

# Immutable JavaScript

---



*Fernando Daciuk*

---

```
$ npm install fdaciuk
```



*Fernando Daciuk*

---

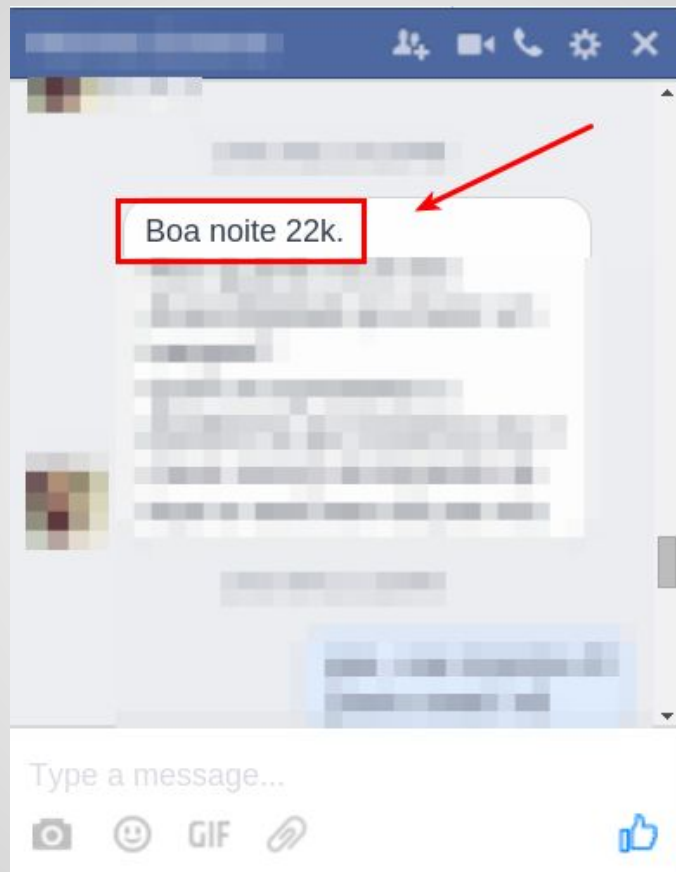
```
$ npm install fdaciuk
```



<http://da2k.com.br>

**Daciuk = Da "Two" k**

**Daciuk = Da2k**



Boa noite 22k.



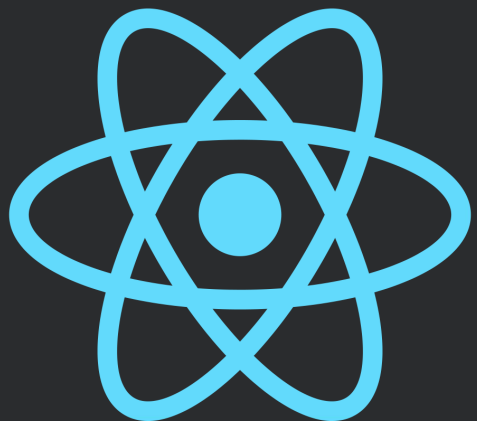


# JAVASCRIPT NINJA



<https://blog.da2k.com.br/cursos>

JS



**React**  
*Ninja*

<https://blog.da2k.com.br/cursos>

<https://queroser.ninja/promocoes>

# Immutable JavaScript

---

# Immutability

is really

**easy**

**to understand**

Think in **something**

that

**NEVER**

changes

**That is**

**Immutability!**

**Okay, but**

**why** should

**I use**

**Immutability?**



**#1**

**concise code**

**Variables must have**

**same value**

**from**

**start**

**to finish**

**#2**

**avoid bugs**

**Immutable**

**code**

**prevents**

**side-effects**

# Bugs

usually

live in

**mutable state**

**#3**

**thread safe**

**Immutable**  
**code**

**doesn't**  
**change**

Therefore,

**it doesn't** have

**race**  
**conditions**



**So, how is that**

**related to**

**JavaScript?**

**two**

**ways:**

**#1**

**assignment**

**#2**

**objects**

**Let's see  
some**

**code!**



```
1 var globalCoords = [0, 0]
2
3 // some code...
4
5 function updateCoords () {
6   globalCoords = [10, 10]
7 }
8
9 // some code...
10
11 updateCoords()
```

```
1 var globalCoords = [0, 0]
```

```
2
```

```
3 // some code...
```

```
4
```

```
5 function updateCoords () {
```

```
6   globalCoords = [10, 10]
```

```
7 }
```

```
8
```

```
9 // some code...
```

```
10
```

```
11 updateCoords()
```

**variable declaration**



```
1 var globalCoords = [0, 0]
2
3 // some code...
4
5 function updateCoords () {
6   globalCoords = [10, 10]
7 }
8
9 // some code...
10
11 updateCoords()
```



**assign an array**





```
1 var globalCoords = [0, 0]
```

```
2
```

```
3 // some code...
```

```
4
```

```
5 function updateCoords () {
```

```
6   globalCoords = [10, 10]
```

```
7 }
```

```
8
```

```
9 // some code...
```

```
10
```

```
11 updateCoords()
```



**function to update coords**



```
1 var globalCoords = [0, 0]
2
3 // some code...
4
5 function updateCoords () {
6   globalCoords = [10, 10]
7 }
8
9 // some code...
10
11 updateCoords()
```

**function call**





```
1 var globalCoords = [0, 0]
2
3 // some code...
4
5 function updateCoords () {
6   globalCoords = [10, 10]
7 }
8
9 // some code...
10
11 updateCoords()
```



**re-assignment**

**That is**

**mutability**

**by**

**assignment**

**How to**

**make**

**that code**

**immutable?**



```
1 const globalCoords = [0, 0]
2
3 // some code...
4
5 function updateCoords () {
6   globalCoords = [10, 10]
7 }
8
9 // some code...
10
11 updateCoords()
```



```
1 const globalCoords = [0, 0]
```

```
2
```

```
3 // some code...
```

```
4
```

```
5 function updateCoords () {
```

```
6   globalCoords = [10, 10]
```

```
7 }
```

```
8
```

```
9 // some code...
```

```
10
```

```
11 updateCoords()
```

**change "var" to "const"**



```
1 const globalCoords = [0, 0]
2
3 // some code...
4
5 function updateCoords () {
6   globalCoords = [10, 10]
7 }
8
9 // some code...
10
11 updateCoords()
```

**function call**







```
1 globalCoords = [10, 10]
```

```
2           ^
```

```
3 TypeError: Assignment to constant variable.
```



```
1 globalCoords = [10, 10]
```

```
2
```

```
      ^
```

```
3 TypeError: Assignment to constant variable.
```



```
1 globalCoords = [10, 10]
```

```
2
```

```
3 TypeError: Assignment to constant variable.
```



```
1 globalCoords = [10, 10]
```

```
2
```

```
3 TypeError: Assignment to constant variable.
```



```
1 globalCoords = [10, 10]
```

```
2
```

```
^
```

```
3 TypeError: Assignment to constant variable.
```

