

This, objects & more...

this keyword

this

- Στο JavaScript, η λέξη-κλειδί **this** αναφέρεται σε ένα **object**.
- Ποιο αντικείμενο εξαρτάται από τον τρόπο με τον οποίο χρησιμοποιείται το **this**
- Η λέξη-κλειδί **this** αναφέρεται σε διαφορετικά αντικείμενα ανάλογα με τον τρόπο με τον οποίο χρησιμοποιείται

this Keyword

- Μόνο του, το this αναφέρεται στο καθολικό αντικείμενο (window στον browser ή global nodejs).
- Σε μια συνάρτηση(function), το **this** αναφέρεται στο καθολικό αντικείμενο .
- Όταν η συνάρτηση είναι **μέθοδος** ενός αντικειμένου, τότε το “this” αναφέρεται στο **αντικείμενο** στο οποίο η μέθοδος είναι μέλος.

this Keyword

- Άρα το this έχει διαφορετικές τιμές ανάλογα με το πού χρησιμοποιείται:
- Μόνο του, το this αναφέρεται στο καθολικό (global) αντικείμενο .
- Σε μια συνάρτηση, το this αναφέρεται στο καθολικό αντικείμενο .
- Ένας τρόπος να διαπιστώσουμε τι θα είναι το this keyword σε ένα function είναι να δούμε τι καλεί τη συνάρτηση. Άμα καλούμε μια συνάρτηση από ένα αντικείμενο (μέθοδο) τότε το this αναφέρεται στο αντικείμενο, αν την καλούμε «μόνη» της, τότε αναφερόμαστε στο καθολικό (global) αντικείμενο
- Γενικά το 'this' αναφέρεται στο **object** που εκτελεί το current function

Καθολικό αντικείμενο (Global object)

- Στη JavaScript, υπάρχει πάντα ένα καθολικό αντικείμενο.
- Σε ένα web browser, όταν δημιουργούμε global μεταβλητές, αυτές δημιουργούνται ως μέλη του **καθολικού** αντικειμένου.
- Σε ένα web browser το **καθολικό** αντικείμενο αναφέρεται στο **παράθυρο** του προγράμματος περιήγησης(browser window). (Το window object αντιπροσωπεύει ένα ανοιχτό παράθυρο σε ένα πρόγραμμα περιήγησης.)

```
→ let x = this;  
document.getElementById("demo").innerHTML = x;
```

this σε συνάρτηση

- Σε μια συνάρτηση **this** αναφέρεται στο καθολικό αντικείμενο.
- Άρα τι περιμένουμε να δούμε στο παράδειγμα παρακάτω?

```
var myVar = 10;

alert("this.myVar = " + this.myVar); ?

function WhoIsThis() {
    var myVar = 20;

    alert("myVar = " + myVar); ?
    alert("this.myVar = " + this.myVar); ?
};

WhoIsThis();
```

this σε συνάρτηση

```
var myVar = 10;
```

```
//When a function is called without an owner object ->value of this becomes the global  
object ->in a web browser the global object is the browser window. (The window object  
represents an open window in a browser.)
```

```
function WhoIsThis() {  
    var myVar = 20;  
  
    alert("myVar = " + myVar); //20  
    alert("this.myVar = " + this.myVar); //10  
};
```

```
WhoIsThis(); // inferred as window.WhoIsThis()
```


this σε συνάρτηση

- Παρακάτω τι περιμένουμε να δούμε?

```
var myVar = 10;

//When a function is called without an owner object
alert("this.myVar = " + this.myVar);

function WhoIsThis() {
  var myVar = 20;
  function WhoIsThis2(){
    alert("this local = " + myVar);
    alert("this.myVar2 = " + this.myVar);
  }
  return WhoIsThis2();
};

WhoIsThis(); // inferred as window.WhoIsThis()
```

this σε συνάρτηση

- Παρακάτω τι περιμένουμε να δούμε: Ακόμη και σε nested function η λέξη-κλειδί "this" της JavaScript αναφέρεται στο global object

```
var myVar = 10;

//When a function is called without an owner object
alert("this.myVar = " + this.myVar); 10

function WhoIsThis() {
  var myVar = 20;
  function WhoIsThis2(){
    alert("this local = " + myVar); 20
    alert("this.myVar2 = " + this.myVar); 10
  }
  return WhoIsThis2();
};

WhoIsThis(); // inferred as window.WhoIsThis()
```

