This, objects & more…

this Keyword

this Keyword

- 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:

- 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 objec
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 objec
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** -> αξιολογείται κατά το run-time, ανάλογα με το context
- Η τιμή του **this σε ένα function** καθορίζεται από το **πως** καλούμε το function.
- Εάν το **this** χρησιμοποιείται εκτός οποιασδήποτε συνάρτησης ή εάν μια συνάρτηση **δεν καλείται ως μέθοδο**ς αυτό αναφέρεται στο καθολικό αντικείμενο.
- Ας δούμε ένα παράδειγμα…

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

Εάν το **this** χρησιμοποιείται εκτός οποιασδήποτε συνάρτησης ή εάν μια συνάρτηση **δεν καλείται ως μέθοδο**ς αυτό αναφέρεται στο καθολικό αντικείμενο.

```
    <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
```

- Arrow functions **do not bind their own this**, instead, they **inherit** the one **from the parent scope**
    - This makes arrow functions to be a great choice in some scenarios but a very bad one in others
- In an arrow function => **this** keyword is automatically bound to the parents' context

```
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?

```
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); //this in global:[object Window]

person.show(); //show is running: jane

person.getone();//jane Inner is running: [object Window]
person.gettwo();//Inner arrow is running: [object Object]


//an arrow function binds the parrent context to itself

```

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

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
- more readable and maintainable code

Read more:

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

Object Accessors

```
const person = {
  _name: "John",

  get name() {
    console.log("Getting name");
    return this._name;
  },

  set name(newName) {
    console.log("Setting name");
    this._name = newName;
  }
};

console.log(person.name); // Calls the getter method
person.name = "Jane";      // Calls the setter method
console.log(person.name); // Calls the getter method
```

```
const anotherperson = {
  _name: "John",

  getName() {
    console.log("Getting name");
    return this._name;
  },

  setName(newName) {
    console.log("Setting name");
    this._name = newName;
  }
};

console.log(anotherperson.getName()); // Calls the getName method
anotherperson.setName("Jane");          // Calls the setName method
console.log(anotherperson.getName()); // Calls the getName method
```

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 "typ
//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 constr
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 t

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 a
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
</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
ƒ 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**

```
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

//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-proto-and-prototype/**

Bind method

- Η μέθοδος bind(): **δημιουργεί** μια **νέα συνάρτηση** που, όταν καλείται, "δένει" τη **λέξη-κλειδί this** με μια τιμή που ορίζουμε εμείς

Σύνταξη:

- bind(thisArg)
- bind(thisArg, arg1)
- bind(thisArg, arg1, arg2)
- bind(thisArg, arg1, arg2, /* …, */ argN)

```
const member = {
  firstName:"Aristea",
  lastName: "Kontogiannh",
}


function sayHi(){
  console.log(this);
  return "Hi I am "+this.firstName;
}

console.log(sayHi());

let hi=sayHi.bind(member);

console.log(hi());
//or
console.log(sayHi.bind(member)());
```

Bind method

- Ας δούμε πως θα μπορούσαμε να χρησιμοποιήσουμε τη μέθοδο bind() με μια μέθοδο ενός αντικειμένου.

Call vs Apply vs Bind

call: **binds** the **this** value, **calls** the function, and accepts a list of arguments

apply: **binds** the **this** value, **calls** the function, and accepts arguments as an **array**

bind: **binds** the **this** value, **returns** a new function (we still need to separately invoke the returned function), and accepts a list of arguments.

Object.assign()

- Object.assign() method-> is used to **copy** the values and properties from one or more source objects to a target object
- Object.assign() is used for **cloning** an object.
- Object.assign() is used to **merge** object with **same properties.**

