# JavaScript Theory: function invocation patterns

Wojciech Trawiński
Sep 1, 2018 · 3 min read

· · ·

## Description

The notion of *this* appearing within a function's or method's body may be quite confusing for newcomers to JavaScript language. I started my programming journey with C++ and Java languages. **When I first came across JavaScript I didn't understand all the complaints about understanding *this*. It was obvious to me,**

**based on C++ and Java experience, that *this* was simply an object upon which a method was called. However, in JavaScript language it's only one out of the four possible function invocation patterns.**

**Goal**

The main aim of today's article is to get familiar with the four function invocation patterns available in JavaScript and understand what is kept under *this* variable in each case.

. . .

## Constructor invocation pattern

One of the possible ways to create an object in JavaScript is to use constructor function.

```
const Car = function(brand, model) {
  this.brand = brand;
  this.model = model;
};

const myFerrari = new Car('Ferrari', 'F40');

console.log(myFerrari);
```

constructor-invocation-pattern.js hosted with ❤ by GitHub view raw

In the above example, *Car* is a constructor function which can be used to create an object with *brand* and *model* properties. **The function's invocation is prepended with the *new* keyword. In this invocation pattern *this* object within the *Car* function's body refers to a newly created object.** After the function's call, the object can be referenced with the aid of *myFerrari* constant.

. . .

## Call/Apply invocation pattern

A JavaScript function has both *call* and *apply* methods available through its prototype. **With the aid of the aforementioned methods you can explicitly indicate what is kept under *this* variable.**

```
const logger = function() {
  console.log('My this ', this);
  console.log(this.message);
};

const error = {
  status: 404,
  message: 'Not found'
};

logger.call(error);
```

call-invocation-pattern.js hosted with ❤️ by GitHub view raw

**Here you simply supply the desired *this* value as the first parameter of the *call* method invoked on the *logger* function.** If the *logger* function accepted arguments, you could provide them as the *call* method's parameters following the first one which indicates *this* value.

. . .

## Method invocation pattern

This pattern is the one I know from C++ and Java programming languages. **It simply means that *this* is an object upon which a method was called.**

```
const error = {
  status: 404,
  message: 'Not found',
  log() {
    console.log('My this ', this);
    console.log(this.message);
  }
};

error.log();
```

method-invocation-pattern.js hosted with ❤️ by GitHub view raw

In the above example, the *error* object has the *log* method which relies on *this* object to derive a *message*. **Since the method is called upon the *error* object, *this* within the *log* method refers to the *error* object.** Simple as that.

## Function invocation pattern

The last pattern simply means that you call a function in the most ordinary way using round parentheses with no *new* keyword or dot operator.

```javascript
const logger = function(message) {
  console.log('My this ', this);
  console.log(message);
};

logger('JavaScript rocks!');
```

function-invocation-pattern.js hosted with ❤ by GitHub view raw

**If a function is invoked using the pattern, *this* within a function's body refers to the global object (*window* if you run this code in a web browser).**

However, if you use the strict mode, *this* value doesn't indicate the global object, but it has *undefined* value.

```javascript
const strictLogger = function(message) {
  'use strict';
  console.log('My this ', this);
  console.log(message);
};

strictLogger('JavaScript rocks!');
```

function-invocation-pattern-strict.js hosted with ❤ by GitHub view raw

### Remarks

If you know the four invocation patterns, you will know longer struggle with the question what is kept under *this* variable. There are only the four possibilities so you just need to recoginze the actual pattern.

The list of exceptions is as follows:

- if you use an arrow function, the above rules doesn't apply, since *this* within an arrow function is determined when a function is defined and is bound to the outer *this* value,
- if you use the *bind* method available through a function's prototype, *this* is bound to the provided value and it can only be changed if you call a function with bound *this* prepended with the *new* keyword.

. . .

Like, love, hate, clap!

JavaScript Programming Function Theory This