Objects: Functions

- JavaScript **methods** : ενέργειες που μπορούν να εφαρμοστούν σε objects.
- A JavaScript **method** : είναι ουσιαστικά μια ιδιότητα(property) ενός object που περιλαμβάνει μια συνάρτηση

```
<script>
var name="test";
var dog = {
  name:"Bob",
  eyes:"Blue",
  woof:function() { return "Woof, woof!";},
  sayone:function myfun() {return "I am a dog"},
  saytwo() {
    // shorter syntax for defining a function property
    return "And I bark";
  }
};

console.log(dog.woof());
console.log(dog.sayone());
console.log(dog.saytwo());
```

this Keyword

- Σε μια μέθοδο αντικειμένου, το **this** αναφέρεται στο αντικείμενο "ιδιοκτήτη" της μεθόδου : **this** is "cat".

```
// Create an object:
var cat = {
  name: "Javie",
  age : "4",
  eyesColor : "blue",
  des : function() {
    return this.name + " " + this.age;
  }
};

// Display data from the object:
document.getElementById("demo").innerHTML = cat.des();
</script>
```

- **this** -> is evaluated during the run-time, depending on the context
- What determines the value of **this** is **how** we call the function.
- If **this** is used outside of any function or if a function is not called as a method this refers to the global object

this Keyword

```
<script>
var name="test";
var dog = {
  name:"Bob",
  sometest:this.name,
  sometest1:this.kati,
  woof:function() { return "Woof, woof!";},
  sayName:function myfun() {return this.name;},
  saysometest() {
    // Method which will display sometest
    return this.sometest;
  }
};

console.log(dog.woof());
console.log(dog.sayName());
console.log(dog.saysometest());
console.log(dog.sometest1);
```

this Keyword

```
<script>
var name="test";
var dog = {
  name:"Bob",
  sometest:this.name,
  sometest1:this.kati,
  woof:function() { return "Woof, woof!";},
  sayName:function myfun() {return this.name;},
  saysometest() {
    // Method which will display sometest
    return this.sometest;
  }
};

console.log(dog.woof());
console.log(dog.sayName());
console.log(dog.saysometest());
console.log(dog.sometest1);
```

Woof, woof!

Bob

test

undefined

- In an arrow function => **this** keyword is automatically bound to the parents' context
- What will we see below?

```
const person = {
  name: 'jane',
  surname: 'ostin',
  show: function () {
    console.log("show is running: "+this.name)
  },
  getone: function getMore(){
    console.log(this.name);
    function inner() {
      console.log('Inner is running: '+this);
    }
    inner();
  },
  gettwo: function getarrowMore(){
    console.log(this.name);
    const inner = () => {
      console.log('Inner arrow is running: '+this);
    }
    inner();
  }
};

console.log("this in global:"+this);
person.show();
person.getone();
person.gettwo();
```


- In an arrow function => **this** keyword is automatically bound to the parents' context
- What will we see below?

Remember: if a function is not called as a method **this** refers to the global object

```
33 const person = {
34   name: 'jane',
35   surname: 'ostin',
36   show: function () {
37     console.log("show is running: "+this.name)
38   },
39   getone: function getMore(){
40     console.log(this.name);
41     function inner() {
42       console.log('Inner is running: '+this);
43     }
44     inner();
45   },
46   gettwo: function getarrowMore(){
47     console.log(this.name);
48     const inner = () => {
49       console.log('Inner arrow is running: '+this);
50     }
51     inner();
52   }
53 };
54
55 console.log("this in global:"+this); //this in global:[object Window]
56
57 person.show(); //show is running: jane
58
59 person.getone();//jane Inner is running: [object Window]
60 person.gettwo();//Inner arrow is running: [object Object]
61
62
63 //an arrow function binds the parent context to itself
64
```

Object Accessors

- ECMAScript 5 (ES5 2009) introduced Getter and Setters.
- get/set keyword

```
// Create an object:
var person = {
  name: "aristea",
  //return an uppercase version of the name
  // while reserving the actual case for internal use.
  get prop() {
    return this.name.toUpperCase();
  }
};

// Display data from the object using a getter
//name to UpperCase as a property not function
//simpler syntax
document.getElementById("name").innerHTML
  = person.prop;
</script>
```

Object Accessors

```
// Create an object:
const person = {
  name: "Aristea",
  age: "",
  set myage(newage) {
    this.age = newage;
  }
};

// Set a property using set:
person.myage = 25;

// Display data from the object:
document.getElementById("demo").innerHTML = person.age;
</script>
```

Object Accessors

- Based on w3schools Using Getters and Setters:
- gives simpler syntax
- allows equal syntax for properties and methods
- It can secure better data quality
- It is useful for doing things behind-the-scenes

Read more:

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get>

JavaScript Function Call

- Οι μέθοδοι `call()` και οι `apply()` μέθοδοι είναι προκαθορισμένες μέθοδοι JavaScript.
- Μπορούν να χρησιμοποιηθούν ώστε -> να καλέσουν μια μέθοδο ενός αντικειμένου με ένα άλλο αντικείμενο ως όρισμα.

```
var cat = {
  det: function() {
    return this.name + " " + this.age;
  }
}

var cat1 = {
  name: "Javie",
  age: "4"
}

var cat2 = {
  name: "Rosa",
  age: "9"
}

console.log(cat2);
//This example calls the det method of cat, using it on cat1:

var x = cat.det.call(cat1); //Javie 4

document.getElementById("demo").innerHTML = x;
</script>
```

call() method => calls the function with a given **this** value and **arguments** provided individually.

- call(thisArg)
 - call(thisArg, arg1)
 - call(thisArg, arg1, /* ..., */ argN)
-
- thisArg: value to use as **this** when calling **function/method**

apply() Method

- apply() method is similar to the call() method
- call() VS apply() method

- call() method -> arguments separately

```
apply(thisArg)  
apply(thisArg, argsArray)
```

- apply() method ->arguments as an array

```
<h2>JavaScript Functions</h2>
<p>This example calls the det method of cat, using it on cat1:
</p>

<p id="demo"></p>
<p id="demo1"></p>
<script>
var cat = {
  det: function(eyecolor) {
    return this.name + " " + this.age+" "+eyecolor;
  }
}
console.log(cat);

var cat1 = {
  name:"Javie",
  age: "4"
}
var cat2 = {
  name:"Rosa",
  age: "9"
}
console.log(typeof(cat1));
console.log(cat2);

//This example calls the det method of cat, using it on cat1:
var x = cat.det.call(cat1,"blue");
document.getElementById("demo").innerHTML = x;

var z = cat.det.apply(cat2,["gold"]);
document.getElementById("demo1").innerHTML = z;
</script>

</body>
</html>
```


Constructor functions

- Another way to create an "object type", is to use an **object constructor function**
- constructor : when we want a "blueprint" for creating many objects of the same "type"
- In a constructor function **this** does not have a value!
- The value of **this** will become the new object when a new object is created.

```
<script>
//constructor : when we want a "blueprint" for creating many objects of the same "type"
//The way to create an "object type", is to use an object constructor function.
//Note: It is considered a good practice to capitalize the first letter of your constructor
function MyCat(x, y,z) {

    this.name = x;
    this.age  = y;
    this.eyesColor = z;
    this.des = function(){ return this.name + " " + this.age};
}

// This creates a new object
//In JavaScript, when this keyword is used in a constructor function, this refers to the object
var x = new MyCat("Javie", "4","blue");
console.log(typeof(x));

//In other words, this.name means the name property of this object.
//Hence, when an object accesses the properties, it can directly access the property of the object
document.getElementById("cat1").innerHTML = x.name;

var y = new MyCat("Rosa", "9","gold");
document.getElementById("cat2").innerHTML = y.des();

//The this keyword in the constructor does not have a value.The value of this will be the object
</script>

</body>
```