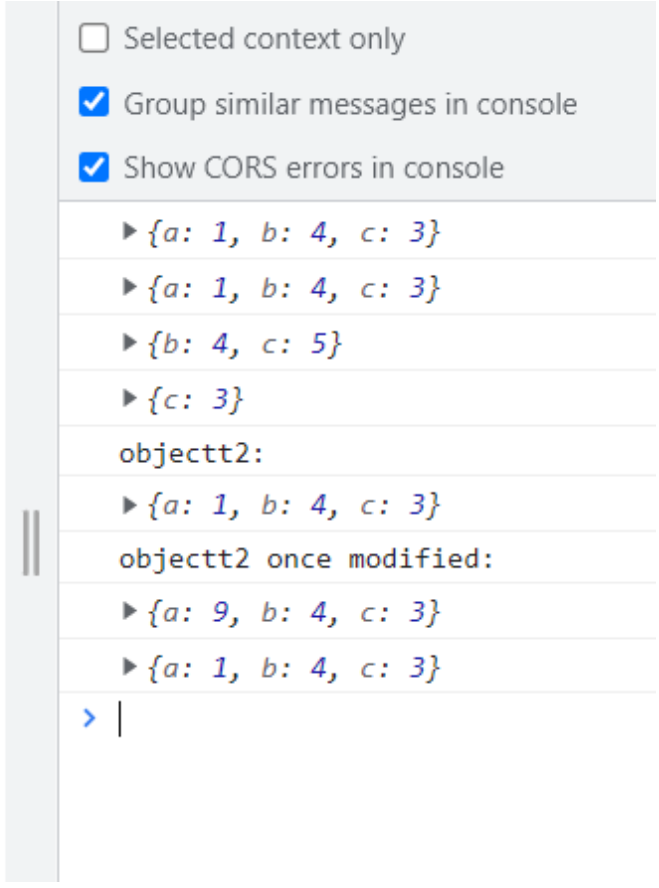
Object.assign()

```
const o1 = { a: 1, b: 1, c: 1 };
const o2 = { b: 4, c: 5 };
const o3 = { c: 3 };

const obj = Object.assign(o1, o2, o3);
console.log(obj);
console.log(o1);
console.log(o2);
console.log(o3);

const obj2 = Object.assign({}, o1);
console.log("objectt2: ");
console.log(obj2);
obj2.a=9;
console.log("objectt2 once modified: ");
console.log(obj2);

console.log(o1);
```



Exception handling

```
try {
   iDontExist();
}
catch(e){
   //process error here    >> ReferenceError: iDontExist is not defined
   out(e);
}
finally {
   //do some work here
}
```

Exception handling

- The **try** statement lets you test a block of code for errors.
- The **catch** statement lets you handle the error.
- The **throw** statement lets you create custom errors.
- The **finally** statement lets you execute code, after try and catch, regardless of the result.

Errors happen for a plethora of reasons!

- To handle them we may use:
- **try statement ->** define a block of code to be tested for errors while it is being executed.
- **catch statement ->** define a block of code to be executed, if an error occurs in the try block.

- Note: use try/catch block when the normal path through the code should proceed without error unless there are truly some exceptional conditions

Example

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Error Handling</h2>

<p>This example demonstrates how to use <b>catch</b> to diplay an error.</p>

<p id="demo"></p>

<script>
try {
  adddlert("Welcome guest!");
}
catch(err) {
  document.getElementById("demo").innerHTML = err.message;
}
</script>

</body>
</html>
```

Classes

- Οι κλάσεις αποτελούν ένα πρότυπο (template) για τη δημιουργία αντικειμένων (objects).

Class declarations

```
    <script>
        class Person {
            constructor(name) {
                this.name = name;
            }

            introduce() {
                console.log(`Hello, my name is ${this.name}`);
            }
        }

        const otto = new Person("Otto");


        otto.introduce(); // Hello, my name is Otto
    </script>
</body>
</html>
```

constructor =>enables us to provide any custom initialization that must be done before any other methods can be called on an instantiated object.

If we don't provide your own constructor=> then a default constructor will be supplied

We could say that classes are an easier way to write constructors…

Constructor Functions:

Syntax:

- Constructor functions are defined using a **regular function declaration.**
- They use the **this** keyword to **define properties and methods**.

```
20  function MyCat(x, y,z) {
21
22    this.name = x;
23    this.age  = y;
24    this.eyesColor  = z;
25    this.des = function(){ return this.name + " " + this.age};
```

Instantiation:

- Objects are created using the new keyword.

```
var y = new MyCat("Rosa", "11","gold");
```

Classes

Syntax:

- Classes were introduced in ECMAScript 2015 (ES6) and provide **a more structured syntax for creating constructor functions.**

Instantiation:

- Objects are created using the new keyword, similar to constructor functions.

Actually:

- Under the hood, a class in JavaScript is still a constructor function and prototype-based inheritance.

```
class Person {
    constructor(name) {
        this.name = name;
    }

    introduce(age) {
        this.age=age;
        console.log("Hello, my name is "+this.name+" "+this.age);
    }
}

const otto = new Person("Otto");
```

More about classes

- https://www.w3schools.com/js/js_class_intro.asp
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this#class_context
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes