Prototype và kế thừa

1. Dự đoán giá trị được in ra dưới đây

```
let person = {
  name: ""
}
let student = {
  __proto__: person,
  name: "Alan"
}
console.log(student.name) // ?
delete student.name
console.log(student.name) // ?
delete person.name
console.log(student.name) // ?
```

Trả lời

```
let person = {
  name: ""
}
let student = {
  __proto__: person,
  name: "Alan"
}
console.log(student.name) // Alan
delete student.name
console.log(student.name) // ""
delete person.name
console.log(student.name) // undefinded
```

2. Cho các object sau

```
let head = {
  glasses: 1
}

let table = {
  pen: 3
}

let bed = {
  sheet: 1,
  pillow: 2
}
```

```
let pockets = {
  money: 2000
}
```

- Sử dụng `__proto__` để kế thừa các object theo đường dẫn `pockets` -> `bed` -> `table` -> `head`. Ví dụ `pockets.pen = 3`, `bed.glasses = 1`
- Truy cấp đến `glasses` bằng `pockets.glasses` và `head.glasses`, cách nào nhanh hơn

Trả lời

```
let head = {
  glasses: 1
}

let table = {
  pen: 3,
  __proto__: head
}

let bed = {
  sheet: 1,
  pillow: 2,
  __proto__: table
}

let pockets = {
  money: 2000,
  __proto__: bed
}

alert(pockets.pen) // 3
alert(bed.glasses) // 1
alert(table.money) // undefined
```

Trong các engine hiện đại ngày nay thì không có sự khác biệt giữa việc lấy một thuộc tính từ object hay từ prototype của nó. Chúng đủ thông minh để thực hiện điều này nên chúng ta không cần quá lo lắng về hiệu năng.

3. Thuộc tính thuộc về đâu? Chúng ta có `rabbit` kế thừa từ `animal`. Nếu gọi `rabbit.eat()`, object nào sẽ nhận thuộc tính `full`: `animal` hay `rabbit`?

```
let animal = {
  eat() {
    this.full = true
  }
}

let rabbit = {
```

```
  __proto__: animal
}

rabbit.eat()
```

Trả lời Câu trả lời là: `rabbit` Bởi vì `this` chính là object trước dấu chấm. Hãy nhớ khai báo và thực thi code là 2 việc khác nhau, nếu thoạt nhìn ban đầu chỉ khai báo thì `this` đại diện cho `animal` nhưng khi thực thi `rabbit.eat()` thì this đại diện cho `rabbit`

4. Tại sao cả 2 hamster đều có thức ăn Chúng ta có 2 loại hamster: `speedy` và `lazy` kế thừa từ object `hamster` Khi chúng ta cho 1 loại hamster ăn, tại sao loại kia cũng có thức ăn. Hãy tìm cách sửa nó!

```
let hamster = {
  stomach: [],

  eat(food) {
    this.stomach.push(food)
  }
}

let speedy = {
  __proto__: hamster
}

let lazy = {
  __proto__: hamster
}

// Cho speedy ăn
speedy.eat("apple")
alert(speedy.stomach) // apple

// lazy vẫn có thức ăn, tại sao? hãy sửa nó
alert(lazy.stomach) // apple
```

Trả lời

Cùng nhìn lại cẩn thận chuyện gì đang xảy ra trong `speedy.eat("apple")`.

1. Phương thức `speedy.eat` được tìm thấy trong prototype, khi thực thi thì `this=speedy` (object trước dấu chấm).
2. Khi `this.stomach.push()` cần tìm thuộc tính `stomach` và gọi `push`. Nó tìm `stomach` trong `this`(`=speedy`), nhưng không tìm thấy.
3. Sau đó nó tìm trong prototype và tìm thấy `stomach` trong `hamster`.
4. Sau đó nó gọi `push` vào đó, thêm thức ăn vào `stomach` trong prototype.

Vì thế tất cả hamster đều được share `stomach`.

