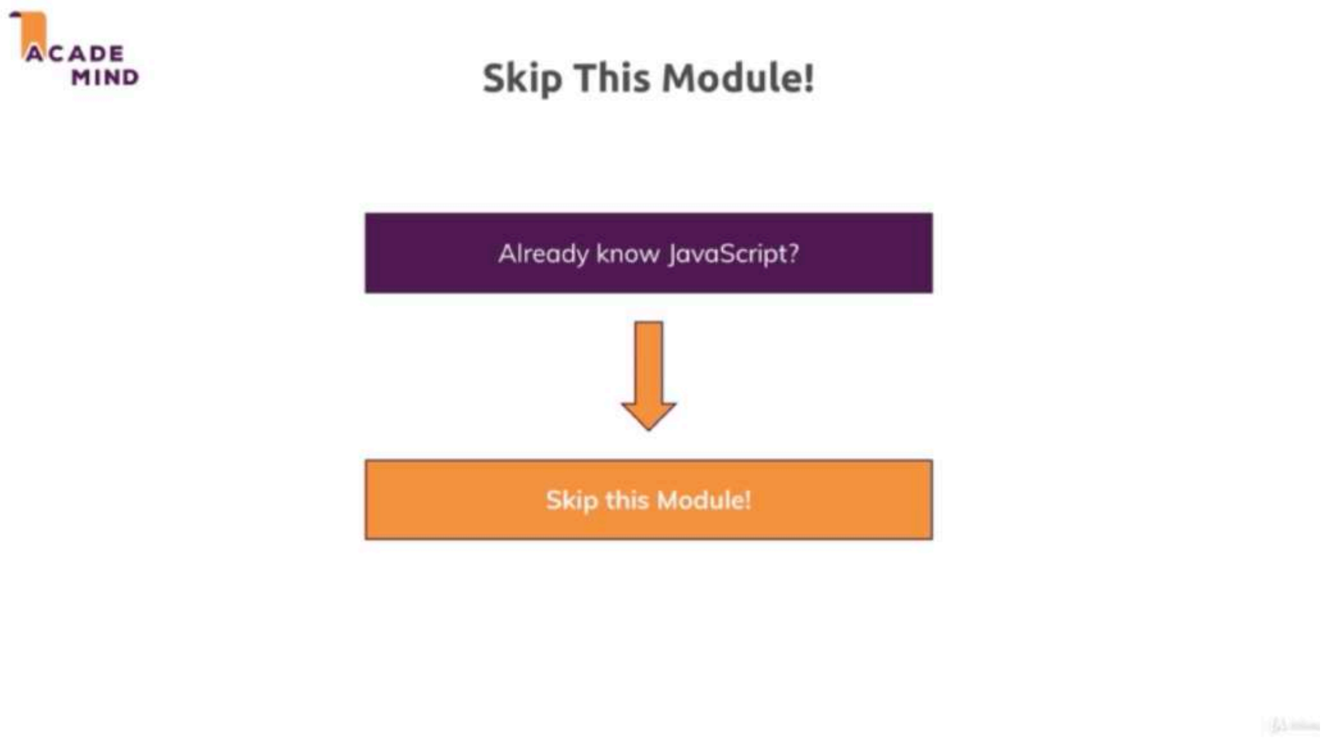
2. Optional JavaScript - A Quick Refresher

* Chapter 9: Module Introduction

![](images/9-module-introduction-1.png)



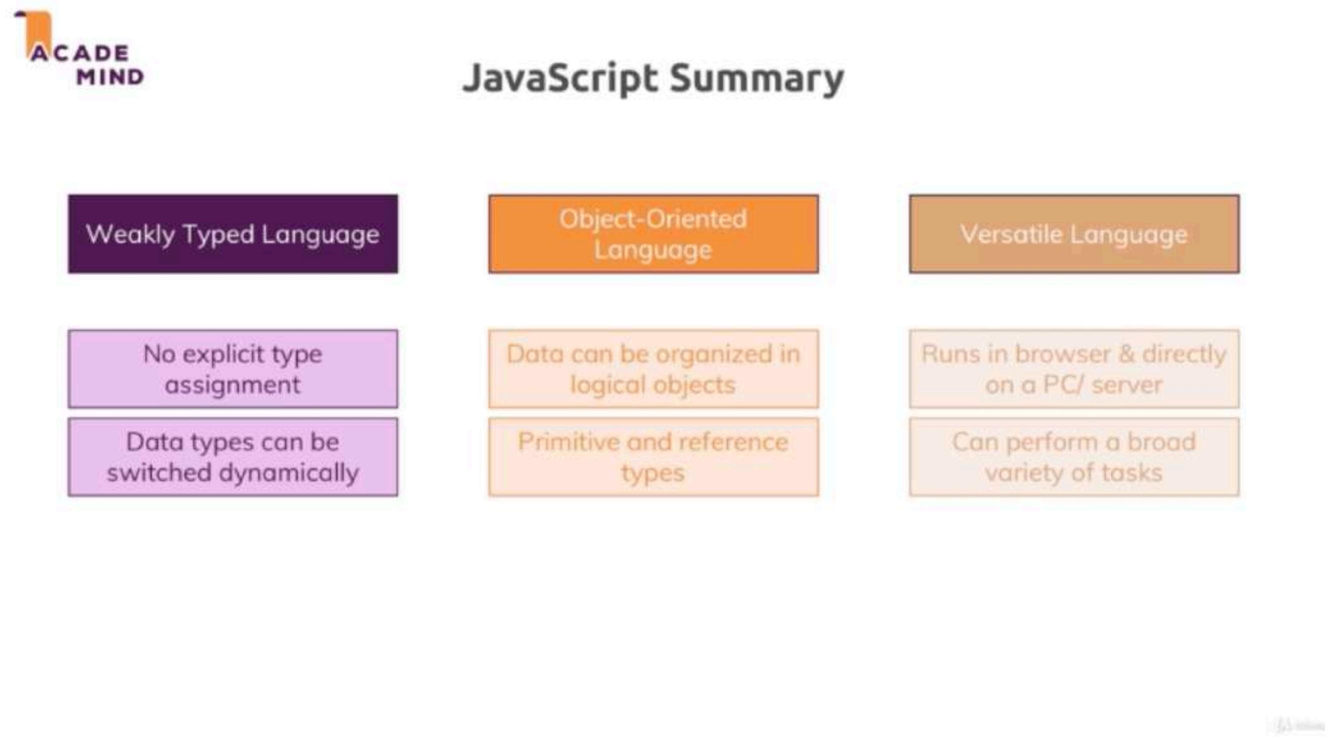
Skip This Module!

Already know JavaScript?

Skip this Module!

* Chapter 10: JavaScript In A Nutshell

![](images/10-javascript-in-a-nutshell-1.png)



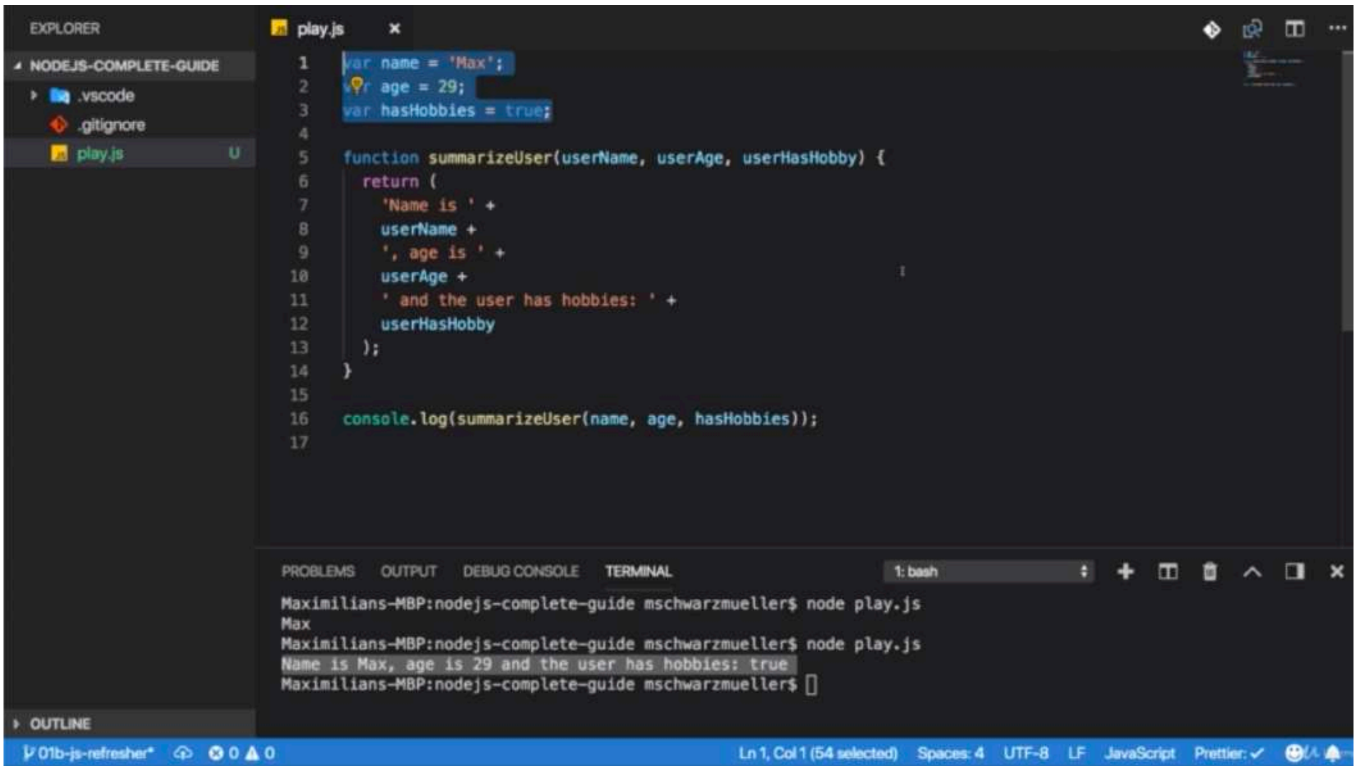
JavaScript Summary

Weakly Typed Language	Object-Oriented Language	Versatile Language
No explicit type assignment	Data can be organized in logical objects	Runs in browser & directly on a PC/ server
Data types can be switched dynamically	Primitive and reference types	Can perform a broad variety of tasks

- Data types can be switched dynamically. but it also lead to errors.

* Chapter 11: Refreshing The Core Syntax

![](images/11-refreshing-the-core-syntax-1.png)
![](images/11-refreshing-the-core-syntax-2.png)
![](images/11-refreshing-the-core-syntax-3.png)
![](images/11-refreshing-the-core-syntax-4.png)
![](images/11-refreshing-the-core-syntax-5.png)
![](images/11-refreshing-the-core-syntax-6.png)
![](images/11-refreshing-the-core-syntax-7.png)
![](images/11-refreshing-the-core-syntax-8.png)
![](images/11-refreshing-the-core-syntax-9.png)
![](images/11-refreshing-the-core-syntax-10.png)
![](images/11-refreshing-the-core-syntax-11.png)
![](images/11-refreshing-the-core-syntax-12.png)
![](images/11-refreshing-the-core-syntax-13.png)



```js
var name = 'Max';
var age = 29;
var hasHobbies = true;

function summarizeUser(userName, userAge, userHasHobby) {
  return (
    'Name is ' +
    userName +
    ', age is ' +
    userAge +
    ' and the user has hobbies: ' +
    userHasHobby
  );
}

console.log(summarizeUser(name, age, hasHobbies));
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Max
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Name is Max, age is 29 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ 
```

What are Primitives?

This article and videos is named "Reference vs Primitive Values".

So let's start - what are "Primitives"?

Here's an example:

```js
var age = 28
```

The `age` variable (you could also use `let` or `const` by the way) stores a number value. The number `28`.

Number values are called "primitive values" because they're very simple building blocks of JavaScript apps.

Other simple core building blocks are:

```js
var name = 'Max' // strings are primitives, too!
var isMale = true // so are booleans
```

So numbers, string, booleans - these are probably very well-known to you. `undefined` and `null` are additional primitive types.

What are Reference Types Then?

So we learned what “Primitives” (or “primitive types”) are.

What are “reference types” then?

`Object`s and `Array`s!

```
var person = {
  name: 'Max',
  age: 28,
}

var hobbies = ['Sports', 'Cooking']
```

Here, `person` is an object and therefore a so-called reference type. Please note that it holds properties that in turn have primitive values. This doesn’t affect the object being a reference type though. And you could of course also have nested objects or arrays inside the `person` object.

The `hobbies` array is also a reference type - in this case, it holds a list of strings. A `string` is a primitive value/ type as you learned but this doesn’t affect the `array`. Arrays are **always** reference types.

What’s the Difference?

Cool, we got two different types of values. What’s the idea behind all of that?

It’s related to memory management.

Behind the scenes, JavaScript of course has to store the values you assign to properties or variable in memory.

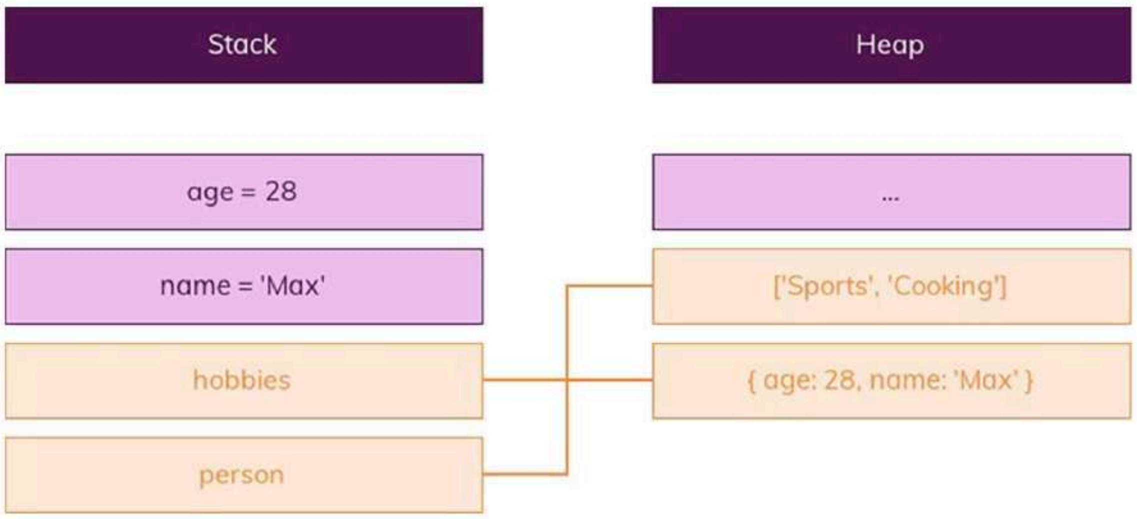
JavaScript knows two types of memory: The **Stack** and the **Heap**. You can dive much deeper if you want to.

Here’s a super-short summary: The stack is essentially an easy-to-access memory that simply manages its items as a - well - stack. Only items for which the size is known in advance can go onto the stack. This is the case for numbers, strings, booleans.

The heap is a memory for items of which you can’t pre-determine the exact size and structure. Since objects and arrays can be mutated and change at runtime, they have to go into the heap therefore.

Obviously, there’s more to it but this rough differentiation will do for now.

For each heap item, the exact address is stored in a pointer which points at the item in the heap. This pointer in turn is stored on the stack. That will become important in a second.



Okay, so we got different memories. But how does that make a difference to us, the developer?

Strange Behavior of “Reference Types”

The fact that only pointers are stored on the stack for reference types matters a lot!

What’s actually stored in the `person` variable in the following snippet?

```
var person = { name: 'Max' }
```

Is it: a) The object (`{ name: 'Max' }`)

b) The pointer to the object

c) A pointer to the `name` property?

It’s **b)**. A pointer to the `person` object is stored in the variable. The same would be the case for the `hobbies` array.

What does the following code spit out then?

```
var person = { name: 'Max' }
var newPerson = person
newPerson.name = 'Anna'
console.log(person.name) // What does this line print?
```

You’ll see `'Anna'` in the console!

Why?

Because you never copied the person object itself to `newPerson`. You only copied the pointer! It still points at the same address in memory though. Hence changing `newPerson.name` also changes `person.name` because newPerson points at the exactly same object!

This is really important to understand! You’re pointing at the same object, you didn’t copy the object.

It’s the same for arrays.

```
var hobbies = ['Sports', 'Cooking']
var copiedHobbies = hobbies
copiedHobbies.push('Music')
console.log(hobbies[2]) // What does this line print?
```

This prints `'Music'` - for the exact same reason as stated above.

How can you copy the actual Value?

Now that we know that we only copy the pointer - how can we actually copy the value behind the pointer? The actual object or array?

You basically need to construct a new object or array and immediately fill it with the properties or elements of the old object or array.

You got multiple ways of doing this - also depending on which kind of JavaScript version you're using (during development).

Here are the two most popular approaches for arrays:

1) Use the `slice()` method

`slice()` is a standard array method provided by JavaScript. You can check out its full documentation here.

```
var hobbies = ['Sports', 'Cooking']
var copiedHobbies = hobbies.slice()
```

It basically returns a new array which contains all elements of the old element, starting at the starting index you passed (and then up to the max number of elements you defined). If you just call `slice()`, without arguments, you get a new array with all elements of the old array.

2) Use the spread operator

If you're using ES6+, you can use the spread operator.

```
var hobbies = ['Sports', 'Cooking']
var copiedHobbies = [...hobbies]
```

Here, you also create a new array (manually, by using `[]`) and you then use the spread operator (`...`) to "pull all elements of the old array out" and add them to the new array.

For objects

1) `Object.assign()`

You can use the `Object.assign()` syntax which is explained in greater detail here.

```
var person = { name: 'Max' }
var copiedPerson = Object.assign({}, person)
```

This syntax creates a new object (the `{}` part) and assigns all properties of the old object (the second argument) to that newly created one. This creates a copy.

2)

Just as with arrays, you can also use the spread operator on objects.

```
var person = { name: 'Max' }
var copiedPerson = { ...person }
```

This will also create a new object (because you used `{ }`) and will then pull all properties of `person` out of it, into the brand-new objects.

Deep Clones?

Now you know how to clone arrays and objects.

Here's on super-important thing to note though: You're not creating deep clones with either approach!

If you cloned array contains nested arrays or objects as elements or if your object contains properties that hold arrays or other objects, then these nested arrays and objects will **not** have been cloned!

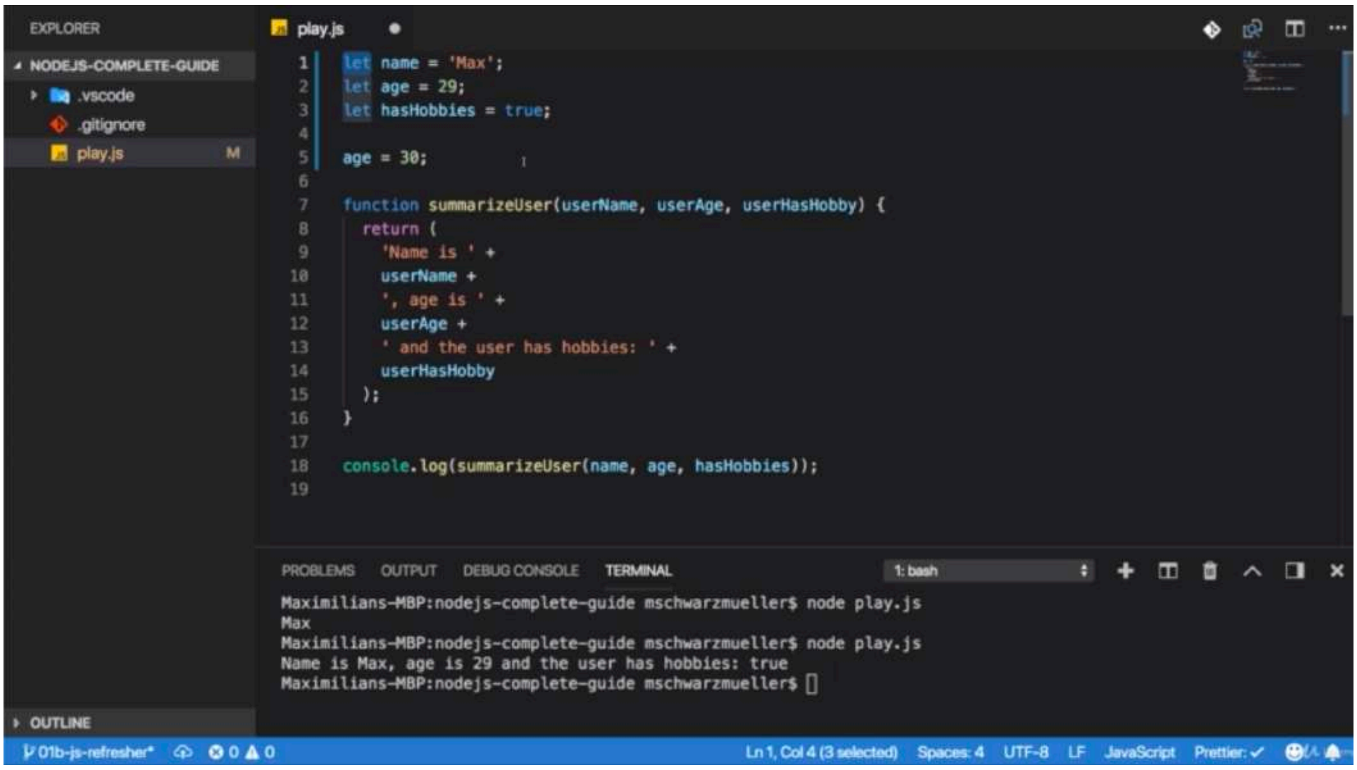
You still have the old pointers, pointing to the old nested arrays/ objects!

You'd have to manually clone every layer that you plan on working with. If you don't plan on changing these nested arrays or objects though, you don't need to clone them.

