

MỤC LỤC

LỜI NÓI ĐẦU	6
I.Hàm function	7
1.1.Kiến thức nền về hàm function	7
1.1.1.Cách tối thiểu hóa số biến toàn cục	8
1.2.Định nghĩa thuật ngữ	13
1.3.So sánh giữa việc khai báo với biểu thức : tên names và sự leo thang hoisting	15
1.4.Thuộc tính name của hàm function	16
1.5.Sự leo thang của hàm function - Function Hoisting.....	17
1.6.Kiểu mẫu callback	19
1.6.1.Một ví dụ mẫu về callback	19
1.6.2.Hàm callback và phạm vi tồn tại của biến trong hàm callback.....	22
1.6.3.Lắng nghe sự kiện không đồng bộ.....	24
1.6.4.Độ trễ Timeouts.....	25
1.6.5.Hàm callbacks trong các thư viện.....	25
1.6.6.Cách trả về trong hàm functions.....	25
1.6.7.Các hàm functions tự định nghĩa.....	27
1.6.8.Các hàm functions tức thời	29
1.6.8.1.Các tham số của 1 hàm tức thời.....	30
1.6.8.2.Các giá trị được trả về từ các hàm tức thời	31
1.6.8.3.Cách sử dụng và ưu điểm.....	33
1.6.9.Cách khởi tạo các đối tượng tức thời.....	34
1.6.10.Phân nhánh thời gian khởi tạo - Init-Time Branching.....	35
1.6.11.Các thuộc tính của functions – kiểu mẫu tối ưu hóa bộ nhớ Memoization Pattern..	37
1.6.12.Các đối tượng cấu hình - Configuration Objects.....	39
1.6.13.Curry	41
1.6.14.Function Application	41
1.6.15.Partial Application.....	42
1.6.16.Currying	44
1.6.17.khi nào ta sử dụng Currying.....	47
II.Tử mảng arrays cho tới các đối tượng objects	48
2.1.Các phân tử, các thuộc tính, các phương thức	50

2.2.Các mảng arrays kết hợp.....	50
2.3.Các truy cập vào các thuộc tính của đối tượng	51
2.4.Cách gọi các phương thức của 1 đối tượng	53
2.5.Cách biến đổi các thuộc tính / phương thức	54
2.6.Các sử dụng từ khóa this.....	55
2.7.Các hàm tạo constructor	55
2.8.Đối tượng toàn cục Global Object.....	57
2.9.Thuộc tính tạo	58
2.10.Toán tử instanceof	59
2.11.Các hàm functions mà trả về các đối tượng objects	59
2.12.Các chuyển tiếp các đối tượng	61
2.13.Cách so sánh các đối tượng objects.....	62
III.Các đối tượng được xây dựng sẵn.....	63
3.1.Đối tượng object.....	63
3.2.Đối tượng Mảng array	64
3.3.Đối tượng Function	67
3.3.1.Các thuộc tính của đối tượng function.....	68
3.3.2.Các phương thức trong các đối tượng objects function	71
IV.Biểu thức chính quy	74
4.1.Các thuộc tính của các đối tượng Objects	74
4.2.Các phương thức của các đối tượng regexp.....	76
4.3.Các phương thức string mà nhận các biểu thức chính quy như các tham số.....	76
4.3.1.search() và match()	77
4.3.2.replace()	77
4.3.3.Các hàm Replace callbacks	78
4.3.4.split().....	80
4.3.5.Cách chuyển tiếp 1 chuỗi string khi 1 regexp được như kì vọng	80
4.4.Quy tắc mẫu trong biểu thức quan hệ.....	81
V.Prototype.....	88
5.1.Thuộc tính prototype	88
5.2.Cách thêm các phương thức và thuộc tính bằng cách sử dụng Prototype	88
5.3.Cách sử dụng các thuộc tính và phương thức của Prototype	90
5.4.Các thuộc tính chính gốc so với các thuộc tính được thêm vào qua prototype	91

5.5.Ghi đè thuộc tính của prototype với thuộc tính chính gốc.....	93
5.6.Các liệt kê các thuộc tính.....	94
5.7.isPrototypeOf().....	97
5.8.Bí mật về __proto__ Link	98
5.9.Cách làm gia tăng các đối tượng được xây dựng sẵn.....	100
5.10.Một vài ghi chú về Prototype	101
VI.Sự kế thừa.....	106
6.1.Các chuỗi móc nối prototype	106
6.2.Ví dụ về chuỗi móc nối prototype	107
6.3.Cách chuyển các thuộc tính được chia sẻ vào bên trong prototype	111
6.4.Cách thừa kế chỉ từ prototype	113
6.5.Hàm tạo Constructor tạm thời - new F().....	115
6.6.Uber – truy cập đối tượng cha từ đối tượng con	117
6.7.Cách phân tách phần kế thừa vào trong 1 hàm function.....	119
VII.Các kiểu mẫu tạo đối tượng object	120
7.1.Các phương thức và thuộc tính riêng tư	120
7.1.2.Các thành viên riêng tư	120
7.1.3.Các phương thức được ưu tiên	121
7.1.4.Các thiếu sót quyền riêng tư.....	122
7.1.5.Object Literals và quyền riêng tư	123
7.1.6.Các Prototypes và quyền riêng tư.....	125
7.1.7.Cách phát hiện các hàm riêng như các phương thức public	126
7.2.Cách thành viên cố định static	127
7.2.1.Các thành viên Public Static.....	128
7.2.2.Các thành viên Private Static	130
7.3.Các hằng số đối tượng	133
7.4.Kiểu mẫu chuỗi hóa - Chaining Pattern.....	135
7.4.1.Uưu điểm và nhược điểm của kiểu mẫu chuỗi hóa - Chaining Pattern	136
7.5.Phương thức method().....	137
VIII.Các kiểu mẫu có khả năng tái sử dụng - Code Reuse Patterns	139
8.1.Kiểu mẫu hướng class so với các kiểu mẫu phân cấp hiện đại.....	139
8.2.Kết quả như mong muốn khi sử dụng sự kế thừa theo hướng classical	140
8.3.Việc kế tiếp chuỗi prototype	141

8.4.Nhược điểm khi sử dụng kiểu mẫu #1.....	144
8.5.Kiểu mẫu Classical Pattern #2 - Rent-a-Constructor – kiểu mẫu vay mượn hàm tạo.	145
8.5.1.Chuỗi prototype	147
8.5.2.Sự đa kế thừa bằng các hàm tạo vay mượn.....	148
8.5.3.Uưu điểm và nhược điểm của kiểu mẫu hàm khởi tạo vay mượn.....	149
8.6.Kiểu mẫu Classical Pattern #3 - Rent and Set Prototype – kiểu mẫu vay mượn và thiết lập prototype	149
8.7.Kiểu mẫu Classical Pattern #4 – chia sẻ prototype	151
8.8.Kiểu mẫu Classical Pattern #5 – 1 hàm tạo tạm thời	152
8.8.1.Cách lưu trữ Superclass	154
8.8.2.Cách cài đặt lại con trỏ hàm khởi tạo	154
8.9.Klass	156
8.10.Sự kế thừa hướng Prototypal.....	159
8.10.1.Thảo luận.....	160
8.10.2.Việc bổ sung ECMAScript 5.....	162
8.10.3.Sự kế thừa bằng cách sao chép các thuộc tính	162
8.10.4.Mix-ins	165
8.10.5.Vay mượn các phương thức Methods.....	166
8.10.6.Ví dụ : vay mượn từ 1 mảng array.....	167
8.10.7.Vay mượn và ràng buộc	168
8.10.8.Function.prototype.bind()	170
IX.Kiểu mẫu thiết kế	172
9.1.Singleton – kiểu mẫu duy nhất.....	172
9.1.1.Cách sử dụng new.....	173
9.1.2.Thực thể trong 1 thuộc tính static	174
9.1.3.Thực thể trong 1 bao đóng	174
9.2.Factory – kiểu mẫu sản xuất đối tượng.....	178
9.2.1.Đối tượng Object Factory được xây dựng sẵn	181
9.3.Iterator – kiểu mẫu biến lặp	182
9.4.Façade	185
9.5.Proxy.....	188
9.5.1.Một ví dụ mẫu	190
9.6.Composite	199
9.6.1.Vấn đề đặt ra.....	199

9.6.2.Cấu trúc của kiểu mẫu composite pattern	202
9.6.3.Các mẫu ví dụ về kiểu mẫu Composite Pattern.....	203
9.6.4.Mẫu ví dụ trong JS sử dụng kiểu mẫu Composite Pattern.....	203
9.6.5.Uưu điểm và nhược điểm của kiểu mẫu Composite Pattern.....	208
9.7.Observer – người quan sát	210
9.7.1.Mẫu ví dụ #1: việc đặt mua tạp chí.....	211
X.Asynchronous patterns - Kiểu mẫu không đồng bộ	216
10.1.Lợi ích và thách thức với lập trình không đồng bộ	216
10.2.Promises	218
10.3.Khám phá Promises trong bộ công cụ JQuery	222
XI.ASYNCH JS : sức mạnh của đối tượng \$.DEFERRED	225
11.1.Các hàm APIS không đồng bộ của trình duyệt	225
11.2.Cách tạo các ứng dụng 1 cách không đồng bộ	227
11.3.Cách hiệu chỉnh sự thất bại.....	228
11.4.\$.DEFERRED	228
11.5.Các trường hợp mẫu	232
11.6.JQuery.Deferred và Promise	233
11.6.1\$.Deferred.....	234
11.6.2.Phương thức deferred.resolve()	235
11.6.3.Phương thức deferred.reject().....	235
11.6.4.Phương thức Promise()	236

LỜI NÓI ĐẦU

Tài liệu được viết giống như 1 bản ghi chép, ghi lại những thứ mà mình đã đọc được từ các ebook tiếng anh, do vậy bố cục sắp xếp của nó có thể chưa chính xác, cách trình bày không theo chuẩn 1 ebook nào cả và nhiều chỗ viết chưa rõ nghĩa và không mạch lạc do hạn chế về ngoại ngữ của bản thân mình. Tài liệu mang đậm tính chất cá nhân do vậy bạn sẽ bắt gặp trong tài liệu này nhiều đoạn kí tự in đậm, in màu, cỡ chữ lớn bất thường và được tô màu khác nhau - đó là các đoạn có liên quan đến nhau hay là những ghi chú quan trọng mà bạn cần phải đọc kĩ.

Nội dung trong tài liệu này được dựa trên các cuốn ebook: **“JavaScript Patterns - Stoyan Stefanov”**, **“Object-Oriented JavaScript - Stoyan Stefanov”**, **“JavaScript: The Good Parts - Douglas Crockford”**, và 1 số bài viết của các tác giả mà mình không còn nhớ rõ nữa.

Trước khi đọc tài liệu này, bạn phải xác định mình nắm rõ những gì thuộc về cơ bản nhất của Javascript như các cách khai báo, các kiểu dữ liệu cơ bản, các phép toán số học, các mệnh đề điều kiện, ...(nếu bạn có kiến thức về 1 ngôn ngữ lập trình cơ bản nào đó, thì bạn sẽ làm quen với những gì tài liệu này bỏ qua trong Javascript 1 cách rất nhanh chóng). Tài liệu này chỉ tập trung vào những gì được coi là đặc biệt nhất của Javascript so với các ngôn ngữ lập trình thông dụng khác, không phải dành cho những người mới bắt đầu học Javascript. 1 điểm nữa bạn cần lưu ý là trong tài liệu này lược bỏ phần tương tác giữa Javascript với DOM & Browser bởi vì bản thân mình dùng JQuery để thay thế.

