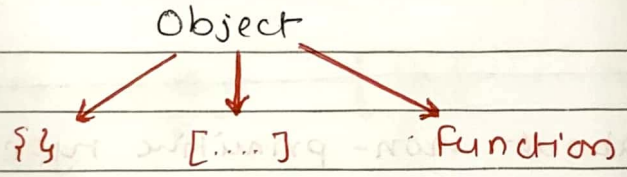
JavaScript - Day 7

TYPES in JavaScript :-

* **Primitive Types :-**
  - → number (5, 6, ...)
  - → boolean (true, false)
  - → undefined (undefined)
  - → null
  - → Symbol (new in ES6)
  - → string ('hi', 'hey', ...)

* **Non Primitive Types :-**



- **What's the difference?**

→ Primitive type is directly stored in the memory (var a = 5) 'a' holds the value, 5.

→ Non-primitive type does not directly hold the value, it holds the reference to that value in memory (like pointer).

* **Built in Objects** (comes with the language)

→ We already have primitive types like number, boolean, etc. So why are these built in object like Number, Boolean. given by language?

→ Every primitive type (number, string etc) have wrappers around them called Number(), string, etc.

- **Okay, but why complicated?**

```
false.toString();
```

output → false

WAIT...!!! false is a primitive type, then how can be run toString() on it?

- What JS does internally?
  Boolean(false).toString()

→ On so these built in objects like Number, Boolean, etc exists for us to run other methods on the primitive types, because we can't directly run the method on primitive type.

➡ So not everything in JS is an object, these are primitive types too, that are not objects.

⇒ Let's now talk about non-primitive types (Objects).

*1 Arrays are objects

Umm.... wait what?

→ Internally arrays are treated like objects as follows

let array = [1, 4, 5]	Internally
	var array = {
	0: 1
	1: 4
	2: 5
	}

Index ⇒ array element

So if you check
> typeof [1, 2, 3]
> object

→ so how do we differentiate and find if it's an

array if type of says it's an object.

Array.isArray([1, 2]) ⇒ true.

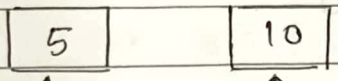
↓

Built in object that has is array property [Modern JS]

Pass by value & Pass by reference :-

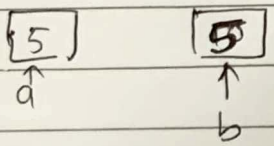
* Pass by value

let a = 5; let b = 10;

Memory ⇒ 

That means 'a' and 'b' actually hold the values we assigned them.

when we do, let a = 5; let b = a;



'a' has no contact with 'b' later a copy of 'a' is assigned to 'b' and 'b' is now independent. This is pass by value.

* Pass by reference :-

• Objects are passed by reference

```
let obj1 = { name: 'John',
             favFood: 'Burger'
           }
```

```
let obj2 = obj1;

obj2.favFood = 'pizza';
console.log (obj1, obj2);
```

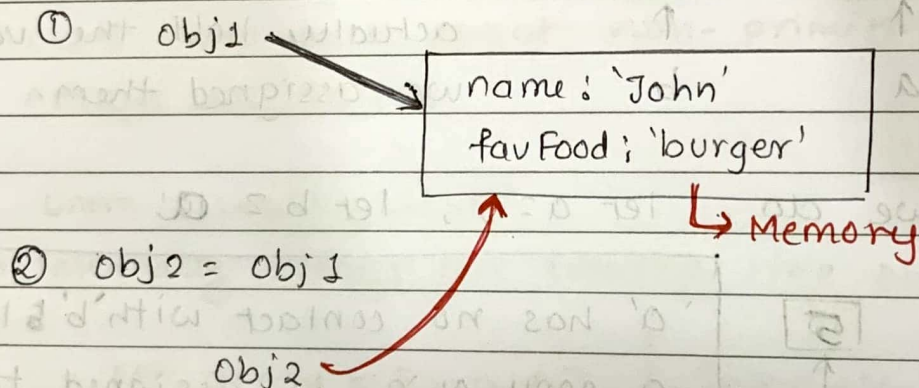
Output >>

obj1 ⇒ {name: 'John', favFood: 'pizza'}

obj2 ⇒ {name: 'John', favFood: 'pizza'}

→ wait, what? obj1 & obj2 are same but we only changed favFood property of obj2 right.

what happens internally



→ That means obj1 & obj2 have the same data, because they are both pointing to the same memory location.

→ So changing obj1 or obj2 will change data for both.

→ So objects are basically references to a some memory locations and obj2 = obj1 does not create a new memory space for obj2.

We are saving the memory space

→ But also it's confusing, what if someone wants to keep them separate?

→ Also remember, since arrays are objects, that ~~same~~ have the same behaviour.

Arrays solution	object solution
let a = [1, 2, 3]	let a = {'a': 0, 'b': 1}
let b = [].concat(a)	let b = object.assign({}, a);

• Exercise :-

Try doing the same with spread operator.

Soln let b = {...a}

what if we have nested objects?

```
let obj = {
        a: 1
        b: 2
        c: {
            nested: true.
        }
}
```

→ Try the above methods and do

```
obj2 = {...obj1};

obj2.c.nested = false;
```

→ you will see the value of nested will change to false in both objects.

→ Opps... We have a problem.
→ whatever approach, we tried on top does not work with nested object because every object is passed by reference and we only cloned the top layer.

" THIS IS CALLED SHALLOW CLONING "

→ But what we need now is deep cloning (all levels)

```
let obj2 = JSON.parse(JSON.stringify(obj1));
```

→ This does deep cloning. However if obj is too large there will be performance issue as parsing the whole object to all levels can take time.

* FUNCTIONS are Objects (JavaScript).

```
function sayHello()
{
    console.log("Hey");
}
sayHello.property = "Bye";
```

well functions are objects then, we should be able to add properties to it right?

How functions look internally

```
const funObject = {
  property1 : Bye,
  name : sayHello():
    console.log("Hey
}
```

So, that basically means, every function is represented as an object. It has all the properties of that function for example property1 that we just ~~declared~~ defined. It also has a name property that is the name of the function. (0 optional, since function can be anonymous, It has () which is basically used to invoke the function.

NOTE :- This example is just for an illustration to understand, the object could look differently.

It's important to understand this because we say functions can be passed as arguments (They are objects.).