More about cloning strategies can be read here

* Chapter 12: let & const

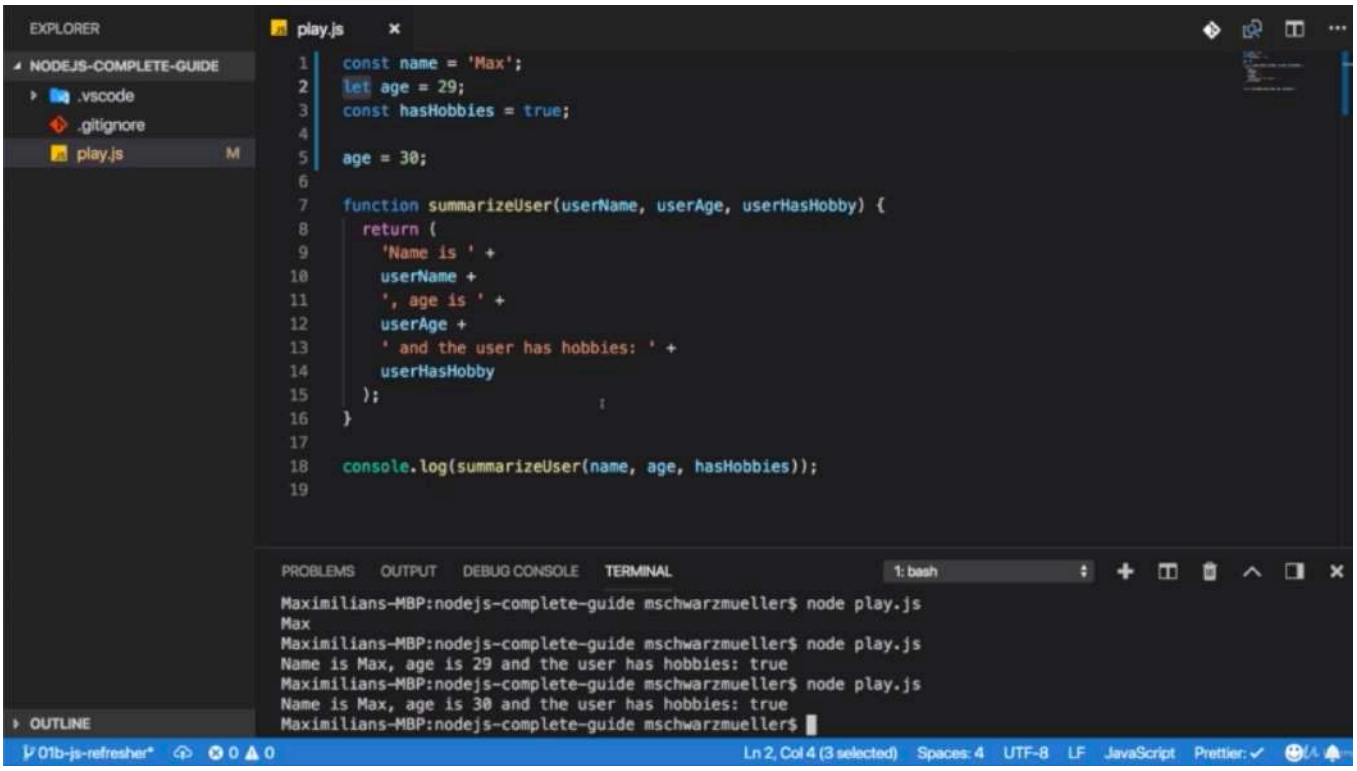
![](images/12-let-and-const-1.png)



```
let name = 'Max';
let age = 29;
let hasHobbies = true;

age = 30;

function summarizeUser(userName, userAge, userHasHobby) {
  return (
    'Name is ' +
    userName +
    ', age is ' +
    userAge +
    ' and the user has hobbies: ' +
    userHasHobby
  );
}

console.log(summarizeUser(name, age, hasHobbies));
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Max
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Name is Max, age is 29 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ 
```

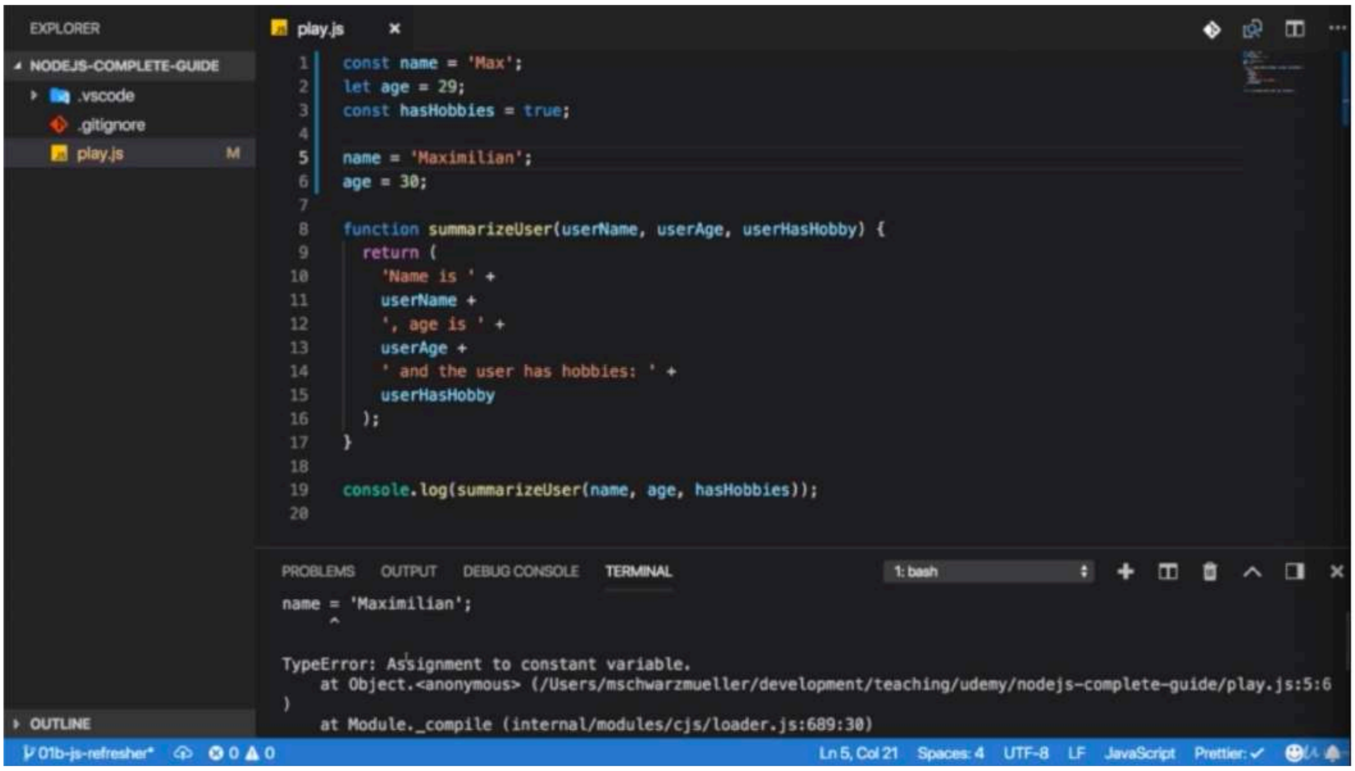
- the core reason for using 'let' is that we now also have another way of creating a variable and with that, i mean a variable which never changes which actually is the case for all 3 variables
![](images/12-let-and-const-2.png)



```
const name = 'Max';
let age = 29;
const hasHobbies = true;

age = 30;

function summarizeUser(userName, userAge, userHasHobby) {
  return (
    'Name is ' +
    userName +
    ', age is ' +
    userAge +
    ' and the user has hobbies: ' +
    userHasHobby
  );
}

console.log(summarizeUser(name, age, hasHobbies));
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Max
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Name is Max, age is 29 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Name is Max, age is 30 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- 'const' keyword makes clear that we never plan on chaning the value of name or hasHobbies. we do plan to change it of age. that's why we have let and that's the reason why we have 2 different keywords for creating variables,
![](images/12-let-and-const-3.png)



```
const name = 'Max';
let age = 29;
const hasHobbies = true;

name = 'Maximilian';
age = 30;

function summarizeUser(userName, userAge, userHasHobby) {
  return (
    'Name is ' +
    userName +
    ', age is ' +
    userAge +
    ' and the user has hobbies: ' +
    userHasHobby
  );
}

console.log(summarizeUser(name, age, hasHobbies));
```

```
name = 'Maximilian';
     ^

TypeError: Assignment to constant variable.
    at Object.<anonymous> (/Users/mschwarzmueller/development/teaching/udemy/nodejs-complete-guide/play.js:5:6
)
    at Module._compile (internal/modules/cjs/loader.js:689:30)
```

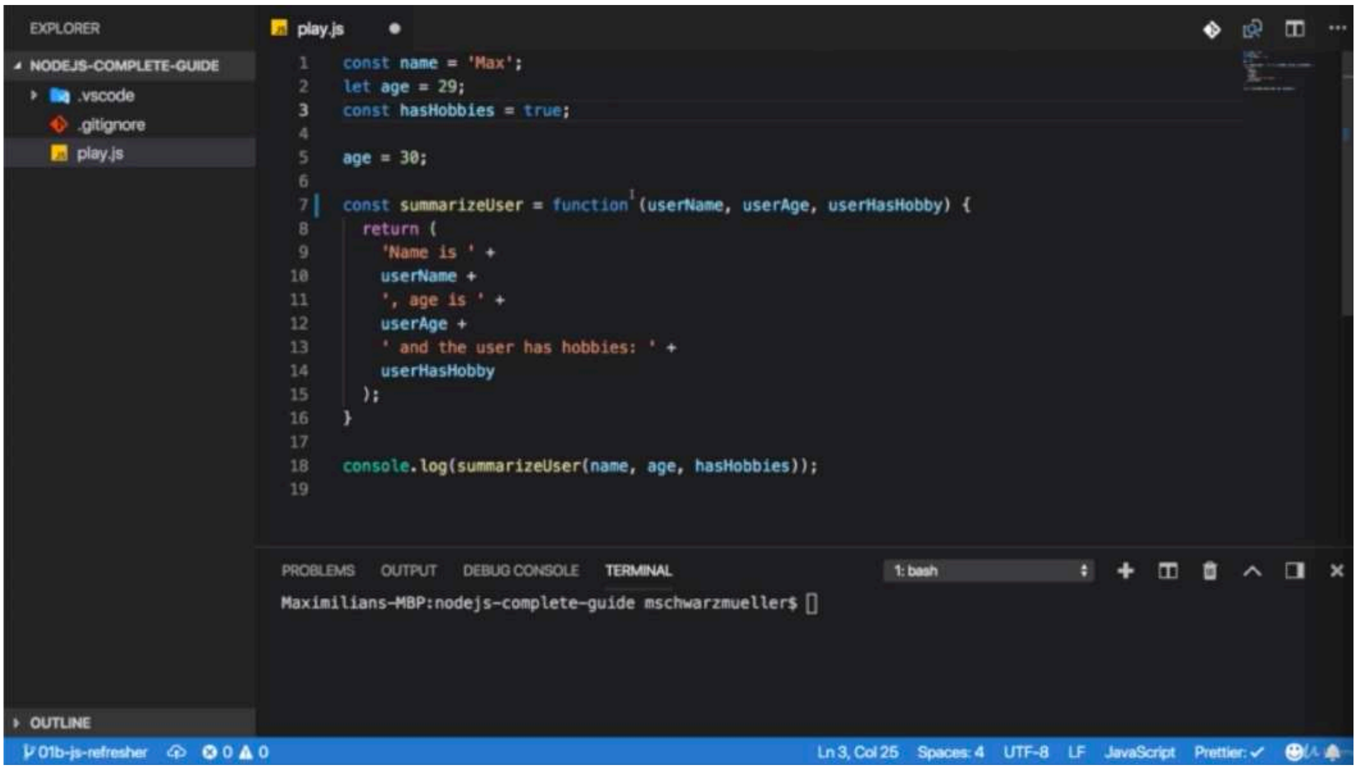
- if i try to assign to a constant variable, we get error.
- you wanna use const as often as possible to be as clear about what happens in your code as possible and if something should never change, make it a const so that you get an error if you do accidentally change it. so here i will revert this change.

```
const name = 'Max';
let age = 29;
const hasHobbies = true;

```

```
age = 30
name = 'kiwon'

function summarizeUser(userName, userAge, userHasHobby){
    return ('Name is '+
    userName +
    ', age is ' +
    userAge +
    ' and the user has hobbies: ' +
    userHasHobby
    )
}

console.log(summarizeUser(name, age, hasHobbies))
```

* Chapter 13: Understanding Arrow Functions

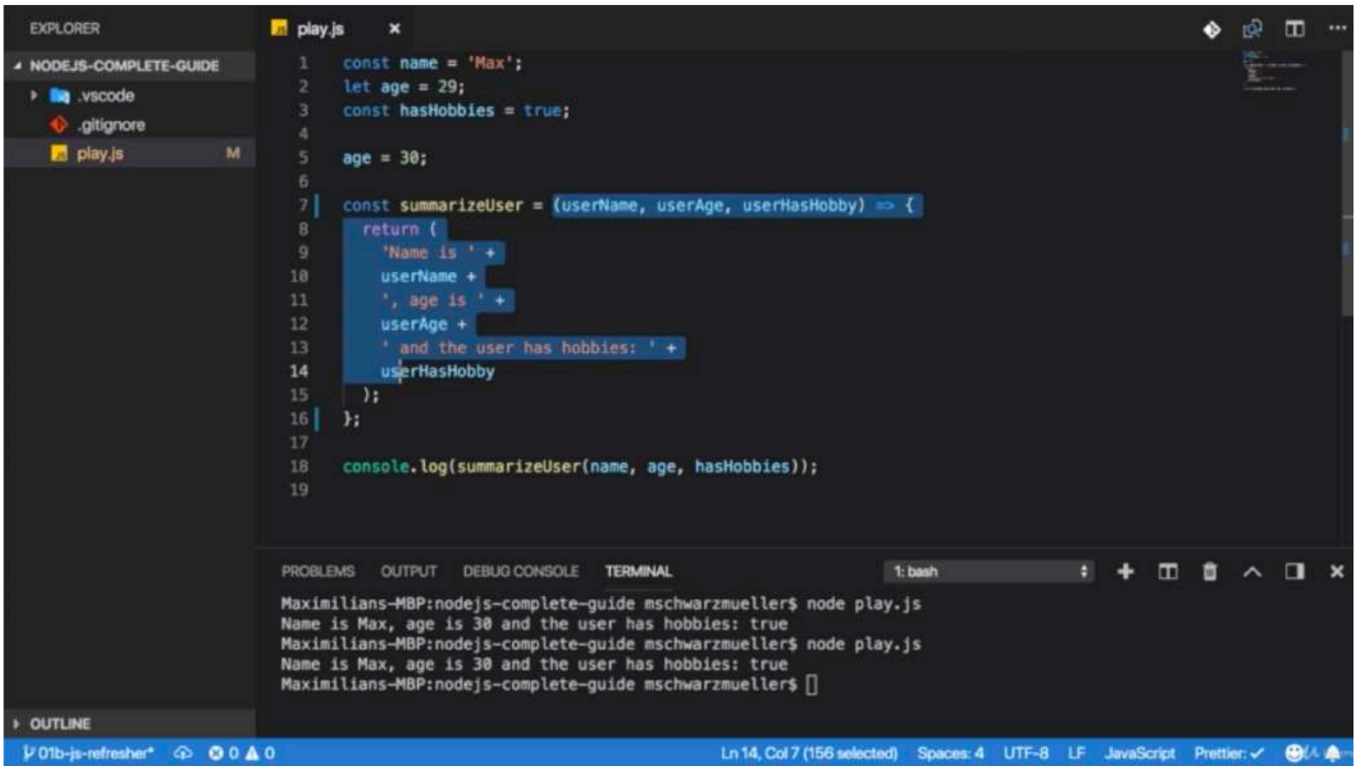
![](images/13-understanding-arrow-functions-1.png)


EXPLORER
NODEJS-COMPLETE-GUIDE
.vscode
.gitignore
play.js
OUTLINE
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: bash
01b-js-refresher
Ln 3, Col 25 Spaces: 4 UTF-8 LF JavaScript Prettier: ✓

```
const name = 'Max';
let age = 29;
const hasHobbies = true;

age = 30;

const summarizeUser = function(userName, userAge, userHasHobby) {
  return (
    'Name is ' +
    userName +
    ', age is ' +
    userAge +
    ' and the user has hobbies: ' +
    userHasHobby
  );
}

console.log(summarizeUser(name, age, hasHobbies));
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

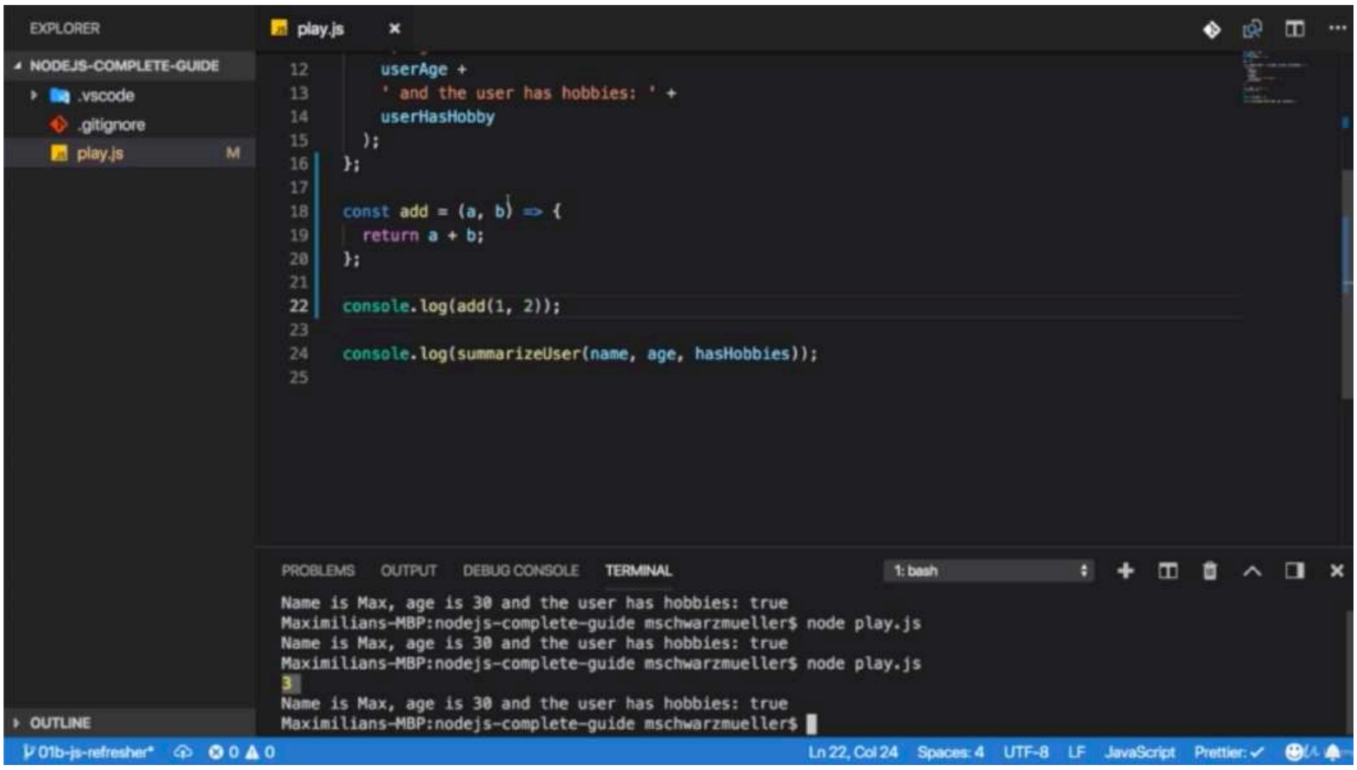
- the part on the right side is a so-called ‘anonymous function’ because we don’t set up a name after function. but we make it a named function implicitly by storing the anonymous function in that named constant. so we can always call that constant which holds a function as a value and we call the value with the syntax and therefore this is like a named function. this is a way of different way of defining a function.
![](images/13-understanding-arrow-functions-2.png)



```
const name = 'Max';
let age = 29;
const hasHobbies = true;

age = 30;

const summarizeUser = (userName, userAge, userHasHobby) => {
  return (
    'Name is ' +
    userName +
    ', age is ' +
    userAge +
    ' and the user has hobbies: ' +
    userHasHobby
  );
};

console.log(summarizeUser(name, age, hasHobbies));
```

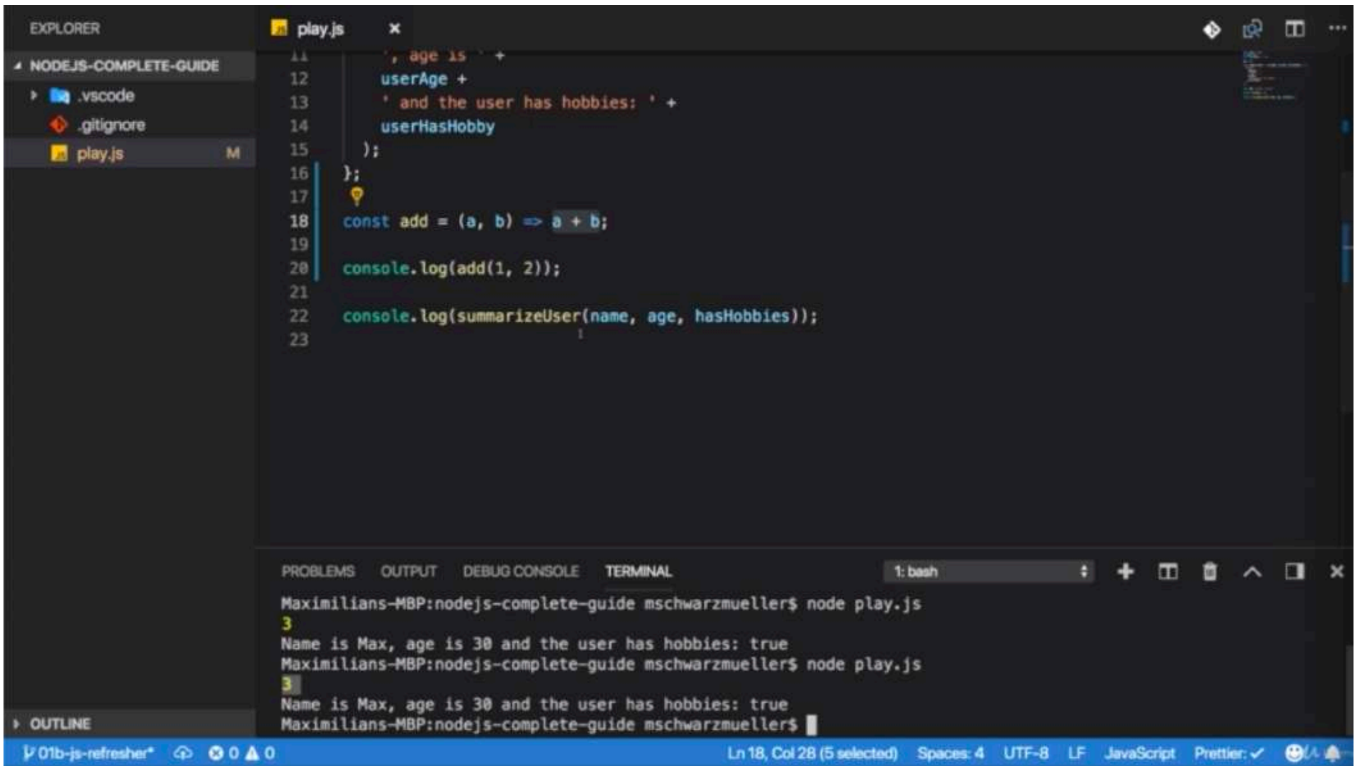
```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Name is Max, age is 30 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Name is Max, age is 30 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- now we remove the ‘function’ keyword and instead we add an arrow between the argument list and the curly braces and this arrow is simply an equal sign and an greater than sign. this also create a function. it’s a bit shorter since we save the ‘function’ keyword and it runs in the same way as this function ran before
- why would we use the syntax except for the reason that it’s a bit shorter? there’s one key difference regarding the ’this’ keyword which javascript knows.
![](images/13-understanding-arrow-functions-3.png)
![](images/13-understanding-arrow-functions-4.png)