**const**

**prevents**

**reassignments**

# Lesson #01:

use

**const**

instead

**var** or **let**

# reassignment problem:

**solved!** 



Let's **see**  
another

**example**



```
1 const array123 = [1, 2, 3]
```



```
1 const array123 = [1, 2, 3]
```

**look this array**





**prevent reassignments**

```
1 const array123 = [1, 2, 3]
```



```
1 const array123 = [1, 2, 3]
```

**concise name**



Let's **do**  
something  
with  
**that array**

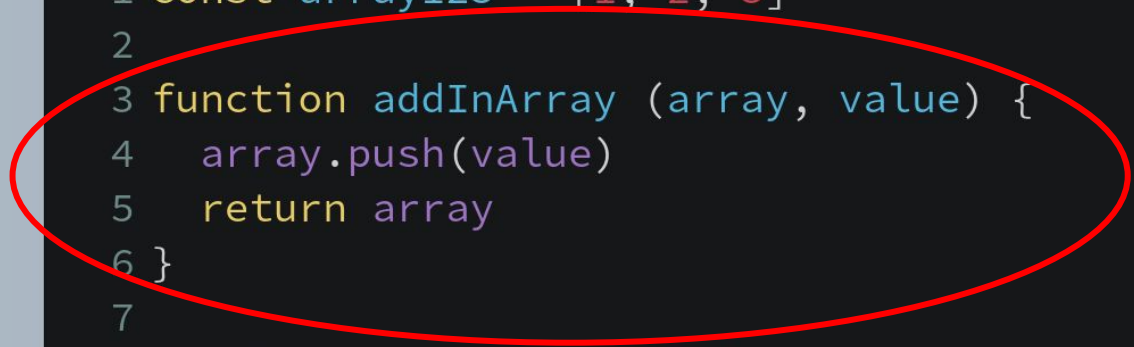


```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {
4   array.push(value)
5   return array
6 }
7
8 const array1234 = addInArray(array123, 4)
9 console.log(array1234) // [1, 2, 3, 4]
10
```



```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {
4   array.push(value)
5   return array
6 }
7
8 const array1234 = addInArray(array123, 4)
9 console.log(array1234) // [1, 2, 3, 4]
10
```

**a function that adds  
a value in an array**

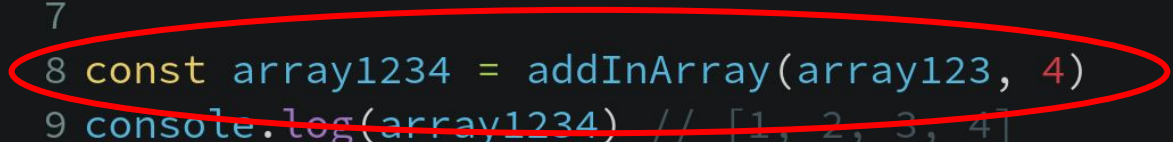






```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {
4   array.push(value)
5   return array
6 }
7
8 const array1234 = addInArray(array123, 4)
9 console.log(array1234) // [1, 2, 3, 4]
10
```

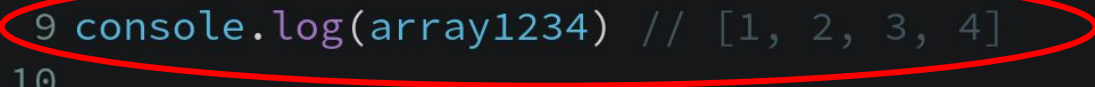
**put a value in  
array123**





```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {
4   array.push(value)
5   return array
6 }
7
8 const array1234 = addInArray(array123, 4)
9 console.log(array123) // [1, 2, 3, 4]
10
```

**log that value on console**



**Nothing**

**new**

**so far**

Let's **do**

another action

with

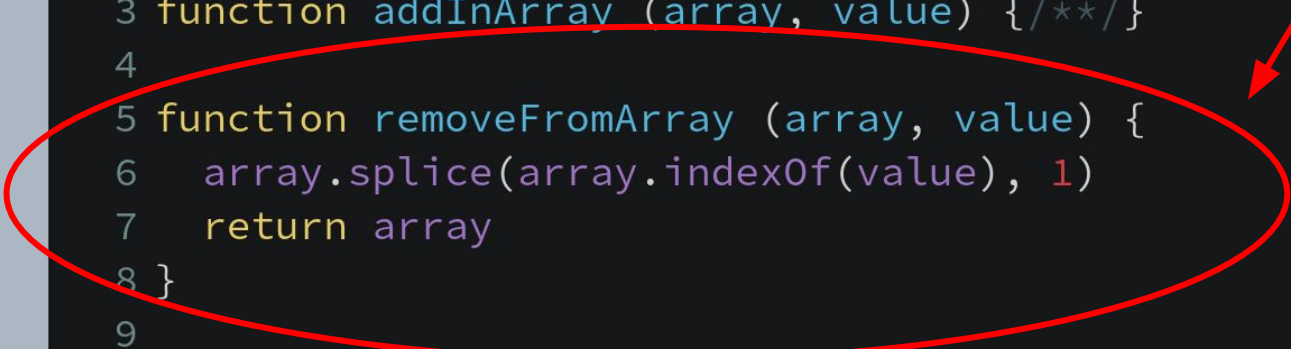
**same array**



```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {/**/}
4
5 function removeFromArray (array, value) {
6   array.splice(array.indexOf(value), 1)
7   return array
8 }
9
10 const array1234 = addInArray(array123, 4)
11 console.log(array1234) // [1, 2, 3, 4]
12
13 const array12 = removeFromArray(array123, 3)
14 console.log(array12) // [1, 2, 4]
```

```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {/**/}
4
5 function removeFromArray (array, value) {
6   array.splice(array.indexOf(value), 1)
7   return array
8 }
9
10 const array1234 = addInArray(array123, 4)
11 console.log(array1234) // [1, 2, 3, 4]
12
13 const array12 = removeFromArray(array123, 3)
14 console.log(array12) // [1, 2, 4]
```

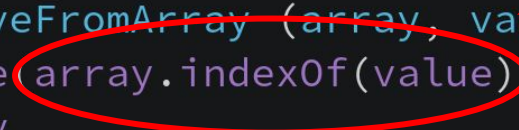
**a function that  
removes a value  
from an array**





```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {/**/}
4
5 function removeFromArray (array, value) {
6   array.splice(array.indexOf(value), 1)
7   return array
8 }
9
10 const array1234 = addInArray(array123, 4)
11 console.log(array1234) // [1, 2, 3, 4]
12
13 const array12 = removeFromArray(array123, 3)
14 console.log(array12) // [1, 2, 4]
```

**index to start  
changing the array**





```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {/**/}
4
5 function removeFromArray (array, value) {
6   array.splice(array.indexOf(value), 1)
7   return array
8 }
9
10 const array1234 = addInArray(array123, 4)
11 console.log(array1234) // [1, 2, 3, 4]
12
13 const array12 = removeFromArray(array123, 3)
14 console.log(array12) // [1, 2, 4]
```

**how many items is  
going to be deleted**



1



```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {/**/}
4
5 function removeFromArray (array, value) {
6   array.splice(array.indexOf(value), 1)
7   return array
8 }
9
10 const array1234 = addInArray(array123, 4)
11 console.log(array1234) // [1, 2, 3, 4]
12
13 const array12 = removeFromArray(array123, 3)
14 console.log(array12) // [1, 2, 4]
```

**the first result  
is actually the  
expected**



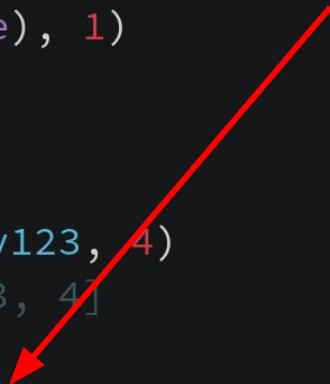
```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {/**/}
4
5 function removeFromArray (array, value) {
6   array.splice(array.indexOf(value), 1)
7   return array
8 }
9
10 const array1234 = addInArray(array123, 4)
11 console.log(array1234) // [1, 2, 3, 4]
12
13 const array12 = removeFromArray(array123, 3)
14 console.log(array12) // [1, 2, 4]
```

**expected:**  
**[1, 2]**



```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {/**/}
4
5 function removeFromArray (array, value) {
6   array.splice(array.indexOf(value), 1)
7   return array
8 }
9
10 const array1234 = addInArray(array123, 4)
11 console.log(array1234) // [1, 2, 3, 4]
12
13 const array12 = removeFromArray(array123, 3)
14 console.log(array12) // [1, 2, 4]
```

???