Nếu bạn sử dụng Javascript kết hợp với Phonegap để tạo ra các ứng dụng cho các nền tảng mobile, thì các tài liệu sau có thể bạn sẽ quan tâm tới :

- **“Phonegap cho người mới học”:**
<http://www.slideshare.net/myloveforyoungt/phonegap-cho-nguoi-moi-hoc>
- **“Cách tối ưu hóa môi trường lập trình ứng dụng cho Android”:**
<http://www.slideshare.net/myloveforyoungt/cch-ti-u-ha-mi-trng-lp-trnh-ng-dng-cho-android-tng-tc-my-o-android>
- **Hoặc đơn giản truy cập vào thư mục chia sẻ sau:** <http://sdrv.ms/VoAXBi>

I.Hàm function

1.1.Kiến thức nền về hàm function

Có 2 tính năng chính của hàm functions trong JS làm cho chúng trở nên đặc biệt.điều đầu tiên là hàm functions cũng giống như các đối tượng class objects và điều thứ 2 là chúng cung cấp 1 phạm vi tồn tại của các biến

Các hàm functions là các đối tượng objects mà :

- Có thể tạo ra 1 cách linh hoạt trong thời gian chạy trong suốt quá trình thực thi của chương trình
- Có thể gán vào các biến và có thể tham chiếu để sao chép giá trị sang các biến khác, có thể tham số hóa và trong 1 số trường hợp nó có thể bị xóa
- Có thể chuyển tiếp như các tham số vào trong hàm functions khác và cũng có thể được trả về bởi hàm functions khác
- Có thể có thuộc tính và các phương thức ở bên trong

Do vậy có thể xảy ra trường hợp 1 hàm function A – là 1 đối tượng object, đối tượng này chứa các thuộc tính và phương thức – 1 hàm B khác, hàm B nhận 1 hàm C là tham số và khi thực thi nó có thể trả về hàm D.

Thông thường khi ta nghĩ về 1 hàm function trong JS thì ta nghĩ về 1 đối tượng object với tính năng đặc biệt duy nhất là đối tượng object này có khả năng gọi được, điều này có nghĩa là nó có thể được thực thi xử lý

Thực tế thì hàm function là đối tượng object và hiển nhiên ta sẽ thấy hàm tạo new Function() như sau :

```
// antipattern
// for demo purposes only
var add = new Function('a, b', 'return a + b');
add(1, 2); // returns 3
```

ở đoạn code này thì không có sự ngờ vực gì về việc add() là 1 đối tượng object. cách sử dụng hàm tạo Function() constructor không phải là cách sử dụng hay vì nó sẽ gây ra 1 vài bất tiện cho việc viết và đọc hiểu

tính năng quan trọng thứ 2 là các hàm function cung cấp 1 phạm vi tồn tại. bất cứ biến nào được định nghĩa với var bên trong 1 hàm function thì là 1 biến địa phương và nó không tồn tại được bên ngoài hàm function này. việc nói rằng dấu ngoặc {} không cung cấp phạm vi địa phương có nghĩa là nếu ta định nghĩa 1 biến với var bên trong 1 mệnh đề if hay bên trong 1 mệnh đề for hay mệnh đề while thì điều này không có nghĩa là biến này là địa phương với if hay for. nó chỉ là địa phương đối với hàm function bao quanh nó và nếu không có hàm function nào bao quanh nó thì nó là 1 biến toàn cục. như đã nói đến ở chương trước thì việc **tối thiểu hóa số biến toàn cục** là 1 sở thích tốt bởi vì các hàm functions là bắt buộc để giữ các biến tồn tại trong 1 phạm vi có kiểm soát

1.1.1. Cách tối thiểu hóa số biến toàn cục

Js sử dụng các hàm functions để kiểm soát phạm vi tồn tại của biến. 1 biến được khai báo trong 1 hàm function là biến địa phương và nó không tồn tại ngoài hàm function này. theo cách khác, các biến toàn cục là các biến được khai báo không ở trong bất cứ hàm functions nào

Mọi môi trường trong JS đều là 1 đối tượng toàn cục global object có thể được truy cập khi ta sử dụng this ở bên ngoài bất cứ hàm functions nào. mọi biến toàn cục mà ta tạo ra trở thành 1 thuộc tính của đối tượng toàn cục. dưới đây là 1 đoạn mã code nhỏ chỉ ra cách tạo và truy cập 1 biến toàn cục trong môi trường trình duyệt :

```
myglobal = "hello"; // antipattern
console.log(myglobal); // "hello"
console.log(window.myglobal); // "hello"
console.log(window["myglobal"]); // "hello"
console.log(this.myglobal); // "hello"
```

1.1.1.2. Các vấn đề nảy sinh với các biến toàn cục

các vấn đề nảy sinh với biến toàn cục là việc chúng được chia sẻ trong phạm vi toàn bộ mã code trong ứng dụng JS. chúng sống trong cùng 1 namespace toàn cục và sẽ luôn luôn có 1 tình huống xảy ra va chạm trong cách đặt tên – khi 2 phần tách riêng của 1 ứng dụng định nghĩa các biến toàn cục với cùng 1 tên nhưng có mục đích sử dụng khác nhau

đây cũng là vấn đề thường xảy ra cho các web pages nhưng mã code không được viết bởi nhà lập trình của trang page đó ví dụ :

- 1 thư viện JS của bên thứ 3
- Các mã Scripts từ 1 đối tác quảng cáo
- Mã code từ 1 mã phân tích và lưu vết của người sử dụng bên thứ 3
- Các loại widgets, badges, và buttons khác nhau

Chúng ta nói rằng 1 trong các mã scripts của bên thứ 3 định nghĩa 1 biến toàn cục được gọi ví dụ như result.sau đó bên trong 1 trong các hàm functions của ta định nghĩa 1 biến toàn cục khác được gọi là result.kết quả của điều này là biến result khai báo cuối cùng sẽ ghi đè lên biến result trước đó và mã script của bên thứ 3 rất có thể không hoạt động

Do vậy điều quan trọng để làm 1 người hàng xóm thân thiện với các mã scripts khác là các chiến thuật tối thiểu hóa số biến toàn cục

Khá là ngạc nhiên là rất dễ dàng tạo ra biến toàn cục 1 cách rất vô tình trong JS bởi vì 2 tính năng của JS.điều đầu tiên là ta sử dụng các biến mà không khai báo chúng.và điều thứ 2 là JS luôn có ngụ ý là toàn cục với bất kì biến nào mà ta không khai báo thì nó sẽ trở thành 1 thuộc tính của đối tượng toàn cục global object.xét ví dụ sau :

```
function sum(x, y) {  
  // antipattern: implied global  
  result = x + y;  
  return result;  
}
```

Trong mã code trên thì result được sử dụng mà không khai báo.đoạn code vẫn hoạt động tốt nhưng sau khi hàm function này ta kết thúc với 1 biến toàn cục result thì đây có thể là nguồn gốc của nhiều vấn đề

Theo kinh nghiệm thì luôn luôn khai báo các biến với var, và ta sửa lại ví dụ trước như sau :

```
function sum(x, y) {  
  var result = x + y;  
  return result;  
}
```

1 trong những lỗi dễ dàng tạo ra các biến theo hướng toàn cục khác chính là chuỗi gán liên tiếp các biến var.xét đoạn ví dụ sau, thì biến a là địa phương nhưng biến b là toàn cục :

```
// antipattern, do not use
function foo() {
var a = b = 0;
// ...
}
```

Nếu ta ngạc nhiên là tại sao lại như vậy thì đó là bởi vì sự đánh giá đi từ phải sang trái.đầu tiên biểu thức `b = 0` được đánh giá trước và tại đây thì b không được khai báo.giá trị trả về của biểu thức này là 0 và nó được gán vào 1 biến địa phương mới được khai báo với var a.theo cách khác ta có thể viết lại như sau :

```
var a = (b = 0);
```

nếu ta đã khai báo tất cả các biến và thực hiện chuỗi gán thì đây là cách tốt và ta không tạo ra các biến toàn cục 1 cách vô tình nữa :

```
function foo() {
var a, b;
// ...
a = b = 0; // both local
}
```

1.1.1.3.Các hiệu ứng phụ khi ta quên var

Đây là 1 điểm hơi khác biệt giữa các biến ngụy ý toàn cục và sự định nghĩa hoàn toàn là 1 biến toàn cục – sự khác biệt là khả năng không xác định được các biến này bằng cách sử dụng toán tử delete :

- Các biến toàn cục được tạo ra với var (các biến được tạo ra bên ngoài bất cứ hàm functions nào) có thể không bị xóa
- Các biến ngụy ý toàn cục không được tạo ra với var (bất chấp nếu nó được tạo ra bên trong hàm function) có thể xóa

Điều này chỉ ra rằng các biến toàn cục nói theo cách kỹ thuật không phải là các biến thật sự nhưng chúng là các thuộc tính của đối tượng toàn cục global object. các thuộc tính có thể bị xóa bởi toán tử delete trái lại cũng có biến không thể bị xóa :

```
// define three globals
```

```
var global_var = 1;
```

```
global_novar = 2; // antipattern
```

```
(function () {
```

```
global_fromfunc = 3; // antipattern
```

```
})();
```

```
// attempt to delete
```

```
delete global_var; // false
```

```
delete global_novar; // true
```

```
delete global_fromfunc; // true
```

```
// test the deletion
```

```
typeof global_var; // "number"
```

```
typeof global_novar; // "undefined"
```

```
typeof global_fromfunc; // "undefined"
```

trong chế độ nghiêm ngặt của ES5 thì việc gán các biến chưa được khai báo (giống như 2 biến antipatterns ở đoạn mã trên) sẽ văng ra 1 lỗi

1.1.1.4.Cách truy cập đối tượng toàn cục

trong các trình duyệt, đối tượng toàn cục có thể truy cập từ bất cứ phần nào của đoạn code thông qua thuộc tính window. nếu ta cần truy cập biến toàn cục mà không gõ mã cứng xác định với window thì ta có thể làm như sau từ bất cứ cấp nào của các phạm vi function lồng nhau :

```
var global = (function () {
```

```
return this;
```

```
})();
```

Cách này ta có thể luôn luôn lấy về đối tượng toàn cục bởi vì bên trong các hàm functions thì this luôn luôn chỉ tới 1 đối tượng toàn cục. **điều này thực sự không còn trong chế độ nghiêm ngặt của ECMAScript 5, vì vậy ta phải chấp nhận 1 kiểu mẫu khác khi mã code của ta ở trong chế độ nghiêm ngặt.** ví dụ nếu ta đang phát triển 1 thư viện thì ta có thể đóng gói mã thư viện vào ngay 1 hàm function (như sẽ nói ở chương 4) và sau đó từ phạm vi toàn cục, thì chuyển tiếp vào 1 tham chiếu tới this như 1 tham số trong hàm function

1.1.1.5. Kiểu mẫu sử dụng var

Bằng cách sử dụng cấu trúc var ở đầu các hàm functions là 1 kiểu mẫu rất hữu dụng và được chấp nhận. nó có những ưu điểm sau :

- Cung cấp 1 địa điểm cho việc xem xét toàn bộ các biến địa phương cần có của function
- Ngăn chặn các lỗi về logic khi 1 biến được sử dụng trước khi nó được định nghĩa
- Giúp ta nhớ đã khai báo các biến nào và do đó tối thiểu hóa các biến toàn cục
- Phải gõ ít mã code hơn

1 kiểu mẫu với var nhìn như sau :

```
function func() {  
  
var a = 1,  
b = 2,  
sum = a + b,  
myobject = { },  
i,  
j;  
// function body...  
}
```

Ta sử dụng 1 cấu trúc var và khai báo nhiều biến được phân tách bởi dấu phẩy. nó là 1 cách thực hành tốt. điều này có thể ngăn chặn các lỗi về logic và cũng khiến đoạn mã code dễ đọc hơn. ta cũng có thể làm điều tương tự với DOM :

```
function updateElement() {  
var el = document.getElementById("result"),
```

```
style = el.style;  
// do something with el and style...  
}
```

1.1.1.6. Sự leo thang : 1 vấn đề với các biến phân tán

JS cho phép ta có nhiều cấu trúc var ở bất cứ đâu trong hàm function và tất cả chúng đều giống với các biến được khai báo trên đầu hàm function. trạng thái xử lý này được gọi là sự leo thang hoisting. điều này có thể hướng tới các lỗi về logic khi ta sử dụng 1 biến và sau đó gán nó lần nữa ở trong hàm function. với JS thì miễn là 1 biến nằm trong cùng 1 phạm vi (cùng trong 1 hàm function) thì nó được coi là đã khai báo và ngay khi nó được sử dụng trước khi khai báo var, ví dụ như sau :

```
// antipattern  
myname = "global"; // global variable  
function func() {  
    alert(myname); // "undefined"  
    var myname = "local";  
    alert(myname); // "local"  
}  
func();
```

Sự khai báo này dù xảy ra trước hay sau thì nó cũng xóa đi sự khai báo toàn cục trước đó – leo thang cản trở lại sự khai báo toàn cục

1.2. Định nghĩa thuật ngữ

ta xét đoạn mã code sau :

```
// named function expression  
var add = function add(a, b) {  
    return a + b;  
};
```

Đoạn mã code này chỉ ra 1 hàm function bằng cách sử dụng biểu thức đặt tên hàm named function expression.

Nếu ta bỏ qua tên (tên hàm chính là **add**) trong biểu thức trên thì ta sẽ có 1 biểu thức

hàm vô danh unnamed function expression hay đơn giản được gọi là function expression hay thông dụng nhất là anonymous function. như ví dụ sau :

```
// function expression, a.k.a. anonymous function
var add = function (a, b) {
  return a + b;
};
```

Khi ta bỏ qua tên của hàm function là add thì chính là ta đang sử dụng biểu thức unnamed function expression, điều này không gây ảnh hưởng gì tới định nghĩa và sự gọi ra của hàm function. chỉ có 1 sự khác biệt là thuộc tính name của đối tượng function object sẽ là 1 chuỗi rỗng. thuộc tính name này là 1 phần mở rộng của ngôn ngữ (nó không phải là 1 phần trong chuẩn ECMA) nhưng nó lại sẵn sàng có trong nhiều môi trường. nếu ta giữ lại tên hàm là add thì thuộc tính add.name sẽ chứa chuỗi add. và thuộc tính name rất hữu dụng khi sử dụng 1 công cụ debug giống như Firebug hay khi ta gọi cùng 1 hàm function 1 cách đệ quy ngay trong thân của nó hay theo mặt khác ta cũng có thể bỏ qua nó

Cuối cùng, ta có các khai báo hàm. nó cũng giống như cách khai báo hàm function trong những ngôn ngữ khác :

```
function foo() {
  // function body goes here
}
```

Trong các điều lệ về ngữ pháp thì biểu thức named function expressions và function declarations là tương tự nhau. về bản chất nếu ta không gán kết quả của biểu thức function expression là 1 biến, thỉnh thoảng cũng không tìm ra sự khác nhau giữa 1 khai báo hàm function declaration và 1 biểu thức function expression

Có sự khác nhau giữa 2 cái là dấu chấm phẩy. **dấu chấm phẩy** là không cần thiết trong khai báo hàm function declarations nhưng nó lại là bắt buộc trong biểu thức function expressions và ta nên luôn luôn sử dụng nó mặc dù cơ chế tự điền dấu chấm phẩy có thể tự làm cho ta

1.3. So sánh giữa việc khai báo với biểu thức : tên names và sự leo thang hoisting

Vậy ta nên sử dụng khai báo hàm function declarations hay sử dụng biểu thức hàm function expressions?

Trong nhiều trường hợp theo cú pháp thì ta không thể sử dụng 1 khai báo. ví dụ nhúng chuyển tiếp 1 đối tượng function object giống như 1 tham số hay định nghĩa các phương thức trong chuỗi miêu tả đối tượng :

```
// this is a function expression,  
// passed as an argument to the function `callMe`
```

```
callMe(function () {  
  
// I am an unnamed function expression  
// also known as an anonymous function  
});
```

```
// this is a named function expression
```

```
callMe(function me() {  
  
// I am a named function expression  
// and my name is "me"  
});
```

```
// another function expression
```

```
var myobject = {  
say: function () {  
  
// I am a function expression  
}  
};
```

Sự khai báo hàm Function declarations chỉ có thể xuất hiện trong các mã hướng thủ tục, điều này có nghĩa là bên trong thân của các hàm functions khác hay nó ở phạm vi toàn cục. định nghĩa của chúng có thể không được gán vào trong các biến hay các thuộc tính hay xuất hiện trong lời gọi hàm như là tham số. đây là 1 ví dụ cho phép dùng các định nghĩa hàm Function declarations với tất cả các hàm foo(), bar(), và local() đều được định nghĩa sử dụng kiểu mẫu khai báo function declaration pattern:

```
// global scope
function foo() {}
function local() {
// local scope
function bar() {}
return bar;
}
```

1.4. Thuộc tính name của hàm function

Những thứ khác dùng để xét đến khi ta chọn 1 định nghĩa hàm function chỉ là có sự xuất hiện của thuộc tính chỉ có thể đọc name của hàm function hay không. nhắc lại 1 lần nữa, thuộc tính này không phải là tiêu chuẩn nhưng nó lại có trong rất nhiều môi trường. trong các định nghĩa hàm **function declarations** và biểu thức **named function expressions** thì thuộc tính name được định nghĩa. trong biểu thức **anonymous function expressions** thì nó phụ thuộc vào cách thực thi; nó có thể không được định nghĩa undefined (IE) hoặc được định nghĩa với 1 chuỗi rỗng empty string (Firefox, WebKit):

```
function foo() {} // declaration

var bar = function () {}; // expression
var baz = function baz() {}; // named expression

foo.name; // "foo"
bar.name; // ""
baz.name; // "baz"
```

thuộc tính name là rất hữu dụng trong công cụ debug như firebug. khi debugger cần hiện thị cho ta thấy 1 lỗi trong 1 hàm function thì nó kiểm tra sự có mặt của thuộc tính name và sử

dùng nó như là 1 vật đánh dấu.thuộc tính name cũng có thể được sử dụng để gọi tới cùng hàm function 1 cách đệ quy từ trong chính thân của nó.nếu ta không thấy thú vị ở trong 2 trường hợp ở trên thì biểu thức **anonymous function expressions** sẽ trở nên dễ dàng hơn và ít dài dòng hơn

Lí do mà ta thích biểu thức **function expressions** là vì biểu thức này làm nổi bật các hàm functions cũng là các đối tượng objects và không có sự khởi tạo đặc biệt nào

chú ý : theo 1 cách kĩ thuật thì ta có thể sử dụng 1 biểu thức named function expression và gán nó vào trong 1 biến có tên khác với tên của hàm function như sau :

```
var foo = function bar() {};
```

tuy nhiên, cách thức thực thi này không được thực thi 1 cách đúng đắn tron 1 vài trình duyệt (IE) vì vậy ta không nên sử dụng như vậy

1.5.Sự leo thang của hàm function - Function Hoisting

từ những thảo luận trước đó, ta có thể kết luận rằng cách thức thực thi của các khai báo function là rất tuyệt tương đương với cách dùng 1 biểu thức named function expression.điều này không thực sự chính xác và có 1 sự khác biệt nằm trong sự leo thang hoisting

như ta đã biết, tất cả các biến và không có thứ gì nằm trong hàm function đã được khai báo có thể leo thang lên đầu hàm function .những ứng dụng giống vậy được áp dụng cho hàm bởi vì chúng chỉ là các đối tượng được gán vào trong các biến.

```
// antipattern
// for illustration only
// global functions
function foo() {
  alert('global foo');
}

function bar() {
  alert('global bar');
}

function hoistMe() {
```

```
console.log(typeof foo); // "function"
```

```
console.log(typeof bar); // "undefined"
```

```
foo(); // "local foo"
```

```
bar(); // TypeError: bar is not a function
```

```
// function declaration:
```

```
// variable 'foo' and its implementation both get hoisted
```

```
function foo() {  
  alert('local foo');  
}
```

```
// function expression:
```

```
// only variable 'bar' gets hoisted
```

```
// not the implementation
```

```
var bar = function () {
```

```
  alert('local bar');
```

```
  };
```

```
}
```

```
hoistMe();
```

Sự leo thang, sự leo thang không có tác dụng khi được gán trong 1 biến và ngược lại khi xảy ra sự leo thang thì nó sẽ ghi đè lên giá trị biến toàn cục

Leo thang thất bại

trong ví dụ trên ta thấy rằng, giống với các biến thông thường, thì sự hiện diện của foo và bar ở bất cứ đâu trong hàm hoistMe() function thì đều chuyển chúng lên trên đầu và ghi đè vào foo và bar toàn cục. điểm khác biệt là định nghĩa địa phương của foo() được leo thang lên đầu và hoạt động bình thường còn định nghĩa của bar() không được leo thang và chỉ có định nghĩa của nó. đây là lí do tại sao cho tới khi đoạn mã code thực thi đến định nghĩa của bar() thì nó không xác định và không được sử dụng như 1 hàm function (trong khi nó vẫn cản trở lại bar() toàn cục)

1.6.Kiểu mẫu callback

hàm function là các đối tượng objects, điều này có nghĩa là chúng có thể được chuyển tiếp qua hàm functions khác.khi ta chuyển tiếp hàm function introduceBugs() như là 1 tham số trong hàm writeCode(), sau đó ở vài điểm thì hàm writeCode() có thể thực thi (hay gọi) hàm introduceBugs().trong trường hợp này thì **introduceBugs() được gọi là hàm callback :**

```
function writeCode(callback) {  
  // do something...  
  callback();  
  // ...  
}  
  
function introduceBugs() {  
  // ... make bugs  
}  
  
writeCode(introduceBugs);
```

chú ý tới cách mà hàm introduceBugs() được chuyển tiếp như 1 tham số tới hàm writeCode() mà không có dấu ngoặc.dấu ngoặc thực thi 1 hàm function trái với trường hợp ta muốn chuyển tiếp chỉ 1 tham chiếu tới hàm function và để writeCode() thực thi nó (hay cách khác là gọi lại nó)

1.6.1.Một ví dụ mẫu về callback

Chúng ta cần 1 ví dụ và bắt đầu không với 1 hàm callback đầu tiên và sau đó ta bàn tới nó sau.để hình dung ra, ta có 1 hàm theo mục đích thông thường mà làm 1 vài công việc biên dịch và trả về 1 tập các dữ liệu lớn như 1 kết quả.hàm function thông thường này được gọi là **findNodes()**, và tác vụ của nó là bò trườn lên cây DOM tree của 1 trang page và trả về 1 mảng các phần tử của trang page mà nó sẽ cuốn hút ta :

```
var findNodes = function () {  
  
  var i = 100000, // big, heavy loop  
  nodes = [], // stores the result  
  found; // the next node found  
  while (i) {  
    i -= 1;  
    // complex logic here...
```

```
nodes.push(found);  
}  
return nodes;  
};
```

Đây là 1 ý tưởng hay nhằm giữ hàm function này 1 cách chung nhất và nó đơn giản trả về 1 mảng array của DOM nodes mà không làm bất cứ gì khác với các đối tượng thực tế. sự logic trong việc hiệu chỉnh các nodes có thể trong 1 hàm function khác ví dụ 1 hàm function gọi là **hide()** như những gì tên nó gợi ý thì nó dùng để ẩn các nodes trên page :

```
var hide = function (nodes) {  
  
    var i = 0, max = nodes.length;  
    for (; i < max; i += 1) {  
        nodes[i].style.display = "none";  
    }  
  
};
```

// executing the functions

```
hide(findNodes());
```



Hiệu năng xử lý không cao bởi vì ta phải duyệt 2 lần mảng nodes, tức là thực hiện vòng lặp tận 2 lần

cách thực thi này là không có hiệu năng xử lý tốt bởi vì hide() phải lặp lại 1 lần nữa thông qua các phần tử trong mảng nodes được trả về bởi hàm findNodes(). nó sẽ trở nên hiệu quả hơn nếu ta có thể tránh vòng lặp này và ẩn đi các nodes giống như việc ta chọn ra chúng trong hàm findNodes(). nhưng nếu ta thực thi điều kiện logic để ẩn trong findNodes(), thì nó sẽ không còn là 1 hàm function thông thường bởi vì có sự ghép cặp của điều kiện logic nhận về và điều kiện logic hiệu chỉnh.

Ta áp dụng mẫu callback – ta chuyển tiếp điều kiện logic để ẩn node như 1 hàm callback và ủy thác cho nó thực thi :

```
// refactored findNodes() to accept a callback  
var findNodes = function (callback) {  
    var i = 100000,
```

```
nodes = [],
found;
// check if callback is callable
if (typeof callback !== "function") {
  callback = false;
}
while (i) {
  i -= 1;
  // complex logic here...
  // now callback:
  if (callback) {
    callback(found);
  }
  nodes.push(found);
}
return nodes;
};
```

Hiệu năng xử lý được cải thiện, cả 2 công việc đều được thực hiện thông qua 1 vòng lặp, ta vừa duyệt phần tử trong mảng nodes và đồng thời ngay sau đó ẩn đi các phần tử được duyệt đó

Sự thực thi này là hiển nhiên, chỉ có 1 tác vụ bổ sung là findNodes() thực thi kiểm tra tham số callback có được thêm vào đúng hay không, nếu đúng nó sẽ thực thi hàm callback là không bắt buộc do vậy hàm findNodes() vẫn có thể được sử dụng giống như lúc trước mà không phá vỡ mã code cũ

Thực thi hide() sẽ được đơn giản hóa hơn bởi vì nó không cần tới vòng lặp thông qua các nodes :

```
// a callback function
var hide = function (node) {
  node.style.display = "none";
};
// find the nodes and hide them as you go
findNodes(hide);
```

hàm callback có thể là 1 hàm đã tồn tại như đã chỉ ra trong ví dụ trước hay nó có thể là hàm vô danh. ví dụ, tại đây cách mà ta có thể chỉ ra các nodes bằng cách sử dụng cùng hàm `findNodes()` thông thường :

```
// passing an anonymous callback

findNodes(function (node) {
  node.style.display = "block";
});
```

1.6.2. Hàm callback và phạm vi tồn tại của biến trong hàm callback

Trong ví dụ trước, phần nơi mà hàm callback được thực thi là :

```
callback(parameters);
```

mặc dù điều này khá là đơn giản và sẽ đủ tốt trong nhiều trường hợp, đây thường là những kịch bản nơi mà hàm callback không là 1 hàm vô danh hay là 1 hàm toàn cục, nhưng nó là 1 phương thức của 1 đối tượng object. phương thức callback sử dụng `this` để tham chiếu tới đối tượng mà nó thuộc về do vậy điều này có thể nảy sinh ra cách thức thực thi không mong muốn

ví dụ hàm callback là hàm **paint()** là 1 phương thức của đối tượng có tên là **myapp** :

```
var myapp = { };
myapp.color = "green";
```

```
myapp.paint = function (node) {
  node.style.color = this.color;
};
```

The function `findNodes()` does something like this:

```
var findNodes = function (callback) {
  // ...
  if (typeof callback === "function") {
    callback(found);
  }
}
```

```
// ...  
};
```

Nếu ta gọi **findNodes(myapp.paint)** thì nó sẽ không hoạt động như mong đợi bởi vì

this.color sẽ không được định nghĩa. đối tượng **this** sẽ tham chiếu tới đối tượng toàn cục **global object** bởi vì **findNodes()** là 1 hàm toàn cục. nếu **findNodes()** là 1 phương thức của 1 đối tượng gọi là **dom** (**dom.findNodes()**) thì **this** bên trong hàm callback sẽ tham chiếu tới đối tượng **dom** thay cho đối tượng mà ta muốn là **myapp**.

Để giải quyết vấn đề này thì ta chuyển tiếp hàm callback và bổ sung chuyển tiếp đối tượng object mà hàm callback này thuộc về như sau :

```
findNodes(myapp.paint, myapp);
```

sau đó ta cũng cần hiệu chỉnh lại **findNodes()** cho hàm này liên kết với đối tượng mà ta chuyển tiếp vào :

```
var findNodes = function (callback, callback_obj) {  
  //...  
  if (typeof callback === "function") {  
    callback.call(callback_obj, found);  
  }  
  // ...  
};
```

1 lựa chọn khác cho việc chuyển tiếp 1 đối tượng và 1 phương thức được sử dụng như 1 hàm callback là chuyển tiếp phương thức giống như 1 chuỗi string và do đó ta không phải viết lặp lại đối tượng 2 lần :

```
findNodes(myapp.paint, myapp);
```

chuyển thành :

```
findNodes("paint", myapp);
```

hàm findNodes() cũng phải sửa lại như sau :

```
var findNodes = function (callback, callback_obj) {  
  if (typeof callback === "string") {  
    callback = callback_obj[callback];  
  }  
  //...  
  if (typeof callback === "function") {  
    callback.call(callback_obj, found);  
  }  
  // ...  
};
```

1.6.3.Lắng nghe sự kiện không đồng bộ

Kiểu mẫu hàm callback được sử dụng hàng ngày ví dụ khi ta đính thêm 1 đăng ký sự kiện event listener vào 1 phần tử trên 1 trang page thì thực chất ta cung cấp 1 con trỏ chỉ tới 1 hàm callback mà sẽ được gọi khi sự kiện xảy ra.đây là 1 ví dụ đơn giản về cách mà console.log() được chuyển tiếp như 1 hàm callback khi lắng nghe sự kiện click :

```
document.addEventListener("click", console.log, false);
```

hầu hết cách lập trình hướng client đều theo hướng thực thi theo sự kiện.khi trang page được tải xong thì nó kích hoạt 1 sự kiện load.sau đó người dùng tương tác với trang page và gây ra 1 vài sự kiện khác nhau giống như click, keypress, mouseover, mousemove, ...JS về bản chất là thích hợp với lập trình theo hướng sự kiện bởi vì kiểu mẫu callback – cho phép chương trình của ta làm việc 1 cách không đồng bộ

câu khẩu hiệu nổi tiếng ở Hollywood là “ đừng gọi cho chúng tôi mà chúng tôi sẽ gọi cho bạn “.có 1 điều không thể cho nhóm tuyển chọn là trả lời tất cả các cuộc gọi từ tất cả các ứng viên trong toàn thời gian.JS điều khiển theo hướng sự kiện k đồng bộ và nó tương tự như vậy.chỉ bằng cách thay thế việc đưa ra số điện thoại của ta thì ta cung cấp 1 hàm callback có thể được gọi khi đúng thời điểm.ta cũng có thể cung cấp nhiều hơn các hàm callback nếu cần.bởi vì rất có thể có sự kiện nào đó không bao giờ xảy ra.ví dụ nếu người dùng không bao giờ click vào “Buy now!” thì hàm function của ta mà dùng để kiểm tra tính hợp lệ của thẻ thanh toán sẽ không bao giờ được gọi

1.6.4.Độ trễ Timeouts

1 ví dụ khác cho kiểu mẫu callback là khi ta sử dụng phương thức timeout được cung cấp bởi windowobject: setTimeout() và setInterval().thì các phương thức này cũng nhận và thực thi các hàm callbacks :

```
var thePlotThickens = function () {  
  console.log('500ms later...');  
};  
setTimeout(thePlotThickens, 500);
```

chú ý lại cách mà hàm thePlotThickens được chuyển tiếp như 1 biến mà không có dấu ngoặc bởi vì ta không muốn nó thực thi ngay lập tức nhưng đơn giản chỉ muốn chỉ tới nó sau khi sử dụng setTimeout().chuyển tiếp chuỗi "thePlotThickens()" thay thế cho 1 con trỏ hàm function là 1 cách không kiểu mẫu tương tự với eval().

1.6.5.Hàm callbacks trong các thư viện

Hàm callback là kiểu mẫu đơn giản và rất mạnh mẽ, nó có thể rất dễ để kiểm soát khi ta thiết kế 1 thư viện.mã code có trong thư viện sẽ trở nên khái quát và tái sử dụng lại nhiều nhất có thể và các hàm callbacks trợ giúp việc khái quát hóa này.ta không cần dự đoán và thực thi mọi tính năng mà ta có thể nghĩ ra bởi vì nó sẽ làm thư viện của ta phình to lên và hầu hết người sử dụng sẽ không bao giờ cần 1 lượng lớn các tính năng đó.để thay thế, ta tập trung vào các chức năng chính và cung cấp các “hooks” trong dạng hàm callback mà cho phép các phương thức của thư viện dễ dàng được xây dựng, mở rộng và tùy chỉnh

1.6.6.Cách trả về trong hàm functions

Các hàm functions là các đối tượng objects vì vậy chúng có thể được sử dụng và trả lại các giá trị.điều này có nghĩa là 1 hàm function không cần trả về 1 vài dạng dữ liệu hay 1 mảng dữ liệu như là kết quả của quá trình thực thi của nó.1 hàm function có thể trả về hàm function cụ thể nào đó hay nó có thể tạo ra 1 hàm function khác theo yêu cầu giống như sau :

```
var setup = function () {
```

```
  alert(1);
```

```
  return function () {
```

```
    alert(2);
```

```
  };
```

```
};
```

```
// using the setup function
```

```
var my = setup(); // alerts 1
```

```
my(); // alerts 2
```

Thực hiện gán giá trị được hàm setup() trả về vào biến my. tất nhiên để gán được thì nó phải khởi chạy hàm setup()

Thực hiện giá trị đã được gán ở biến my. giá trị được gán là 1 hàm function

bởi vì setup() trả về 1 hàm function, do đó nó tạo ra 1 sự đóng kín và ta có thể sử dụng sự đóng kín này để lưu trữ 1 vài dữ liệu riêng tư – dữ liệu có khả năng truy cập bởi hàm được trả về nhưng không truy cập được từ bên ngoài. 1 ví dụ ngược lại – mục đích đếm số lần mà mỗi lần ta thực hiện gọi hàm :

```
var setup = function () {
```

```
  var count = 0;
```

```
  return function () {
```

```
    return (count += 1);
```

```
  };
```

```
};
```

```
// usage
```

```
var next = setup();
```

```
next(); // returns 1
```

```
next(); // 2
```

```
next(); // 3
```

Next() là 1 hàm function được trả về và nó vẫn có khả năng thực hiện xử lý liên kết với các biến trong hàm đã tạo ra nó

Biến count vẫn được giữ liên kết. do đó hàm next() mỗi khi thực thi sẽ cộng 1 vào giá trị của biến count

ta có thể hình dung thế này : next ở đây giống như là 1 đối tượng được thực thể hóa từ hàm setup và count giống như 1 thuộc tính trong đối tượng này. mỗi lần gọi next(); chính là ta gọi tới 1 phương thức : phương thức này có tác dụng là cộng 1 vào thuộc tính count

1.6.7. Các hàm functions tự định nghĩa

hàm functions có thể định nghĩa 1 cách linh động và có thể được gán vào các biến. nếu ta tạo ra 1 hàm mới và gán nó vào trong cùng 1 biến – biến này vừa lưu giữ 1 hàm function khác thì tức là ta ghi đè hàm function cũ với hàm mới. theo cách này, ta đang luân chuyển con trỏ chỉ tới hàm function cũ thành chỉ tới hàm function mới. và tất cả điều này có thể xảy ra bên trong thân của hàm function cũ. trong trường hợp này hàm function được ghi đè và được định nghĩa lại chính bản thân nó với 1 cách thực thi hoàn toàn mới. nghe thì có vẻ phức tạp nên ta đi vào xét ví dụ sau :

```
var scareMe = function () {  
  
    alert("Boo!");  
  
    scareMe = function () {  
        alert("Double boo!");  
    };  
};  
  
// using the self-defining function  
scareMe(); // Boo!  
scareMe(); // Double boo!
```

Đây là 1 kiểu mẫu rất hữu dụng khi hàm function có 1 vài công việc chuẩn bị cho việc khởi tạo và nó cần làm duy nhất chỉ 1 lần. bởi vì không có lí do gì để làm lặp lại công việc khi nó có thể bị hủy bỏ thì 1 phần của hàm function có thể không còn bắt buộc. trong nhiều trường hợp như vậy thì hàm tự định nghĩa có thể tự cập nhật chính thực thi của nó

Cách sử dụng kiểu mẫu này có thể rõ ràng trợ giúp cải thiện hiệu năng của ứng dụng bởi vì ta định nghĩa lại hàm function đơn giản làm làm ít việc hơn

Chú ý : tên gọi khác cho kiểu mẫu này là “lazy function definition,” bởi vì hàm

function không định nghĩa 1 cách đúng đắn cho tới thời điểm đầu tiên nó được sử dụng và nó trở nên lười biếng và làm ít việc hơn

1 hạn chế của kiểu mẫu này là bất kì thuộc tính nào mà ta thêm vào trước đó trong hàm function gốc thì sẽ bị mất khi nó tự định nghĩa lại. nếu hàm function được sử dụng với các tên khác nhau ví dụ được gán vào 1 biến khác hay được sử dụng như 1 phương thức của 1 đối tượng thì sau đó phần tự định nghĩa lại sẽ không bao giờ xảy ra và thân hàm function gốc sẽ được thực thi

Ta xét 1 ví dụ khi hàm **scareMe()** được sử dụng trong cách mà 1 đối tượng đầu tiên được sử dụng :

1. 1 thuộc tính mới được thêm vào
2. Đối tượng hàm function được gán vào trong 1 biến mới
3. Hàm function cũng được sử dụng như là 1 phương thức

// 1. Thêm vào 1 thuộc tính mới

scareMe.property = "properly";

// 2. Gán tên khác cho đối tượng

var prank = scareMe;

// 3. Sử dụng nó như 1 phương thức trong 1 đối tượng khác

var spooky = {

boo: scareMe

};

// gọi nó với 1 tên mới – vừa gán ở bước 2

prank(); // "Boo!"

console.log(**prank.property**); // "properly"

// gọi nó như là 1 phương thức

spooky.boo(); // "Boo!"

console.log(**spooky.boo.property**); // "properly"

// sau lần gọi thứ nhất, ở lần gọi thứ 2 thì hàm function gốc đã được định nghĩa lại

Khi gọi **prank()**; thì hàm scareMe() toàn cục phải bị ghi đè. tuy nhiên **prank()**; lại giữ lại các mã thực thi của hàm scareMe() gốc. đối với cách gọi phương thức **spooky.boo()**; cũng tương tự như vậy

```
scareMe(); // Double boo!
```

//thuộc tính mới thêm vào đã biến mất

```
console.log(scareMe.property); // undefined
```

Sự ghi đè đã diễn ra từ trước đó

như ta nhìn thấy, sự tự định nghĩa không xảy ra theo như ta mong muốn khi hàm được gán vào trong 1 biến mới. mỗi lần `prank()` được gọi thì nó hiện ra thông báo “Boo!”. vào cùng thời điểm đó nó đã **ghi đè lên hàm `scareMe()` toàn cục** nhưng `prank()` chính bản thân nó giữ lại định nghĩa cũ bao gồm cả thuộc tính `property`. điều tượng tự xảy ra khi hàm được sử dụng như là phương thức `boo()` trong đối tượng `spooky`. tất cả lời gọi đều ghi lại con trỏ hàm toàn cục `scareMe()` do vậy khi cuối cùng nó được gọi thì nó được cập nhật lại định nghĩa và hiện ra thông báo “Double boo” và nó cũng không còn thuộc tính `scareMe.property`

1.6.8. Các hàm functions tức thời

kiểu mẫu hàm function tức thời là 1 cú pháp cho phép ta thực thi 1 hàm function giống như nó đã được định nghĩa. ví dụ:

```
(function () {  
    alert('watch out!');  
})();
```

Kiểu mẫu này về bản chất chỉ là 1 thực thi hàm function (giống như hàm function được đặt tên hoặc vô danh). nó dùng được thực thi ngay tức khắc ngay sau khi nó được tạo ra. cụm từ hàm function tức thời - immediate function không được định nghĩa trong chuẩn ECMA Script nhưng nó ngắn gọn và giúp miêu tả kiểu mẫu đang được nhắc tới

Kiểu mẫu này bao gồm các phần sau :

- Ta định nghĩa 1 hàm function bằng cách sử dụng 1 biểu thức function expression
- Ta thêm 1 cặp dấu ngoặc `()` vào phần cuối – chính điều này sẽ làm cho hàm function được thực thi ngay tức khắc

- Ta đóng gói toàn bộ hàm function vào trong 1 cặp dấu ngoặc **(...)** (điều này là bắt buộc nếu ta không gán hàm function vào 1 biến)

Cũng có 1 cú pháp khác thường được dùng (chú ý tới điểm đặt dấu đóng ngoặc đơn) nhưng trong JSLint thì nó thích cái cú pháp đầu tiên hơn :

```
(function () {  
    alert('watch out!');  
})();
```

Kiểu mẫu này là hữu dụng bởi vì nó cung cấp 1 phạm vi hộp cát sandbox cho việc khởi tạo mã code. suy nghĩ về 1 kịch bản sau : mã code phải thực thi 1 vài tác vụ thiết lập khi trang page load giống như đính thêm các hàm tùy chỉnh event handlers, tạo ra các đối tượng objects,...tất cả công việc này cần được thực hiện chỉ 1 lần do đó không có lí do gì để ta phải tạo ra 1 hàm named function có khả năng tái sử dụng.nhưng mã code cũng đòi hỏi 1 vài biến tạm – những biến mà ta sẽ không cần sau khi trạng thái khởi tạo hoàn tất.ý tưởng tồi nếu tạo ra tất cả các biến này là các biến toàn cục.đây là lí do tại sao ta cần 1 hàm function tức thời – để đóng gói mã code vào bên trong phạm vi của nó và không làm rò rỉ ra bất cứ biến nào trong phạm vi toàn cục :

```
(function () {  
    var days = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'],  
        today = new Date(),  
        msg = 'Today is ' + days[today.getDay()] + ', ' + today.getDate();  
    alert(msg);  
})(); // "Today is Fri, 13"
```

Nếu mã code này không được đóng gói bên trong 1 hàm tức thời thì các biến days,today, và msg sẽ là các biến toàn cục còn thừa lại sau khi mã code được khởi tạo

1.6.8.1.Các tham số của 1 hàm tức thời

Ta cũng có thể chuyển tiếp các tham số vào các hàm tức thời như ví dụ sau :

```
// prints:
```

```
// I met Joe Black on Fri Aug 13 2010 23:26:59 GMT-0800 (PST)
(function (who, when) {
  console.log("I met " + who + " on " + when);
}("Joe Black", new Date()));
```

Theo thông thường thì đối tượng toàn cục global object được chuyển tiếp như 1 tham số tới hàm tức thời do đó nó có khả năng truy cập ở bên trong hàm function mà không phải sử dụng window. cách này khiến mã code tương thích hơn trong nhiều môi trường ngoài trình duyệt :

```
(function (global) {
  // access the global object via `global`
})(this);
```

Chú ý rằng thông thường ta không nên chuyển tiếp quá nhiều tham số vào 1 hàm tức thời bởi vì nó có thể nhanh chóng trở thành 1 gánh nặng cho người lập trình phải kéo từ đầu cho tới cuối hàm function để hiểu nó hoạt động như thế nào

1.6.8.2. Các giá trị được trả về từ các hàm tức thời

Giống như các hàm function khác thì 1 hàm tức thời có thể trả về các giá trị và những giá trị trả về này có thể được gán vào trong các biến :

```
var result = (function () {
  return 2 + 2;
})();
```

Cách khác để thu được giống như trên là bỏ qua cặp dấu ngoặc **()** dùng để đóng gói xung quanh hàm function bởi vì nó không là bắt buộc khi ta gán giá trị được trả về trong 1 hàm tức thời vào trong 1 biến. bỏ qua cặp dấu ngoặc như sau :

```
var result = function () {
  return 2 + 2;
}();
```

Cú pháp này đơn giản hơn nhưng nó có thể trông như 1 chút nhầm lẫn. việc thiếu dấu **()** ở cuối hàm function, thì 1 vài người đọc mã code có thể hiểu là result chỉ tới 1 hàm function. thực tế thì result chỉ tới giá trị được trả về bởi hàm tức thời, trong trường hợp này nó là 4

1 cách khác cũng tương tự :

```
var result = (function () {  
    return 2 + 2;  
})();
```

Các ví dụ trước trả về 1 giá trị nguyên cơ bản. nhưng thay cho việc trả về giá trị cơ bản thì 1 hàm tức thời có thể trả về bất kì dạng giá trị nào bao gồm cả 1 hàm function khác. ta có thể sử dụng phạm vi của hàm tức thời để lưu trữ 1 cách riêng tư 1 vài dữ liệu

Trong ví dụ tiếp theo, giá trị được trả về bởi hàm tức thời là 1 hàm function mà nó sẽ được gán vào trong biến getResult và sẽ đơn giản trả về giá trị trong res – 1 giá trị sẽ được tính toán trước và lưu trữ trong sự bao đóng của hàm tức thời :

```
var getResult = (function () {  
    var res = 2 + 2;  
    return function () {  
        return res;  
    };  
})();
```

Các hàm functions tức thời cũng có thể được sử dụng khi ta định nghĩa các thuộc tính của đối tượng object. để hình dung ta cần định nghĩa 1 thuộc tính – thuộc tính này sẽ không bao giờ thay đổi trong suốt vòng đời của đối tượng object nhưng trước khi ta định nghĩa nó thì 1 số việc cần phải được thực thi để tìm ra giá trị đúng. ta có thể sử dụng 1 hàm tức thời để đóng gói công việc đó và giá trị được trả về của hàm tức thời sẽ trở thành giá trị của thuộc tính :

```
var o = {
```



```
message: (function () {
```

```
var who = "me",
```

```
what = "call";
```

```
return what + " " + who;
```

```
})();
```

```
getMsg: function () {
```

```
return this.message;
```

```
}
```

```
};
```

```
// usage
```

```
o.getMsg(); // "call me"
```

```
o.message; // "call me"
```

trong ví dụ trên thì `o.message` là 1 thuộc tính kiểu chuỗi string chứ không phải là 1 hàm function nhưng nó cần tới 1 hàm function để thực thi trong suốt quá trình mã script đang được load và trợ giúp định nghĩa thuộc tính

1.6.8.3.Cách sử dụng và ưu điểm

kiểu mẫu hàm tức thời được sử dụng 1 cách rộng rãi. nó trợ giúp ta đóng gói 1 lượng các công việc mà ta muốn thực hiện mà không phải giữ lại bất cứ biến toàn cục nào. tất cả các biến mà ta định nghĩa sẽ là địa phương và ta không phải lo lắng gì về việc làm dư thừa không gian toàn cục với các biến tạm

chú ý : tên gọi khác của hàm tức thời là hàm tự gọi hay hàm tự thực thi “self-invoking” or “self-executing” function

kiểu mẫu này thường được sử dụng trong bookmarklets bởi vì bookmarklets chạy trên bất cứ trang page nào

kiểu mẫu này cũng cho phép ta đóng gói các tính năng tách riêng vào trong các self-contained modules (các modules khép kín).

1.6.9.Cách khởi tạo các đối tượng tức thời

cách khác để bảo vệ phạm vi toàn cục tương tự với hàm tức thời chính là kiểu mẫu khởi tạo các đối tượng tức thời. kiểu mẫu này sử dụng 1 đối tượng với 1 phương thức `init()` – sẽ được thực thi ngay sau khi đối tượng được tạo ra. hàm `init()` sẽ chăm lo tới toàn bộ tác vụ khởi tạo.

đây là 1 ví dụ về kiểu mẫu khởi tạo các đối tượng tức thời

```
( {  
  
  // here you can define setting values  
  // a.k.a. configuration constants  
  maxwidth: 600,  
  maxheight: 400,  
  
  // you can also define utility methods  
  gimmeMax: function () {  
    return this.maxwidth + "x" + this.maxheight;  
  },  
  
  // initialize  
  init: function () {  
    console.log(this.gimmeMax());  
    // more init tasks...  
  }  
}).init();
```

Ta tạo ra 1 đối tượng object theo cách thông thường và cũng đóng gói nó vào 1 cặp ngoặc đơn `()`. sau khi ta đóng cặp ngoặc đơn thì ta gọi ngay phương thức `init()`

Ta cũng có thể đóng gói đối tượng và lời gọi `init()` theo cách sau :

```
( {... }).init();
```

```
( {... }.init());
```

Ưu điểm của kiểu mẫu này cũng giống với kiểu mẫu hàm tức thời là ta bảo vệ namespace toàn cục trong khi thực thi các tác vụ khởi tạo

1 nhược điểm của kiểu mẫu này là đa số các tối thiểu mã JS không thể tối thiểu hóa kiểu mẫu này 1 cách có hiệu quả giống như hàm function. các thuộc tính và phương thức riêng tư sẽ không thể đổi tên ngắn hơn bởi vì từ quan điểm của 1 trình tối thiểu hóa nó không làm được việc đó 1 cách an toàn

1.6.10. Phân nhánh thời gian khởi tạo - Init-Time Branching

Init-time branching (hay còn được gọi là load-time branching) là 1 kiểu mẫu tối ưu hóa. ta đã biết rằng 1 điều kiện nào đó sẽ không thay đổi trong suốt vòng đời của chương trình thì nó sẽ tạo ra khả năng kiểm tra điều kiện này duy nhất 1 lần.

Ví dụ sau khi ta nhận biết được XMLHttpRequest được hỗ trợ như là 1 native object, thì không có cơ hội nào nằm dưới trình duyệt sẽ thay đổi ở phần giữa của thực thi chương trình và tất cả của sự bất thành linh này ta sẽ cần làm việc với các đối tượng ActiveX objects. bởi vì môi trường không thay đổi nên không có lí do gì để giữ sự đánh hơi vào mỗi lần ta cần đối tượng XHR object khác

Để hình dung ra các dạng được tính toán của 1 phần tử DOM hay đính thêm hàm tùy chỉnh sự kiện event handlers là các ứng viên khác mà có thể có lợi từ kiểu mẫu init-time branching pattern. đa số nhà lập trình đã từng code – ít nhất 1 lần trong đời lập trình hướng client – 1 tiện ích với các phương thức dùng cho việc đính thêm và xóa bỏ các event listeners như sau :

```
// BEFORE
```

```
var utils = {  
  
  addListener: function (el, type, fn) {  
    if (typeof window.addEventListener === 'function') {  
      el.addEventListener(type, fn, false);  
    } else if (typeof document.attachEvent === 'function') { // IE  
      el.attachEvent('on' + type, fn);  
    } else { // older browsers  
      el['on' + type] = fn;  
    }  
  },  
}
```

```
removeListener: function (el, type, fn) {  
  // pretty much the same...  
}  
  
};
```

Vấn đề với đoạn code này là nó không mang lại hiệu quả cao. mỗi lần ta gọi tới `utils.addListener()` hay `utils.removeListener()` thì các kiểm tra giống nhau sẽ thực thi hơn và hơn nữa

Cách sử dụng init-time branching, thì ta đánh hơi các tính năng của trình duyệt trong suốt quá trình khởi tạo của mã script. vào thời điểm ta định nghĩa lại cách mà hàm function hoạt động trong vòng đời của trang page. ví dụ sau là cách ta tiếp cận tác vụ này :

```
// AFTER  
// the interface  
  
var utils = {  
  
  addListener: null,  
  removeListener: null  
  
};  
  
// the implementation  
if (typeof window.addEventListener === 'function') {  
  utils.addListener = function (el, type, fn) {  
    el.addEventListener(type, fn, false);  
  };  
  utils.removeListener = function (el, type, fn) {  
    el.removeEventListener(type, fn, false);  
  };  
} else if (typeof document.attachEvent === 'function') { // IE  
  utils.addListener = function (el, type, fn) {  
    el.attachEvent('on' + type, fn);  
  };  
  utils.removeListener = function (el, type, fn) {
```

```
el.detachEvent('on' + type, fn);  
};  
} else { // older browsers  
utils.addListener = function (el, type, fn) {  
  el['on' + type] = fn;  
};  
utils.removeListener = function (el, type, fn) {  
  el['on' + type] = null;  
};  
}
```

Nếu ta nhận biết rằng trình duyệt không hỗ trợ `window.addEventListener`, thì không cần giả định rằng trình duyệt mà ta đang làm việc với là IE và nó không hỗ trợ XMLHttpRequest. mặc dù điều này đúng tại 1 vài thời điểm trong lịch sử của trình duyệt. có thể có vài trường hợp mà ta giả định 1 cách an toàn rằng các tính năng này sẽ có cùng nhau giống như `.addEventListener` và `.removeEventListener` nhưng thông thường các tính năng của trình duyệt thay đổi 1 cách độc lập. chiến lược tốt nhất cho việc nhận biết các tính năng 1 cách riêng biệt và sau đó sử dụng load-time branching để làm công việc nhận biết này chỉ duy nhất 1 lần

1.6.11. Các thuộc tính của functions – kiểu mẫu tối ưu hóa bộ nhớ

Memoization Pattern

Các functions là các đối tượng objects vì vậy chúng có các thuộc tính của riêng chúng. thực tế chúng phải có các thuộc tính và các phương thức. ví dụ mọi hàm function thì không có vấn đề gì về cú pháp mà ta sử dụng để tạo ra chúng, 1 cách tự động có thể lấy về thuộc tính `length` dùng để chỉ ra số tham số của hàm function như sau :

```
function func(a, b, c) {}  
console.log(func.length); // 3
```

ta có thể thêm vào các thuộc tính tùy chỉnh tới hàm function vào bất cứ lúc nào. một trường hợp sử dụng cho các thuộc tính tùy chỉnh là cache các kết quả (giá trị được trả về) của 1 hàm function vì vậy và thời điểm sau đó khi hàm function được gọi thì nó không phải thực

hiện lại các tính toán nặng nhọc. việc caching các kết quả của 1 hàm function cũng được biết như là 1 cách tối ưu hóa bộ nhớ

trong ví dụ sau, hàm myFunc tạo ra 1 thuộc tính cache có thể truy cập thông qua myFunc.cache. thuộc tính cache là 1 đối tượng (1 hash) với các tham số param được chuyển tiếp vào hàm function này được sử dụng như 1 key và kết quả của sự tính toán như là value. kết quả có thể là bất cứ cấu trúc dữ liệu phức tạp nào mà ta cần :

```
var myFunc = function (param) {  
  
  if (!myFunc.cache[param]) {  
  
    var result = {};  
    // ... expensive operation ...  
    myFunc.cache[param] = result;  
  }  
  
  return myFunc.cache[param];  
  
};  
  
// cache storage  
myFunc.cache = {};
```

đoạn mã code trên giả định rằng hàm function chỉ cần duy nhất 1 tham số là param và nó là 1 kiểu dữ liệu cơ bản (giống như là 1 chuỗi string). nếu ta có nhiều hơn các tham số và nhiều kiểu dữ liệu phức tạp hơn thì cách giải quyết thông thường sẽ là nối tiếp hóa chúng. ví dụ ta có thể nối tiếp hóa đối tượng tham số như 1 chuỗi JSON và sử dụng chuỗi này như là 1 key trong đối tượng cache object :

```
var myFunc = function () {  
  
  var cachekey = JSON.stringify(Array.prototype.slice.call(arguments)),  
  result;  
  
  if (!myFunc.cache[cachekey]) {  
  
    result = {};  
    // ... expensive operation ...  
    myFunc.cache[cachekey] = result;  
  }  
  
}
```

```
return myFunc.cache[cachekey];
};
// cache storage
myFunc.cache = {};
```

chú ý rằng trong nối tiếp hóa ở đây, thì sự nhận biết của các đối tượng objects đã mất. nếu ta có 2 đối tượng khác nhau mà có cùng các thuộc tính thì cả 2 sẽ chia sẻ cùng thực thể cache

cách khác để viết hàm function phía trên là sử dụng **arguments.callee** để tham chiếu tới hàm function thay thế cho việc đặt mã code cứng tên của hàm function. mặc dù hiện tại thì có thể nhưng chú ý rằng arguments.callee không được cho phép ở chế độ nghiêm ngặt của ECMAScript 5:

```
var myFunc = function (param) {
var f = arguments.callee,
result;
if (!f.cache[param]) {
result = {};
// ... expensive operation ...
f.cache[param] = result;
}
return f.cache[param];
};
// cache storage
myFunc.cache = {};
```

1.6.12. Các đối tượng cấu hình - Configuration Objects

kiểu mẫu đối tượng cấu hình là 1 cách cung cấp các APIs rõ ràng hơn, về bản chất nếu ta đang xây dựng 1 thư viện hay bất cứ mã code nào khác mà sẽ được dùng bởi các chương trình khác

đó là thực tế cuộc sống mà các phần mềm đòi hỏi những thay đổi cũng giống như việc phần mềm được phát triển và được bảo trì. điều này thường xuyên xảy ra khi ta bắt đầu làm việc với 1 vài yêu cầu trong suy nghĩ nhưng sẽ có nhiều chức năng hơn được thêm vào sau đó

để hình dung ta đang viết 1 hàm được gọi là `addPerson()`, hàm này nhận 2 tham số `first` và `last name` dùng để thêm 1 người vào 1 danh sách :

```
function addPerson(first, last) { ... }
```

sau đó ta tìm hiểu được thêm rằng cần thực sự thêm ngày sinh nhật , giới tính và địa chỉ vào. vì vậy ta hiệu chỉnh lại hàm function bằng cách thêm vào các tham số mới và cẩn thận không làm xáo trộn trật tự của các tham số :

```
addPerson("Bruce", "Wayne", new Date(), null, null, "batman");
```

cách chuyển tiếp 1 lượng lớn các tham số không được tiện cho lắm. và cách tiếp cận tốt hơn là thay thế tất cả các tham thành duy nhất chỉ một đối tượng object. ta gọi đối tượng này là `conf` :

```
addPerson(conf);
```

ta biến đổi như sau :

```
var conf = {  
  username: "batman",  
  first: "Bruce",  
  last: "Wayne"  
};  
addPerson(conf);
```

thể mạnh của các đối tượng cấu hình là :

- Không cần phải nhớ các tham số và thứ tự của chúng
- Ta có thể bỏ qua 1 cách an toàn các tham số không bắt buộc
- Dễ dàng để đọc và sửa chữa
- Dễ dàng để thêm và xóa các tham số

Hạn chế của các đối tượng cấu hình là :

- Ta cần nhớ tên các tham số

- Các thuộc tính names có thể bị tối thiểu hóa

Kiểu mẫu này có thể hữu dụng khi hàm function của ta tạo ra các phần tử DOM, ví dụ trong các thiết lập CSS styles của 1 phần tử bởi vì các phần tử và styles có thể có 1 số lượng lớn các thuộc tính và đặc tính không bắt buộc

1.6.13.Curry

1.6.14.Function Application

Trong 1 vài ngôn ngữ lập trình theo hướng functional 1 cách tuyệt đối thì 1 hàm function không được mô tả là được gọi called hay invoked, mà là áp dụng applied. trong JS thì ta cũng có thứ giống hệt như vậy – ta có thể áp dụng 1 hàm function bằng cách sử dụng phương thức `Function.prototype.apply()` bởi vì các hàm functions trong JS thực chất là các đối tượng objects do vậy chúng cũng có các phương thức

```
// define a function
var sayHi = function (who) {
  return "Hello" + (who ? ", " + who : "") + "!";
};

// invoke a function
sayHi(); // "Hello"
sayHi('world'); // "Hello, world!"

// apply a function
sayHi.apply(null, ["hello"]); // "Hello, hello!"
```

như ta có thể thấy trong ví dụ, cả việc gọi 1 hàm function và việc áp dụng nó đều có cùng 1 kết quả. `apply()` cần 2 tham số : cái đầu tiên là 1 đối tượng dùng liên kết với this nằm bên trong hàm function và thứ 2 là 1 mảng array các tham số. tham số đầu tiên là null thì this chỉ tới đối tượng toàn cục global object – đây chính xác là những gì xảy ra khi ta gọi 1 hàm function mà nó không phải là phương thức của 1 đối tượng nào đó

khi 1 hàm function là 1 phương thức của 1 đối tượng thì không có null tham chiếu được chuyển tiếp ra xung quanh, tại đây đối tượng trở thành tham số đầu tiên trong `apply()`:

```
var alien = {
  sayHi: function (who) {
```

```
return "Hello" + (who ? ", " + who : "") + "!";  
}  
};  
alien.sayHi('world'); // "Hello, world!"  
sayHi.apply(alien, ["humans"]); // "Hello, humans!"
```

trong đoạn ví dụ trên, thì this ở trong sayHi() chỉ tới alien.

