Object method, This và Function nâng cao

Object Method

Object thường được tạo ra để đại diện cho các thực thể của thế giới thực, như người dùng, đơn hàng, v.v.

```
let user = {
  name: 'John',
  age: 30
}
```

Và trong thế giới thực thì một user có thể hành động: chọn thứ gì đó từ giỏ hàng, login, logout,...

Các hành động thì được đại diện bằng Javascript function, cụ thể hơn là các function trong các thuộc tính.

Ví dụ Method (phương thức)

Để bắt đầu, chúng ta cùng dạy cho `user` nói hello:

```
let user = {
  name: 'John',
  age: 30
}
user.sayHi = function () {
  alert('Hello!')
}
user.sayHi() // Hello!
```

Ở đây chúng ta sử dụng một Function Expression để tạo một function và gán nó vào thuộc tính `user.sayHi` của object.

Sau đó chúng ta có thể gọi nó bằng cách `user.sayHi()`. Bây giờ user có thể nói!

Một function là thuộc tính của object thì được gọi là method (phương thức)

Vì thế, chúng ta được method `sayHi` của object `user`

Tất nhiên là chúng ta cũng có thể sử dụng một function đã được khai báo từ trước như thế này:

```
let user = {
  // ...
}
// first, declare
function sayHi() {
  alert('Hello!')
}
```

```
// Sau đó thêm một method
user.sayHi = sayHi
user.sayHi() // Hello!
```

Method shorthand

Có một cách viết ngắn gọn cho method trong object literal:

```
user = {
  sayHi: function () {
    alert('Hello')
  }
}
// method bây giờ nhìn xịn hơn phải không nào?
user = {
  sayHi() {
    // tương tự "sayHi: function()"
    alert('Hello')
  }
}
```

Như đã viết bên trên thì chúng ta có thể bỏ qua `"function"` và chỉ cần viết `sayHi()`.

Thành thật mà nói thì 2 cách trên không hoàn toàn là giống nhau đâu. Có những khác biệt nhỏ liên quan đến kế thừa đối tượng, nhưng hiện tại chúng không quan trọng lắm. Trong hầu hết các trường hợp thì cú pháp ngắn gọn được ưu tiên hơn.

This

“this” trong method

Có một điều phổ biến là object method cần truy cập thông tin được lưu trong object để thực hiện một công việc gì đó.

Ví dụ, code bên trong `user.sayHi()` cần name của `user`.

Để truy cập object, một method có thể sử dụng từ khóa this.

Giá trị của `this` là object “trước dấu chấm”, chính là object mà dùng để gọi method.

Ví dụ:

```
let user = {
  name: 'John',
  age: 30,
  sayHi() {
    // "this" là object hiện tại
    alert(this.name)
  }
```

```
}
user.sayHi() // John
```

Ở đây trong suốt quá trình thực thi `user.sayHi()`, giá trị của `this` sẽ là `user`.

Về mặt kỹ thuật, có thể truy cập đến object mà không cần `this`, bằng cách tham chiếu trực tiếp đến biến.

```
let user = {
  name: 'John',
  age: 30,
  sayHi() {
    alert(user.name) // "user" thay vì "this"
  }
}
```

Nhưng code như vậy thì không đáng tin cậy cho lắm. Nếu chúng ta quyết định copy `user` sang một biến khác, ví dụ `admin = user` và sau đó ghi đè `user`, như vậy thì nó sẽ truy cập đến một object sai.

Điều này được minh họa bên dưới:

```
let user = {
  name: 'John',
  age: 30,
  sayHi() {
    alert(user.name) // Điều này sẽ dẫn đến 1 lỗi
  }
}
let admin = user
user = null // ghi đè để làm thứ gì đó
admin.sayHi() // TypeError: Cannot read property 'name' of null
```

Nếu chúng ta tạo `this.name` thay vì `user.name` bên trong `alert` thì code sẽ hoạt động đúng.