**What  
happened?**



```
1 console.log(array123) // [1, 2, 3]
2
3 const array1234 = addInArray(array123, 4)
4 console.log(array1234) // [1, 2, 3, 4]
5 console.log(array123) // [1, 2, 3, 4]
6
7 const array12 = removeFromArray(array123, 3)
8 console.log(array12) // [1, 2, 4]
9 console.log(array123) // [1, 2, 4]
```

```
1 console.log(array123) // [1, 2, 3]
2
3 const array1234 = addInArray(array123, 4)
4 console.log(array1234) // [1, 2, 3, 4]
5 console.log(array123) // [1, 2, 3, 4]
6
7 const array12 = removeFromArray(array123, 3)
8 console.log(array12) // [1, 2, 4]
9 console.log(array123) // [1, 2, 4]
```

**before call  
addInArray and  
removeFromArray  
functions**



```
1 console.log(array123) // [1, 2, 3]
2
3 const array1234 = addInArray(array123, 4)
4 console.log(array1234) // [1, 2, 3, 4]
5 console.log(array123) // [1, 2, 3, 4]
6
7 const array12 = removeFromArray(array123, 3)
8 console.log(array12) // [1, 2, 4]
9 console.log(array123) // [1, 2, 4]
```

**initial value**



```
1 console.log(array123) // [1, 2, 3]
2
3 const array1234 = addInArray(array123, 4)
4 console.log(array1234) // [1, 2, 3, 4]
5 console.log(array123) // [1, 2, 3, 4]
6
7 const array12 = removeFromArray(array123, 3)
8 console.log(array12) // [1, 2, 4]
9 console.log(array123) // [1, 2, 4]
```

**array123 has  
changed**

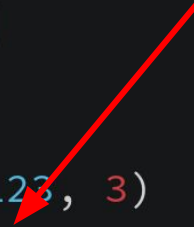






```
1 console.log(array123) // [1, 2, 3]
2
3 const array1234 = addInArray(array123, 4)
4 console.log(array1234) // [1, 2, 3, 4]
5 console.log(array123) // [1, 2, 3, 4]
6
7 const array12 = removeFromArray(array123, 3)
8 console.log(array12) // [1, 2, 4]
9 console.log(array123) // [1, 2, 4]
```

**array123 has  
changed AGAIN**



**Why** **is**  
**that**  
**happened?**

**For a**  
**complete**  
**understanding...**

Let's **talk**  
about

**objects**

**But**

**I do NOT**

**mean { }**

**I mean**

**object**

**data type**

**JavaScript** has  
two groups  
of  
**data types:**

**Primitives**

**and**

**objects**



# Primitives are:

**String**

**Number**

**Boolean**

**Undefined**

**Null**

**Symbol (ES6+)**

# Objects are all others:

**Object**    **Array**

**Function**    **RegExp**

**etc...**

**Primitive**

**values**

**are**

**immutable**

**Objects**  
are

always  
**mutable**

**What** **is**  
**that**  
**mean?**



```
1 const event = 'Front in Sampa'  
2 console.log(event) // "Front in Sampa"  
3  
4 const eventUpperCased = event.toUpperCase()  
5 console.log(eventUpperCased) // "FRONT IN SAMPA"  
6 console.log(event) // "Front in Sampa"
```



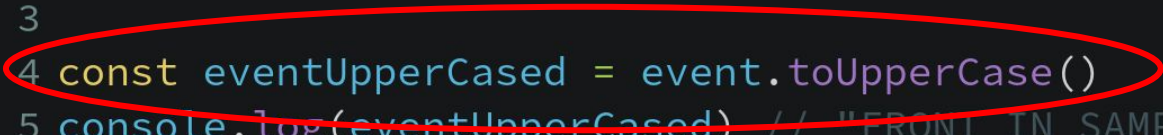
```
1 const event = 'Front in Sampa'  
2 console.log(event) // "Front in Sampa"  
3  
4 const eventUpperCased = event.toUpperCase()  
5 console.log(eventUpperCased) // "FRONT IN SAMPA"  
6 console.log(event) // "Front in Sampa"
```

**initial value**



```
1 const event = 'Front in Sampa'  
2 console.log(event) // "Front in Sampa"  
3  
4 const eventUpperCased = event.toUpperCase()  
5 console.log(eventUpperCased) // "FRONT IN SAMPA"  
6 console.log(event) // "Front in Sampa"
```

**change the  
string**

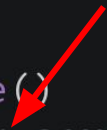






```
1 const event = 'Front in Sampa'
2 console.log(event) // "Front in Sampa"
3
4 const eventUpperCased = event.toUpperCase()
5 console.log(eventUpperCased) // "FRONT IN SAMPA"
6 console.log(event) // "Front in Sampa"
```

**initial value still  
the same**





```
1 const event = 'Front in Sampa'  
2 console.log(event) // "Front in Sampa"  
3  
4 const eventUpperCased = event.toUpperCase()  
5 console.log(eventUpperCased) // "FRONT IN SAMPA"  
6 console.log(event) // "Front in Sampa"
```

**a new string  
is created**



**That's**

**what**

**Immutability**

**means!**

**And**

**what about**

**objects?**



```
1 const person = { name: 'John Doe' }  
2 console.log(person) // { name: "John Doe" }  
3  
4 person.name = 'Jane Doe'  
5 console.log(person) // { name: "Jane Doe" }
```

---

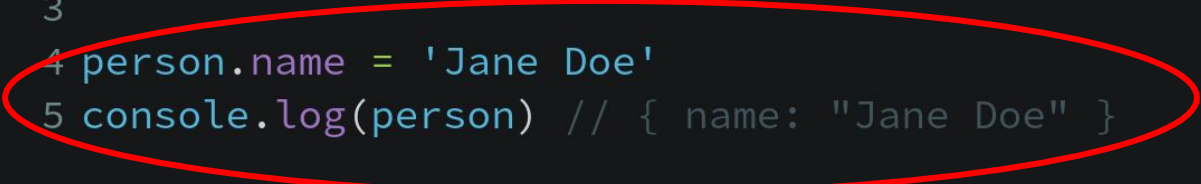
```
1 const person = { name: 'John Doe' }
2 console.log(person) // { name: "John Doe" }
3
4 person.name = 'Jane Doe'
5 console.log(person) // { name: "Jane Doe" }
```

**initial  
value**



```
1 const person = { name: 'John Doe' }  
2 console.log(person) // { name: "John Doe" }  
3  
4 person.name = 'Jane Doe'  
5 console.log(person) // { name: "Jane Doe" }
```

**object has  
changed**



**Remember:**



**Objects**  
are

always  
**mutable**

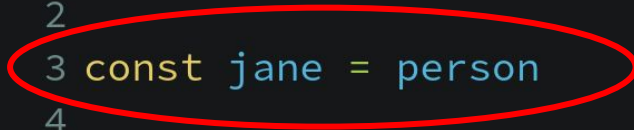


```
1 const person = { name: 'John Doe' }  
2  
3 const jane = person  
4  
5 jane.name = 'Jane Doe'  
6  
7 console.log(jane) // { name: "Jane Doe" }  
8 console.log(person) // { name: "Jane Doe" }
```



```
1 const person = { name: 'John Doe' }  
2  
3 const jane = person  
4  
5 jane.name = 'Jane Doe'  
6  
7 console.log(jane) // { name: "Jane Doe" }  
8 console.log(person) // { name: "Jane Doe" }
```

