

"How JS Works Behind the Scene"

This keyword: →

→ This keyword in global will point to window object which is the Parent.

```
const calcAge = function (birthYear) {  
  console.log(2037 - birthYear);  
  console.log(this);  
};  
calcAge(1991);
```

* in Regular Function it will be undefined.

```
const calcAgeArrow = birthYear => {  
  console.log(2037 - birthYear);  
  console.log(this);  
};  
calcAgeArrow(1980);
```

* in arrow function it uses the this of its Parent which is global which is window.

"This keyword inside methods"

```
const jonas = {  
  year: 1991,  
  calcAge: function () {  
    console.log(this);  
  },  
};  
jonas.calcAge();
```

* In Case of Methods the this keyword will point to the Parent object of the method which is jonas.

```
const jonas = {  
  year: 1991,  
  calcAge: function () {  
    console.log(this);  
    console.log(2037 - this.year);  
  },  
};  
jonas.calcAge();
```

```
const matilda = {  
  year: 2017,  
};  
  
matilda.calcAge = jonas.calcAge;  
matilda.calcAge();
```

here this will point to Matilda.

here we borrow the calcAge method From Jonas to Matilda so that

```

calcAgeArrow(1980);

const jonas = {
  year: 1991,
  calcAge: function () {
    console.log(this);
    console.log(2037 - this.year);
  },
};
jonas.calcAge();

const matilda = {
  year: 2017,
};

matilda.calcAge = jonas.calcAge;
matilda.calcAge();

const f = jonas.calcAge;
f();

```

* Here we assign the calcAge method of jonas to the f var but when we call it the this keyword will give me undefined because in this case this function doesn't belong to any one, it is just Regular Function.


```

const calcAge = function (birthYear) {
  console.log(2037 - birthYear);
  console.log(this);
};
calcAge(1991);

```

The above f function will be treated like this which is Regular Function which give me undefined.

"Regular Function vs arrow Function"




```
const jonas = {
  firstName: 'Jonas',
  year: 1991,
  calcAge: function () {
    console.log(this);
    console.log(2037 - this.year);
  },
  greet: () => console.log(`Hey ${this.firstName}`),
};
jonas.greet();
```

* The output here will be "Hey undefined"

* That because this is an arrow function and the this keyword will be attached to its parent

which is the window and window doesn't contain anything related to firstName.




```
var firstName = 'Matilda';

const jonas = {
  firstName: 'Jonas',
  year: 1991,
  calcAge: function () {
    console.log(this);
    console.log(2037 - this.year);
  },
  greet: () => {
    console.log(this);
    console.log(`Hey ${this.firstName}`);
  },
};
jonas.greet();
```

• The output here will be "Hey matilda"

That's because the var will add the firstName as a property in the window



As this is an arrow function, it will relate this keyword to window, hence window.firstName is defined because of var.

```
const jonas = {
  firstName: 'Jonas',
  year: 1991,
  calcAge: function () {
    // console.log(this);
    console.log(2037 - this.year);

    const isMillenial = function () {
      console.log(this);
      console.log(this.year >= 1981 && this.year <= 1996);
    };
    isMillenial();
  },
  greet: () => {
    console.log(this);
    console.log(`Hey ${this.firstName}`);
  },
};
jonas.greet();
jonas.calcAge();
```

why this here prints undefined?

its not an arrow function
So that this must be related
to jonas object Right?

* That doesn't apply here
because in the last it is
considered a regular Funct
and this keyword in
Regular Function will
result to undefined

"There is two solution"

Q First: →

assign this to a variable Before declaring the function
and in this case self has access to the this related
to jonas object.

```
const jonas = {
  firstName: 'Jonas',
  year: 1991,
  calcAge: function () {
    // console.log(this);
    console.log(2037 - this.year);

    const self = this; // self or that
    const isMillenial = function () {
      console.log(self);
      console.log(self.year >= 1981 && self.year <= 1996);
      // console.log(this.year >= 1981 && this.year <= 1996);
    };
    isMillenial();
  },
  greet: () => {
    console.log(this);
    console.log(`Hey ${this.firstName}`);
  },
};
jonas.greet();
jonas.calcAge();
```

as it is here

* The second: →

The second solution is to use arrow function and as you know it inherits the this from its parent scope which is Jonas object.

