# Introduction to 'this'

The this keyword in JavaScript is often confusing for developers.

Unlike other programming languages, this in JavaScript doesn't always refer to the current object or class instance.

The value of this depends on how a function is called, not where it's defined.

```
console.log(this); // In global context: window object

function showThis() {
  console.log(this); // Depends on how showThis is called
}
```

Understanding this is essential for writing effective JavaScript code.

# Execution Context

Every line of JavaScript code runs in an execution context.

- The JavaScript runtime maintains a stack of contexts, with the top one being active.
- There are three types of executable code: Global code, Function code, and Eval code.
- The value of this is determined when control enters a new execution context.

```
// Global execution context
console.log(this); // window object (in browser)

function myFunction() {
  // Function execution context
  console.log(this); // Depends on how it's called
}
```

# Global Context

In the global scope, this refers to the global object.

- In browsers, the global object is window.
- In Node.js, it's the global object.

```
// In a browser
console.log(this === window); // true

var name = "Global";
console.log(this.name); // "Global"
console.log(window.name); // "Global"
```

> In strict mode, this is undefined in global functions.

# Method Invocation

When a function is called as a property of an object, this refers to the object that owns the method.

> **Rule:**
>
> When a function is called as object.method(), this inside the method refers to the object.

```
const person = {
  name: "John",
  greet: function() {
    console.log("Hello, my name is " + this.name);
  }
};

person.greet(); // "Hello, my name is John"
```

This works with both dot notation and bracket notation:

```
person["greet"](); // "Hello, my name is John"
```

# Function Invocation

When a function is called directly (not as a method), this behaves differently depending on the mode.

| Normal Mode | Strict Mode |
| --- | --- |
| this refers to the global object (window in browsers) | this is undefined |

```
function showThis() {
  console.log(this);
}

showThis(); // window object (in browser)

// With strict mode
"use strict";
function strictShowThis() {
  console.log(this);
}

strictShowThis(); // undefined
```

This is a common source of bugs when functions are passed as callbacks.

# Constructor Invocation

When a function is called with the new keyword, it becomes a constructor.

1. A new empty object is created
2. The function is called with this set to the new object
3. The object is returned (unless the function returns another object)

```
function Person(name) {
  this.name = name;
  this.greet = function() {
    console.log("Hello, I'm " + this.name);
  };
}

const john = new Person("John");
john.greet(); // "Hello, I'm John"

// Without 'new' keyword:
const wrongPerson = Person("Alice");
console.log(wrongPerson); // undefined
console.log(window.name); // "Alice" (in browser)
```

# Arrow Functions

Arrow functions handle this differently from regular functions.

| Regular Function | Arrow Function |
| --- | --- |
| Creates its own this binding based on how it's called | Inherits this from the surrounding scope (lexical this) |

```
const obj = {
  name: "Object",
  regularMethod: function() {
    console.log(this.name); // "Object"

    // Inside regular function
    setTimeout(function() {
      console.log(this.name); // undefined (window)
    }, 100);

    // Inside arrow function
    setTimeout(() => {
      console.log(this.name); // "Object"
    }, 100);
  }
};
```

> Arrow functions are permanently bound to their lexical context and cannot be changed with call(), apply(), or bind().

# Explicit Binding: bind, call, apply

JavaScript provides three methods to explicitly set the value of this:

| Method | Description |
| --- | --- |
| call(thisArg, arg1, arg2, ...) | Calls function with specified this and individual arguments |
| apply(thisArg, [argsArray]) | Calls function with specified this and arguments as array |
| bind(thisArg, arg1, arg2, ...) | Returns a new function with this permanently bound |

```
function greet(greeting) {
  console.log(greeting + ", " + this.name);
}

const person = { name: "John" };

// Using call
greet.call(person, "Hello"); // "Hello, John"

// Using apply
greet.apply(person, ["Hi"]); // "Hi, John"

// Using bind
const greetJohn = greet.bind(person);
greetJohn("Hey"); // "Hey, John"
```

# Common Mistakes

### ⚠️ Losing this in callbacks

```
const user = {
  name: "John",
  greet: function() {
    setTimeout(function() {
      console.log("Hello, " + this.name);
    }, 1000);
  }
};
user.greet(); // "Hello, undefined"
```

**Solution:**

Use arrow functions or bind()

### ⚠️ Method reference loses context

```
const user = {
  name: "John",
  greet: function() {
    console.log("Hello, " + this.name);
  }
};
const greet = user.greet;
greet(); // "Hello, undefined"
```

**Solution:**

Use bind() when storing method references

# Summary

The value of this depends on how a function is called:

> **Global Context**
>
> this refers to the global object (window in browsers)

> **Method Invocation**
>
> this refers to the object that owns the method

> **Function Invocation**
>
> this refers to global object (or undefined in strict mode)

> **Constructor Invocation**
>
> this refers to the newly created object

> **Arrow Functions**
>
> this is inherited from the surrounding scope

---


1. Understanding the "this" keyword in JavaScript - unschooled.org
2. Gentle Explanation of "this" in JavaScript - dmitripavlutin.com
3. Understand JavaScript's "this" With Clarity - javascriptissexy.com