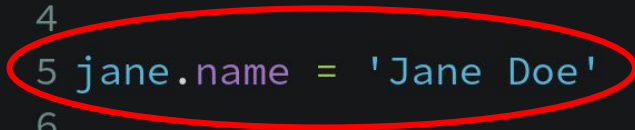
**copy (?)**





```
1 const person = { name: 'John Doe' }  
2  
3 const jane = person  
4  
5 jane.name = 'Jane Doe'  
6  
7 console.log(jane) // { name: "Jane Doe" }  
8 console.log(person) // { name: "Jane Doe" }
```

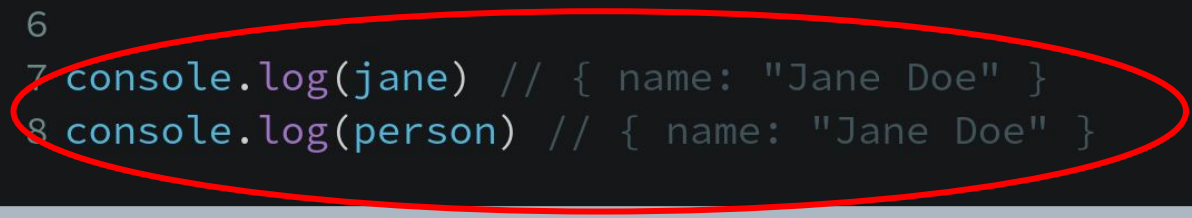
**change the  
copy (?)**





```
1 const person = { name: 'John Doe' }  
2  
3 const jane = person  
4  
5 jane.name = 'Jane Doe'  
6  
7 console.log(jane) // { name: "Jane Doe" }  
8 console.log(person) // { name: "Jane Doe" }
```

**both objects  
log the same**





```
1 const person = { name: 'John Doe' }  
2  
3 const jane = person  
4  
5 jane.name = 'Jane Doe'  
6  
7 console.log(jane) // { name: "Jane Doe" }  
8 console.log(person) // { name: "Jane Doe" }
```

**it is not a copy**

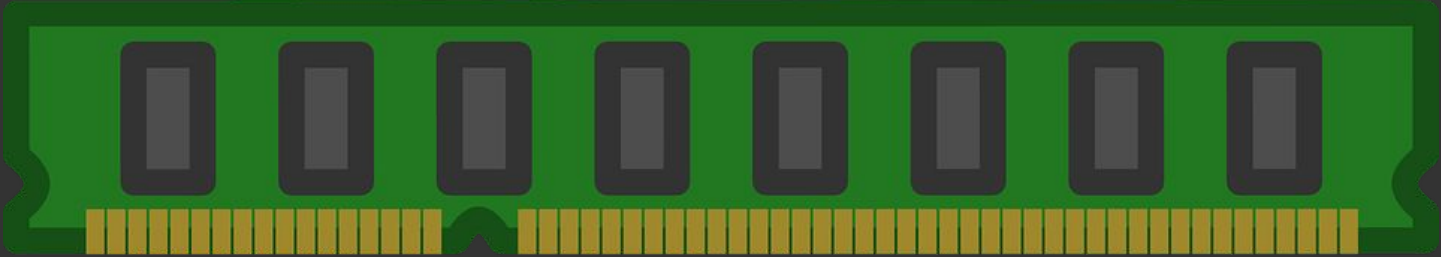


**Objects** are  
passed  
by  
**reference**

**What** **is**  
**that**  
**mean?**



# memory chip





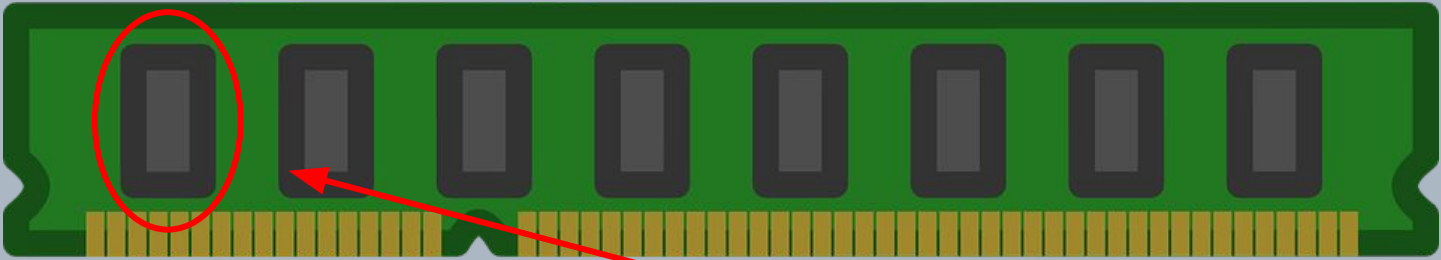
```
1 const person = { name: 'John Doe' }
```



```
1 const person = { name: 'John Doe' }
```

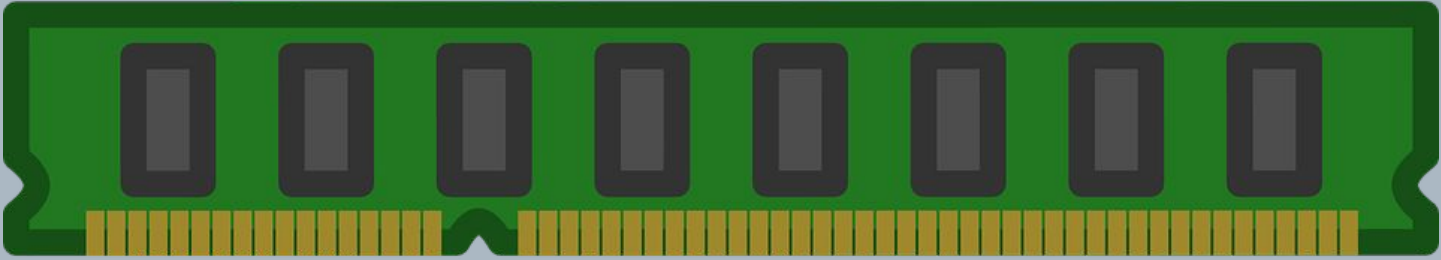
**new object**





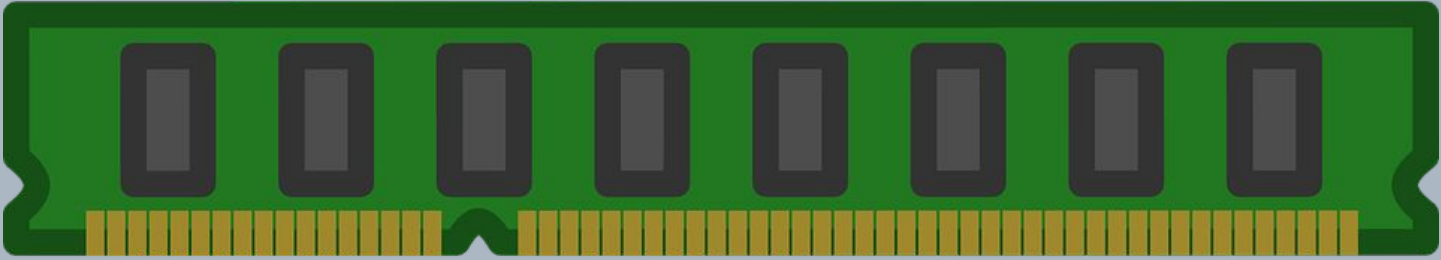
```
1 const person = { name: 'John Doe' }
```