“this” không bị ràng buộc

Trong javascript, từ khóa `this` không hoạt động giống như hầu hết các ngôn ngữ lập trình khác. Nó có thể được sử dụng bên trong bất kỳ function nào, ngay cả khi nó không phải là một method của một object

Không có lỗi nào xảy ra trong đoạn code dưới đây:

```
function sayHi() {
  alert(this.name)
}
sayHi()
```

Lưu ý ở đây chúng ta chạy trong **non-strict mode** thì this lúc này là global object (như **Window**), nếu bạn ở **strict mode** thì sẽ gặp lỗi vì `this` là `undefined`.

```
'use strict'
function sayHi() {
  alert(this.name) // Cannot read property 'name' of undefined
}
sayHi()
```

Giá trị của `this` được tính toán suốt quá trình chạy code, dựa vào ngữ cảnh của nó.

Ví dụ, đây là cùng một function được gán cho 2 object khác nhau và nó this khác nhau khi gọi:

```
let user = { name: 'John' }
let admin = { name: 'Admin' }
function sayHi() {
  alert(this.name)
}
// sử dụng cùng 1 function trong 2 object
user.f = sayHi
admin.f = sayHi
// Chúng có this khác nhau
// "this" bên trong function là object trước dấu chấm
user.f() // John  (this == user)
admin.f() // Admin  (this == admin)
admin['f']() // Admin (dùng dấu chấm hay dấu ngoặc vuông truy cập đến method - không thành vấn đề)
```

Quy luật đơn giản thôi: nếu `obj.f()` được gọi, thì `this` là `obj` trong suốt quá trình gọi `f`. Vì thế nó có thể là `user` hoặc `admin` ở ví dụ trên.

Hậu quả của việc không ràng buộc `this`

Nếu bạn đến từ một ngôn ngữ lập trình khác, thì bạn có thể sẽ quen thuộc ý tưởng "`this` ràng buộc", nơi mà các method được định nghĩa bên trong object luôn luôn có this tham chiếu đến object đó.

Trong Javascript `this` thì "tự do", giá trị của nó được tính toàn ngay tại lúc gọi và không dựa vào nơi method được khai báo, nhưng đúng hơn là đối tượng "trước dấu chấm".

Concept của việc run-time tính toán `this` có cả ưu điểm và nhược điểm. Một mặt, một chức năng có thể được sử dụng cho các object khác nhau. Mặc khác, tính linh hoạt cao hơn tạo ra nhiều khả năng mắc sai lầm hơn.

Ở đây, quan điểm của chúng ta không phải phán xét ngôn ngữ này tốt hay xấu. Cách chúng ta nên tiếp cận là hiểu cách hoạt động của nó, cách để nhận được các lợi ích từ việc this không bị ràng buộc và tránh các vấn đề.

Arrow function không có "this"

Arrow function thì đặc biệt: chúng không có `this` của nó.

Nếu chúng ta tham chiếu đến `this` trong một arrow function, nó sẽ lấy `this` bên ngoài nó.

Ví dụ:

```
'use strict'
function handle1() {
  console.log(this)
}
const handle2 = () => {
  console.log(this)
}
handle1() // undefined
handle2() // Window object
```

Một ví dụ khác, ở đây `arrow()` sẽ sử dụng `this` bên ngoài `arrow()`, tức lúc này là object `user`

```
'use strict'
let user = {
  firstName: 'Ilya',
  sayHi() {
    let arrow = () => alert(this.firstName)
    arrow()
  }
}
user.sayHi() // Ilya
```

Đó là tính năng đặc biệt của arrow function, nó hữu ích khi chúng ta tạo thực sự không muốn có một `this` riêng biệt.

Nếu không dùng arrow function thì lúc gọi this sẽ là `this` của function `arrow()` , mà `this` trong function `arrow()` là `undefined`

```
'use strict'
let user = {
  firstName: 'Ilya',
  sayHi() {
    function arrow() {
      alert(this.firstName) // Cannot read property 'firstName' of undefined
    }
    arrow()
  }
}
user.sayHi()
```

this ở trong một Event Handler

Trong một HTML event handler, `this` đề cập đến HTML element mà nó nhận event.

