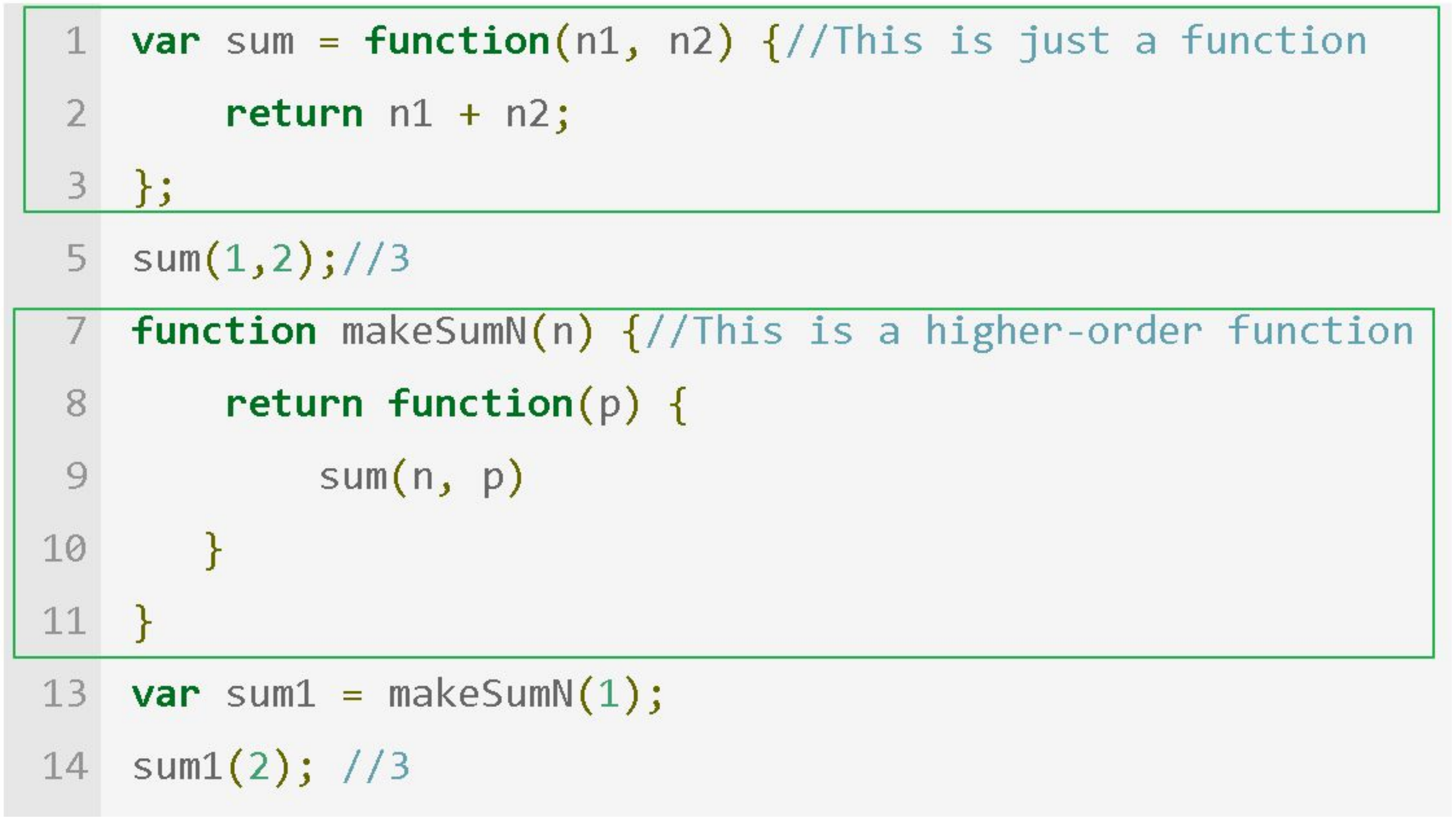
We can also add functions to arrays in JavaScript:

```
const messages = [
    "They can be inserted into arrays",
    message => console.log(message),
    "like variables",
    message => console.log(message)
]

messages[1](messages[0])   // They can be inserted into arrays
messages[3](messages[2])   // like variables
```

Higher Order Functions

Higher-Order Functions

```
function doTwice(action) {
  action();
  action();
}

doTwice(function() {
  console.log('called!');
})
```



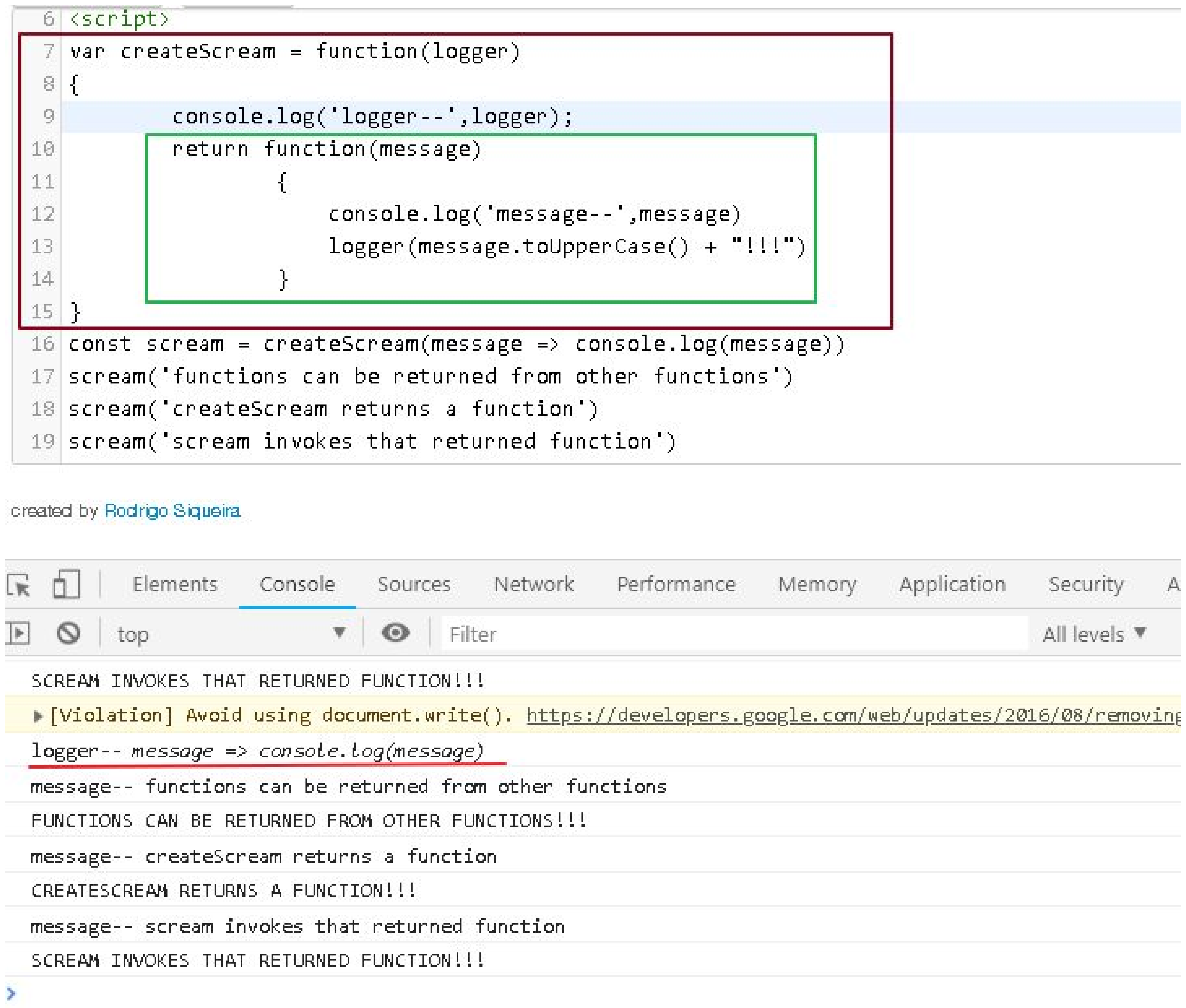
```javascript
var sum = function(n1, n2) {//This is just a function
    return n1 + n2;
};
sum(1,2);//3
function makeSumN(n) {//This is a higher-order function
    return function(p) {
        sum(n, p)
    }
}
var sum1 = makeSumN(1);
sum1(2); //3
```

Functions can be sent to other functions as arguments, just like other variables

```
const insideFn = logger =>
  logger("They can be sent to other functions as arguments");


insideFn(message => console.log(message))

// They can be sent to other functions as arguments
```

They can also be returned from other functions, just like variables:

```
var createScream = function(logger) {
    return function(message) {
        logger(message.toUpperCase() + "!!!")
    }
}

const scream = createScream(message => console.log(message))

scream('functions can be returned from other functions')
scream('createScream returns a function')
scream('scream invokes that returned function')

// FUNCTIONS CAN BE RETURNED FROM OTHER FUNCTIONS!!!
// CREATESCREAM RETURNS A FUNCTION!!!
// SCREAM INVOKES THAT RETURNED FUNCTION!!!
```



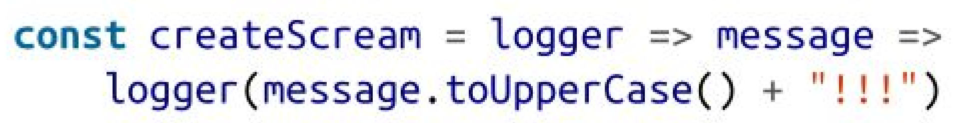
```
<script>
var createScream = function(logger)
{
        console.log('logger--',logger);
        return function(message)
                {
                    console.log('message--',message)
                    logger(message.toUpperCase() + "!!!")
                }
}
const scream = createScream(message => console.log(message))
scream('functions can be returned from other functions')
scream('createScream returns a function')
scream('scream invokes that returned function')
```

created by Rodrigo Siqueira

```
SCREAM INVOKES THAT RETURNED FUNCTION!!!
▸[Violation] Avoid using document.write(). https://developers.google.com/web/updates/2016/08/removing
logger-- message => console.log(message)
message-- functions can be returned from other functions
FUNCTIONS CAN BE RETURNED FROM OTHER FUNCTIONS!!!
message-- createScream returns a function
CREATESCREAM RETURNS A FUNCTION!!!
message-- scream invokes that returned function
SCREAM INVOKES THAT RETURNED FUNCTION!!!
>
```

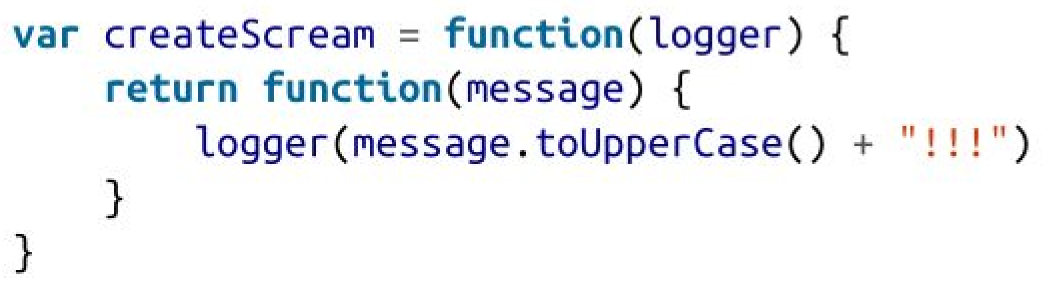
The last example was of higher-order functions, functions that either take or return other functions.

Using ES6 syntax, we could describe the same createScream higher-order function with arrows:



```
const createScream = logger => message =>
    logger(message.toUpperCase() + "!!!")
```

The same as above functions



```
var createScream = function(logger) {
    return function(message) {
        logger(message.toUpperCase() + "!!!")
    }
}
```

From here on out, we need to pay attention to the number of arrows used during function declaration. More than one arrow means that we have a higher-order function. We can say that JavaScript is a functional language because its functions are first-class citizens. This means that functions are data. They can be saved, retrieved, or flow through your applications just like variables.

```
const invokeIf = (condition, fnTrue, fnFalse) =>
    (condition) ? fnTrue() : fnFalse()

const showWelcome = () =>
    console.log("Welcome!!!")

const showUnauthorized = () =>
    console.log("Unauthorized!!!")

invokeIf(true, showWelcome, showUnauthorized)    // "Welcome"
invokeIf(false, showWelcome, showUnauthorized)   // "Unauthorized"
```

Currying

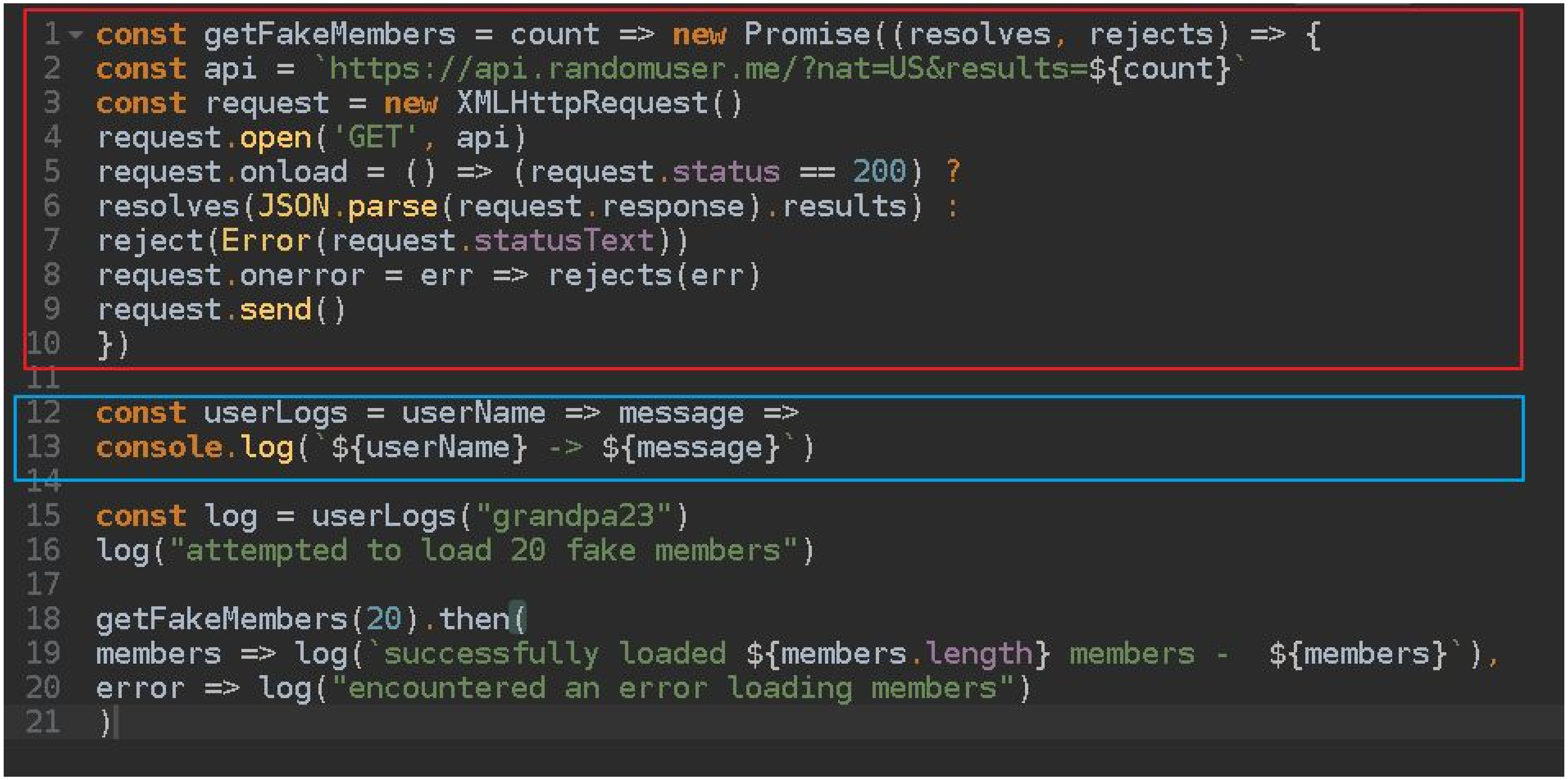
Currying is a technique of evaluating function with *multiple arguments*, into sequence of function with single argument.

```
function add(a,b,c){
 return a + b + c;
}
```

