Date: / /

The Javascript this keyword

In Js, this keyword allows us to

* Reuse functions in different execution contexts. It means, a function once defined can be invoked for different objects using this keyword.

* Identifying the object in the current execution context when we invoke a method.

The this keyword is very closely associated with Javascript functions

When it comes to this, the fundamental thing to understand where a function is invoked.

Becoz we don't know this keyword until the function is invoked.

Date: / /

Types of Binding In Javascript

- Default Binding
- Implicit Binding
- Explicit Binding
- Constructor Call Binding

Default Binding In JavaScript

One of the first rule to remember is that if the function housing a this reference is standalone function, then that function is bound to the global object.

For eg →

```
function alert() {
console.log (this.name + 'is calling');
}
  cons name = 'deepa';
  alert(); // deepa is calling
```

Standalone function

As you can see, name() is a standalone, unattached function, so it is bound to the global scope.

As a result this.name reference resolves to the global variable

```
cons name = 'deepa'
```

This rule doesn't hold if name() were to be defined in strict mode.

It will output // TypeError: 'this' is undefined

Implicit Binding In JavaScript

Acc. to binding rule in JavaScript, a function can use an object as its context only if that object is bound to it at call site.

```
Foreg → function alert() {
            console.log(this.age + 'years');
        }
        cons myObj = {
            age: 22,
            alert: alert
        }
        myObj.alert() //22 years
```

When you call a function using dot notation, this is implicitly bound to the object the function is being called from.

In this eg Since alert is being called from myObj, the this keyword is bound to myObj.

So when alert is called with myObj.alert() this.age is 22, which is age property of myObj

Another Eg → function alert() {
    console.log(this.age + 'years');
}

```
const myObj = {
    age: 22,
    alert: alert,
    nestedObj: {
        age: 26,
        alert: alert
    }
};

myObj.alert();          // 22 years
myObj.nestedObj.alert(); // 26 years
```

Explicit binding In JavaScript

If we want to force a function to use an object as its context without putting a property function reference to object.

We have 2 utility methods
- call()
- apply()

Along with other set of utility functions, These 2 utilities are available to all functions in Javascript via [[Prototype]] mechanism.

To explicitly bind a function call to context, you simply invoke a call() on that function and pass in context object as parameter.

For eg →
```
Function alert(){
    console.log(this.age + 'years');
}
const myObj = {
    age: 22;
}
```

```
alert.call(myObj); // 22 years
```

Even if you were to pass around the function multiple times to new variables (currying), every invocation will use same context because it has been locked (explicitly bound) to that object.

This is called hard binding

Eg-
```
Function alert() {
    console.log(this.age);
}
const myObj = {
    age: 22;
};
const bar = function () {
    alert.call(myObj);
};
bar(); // 22
SetTimeout(bar, 100); // 22
// a hard bound 'bar' can no longer have
    it's 'this' context overridden
bar.call(window); // 22
```

Hard binding is perfect way to lock a context into a function call and truly make that function into a method.

Constructor Call binding In JavaScript

When a function is invoked with new keyword in front of it, known as Constructor call, following things occur

- ✯ A brand new object is created
- ✯ The newly constructed object is [[Prototype]] - linked to the function that Constructed it.
- ✯ The newly constructed object is set as the this binding for that function call.

Eg →
```
Function giveAge (age) {
    this.age = age;
}
const bar = new giveAge (22);
console.log(bar.age); //22
```

By calling giveAge(...) with new in front, we've constructed a new object and set the new objects as the this for call.

So new is final way that you can bind a function call's this.

JavaScript "this" keyword - Explained

"Run" is a polysemic word. It's meaning depends on the context in which it is used.

Where it's Used	Meaning
"I will run home" →	move quickly on foot
"He will run for president" →	vying for election
"The app is still running' →	Software application is open and active

The this Keyword is very similar to "run", it points to a different object/scope depending on where it is used.

Where it's Used

1. In a method

```
var me = {
  firstName : "Wingsley",
  lastName  : "Ubah",
  fullName  : function() {
    return    this.firstName + " " + this.lastName;
  }
};
```

→ Refers to the owner object (me)

2. Alone and independent

```
var x = this.name;
```

→ Refers to the global object (Window)

3. In a function

```
function myFunction() {
  return this.name;
}
```

→ Also refers to the global object

JavaScript "this" keyword - Explained

Where it's Used

4. In a function (strict mode)

```
"use strict";
function myFunction() {
  return this.name;
}
```

➔ Returns "undefined"

4. call() and apply() - Explicit binding

```
var person1 = {
  fullName () {
    return this.firstName + " " + this.lastName;
  }
}
var person2 = {
  firstName:"Kingsley",
  lastName: "Ubah",
}
person1.fullName.call(person2);  // Kingsley Ubah
```

➔ Forces the function - fullName() - to refer to the object - person2.

5. In Event Handlers

```
<button onclick="this.style.display='none'">
  Click to Remove Me!
</button>
```

➔ Refers to the HTML Element that received that event (button)