

THIS
is the end

Sorohan Valentin - Speciale Gabriele

What is THIS?

In JavaScript, **'this'** is a special expression that refers to an object, and the value of **'this'** depends on the execution context in which a function is invoked.



- **'this'** allows you **to access the properties and methods** of the "owner" object of the function in which it is used.
- The value of **'this'** is **determined automatically** at runtime, and is different depending on how and where the function is called.

Object Context

When **'this'** is used inside an **object's method**, its value refers to the object that owns the method itself, the object on which the method was called.



```
1  const car = {
2    brand: "Toyota",
3    getBrand: function () {
4      console.log(this.brand); // Outputs: Toyota
5    },
6  };
7
8  car.getBrand();
```

Global Context

In JavaScript, when **'this'** is used in the *global context*, **its value depends on the execution environment**. The behavior varies between:

- **Browser:** in a browser, **'this'** refers to the global object called *window*.
- **Node.js:** in Node.js, **'this'** in the global context refers to the *global object*.



```
1 // outputs the Window object
2 console.log(this);
```



```
1 // outputs the global object
2 console.log(this);
```

Constructor Function Context

When a **constructor function** is invoked with the **new** keyword, the value of **'this'** refers to the **new object that is created**.

- **Constructor function:** a constructor function is a normal function that, when called with *new*, initializes a new object.
- The value of **'this'** is the new object created by *new*

```
1 function Animal(type) {  
2   this.type = type;  
3 }  
4  
5 const dog = new Animal("Dog");  
6 console.log(dog.type); // Outputs: Dog
```

Class Context

When you use **'this'** inside a **class** in JavaScript, it refers to the instance of the class, which is the object created by the class itself.

The behavior of this in classes is similar to that in constructor functions, but with a more modern syntax.

```
1  class Person {  
2    constructor(name) {  
3      this.name = name;  
4    }  
5  
6    greet() {  
7      console.log(`Hello, ${this.name}`);  
8    }  
9  }  
10  
11  const john = new Person("John");  
12  john.greet(); // Outputs: Hello, John
```

Event Handler Context

In inline event handlers, such as those defined directly in the HTML code, the value of **this** refers to **the DOM element that generated the event**.

When using an event handler directly in HTML markup (E.g: with the onclick attribute), **this** refers to **the element that received the event**.



```
1 <button id="btn">Click me</button>
2 <script>
3   document.getElementById("btn").addEventListener("click", function () {
4     console.log(this); // Outputs: <button id="btn">Click me</button>
5   });
6 </script>
```

Function Context

When a **regular function** is called, the value of **'this'** depends on how the function is called:

- **Strict Mode:** when the function is in strict mode (**"use strict"**), this will be undefined if it is not explicitly bound to an object.
- **Non-strict mode:** the value of **'this'** in the context of a global function or a simple function (invoked without an object) refers to the global object (window in the browser).

```
1  "use strict";
2
3  function show() {
4      console.log(this); // Outputs: undefined
5  }
6  show();
```

```
1  function show() {
2      console.log(this); // Outputs: `window` in browsers
3  }
4  show();
```


Arrow Functions Context

'this' in an **arrow function**, it refers to the value of this in the external context (scope) in which the *function was defined*.

- **Arrow functions do not bind this:** arrow functions DO NOT create their own **'this'**. Instead, they inherit the value of **'this'** from the environment in which they were created.
- **Typical usage:** are useful when you want to *preserve the value* of **'this'** from its surroundings, such as inside a method or callback.

```
1  const obj = {
2    value: 42,
3    regularFunction: function () {
4      console.log(this.value, this); // Outputs: 42 and object
5    },
6    arrowFunction: () => {
7      console.log(this.value, this);
8      /* Outputs: undefined and window,
9         because `this` is inherited
10        from the outer context */
11    },
12  };
13
14  obj.regularFunction();
15  obj.arrowFunction();
```

Losing Context in Callback Functions

The behavior of **'this'** in callback functions can be difficult to manage due to the *loss of context*. When a function is passed as a callback (as in the case of **setTimeout**), the value of **'this'** can change.



```
1  const user = {
2    name: "Gabriele",
3    greet() {
4      console.log(this.name);
5    },
6  };
7
8  setTimeout(user.greet, 1000); // Outputs: Undefined
```

Losing Context in Callback Functions **solutions**

- Using an **arrow function** to call `user.greet()` solves the context loss problem.
- The arrow functions do not have their own 'this' and therefore inherit the `this` of the external context (which in this case is `user`).
- The **`.bind()`** method creates a new function that "binds" the value of **'this'** to the `user` object.
- So even if the function is passed as a callback, **'this'** remains correct and refers to user.

```
1  const user = {
2    name: "Gabriele",
3    greet() {
4      console.log(this.name);
5    },
6  };
7
8  setTimeout(() => user.greet(), 1000); // Outputs: Gabriele
```

```
1  const user = {
2    name: "Gabriele",
3    greet() {
4      console.log(this.name);
5    },
6  };
7
8  setTimeout(user.greet.bind(user), 1000); // Outputs: Gabriele
```

Bind, call, apply

Function.prototype.bind()

The **bind()** method of **Function** instances creates a *new* function that, when called, calls this function with its **'this'** keyword set to the provided value, and a given sequence of arguments preceding any provided when the new function is called.

```
1  function greet() {  
2      console.log(`Hello, ${this.name}`);  
3  }  
4  
5  const user = { name: "Alice", age: 20 };  
6  
7  const boundGreet = greet.bind(user);  
8  
9  boundGreet(); // Outputs: Hello, Alice
```

[MDN|.bind\(\)](#)

Function.prototype.call()

The **call()** method of **Function** instances calls this function with a given this value and arguments provided individually.


```
1 function greet() {
2   console.log(`Hello, ${this.name}`);
3 }
4
5 const user = { name: "Alice", age: 20 };
6
7 greet.call(user); // Outputs: Hello, Alice
```

```
1 function greet(country) {
2   console.log(`Hello ${this.name}, I'm from ${country}`);
3 }
4
5 const user = { name: "Alice", age: 20 };
6
7 greet.call(user, "Spain"); // Outputs: Hello Alice, I'm from Spain
```

[MDN|.call\(\)](#)

Function.prototype.apply()

The **apply()** method of **Function** instances calls this function with a given this value, and arguments provided as an array (or an array-like object).



```
1 function greet(country, food) {  
2   console.log(`Hello ${this.name}, I'm from ${country}, I like ${food}`);  
3 }  
4  
5 const user = { name: "Alice", age: 20 };  
6  
7 greet.apply(user, ["Spain", "pizza"]);  
8 // Outputs: Hello Alice, I'm from Spain, I like pizza  
9
```

[MDN](#) | [.apply\(\)](#)