You can call it with too few (with odd results), or too many (excess arguments get ignored).

```
add(1,2,3) --> 6
add(1,2) --> NaN
add(1,2,3,4) --> 6 //Extra parameters will be ignored.
```

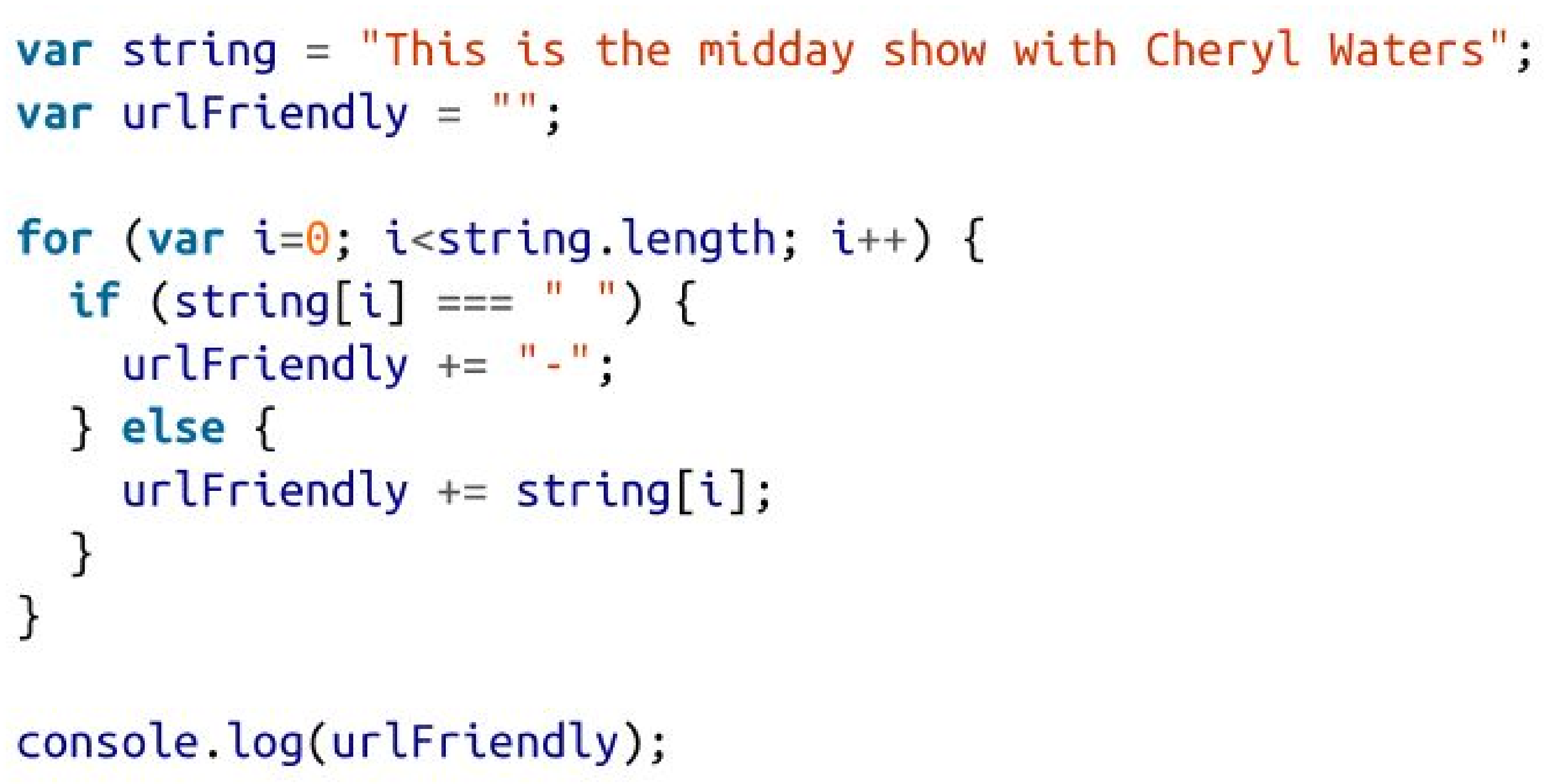
```
//import or load lodash

var abc = function(a, b, c) {
  return a + b + c;
};

var curried = _.curry(abc);

var addBy2 = curried(2);

console.log(addBy2(0,0));
// => 2

console.log(addBy2(1,1));
// => 4

console.log(curried(4)(5)(6));
// => 15
```



```javascript
const getFakeMembers = count => new Promise((resolves, rejects) => {
const api = `https://api.randomuser.me/?nat=US&results=${count}`
const request = new XMLHttpRequest()
request.open('GET', api)
request.onload = () => (request.status == 200) ?
resolves(JSON.parse(request.response).results) :
reject(Error(request.statusText))
request.onerror = err => rejects(err)
request.send()
})

const userLogs = userName => message =>
console.log(`${userName} -> ${message}`)

const log = userLogs("grandpa23")
log("attempted to load 20 fake members")

getFakeMembers(20).then(
members => log(`successfully loaded ${members.length} members -  ${members}`),
error => log("encountered an error loading members")
)
```

Declarative vs Imperative

Imperative programming is only concerned with how to achieve results with code.



```
var string = "This is the midday show with Cheryl Waters";
var urlFriendly = "";

for (var i=0; i<string.length; i++) {
  if (string[i] === " ") {
    urlFriendly += "-";
  } else {
    urlFriendly += string[i];
  }
}

console.log(urlFriendly);
```

In a declarative program, the syntax itself describes what should happen and the details of how things happen are abstracted away.

```
const string = "This is the mid day show with Cheryl Waters"
const urlFriendly = string.replace(/ /g, "-")

console.log(urlFriendly)
```

Now, let's consider the task of building a document object model, or DOM. An imperative approach would be concerned with how the DOM is constructed:

```
var target = document.getElementById('target');
var wrapper = document.createElement('div');
var headline = document.createElement('h1');

wrapper.id = "welcome";
headline.innerText = "Hello World";

wrapper.appendChild(headline);
target.appendChild(wrapper);
```

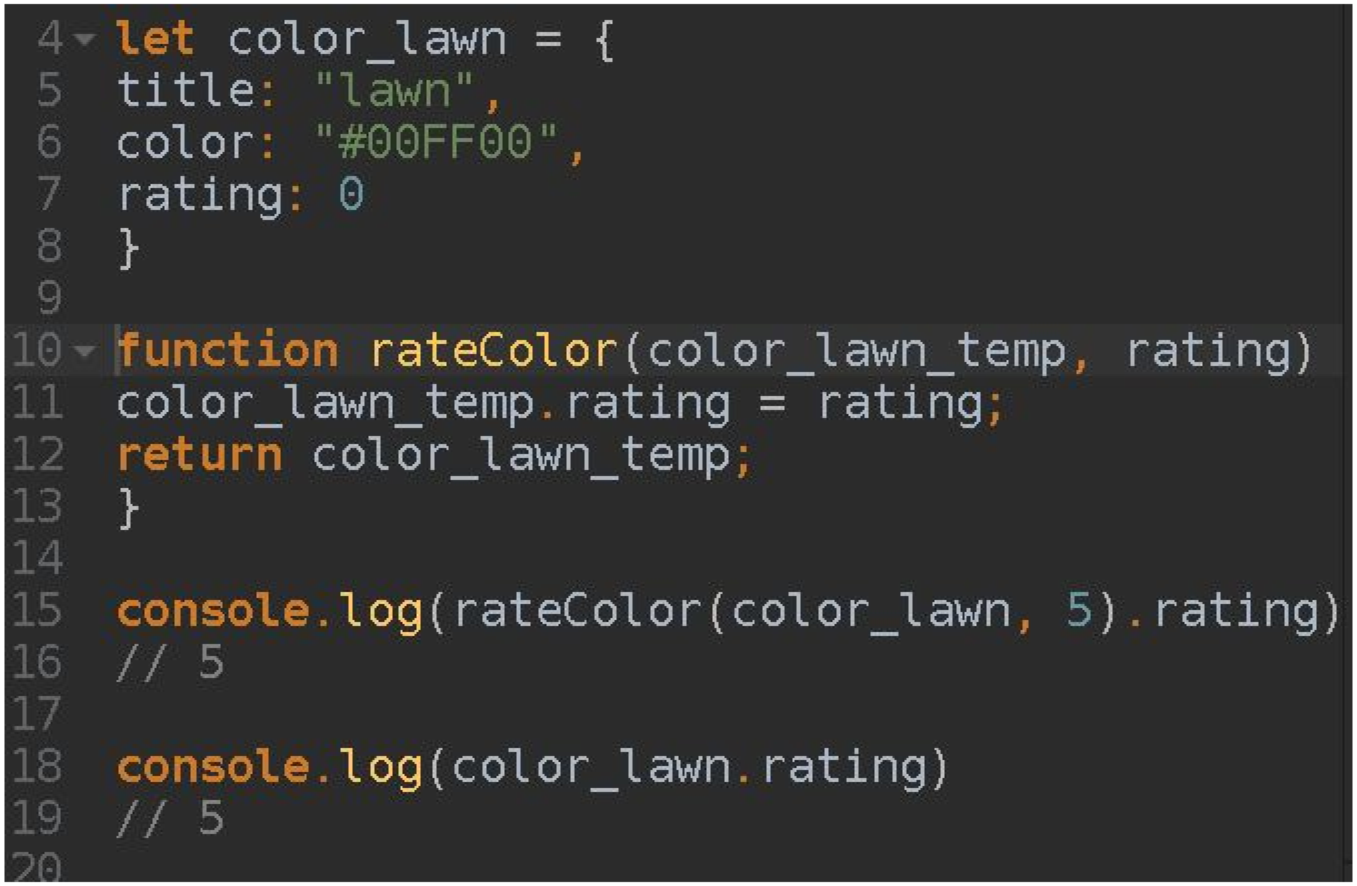
```
const { render } = ReactDOM

const Welcome = () => (
    <div id="welcome">
        <h1>Hello World</h1>
    </div>
)

render(
    <Welcome />,
    document.getElementById('target')
)
```

React is declarative

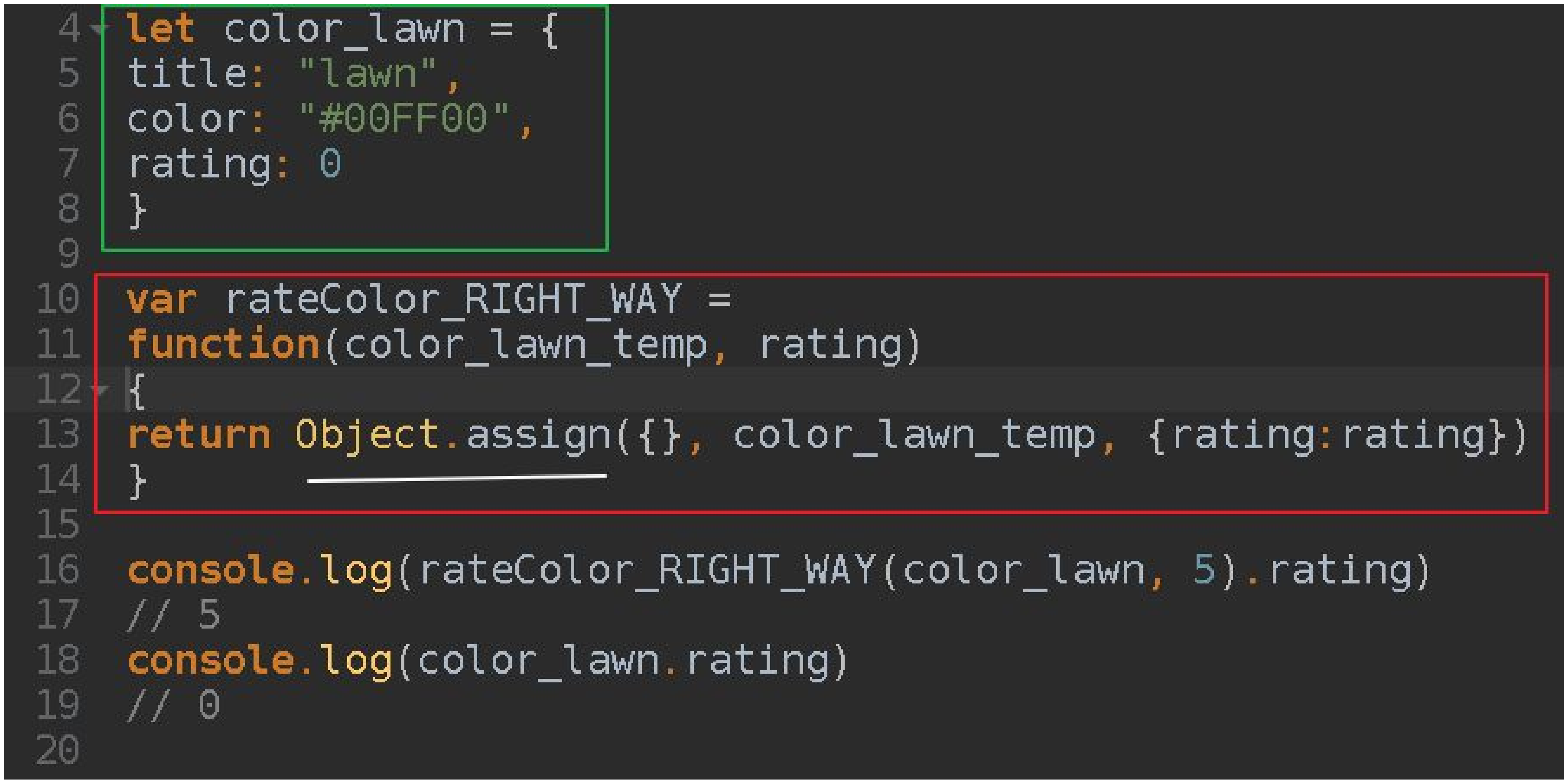
IMPERATIVE	DECLARATIVE
is a programming paradigm that uses statements that change a program's state.	which focuses on what the program should accomplish without specifying how the program should achieve the result.
Go to kitchen Open fridge Remove chicken from fridge … Bring food to table	I want a dinner with chicken.

Immutability

immutable is to be unchangeable. In a functional program, data is immutable. It never changes.



```javascript
let color_lawn = {
title: "lawn",
color: "#00FF00",
rating: 0
}

function rateColor(color_lawn_temp, rating)
color_lawn_temp.rating = rating;
return color_lawn_temp;
}

console.log(rateColor(color_lawn, 5).rating)
// 5

console.log(color_lawn.rating)
// 5
```



```
let color_lawn = {
title: "lawn",
color: "#00FF00",
rating: 0
}

var rateColor_RIGHT_WAY =
function(color_lawn_temp, rating)
{
return Object.assign({}, color_lawn_temp, {rating:rating})
}

console.log(rateColor_RIGHT_WAY(color_lawn, 5).rating)
// 5
console.log(color_lawn.rating)
// 0

```

We can write the same function using an ES6 arrow function along with the ES7 object spread operator. This `rateColor` function uses the spread operator to copy the color into a new object and then overwrite its rating:

```
const rateColor = (color, rating) =>
    ({
        ...color,
        rating
    })
```

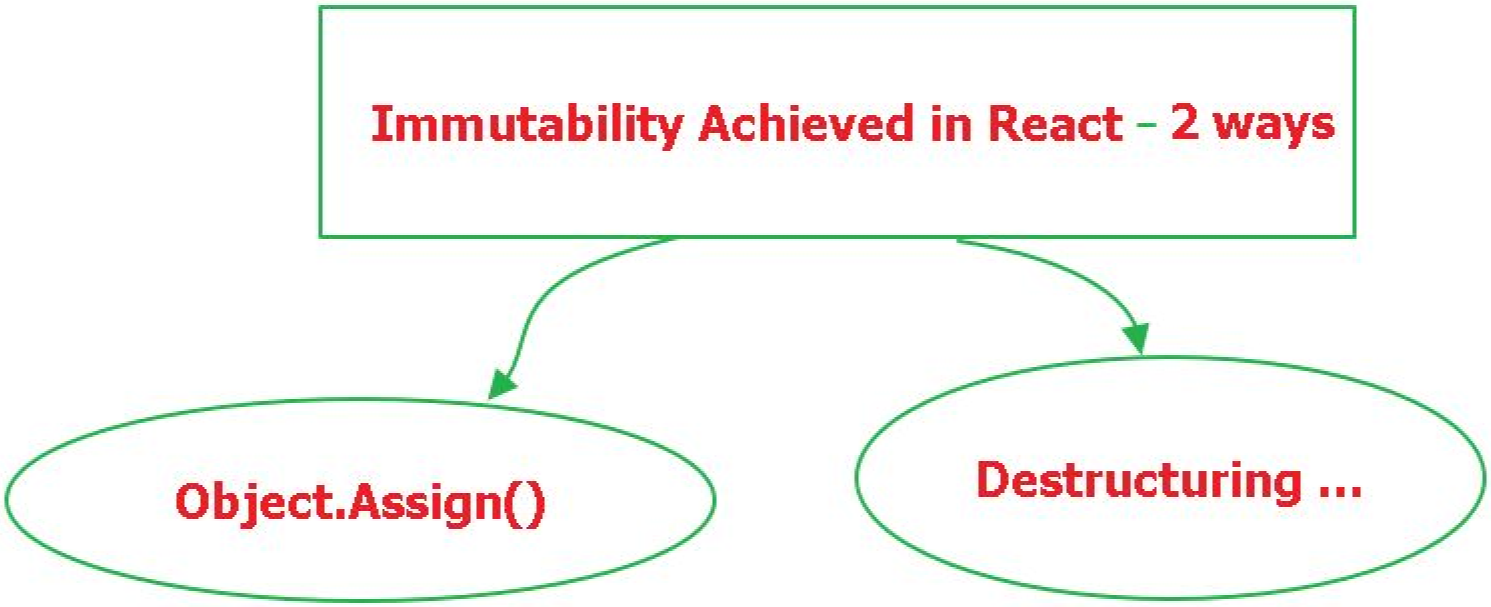
```
let list = [
    { title: "Rad Red"},
    { title: "Lawn"},
    { title: "Party Pink"}
]
```

We could create a function that will add colors to that array using `Array.push`:

```
var addColor = function(title, colors) {
  colors.push({ title: title })
  return colors;
}

console.log(addColor("Glam Green", list).length)        // 4
console.log(list.length)                                 // 4
```

```
const addColor = (title, array) => array.concat({title})

console.log(addColor("Glam Green", list).length)        // 4
console.log(list.length)                                 // 3
```

```
const addColor = (title, list) => [...list, {title}]
```


Immutability Achieved in React - 2 ways
Object.Assign()
Destructuring ...

Pure Functions

```
function sum(x,y) {
  return x + y ;
}
sum(10, 20) // returns 30
sum(10, 20) // returns 30
sum(10, 20) // returns 30
```

pure

impure

```
var count = 10;
function increaseCount(value) {
  count += value;
}
```

```
var frederick = {
    name: "Frederick Douglass",
    canRead: false,
    canWrite: false
}
```

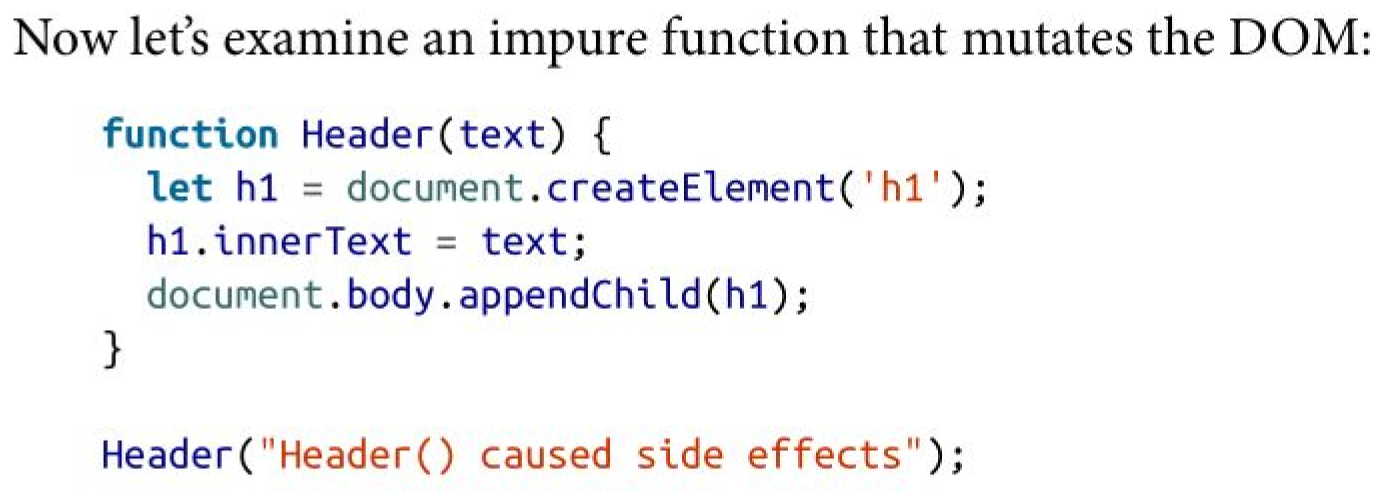
```
function selfEducate() {
    frederick.canRead = true
    frederick.canWrite = true
    return frederick
}
```

```
selfEducate()
console.log( frederick )

// {name: "Frederick Douglass", canRead: true, canWrite: true}
```

A *pure function* is a function that returns a value that is computed based on its arguments. Pure functions take at least one argument and always return a value or another function. They do not cause side effects, set global variables, or change anything about application state. They treat their arguments as immutable data.

```
const frederick = {
    name: "Frederick Douglass",
    canRead: false,
    canWrite: false
}

const selfEducate = person =>
    ({
        ...person,
        canRead: true,
        canWrite: true
    })

console.log( selfEducate(frederick) )
console.log( frederick )

// {name: "Frederick Douglass", canRead: true, canWrite: true}
// {name: "Frederick Douglass", canRead: false, canWrite: false}
```

1. The function should take in at least one argument.
2. The function should return a value or another function.
3. The function should not change or mutate any of its arguments.



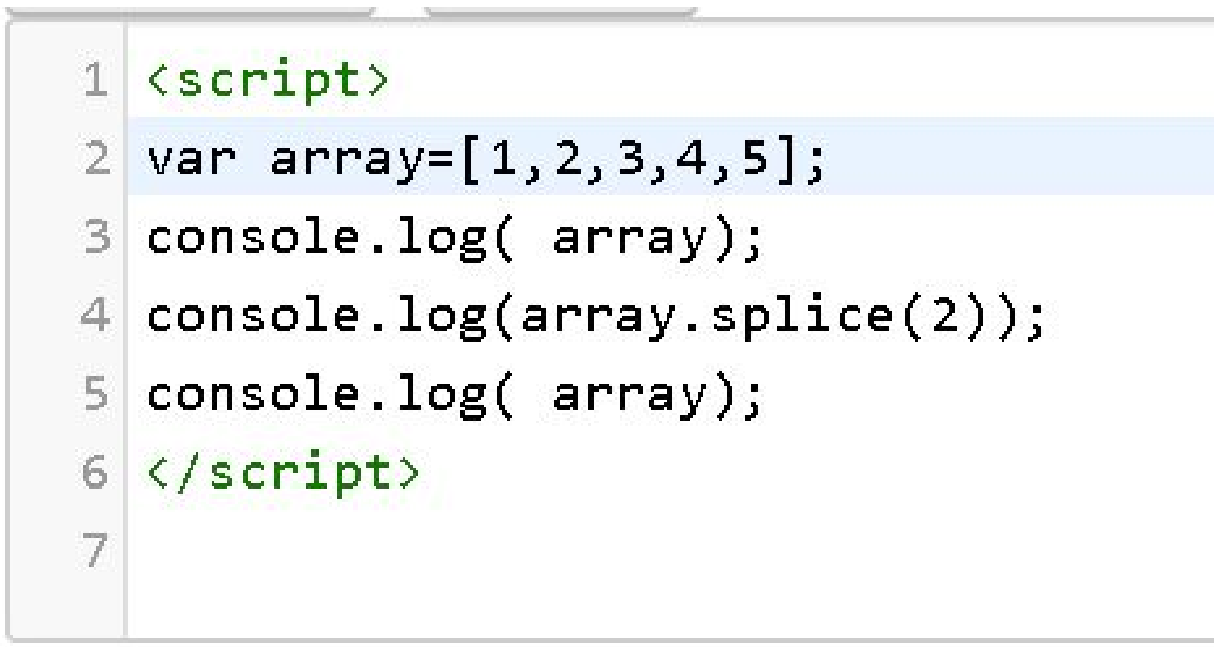
Now let's examine an impure function that mutates the DOM:

```
function Header(text) {
  let h1 = document.createElement('h1');
  h1.innerText = text;
  document.body.appendChild(h1);
}

Header("Header() caused side effects");
```

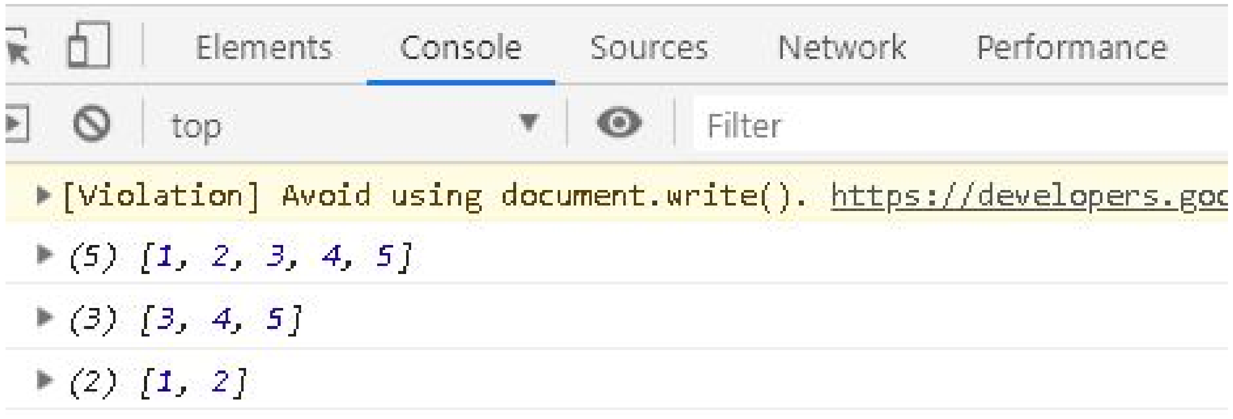
In React, the UI is expressed with pure functions. In the following sample, `Header` is a pure function that can be used to create heading—one elements just like in the previous example. However, this function on its own does not cause side effects because it does not mutate the DOM. This function will create a heading-one element, and it is up to some other part of the application to use that element to change the DOM:

```
const Header = (props) => <h1>{props.title}</h1>
```

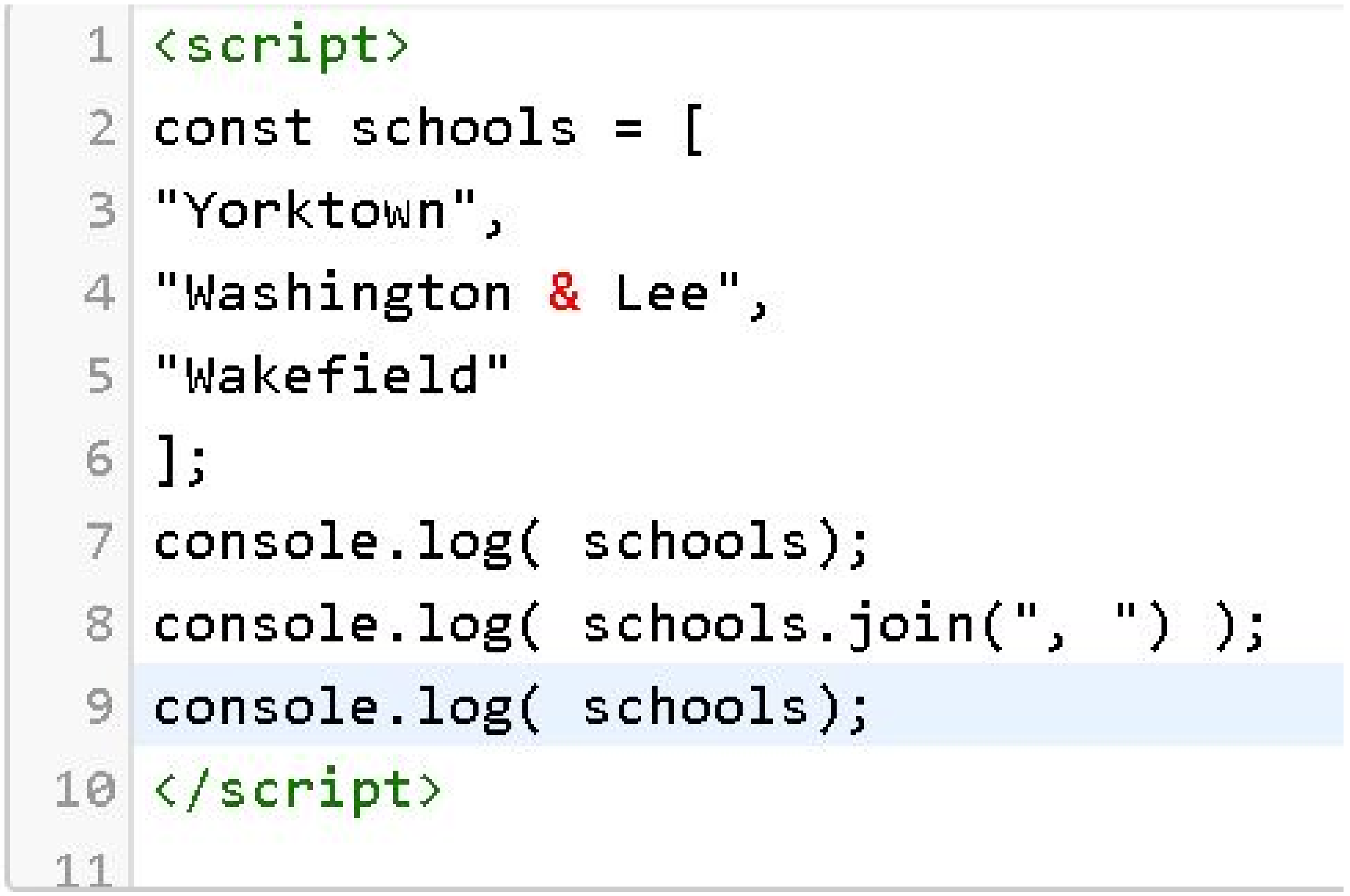
Data Transformations - Map, Reduce, Filter



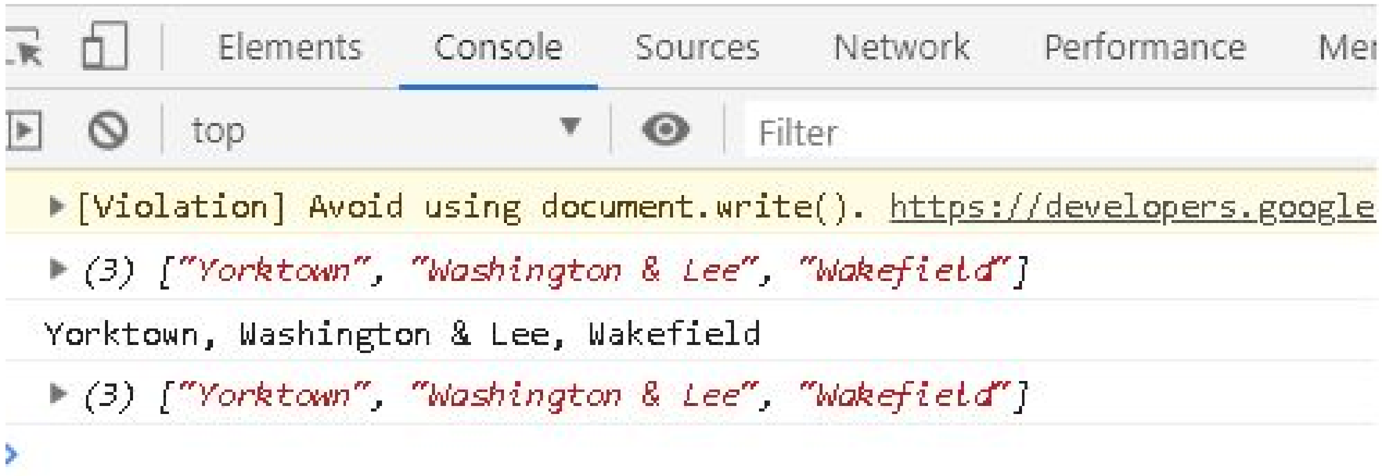
```
<script>
var array=[1,2,3,4,5];
console.log( array);
console.log(array.splice(2));
console.log( array);
</script>
```



```
▶ [Violation] Avoid using document.write(). https://developers.goo
▶ (5) [1, 2, 3, 4, 5]
▶ (3) [3, 4, 5]
▶ (2) [1, 2]
```



```
<script>
const schools = [
"Yorktown",
"Washington & Lee",
"Wakefield"
];
console.log( schools);
console.log( schools.join(", ") );
console.log( schools);
</script>
```



```
▶ [Violation] Avoid using document.write(). https://developers.google
▶ (3) ["Yorktown", "Washington & Lee", "Wakefield"]
Yorktown, Washington & Lee, Wakefield
▶ (3) ["Yorktown", "Washington & Lee", "Wakefield"]
```

Consider this array of high schools:

```
const schools = [
  "Yorktown",
  "Washington & Lee",
  "Wakefield"
]
```

```
const wSchools = schools.filter(school => school[0] === "W")
```

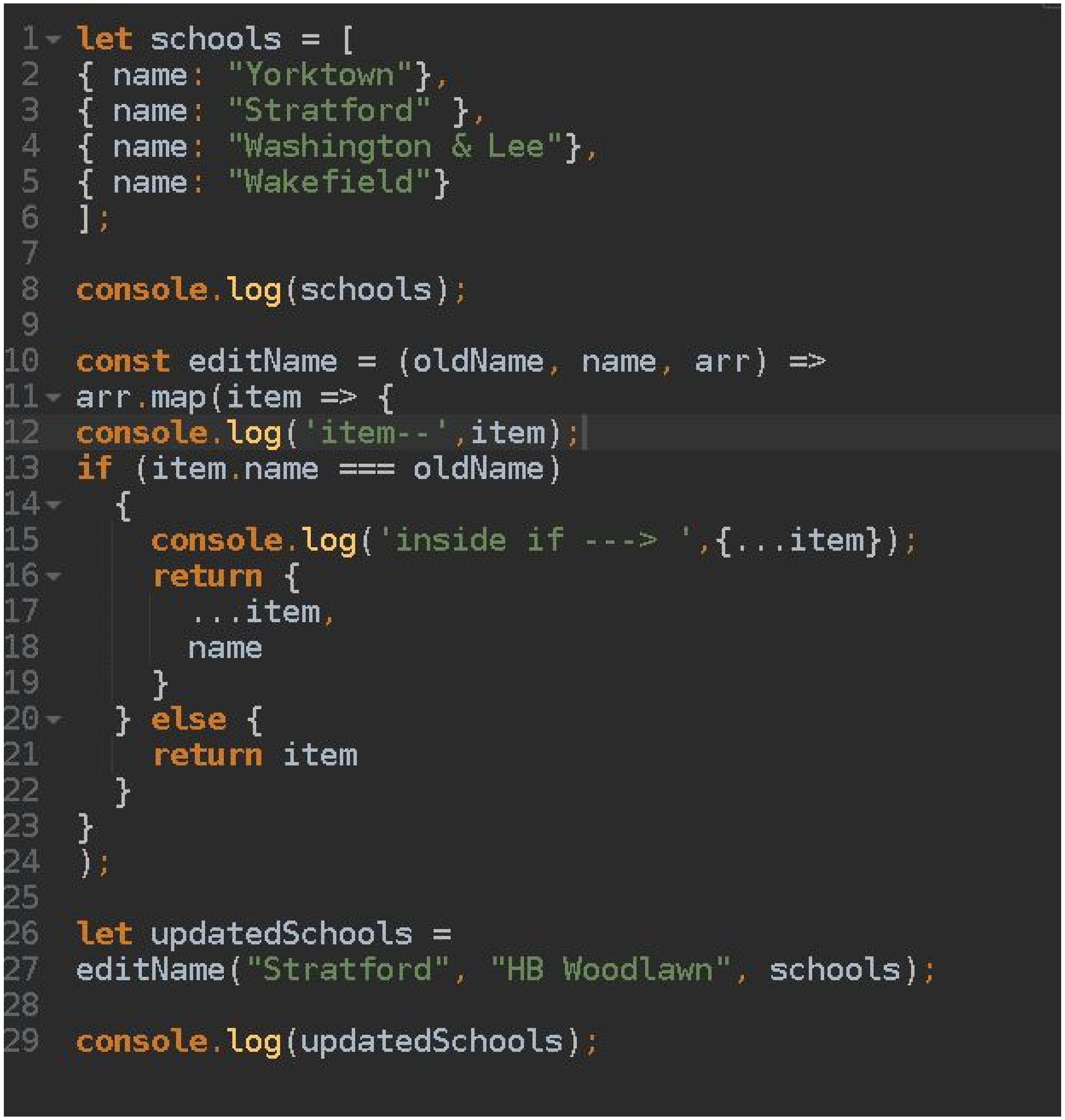
```
console.log( wSchools )
// ["Washington & Lee", "Wakefield"]
```

```javascript
const schools = [
  "Yorktown",
  "Washington & Lee",
  "Wakefield"
]


  const highSchools = schools.map(school => `${school} High School`)

  console.log(highSchools.join("\n"))

  //  Yorktown High School
  //  Washington & Lee High School
  //  Wakefield High School

  console.log(schools.join("\n"))

  //  Yorktown
  //  Washington & Lee
  //  Wakefield
```

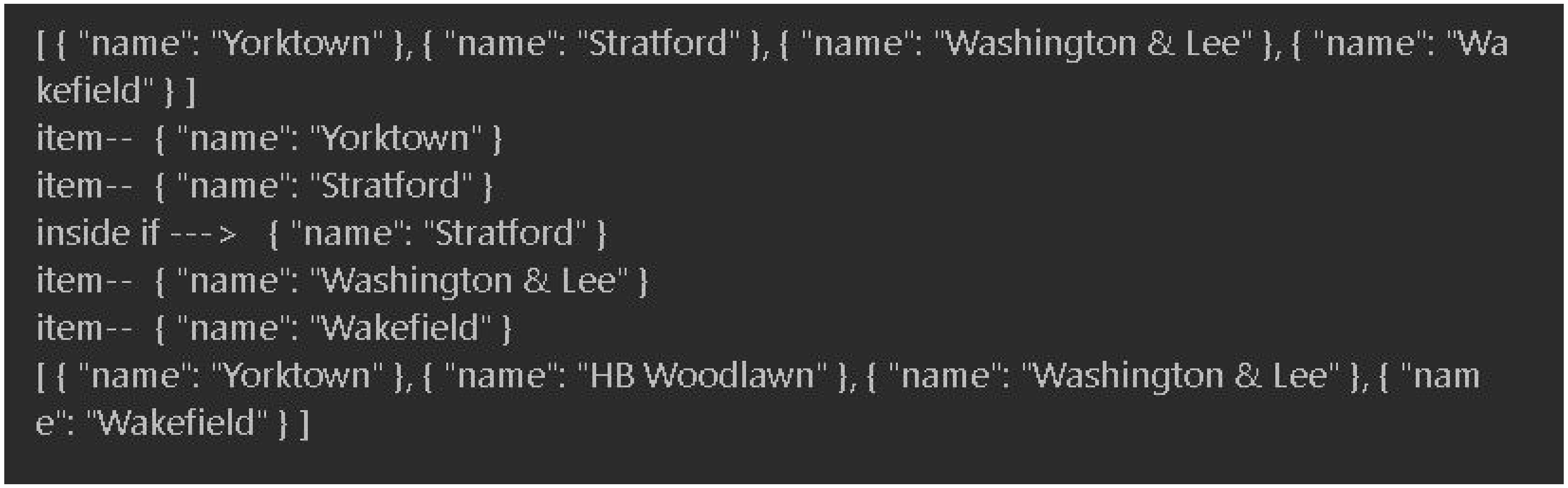
```javascript
const schools = [
  "Yorktown",
  "Washington & Lee",
  "Wakefield"
]
```

```javascript
const highSchools = schools.map(school => ({ name: school }))

console.log( highSchools )

// [
//   { name: "Yorktown" },
//   { name: "Washington & Lee" },
//   { name: "Wakefield" }
// ]
```

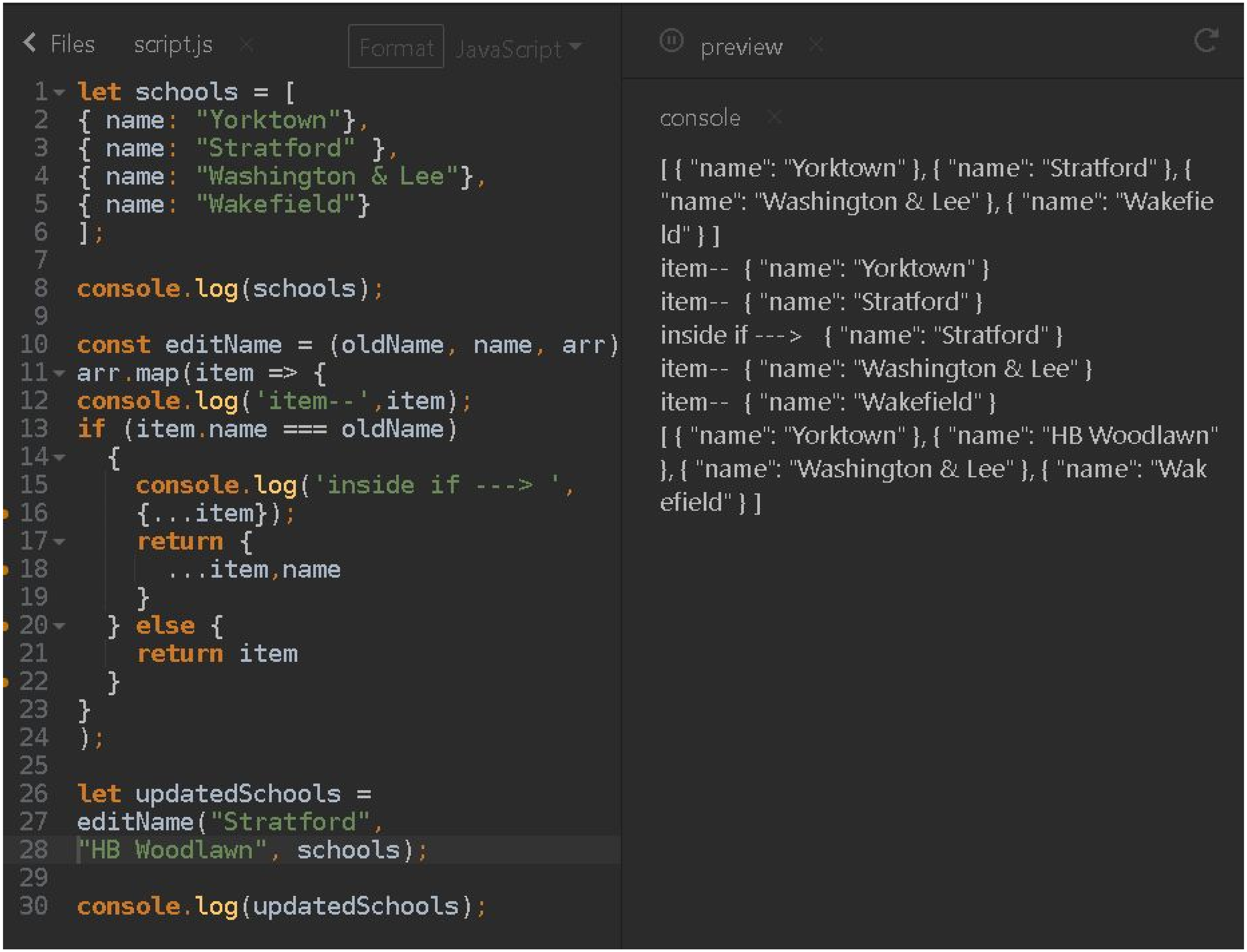
```javascript
const schools = [
  "Yorktown",
  "Washington & Lee",
  "Wakefield"
]
```



```
let schools = [
{ name: "Yorktown"},
{ name: "Stratford" },
{ name: "Washington & Lee"},
{ name: "Wakefield"}
];

console.log(schools);

const editName = (oldName, name, arr) =>
arr.map(item => {
console.log('item--',item);
if (item.name === oldName)
  {
    console.log('inside if ---> ',{...item});
    return {
      ...item,
      name
    }
  } else {
    return item
  }
}
);

let updatedSchools =
editName("Stratford", "HB Woodlawn", schools);

console.log(updatedSchools);
```



