F.prototype

Hãy nhớ, một object mới có thể được tạo với một constructor function, như là `new F()`

Nếu `F.prototype` là một object, thì dùng `new` sẽ sử dụng nó để set Prototype cho object mới.

Ghi chú: Javascript có tính năng kế thường prototype từ khi bắt đầu. Nó chỉ là một trong những tính năng lõi của ngôn ngữ. Nhưng trong quá khứ không có cách nào để truy cập trực tiếp đến nó. Chỉ có cách là dùng thuộc tính `prototype` của constructor function, được mô tả trong bài này.

Lưu ý răng `F.prototype` ở đây nghĩa là một thuộc tính bình thường được đặt tên là `prototype` trên `F`. Nó nghe có vẻ tương tự "prototype" mà chúng ta thường nhắc đến, nhưng ở đây chúng chỉ là một cái tên thôi.

Ví dụ:

```
let animal = {
  eats: true
}

function Rabbit(name) {
  this.name = name
}

Rabbit.prototype = animal

let rabbit = new Rabbit('White Rabbit') //  rabbit.__proto__ == animal

alert(rabbit.eats) // true
```

Lưu ý: Nếu gán lại (hoặc thêm vào) thuộc tính `prototype` cho Function thì những object mà đã tạo từ constructor function sẽ vẫn giữ nguyên `[[Prototype]]` của nó, chỉ những object được tạo sau đó mới có `[[Prototype]]` mới

```
const animal = {
  eats: true
}
const human = {
  talks: true
}

function Rabbit(name) {
  this.name = name
}

Rabbit.prototype = animal
const rabbit_1 = new Rabbit('White Rabbit') //  rabbit_1.__proto__ == animal

Rabbit.prototype = human
```

```
const rabbit_2 = new Rabbit('Black Rabbit') //  rabbit_2.__proto__ == human
console.log(rabbit_1.eats) // true
console.log(rabbit_2.eats) // undefined
```

F.prototype mặc định, thuộc tính constructor

Mỗi function đều có thuộc tính `prototype` ngay cả khi chúng ta chưa gán hay cung cấp Mặc định thuộc tính `prototype` là một object chỉ có thuộc tính duy nhất là `constructor` trỏ ngược lại chính function đó.

Giống như thế này:

```
function Rabbit() {}

/* default prototype
Rabbit.prototype = { constructor: Rabbit };
*/
```

Chúng ta có thể kiểm tra:

```
function Rabbit() {}
// by default:
// Rabbit.prototype = { constructor: Rabbit }

alert(Rabbit.prototype.constructor == Rabbit) // true
```

Bình thường, nếu chúng ta không làm gì, thuộc tính `constructor` có sẵn trên tất cả rabbits thông qua `[[Prototype]]`

```
function Rabbit() {}
// Mặc định:
// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit() // kế thừa từ {constructor: Rabbit}

alert(rabbit.constructor == Rabbit) // true (from prototype)
```

Chúng ta có thể sử dụng thuộc tính `constructor` để tạo một object mới dựa trên một object đã tồn tại

Ví dụ:

```
function Rabbit(name) {
  this.name = name
  alert(name)
}
```

```
let rabbit = new Rabbit('White Rabbit')

let rabbit2 = new rabbit.constructor('Black Rabbit')
```

Điều này khá là tiện dụng nếu chúng ta có một object mà không biết nó được tạo từ một constructor function nào (Ví dụ nó đến từ một thư viện ngoài), và chúng ta cần tạo một object khác tương tự.

Nhưng điều quan trọng về `constructor` là **Javascript không đảm bảo đúng constructor mà bạn cần**

Nếu bạn thay thế prototype mặc định của function thì sẽ không còn `constructor` nữa. Ví dụ:

```
function Rabbit() {}
Rabbit.prototype = {
  jumps: true
}

let rabbit = new Rabbit()
alert(rabbit.constructor === Rabbit) // false
```

Vậy nên để giữ đúng `constructor`, chúng ta có thể thêm hoặc xóa các thuộc tính thông qua thuộc tính `prototype`

```
function Rabbit() {}

// Không ghi đè lên Rabbit.prototype
// Chỉ thêm vào
Rabbit.prototype.jumps = true
// Rabbit.prototype.constructor mặc định được bảo toàn
```

Hoặc có thể tạo một `constructor` bằng tay như thế này

```
Rabbit.prototype = {
  jumps: true,
  constructor: Rabbit
}
```