**new object has  
a memory  
reference**

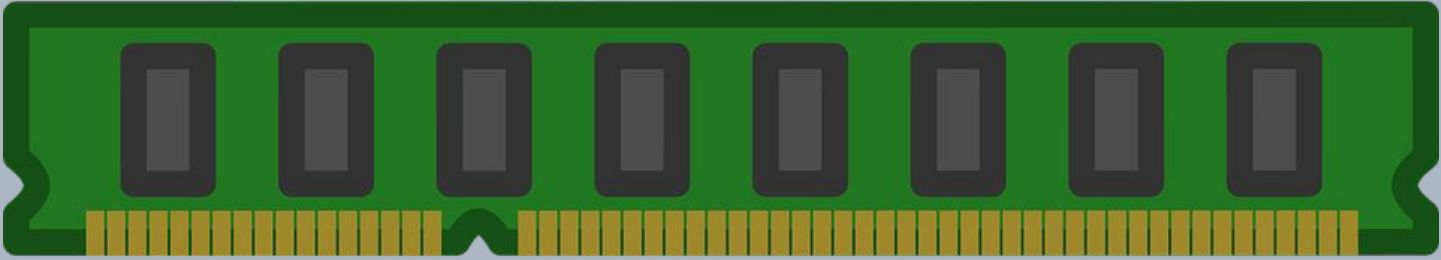


```
1 const person = { name: 'John Doe' }
```

**variable name is just an “alias” (pointer)**

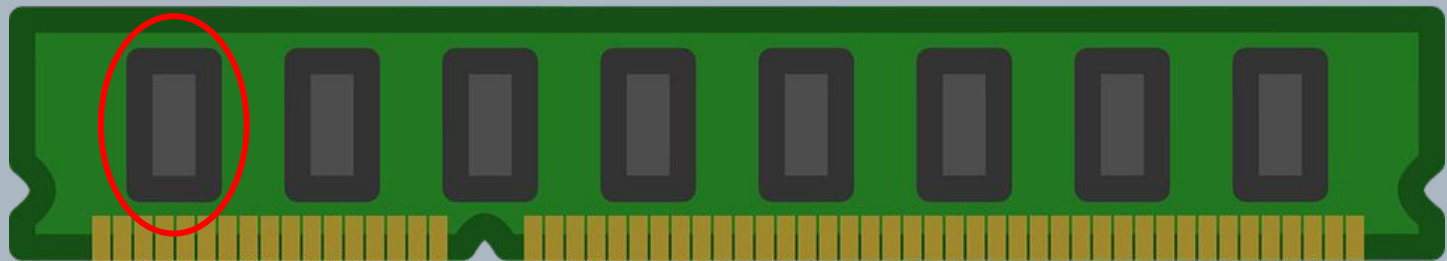


```
1 const person = { name: 'John Doe' }  
2  
3 const jane = person
```



```
1 const person = { name: 'John Doe' }  
2  
3 const jane = person
```

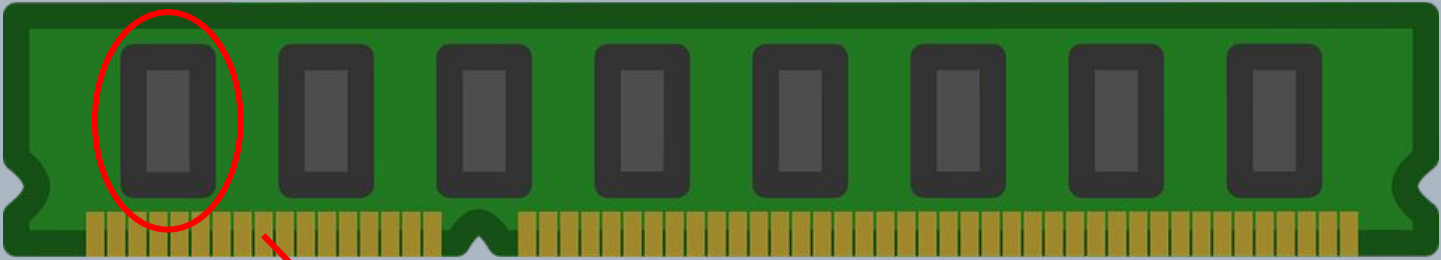
**two alias  
same object**



```
1 const person = { name: 'John Doe' }  
2  
3 const jane = person
```

**two alias  
same object**





```
1 const person = { name: 'John Doe' }
2
3 const jane = person
```

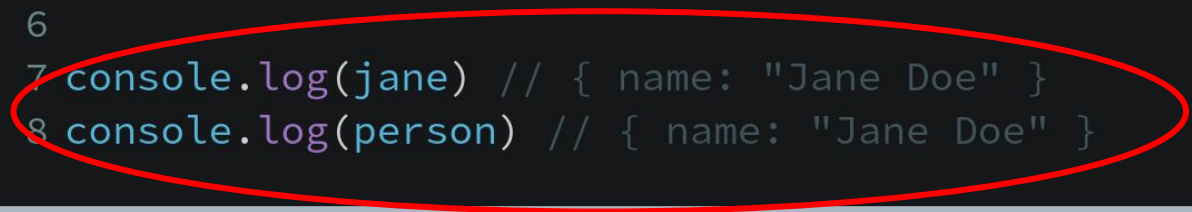
**two alias  
same object**



```
1 const person = { name: 'John Doe' }  
2  
3 const jane = person  
4  
5 jane.name = 'Jane Doe'  
6  
7 console.log(jane) // { name: "Jane Doe" }  
8 console.log(person) // { name: "Jane Doe" }
```



**that's why  
when you  
change one,  
the other one  
also changes**





```
1 console.log(jane === person) // true
```

---



```
1 console.log(jane === person) // true
```

**comparisons  
are always true**

**One**  
**more**

**example**



```
1 const personOne = { name: 'John Doe' }  
2 const personTwo = { name: 'John Doe' }  
3  
4 console.log(personOne === personTwo) // false
```

```
1 const personOne = { name: 'John Doe' }
2 const personTwo = { name: 'John Doe' }
3
4 console.log(personOne === personTwo) // false
```

**both objects  
“look” the  
same**



```
1 const personOne = { name: 'John Doe' }  
2 const personTwo = { name: 'John Doe' }  
3  
4 console.log(personOne === personTwo) // false
```

**but  
comparison  
is false**







```
1 const personOne = { name: 'John Doe' }  
2 const personTwo = { name: 'John Doe' }  
3  
4 console.log(personOne === personTwo) // false
```

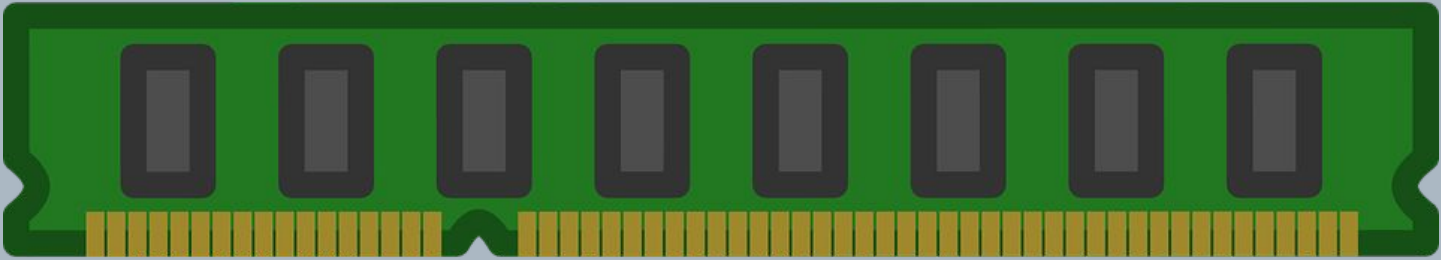
**it is an  
object**





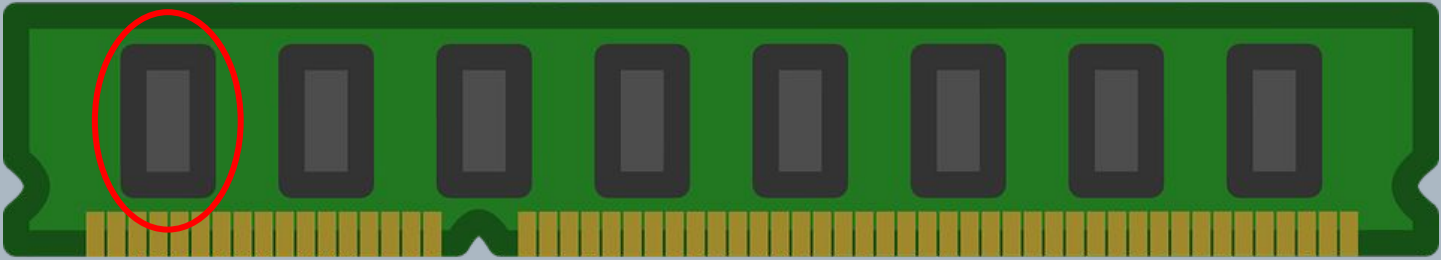
```
1 const personOne = { name: 'John Doe' }  
2 const personTwo = { name: 'John Doe' }  
3  
4 console.log(personOne === personTwo) // false
```