Chú ý bổ sung thêm cho apply() thì có 1 phương thức là call() của đối tượng Function.prototype object, chức năng của nó vẫn giống với apply().khi ta có 1 hàm function mà chỉ cần duy nhất 1 tham số thì ta có thể tiết kiệm việc tạo ra 1 mảng array chỉ với 1 phần tử :

```
// the second is more efficient, saves an array  
sayHi.apply(alien, ["humans"]); // "Hello, humans!"  
sayHi.call(alien, "humans"); // "Hello, humans!"
```

1.6.15.Partial Application

Giờ ta biết rằng việc gọi 1 hàm function thực chất là áp dụng 1 các tham số vào 1 hàm, thì nó có thể chuyển tiếp chỉ 1 vài tham số hay không phải tất cả trong số chúng ?đây thực sự giống với các mà ta thường làm việc này, nếu ta đang làm việc với 1 hàm math function

Nói ta có 1 hàm function add() được dùng để cộng 2 số vào với nhau : x và y.đoạn mã sau chỉ cho ta cách mà ta có thể tiếp cận 1 giải pháp đã cho x = 5 và y = 4 :

```
// for illustration purposes  
// not valid JavaScript  
// we have this function  
function add(x, y) {  
  return x + y;  
}  
// and we know the arguments  
add(5, 4);  
// step 1 -- substitute one argument  
function add(5, y) {
```

```
return 5 + y;
}
// step 2 -- substitute the other argument
function add(5, 4) {
return 5 + 4;
}
```

Trong đũa mã trên có 2 bước steps 1 và 2 đều không đúng trong JS nhưng đây là cách mà ta giải quyết vấn đề này bằng tay. ta lấy giá trị trong tham số đầu tiên và thay thế x chưa biết thành 1 giá trị đã biết là 5 thông qua hàm function. sau đó lặp lại như vậy cho tới khi ta đảo lại tham số

Step 1 trong ví dụ này có thể được gọi là partial application: ta chỉ áp dụng tham số đầu tiên. khi ta thực thi 1 partial application thì ta không lấy về được kết quả nhưng ta lấy về được hàm thay thế

Đoạn mã tiếp theo minh họa cách sử dụng 1 phương thức ảo partialApply() :

```
var add = function (x, y) {
return x + y;
};
// full application
add.apply(null, [5, 4]); // 9
// partial application
var newadd = add.partialApply(null, [5]);
// applying an argument to the new function
newadd.apply(null, [4]); // 9
```

như ta thấy, partial application đưa cho ta hàm function khác – cái mà sau đó có thể được gọi với các tham số khác. điều này thực sự tương đương với 1 vài thứ như add(5)(4) bởi vì add(5) trả về 1 hàm function mà có thể được gọi với (4).

Giờ quay trở lại thực tại : không có phương thức partialApply() nào trong JS. trong JS không có cách thực thi nào như vậy nhưng ta có thể tạo ra chúng bởi vì JS rất linh hoạt

1.6.16.Currying

Currying không có liên quan gì tới món cà ri ẩn độ cả, nó đến từ tên của 1 nhà toán học tên là Haskell Curry.(ngôn ngữ lập trình Haskell cũng được đặt tên theo tên của ông). Currying là 1 tiến trình biến đổi – ta biến đổi 1 hàm function.1 tên gọi khác cho currying có thể là schönfinkelisation

Vậy cách mà ta biến đổi 1 hàm function như thế nào ?theo ngôn ngữ lập trình theo hướng functional thì có thể xây dựng ngay vào bên trong ngôn ngữ đó và tất cả các hàm functions đều được biến đổi theo mặc định.trong JS thì ta có thể hiệu chỉnh hàm add() vào trong 1 hàm đã curried

Xét ví dụ :

```
// a curried add()
// accepts partial list of arguments

function add(x, y) {

  var oldx = x, oldy = y;

  if (typeof oldy === "undefined") { // partial

    return function (newy) {

      return oldx + newy;

    };

  }

  // full application
  return x + y;

}

// test

typeof add(5); // "function"
```

Tác dụng của hàm function này là áp dụng tham số vào từng phần lần lượt vào hàm function có 2 tham số mà không gây ra lỗi.thay vì phải luôn luôn gọi hàm là add(3,4) thì ta có thể gọi lần lượt như sau mà không gây lỗi : add(3) và sau đó là add(3)(4)

```
add(3)(4); // 7
```

```
// create and store a new function
```

```
var add2000 = add(2000);
```

```
add2000(10); // 2010
```

Tương đương với hàm sau :

```
Function(newy){  
  Return 2000 + newy;  
}
```

trong đoạn mã trên thì đầu tiên ta gọi add(0) thì nó tạo ra 1 bao đóng xung quanh hàm function bên trong được nó trả về. bao đóng này lưu trữ các giá trị gốc của x và y vào bên trong 2 biến riêng tư oldx và oldy. biến đầu tiên oldx được sử dụng khi hàm bên trong được thực thi. sẽ không có partial application khi cả x và y được chuyển tiếp và hàm function sẽ được thực thi 1 cách đơn giản là cộng 2 số này vào. thực thi của add() sẽ trở nên dài dòng hơn mức cần thiết. 1 phiên bản gọn gàng hơn được chỉ ra trong ví dụ tiếp theo, nó sẽ không có oldx và oldy đơn giản bởi vì x gốc được lưu giữ trong 1 bao đóng hoàn toàn và ta sử dụng lại y như là biến địa phương thay cho việc phải tạo ra 1 biến mới là newy như ta đã làm trong ví dụ trước :

```
// a curried add
```

```
// accepts partial list of arguments
```

```
function add(x, y) {
```

```
  if (typeof y === "undefined") { // partial
```

```
    return function (y) {
```

```
      return x + y;
```

```
    };
```

```
  }
```

```
// full application
```

```
  return x + y;
```

```
}
```

Từ những ví dụ trên ta có thể biến đổi bất cứ 1 hàm function nào thành 1 hàm mới chấp nhận các tham số từng phần? ví dụ tiếp theo chỉ ra 1 mẫu của 1 hàm function có mục đích thông thường, ta gọi nó là schonfinkelize() :

```
function schonfinkelize(fn) {
```

```
  var slice = Array.prototype.slice,
```

```
stored_args = slice.call(arguments, 1);
return function () {
  var new_args = slice.call(arguments),
  args = stored_args.concat(new_args);
  return fn.apply(null, args);
};
}
```

Hàm `schonfinkelize()` chắc chắn phức tạp hơn những gì nó nên là thế nhưng chỉ bởi vì `arguments` không phải là 1 mảng array thực sự trong JS. việc vay mượn phương thức method từ `Array.prototype` trợ giúp cho ta biến `arguments` vào trong 1 mảng array và làm việc với nó 1 cách tiện lợi hơn. khi `schonfinkelize()` được gọi vào lần đầu tiên thì nó lưu trữ 1 tham chiếu riêng tư tới phương thức `slice()` (được gọi là `slice`) và cũng lưu trữ các tham số này trong `stored_args`, chỉ cắt tham số đầu tiên bởi vì tham số đầu tiên là hàm function được `curried`. sau đó `schonfinkelize()` trả về 1 hàm mới. khi hàm mới được gọi thì nó có truy cập (thông qua bao đóng) tới các tham số được lưu trữ 1 cách riêng tư `stored_args` và tham chiếu `slice`. hàm function mới chỉ nối với các tham số được áp dụng riêng phần cũ (`stored_args`) với tham số mới (`new_args`) và sau đó áp dụng chúng vào hàm gốc `fn`.

Giờ ta thử kiểm tra 1 vài mẫu test :

```
// a normal function
function add(x, y) {
  return x + y;
}

// curry a function to get a new function
var newadd = schonfinkelize(add, 5);
newadd(4); // 9

// another option -- call the new function directly
schonfinkelize(add, 6)(7); // 13
```

hàm biến đổi `schonfinkelize()` không bị giới hạn bởi các tham số đơn hay từng bước `currying`. đây là 1 ví dụ thường dùng hơn :

```
// a normal function
```

```
function add(a, b, c, d, e) {  
  return a + b + c + d + e;  
}  
  
// works with any number of arguments  
schonfinkelize(add, 1, 2, 3)(5, 5); // 16  
  
// two-step currying  
var addOne = schonfinkelize(add, 1);  
addOne(10, 10, 10, 10); // 41  
var addSix = schonfinkelize(addOne, 2, 3);  
addSix(5, 5); // 16
```

1.6.17.khi nào ta sử dụng Currying

khi ta tìm kiếm lời gọi chính hàm function và chuyển tiếp hầu hết cùng các tham số sau đó hàm function hầu như chắc chắn là 1 ứng cử viên tốt cho việc currying.ta có thể tạo ra 1 hàm mới 1 cách linh hoạt bằng cách áp dụng từng phần 1 tập các tham số vào hàm function.hàm function mới sẽ giữ các tham số được lặp lại (do đó ta không phải chuyển tiếp thêm chúng vào mỗi lần gọi tiếp theo) và ta sẽ sử dụng chúng để điền trước vào danh sách đầy đủ các tham số mà hàm function gốc mong muốn

II. Tủ mảng arrays cho tới các đối tượng objects

1 mảng array là danh sách các giá trị value. mỗi giá trị value có 1 chỉ mục indexn (1 chỉ mục key dạng số) bắt đầu từ 0 và tăng dần lên 1 tương ứng với mỗi phần tử tiếp theo

```
>>>> var myarr = ['red', 'blue', 'yellow', 'purple'];
>>> myarr;
["red", "blue", "yellow", "purple"]
>>> myarr[0]
"red"
>>> myarr[3]
"purple"
```

Nếu ta đặt các chỉ mục index vào 1 cột và các giá trị tương ứng của nó sang cột còn lại thì ta sẽ được bảng các cặp key/value như sau :

Key	Value
0	red
1	blue
2	yellow
3	purple

1 đối tượng cũng rất giống với 1 mảng array nhưng có điểm khác là ta định nghĩa các chỉ mục key tùy ý theo riêng mình (tức nó không bắt buộc là kiểu số nữa)

Xét ví dụ đơn giản về 1 đối tượng sau :

```
var hero = {
  breed: 'Turtle',
  occupation: 'Ninja'
};
```

Ta có thể thấy rằng :

- Tên của biến bao gồm 1 đối tượng là hero
- Thay thế cho dấu ngoặc vuông `[]` mà ta định nghĩa trong mảng array thì ta sử dụng dấu móc đơn `{ }` cho đối tượng
- Ta phân chia các phần tử (hay đúng hơn gọi là các thuộc tính) được chứa trong đối tượng với dấu phẩy “,”
- Các cặp key/value được phân tách nhau bởi dấu 2 chấm “:”

Các chỉ mục keys (hay là tên các thuộc tính) có thể không bị bắt buộc bởi cặp dấu nháy đơn hoặc kép. như ví dụ sau , các khai báo này đều là 1 :

```
var o = {prop: 1};
```

```
var o = { "prop" : 1};
```

```
var o = { 'prop' : 1 }
```

chú ý ta không nên đặt dấu nháy đơn hoặc kép vào các chỉ mục keys (hay là tên các thuộc tính) vì lí do tiếp kiệm việc phải gõ code , nhưng có 1 vài trường hợp ta bắt buộc phải sử dụng cặp dấu nháy này :

- Nếu tên thuộc tính là 1 trong các từ được dùng riêng cho javascript
- Nếu nó chứa dấu cách hay các kí tự đặc biệt (hay bất cứ thứ gì khác không phải thuộc danh kí tự, số và dấu gạch dưới)
- Nếu nó bắt đầu với 1 số

Về cơ bản, nếu tên mà ta chọn cho 1 thuộc tính không phải là 1 tên đúng chuẩn giống như tên biến trong JS thì ta cần đặt nó vào trong cặp dấu nháy

1 ví dụ về đối tượng object dùng tên của thuộc tính không theo chuẩn :

```
var o = {  
  something: 1,  
  'yes or no' : 'yes',
```

```
'!@#$$%^&*': true  
};
```

Đây là 1 đối tượng object hợp lệ, dấu nháy là bắt buộc cho các thuộc tính thứ 2 và thứ 3. nếu không sẽ gặp phải lỗi

