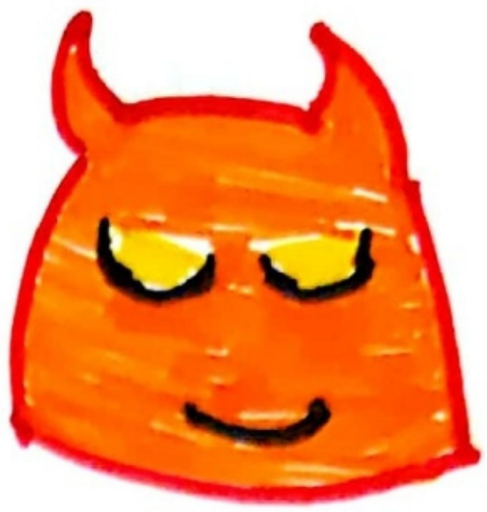
THIS KEYWORD



This refers to the object on which we call our function:

Example

② sayHello has 'this' (just like all other functions) and 'this's value is obj

obj.sayHello(this)

① obj is the object on which we're calling sayHello

@codeWithSimran

* Inside of a function the value of 'this' is the object we used to call the function.

What if we simply say
> sayHello(), we're not calling sayHello on any object.
> In this case, 'this' refers to the window object.

codeWithSimran codeWithSimran_

* So just saying sayHello() is like saying window.sayHello()

① Look before the dot what's there?

@codeWithSimran

SomeObject . SomeFunction()

② It's someObject → the value of 'this'

③ If there is not . means directly calling function, then look at it as

window . someFunction()

what's b4 before the dot is window internally.

@codeWithSimran

codeWithSimran codeWithSimran_

* On global execution context the value of 'this' is window

Example 1

```
function sayHello() {
    console.log(this)
}
sayHello()
```

> window [Because sayHello is called without any object, iternally it's window (window.sayHello())

Example 2

@codeWithSimran

```
const obj = {
    name : 'Simran'
    sayHello : function(){
        return 'Hi' + this.name
    }
}
```

If we want access to name in sayHello this is how we can do it. Since sayHello is a part of obj, that value of 'this' inside sayHello is 'obj'

codeWithSimran codeWithSimran_

To run it we will use

obj.sayHello()

And what's before that.? It's 'obj' ,that means value of 'this' inside sayHello should be 'obj'

Example 3

@codeWithSimran

* Execute same code for multiple object (Reuse)

```
function sayHello() {
        console.log(this.name)
}

const obj1 = {
        name : 'Simran
        sayHello : sayHello
    }
const obj2 = {
            name : 'John'
            sayHello : sayHello
    }
```

codeWithSimran codeWithSimran_

```
const name = 'Lara';
```

> window.sayHello() OR sayHello()
→ Lara

> obj1.sayHello()
→ Simran

> obj2.sayHello()
→ John

@codeWithSimran

Conclusion

* This gives methods (functions inside your object) access to the object they're defined in
* Let's us execute same code (sayHello) by multiple objects (obj1 and obj2)

codeWithSimran codeWithSimran_

Example 4

```
Const a = function() {
    console.log('a', this)
    const b = function() {
        console.log('b', this)
        const c = {
            sayHello : function() {
                console.log('c', this)
            }}
        c.sayHello()
    }
    b()
}
a()
```

@codeWithSimran

- a → Window
  (because there is nothing before a()
  its effectively window.a())
- b → Window
  (same case as a)

codeWithSimran codeWithSimran_

Note: value of 'this' inside b is window means it DOES NOT MATTERS WHERE WE DEFINED FUNCTION B, what matters is how it's called

But Javascript follows lexical (static) scoping right? Scope of a function is decided by where it is defined and not where it's called. @codeWithSimran

No wonder 'this' is like an alien, that follows dynamic scope, value of this depends on how we called the function not where it is actually defined.

codeWithSimran codeWithSimran_

▶ c → c 'this'
sayHello

Inside ▲ the value of 'e is the object c because we called sayHello as

c.sayHello()

@codeWithSimran

Example 5

```
const obj = {
    name: 'Simran'
    sayHello: sayHello()
        console.log('Hello', this);
```

Example 5.

```
const obj = {
    name: 'Simran'
    sayHello: function() {
        console.log('Hello', this)
        var sayBye = function() {
            console.log('Bye', this)
        }
        sayBye()
}
}
```

codeWithSimran codeWithSimran_

> obj.sayHello()

- Hello obj
- Bye Window

So how do we solve this problem and make it work as lexically scoped, value of this should depend on where the function was defined because that's how rest of javascript works

Soln Arrow functions 

if you convert sayBye to an arrow function out will be

- Hello obj
- Bye obj

@codeWithSimran

Why? Because inside arrow function value of this depends on where that arrow function is present or defined And its defined inside obj right? So value of this will be obj

codeWithSimran codeWithSimran_

Wait!! Arrow functions were introduced in ES6 right? What did people do before that?

* To get the same result with normal function as arrow function you can use the bind method

@codeWithSimran

~~sayBye()~~

sayBye.bind(this);

↓

you're basically binding the current value of this which is (obj) to sayBye as well.

codeWithSimran codeWithSimran_

In order to play with 'this' keyword, we have 3 methods.
call(), apply(), bind()

* Internally every function uses call() when it's invoked.

→ so when you call a function by sayHello()
you're basically doing
sayHello.call() [Internally]

Let's take an example of a game where you've a battery life (of player) and in some situations the player can charge itself to 100%.

```
const player1 = {
        name: 'John'
        battery: 62
        charge: function(){
                this.battery = 100
        }
}
```

@codeWithSimran

So basically player1 is an object with a method charge (to charge 100%.)

codeWithSimran codeWithSimran_

To simply charge player 1
we can do

→ player1.charge()
[battery: 100]

Now lets introduce a second
player, player2

```
const player2 = {
        name: 'Lara'
        battery: 20
}
```

@codeWithSimran

OKAY, player2 does not have
a charge method, so can it
borrow the same from player1
Also not we dont want to
repeat code that can be reused

→ call the charge method of player1

player1.charge.call(player2)

But call it on player2

codeWithSimran codeWithSimran_

We're esentially able to pass reference to player2 (which will be value of this) inside battery method.

So argument to call is overding what the value of 'this' will be when that method is 'invoked.

@codeWithSimran

*Also you can pass arguments using call which will be recieved by battery method.

player1.charge.call(player2, 20,30,40)

arguments

and your battery function might look like this

battery: function (arg1, arg2, arg3)

NOTE: You don't need to accept player2 as an argument. 'this' already contains player2.

codeWithSimran codeWithSimran_

apply

It's pretty much same as call but the way it takes argument is different.

It takes an array of arguments

```
player1.charge.apply(player2,[20,30,40])
```

takes an array of arguments

@codeWithSimran

bind()

call and apply immediatly invoke the function, whereas bind returns a new function binded with a new *this* which we can run later

```
const callLater =
player1.charge.bind(player2,20,30,40)
[syntax same as call]
to actually call it we can later do
  callLater()
```

codeWithSimran codeWithSimran_