Jonas object ↗ Parent scope
Arrow function = > { this
j j

↪ arguments doesn't exist in arrow function.

"Hoisting and TDZ in Practice"

```
console.log(me);  
console.log(job);  
console.log(year);
```

```
var me = 'Jonas';  
let job = 'teacher';  
const year = 1991;
```

* var is hoisted with initial value of undefined.

* let and const follow the Temporal dead zone.

↳ Can't be used before declared.

```
// Functions
```

```
console.log(addDecl(2, 3));  
console.log(addExpr(2, 3));  
console.log(addArrow(2, 3));
```

```
function addDecl(a, b) {  
  return a + b;  
}
```

```
const addExpr = function (a, b) {  
  return a + b;  
};
```

```
const addArrow = (a, b) => a + b;
```

* Function declaration can be used before initialization.

* unlike with Regular and arrow functions.

* Const → TDZ

* If I change const to var what do you expect?

↳ It's gonna tell you addExp is not a function that's because undefined will be stored in it so how to pass undefined function.

* Remember that var will be added to window unlike let and const.

"Primitive vs Reference Type"

* Primitive: → number, string, Boolean, undefined, ...
→ it stored in call stack.

* Reference Type: — such as object, functions, arrays
→ They are stored in Heap.

```
let age = 30;  
let oldAge = age;  
age = 31;  
console.log(age); // 31  
console.log(oldAge); // 30
```

address → value
age → 001 → 30
old age → 001 → 30

when age = 31

old age → 001 → 30
age → 002 → 31

This how Primitive works.

```
const me = {  
  name: 'Jonas',  
  age: 30,  
};  
const friend = me;  
friend.age = 27;  
console.log('Friend:', friend);  
console.log('Me', me);
```

this in side stack
me → 001 → x
friend → 001 → x
↑ address of obj in the heap.
Jonas 30

so if I change either the friend or me it will be reflected in the heap, it doesn't work the same way of primitive.

How it looks?

Identifier	Address	Value
age	0001	30 31
oldAge	0002	31
me	0003	D30F
friend		

CALL STACK

Address	Value
D30F	{ name: 'Jonas'; age: 30 ; }

Reference to memory
address in Heap

27



```
const jessica = {
  firstName: 'Jessica',
  lastName: 'Williams',
  age: 27,
};
const marriedJessica = jessica;
marriedJessica.lastName = 'Davis';
console.log('Before marriage:', jessica);
console.log('After marriage: ', marriedJessica);

// marriedJessica = {};
```

* This will point to new empty object which is not allowed in const.
 * It will change address.


it is const and still be able to change the property why?

Because these are stored in the heap and the only thing which stored in stack is the address. which I don't change.

Note that: → Jessica and marriedJessica Both of them their lastName will be David.

That's Because Both of them are nick names for the same address.

if I want to make a copy of Jessica to a new address, How to achieve that?



```
const jessica2 = {
  firstName: 'Jessica',
  lastName: 'Williams',
  age: 27,
  family: ['Alice', 'Bob'],
};

const jessicaCopy = Object.assign({}, jessica2);
jessicaCopy.lastName = 'Davis';

jessicaCopy.family.push('Mary');
jessicaCopy.family.push('John');

console.log('Before marriage:', jessica2);
console.log('After marriage: ', jessicaCopy);
```

The only problem with object.assign that it copy only the first level only not every thing.

* This mean if there is another object in the Parent object it won't be Copy and it will be the same in origin and Copy.

* Note that: arrays is considered object.

```
jessicaCopy.lastName = 'Davis';  
jessicaCopy.family.push('Mary');  
jessicaCopy.family.push('John');
```

lastName will change in the Copied only.

As firstName is Considered first level.

Mary and John will be pushed in origin and copied version because it is not Considered First level.

* object.assign only Copy the first level of the object.