Hãy lưu ý là những trường hợp như thế này sẽ không xảy ra khi chúng ta gán `this.stomach=`:

```
let hamster = {
  stomach: [],

  eat(food) {
    // assign to this.stomach instead of this.stomach.push
    this.stomach = [food]
  }
}

let speedy = {
  __proto__: hamster
}

let lazy = {
  __proto__: hamster
}

// Speedy one found the food
speedy.eat("apple")
alert(speedy.stomach) // apple

// Lazy one's stomach is empty
alert(lazy.stomach) // <nothing>
```

Bây giờ mọi thứ đều hoạt động ổn bởi vì `this.stomach=` không tìm `stomach` nữa mà nó sẽ viết trực tiếp vào object `this`

Chúng ta cũng có thể tránh vấn đề này bằng cách khai báo mỗi hamster thuộc tính `stomatch` của chính nó.

```
let hamster = {
  stomach: [],

  eat(food) {
    this.stomach.push(food)
  }
}

let speedy = {
  __proto__: hamster,
  stomach: []
}

let lazy = {
  __proto__: hamster,
  stomach: []
}

// Speedy one found the food
speedy.eat("apple")
```

```
alert(speedy.stomach) // apple

// Lazy one's stomach is empty
alert(lazy.stomach) // <nothing>
```

5. Thay đổi `prototype` Trong đoạn code phía dưới, chúng ta tạo mới một `Rabbit`, sau đó thử thay đổi prototype của nó Đầu tiên chúng ta có code này:

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

let rabbit = new Rabbit()

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

1. Thay đổi một chút, alert sẽ hiển thị gì

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

let rabbit = new Rabbit()

Rabbit.prototype = {}

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

2. Và nếu code như thế này

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

let rabbit = new Rabbit()

Rabbit.prototype.eats = false

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

3. Hoặc như thế này

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

let rabbit = new Rabbit()

delete rabbit.eats

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

4. Cuối cùng là như thế này

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

let rabbit = new Rabbit()

delete Rabbit.prototype.eats

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

Trả lời

1. `true` Việc gán `Rabbit.prototype` sẽ làm thay đổi prototype cho các object mới, nhưng nó không có hiệu lực đối với các object đã tồn tại
2. `false` Chúng ta thực hiện mutate prototype, việc này không phải là tạo mới prototype nên khi thay đổi giá trị bên trong prototype thì các object tham chiếu đến cũng sẽ bị thay đổi theo.
3. `true` Toán tử `delete` áp dụng trực tiếp lên object. Ở đây `delete rabbit.eats` cố xóa thuộc tính `eats` từ `rabbit`, nhưng nó không có ở đó. Vì thế hành động này không có hiệu lực.
4. `undefined` Thuộc tính `eats` bị xóa trực tiếp từ prototype, nó không còn tồn tại.

6. Tạo object với cùng constructor Giả sử chúng ta có object là `obj` được tạo từ một constructor function bất kỳ. Chúng ta có thể tạo một object `obj2` như thế này được không

```
let obj2 = new obj.constructor()
```

Đưa ra ví dụ về một constructor function như vậy mà hoạt động đúng, đưa ra 1 ví dụ khác mà nó hoạt động sai.

Trả lời Ví dụ về hoạt động

```
function User(name) {
  this.name = name
}

let user = new User("John")
let user2 = new user.constructor("Pete")

alert(user2.name) // Pete (worked!)
```

Ví dụ về không hoạt động

```
function User(name) {
  this.name = name
}
User.prototype = {} // (*)

let user = new User("John")
let user2 = new user.constructor("Pete")

alert(user2.name) // undefined
```

Tại sao `user2.name` là `undefined`? Đây là cách mà `new user.constructor('Pete')` làm việc

1. Đầu tiên nó nhìn vào `constructor` trong user. Không có.
2. Sau đó nó tìm trong prototype. Prototype của `user` là `User.prototype`, và nó cũng không có.
3. Giá trị của `User.prototype` là một object rỗng `{}`, ví thế prototype tiếp theo của nó là `Object.prototype`. Và ta có `Object.prototype.constructor === Object` được sử dụng.

Cuối cùng, chúng ta có `let user2 = new Object('Pete')`.

7. Sử dụng prototype thêm phương thức `defer(ms)` vào tất cả các function, phương thức này sẽ làm hàm thực thi sau mili giây

```
function f() {
  alert("Hello!")
}

f.defer(1000) // shows "Hello!" after 1 second
```

Trả lời

```
Function.prototype.defer = function (ms) {
  setTimeout(this, ms)
}

function f() {
```

```
  alert("Hello!")
}

f.defer(1000) // shows "Hello!" after 1 sec
```

8. Tương tự bài trên

```
function f(a, b) {
  alert(a + b)
}

f.defer(1000)(1, 2) // shows 3 after 1 second
```

Trả lời

```
Function.prototype.defer = function (ms) {
  let f = this
  return function (...args) {
    setTimeout(() => f(...args), ms)
  }
}

// check it
function f(a, b) {
  alert(a + b)
}

f.defer(1000)(1, 2) // shows 3 after 1 sec
```