Khi nhấn vào button dưới đây thì nó sẽ được set `display:none`

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

this ở trong callback

`this` trong đoạn code này sẽ không đề cập đến object `delay`

```
const delay = {
  lastName: 'Duoc',
  print() {
    setTimeout(function () {
      console.log(this.lastName) // undefined
    }, 1000)
  }
}
delay.print()
```

để fix vấn đề này thì có thể dùng **arrow function**

```
const delay = {
  lastName: 'Duoc',
  print() {
    setTimeout(() => {
      console.log(this.lastName) // Duoc
    }, 1000)
  }
}
delay.print()
```

Lưu ý là `this` trong callback không đề cập đến function chứa callback đó, hãy cẩn thận!

`this` dưới đây không đề cập đến `broke` mà nó đề cấp đến obj.

```
function broke(func) {
  const obj = {
    name: 'duoc',
    func
  }
  return obj.func()
}

broke(function () {
```

```
  console.log(this) // obj
})
```

Vì thế để biết this trong callback đề cập đến cái nào thì phải hiểu được hàm chứa callback gọi callback như thế nào.

Tóm lại

- Function được lưu trữ bên trong thuộc tính object được gọi là "method" (phương thức).
- Method cho phép object "hành động" như là `object.doSomething()`.
- Method có thể tham chiếu đến object bằng cách dùng `this`.

Giá trị của `this` được xác định lúc **run-time**.

- Khi một function được khai báo, nó có thể sử dụng `this`, nhưng `this` không có giá trị cho đến khi function được gọi.
- Một function có thể được copy giữa các object
- Khi một function được gọi theo cú pháp "method": `object.method()`, giá trị của `this` ở method trong suốt quá trình chạy là `object`.

Hãy lưu ý rằng arrow function không có `this`. Khi `this` được truy cập bên trong arrow function, nó sẽ được lấy từ bên ngoài.

Xem thêm bài tập tại đây: Object methods và "this" trong Javascript, có bài tập thực hành

Higher order function

Mình đã viết 1 bài chi tiết tại đây: Chinh phục Higher Order Function, Closures, Currying và Callback trong Javascript

Higher order function là một **function** mà nhận vào tham số là **function** hoặc return về một **function**

```
const tinhTong = (a) => (b) => a + b
const ketQua = [1, 2, 3, 4, 5].map((item) => item * item)
console.log(tinhTong(1)(2)) // 3
console.log(ketQua) // [ 1, 4, 9, 16, 25 ]
```

Callback function

Callback function là một **function** mà được truyền vào một **function khác** như một tham số

```
const num = [2, 4, 6, 8]
num.forEach((item, index) => {
  console.log('STT: ', index, 'la ', item)
})
const result = num.map((item, index) => `STT: ${index} la ${item}`)
```

Closure

Closure là cách mà một **function cha** return về một **function con** bên trong nó. Ở trong **function con** đó có thể truy cập và thực thi các biến của **function cha**. Phải đủ 2 điều kiện này mới được gọi là **Closure** nhé.

```
const increase = () => {
  let x = 0
  const increaseInner = () => ++x
  return increaseInner
}
const myFunc = increase()
console.log(increase()()) // 1
console.log(increase()()) // 1
console.log(myFunc()) // 1
console.log(myFunc()) // 2
console.log(myFunc()) // 3
```

Currying

Currying là một kỹ thuật mà cho phép chuyển đổi một **function nhiều tham số** thành những **function liên tiếp có một tham số**.

```
const findNumber = (num) => (func) => {
  const result = []
  for (let i = 0; i < num; i++) {
    if (func(i)) {
      result.push(i)
    }
  }
  return result
}
findNumber(10)((number) => number % 2 === 1)
findNumber(20)((number) => number % 2 === 0)
findNumber(30)((number) => number % 3 === 2)
```