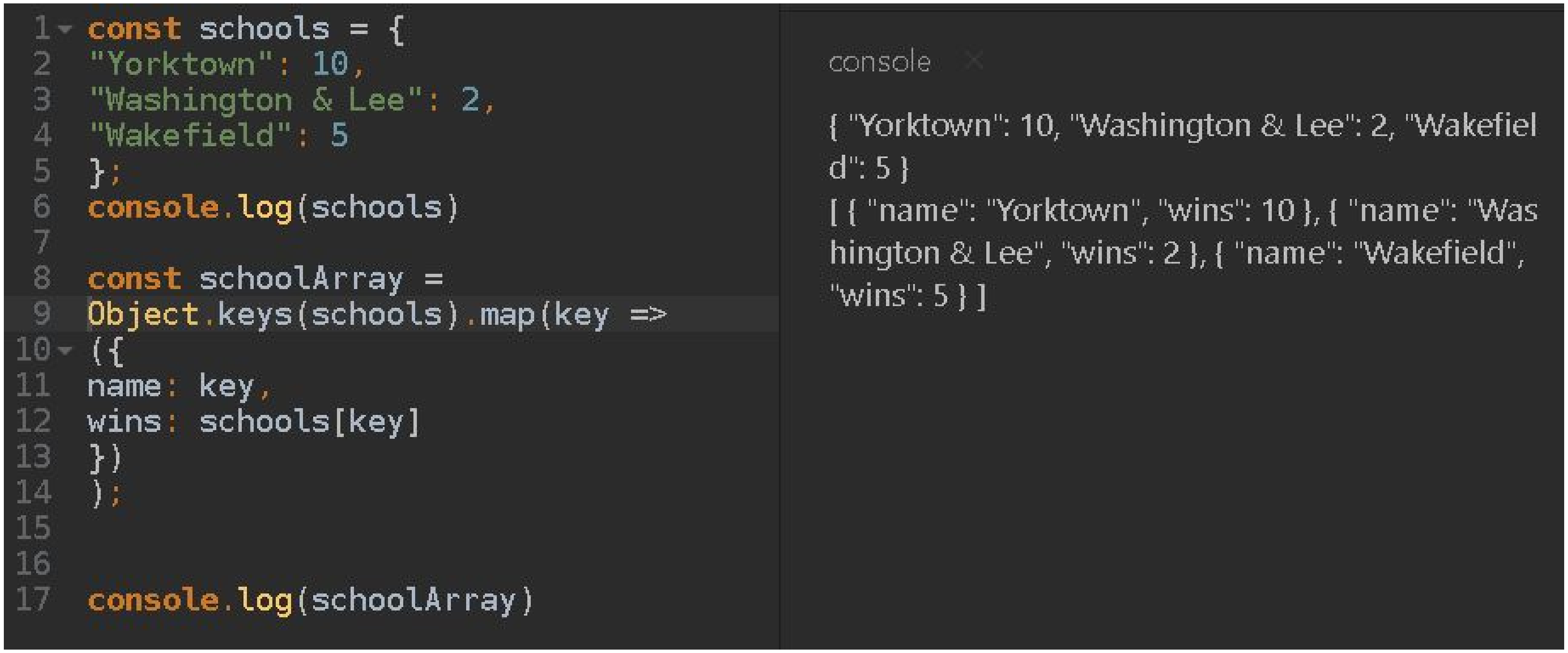
```
[ { "name": "Yorktown" }, { "name": "Stratford" }, { "name": "Washington & Lee" }, { "name": "Wa
kefield" } ]
item--  { "name": "Yorktown" }
item--  { "name": "Stratford" }
inside if --->   { "name": "Stratford" }
item--  { "name": "Washington & Lee" }
item--  { "name": "Wakefield" }
[ { "name": "Yorktown" }, { "name": "HB Woodlawn" }, { "name": "Washington & Lee" }, { "nam
e": "Wakefield" } ]
```



```javascript
let schools = [
{ name: "Yorktown"},
{ name: "Stratford" },
{ name: "Washington & Lee"},
{ name: "Wakefield"}
];

console.log(schools);

const editName = (oldName, name, arr)
arr.map(item => {
console.log('item--',item);
if (item.name === oldName)
  {
    console.log('inside if ---> ',
    {...item});
    return {
      ...item,name
    }
  } else {
    return item
  }
}
);

let updatedSchools =
editName("Stratford",
"HB Woodlawn", schools);

console.log(updatedSchools);
```

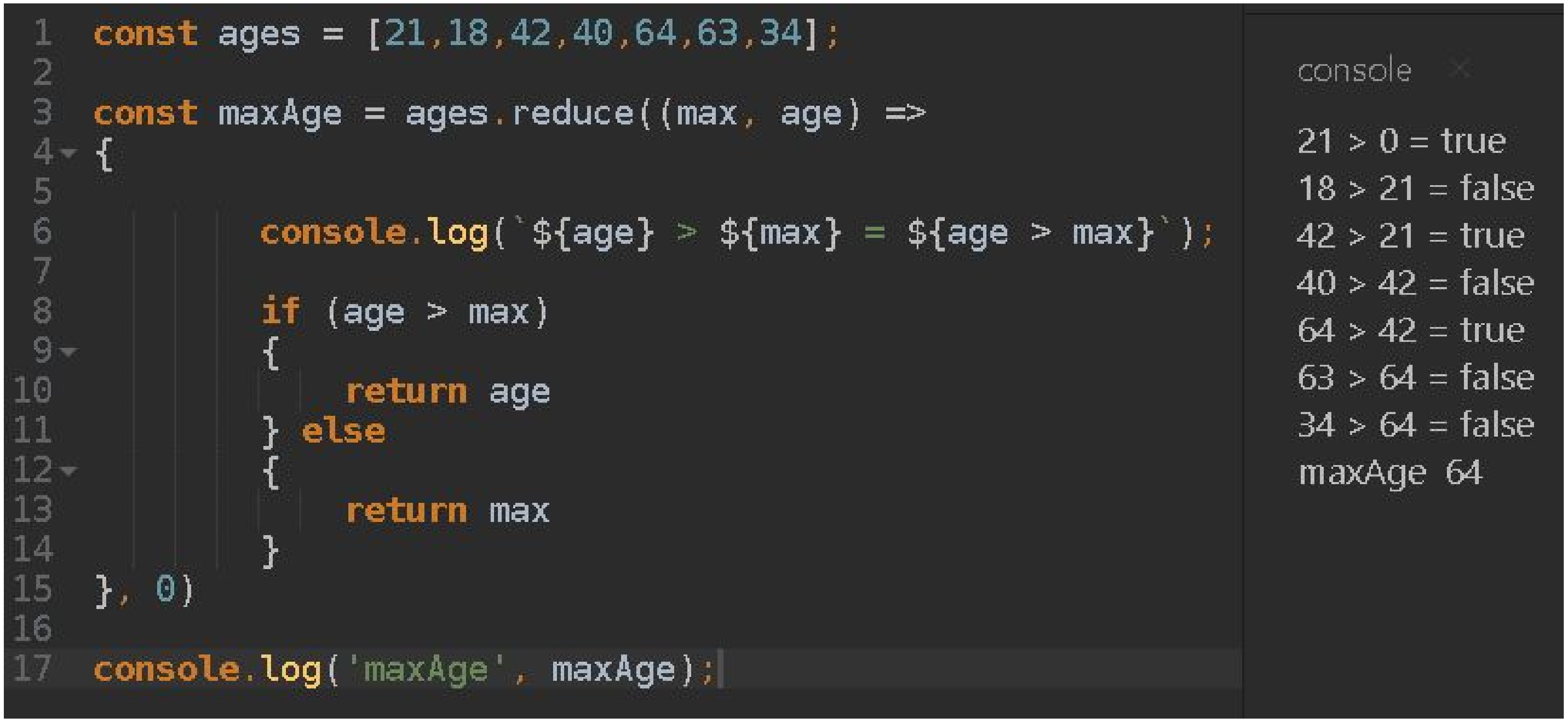
```
[ { "name": "Yorktown" }, { "name": "Stratford" }, {
"name": "Washington & Lee" }, { "name": "Wakefie
ld" } ]
item-- { "name": "Yorktown" }
item-- { "name": "Stratford" }
inside if ---> { "name": "Stratford" }
item-- { "name": "Washington & Lee" }
item-- { "name": "Wakefield" }
[ { "name": "Yorktown" }, { "name": "HB Woodlawn"
}, { "name": "Washington & Lee" }, { "name": "Wak
efield" } ]
```

```
const editName = (oldName, name, arr) =>
    arr.map(item => (item.name === oldName) ?
        ({...item,name}) :
        item
    )
```



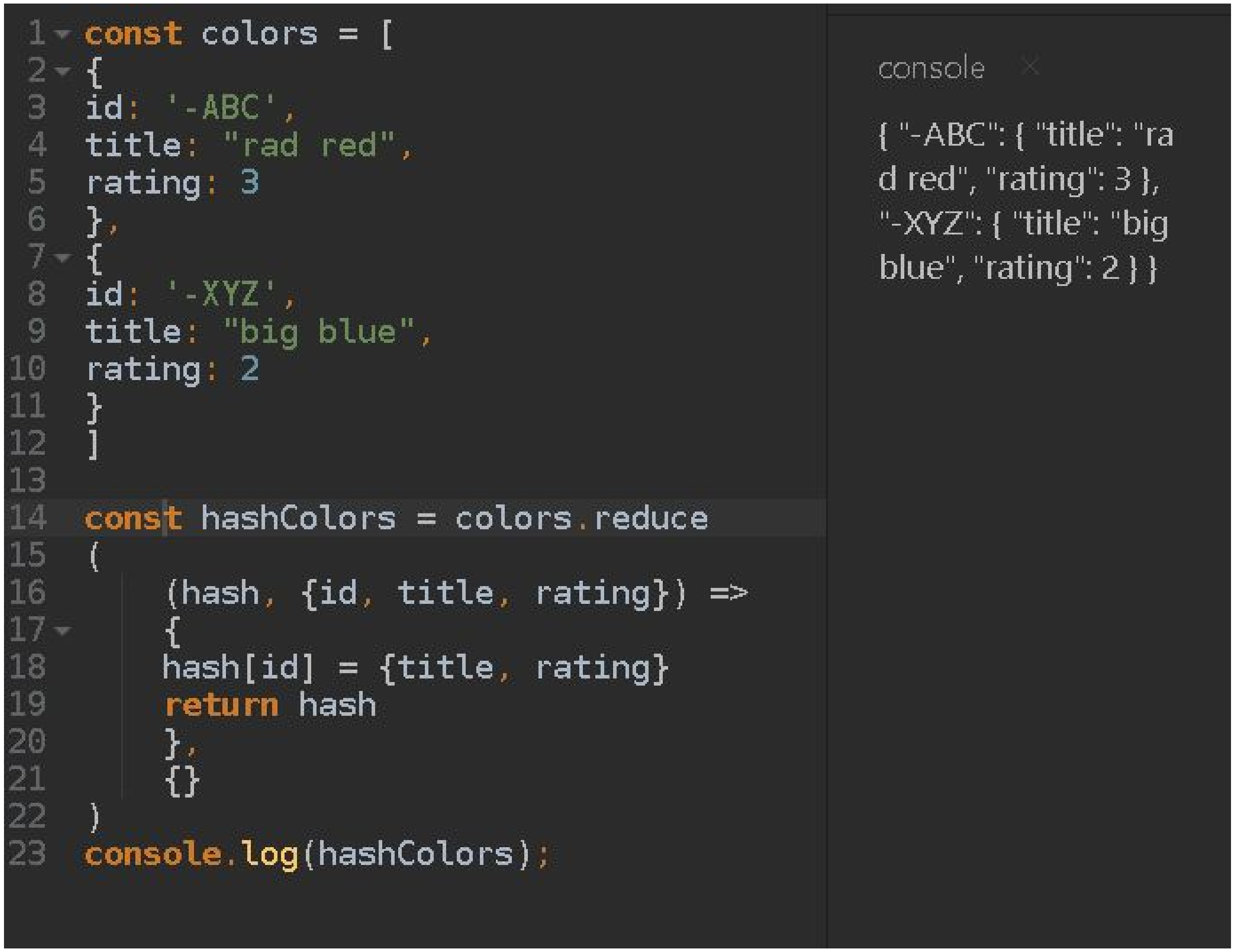
```javascript
const schools = {
"Yorktown": 10,
"Washington & Lee": 2,
"Wakefield": 5
};
console.log(schools)

const schoolArray =
Object.keys(schools).map(key =>
({
name: key,
wins: schools[key]
})
);


console.log(schoolArray)
```

```
{ "Yorktown": 10, "Washington & Lee": 2, "Wakefiel
d": 5 }
[ { "name": "Yorktown", "wins": 10 }, { "name": "Was
hington & Lee", "wins": 2 }, { "name": "Wakefield",
"wins": 5 } ]
```



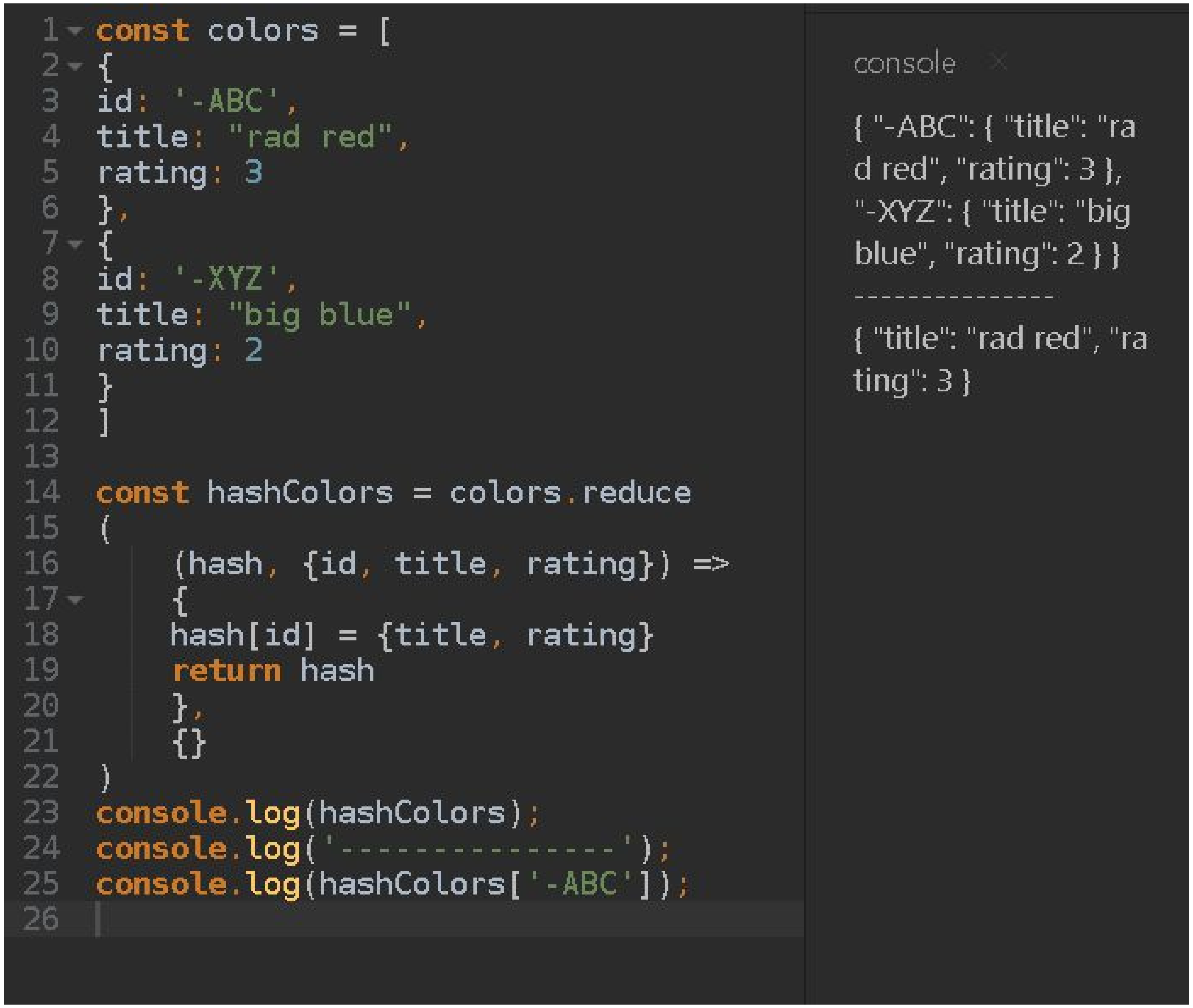
```
const ages = [21,18,42,40,64,63,34];

const maxAge = ages.reduce((max, age) =>
{

      console.log(`${age} > ${max} = ${age > max}`);

      if (age > max)
      {
        return age
      } else
      {
        return max
      }
}, 0)

console.log('maxAge', maxAge);
```

```
21 > 0 = true
18 > 21 = false
42 > 21 = true
40 > 42 = false
64 > 42 = true
63 > 64 = false
34 > 64 = false
maxAge 64
```



```
const colors = [
{
id: '-ABC',
title: "rad red",
rating: 3
},
{
id: '-XYZ',
title: "big blue",
rating: 2
}
]

const hashColors = colors.reduce
(
    (hash, {id, title, rating}) =>
    {
    hash[id] = {title, rating}
    return hash
    },
    {}
)
console.log(hashColors);
```

```
{ "-ABC": { "title": "ra
d red", "rating": 3 },
"-XYZ": { "title": "big
blue", "rating": 2 } }
```



```
const colors = [
{
id: '-ABC',
title: "rad red",
rating: 3
},
{
id: '-XYZ',
title: "big blue",
rating: 2
}
]

const hashColors = colors.reduce
(
    (hash, {id, title, rating}) =>
    {
    hash[id] = {title, rating}
    return hash
    },
    {}
)
console.log(hashColors);
console.log('---------------');
console.log(hashColors['-ABC']);