- Add a property to an existing object

```
function MyCat(x, y,z) {  
  
    this.name = x;  
    this.age  = y;  
    this.eyesColor = z;  
    this.des = function(){ return this.name + " " + this.age};  
}  
  
// This creates a new object  
//In JavaScript, when this keyword is used in a constructor function,  
//this refers to the object when the object is created. For example,  
  
var x = new MyCat("Javie", "4","blue");  
console.log(typeof(x));  
  
//In other words, this.name means the name property of this object.  
//Hence, when an object accesses the properties, it can directly access the property as  
document.getElementById("cat1").innerHTML = x.name;  
  
var y = new MyCat("Rosa", "9","gold");  
document.getElementById("cat2").innerHTML = y.des();  
  
//The this keyword in the constructor does not have a value.  
//The value of this will be the new object created when the function is invoked.  
  
//add a property to a existing object  
x.food="everything";  
  
x.prefers = function(){  
    return "My cat likes: "+ this.food;  
};  
  
console.log(x.food+ " - " + x.prefers());  
console.log(MyCat);
```

```
everything - My cat likes: everything
```

```
19functionConst
```

```
f MyCat(x, y,z) {
```

```
19functionConst
```

```
    this.name = x;  
    this.age  = y;  
    this.eyesColor = z;  
    this.des = function(){ return this.name + " " + this.age};
```

```
}
```

```
>
```

We cannot add a new method/property to an **object constructor** the same way you add a new method/property to an existing object.

Adding methods to an object constructor must be done **inside** the constructor function or **with *prototypes***

Prototype Inheritance

Every JavaScript object inherits the properties and methods of a prototype:

- **Date** objects -> inherit from Date.prototype
- **Array** objects -> inherit from Array.prototype
- **MyCat** objects inherit from **MyCat**.prototype

- The Object.prototype is on the top of the prototype inheritance chain:
 - **Date** objects, **Array** objects, and **MyCat** objects inherit from Object.prototype.

- Source: https://www.w3schools.com/js/js_object_prototypes.asp

Prototype chain

- In Javascript each object has a **private** property that contains a **link** to its prototype object
- This prototype object has **its own prototype**, and so on until an object with **null** as its prototype is reached (last prototype in this chain)
- Objects in JavaScript are dynamic "bags" of properties
- When attempting to access a property of an object, the property is also **sought on the object's prototype**, the prototype of the prototype, and so on, until a property with a matching name is found or the prototype chain is exhausted.
- Άρα τα prototypes διαμορφώνουν μια αλυσίδα...

- in JavaScript, new objects -> have generic methods like **toString()** & **valueOf()**
- **[[Prototype]]**: is a **somehow-hidden property** on every object which is **accessed** if some property which is being read on the object is not available.
 -

JavaScript *prototype* property

- You may wish to **add new properties (or methods)** to all existing objects of a particular type.
- Thus, it may be necessary to **add new properties (or methods)** to an object's **constructor**.
- Utilizing the JavaScript *prototype* property allows us to add new properties/methods to object constructors


```

//Note: It is considered a good practice to capitalize the first letter of your construct
function MyCat(x, y,z) {
    this.name = x;
    this.age = y;
    this.eyesColor = z;
    this.des = function(){ return this.name + " " + this.age};
}

// This creates a new object
//In JavaScript, when this keyword is used in a constructor function,
//this refers to the object when the object is created. For example,

var x = new MyCat("Javie", "4","blue");
console.log(typeof(x));

//The this keyword in the constructor does not have a value.
//The value of this will be the new object created when the function is invoked.

//add a property to a existing object
MyCat.prototype.food="everything";

MyCat.prototype.prefers = function(){
    return "My cat likes: "+this.food;
};

console.log(x.food+ " - " + x.prefers());
console.log(y.food+ " - " + y.prefers());

// changing the property value of prototype
MyCat.prototype.food= 'chicken';

// creating new object
const stilvi = new MyCat('Stilvi', '1','gold');
console.log(stilvi);

console.log(stilvi.prefers()); // 50

```

- When the program is executed, **x.food** examines the constructor function to determine if the **food** property exists
- Since the **constructor function** lacks a **food** property, the program examines the
- **constructor function's prototype object**, and x inherits the property from the prototype object (if available).
- **Check** <https://www.geeksforgeeks.org/difference-between-prototype-and-prototype/>