```
    userAge +
    ' and the user has hobbies: ' +
    userHasHobby
  );
};

const add = (a, b) => {
  return a + b;
};

console.log(add(1, 2));

console.log(summarizeUser(name, age, hasHobbies));
```

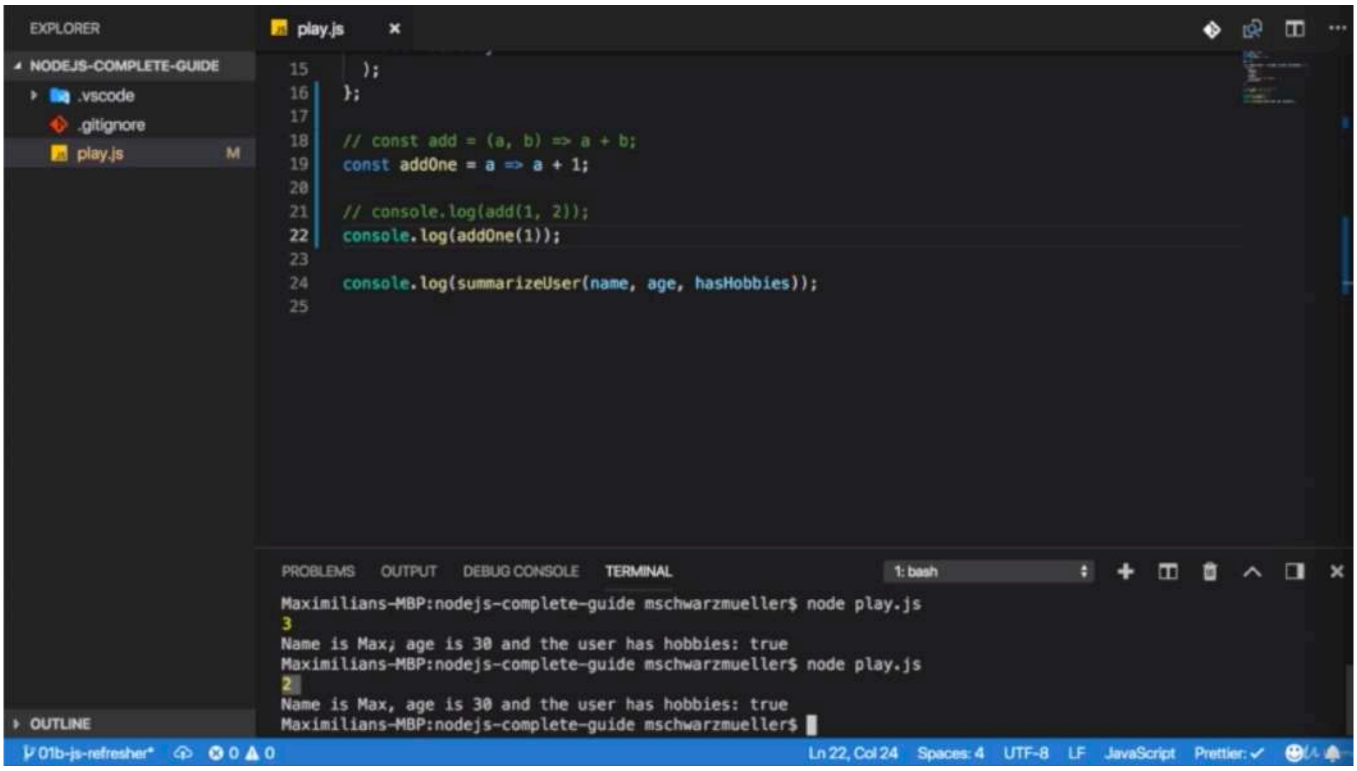
```
Name is Max, age is 30 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Name is Max, age is 30 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
3
Name is Max, age is 30 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```



```
    ', age is ' +
    userAge +
    ' and the user has hobbies: ' +
    userHasHobby
  );
};

const add = (a, b) => a + b;

console.log(add(1, 2));

console.log(summarizeUser(name, age, hasHobbies));
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
3
Name is Max, age is 30 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
3
Name is Max, age is 30 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

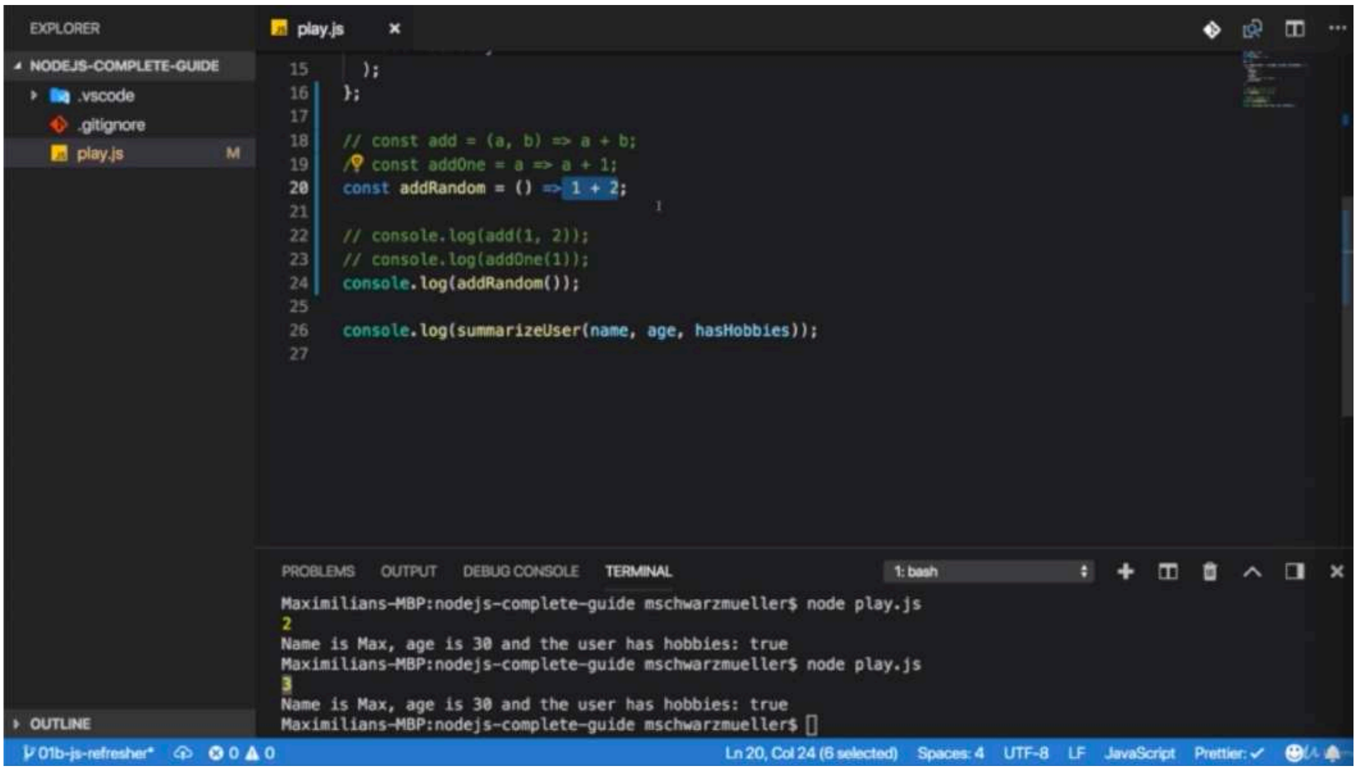
- this simply is the same syntax as before with the curly braces and with return and this function will now always return the result of this statement
![](images/13-understanding-arrow-functions-5.png)



```
  );
};

// const add = (a, b) => a + b;
const addOne = a => a + 1;

// console.log(add(1, 2));
console.log(addOne(1));

console.log(summarizeUser(name, age, hasHobbies));
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
3
Name is Max, age is 30 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
2
Name is Max, age is 30 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- if you only have 1 argument and that’s the case really for that case only, if you have only one argument only, then you can just have the argument name without parentheses and it will work just as it will work with parentheses
![](images/13-understanding-arrow-functions-6.png)


EXPLORER
play.js
NODEJS-COMPLETE-GUIDE
.vscode
.gitignore
play.js M
OUTLINE
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: bash
01b-js-refresher*
Ln 20, Col 24 (6 selected) Spaces: 4 UTF-8 LF JavaScript Prettier: ✓

```javascript
  );
};

// const add = (a, b) => a + b;
// const addOne = a => a + 1;
const addRandom = () => 1 + 2;

// console.log(add(1, 2));
// console.log(addOne(1));
console.log(addRandom());

console.log(summarizeUser(name, age, hasHobbies));
```

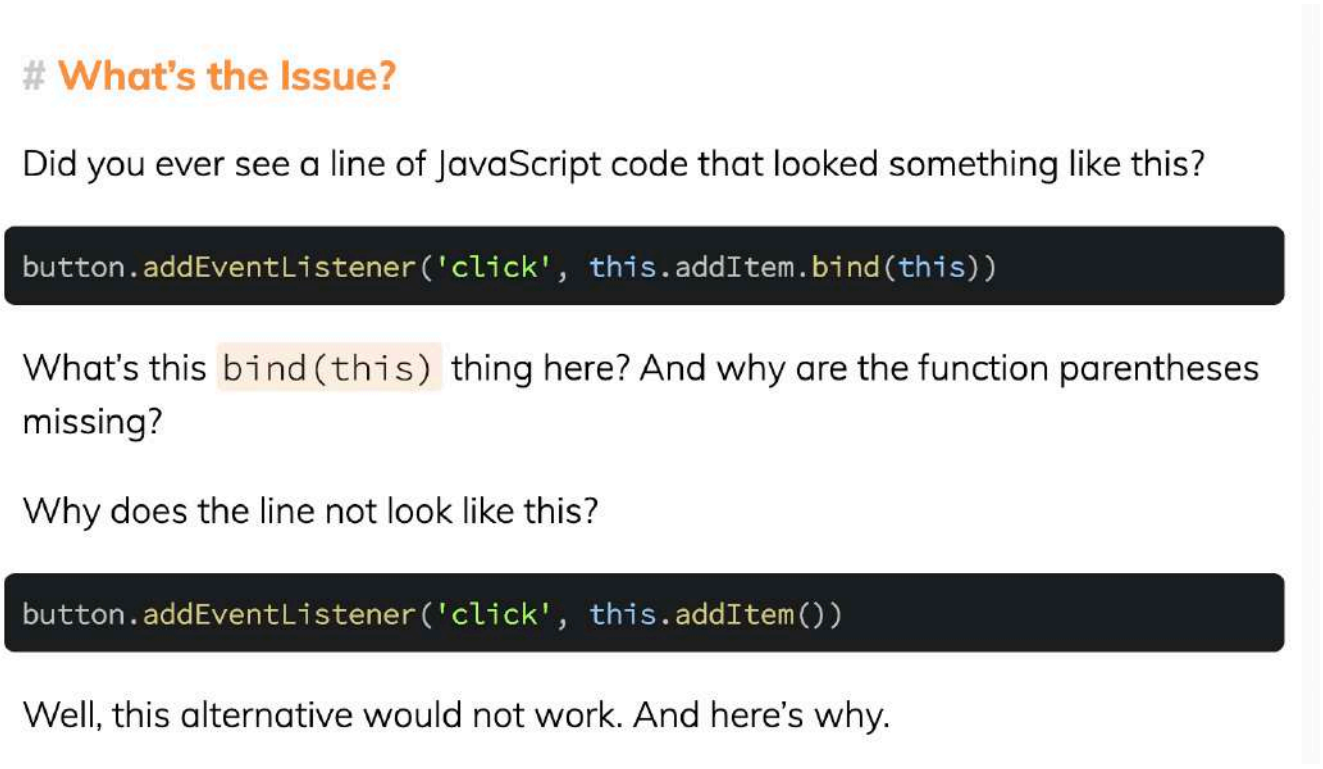
```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
2
Name is Max, age is 30 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
3
Name is Max, age is 30 and the user has hobbies: true
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- if you have an arrow function with no arguments at random, then you have to specify an empty pair and then you can have your code there which obviously uses no arguments because that's exactly what i wanna show here. you can't have just whitespace.

```javascript
const name = 'Max';
let age = 29;
const hasHobbies = true;

age = 30;

const summarizeUser = (userName, userAge, userHasHobby) => {
  return (
    'Name is ' +
    userName +
    ', age is ' +
    userAge +
    ' and the user has hobbies: ' +
    userHasHobby
  );
};

// const add = (a, b) => a + b;
// const addOne = a => a + 1;
const addRandom = () => 1 + 2;

// console.log(add(1, 2));
// console.log(addOne(1));
console.log(addRandom());

console.log(summarizeUser(name, age, hasHobbies));
```

![](images/13-understanding-arrow-functions-7.png)
![](images/13-understanding-arrow-functions-8.png)
![](images/13-understanding-arrow-functions-9.png)
![](images/13-understanding-arrow-functions-10.png)

![](images/13-understanding-arrow-functions-11.png)
![](images/13-understanding-arrow-functions-12.png)
![](images/13-understanding-arrow-functions-13.png)
![](images/13-understanding-arrow-functions-14.png)
![](images/13-understanding-arrow-functions-15.png)
![](images/13-understanding-arrow-functions-16.png)



What's the Issue?

Did you ever see a line of JavaScript code that looked something like this?

```
button.addEventListener('click', this.addItem.bind(this))
```

What's this `bind(this)` thing here? And why are the function parentheses missing?

Why does the line not look like this?

```
button.addEventListener('click', this.addItem())
```

Well, this alternative would not work. And here's why.

Calling Functions & Using Function References

Obviously, you call a function like this in JavaScript:

```
function someFunction() {
  // do something here ...
}
someFunction()
```

And inside an object/ class, you call a method in a similar way:

```
class MyClass {
  constructor() {
    this.myMethod()
  }

  myMethod() {
    // do something here
  }
}
```

Using the above code will execute the methods immediately when the code is run for the first time. In the class, the method `myMethod` is being executed when the constructor is called - i.e. when the class is instantiated.

```
ew MyClass() // this will trigger the constructor and hence call myMethod()
```

Sometimes, you don't want to execute a function/ method immediately though.

Consider an event listener on a button:

```
function someFunction() { ... }
const button = document.querySelector('button');

button.addEventListener('click', someFunction());
```

In this snippet, `someFunction` would actually **not** wait for the click to occur but instead also execute right when the code is first parsed/ executed.

That is not what we want though. We just want to "tell JavaScript/ the Browser" that it should execute `someFunction` for us when the button is clicked. This will also ensure that the function can run multiple times - one time for every button click.

What do you do when you want to make sure your friend can visit your parents once he's done with his work for the day?

You don't send him there immediately - instead you tell him where your parents live. This allows your friend to visit them once he got the time. You basically give your friend the address of your parents instead of taking him with you.

The same concept can be used in JavaScript. You can "give JavaScript/ the Browser" the address of something (=> a function) instead of executing it manually right away.

This is done by passing a so called **"reference"**.

For the event listener, the following code passes a reference to the "to-be-executed" function to the event listener (i.e. to the button in this case).

```
function someFunction() { ... };

const button = ...;

button.addEventListener('click', someFunction);
```

Please note, that the parentheses **are missing** after `someFunction`. Therefore, we don't call the function - instead we just pass a pointer to the function (a so called reference) to the event listener (and hence to the button object).

In the context of a JavaScript class, the code looks pretty much the same:

```
class MyClass {
  constructor() {
    const button = ...;
    button.addEventListener('click', this.myMethod);
  }

  myMethod() { ... }
}
```

The `this` keyword is important here though. It basically points at the object that is created based on the class. And since `myMethod` is a method of the class/ object, it can only be accessed via `this`.

That's how you pass a reference to a function instead of calling it immediately. And that's why you would use this syntax without the parentheses.

When "this" Behaves Strangely

`this` is required to access class methods or properties from anywhere inside of that class/ object.

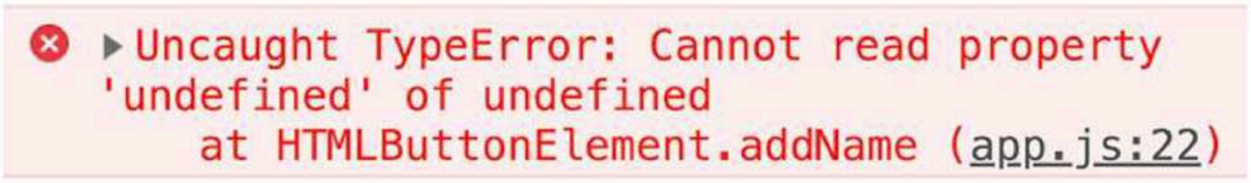
> In case classes are brand-new to you, consider taking my ES6 course as I dive deeper into classes there.

Let's move to a real example - one where `this` will actually lead to a strange behavior.

```
class NameGenerator {
  constructor() {
    const btn = document.querySelector('button')
    this.names = ['Max', 'Manu', 'Anna']
    this.currentName = 0
    btn.addEventListener('click', this.addName)
  }

  addName() {
    const name = new NameField(this.names[this.currentName])
    this.currentName++
    if (this.currentName >= this.names.length) {
      this.currentName = 0
    }
  }
}
```

Let's not worry about what `new NameField(...)` does - you can watch the video (at the top of this page) to see the full example. It basically just renders a new `<li>` with the name as text into the DOM.

But let's worry about whether that succeeds or not. Because we'll actually get an error:



```
▶Uncaught TypeError: Cannot read property
'undefined' of undefined
    at HTMLButtonElement.addName (app.js:22)
```

This line is causing the error:

```
const name = new NameField(this.names[this.currentName])
```

Somehow, accessing `this.names` and `this.currentName` fails here.

But why? Doesn't `this` refer to the object/ class?

Well, it actually doesn't. `this` is not defined to refer to the object that encloses it when you write your code.

Instead, `this` refers to "whoever called the code in which it's being used".

And in this case, the `button` is responsible for executing `addName`.

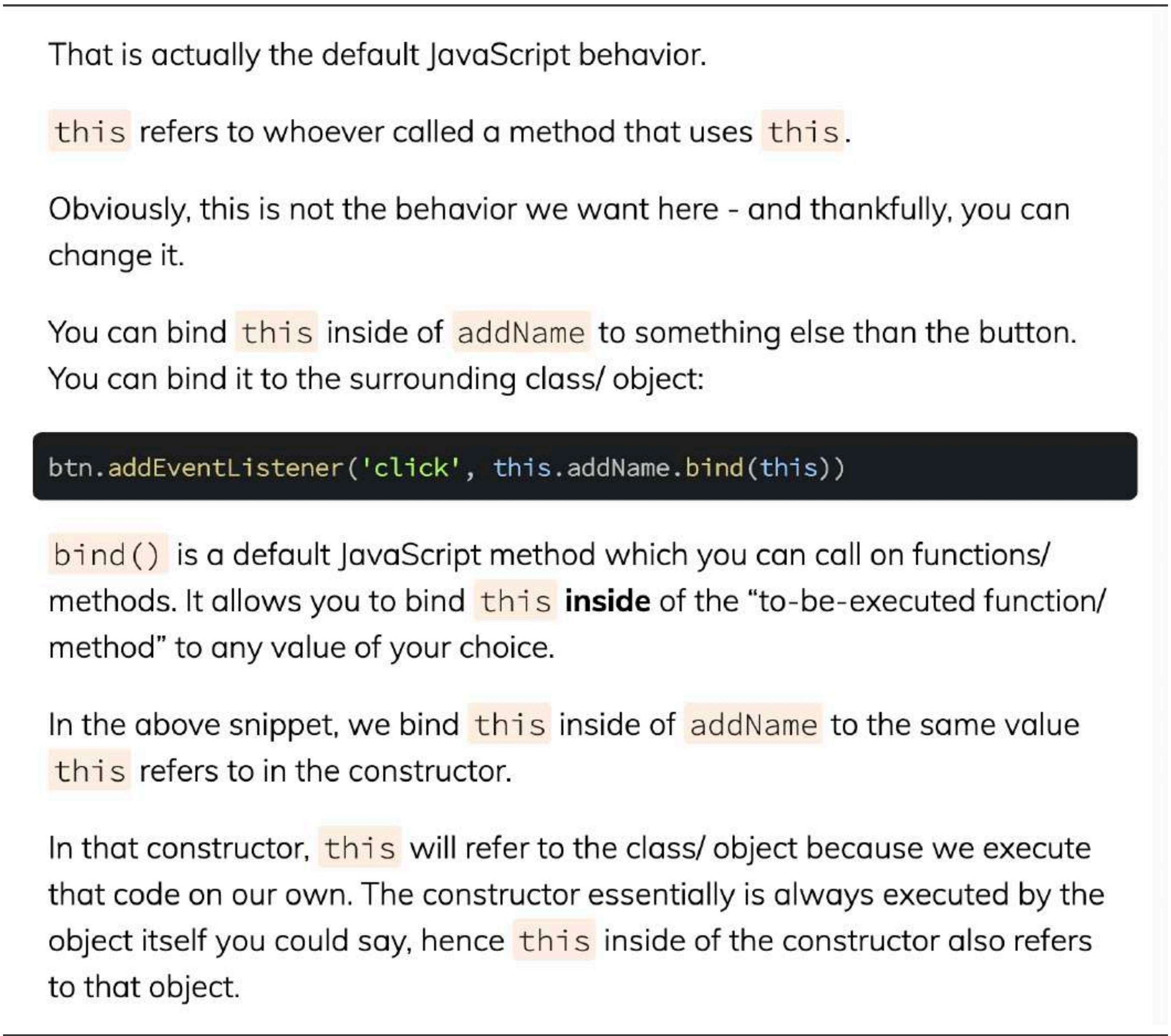
We can see that, if we log the value of `this` inside of `addName`:

```
addName() {
  console.log(this);
  ...
}
```

This will print:

```
 <button>Add Name</button>                              app.js:21
```

So `this` is now referring to the `<button>` element to which we attached the `click` event listener.



That is actually the default JavaScript behavior.

`this` refers to whoever called a method that uses `this`.

Obviously, this is not the behavior we want here - and thankfully, you can change it.

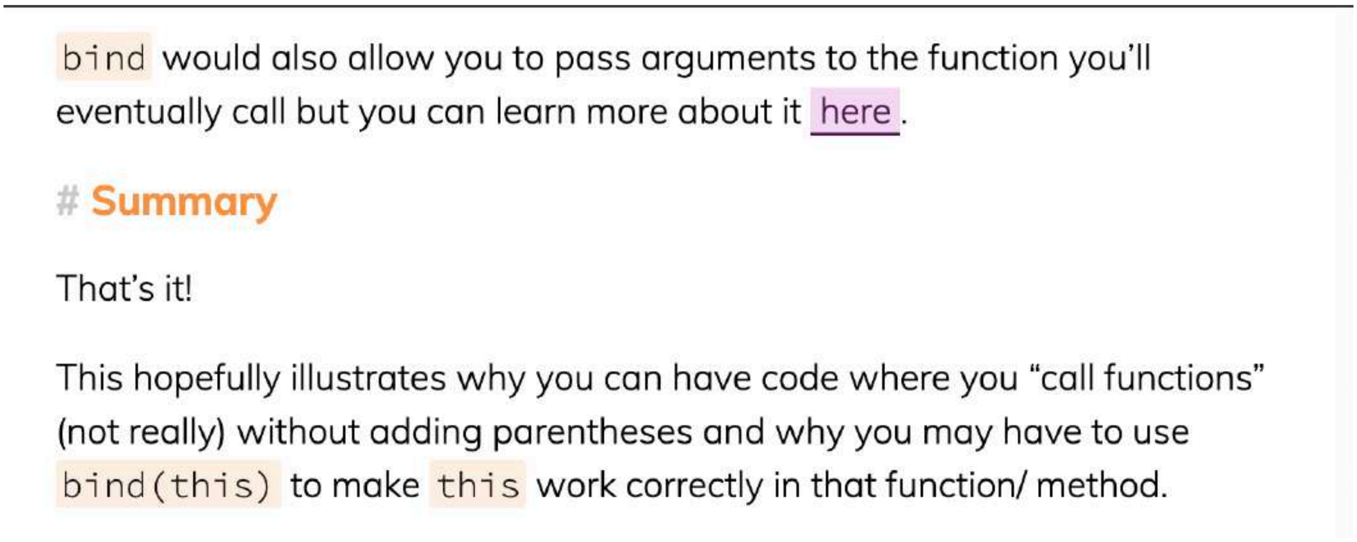
You can bind `this` inside of `addName` to something else than the button. You can bind it to the surrounding class/ object:

```
btn.addEventListener('click', this.addName.bind(this))
```

`bind()` is a default JavaScript method which you can call on functions/ methods. It allows you to bind `this` **inside** of the "to-be-executed function/ method" to any value of your choice.

In the above snippet, we bind `this` inside of `addName` to the same value `this` refers to in the constructor.

In that constructor, `this` will refer to the class/ object because we execute that code on our own. The constructor essentially is always executed by the object itself you could say, hence `this` inside of the constructor also refers to that object.



`bind` would also allow you to pass arguments to the function you'll eventually call but you can learn more about it here.

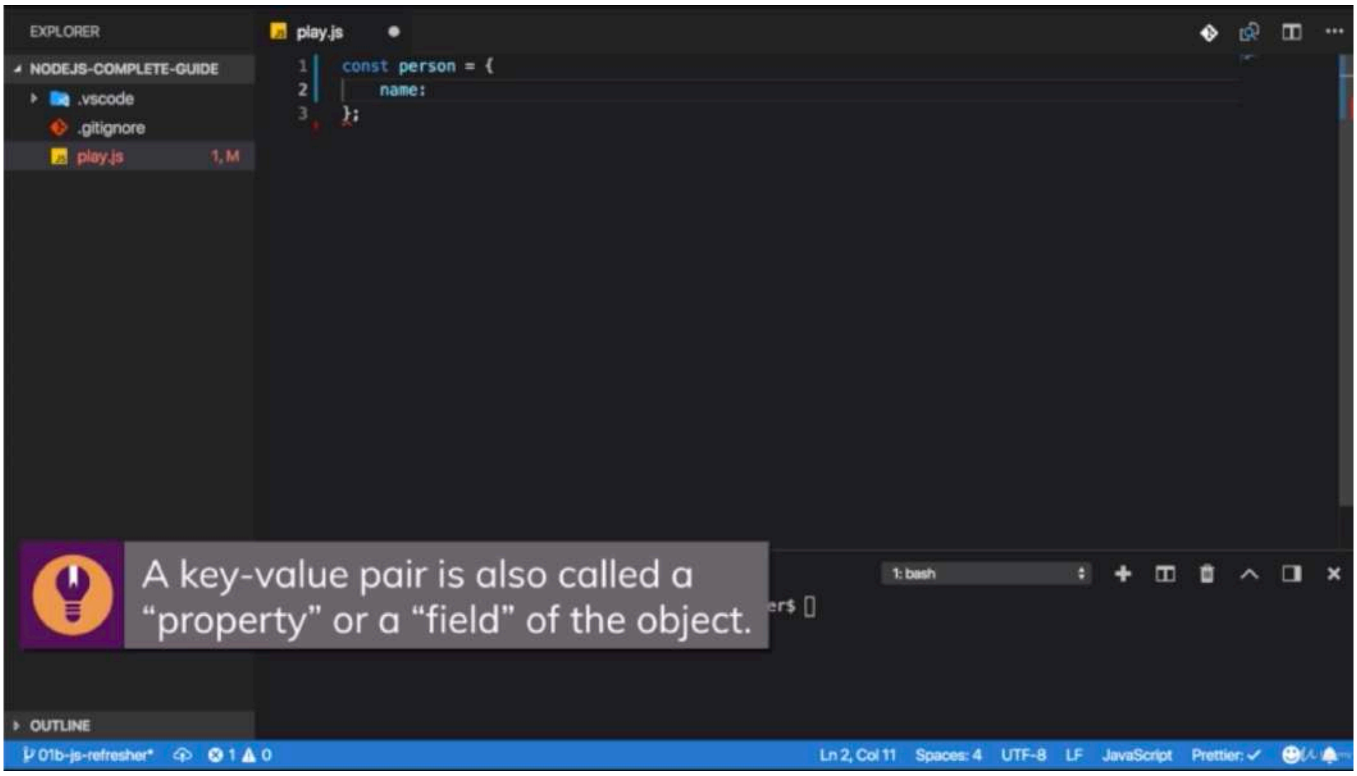
Summary

That's it!

This hopefully illustrates why you can have code where you "call functions" (not really) without adding parentheses and why you may have to use `bind(this)` to make `this` work correctly in that function/ method.

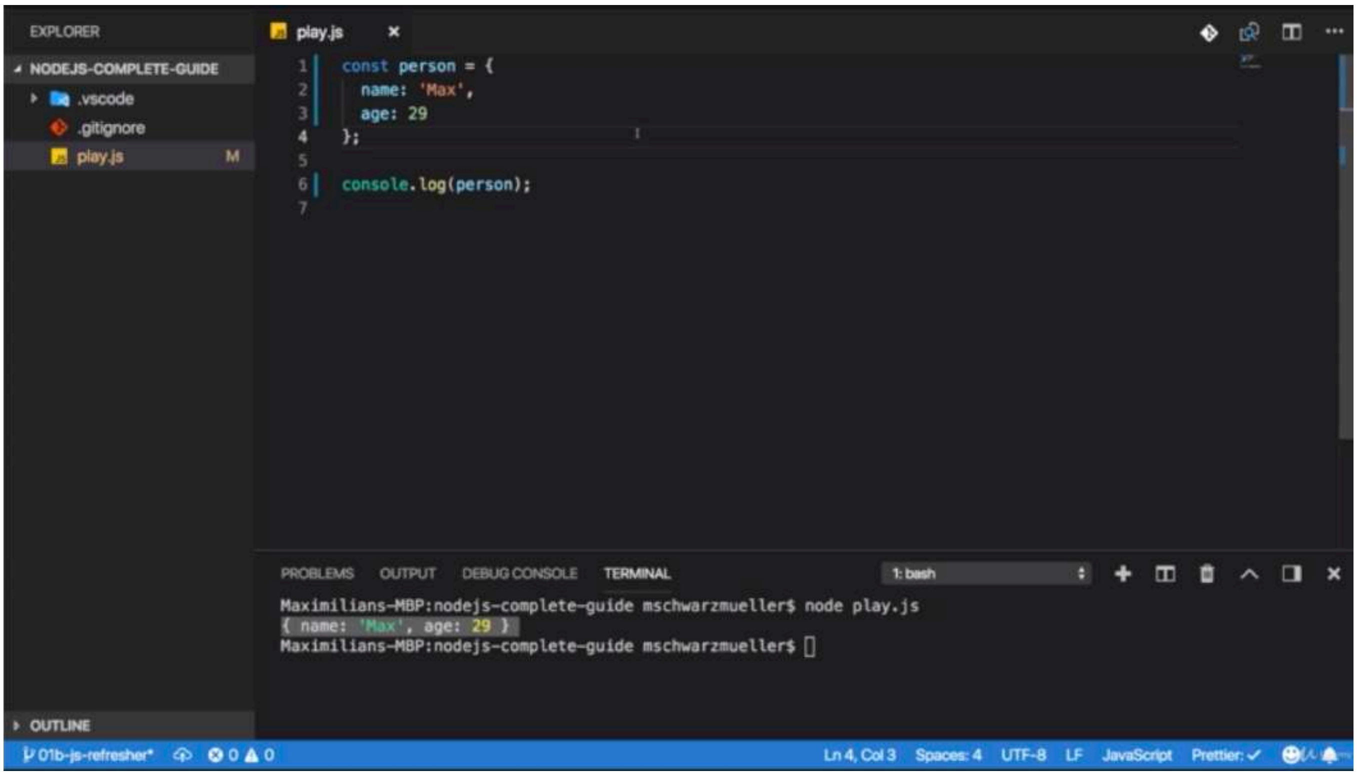
* Chapter 14: Working With Objects, Properties & Methods

![](images/14-working-with-objects-properties-methods-1.png)
![](images/14-working-with-objects-properties-methods-2.png)



```
const person = {
    name:
};
```

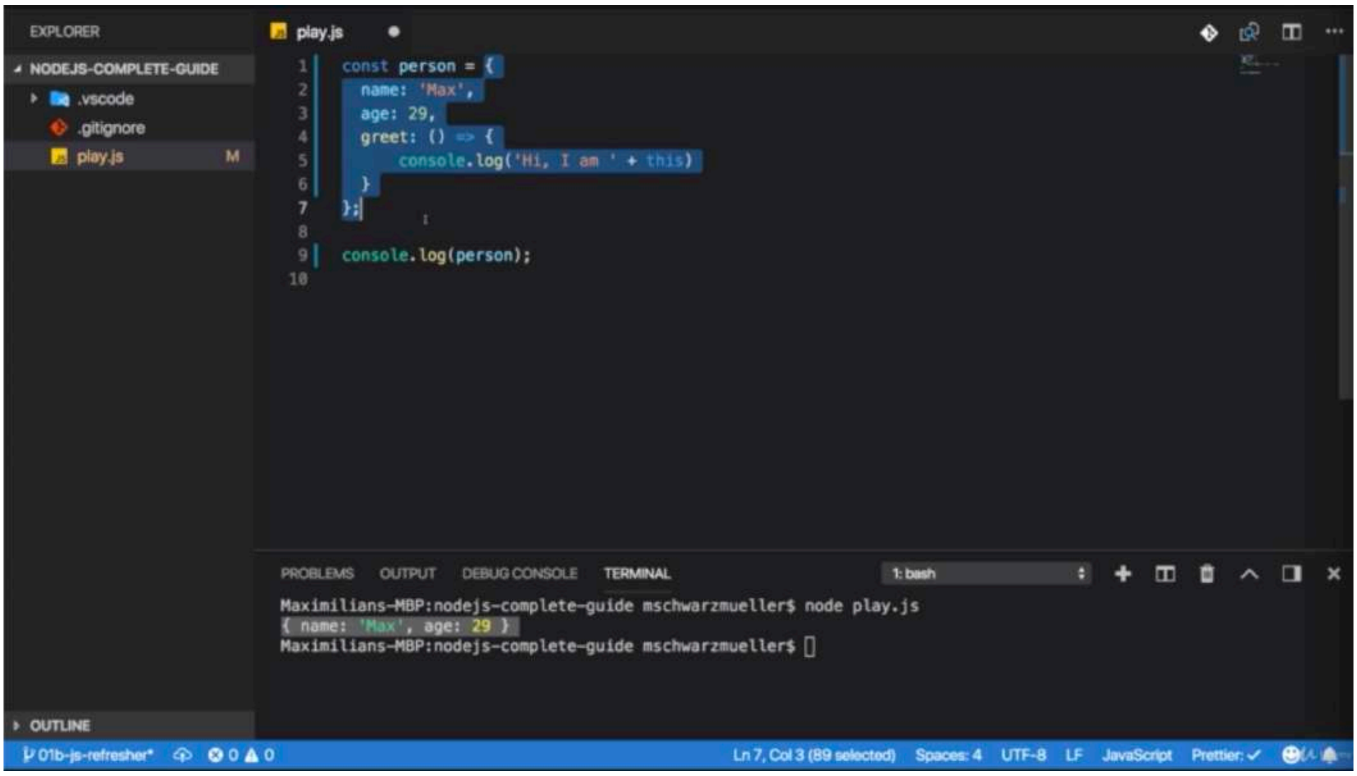
A key-value pair is also called a "property" or a "field" of the object.


EXPLORER
NODEJS-COMPLETE-GUIDE
.vscode
.gitignore
play.js
play.js
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: bash
OUTLINE
01b-js-refresher*
Ln 4, Col 3 Spaces: 4 UTF-8 LF JavaScript Prettier: ✓

```
const person = {
  name: 'Max',
  age: 29
};

console.log(person);
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
{ name: 'Max', age: 29 }
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

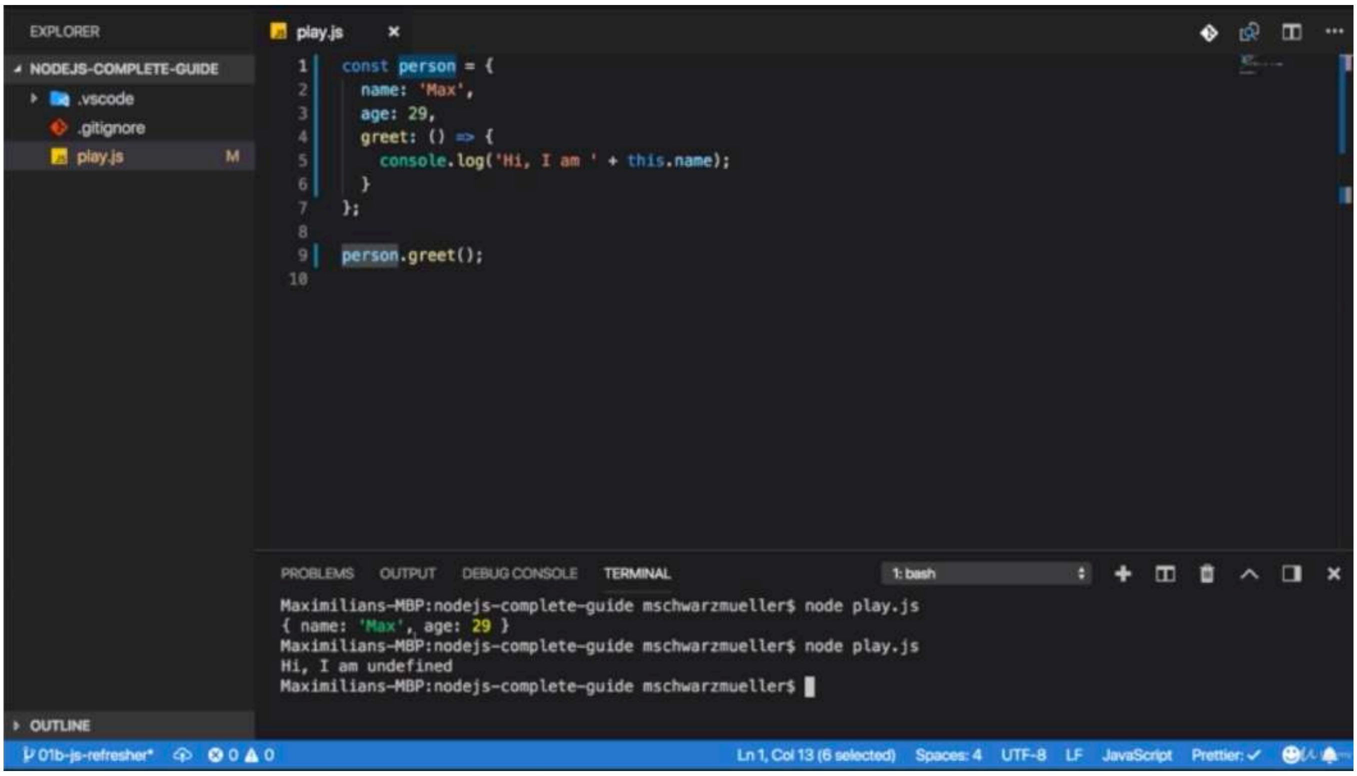
- objects allow us to group data together. you can also not just have variables in there so to say but you can also have functions in there.
![](images/14-working-with-objects-properties-methods-3.png)



```
const person = {
  name: 'Max',
  age: 29,
  greet: () => {
      console.log('Hi, I am ' + this)
  }
};

console.log(person);
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
{ name: 'Max', age: 29 }
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

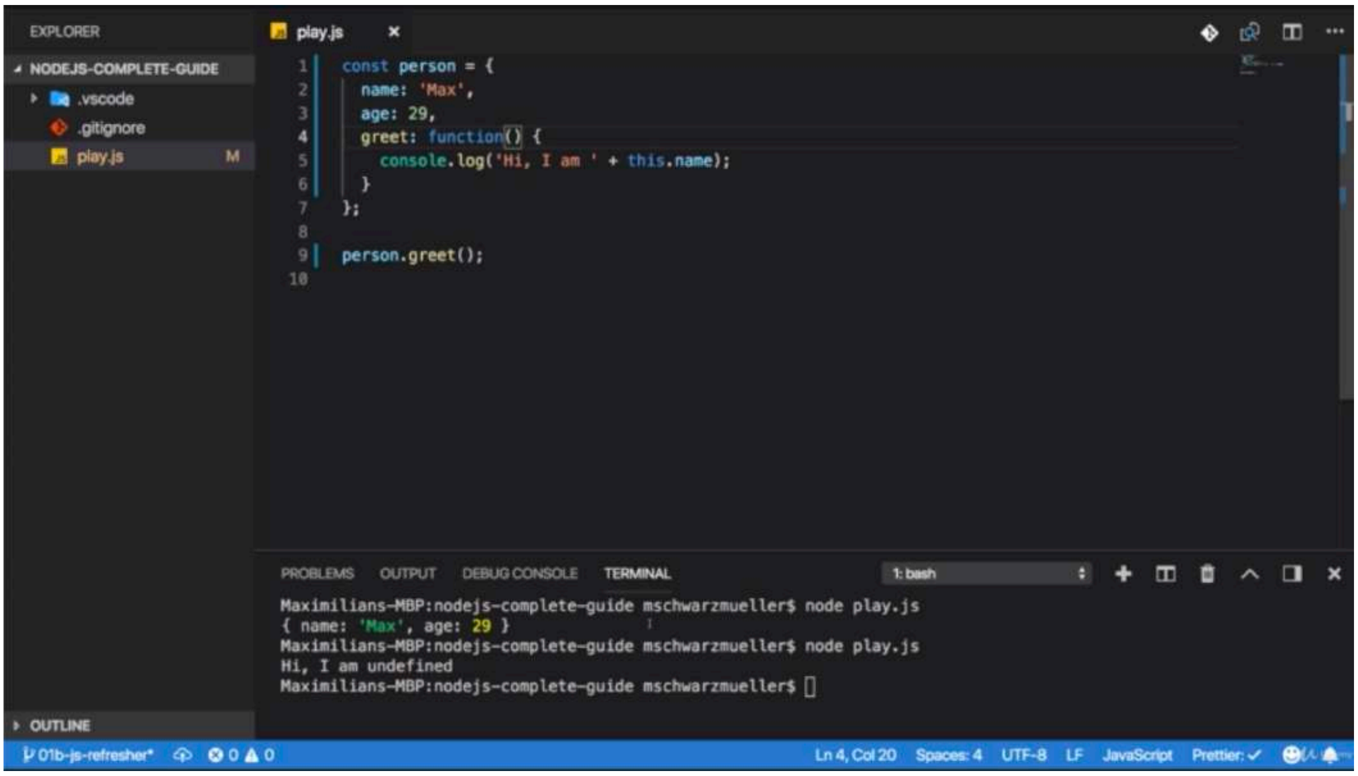
- this would now be accessed with the 'this' keyword, instead of an object you have to use this. 'this' keyword refers to the surrounding object. so dot to acess properties or methods, so variables or functions inside of that object. so we could now use this name.
![](images/14-working-with-objects-properties-methods-4.png)



```
const person = {
  name: 'Max',
  age: 29,
  greet: () => {
    console.log('Hi, I am ' + this.name);
  }
};

person.greet();
```

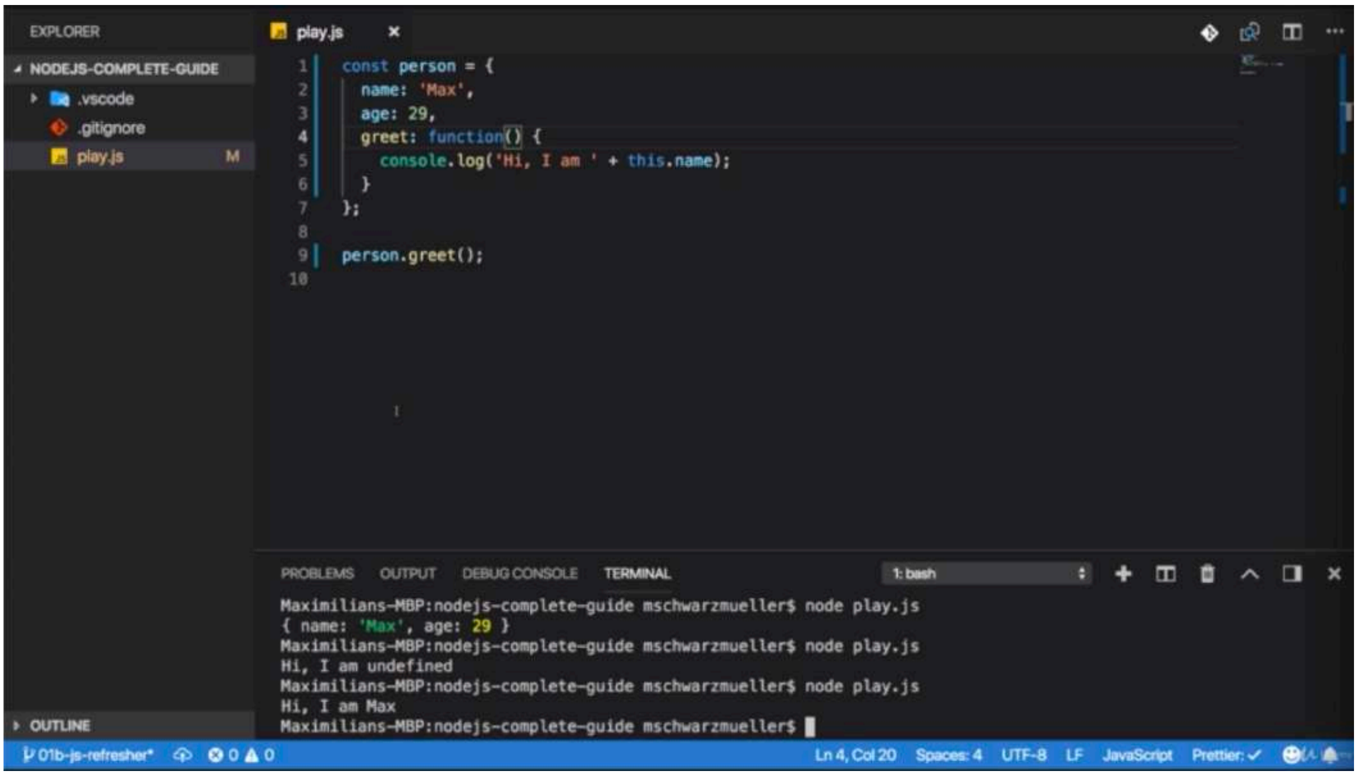
```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
{ name: 'Max', age: 29 }
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Hi, I am undefined
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- the reason for the undefined is arrow functions. 'this' keyword now refers to the global scope, to the global node runtime scope and not to this object
- 'this' keyword in this photo refers 'window' object which is global object, not the object which have the method calling 'this' keyword in arrow function.
![](images/14-working-with-objects-properties-methods-5.png)
![](images/14-working-with-objects-properties-methods-6.png)



```javascript
const person = {
  name: 'Max',
  age: 29,
  greet: function() {
    console.log('Hi, I am ' + this.name);
  }
};

person.greet();
```

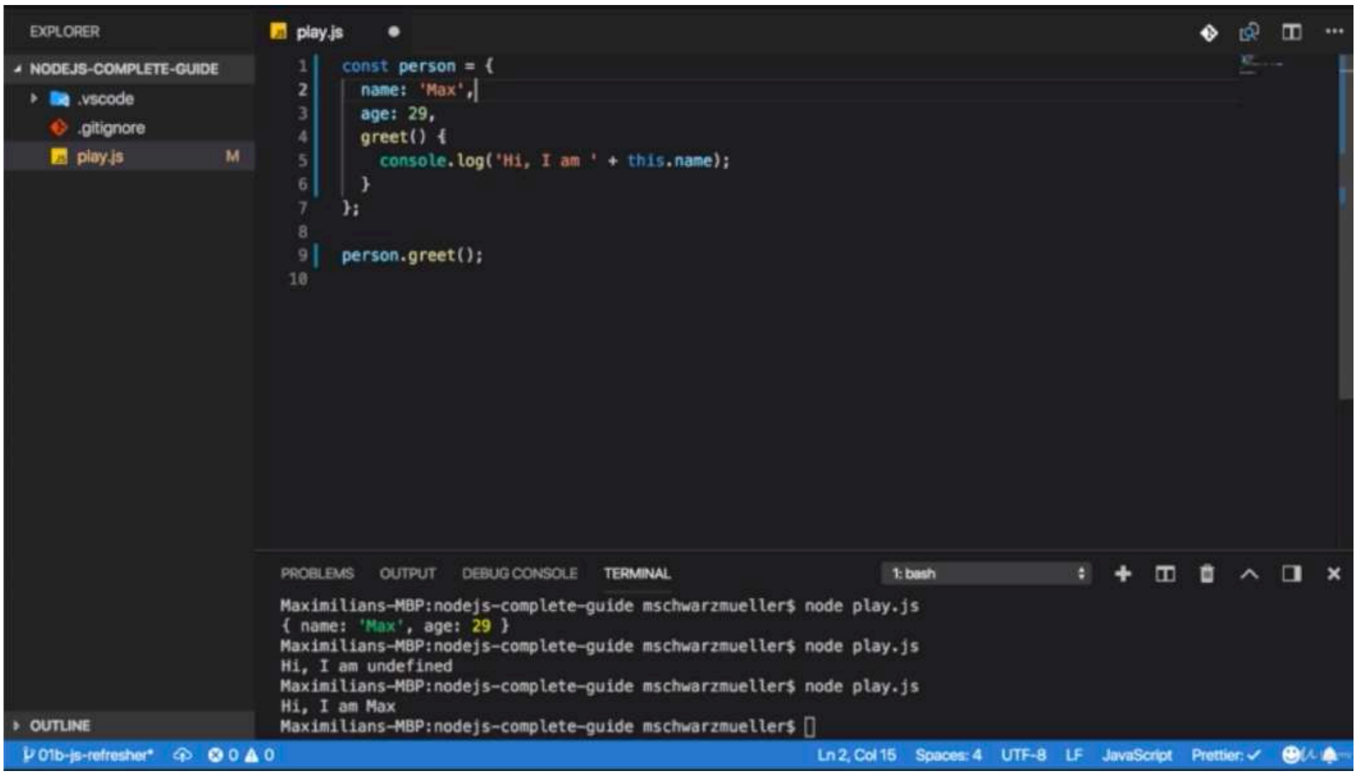
```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
{ name: 'Max', age: 29 }
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Hi, I am undefined
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```



```javascript
const person = {
  name: 'Max',
  age: 29,
  greet: function() {
    console.log('Hi, I am ' + this.name);
  }
};

person.greet();
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
{ name: 'Max', age: 29 }
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Hi, I am undefined
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Hi, I am Max
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- to have it refer to object, we either have to use the old school function, like this.
![](images/14-working-with-objects-properties-methods-7.png)



```
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

person.greet();
```

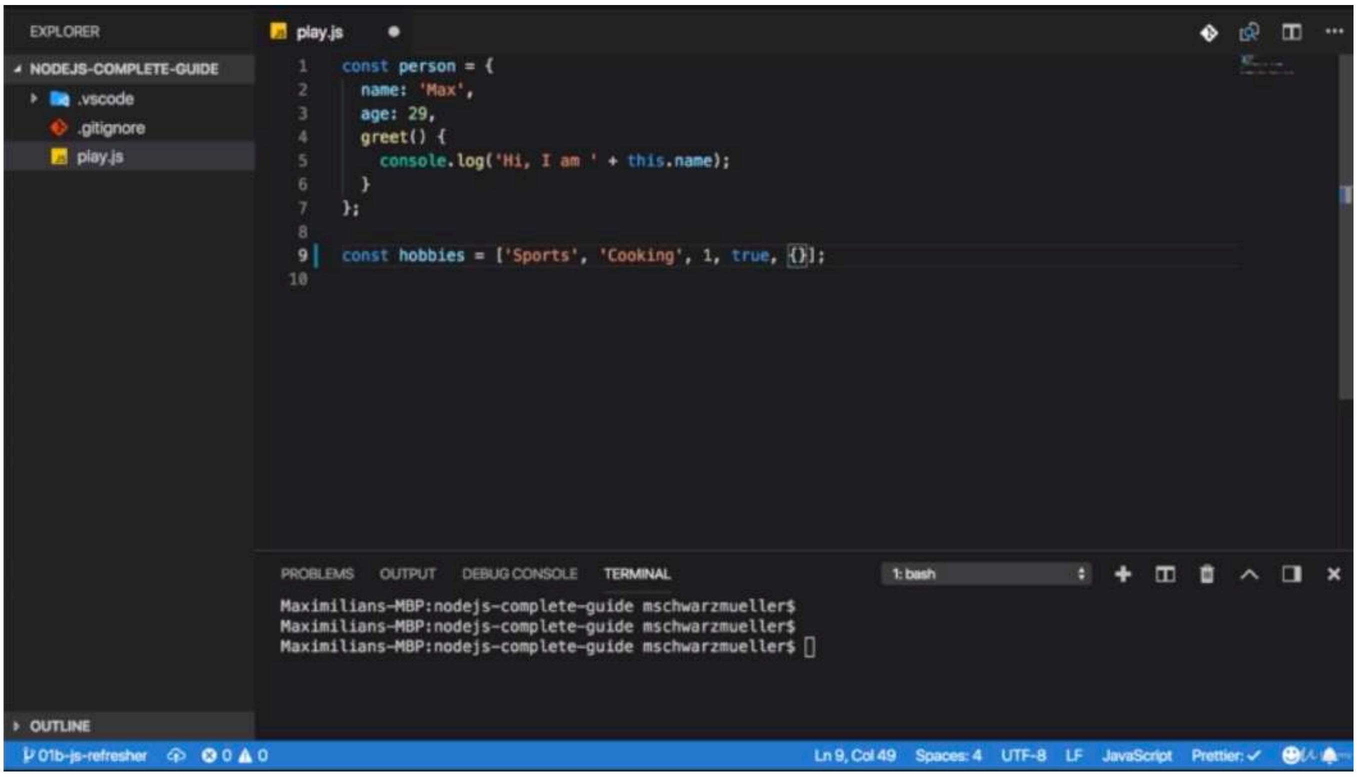
```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
{ name: 'Max', age: 29 }
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Hi, I am undefined
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Hi, I am Max
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- or use like that. omit the colon, add the parentheses after the key name and then without a function keyword or anything like that, you add your function body.

```
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

person.greet();

```

* Chapter 15: Array & Array Methods

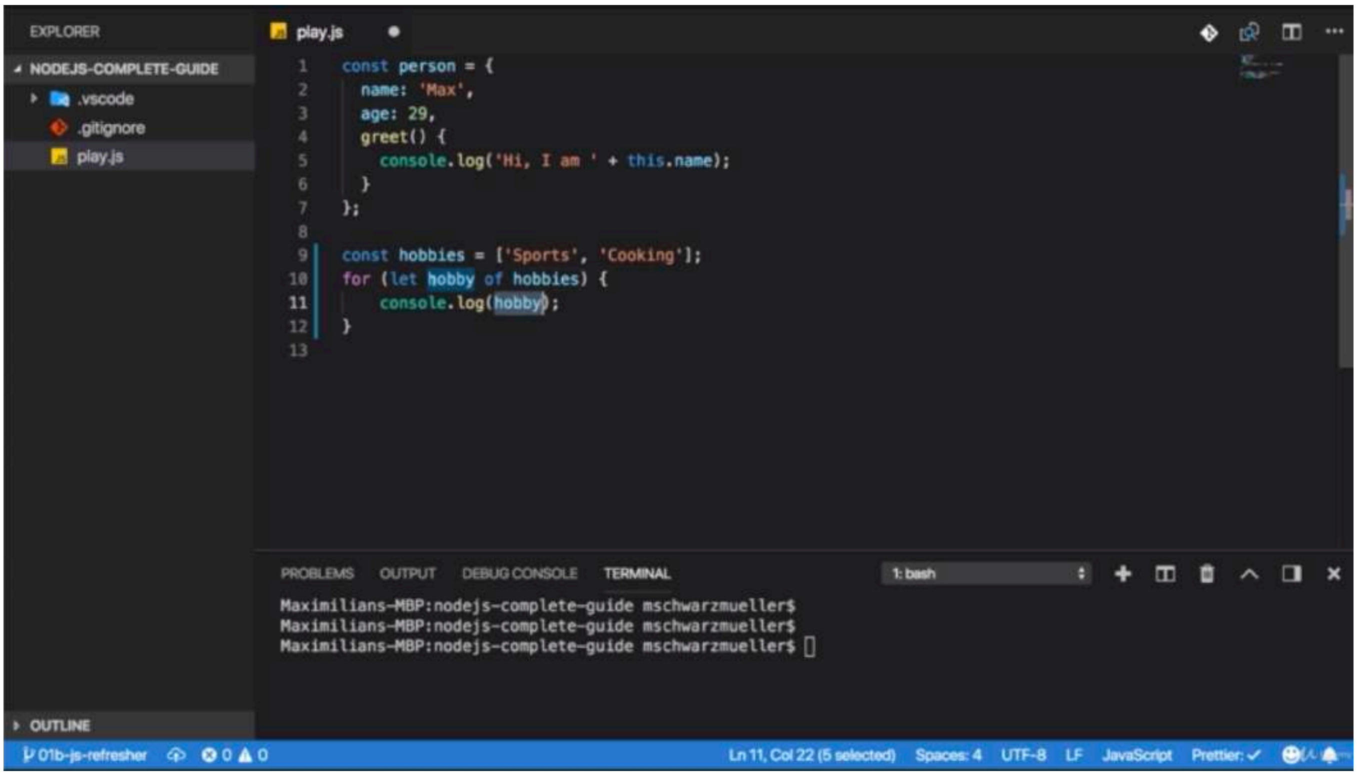
![](images/15-array-and-array-methods-1.png)


EXPLORER
NODEJS-COMPLETE-GUIDE
.vscode
.gitignore
play.js
OUTLINE
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: bash
01b-js-refresher
Ln 9, Col 49 Spaces: 4 UTF-8 LF JavaScript Prettier: ✓

```javascript
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const hobbies = ['Sports', 'Cooking', 1, true, {}];
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

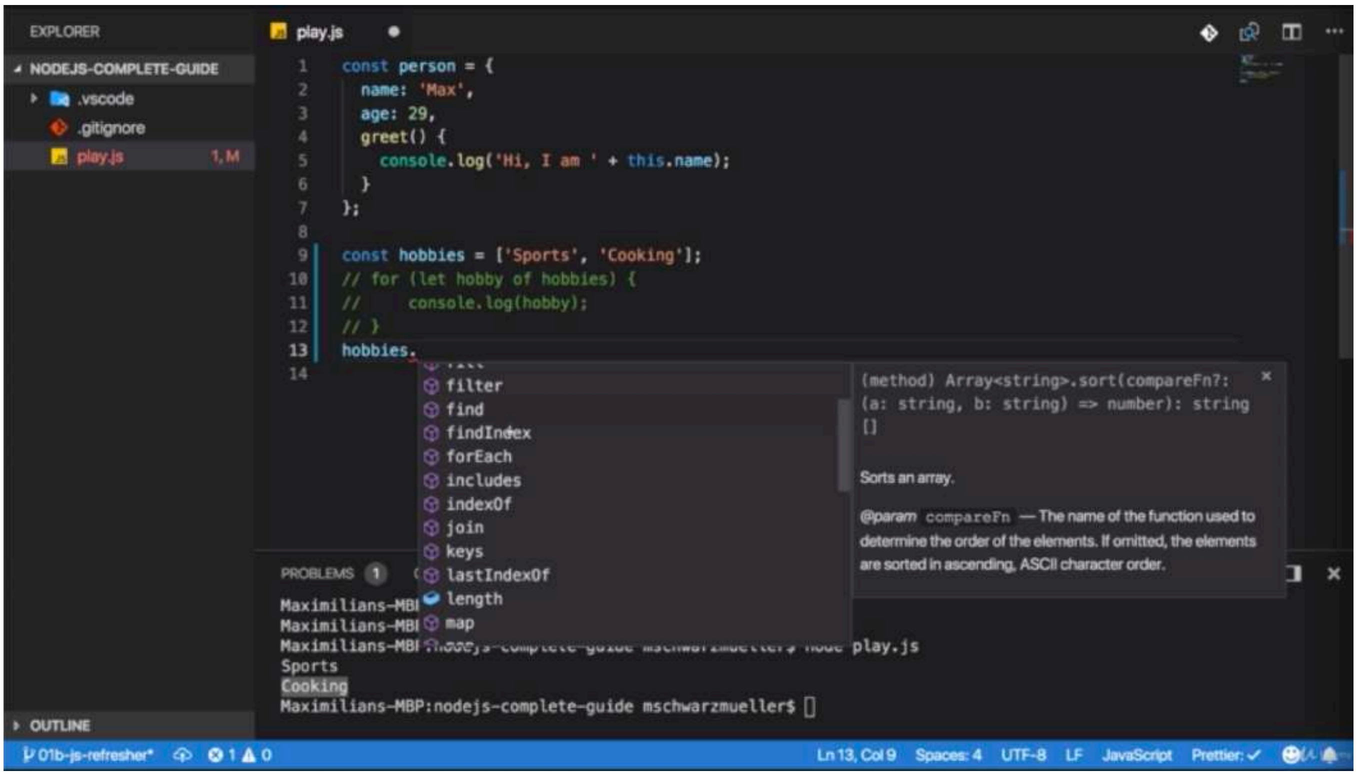
- An array is defined with square brackets [ ] and in that array, you can have any data you could normally use too, you can use string, numbers, boolean values, object and you don’t have to use one and the same type in that array. here we are mixing text and numbers.
![](images/15-array-and-array-methods-2.png)


EXPLORER
NODEJS-COMPLETE-GUIDE
.vscode
.gitignore
play.js
OUTLINE
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
1: bash
01b-js-refresher
Ln 11, Col 22 (5 selected) Spaces: 4 UTF-8 LF JavaScript Prettier: ✓

```javascript
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const hobbies = ['Sports', 'Cooking'];
for (let hobby of hobbies) {
    console.log(hobby);
}
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- and you can use for loops to go through that with this syntax. for example with the ‘for of’ loop where we store each element for each iteration in that hobby variable
![](images/15-array-and-array-methods-3.png)



```
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
hobbies.
```

```
filter
find
findIndex
forEach
includes
indexOf
join
keys
lastIndexOf
length
map
```

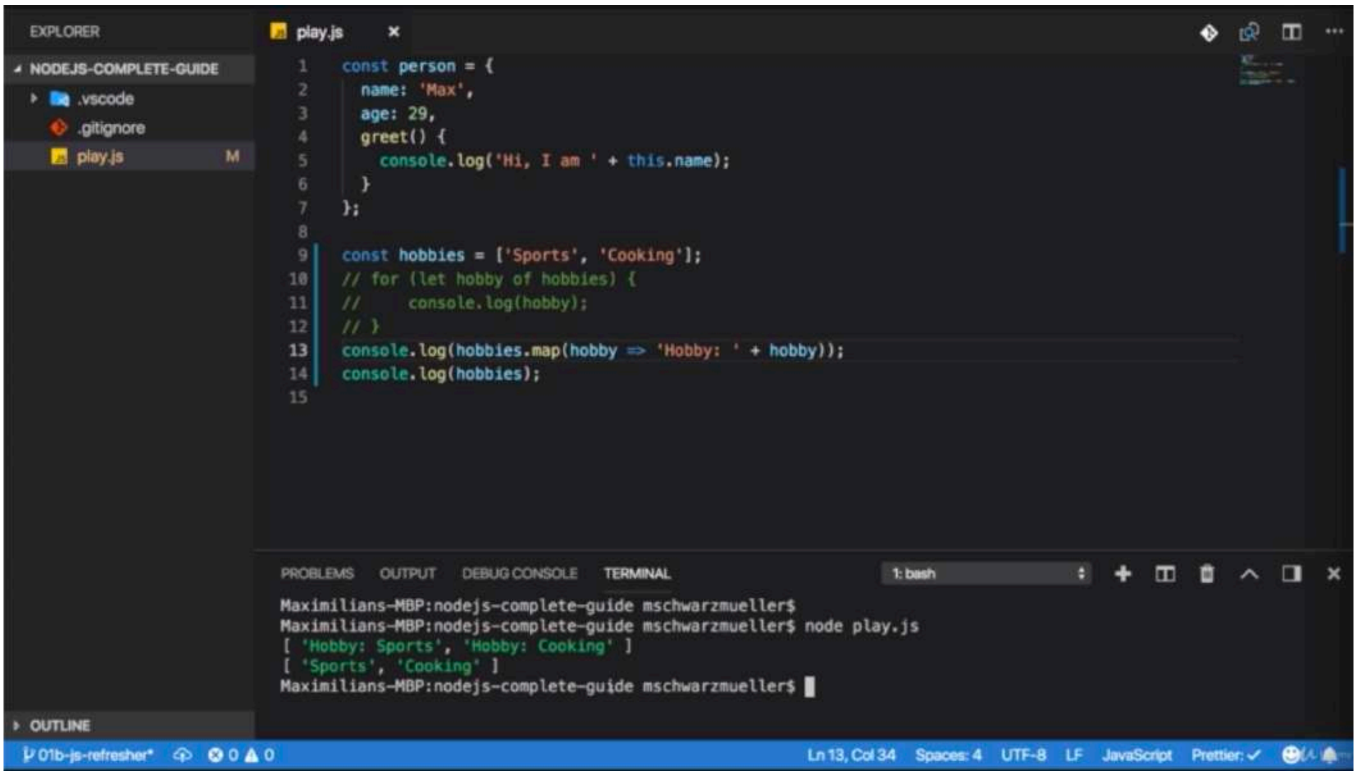
```
(method) Array<string>.sort(compareFn?:
(a: string, b: string) => number): string
[]
```

Sorts an array.

@param compareFn — The name of the function used to determine the order of the elements. If omitted, the elements are sorted in ascending, ASCII character order.

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Sports
Cooking
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- in javascript, we got a lot of methods we can use on arrays. all these methods help me go through the elements in the array, manipulate them, get a subset of that array, whatever i need.
- often you will see ‘map’ method which allows you to transform an array or transform the values and map will return a new array. so it will not edit the old one but give you a new one.
- and now map always takes a function where you define how to edit that array or how to edit the elements. that function will be executed on every elelemtn in the array one after another and you return the updated version of the element.
![](images/15-array-and-array-methods-4.png)



```
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
console.log(hobbies);
```

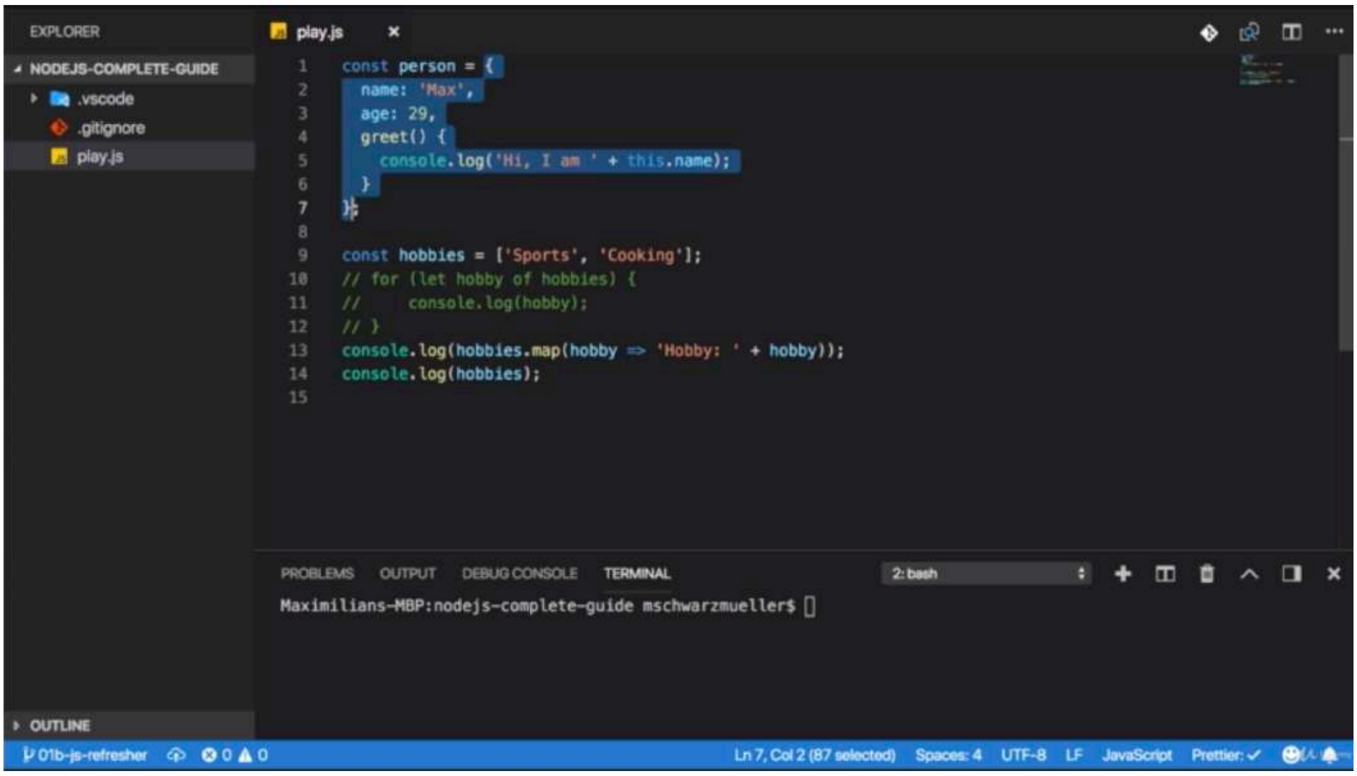
```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
[ 'Hobby: Sports', 'Hobby: Cooking' ]
[ 'Sports', 'Cooking' ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- first one is new one and second one is old one.
- i see the old array was no t edited. that is my second output which is coming from the original array.
- but the result of my ‘map’, new array pops up where i have my edited items with hobby added in front of every item.

```
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies){
//     console.log(hobby);
// }
console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
console.log(hobbies);
```

* Chapter 16: Arrays, Objects & Reference Types

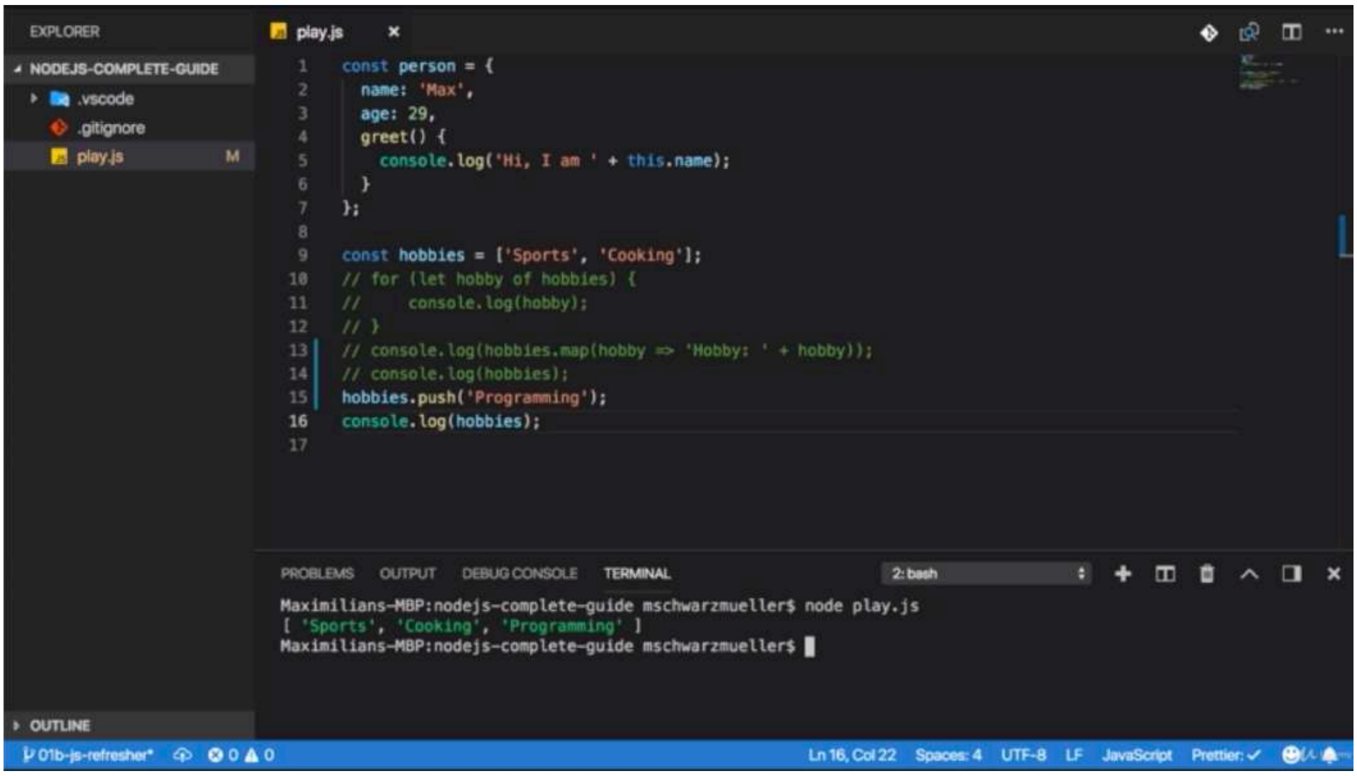
![](images/16-arrays-objects-reference-types-1.png)



```
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
console.log(hobbies);
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ 
```

