# JavaScript

Function Invocations

@nxpatterns - https://x.com/nxpatterns

2022-09-01

## Contents



**Disclaimer and Professional Notice**

The concepts, architectures, and implementations described in this documentation are intended for use under the guidance of experienced software architects and professionals. While the presented approaches and patterns have been thoroughly tested in production environments, their successful implementation requires:

- Oversight by experienced software architects
- Proper evaluation of specific use cases and requirements
- Thorough understanding of architectural implications
- Appropriate risk assessment and mitigation strategies
- Consideration of organizational context and constraints

**THIS DOCUMENTATION AND THE ACCOMPANYING CODE ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO:**

- The documentation and code examples are provided for educational and informational purposes only
- No warranty or guarantee is made regarding the accuracy, reliability, or completeness of the content
- The author(s) assume no responsibility for errors or omissions in the content
- The author(s) are not liable for any damages arising from the use of this documentation or implementations derived from it. Implementation decisions and their consequences remain the sole responsibility of the implementing organization
- Success in one context does not guarantee success in another; proper evaluation is required

**To minimize risks and ensure successful implementation:**

- Conduct thorough architecture reviews before implementation
- Engage experienced software architects throughout the project lifecycle
- Implement comprehensive testing strategies
- Maintain proper documentation of architectural decisions
- Consider scalability, maintenance, and long-term implications
- Establish proper governance and review processes
- Ensure adequate team training and knowledge transfer

**Implementation of the described architectures and patterns requires substantial expertise in:**

- Enterprise software architecture
- Monorepo management and tooling
- Modern development practices and toolchains
- System design and integration
- Performance optimization and scaling
- Security best practices
- DevOps and CI/CD processes
- Azure, AWS, Google Cloud, Hetzner Cloud, or other cloud platforms

# 1 Introduction

When you call (invoke) a function in JavaScript, think of it like pressing pause on your current task to do something else. Here's what happens:

- The function gets its regular inputs (parameters) plus two special ones:
  - `this`: Points to who/what owns the function
  - `arguments`: Contains all inputs passed to the function
- There are four ways to call a function, each affecting how `this` behaves:
  - As a **method** (belonging to an object)
  - As a standalone **function**
  - As a **constructor** (creating new objects)
  - Using `apply` (manually controlling `this`)
- To call a function, add parentheses after it's name: `functionName()`. You can put inputs inside these parentheses.
- JavaScript is very flexible with function inputs:
  - Too many inputs? Extra ones are ignored
  - Too few inputs? Missing ones become `undefined`
  - Any type of input is allowed for any parameter

Think of it like ordering at a restaurant:

- The waiter (function) takes your order (parameters)
- The kitchen (execution) pauses other orders to work on yours
- The restaurant (JavaScript) is flexible - you can order items not on the menu, skip items, or order extra

## 2 The Method Invocation Pattern

A method is simply a function that belongs to an object. When you call a method:

- The keyword `this` inside the method refers to the object that owns it
- You access methods using dot notation (`object.method()`) or bracket notation (`object['method']()`)

```
// Create a counter object
var counter = {
    value: 0,
    // Method to increase the value
    increment: function(amount) {
        // 'this' refers to 'counter'
        this.value += amount || 1;
    }
};

counter.increment();     // Adds 1
counter.increment(2);    // Adds 2
```

Think of it like a TV remote:

- The remote (object) has buttons (methods)
- Each button knows it belongs to its remote (`this`)
- Pressing volume+ (method) knows to increase *this* TV's volume, not another TV's

The neat part is that the function only gets connected to its object when called, making methods highly reusable across different objects.