**it is  
another  
one**



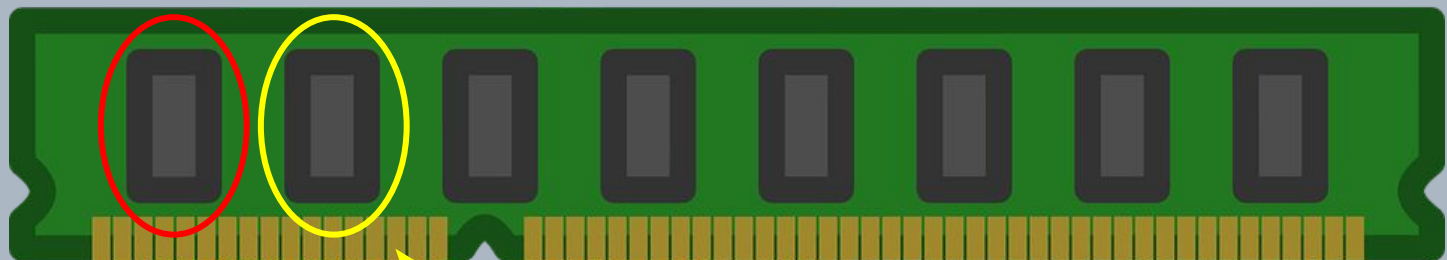
```
1 const personOne = { name: 'John Doe' }  
2 const personTwo = { name: 'John Doe' }  
3  
4 console.log(personOne === personTwo) // false
```

**each one  
has its  
own  
memory  
space**



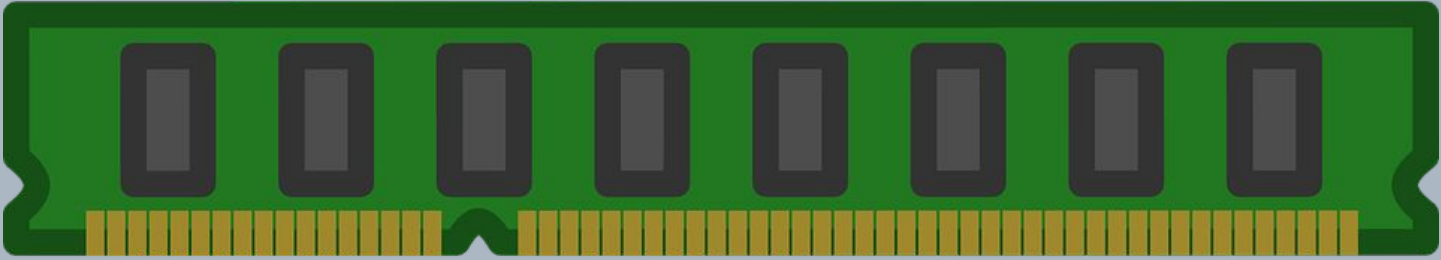
```
1 const personOne = { name: 'John Doe' }  
2 const personTwo = { name: 'John Doe' }  
3  
4 console.log(personOne === personTwo) // false
```

**each one  
has its  
own  
memory  
space**



```
1 const personOne = { name: 'John Doe' }  
2 const personTwo = { name: 'John Doe' }  
3  
4 console.log(personOne === personTwo) // false
```

**each one  
has its  
own  
memory  
space**



```
1 const personOne = { name: 'John Doe' }  
2 const personTwo = { name: 'John Doe' }  
3  
4 console.log(personOne === personTwo) // false
```

that's why  
that  
comparison  
is false

**So, how to**  
**avoid**  
**mutability**  
**with objects?**



```
1 const person = Object.freeze({ name: 'John Doe' })  
2  
3 person.name = 'Jane Doe'  
4  
5 console.log(person) // { name: "John Doe" }
```

---





```
1 const person = Object.freeze({ name: 'John Doe' })
2
3 person.name = 'Jane Doe'
4
5 console.log(person) // { name: "John Doe" }
```

**you can't  
mutate a  
frozen object**



```
1 const person = Object.freeze({ name: 'John Doe' })
```

```
2
```

```
3 person.name = 'Jane Doe'
```

```
4
```

```
5 console.log(person) // { name: "John Doe" }
```

**you may try...**



```
1 const person = Object.freeze({ name: 'John Doe' })
2
3 person.name = 'Jane Doe'
4
5 console.log(person) // { name: "John Doe" }
```

**...but you won't  
be able to**



But **internal**

**objects** are

**new**

**references**



```
1 const user = Object.freeze({
2   name: 'John Doe',
3   posts: [{ title: 'Post 1' }, { title: 'Post 2' }]
4 })
5
6 user.posts[0].title = 'Post 1 updated'
7 console.log(user)
8 // {
9 //   name: 'John Doe',
10 //   posts: [{ title: 'Post 1 updated' }, { title: 'Post 2' }]
11 // }
```



```
1 const user = Object.freeze({
2   name: 'John Doe',
3   posts: [{ title: 'Post 1' }, { title: 'Post 2' }]
4 })
5
6 user.posts[0].title = 'Post 1 updated'
7 console.log(user)
8 // {
9 //   name: 'John Doe',
10 //   posts: [{ title: 'Post 1 updated' }, { title: 'Post 2' }]
11 // }
```

**posts are not  
frozen (inside a  
frozen object)**

```
1 const user = Object.freeze({
2   name: 'John Doe',
3   posts: [{ title: 'Post 1' }, { title: 'Post 2' }]
4 })
5
6 user.posts[0].title = 'Post 1 updated'
7 console.log(user)
8 // {
9 //   name: 'John Doe',
10 //   posts: [{ title: 'Post 1 updated' }, { title: 'Post 2' }]
11 // }
```

**so, it can be mutated**

```
1 const user = Object.freeze({
2   name: 'John Doe',
3   posts: [{ title: 'Post 1' }, { title: 'Post 2' }]
4 })
5
6 user.posts[0].title = 'Post 1 updated'
7 console.log(user)
8 // {
9 //   name: 'John Doe',
10 //   posts: [{ title: 'Post 1 updated' }, { title: 'Post 2' }]
11 // }
```

**so, it can be mutated**





The **best** way  
to **make**  
immutable  
**code...**

Is **thinking**  
in  
**immutability**

Instead of

**freeze**

an  
**object...**

**Just**  
**don't**  
**mutate it**

And **if** do  
**I need**  
to **change**  
an **object?**

Then  
**you** are  
going to  
**transform**  
it...

**And**  
**will create a**  
**new copy**  
**from it**



```
1 const ball = { diameter: 30, shape: 'circle' }  
2  
3 const soccerBall = Object.assign({}, ball, { diameter: 22 })  
4  
5 console.log(ball) // { diameter: 30, shape: 'circle' }  
6 console.log(soccerBall) // { diameter: 22, shape: 'circle' }  
7  
8 console.log(ball === soccerBall) // false
```



```
1 const ball = { diameter: 30, shape: 'circle' }  
2  
3 const soccerBall = Object.assign({}, ball, { diameter: 22 })  
4  
5 console.log(ball) // { diameter: 30, shape: 'circle' }  
6 console.log(soccerBall) // { diameter: 22, shape: 'circle' }  
7  
8 console.log(ball === soccerBall) // false new object
```



```
1 const ball = { diameter: 30, shape: 'circle' }
2
3 const soccerBall = Object.assign({}, ball, { diameter: 22 })
4
5 console.log(ball) // { diameter: 30, shape: 'circle' }
6 console.log(soccerBall) // { diameter: 22, shape: 'circle' }
7
8 console.log(ball === soccerBall) // false
```

**new object**



**mutate first  
argument**

```
1 const ball = { diameter: 30, shape: 'circle' }
2
3 const soccerBall = Object.assign({}, ball, { diameter: 22 })
4
5 console.log(ball) // { diameter: 30, shape: 'circle' }
6 console.log(soccerBall) // { diameter: 22, shape: 'circle' }
7
8 console.log(ball === soccerBall) // false
```



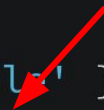
**empty object**

```
1 const ball = { diameter: 30, shape: 'circle' }
2
3 const soccerBall = Object.assign({}, ball, { diameter: 22 })
4
5 console.log(ball) // { diameter: 30, shape: 'circle' }
6 console.log(soccerBall) // { diameter: 22, shape: 'circle' }
7
8 console.log(ball === soccerBall) // false
```



```
1 const ball = { diameter: 30, shape: 'circle' }  
2  
3 const soccerBall = Object.assign({}, ball, { diameter: 22 })  
4  
5 console.log(ball) // { diameter: 30, shape: 'circle' }  
6 console.log(soccerBall) // { diameter: 22, shape: 'circle' }  
7  
8 console.log(ball === soccerBall) // false
```

**copy all  
props from  
ball**





**copy all props from  
new object**

```
1 const ball = { diameter: 30, shape: 'circle' }
2
3 const soccerBall = Object.assign({}, ball, { diameter: 22 })
4
5 console.log(ball) // { diameter: 30, shape: 'circle' }
6 console.log(soccerBall) // { diameter: 22, shape: 'circle' }
7
8 console.log(ball === soccerBall) // false
```



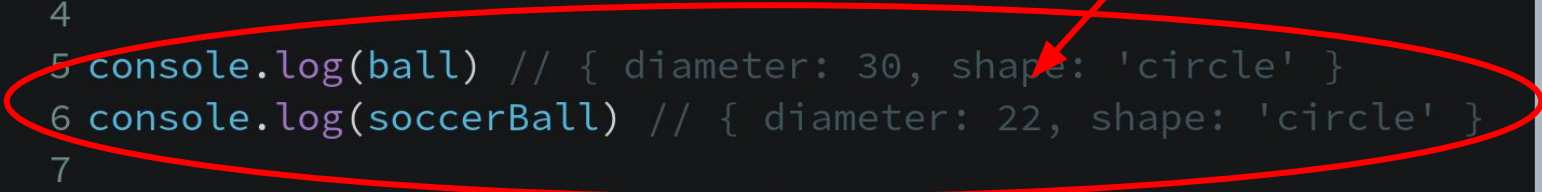
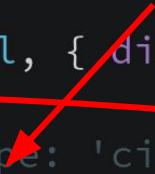
```
1 const ball = { diameter: 30, shape: 'circle' }
2
3 const soccerBall = Object.assign({}, ball, { diameter: 22 })
4
5 console.log(ball) // { diameter: 30, shape: 'circle' }
6 console.log(soccerBall) // { diameter: 22, shape: 'circle' }
7
8 console.log(ball === soccerBall) // false
```

**diameter exists on  
"ball" object**



```
1 const ball = { diameter: 30, shape: 'circle' }  
2  
3 const soccerBall = Object.assign({}, ball, { diameter: 22 })  
4  
5 console.log(ball) // { diameter: 30, shape: 'circle' }  
6 console.log(soccerBall) // { diameter: 22, shape: 'circle' }  
7  
8 console.log(ball === soccerBall) // false
```

**diameter was  
changed on  
"soccerBall"**







**“ball” object still  
the same**

```
1 const ball = { diameter: 30, shape: 'circle' }  
2  
3 const soccerBall = Object.assign({}, ball, { diameter: 22 })  
4  
5 console.log(ball) // { diameter: 30, shape: 'circle' }  
6 console.log(soccerBall) // { diameter: 22, shape: 'circle' }  
7  
8 console.log(ball === soccerBall) // false
```



**two different  
objects**

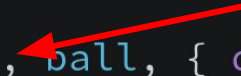
```
1 const ball = { diameter: 30, shape: 'circle' }  
2  
3 const soccerBall = Object.assign({}, ball, { diameter: 22 })  
4  
5 console.log(ball) // { diameter: 30, shape: 'circle' }  
6 console.log(soccerBall) // { diameter: 22, shape: 'circle' }  
7  
8 console.log(ball === soccerBall) // false
```



```
1 const ball = { diameter: 30, shape: 'circle' }
2
3 const soccerBall = Object.assign({}, ball, { diameter: 22 })
4
5 console.log(ball) // { diameter: 30, shape: 'circle' }
6 console.log(soccerBall) // { diameter: 22, shape: 'circle' }
7
8 console.log(ball === soccerBall) // false
```

instead of

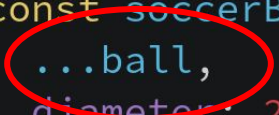
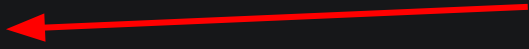
**Object.assign...**





```
1 const ball = { diameter: 30, shape: 'circle' }
2
3 const soccerBall = {
4   ...ball,
5   diameter: 22
6 }
7
8 console.log(ball) // { diameter: 30, shape: 'circle' }
9 console.log(soccerBall) // { diameter: 22, shape: 'circle' }
10
11 console.log(ball === soccerBall) // false
```

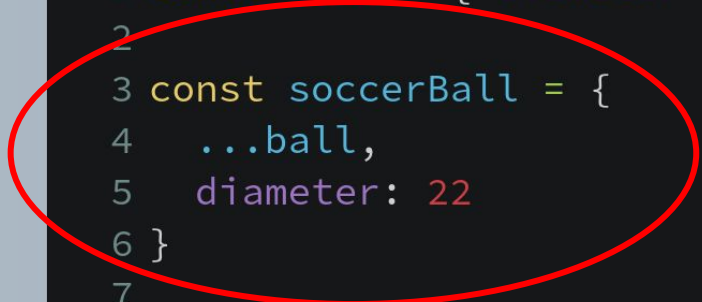
**we may use the new  
“spread operator”**





```
1 const ball = { diameter: 30, shape: 'circle' }
2
3 const soccerBall = {
4   ...ball,
5   diameter: 22
6 }
7
8 console.log(ball) // { diameter: 30, shape: 'circle' }
9 console.log(soccerBall) // { diameter: 22, shape: 'circle' }
10
11 console.log(ball === soccerBall) // false
```

**it spreads all props  
on a new object**





```
1 const ball = { diameter: 30, shape: 'circle' }
2
3 const soccerBall = {
4   ...ball,
5   diameter: 22
6 }
7
8 console.log(ball) // { diameter: 30, shape: 'circle' }
9 console.log(soccerBall) // { diameter: 22, shape: 'circle' }
10
11 console.log(ball === soccerBall) // false
```

**next props will be changed**

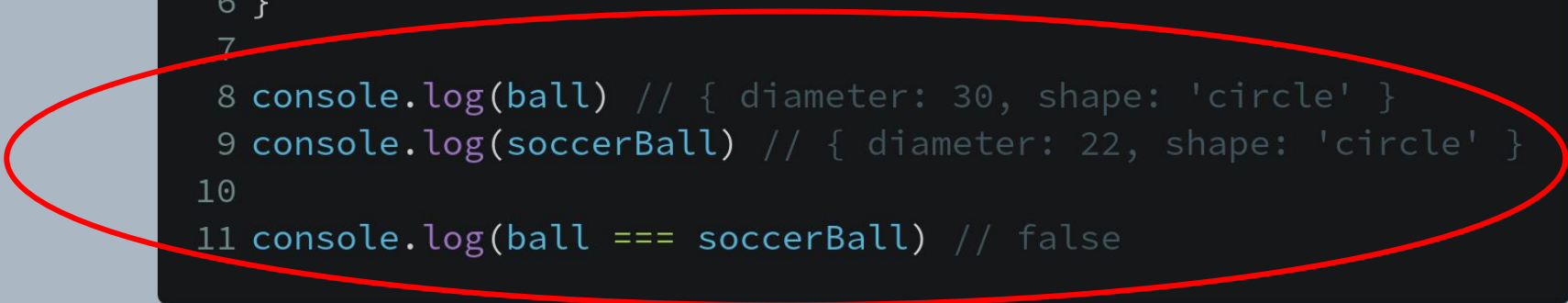
diameter: 22





```
1 const ball = { diameter: 30, shape: 'circle' }
2
3 const soccerBall = {
4   ...ball,
5   diameter: 22
6 }
7
8 console.log(ball) // { diameter: 30, shape: 'circle' }
9 console.log(soccerBall) // { diameter: 22, shape: 'circle' }
10
11 console.log(ball === soccerBall) // false
```

**results are  
the same**



The **same** goes

for **any** kind of

**object**





```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {
4   array.push(value)
5   return array
6 }
7
8 const array1234 = addInArray(array123, 4)
9 console.log(array1234) // [1, 2, 3, 4]
10
```



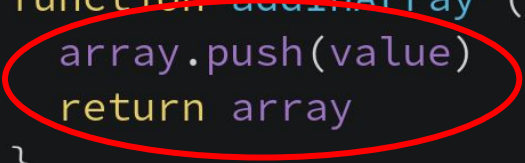
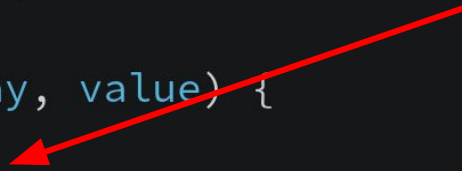
```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {
4   array.push(value)
5   return array
6 }
7
8 const array1234 = addInArray(array123, 4)
9 console.log(array1234) // [1, 2, 3, 4]
10
```

**remember  
that?**



```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {
4   array.push(value)
5   return array
6 }
7
8 const array1234 = addInArray(array123, 4)
9 console.log(array1234) // [1, 2, 3, 4]
10
```

**let's make  
that code  
immutable**





```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {
4   return array.concat(value)
5 }
6
7 const array1234 = addInArray(array123, 4)
8
9 console.log(array1234) // [1, 2, 3, 4]
10 console.log(array123) // [1, 2, 3]
```



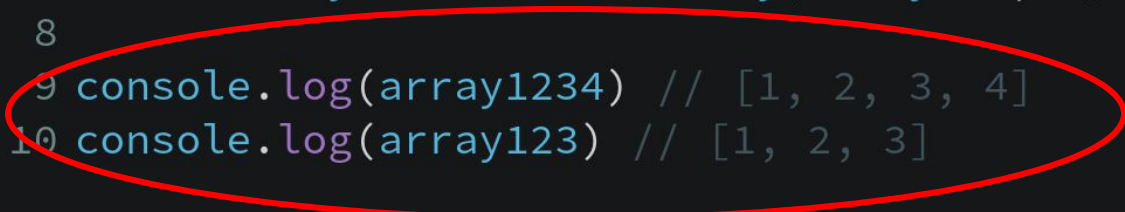
```
1 const array123 = [1, 2, 3]
2
3 function addToArray (array, value) {
4   return array.concat(value)
5 }
6
7 const array1234 = addToArray(array123, 4)
8
9 console.log(array1234) // [1, 2, 3, 4]
10 console.log(array123) // [1, 2, 3]
```

**just change  
“push” to  
“concat” and  
return it**



```
1 const array123 = [1, 2, 3]
2
3 function addInArray (array, value) {
4   return array.concat(value)
5 }
6
7 const array1234 = addInArray(array123, 4)
8
9 console.log(array1234) // [1, 2, 3, 4]
10 console.log(array123) // [1, 2, 3]
```

**everything is  
working without  
side effects**



**Immutability**

**in practice**

Instead of

**array.push**

use

**array.concat**



Instead of

**array.splice**

use

**array.slice**

Instead of

**array.pop** and

**array.shift**

use

**array.filter**

**Create**

**a new array**

**before using**

**mutable**

**methods...**

like  
**array.sort**  
and  
**array.reverse**  
for example

**Instead of**  
**loops (for/while)**  
**use**  
**array methods**

**array.map**

**array.filter**

**array.find**



**array.reduce**

**array.some**

**array.every**

Transform  
**objects** in  
arrays  
before **iterate:**

**Object.keys()**

**Object.values()**

**Object.entries()**

Or **prefer**  
to use

the **new**  
**for of**



Should I use  
**immutability**  
**everywhere** in  
my app?

**No!**

Just **try** to keep

mutable **state**

**isolated**

If you find

**bugs,** they will

**probably**

be on **that** state

**But**

**creating**

**more objects**

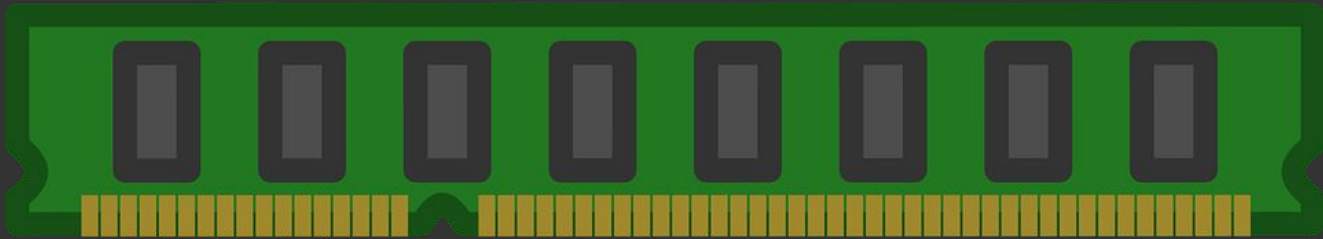
**spend more**

**memory...**

**What**  
**about**  
**performance?**

**Don't**  
**worry!**

**Memory** chips  
are **cheap**





With **little data,**

it makes

**no difference**

whether or not

to use

**immutability**

**And  
with  
lots of  
data...**

**Neither** mutable  
nor **immutable**  
**code** will  
**help you**

**Worry**  
about

**perceived**  
**performance**

Use

**async**

code



**How to**  
**learn**  
**more**  
**about**  
**immutability?**

# Practicing

a **lot!**





And  
**getting**  
involved

with the  
**community**



 ***/training-center/sobre***



 ***/frontendbr/forum***

<https://bit.ly/frontninja>

**You will**

**always**

**find help**

As long  
as **you**  
have...

**Respect.**



*Fernando Daciuk*

---

`$ npm install fdaciuk`

 /fdaciuk/talks