- objects and arrays are so-called ‘reference types’ and you will learn all about that. they are reference types and therefore when i store an array in a constant hobbies, i can still edit this array. without violating the restriction that the constants must not change
![](images/16-arrays-objects-reference-types-2.png)


```javascript
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
// console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
// console.log(hobbies);
hobbies.push('Programming');
console.log(hobbies);
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
[ 'Sports', 'Cooking', 'Programming' ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- you see we get no error about editing this constant. the reason for that is that reference types only store an address pointing at the place in memory where that array is stored and that pointer this address has not changed by us adding a new element.
- so the thing stored in this constant is just this pointer, just this address and this has not changed. therefore our constant value has not changed.
- the thing it's pointing at has changed but that totally doesn't matter here.
- We are not really editing a thing that is stored in a constant, but we are only editing the thing that constant thing is pointing at
(we didn't change the place where we are pointing, but we changed the thing we are pointing at)

```javascript
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
// console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
// console.log(hobbies);
hobbies.push('Programming');
console.log(hobbies);
```

Reference Vs Primitive Values / Types

- Primitive Types
  1. number
  2. string

3. boolean
4. undefined
5. null

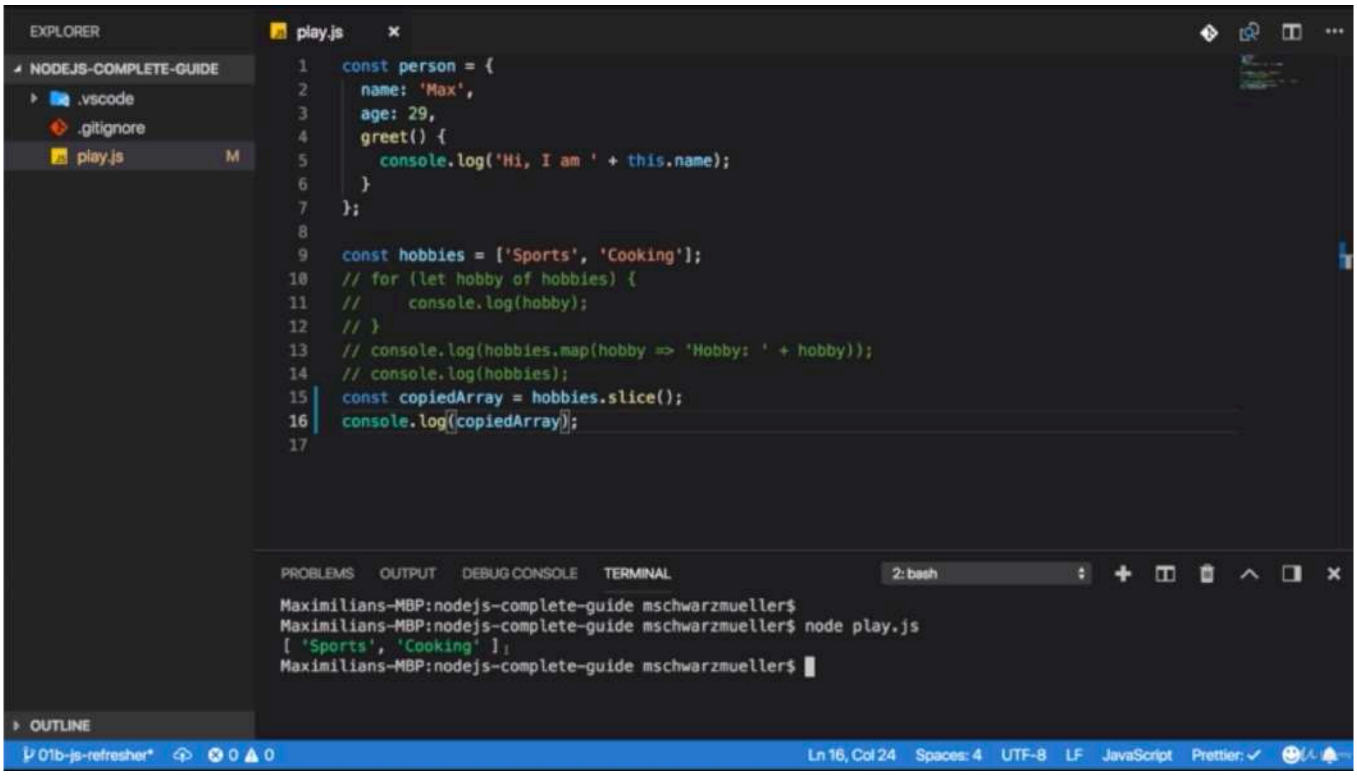
- reference types
  1. object
  2. array

```
//these are 'primitive types'

var age = 28;       //number
var name = 'Max'    //string
var isMale = true   //boolean
undefined           //undefined
null                //null

//these are 'reference types'

var person = {      //Object
    name: 'Max',
    age: 28
}

var hobbies = [     //Array
    'Sports', 'Cooking'
]
```

```
//reference types: object, array
/*
reference means that
when you create instance of object or array and change properties with that instance,
you change instance and original object or array both.
because you just only copied address(so-called pointer in C++ or C something) of original object or array.
so in result, you change properties along the address.
*/
var person = { name: 'Max' }
var newPerson = person
newPerson.name = 'Anna'
console.log(person.name, 'a')
console.log(newPerson.name, 'b')

//-----------

var hobbies = ['Sports', 'Cooking']
var copiedHobbies = hobbies
copiedHobbies.push('music')
console.log(hobbies[2], 'c')
console.log(copiedHobbies[2], 'd')

//-----------

//these are how you can copy the actual value
//## for array
//1. use slice() without arguments
var hobbies = ['Sports', 'Cooking']
var copiedHobbies = hobbies.slice()
copiedHobbies.push('programming')
console.log(hobbies,'e')
console.log(copiedHobbies,'f')

//2. use the spread operator
var hobbies = ['Sports', 'Cooking']
var copiedHobbies = [...hobbies]
copiedHobbies.push('Composing')
console.log(hobbies, 'g')
console.log(copiedHobbies, 'h')

//## for object
//1. Object.assign()
var person = { name: 'Max' }
var copiedHobbies = Object.assign({}, person)
console.log(person, 'i')
console.log(copiedHobbies, 'j')

//2. use spread operator on object
var person = { name: 'Max' }
var copiedPerson = {...person}
console.log(person, 'k')
console.log(copiedHobbies, 'l')
```

* Chapter 17: Understanding Spread & Rest Operators

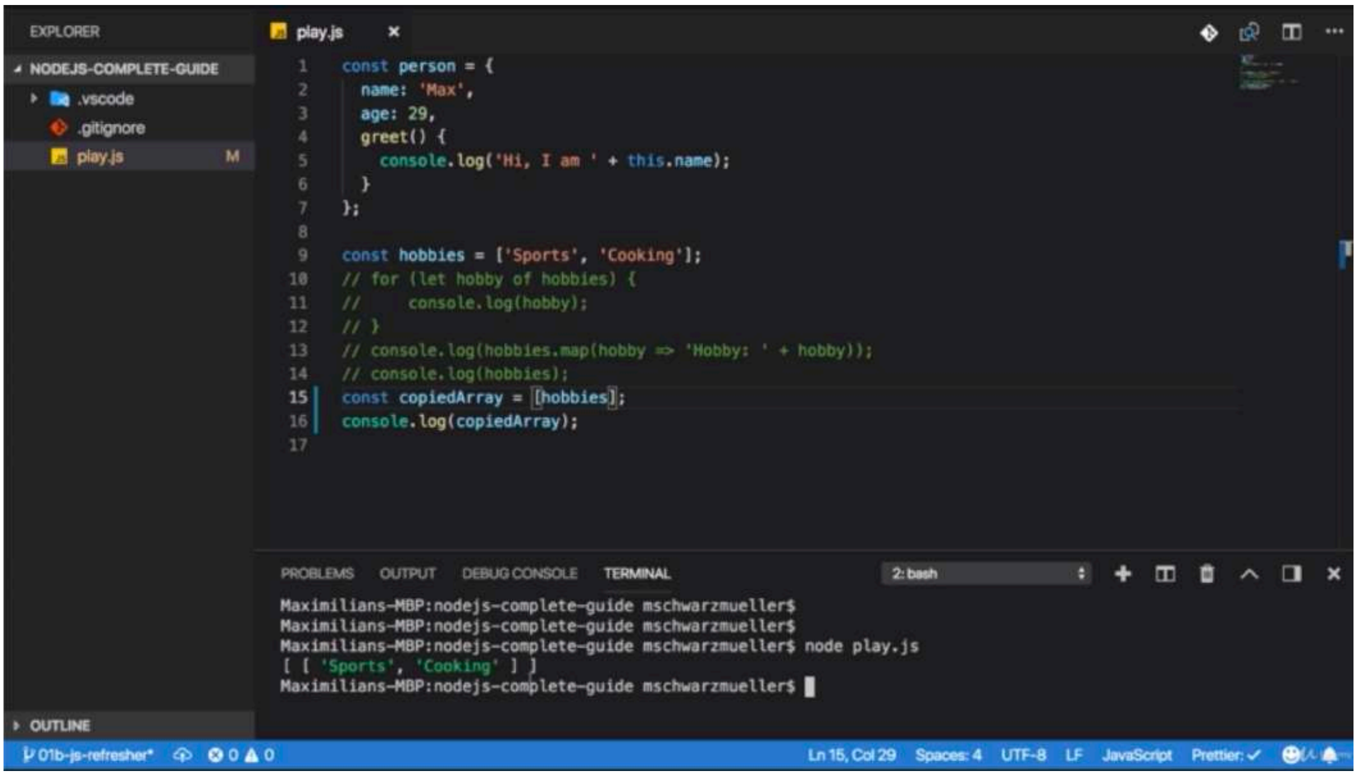
- let’s say we wanna implement a pattern where when we add a new hobby, we don’t edit the original array. but we create a new array with all the old values and the new values which is actually a pretty common pattern called ‘immutability’ where we never edit existing values but we always replace them with copies + changes.
- the idea behind that is that we avoid errors becasue we always have this clear approach of copy then edit. Don’t edit existing objects which might lead to more unreadable code.
![](images/17-understanding-spread-and-rest-operators-1.png)



```
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
// console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
// console.log(hobbies);
const copiedArray = hobbies.slice();
console.log(copiedArray);
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
[ 'Sports', 'Cooking' ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

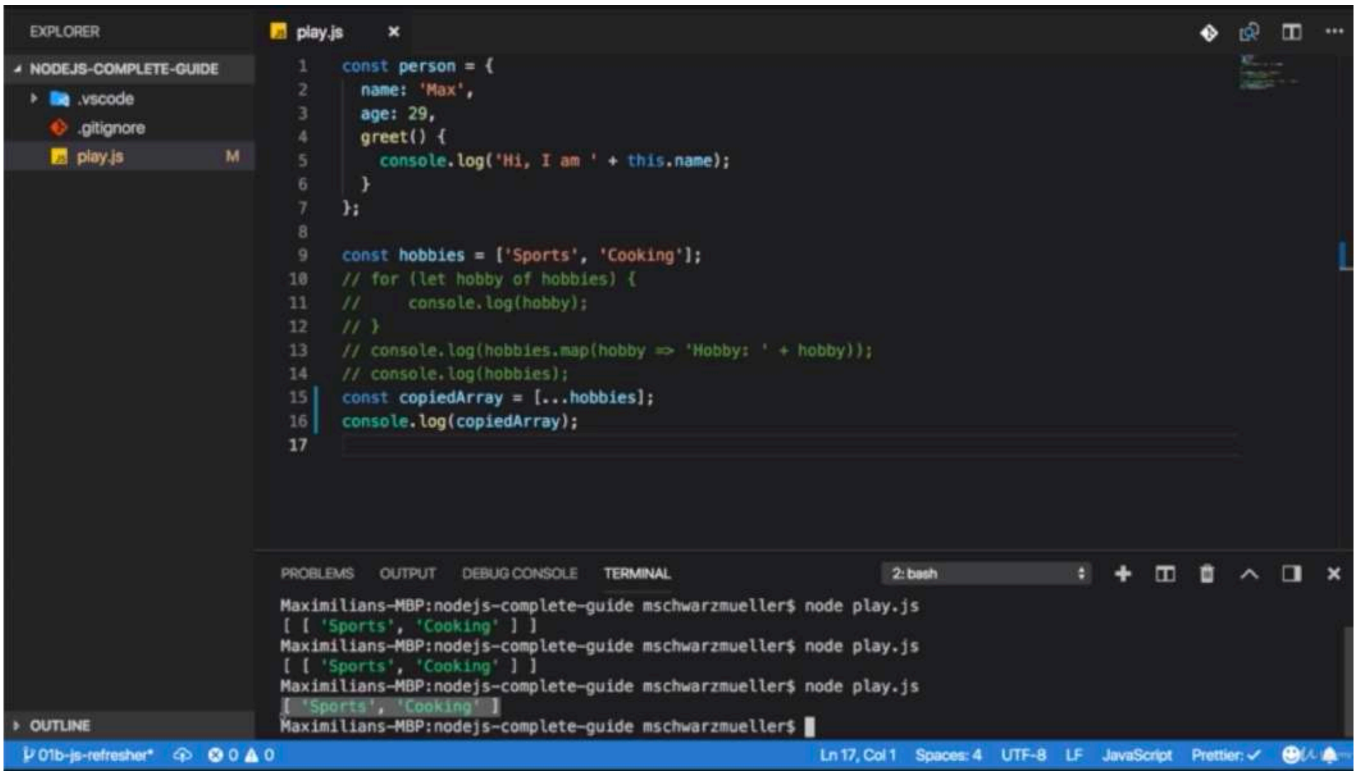
- we got a couple of possible techniques to copy things which is slice(). slice() simply copies an array, we can pass arguments to narrow down the range of elements we want to copy with no arguments, we copy the entire array.
![](images/17-understanding-spread-and-rest-operators-2.png)



```
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
// console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
// console.log(hobbies);
const copiedArray = [hobbies];
console.log(copiedArray);
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
[ [ 'Sports', 'Cooking' ] ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

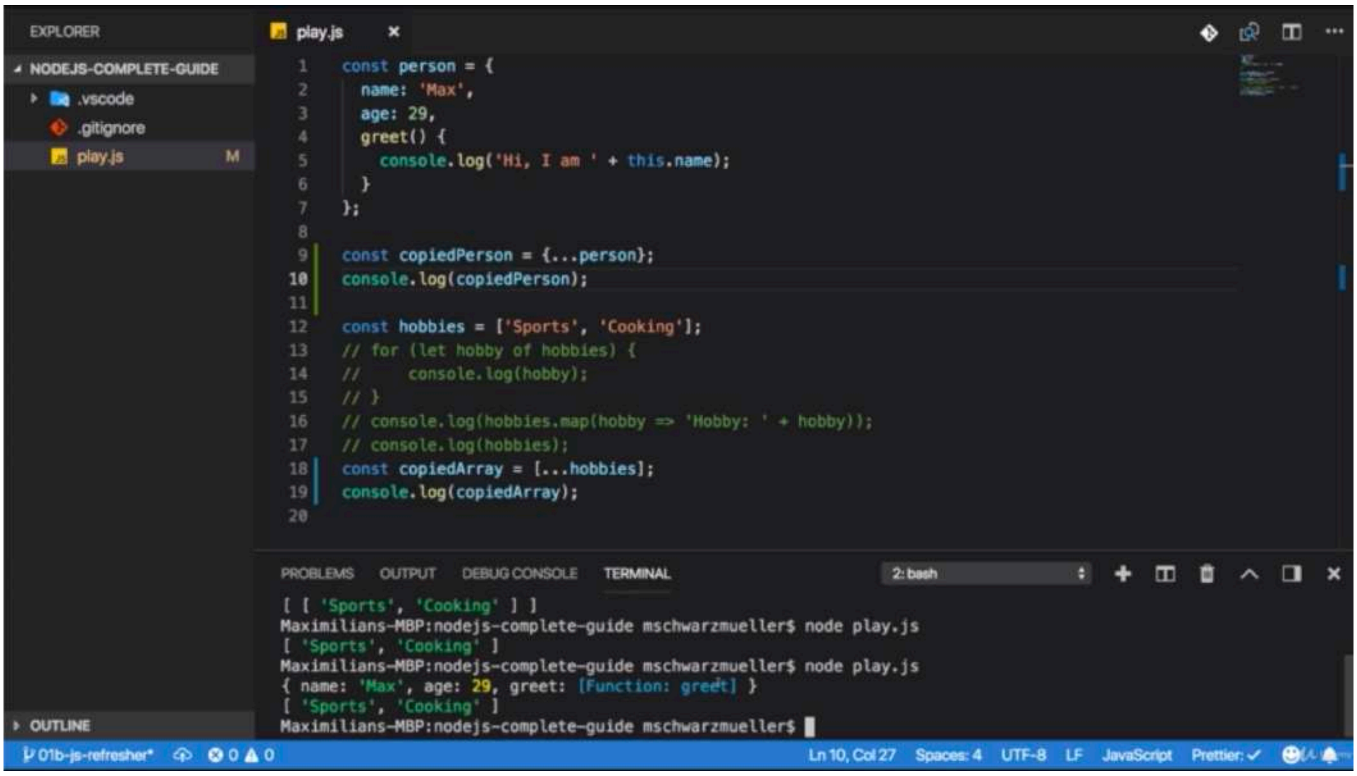
- another copy technique is that we create bracket and put in what we wanna copy.
- it looks like a copy on first sight, but actually the outer array has only one element and that’s the inner array. so it’s not a copy but a new array where the first element is the old array. and with that i mean the exact same object, not a copy of that.
![](images/17-understanding-spread-and-rest-operators-3.png)



```
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
// console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
// console.log(hobbies);
const copiedArray = [...hobbies];
console.log(copiedArray);
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
[ [ 'Sports', 'Cooking' ] ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
[ [ 'Sports', 'Cooking' ] ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
[ 'Sports', 'Cooking' ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

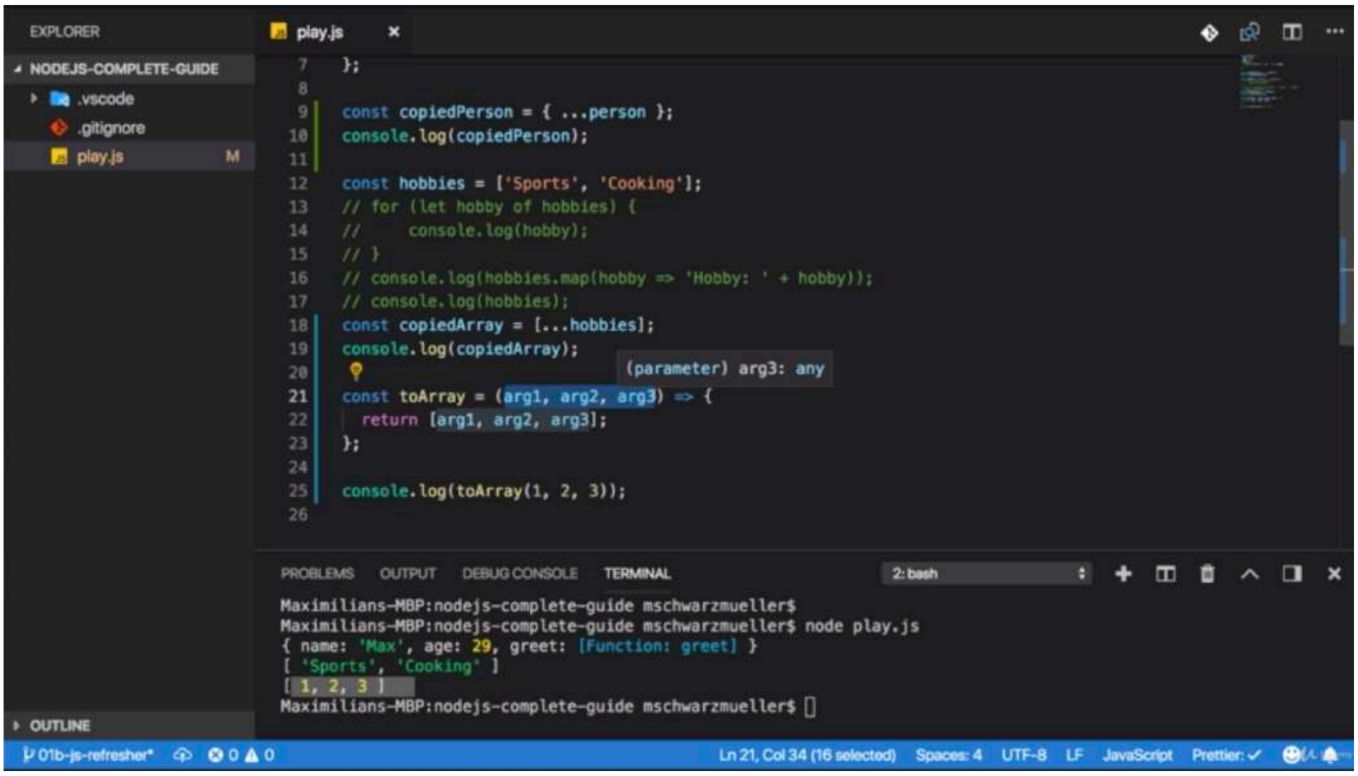
 - as you see, these 3 dots do one thing. they take the array or object after the operator and pull out all the elements or properties and put it to whatever is around that spread operator
- in this case, we got square brackets around the spread operator and therefore all the elemtns which are pulled out of the existing array are added to the new array.
- now this is a copy of the old one because we take the spread operator to pull out these elements and add them one by one to the new array.
![](images/17-understanding-spread-and-rest-operators-4.png)



```
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const copiedPerson = {...person};
console.log(copiedPerson);

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
// console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
// console.log(hobbies);
const copiedArray = [...hobbies];
console.log(copiedArray);
```

```
[ [ 'Sports', 'Cooking' ] ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
[ 'Sports', 'Cooking' ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
{ name: 'Max', age: 29, greet: [Function: greet] }
[ 'Sports', 'Cooking' ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

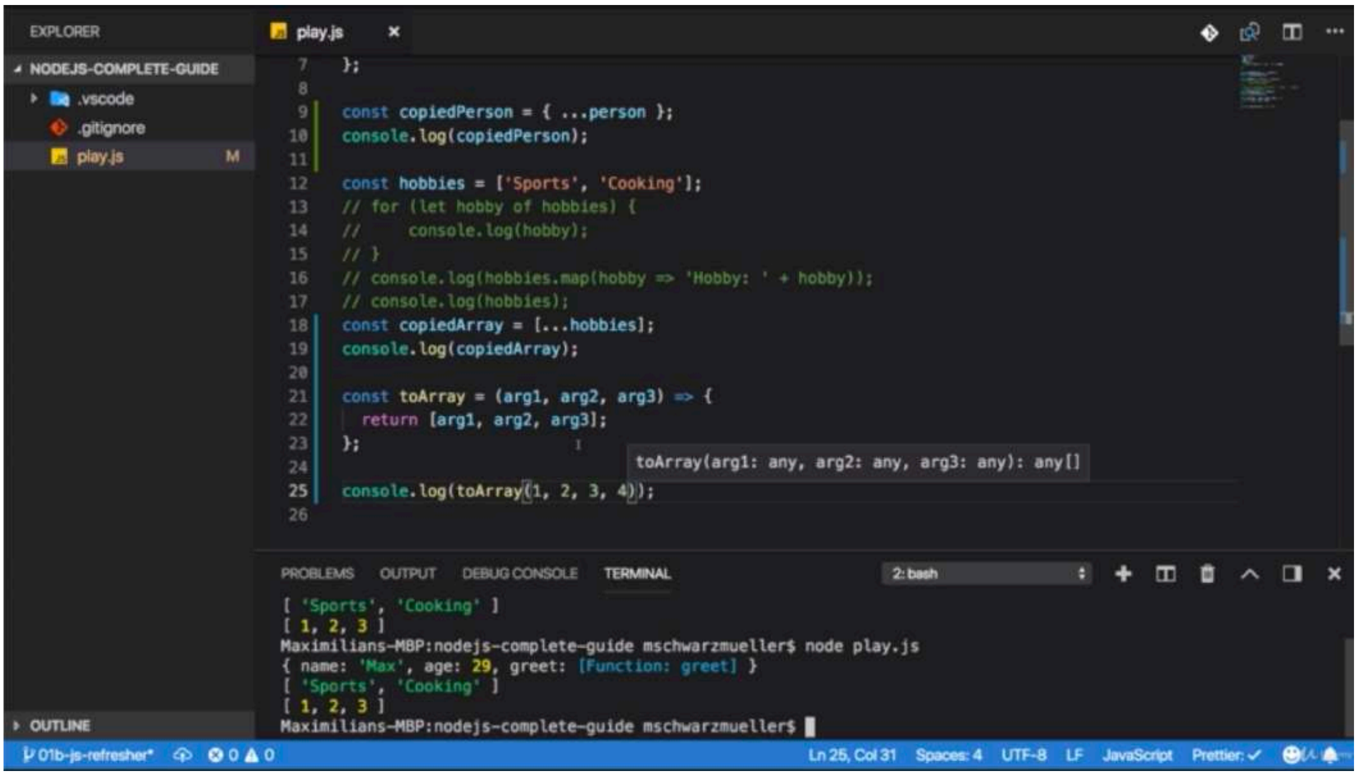
- in the same way, we could have our copied person by using curly braces then the spread operator.
![](images/17-understanding-spread-and-rest-operators-5.png)



```
};

const copiedPerson = { ...person };
console.log(copiedPerson);

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
// console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
// console.log(hobbies);
const copiedArray = [...hobbies];
console.log(copiedArray);

const toArray = (arg1, arg2, arg3) => {
  return [arg1, arg2, arg3];
};

console.log(toArray(1, 2, 3));
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
{ name: 'Max', age: 29, greet: [Function: greet] }
[ 'Sports', 'Cooking' ]
[ 1, 2, 3 ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

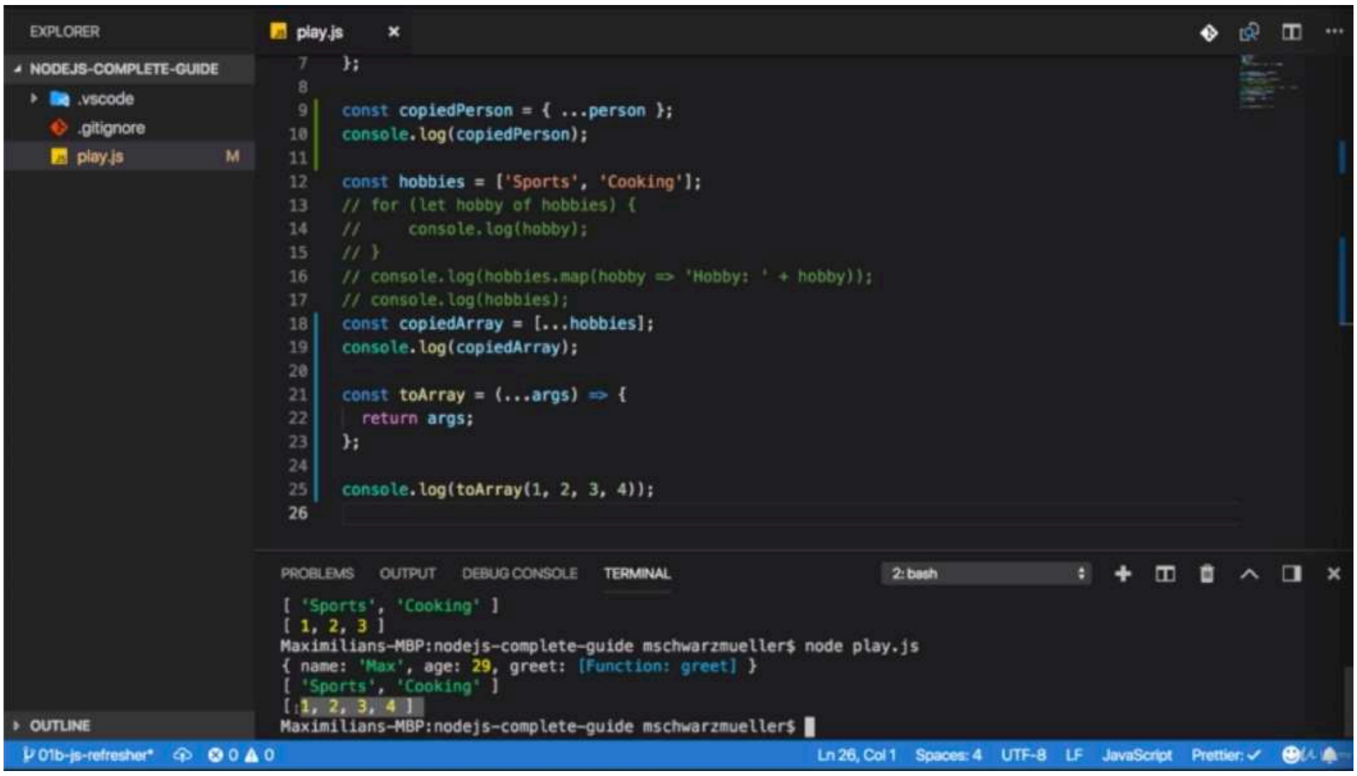
- by the way, the Rest Operator is essentially the opposite to spread operator.
- this is working but this is not totally flexible. what if we wanna pass 4 arguments?
![](images/17-understanding-spread-and-rest-operators-6.png)



```
};

const copiedPerson = { ...person };
console.log(copiedPerson);

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
// console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
// console.log(hobbies);
const copiedArray = [...hobbies];
console.log(copiedArray);

const toArray = (arg1, arg2, arg3) => {
  return [arg1, arg2, arg3];
};

console.log(toArray(1, 2, 3, 4));
```

```
[ 'Sports', 'Cooking' ]
[ 1, 2, 3 ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
{ name: 'Max', age: 29, greet: [Function: greet] }
[ 'Sports', 'Cooking' ]
[ 1, 2, 3 ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- it doesn't get added because we only work with 3 arguments. what we could do is rest operator which is like '… args'
![](images/17-understanding-spread-and-rest-operators-7.png)



```
};

const copiedPerson = { ...person };
console.log(copiedPerson);

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
// console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
// console.log(hobbies);
const copiedArray = [...hobbies];
console.log(copiedArray);

const toArray = (...args) => {
  return args;
};

console.log(toArray(1, 2, 3, 4));

```

```
[ 'Sports', 'Cooking' ]
[ 1, 2, 3 ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
{ name: 'Max', age: 29, greet: [Function: greet] }
[ 'Sports', 'Cooking' ]
[ 1, 2, 3, 4 ]
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

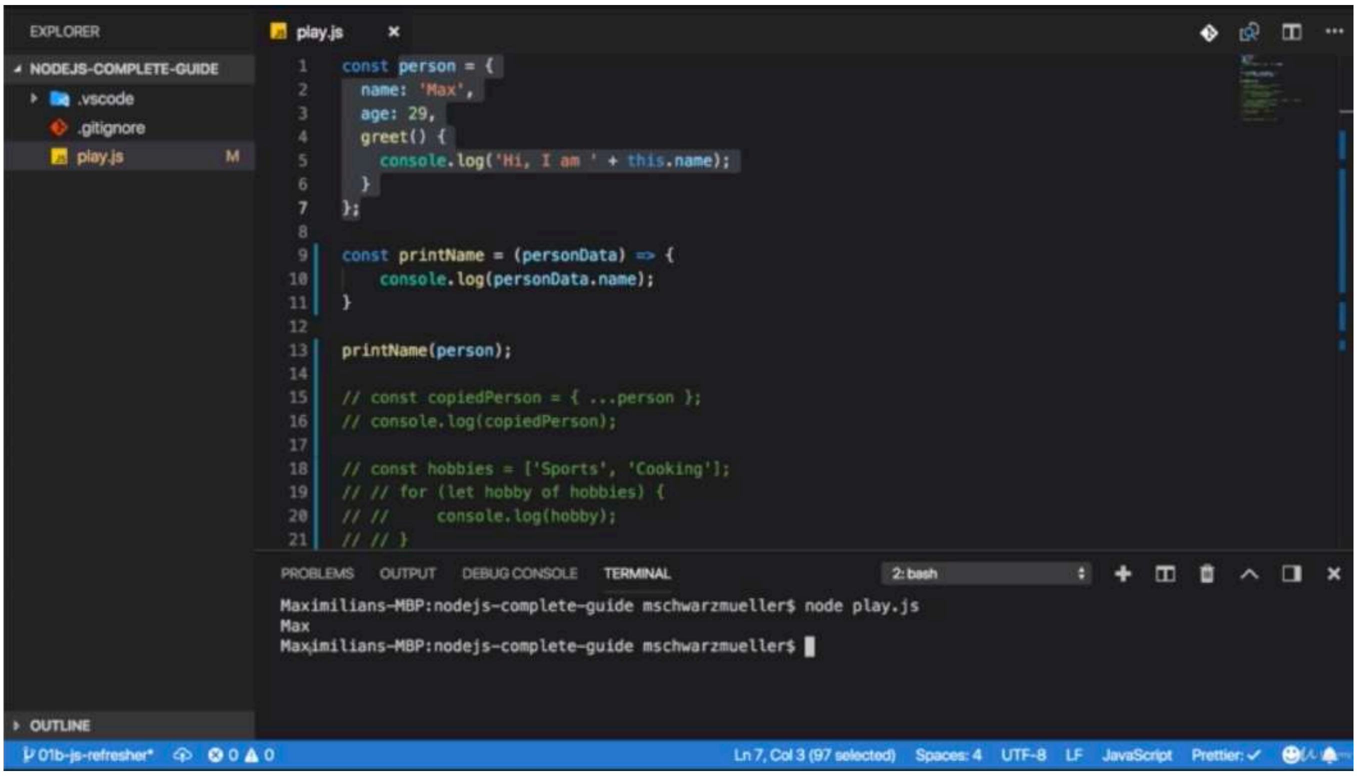
- '...args' is totally take all the arguments, how many we might specify that doesn't matter and it will bundle them up in a array for us.

```
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const copiedPerson = { ...person };
console.log(copiedPerson);

const hobbies = ['Sports', 'Cooking'];
// for (let hobby of hobbies) {
//     console.log(hobby);
// }
// console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
// console.log(hobbies);
const copiedArray = [...hobbies];
console.log(copiedArray);

const toArray = (...args) => {
  return args;
};

console.log(toArray(1, 2, 3, 4));

```

* Chapter 18: Destructuring

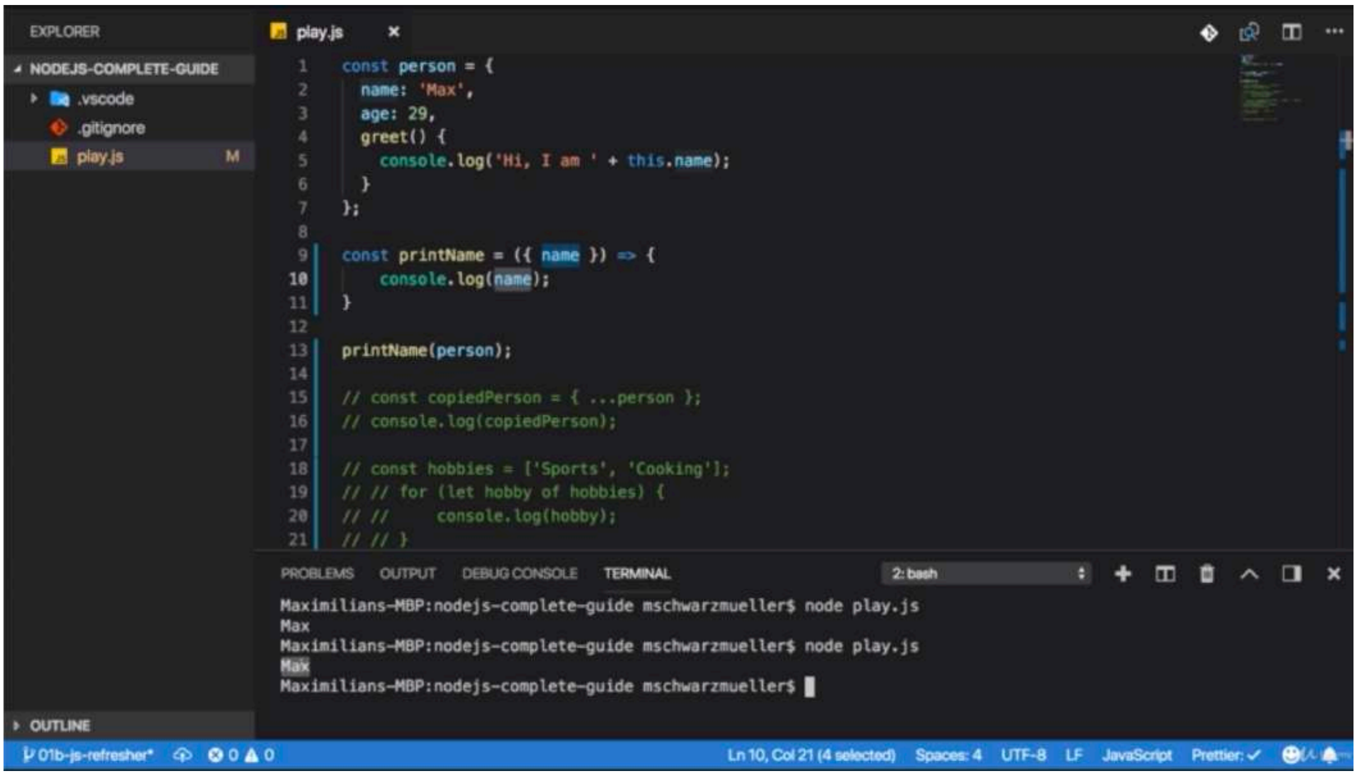
![](images/18-destructuring-1.png)

![](images/18-destructuring-2.png)
![](images/18-destructuring-3.png)
![](images/18-destructuring-4.png)



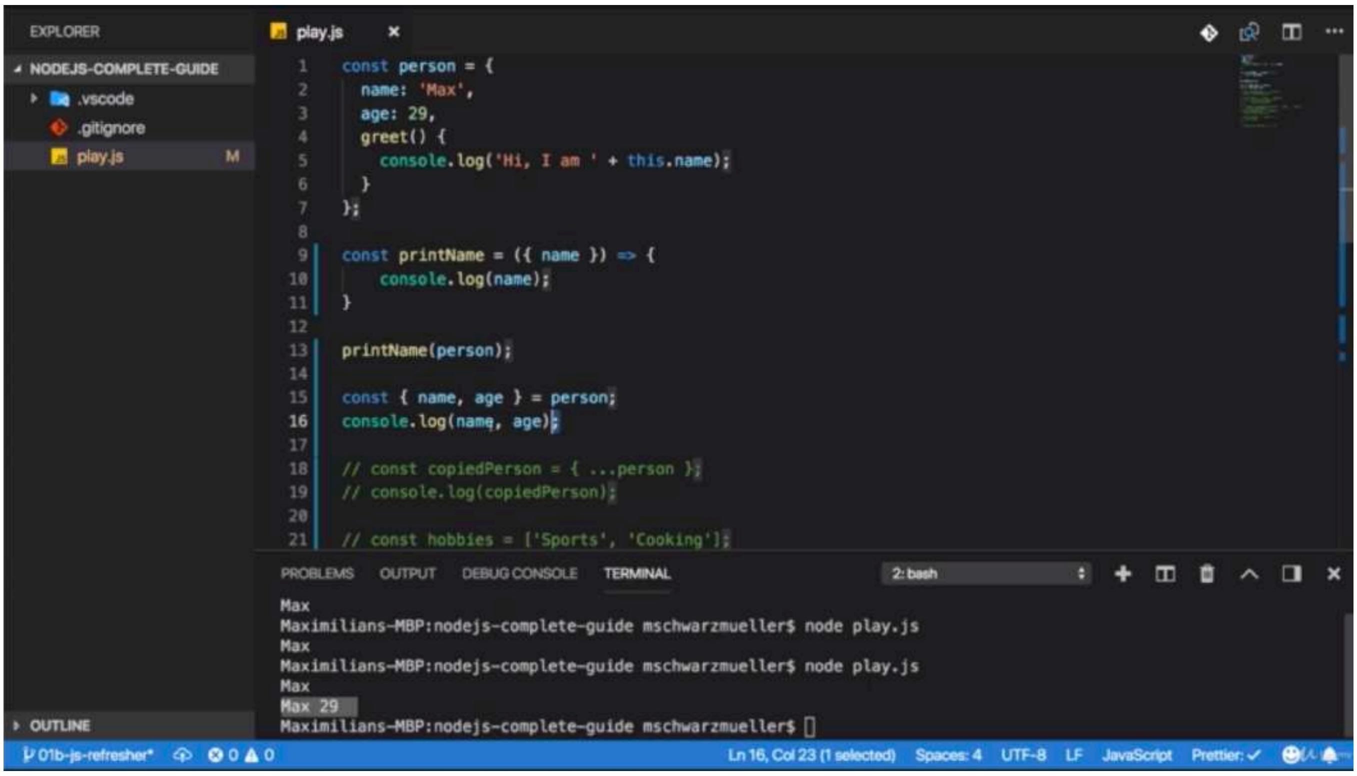
```javascript
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const printName = (personData) => {
    console.log(personData.name);
}

printName(person);

// const copiedPerson = { ...person };
// console.log(copiedPerson);

// const hobbies = ['Sports', 'Cooking'];
// // for (let hobby of hobbies) {
// //     console.log(hobby);
// // }
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Max
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```



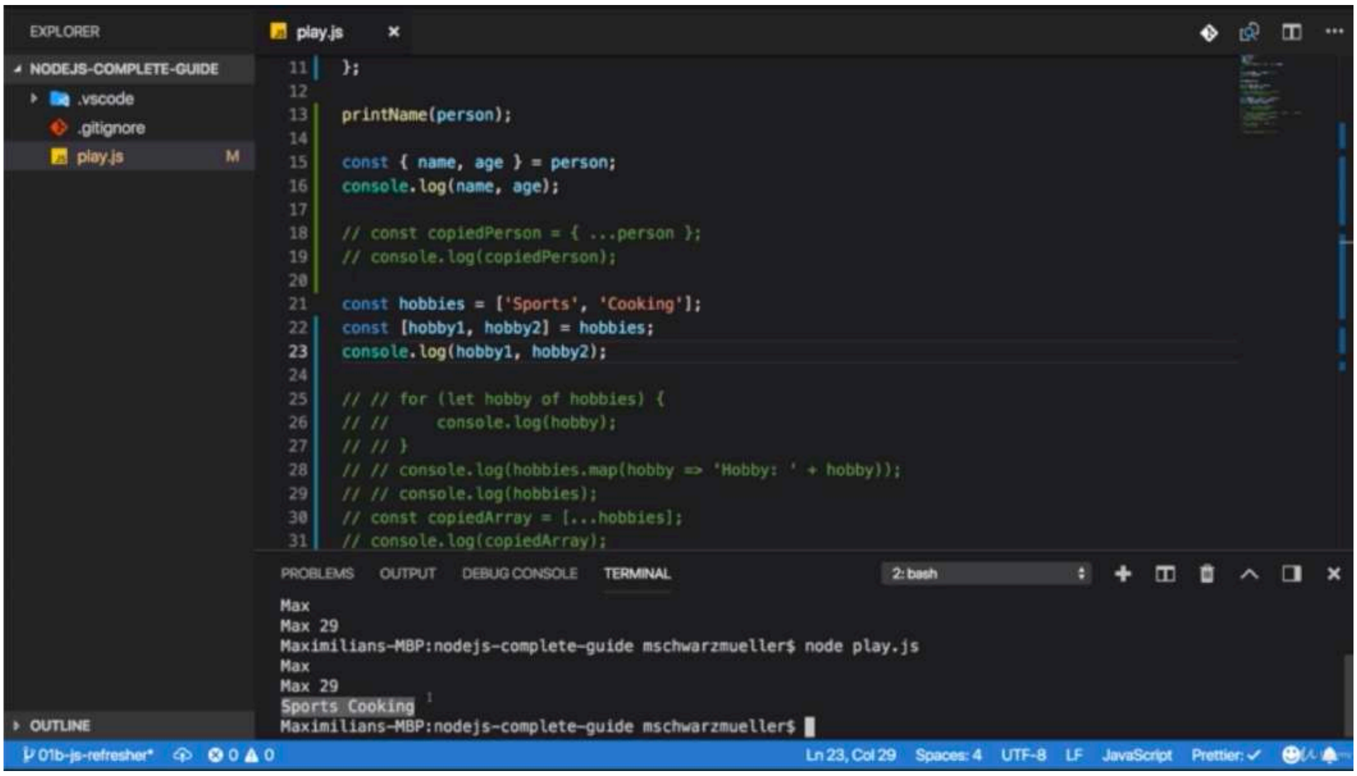
```javascript
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const printName = ({ name }) => {
    console.log(name);
}

printName(person);

// const copiedPerson = { ...person };
// console.log(copiedPerson);

// const hobbies = ['Sports', 'Cooking'];
// // for (let hobby of hobbies) {
// //     console.log(hobby);
// // }
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Max
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Max
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```



```js
const person = {
    name: 'Max',
    age: 29,
    greet() {
        console.log('Hi, I am ' + this.name);
    }
};

const printName = ({ name }) => {
    console.log(name);
}

printName(person);

const { name, age } = person;
console.log(name, age);

// const copiedPerson = { ...person };
// console.log(copiedPerson);

// const hobbies = ['Sports', 'Cooking'];
```

```
Max
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Max
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Max
Max 29
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```



```js
};

printName(person);

const { name, age } = person;
console.log(name, age);

// const copiedPerson = { ...person };
// console.log(copiedPerson);

const hobbies = ['Sports', 'Cooking'];
const [hobby1, hobby2] = hobbies;
console.log(hobby1, hobby2);

// // for (let hobby of hobbies) {
// //     console.log(hobby);
// // }
// // console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
// // console.log(hobbies);
// const copiedArray = [...hobbies];
// console.log(copiedArray);
```

```
Max
Max 29
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Max
Max 29
Sports Cooking
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

```js
const person = {
    name: 'Max',
    age: 29,
    greet(){
        console.log('Hi, I am ' + this.name);
    }
}

/*
this is actually takes the full person object because for whatever reason we write, wrote it like this
or we simply have a function where we are able to get multiple arguments or a full object because some third party package always gives us that person we can't change that,
```

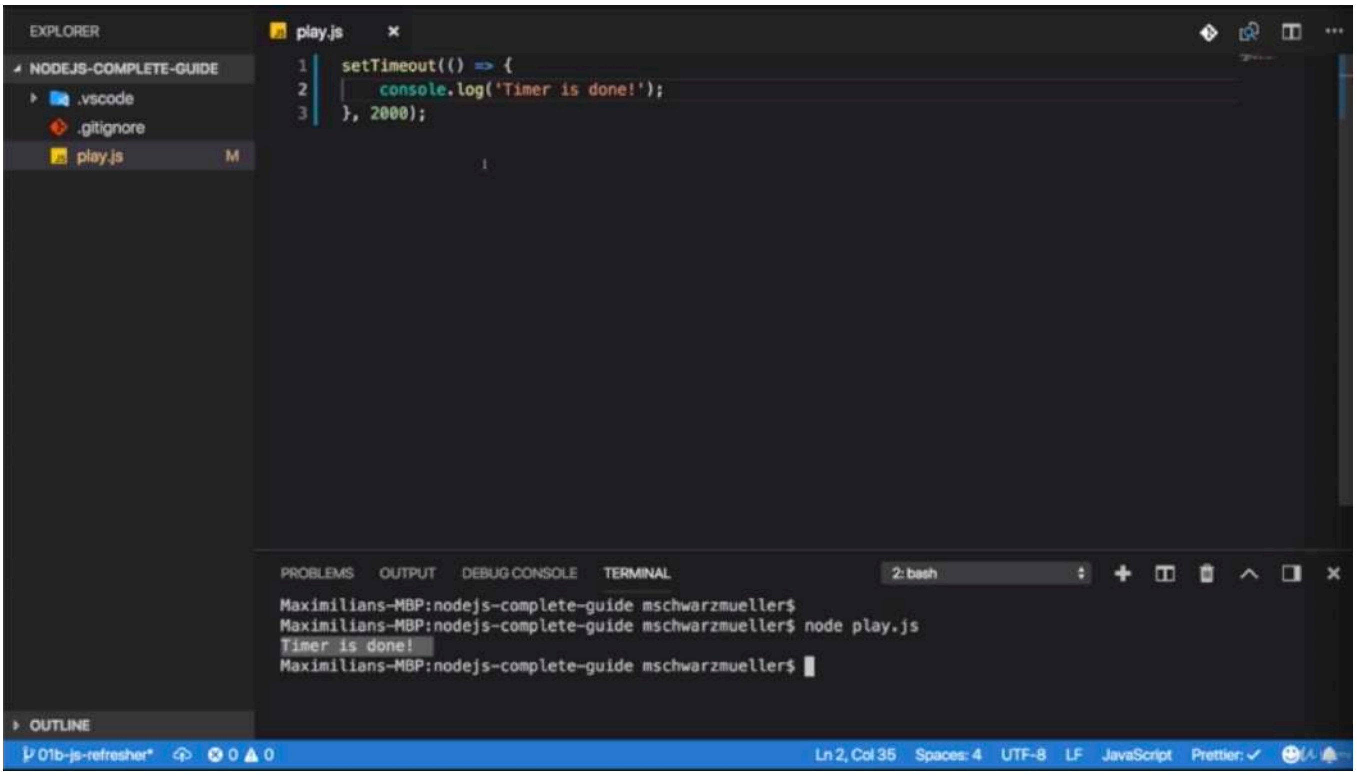
```javascript
so we get the person here
*/
/**now to avoid naming confusion, you can name this here however you want. */
const printName = (personData) => {
    console.log(personData.name);
}

printName(person);

/**or we can print out 'name' property by using called 'destructuring' */
const printName = ({ name }) => {
    console.log(name)
}

printName(person)

/**you can also use destructuring outside of function
 * these names(name, age in here) have to match the property names of the person.
*/
const { name, age } = person;
console.log(name, age);

/**you can also have destructure arrays.
 * there are no square brackets around them in the console log
 * because we are not logging an array here,
 * we are logging 2 individual values which we got via array destructuring
 * unlike the object destructuring, you can choose any names you want because in arrays your
elements have no names,
 * they are instead pulled out by position
*/
const hobbies = ['Sports', 'Cooking']
const [hobby1, hobby2] = hobbies;
console.log(hobby1, hobby2)
```

```javascript
const person = {
  name: 'Max',
  age: 29,
  greet() {
    console.log('Hi, I am ' + this.name);
  }
};

const printName = ({ name }) => {
  console.log(name);
};

printName(person);

const { name, age } = person;
console.log(name, age);

// const copiedPerson = { ...person };
// console.log(copiedPerson);

const hobbies = ['Sports', 'Cooking'];
const [hobby1, hobby2] = hobbies;
console.log(hobby1, hobby2);
```

```
// // for (let hobby of hobbies) {
// //     console.log(hobby);
// // }
// // console.log(hobbies.map(hobby => 'Hobby: ' + hobby));
// // console.log(hobbies);
// const copiedArray = [...hobbies];
// console.log(copiedArray);

// const toArray = (...args) => {
//   return args;
// };

// console.log(toArray(1, 2, 3, 4));

```

* Chapter 19: Async Code & Promises

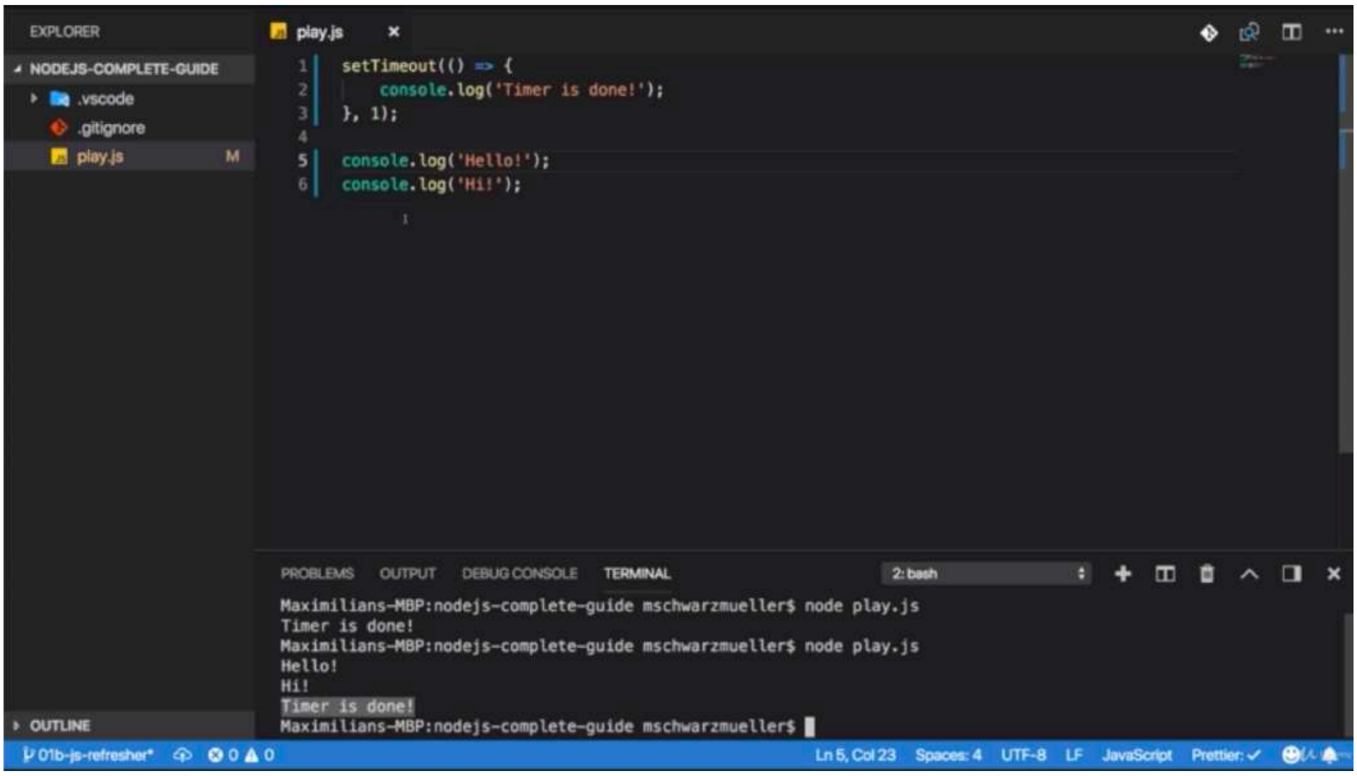
![](images/19-async-code-and-promises-1.png)



```
setTimeout(() => {
    console.log('Timer is done!');
}, 2000);
```

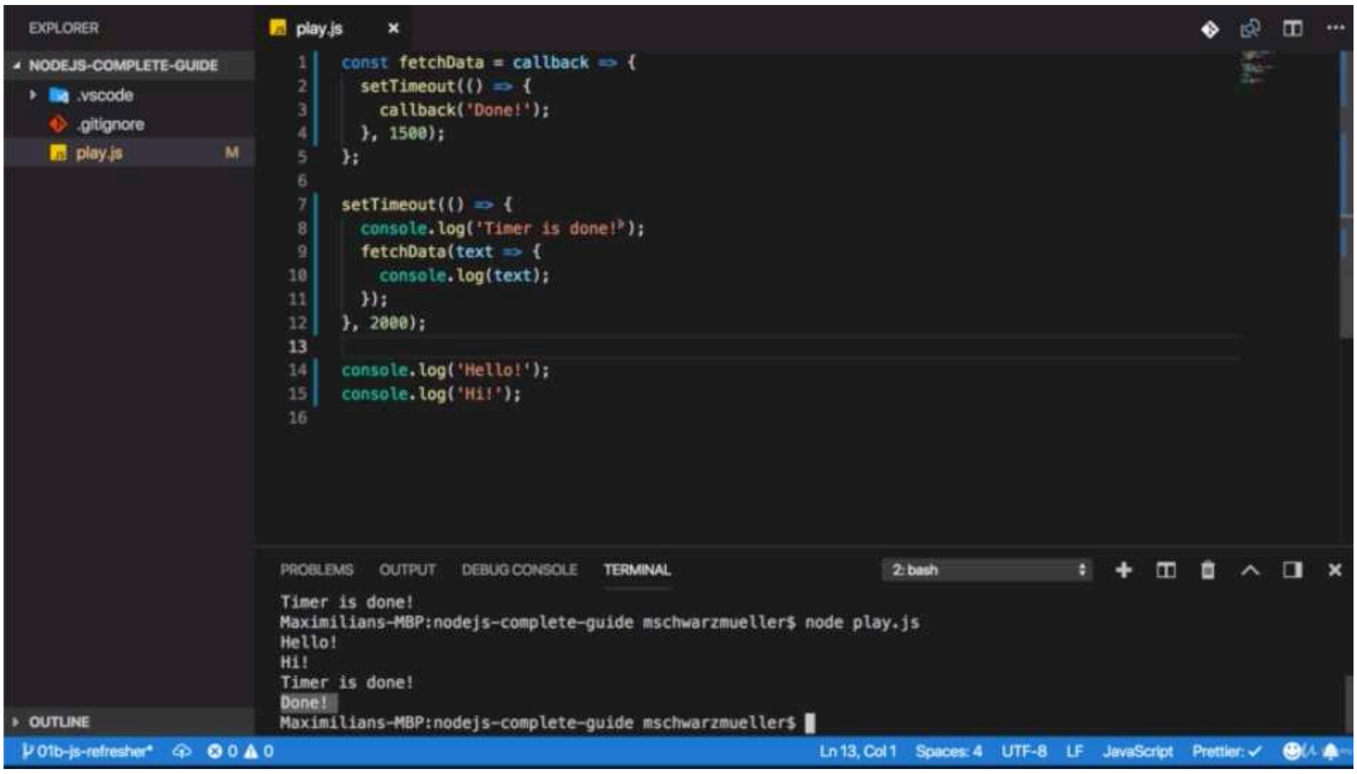
```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Timer is done!
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- this is asynchronous code because it doesn’t finish immediately. and it would even be async code if we had one millisecond there. so if it’s super fast, it does not happen immediately.
![](images/19-async-code-and-promises-2.png)

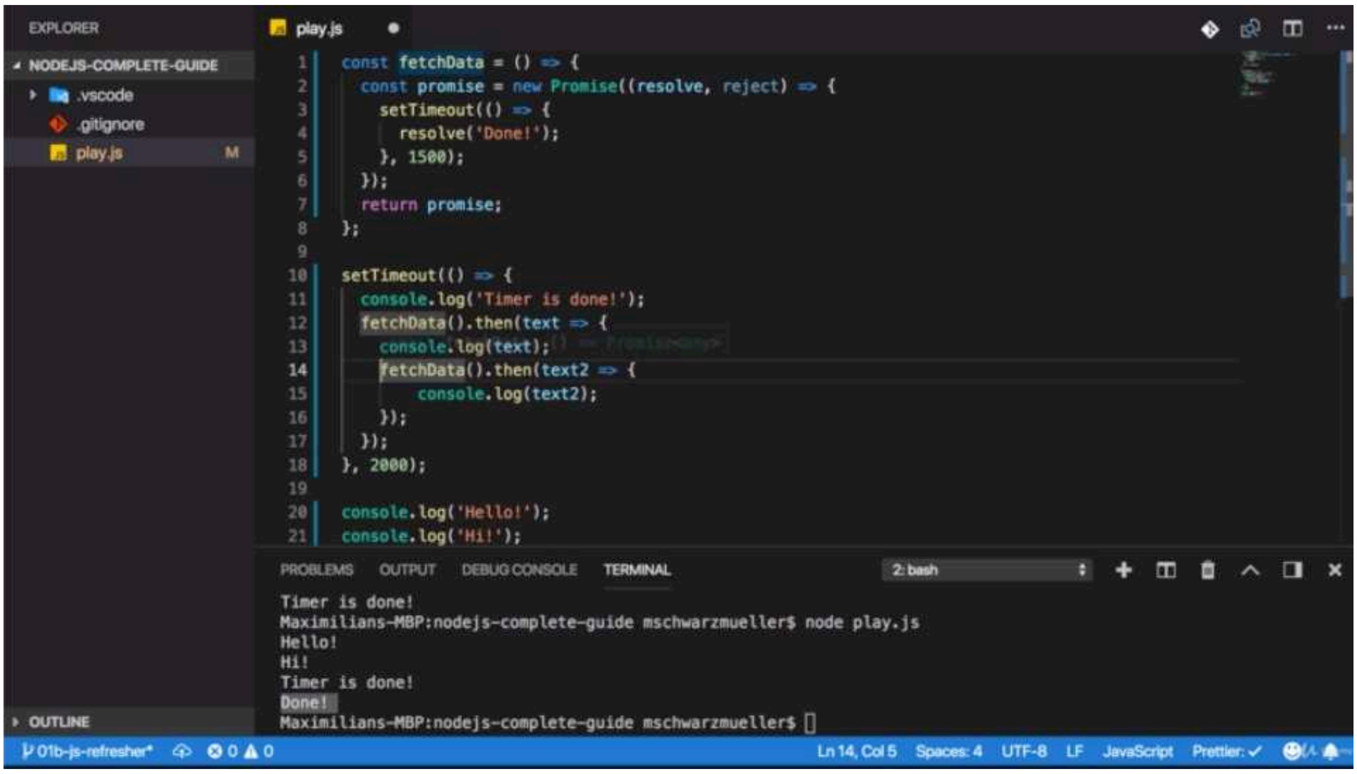

```
setTimeout(() => {
    console.log('Timer is done!');
}, 1);

console.log('Hello!');
console.log('Hi!');
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Timer is done!
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Hello!
Hi!
Timer is done!
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- this 2 codes are synchronous code because they are executed right after each other and technically node will take some time to execute them but there is no delay other than your hardware.
- if i execute this file like this, you see 'Hello!' and 'Hi!' befor you see 'Timer is done!' even though it's super fast becasue node.js and javascript in general does not block your code execution until that is done, it will recognize this so-called callback(() => {console.log('Timer is done!'}). so a function that should execute in the future, it should call back later once it is done so once this timer expired here, it will just recognize that and will then immediately move onto the next line. and it will execute all the synchronous code and then execute your async code once this is done which is why we see 'Hello!' and 'Hi!' first even though 'Timer is done' is printed first.
![](images/19-async-code-and-promises-3.png)
![](images/19-async-code-and-promises-4.png)


```
const fetchData = callback => {
  setTimeout(() => {
    callback('Done!');
  }, 1500);
};

setTimeout(() => {
  console.log('Timer is done!');
  fetchData(text => {
    console.log(text);
  });
}, 2000);

console.log('Hello!');
console.log('Hi!');
```

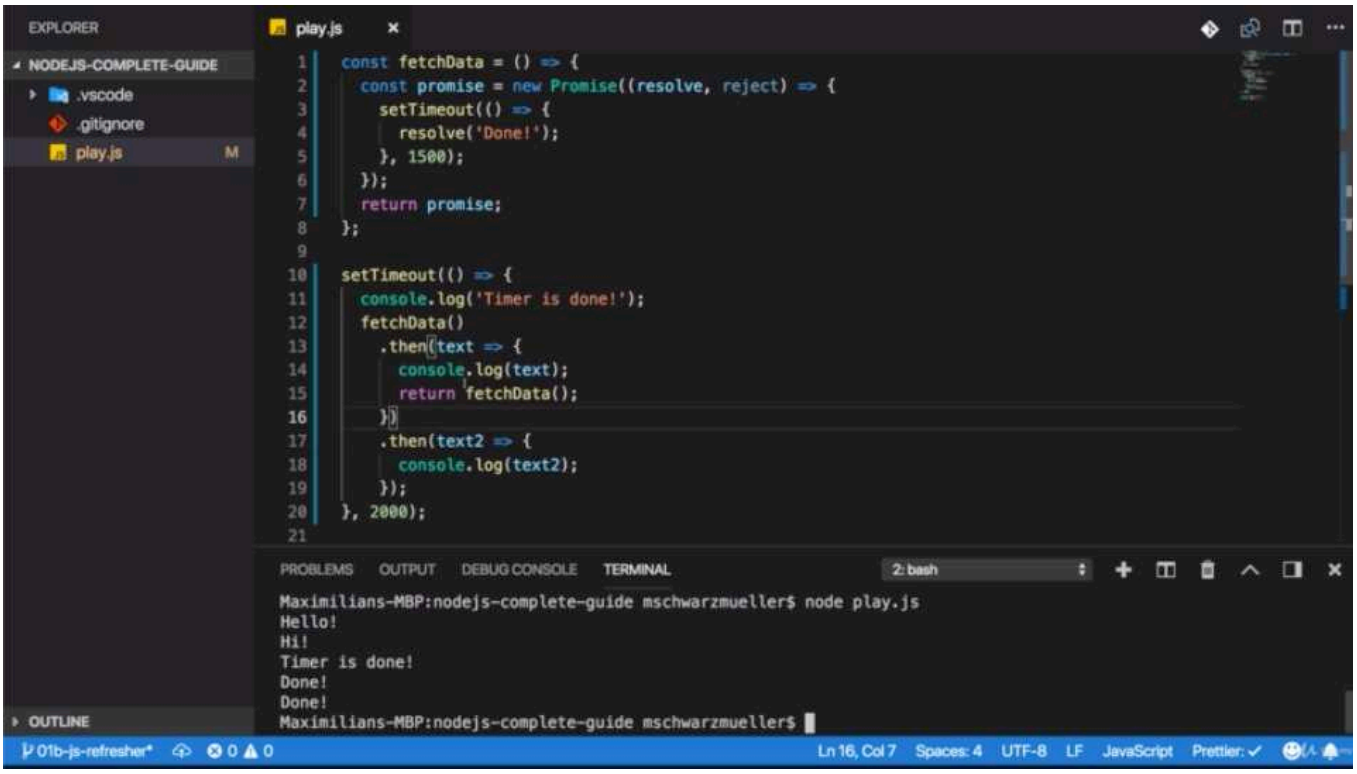
```
Timer is done!
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Hello!
Hi!
Timer is done!
Done!
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```


EXPLORER
NODEJS-COMPLETE-GUIDE
.vscode
.gitignore
play.js

```
const fetchData = () => {
  const promise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Done!');
    }, 1500);
  });
  return promise;
};

setTimeout(() => {
  console.log('Timer is done!');
  fetchData().then(text => {
    console.log(text);
    fetchData().then(text2 => {
        console.log(text2);
    });
  });
}, 2000);

console.log('Hello!');
console.log('Hi!');
```

```
Timer is done!
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Hello!
Hi!
Timer is done!
Done!
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- what is the advantage of this is that if we had multiple such as promises, let's say i called fetchData again in there. i don't have to use then like this.
![](images/19-async-code-and-promises-5.png)


EXPLORER
NODEJS-COMPLETE-GUIDE
.vscode
.gitignore
play.js

```
const fetchData = () => {
  const promise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Done!');
    }, 1500);
  });
  return promise;
};

setTimeout(() => {
  console.log('Timer is done!');
  fetchData()
    .then(text => {
      console.log(text);
      return fetchData();
    })
    .then(text2 => {
      console.log(text2);
    });
}, 2000);
```

```
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$ node play.js
Hello!
Hi!
Timer is done!
Done!
Done!
Maximilians-MBP:nodejs-complete-guide mschwarzmueller$
```

- but instead inside of a promise and then block is part of promise. so i can just return a new promise and then add the next then block after the previous one like this

```
/**i need some way of doing something when inner 'Timer is done'
 * so i will actually expect an argument which i will name 'callback'
 * because this argument will be a function
 * i will eventually call in my inner function once i'm done with the 'Timer is done'
 * and there i can pass the value called 'Done!'
*/

const fetchData = () => {
```

```javascript
/* we will use 3rd party packages that already use promises for us,
 * so the syntax i will show you now is one you rarely have to write on your own.
 * That will be done by the packages behind the scenes.
 */
  const promise = new Promise((resolve, reject) => {
/* 'resolve' completes the promise successfully, it resolves it successfully
 * 'reject' rejects it which is like throwing an error.
 */
    setTimeout(() => {
      resolve('Done!');
    }, 1500);
  });
/* 'promise' is synchronous code. so this will be returned immediately
 * after the promise gets created
 * before the code in the promise run which will happen sometimes later
 * when we call fetchData function and setTimeout function complete
 */
  return promise;
};

setTimeout(() => {
  console.log('Timer is done!');
  fetchData()
/*'text' is what is passed by the callback in my callback() when i excute it.*/
/* 'then()' is callable on a promise
 * and we return a promise and this simply allows you to define the callback function
 * which will execute once the promise is resolved
*/
    .then(text => {
      console.log(text);
      return fetchData();
    })
    .then(text2 => {
      console.log(text2);
    });
}, 2000);

console.log('Hello!');
console.log('Hi!');
```

```javascript
//clear version

const fetchData = () => {
  const promise = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Done!');
    }, 1500);
  });
  return promise;
};

setTimeout(() => {
  console.log('Timer is done!');
  fetchData()
    .then(text => {
      console.log(text);
      return fetchData();
    })
    .then(text2 => {
      console.log(text2);
    });
}, 2000);

console.log('Hello!');
console.log('Hi!');
```

* Chapter 20: Template Literals

- Template Literals is a different way of writing strings.
- instead of using double or single quotation marks like ‘A String’ or “Another string”, you can use backticks(`) like `Another way of writing strings`
- now why would we use that way of creating strings?
- with that syntax, you can dynamically add data into a string like this

```
const name = "Max";
const age = 29
console.log(`My name is ${name} and I am ${age} years old`);
```

- this is the shorter and easier to read than the ‘old’ way of concatenating strings

```
const name = "Max"
const age = 29
console.log("My name is " + name + " and I am " + age + " years old");
```