# 3 The Function Invocation Pattern

Here's a simple explanation of the function invocation pattern using a counter example:

**Problem:**

```
let counter = {
    count: 0,
    // Method works - 'this' refers to counter
    increment: function() {
        this.count += 1;
    },
    // Problem: Inner function loses 'this'
    incrementLater: function() {
        setTimeout(function() {
            this.count += 1;  // Fails: 'this' is not counter!
        }, 1000);
    }
};
```

**Solution:**

```
let counter = {
    count: 0,
    incrementLater: function() {
        // Store 'this' in 'that'
        let that = this;
        setTimeout(function() {
            that.count += 1;  // Works: 'that' is counter
        }, 1000);
    }
};
```

Think of it like a remote control:

— When you press a button directly (regular method) - it knows which TV to control
— When you set a timer (inner function) - it forgets which TV unless you write it down (`that`)

**This is a JavaScript design flaw**. Modern JavaScript fixes this using **arrow functions**:

```
incrementLater: function() {
    setTimeout(() => {
        this.count += 1;  // Works: arrow functions keep 'this'
    }, 1000);
}
```

# 4 The Constructor Invocation Pattern

JavaScript uses object-based inheritance (prototypes) instead of classes, though it tries to look class-like. Here's a simpler example:

```
// Constructor function (like a blueprint)
function Car(model) {
    this.model = model;
}

// Shared method for all cars
Car.prototype.getInfo = function() {
    return this.model;
};

// Create a new car
let myCar = new Car("Tesla");
myCar.getInfo();  // Returns "Tesla"
```

Think of it like a factory:

- The constructor (`Car`) is the blueprint
- `new Car()` creates a fresh car
- All cars share the same methods through `prototype`
- Each car has its own properties (like `model`)

Problems with this approach:

- Forgetting `new` causes bugs (the function runs but `this` points to the wrong place)
- It tries to look like class-based languages but works differently
- The syntax is confusing and error-prone

Recommendation: Use modern `class` syntax instead of constructor functions:

```
class Car {
    constructor(model) {
        this.model = model;
    }

    getInfo() {
        return this.model;
    }
}
```

# 5 The Apply Invocation Pattern

Here's a simpler explanation of JavaScript's `apply` method:

The `apply` method is like a function remote control - it lets you:

— Call any function
— Set what `this` means inside the function
— Pass arguments as an array

Example 1 - Basic function call:

```
function sum(a, b) {
    return a + b;
}

// Two ways to call sum:
sum(3, 4);                    // Normal way: 7
sum.apply(null, [3, 4]);      // Using apply: 7
```

Example 2 - Borrowing methods:

```
// Original object with method
const dog = {
    name: "Rex",
    speak: function() {
        return `${this.name} says woof`;
    }
};

// Borrow the speak method
const cat = { name: "Whiskers" };
dog.speak.apply(cat);  // "Whiskers says woof"
```

Think of `apply` like a universal remote:

— It can control any function (like a universal remote works with any TV)
— You tell it which device to control (`this`)
— You give it all settings at once (arguments array)

Modern JavaScript often uses the spread operator (`...`) instead:

```
sum(...[3, 4]);  // Cleaner than apply
```

# 6 Call, Bind and Arrow Functions

`call()` method is similar to `apply` but takes arguments individually:

```
sum.call(null, 3, 4);  // vs sum.apply(null, [3, 4])
```

`bind()` method creates a new function with fixed `this`:

```
const boundSpeak = dog.speak.bind(cat);
boundSpeak();  // "Whiskers says woof"
```

Arrow functions are them odern solution for the `this` problem:

```
const obj = {
    value: 0,
    increment: () => {
        this.value++;  // 'this' is lexically scoped
    }
};
```

# 7 Short Summary

- Method Invocation: ‘this’ refers to the owner object
  - `object.method()`
- Function Invocation: ‘this’ refers to global object (design flaw)
  - `function()`
- Constructor Invocation: ‘this’ refers to new instance
  - `new Constructor()`
- Apply/Call Invocation: explicitly set ‘this’ and arguments
  - `function.apply(thisValue, [args])`
  - `function.call(thisValue, arg1, arg2)`
- Modern Solutions:
  - Arrow functions for lexical ‘this’
  - `bind()` for fixed ‘this’
  - `class` syntax instead of constructors