```

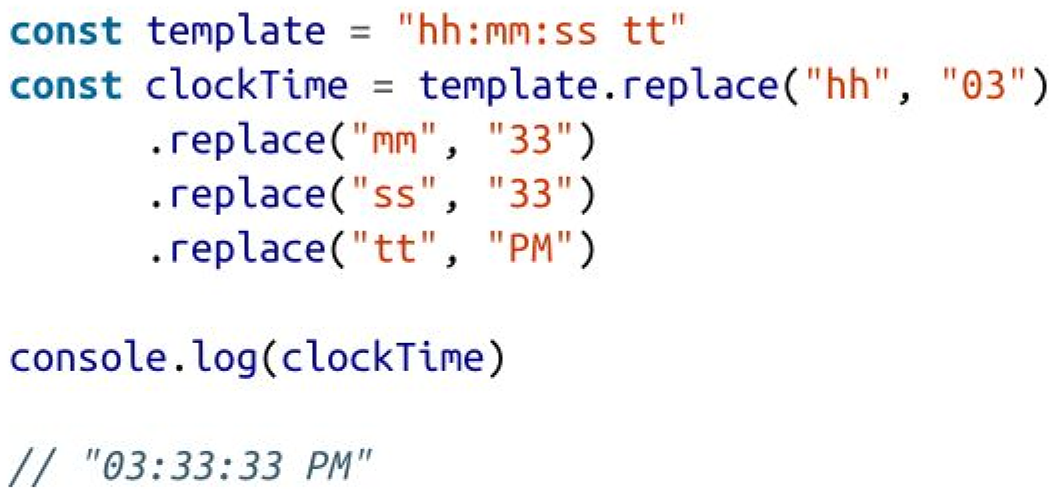
console

```
{ "-ABC": { "title": "ra
d red", "rating": 3 },
"-XYZ": { "title": "big
blue", "rating": 2 } }
---------------
{ "title": "rad red", "ra
ting": 3 }
```

```
const colors = ["red", "red", "green", "blue", "green"];

const distinctColors = colors.reduce(
    (distinct, color) =>
        (distinct.indexOf(color) !== -1) ?
            distinct :
            [...distinct, color],
    []
)

console.log(distinctColors)

// ["red", "green", "blue"]
```

Composition

Chaining



```javascript
const template = "hh:mm:ss tt"
const clockTime = template.replace("hh", "03")
      .replace("mm", "33")
      .replace("ss", "33")
      .replace("tt", "PM")

console.log(clockTime)

// "03:33:33 PM"
```

```
const both = date => appendAMPM(civilianHours(date))
```

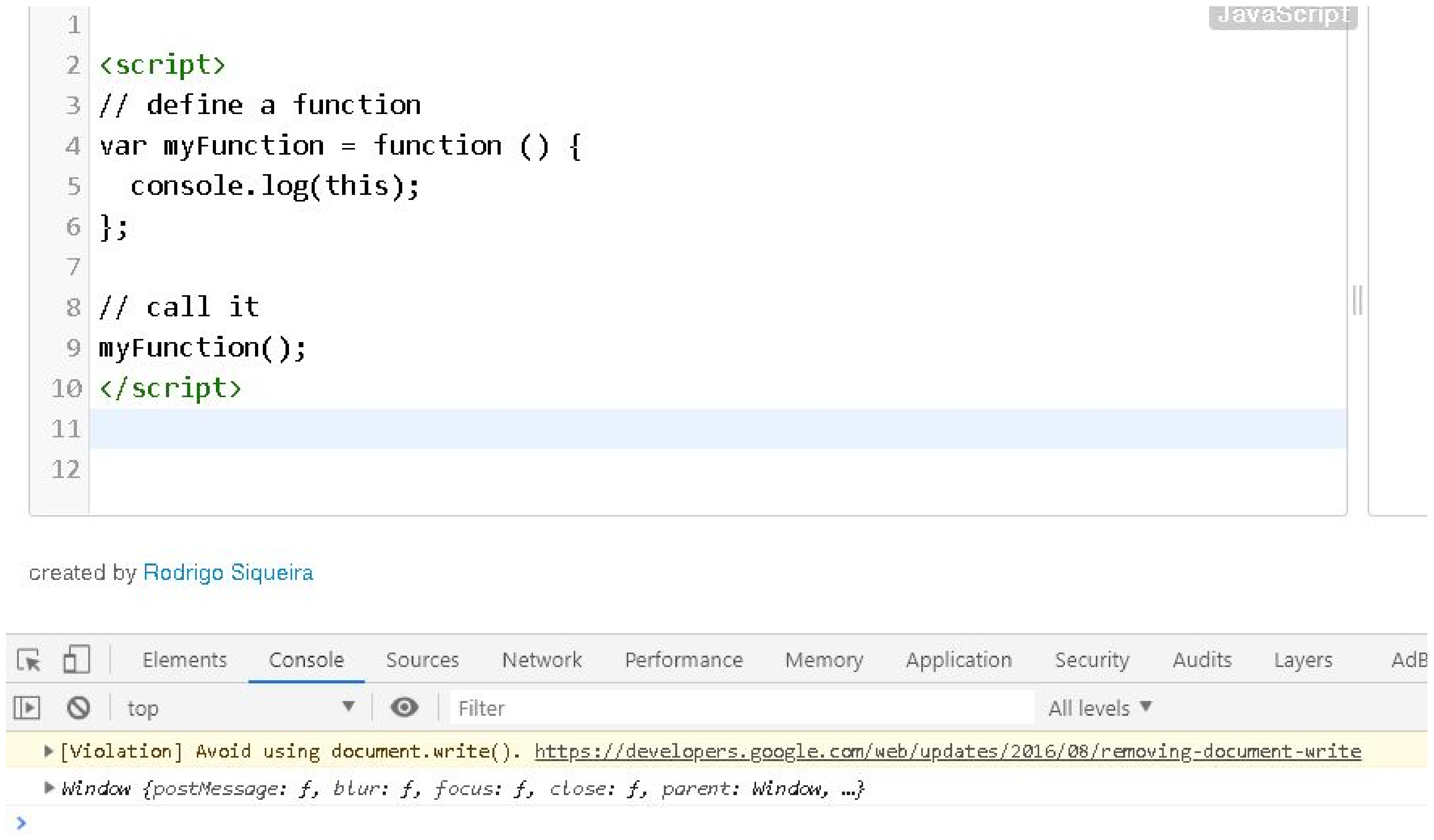
```
const both = compose(
    civilianHours,
    appendAMPM
)

both(new Date())
```

Compose takes in functions as arguments and returns a single function

This, Call, Apply, Bind

.

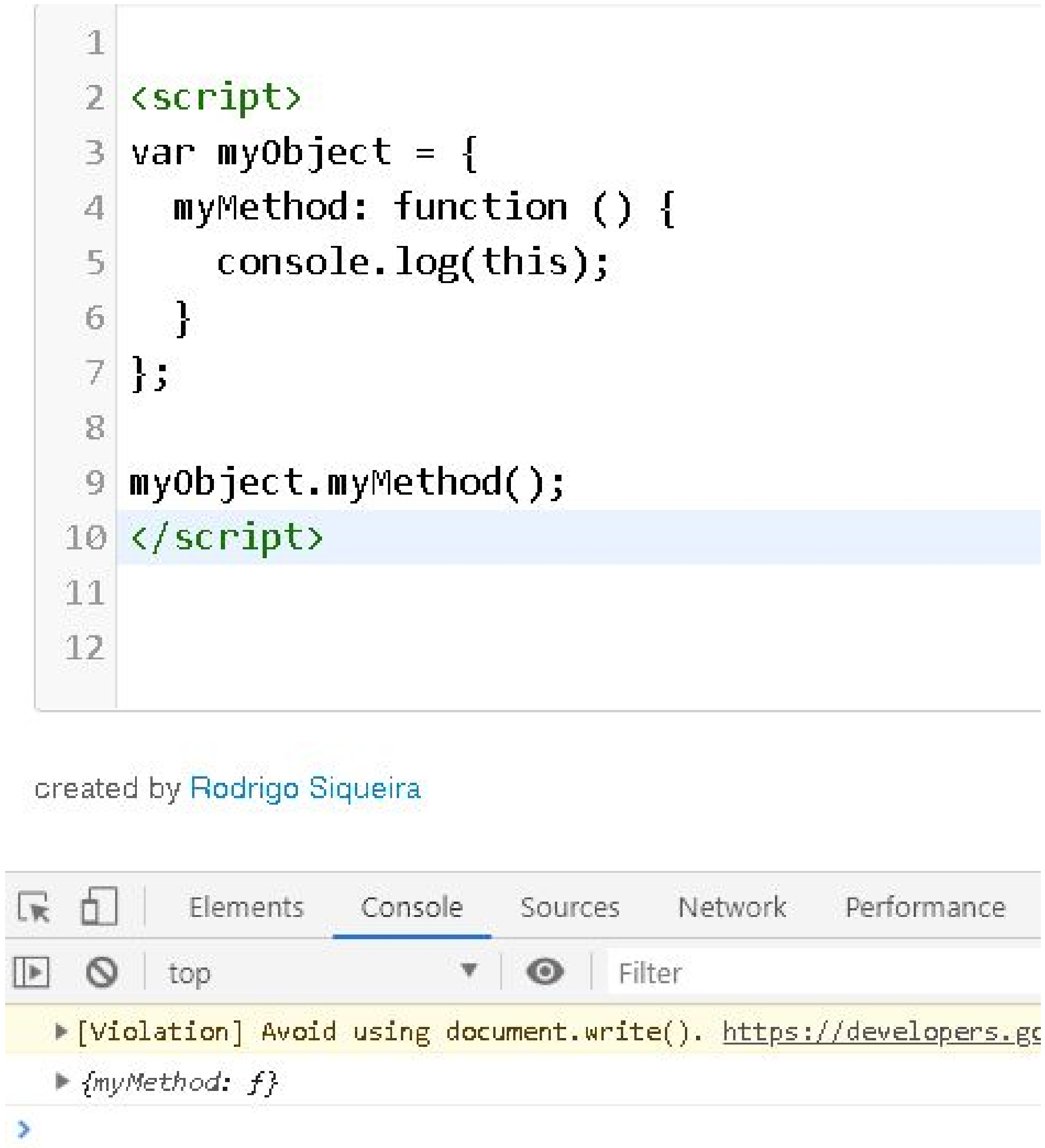
This



```

<script>
// define a function
var myFunction = function () {
  console.log(this);
};

// call it
myFunction();
</script>


```

created by Rodrigo Siqueira

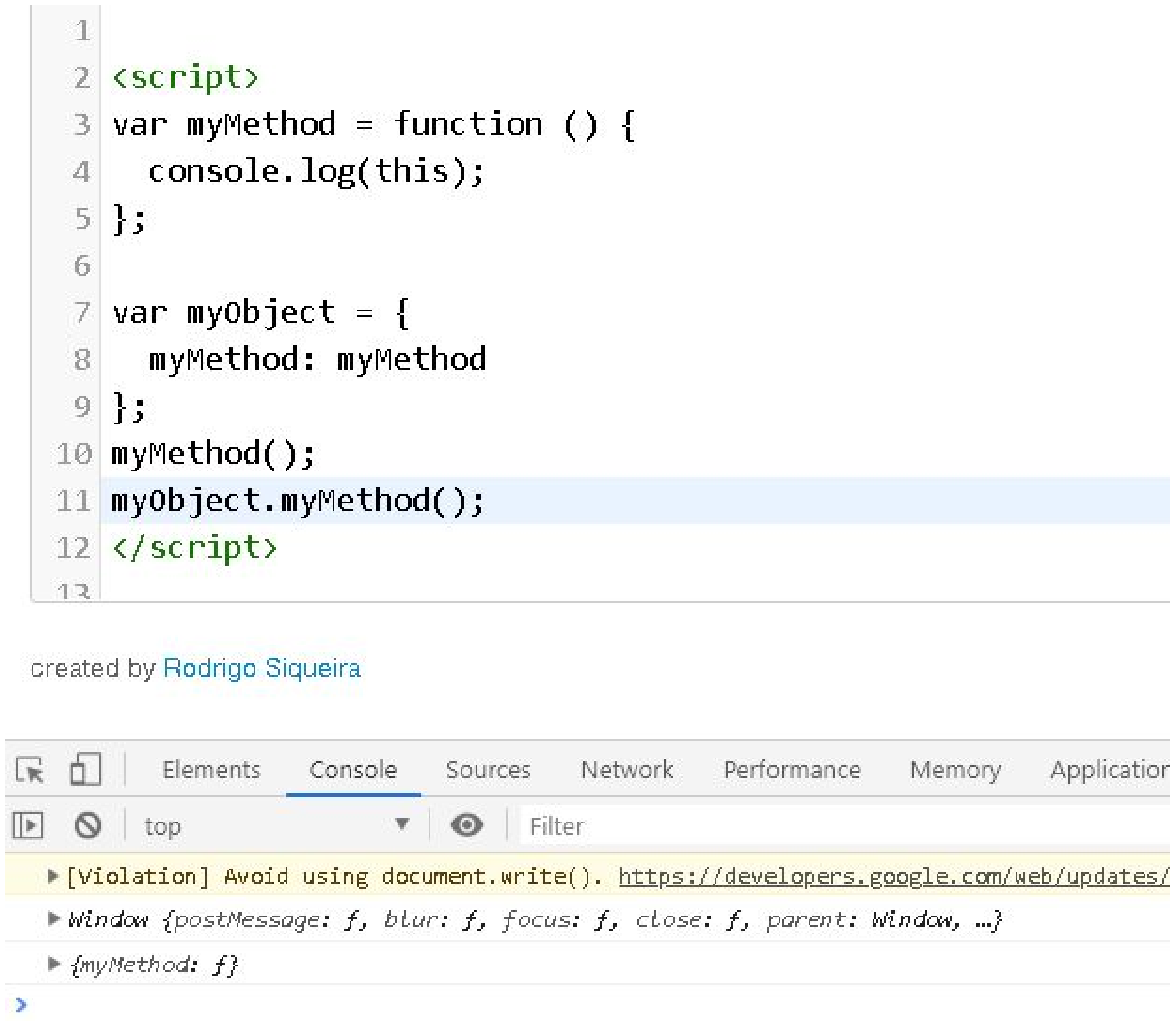
```
▸ [Violation] Avoid using document.write(). https://developers.google.com/web/updates/2016/08/removing-document-write
▸ Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …}
>
```



```

<script>
var myObject = {
  myMethod: function () {
    console.log(this);
  }
};

myObject.myMethod();
</script>


