

Functions are objects (JavaScript)

@codeWithSimran

```
function sayHello() {  
    console.log("Hey")  
}
```

```
SayHello.property1 = 'Bye';
```

Well if functions are objects
we should be able to add
Properties to it right?

How functions look internally

```
const functionObj = {  
    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 defined.
It also has a name property that is the
name of the function. (Optional since
function can be anonymous. It has
() which is basically used to invoke
the function.

@codeWithSimran

Note: This example is just 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're objects)



codeWithSimran



codeWithSimran_

Types in JavaScript

* Primitive type

→ number (5, 6...)

→ boolean (true, false)

→ undefined (undefined)

→ null

→ symbol (new in ES6)

→ string ('hi', 'hey!...')

* Non Primitive type

object

{ } [...] function

@codeWithSimran

What's the difference?

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

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



codeWithSimran



codeWithSimran_

Built in Objects (come with the language)

we already have primitive types like number, boolean etc

So why are there built-in objects like **Number, Boolean** given by the language?

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

Okay but why complicate?

> `false.toString();` @codeWithSimran
→ 'false'

Wait!! false is a primitive type
how can be run toString() on it?

What JS does internally
`Boolean(false).toString()`



codeWithSimran



codeWithSimran_

Oh so these built in objects like Number, Boolean etc exist for us to ~~run~~ 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, there are primitive types too that are not Objects

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

* 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  
}
```

@codewithSimran



codeWithSimran



codeWithSimran_

So if you check

```
> type of [1,2,3]
```

@codeWithSimran

> object

So how do we differentiate and find if its an array if type of says its an object.

`Array.isArray([1, 2])` → true



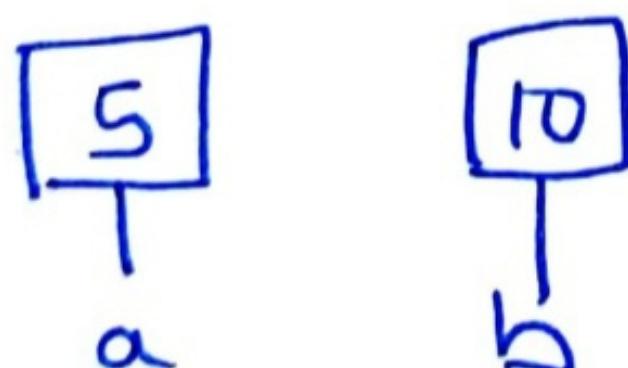
Built-in Objekt
that has isArray property [JS] [modern]

Pass by value & Pass by reference

* Pass by value

Let $a = 5$; let $b = 10$

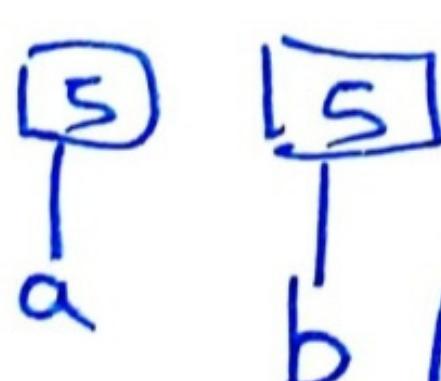
memory



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

* When we do

let a = 5; let b = a;



\Rightarrow ; let $b = a$;
a has no contact with b later

a has no contact with b
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; // @codewithsimran
```

```
obj2.favFood = 'pizza';
```

```
console.log(obj1, obj2)
```

Obj1

↳ { name: 'John', favFood: 'pizza'}

Obj2

↳ { name: 'John', favFood: 'pizza'}

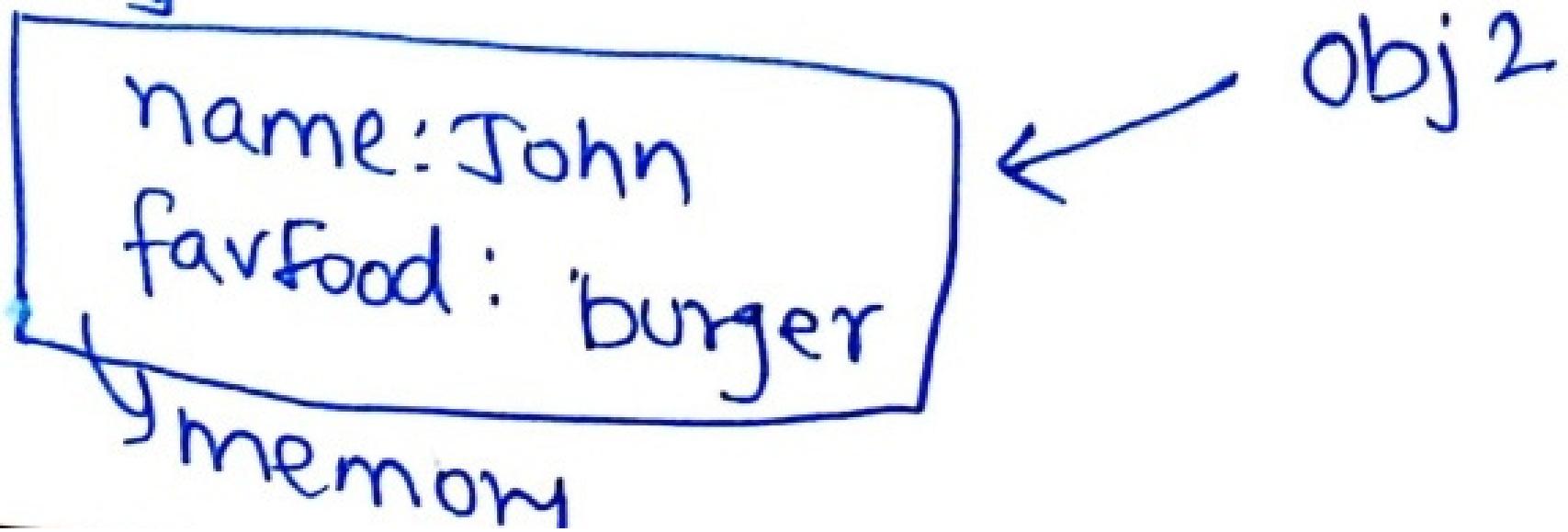
Wait, what? Obj1 and obj2
are same but we only changed
favFood property of obj2 right?

What happens internally

①

obj1

② obj2 = obj1



codeWithSimran



codeWithSimran_

That means obj1 and obj2 have the same data, because they're both pointing to the same memory location.

So changing obj1 or obj2 will change data for both.

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

We're saving memory space 

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

@codeWithSimran

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

Arrays soln

```
let a = [1,2,3]  
let b = [].concat(a)
```

Object soln

```
let a = {'a':0,'b':1}  
let b = Object.assign({},a)
```



codeWithSimran



codeWithSimran_

Exercise:

Try doing the same with spread operator.

Soln let b = { ... a }

What if we have nested objects?

```
let obj1 = {  
    a : 1  
    b : 2  
    c : {  
        nested : true  
    }  
}
```

@codewithSimran

Try the above methods and do

obj2 = ~~obj1~~ { ... obj1 }

obj2.c.nested = false;

You'll 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.

@codeWithSimran



codeWithSimran



codeWithSimran_