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.

MDN|this

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.

```javascript
const car = {
  brand: "Toyota",
  getBrand: function () {
    console.log(this.brand); // Outputs: Toyota
  },
};

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*.

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

- **Node.js**: in Node.js, **'this'** in the global context refers to the *global object*.

```
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_**

```javascript
function Animal(type) {
  this.type = type;
}

const dog = new Animal("Dog");
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.

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

  greet() {
    console.log(`Hello, ${this.name}`);
  }
}

const john = new Person("John");
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 <u>onclick attribute</u>), **'this'** refers to **the element that received the event**.

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

```javascript
"use strict";

function show() {
  console.log(this); // Outputs: undefined
}
show();
```

- **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*).

```javascript
function show() {
  console.log(this); // Outputs: `window` in browsers
}
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.

```javascript
const user = {
  name: "Gabriele",
  greet() {
    console.log(this.name);
  },
};

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).

```javascript
const user = {
  name: "Gabriele",
  greet() {
    console.log(this.name);
  },
};

setTimeout(() => user.greet(), 1000); // Outputs: Gabriele
```

- 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**.

```javascript
const user = {
  name: "Gabriele",
  greet() {
    console.log(this.name);
  },
};

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.

```javascript
function greet() {
  console.log(`Hello, ${this.name}`);
}

const user = { name: "Alice", age: 20 };

const boundGreet = greet.bind(user);

boundGreet(); // Outputs: Hello, Alice
```

Function.prototype.call()

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

```javascript
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
```

```javascript
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
```

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).

```js
function greet(country, food) {
  console.log(`Hello ${this.name}, I'm from ${country}, I like ${food}`);
}


const user = { name: "Alice", age: 20 };


greet.apply(user, ["Spain", "pizza"]);
// Outputs: Hello Alice, I'm from Spain, I like pizza

```

MDN | .apply()