```

created by Rodrigo Siqueira

```
▸ [Violation] Avoid using document.write(). https://developers.gc
▸ {myMethod: ƒ}
>
```

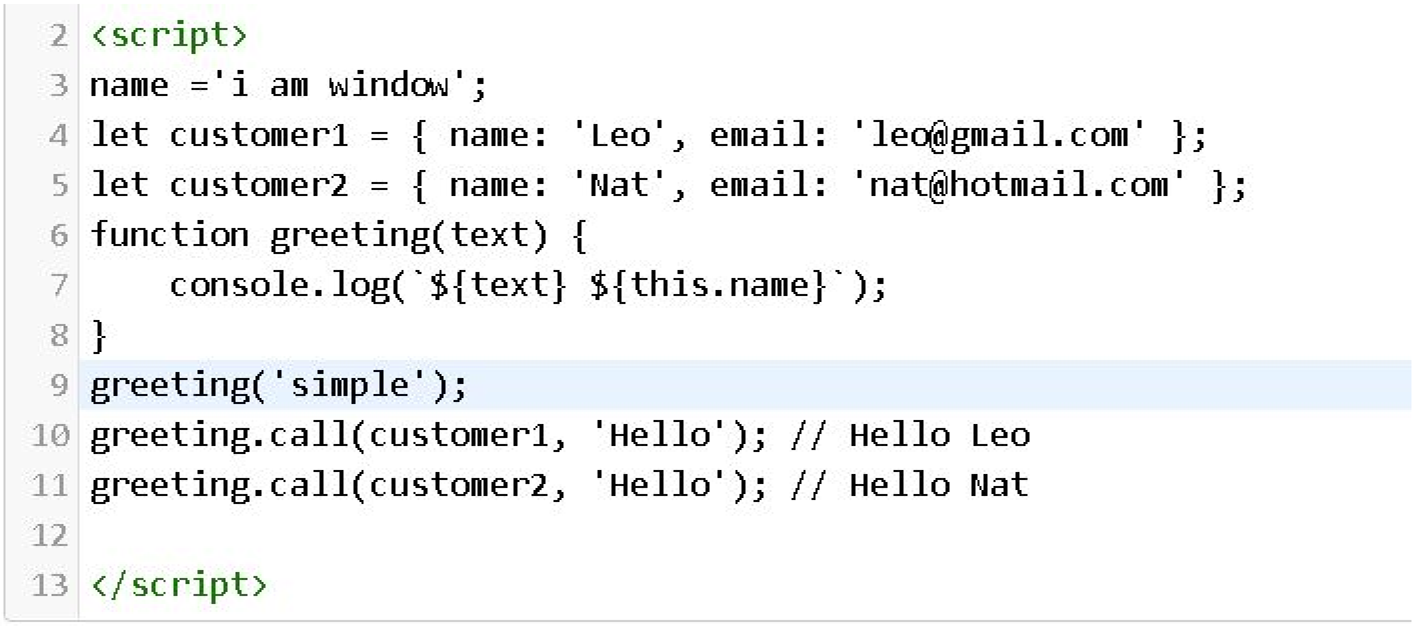


```
<script>
var myMethod = function () {
  console.log(this);
};

var myObject = {
  myMethod: myMethod
};
myMethod();
myObject.myMethod();
</script>
```

created by Rodrigo Siqueira

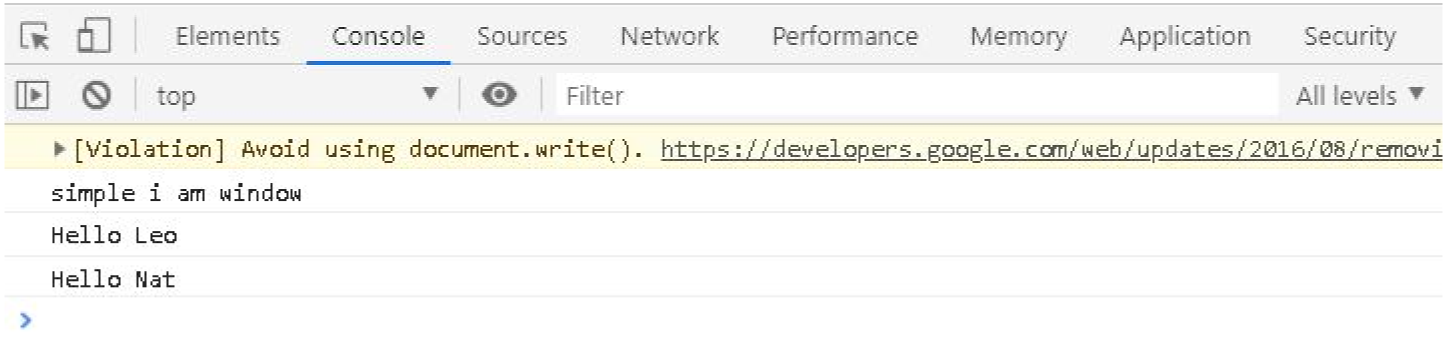
```
▶ [Violation] Avoid using document.write(). https://developers.google.com/web/updates/
▶ Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …}
▶ {myMethod: ƒ}
>
```

Call



```
<script>
name ='i am window';
let customer1 = { name: 'Leo', email: 'leo@gmail.com' };
let customer2 = { name: 'Nat', email: 'nat@hotmail.com' };
function greeting(text) {
    console.log(`${text} ${this.name}`);
}
greeting('simple');
greeting.call(customer1, 'Hello'); // Hello Leo
greeting.call(customer2, 'Hello'); // Hello Nat

</script>
```

created by Rodrigo Siqueira



```
▶[Violation] Avoid using document.write(). https://developers.google.com/web/updates/2016/08/removi
simple i am window
Hello Leo
Hello Nat
```

Apply

```
let customer1 = { name: 'Leo', email: 'leo@gmail.com' };
let customer2 = { name: 'Nat', email: 'nat@hotmail.com' };

function greeting(text, text2) {
   console.log(`${text} ${this.name}, ${text2}`);
}

greeting.apply(customer1, ['Hello', 'How are you?']); // output
Hello Leo, How are you?
greeting.apply(customer2, ['Hello', 'How are you?']); // output
Hello Natm How are you?
```

Bind

```
let customer1 = { name: 'Leo', email: 'leo@gmail.com' };
let customer2 = { name: 'Nat', email: 'nat@hotmail.com' };

function greeting(text) {
   console.log(`${text} ${this.name}`);
}

let helloLeo = greeting.bind(customer1);
let helloNat = greeting.bind(customer2);

helloLeo('Hello'); // Hello Leo
helloNat('Hello'); // Hello Nat
```

When a function is created, a keyword called **this** is created (behind the scenes), which links to the object in which the function operates.

The **this** keyword value has nothing to do with the function itself, how the function is called determines this's value

Call and **Apply** are interchangeable. You can decide whether it’s easier to send in an array or a comma separated list of arguments.

Bind is different. It always returns a new function.

Closure

```
// Initiate counter
var counter = 0;

// Function to increment counter
function add() {
  counter += 1;
}

// Call add() 3 times
add();
add();
add();

// The counter should now be 3
// Initiate counter
var counter = 0;

// Function to increment counter
function add() {
  var counter = 0;
  counter += 1;
}

// Call add() 3 times
add();
add();
add();

//The counter should now be 3. But it is 0
```

```
// Function to increment counter
function add() {
  var counter = 0;
  counter += 1;
  return counter;
}

// Call add() 3 times
add();
add();
add();

//The counter should now be 3. But it is 1.
```

```
function add() {
  var counter = 0;
  function plus() {counter += 1;}
  plus();
  return counter;
}
```

Try it Yourself »

This could have solved the counter dilemma, if we could reach the `plus()` function from the outside.

We also need to find a way to execute `counter = 0` only once.

We need a closure.

```
var add = (function () {
  var counter = 0;
  return function () {counter += 1; return counter}
})();

add();
add();
add();

// the counter is now 3
```

The variable `add` is assigned the return value of a self-invoking function.

The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.

This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.

This is called a JavaScript **closure.** It makes it possible for a function to have "**private**" variables.

The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

A closure is a function having access to the parent scope, even after the parent function has closed.

Callback, Promises, Async/Await

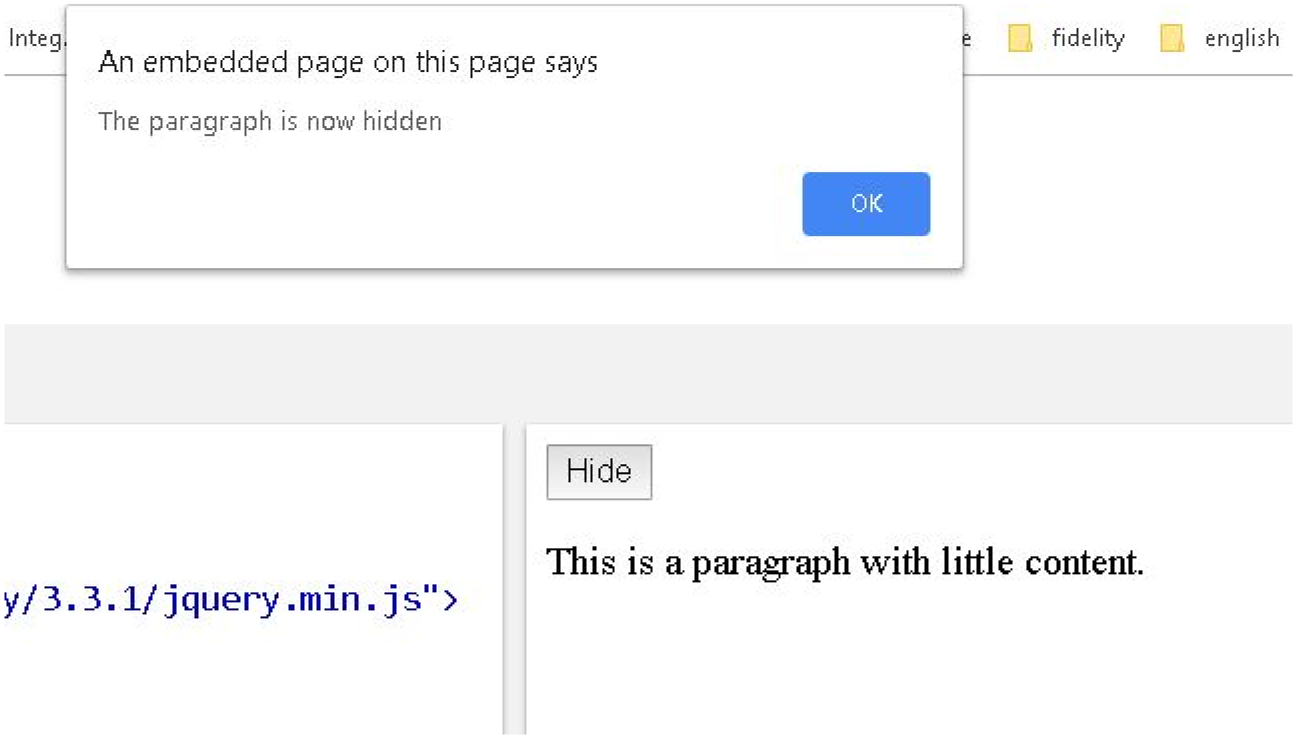
Callbacks

Typical syntax: **$(*selector*).hide(*speed,callback*);**

A callback is a function that is to be executed **after** another function has finished executing — hence the name ‘call back’.

Example without Callback

```
$("button").click(function(){
  $("p").hide(1000);
  alert("The paragraph is now hidden");
});
```



An embedded page on this page says

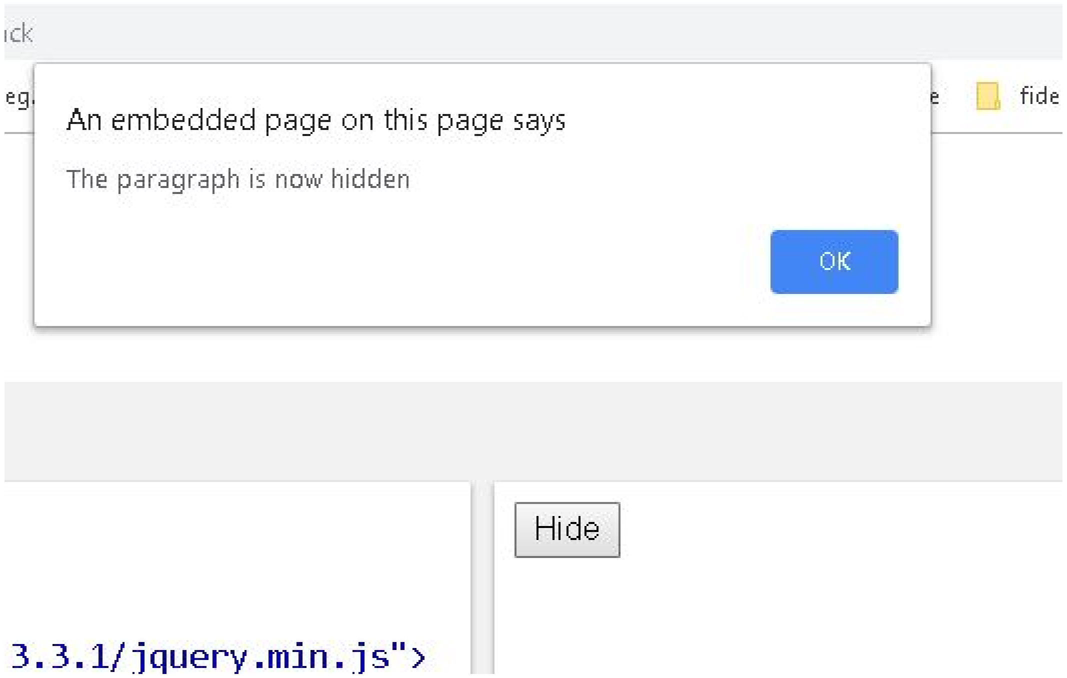
The paragraph is now hidden

```
y/3.3.1/jquery.min.js">
```

This is a paragraph with little content.

