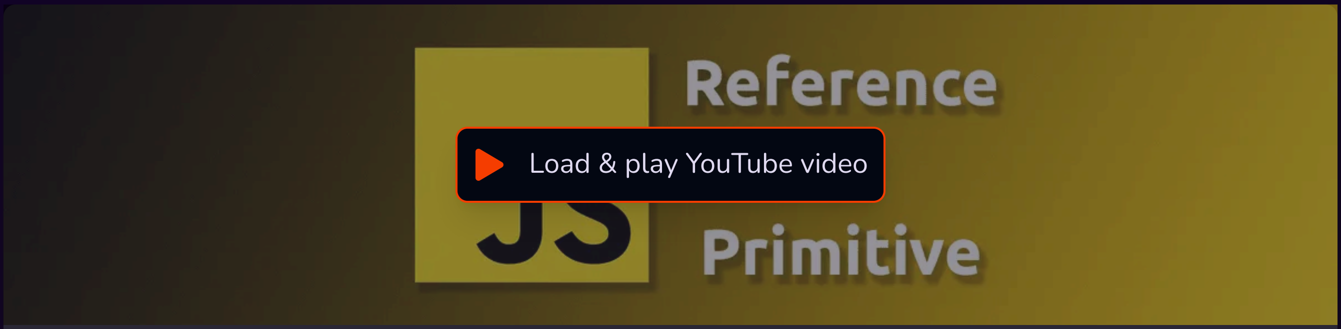
ACADEMIND



Reference vs Primitive Values

Learn why most people copy objects and arrays in JavaScript incorrectly. And why you won't make that mistake!

Created by Maximilian Schwarzmüller

April 9, 2017

Watch Video

# What are Primitives?

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

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

Here's an example:

```
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:

```
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.

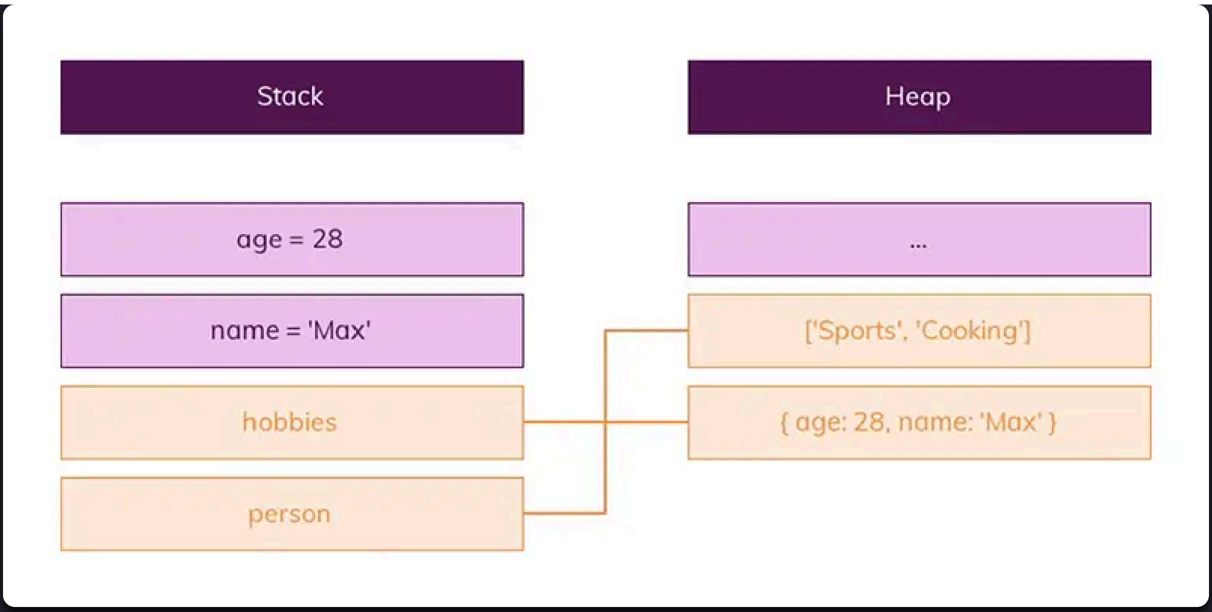
JavaScript knows two types of memory: The **Stack** and the **Heap**. You can dive much deeper if you want to.

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()` 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)

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 one 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

Oct 23 52.5h

Was the article helpful?

In that case, you might also find our **monthly insights** newsletter helpful!

We curate & share industry news, trends and anything else you need to know.

Enter your email address

By registering, you confirm that you read & accepted our privacy policy.

Register

COURSES

All Courses

Academind Pro

OTHER RESOURCES

Tutorial Articles

Podcast

Academind Community

SUBSCRIBE TO OUR NEWSLETTER

Be the first to be informed about our new courses, articles & more.

Enter your email address

By registering, you confirm that you read & accepted our privacy policy.

Register

Impressum Imprint Privacy Policy