Example with Callback

```
$("button").click(function(){
  $("p").hide("slow", function(){
    alert("The paragraph is now hidden");
  });
});
```



An embedded page on this page says

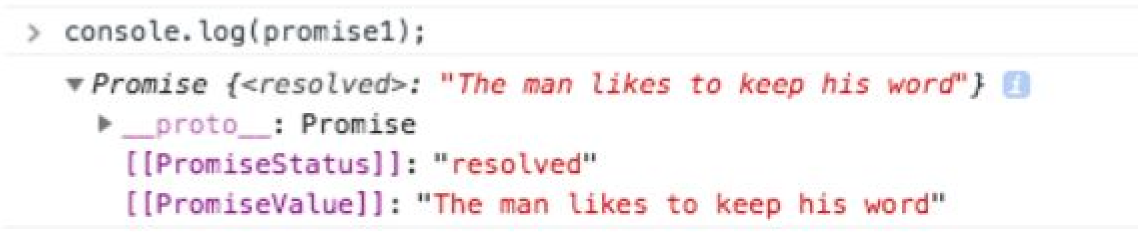
The paragraph is now hidden

```
3.3.1/jquery.min.js">
```

Promises

Promises are a new feature in the ES6 (ES2015) JavaScript spec that allow you to very easily deal with asynchronous code without resolving to multiple levels of callback functions. Goodbye callback hell!

```
var keepsHisWord;
keepsHisWord = true;
promise1 = new Promise(function(resolve, reject) {
  if (keepsHisWord) {
    resolve("The man likes to keep his word");
  } else {
    reject("The man doesnt want to keep his word");
  }
});
console.log(promise1);
```

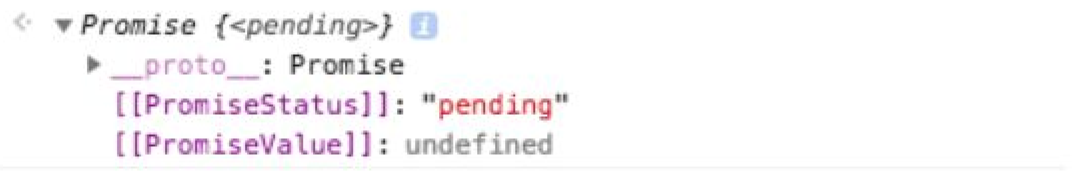


```
> console.log(promise1);
  ▼Promise {<resolved>: "The man likes to keep his word"}
    ▶__proto__: Promise
     [[PromiseStatus]]: "resolved"
     [[PromiseValue]]: "The man likes to keep his word"
```

Every promise has a state and value

```
promise2 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    resolve({
      message: "The man likes to keep his word",
      code: "aManKeepsHisWord"
    });
  }, 10 * 1000);
});
console.log(promise2);
```

The above code just creates a promise that will resolve unconditionally after 10 seconds. So we can checkout the state of the promise until it is resolved.

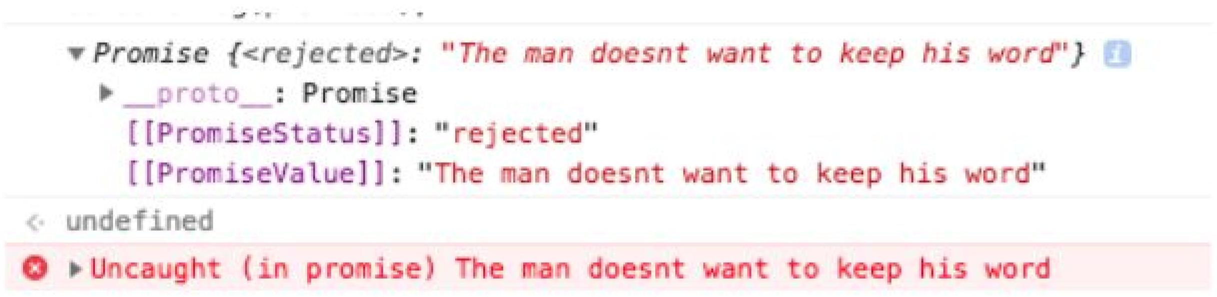


```
▼Promise {<pending>}
  ▶__proto__: Promise
   [[PromiseStatus]]: "pending"
   [[PromiseValue]]: undefined
```

state of promise until it is resolved or rejected

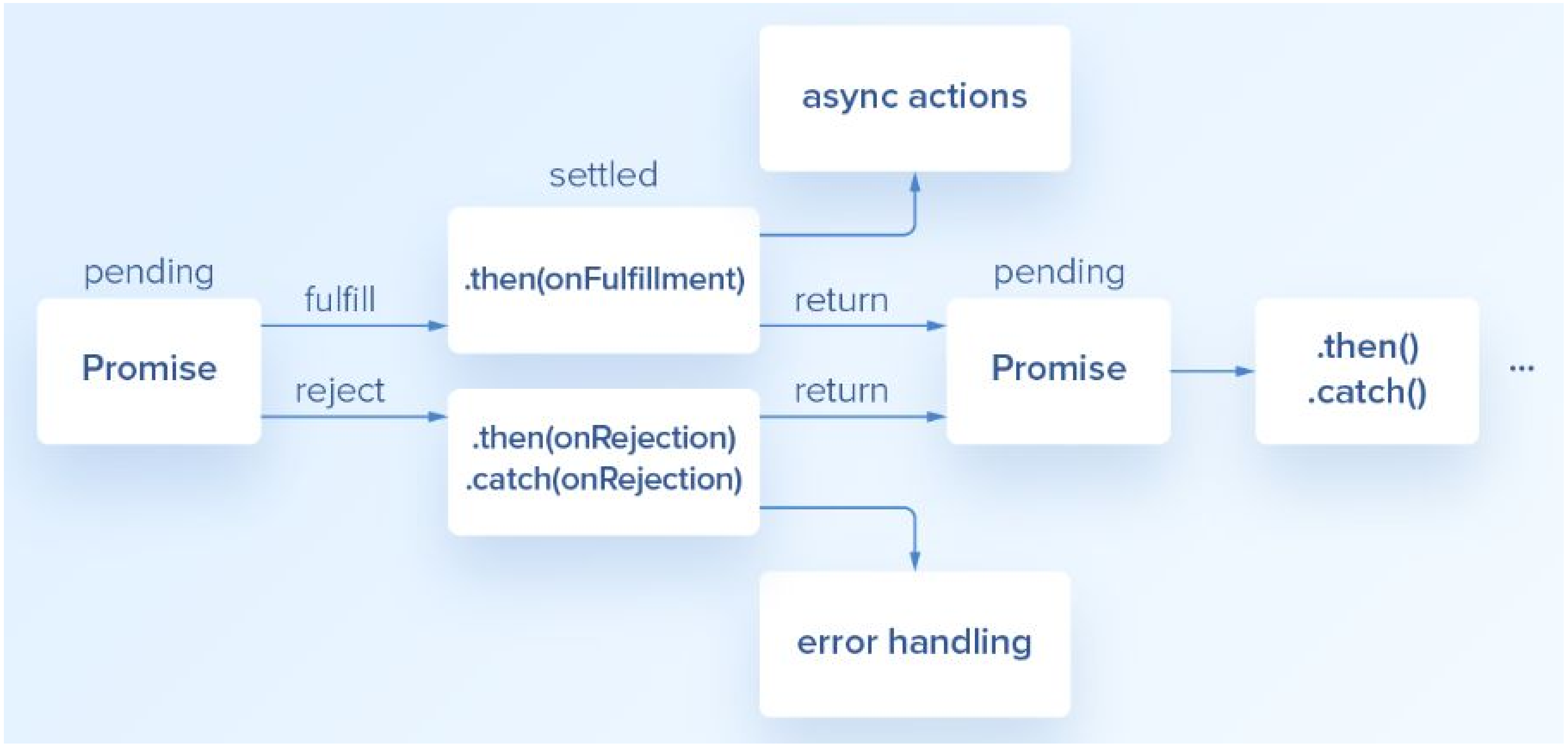
```
keepsHisWord = false;
promise3 = new Promise(function(resolve, reject) {
  if (keepsHisWord) {
    resolve("The man likes to keep his word");
  } else {
    reject("The man doesn't want to keep his word");
  }
});
console.log(promise3);
```

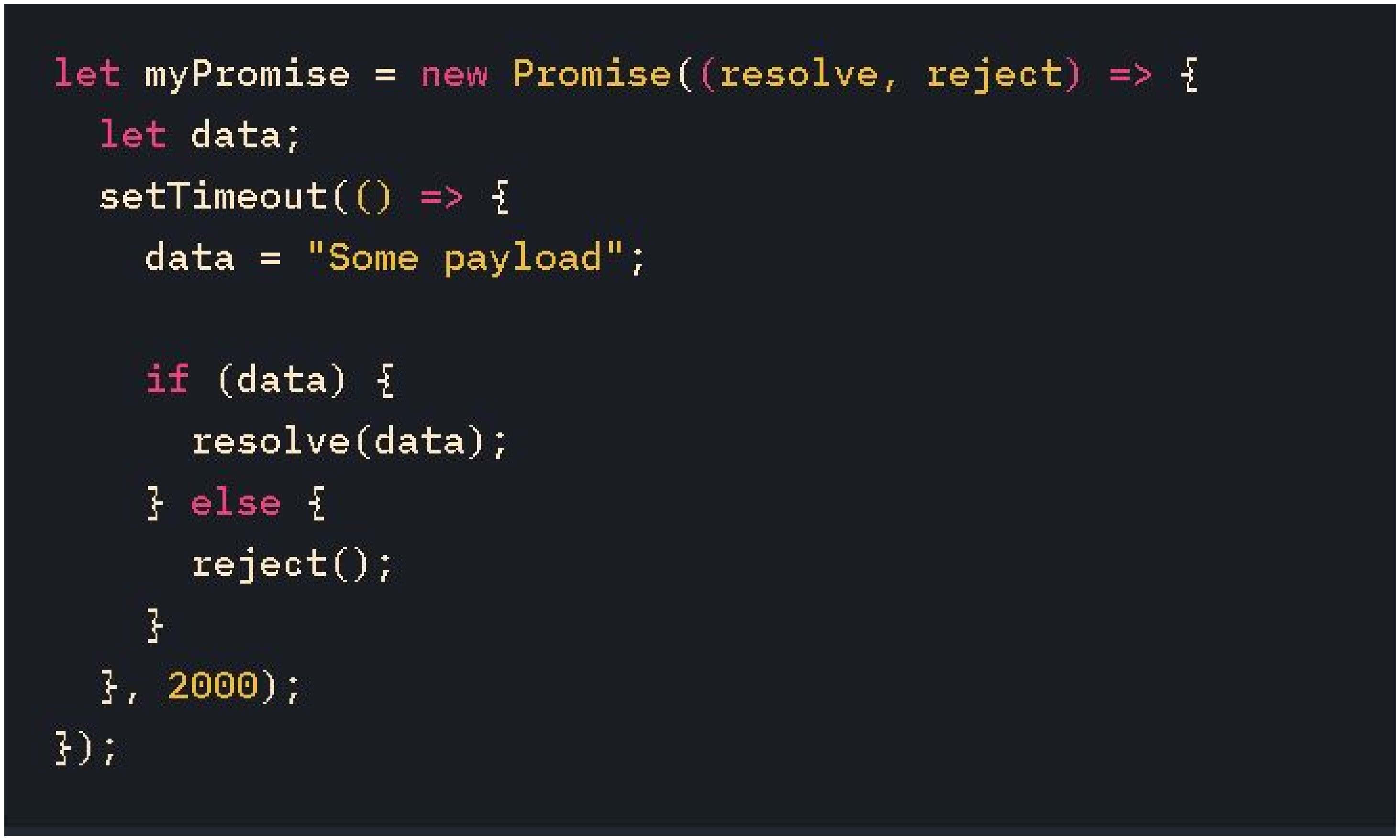
Since this will create a unhanded rejection chrome browser will show an error. You can ignore it for now. We will get back to that later.



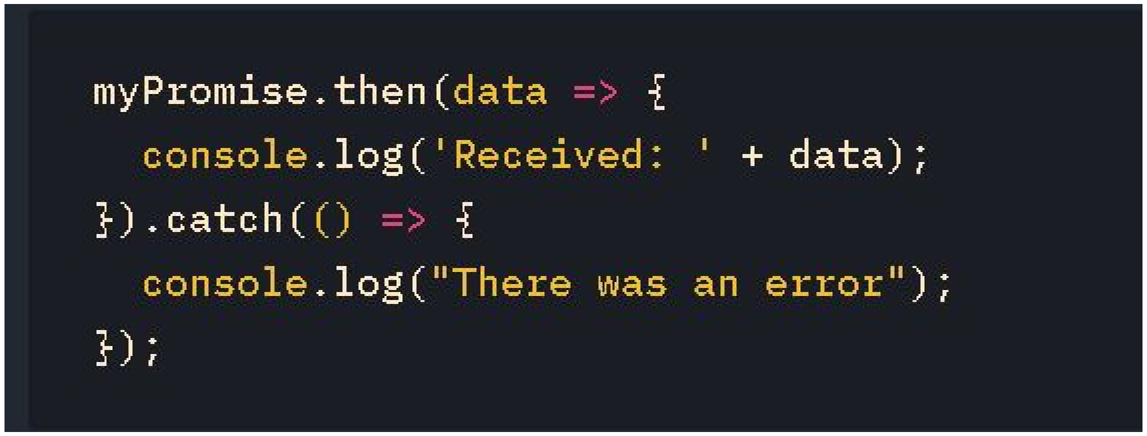
```
▼Promise {<rejected>: "The man doesnt want to keep his word"}
  ▶__proto__: Promise
   [[PromiseStatus]]: "rejected"
   [[PromiseValue]]: "The man doesnt want to keep his word"
<· undefined
⊗ ▶Uncaught (in promise) The man doesnt want to keep his word
```

rejections in promises


async actions
settled
pending
fulfill
.then(onFulfillment)
return
pending
Promise
reject
return
Promise
.then()
.catch()
...
.then(onRejection)
.catch(onRejection)
error handling



```
let myPromise = new Promise((resolve, reject) => {
  let data;
  setTimeout(() => {
    data = "Some payload";

    if (data) {
      resolve(data);
    } else {
      reject();
    }
  }, 2000);
});
```



```
myPromise.then(data => {
  console.log('Received: ' + data);
}).catch(() => {
  console.log("There was an error");
});
```

Async and Await

Async and Await are extensions of promises.

An asynchronous function is a function which operates asynchronously via the event loop, using an implicit `Promise` to return its result

```
async function msg() {
  const a = await who();
  const b = await what();
  const c = await where();

  console.log(`${ a } ${ b } ${ c }`);
}
```

```
async function msg() {
  const [a, b, c] = await Promise.all([who(), what(), where()]);

  console.log(`${ a } ${ b } ${ c }`);
}
```

Mixins

So if not inheritance, what are the alternatives?

Composition is the obvious answer. Objects that contain additional functionality can be injected, rather than relying on the prototype chain.

```javascript
// mixin
let sayHiMixin = {
  sayHi() {
    alert(`Hello ${this.name}`);
  },
  sayBye() {
    alert(`Bye ${this.name}`);
  }
};

// usage:
class User {
  constructor(name) {
    this.name = name;
  }
}

// copy the methods
Object.assign(User.prototype, sayHiMixin);

// now User can say hi
new User("Dude").sayHi(); // Hello Dude!
```