2.1. Các phần tử, các thuộc tính, các phương thức

Khi ta nói về các mảng arrays, ta nói rằng chúng chứa các phần tử. khi ta nói về các đối tượng objects thì ta nói rằng chúng chứa các thuộc tính

1 thuộc tính của **1 đối tượng object có thể chứa 1 hàm function**, bởi vì các hàm functions này cũng chỉ là 1 kiểu dữ liệu. trong trường hợp này ta nói thuộc tính này chính là 1 phương thức :

```
var dog = {  
  name: 'Benji',  
  talk: function() {  
    alert('Woof, woof!');  
  }  
};
```

Ta cũng có thể lưu các hàm functions như các phần tử mảng array và gọi chúng :

```
>>> var a = [];  
>>> a[0] = function(what){alert(what);};  
>>> a[0]('Boo!');
```

2.2. Các mảng arrays kết hợp

JS sử dụng các mảng array để biểu diễn các mảng array thông thường và các đối tượng và nó được gọi là các mảng array kết hợp hay gọi là hash. nếu ta muốn 1 hash trong JS, thì ta phải sử dụng 1 đối tượng

2.3.Các truy cập vào các thuộc tính của đối tượng

Có 2 cách để truy cập vào 1 thuộc tính của đối tượng :

- Cách sử dụng kí tự dấu ngoặc vuông [] ví dụ như hero['occupation']
- Cách sử dụng dấu chấm • ví dụ như hero.occupation

Cách sử dụng dấu chấm là dễ hơn để đọc và viết nhưng nó không phải lúc nào cũng được sử dụng cùng cách với việc thêm vào dấu nháy cho tên các thuộc tính. nếu tên các thuộc tính không phải kiểu chuẩn tắc thì ta không thể sử dụng dấu chấm

Ví dụ :

```
var hero = {  
  breed: 'Turtle',  
  occupation: 'Ninja'  
};
```

Để truy cập vào 1 thuộc tính bằng cách sử dụng dấu chấm :

```
>>> hero.breed;
```

"Turtle"

Để truy cập vào 1 thuộc tính bằng cách sử dụng dấu ngoặc vuông :

```
>>> hero['occupation'];
```

"Ninja"

Việc truy cập vào 1 thuộc tính không tồn tại thì trả về **undefined** :

```
>>> 'Hair color is ' + hero.hair_color;
```

"Hair color is undefined"

Các đối tượng có thể chứa bất cứ dữ liệu nào, bao gồm các đối tượng objects khác :

```
var book = {  
  name: 'Catch-22',
```

published: 1961,

```
author: {  
  firstname: 'Joseph',  
  lastname: 'Heller'  
}
```

```
};
```

Để lấy về thuộc tính firstname của đối tượng object chứa trong thuộc tính author của đối tượng book object thì ta có thể sử dụng :

```
>>> book.author.firstname
```

```
"Joseph"
```

Hay sử dụng theo cách dấu ngoặc vuông :

```
>>> book['author']['lastname']
```

```
"Heller"
```

Hay là theo cách kết hợp cả 2 :

```
>>> book.author['lastname']
```

```
"Heller"
```

```
>>> book['author'].lastname
```

```
"Heller"
```

1 trường hợp khác ta buộc phải sử dụng theo cách móc vuông nếu tên của thuộc tính mà ta cần truy cập là không được biết trước (hay cách sử dụng động).trong suốt thời gian thực thi, nó được lưu trữ động trong 1 biến :

```
>>> var key = 'firstname';
```

```
>>> book.author[key];
```

```
"Joseph"
```

2.4.Cách gọi các phương thức của 1 đối tượng

Bởi vì 1 phương thức cũng chỉ là 1 thuộc tính mà nó có chức năng như 1 hàm function, nên ta có thể truy cập các phương thức này trong cùng cách với ta truy cập các thuộc tính : bằng cách sử dụng kí tự dấu chấm hay bằng cách sử dụng kí tự dấu ngoặc vuông.cách gọi 1 phương thức cũng như cách gọi của bất cứ các hàm function nào khác : cũng thêm vào dấu móc đơn () và phía sau tên của phương thức :

```
var hero = {  
  breed: 'Turtle',  
  occupation: 'Ninja',  
  say: function() {  
    return 'I am ' + hero.occupation;  
  }  
}
```

```
>>> hero.say();
```

```
"I am Ninja"
```

Với bất cứ tham số nào, ta muốn chuyển tiếp vào 1 phương thức thì ta cũng xử lý nó như các hàm thông thường :

```
>>> hero.say('a', 'b', 'c');
```

Bởi vì ta có thể sử dụng các dấu ngoặc vuông giống như trong mảng array để truy cập 1 thuộc tính, điều này có nghĩa là ta cũng có thể sử dụng các dấu ngoặc này để truy cập và gọi các phương thức dù nó không hay được sử dụng :

```
>>> hero['say']();
```

2.5.Cách biến đổi các thuộc tính / phương thức

JS là 1 ngôn ngữ rất linh hoạt, nó cho phép ta biến đổi các phương thức và thuộc tính của các đối tượng đã tồn tại vào bất cứ lúc nào.điều này bao gồm thêm các thuộc tính mới hay xóa chúng.ta có thể bắt đầu với 1 đối tượng rỗng và thêm các thuộc tính vào sau đó.

Ví dụ :

1 đối tượng rỗng :

```
>>> var hero = { };
```

Truy cập vào 1 thuộc tính không tồn tại :

```
>>> typeof hero.breed
```

"undefined"

Thêm vào 1 số thuộc tính và 1 phương thức :

```
>>> hero.breed = 'turtle';
```

```
>>> hero.name = 'Leonardo';
```

```
>>> hero.sayName = function() {return hero.name;};
```

Gọi phương thức :

```
>>> hero.sayName();
```

"Leonardo"

Xóa 1 thuộc tính :

```
>>> delete hero.name;
```

True

Gọi phương thức lại 1 lần nữa :

```
>>> hero.sayName();
```

reference to undefinedproperty hero.name

2.6.Các sử dụng từ khóa this

trong ví dụ trước thì phương thức sayName() sử dụng hero.name để truy cập vào thuộc tính name của đối tượng hero object.khi ta đang ở trong 1 phương thức thì có 1 cách khác để truy cập tới phương thức này của đối tượng là cách sử dụng từ đặc biệt **this**

```
var hero = {  
  name: 'Rafaelo',  
  sayName: function() {  
    return this.name;  
  }  
}  
  
>>> hero.sayName();  
"Rafaelo"
```

Do vậy khi ta nói **this** nghĩa là thực chất ta đang nói tới đối tượng này hay đối tượng hiện tại

2.7.Các hàm tạo constructor

Có 1 cách khác để tạo ra các đối tượng objects : bằng cách sử dụng các hàm tạo.như ví dụ :

```
function Hero() {  
  
  this.occupation = 'Ninja';  
  
}
```

Để tạo ra 1 đối tượng bằng cách sử dụng hàm function này ta thêm vào 1 toán tử **new** giống như sau :

```
>>> var hero = new Hero();  
  
>>> hero.occupation;  
  
"Ninja"
```

Lợi ích của việc sử dụng hàm tạo là nó cho phép ta thêm vào các tham số để tạo ra 1 đối tượng mới.ta chỉnh sửa lại hàm tạo để nhận 1 tham số và gán nó thành giá trị của thuộc tính name

```
function Hero(name) {  
  this.name = name;  
  this.occupation = 'Ninja';  
  this.whoAreYou = function() {  
    return "I'm " + this.name + " and I'm a " + this.occupation;  
  }  
}
```

Giờ ta có thể tạo ra các đối tượng khác nhau bằng cách sử dụng cùng 1 hàm tạo :

```
>>> var h1 = new Hero('Michelangelo');
```

```
>>> var h2 = new Hero('Donatello');
```

```
>>> h1.whoAreYou();
```

"I'm Michelangelo and I'm a Ninja" "I'm Michelangelo and I'm a Ninja"

```
>>> h2.whoAreYou();
```

"I'm Donatello and I'm a Ninja" "I'm Donatello and I'm a Ninja"

Theo định chuẩn, ta nên viết hoa chữ đầu tiên tên của hàm tạo vì nó khác với các hàm thông thường. nếu ta gọi 1 hàm mà nó được thiết kế là 1 hàm tạo, nhưng ta bỏ qua toán tử **new** thì không xảy ra lỗi nào cả nhưng cách thức xử lý sẽ không theo ý ta muốn :

```
>>> var h = Hero('Leonardo');
```

```
>>> typeof h
```

"undefined"

Thế chuyện gì xảy ra tại đây ?

Không có toán tử **new** thì ta không tạo ra 1 đối tượng mới nào cả. hàm này sẽ được gọi giống như bất kì các hàm thông thường nào khác, do đó h chứa giá trị trả về trong hàm. mà hàm này không trả về bất cứ thứ gì, do vậy nó trả về **undefined**, và nó sẽ được gán vào trong h

Thế trong trường hợp này thì **this** tham chiếu tới cái gì ?

Câu trả lời là nó tham chiếu tới đối tượng toàn cục **global object**.

2.8.Đối tượng toàn cục Global Object

Môi trường máy chủ cung cấp 1 đối tượng toàn cục và tất cả các biến toàn cục thực tế là các thuộc tính của đối tượng này

Nếu môi trường máy chủ là 1 trình duyệt web, thì đối tượng toàn cục global object được gọi là **window**

Để minh họa, ta có thể thử gán 1 biến toàn cục bên ngoài 1 hàm như sau :

```
>>> var a = 1;
```

Sau đó ta có thể truy cập biến toàn cục này trong các cách khác nhau :

- Như 1 biến a thông thường
- Như 1 thuộc tính của đối tượng toàn cục ví dụ : **window['a']** hay **window.a**

Giờ ta quay trở lại ví dụ trước nơi ta định nghĩa 1 hàm tạo và gọi nó mà không có toán tử **new**. trong trường hợp như thế thì **this** tham chiếu tới đối tượng toàn cục global object và tất cả các thuộc tính được thiết lập với **this** trở thành thuộc tính của đối tượng **window**

Để khai báo 1 hàm tạo và gọi nó mà không có **new** thì trả về "undefined":

```
>>> function Hero(name) {this.name = name;}
```

```
>>> var h = Hero('Leonardo');
```

```
>>> typeof h
```

```
"undefined"
```

```
>>> typeof h.name
```

```
h has no properties
```

bởi vì ta có this bên trong hero, nên 1 biến toàn cục (hay 1 thuộc tính của đối tượng toàn cục) được gọi là name được tạo

```
>>> name
```

```
"Leonardo"
```

```
>>> window.name
```

"Leonardo"

Nếu ta gọi tới cùng 1 hàm tạo đó nhưng lúc này ta sử dụng new, thì sau đó 1 đối tượng mới được trả về và **this** tham chiếu tới nó :

```
>>> var h2 = new Hero('Michelangelo');
```

```
>>> typeof h2
```

"object"

```
>>> h2.name
```

"Michelangelo"

2.9. Thuộc tính tạo

Khi 1 đối tượng được tạo, thì 1 thuộc tính đặc biệt được gán vào nó ở sau cảnh – thuộc tính tạo. nó chứa 1 tham chiếu tới hàm tạo được sử dụng để tạo ra đối tượng object

Tiếp tục với ví dụ trước :

```
>>> h2.constructor
```

Hero(name)

Bởi vì thuộc tính tạo chứa 1 tham chiếu tới 1 hàm, nên ta có thể gọi hàm này để tạo ra 1 đối tượng mới. như đoạn code sau nói rằng “ta không quan tâm tới cách mà đối tượng h2 được tạo ra nhưng ta muốn 1 cái khác giống như vậy “

```
>>> var h3 = new h2.constructor('Rafaello'); // h3 được tạo sử dụng hàm tạo của h2
```

```
>>> h3.name;
```

"Rafaello"

Đây là 1 dạng mờ hóa, ở đây ta muốn tạo ra 1 đối tượng h3 mới mà nó dựa vào 1 hàm tạo – hàm đã tạo nên đối tượng h2 mà không cần phải quan tâm cụ thể tới hàm tạo cụ thể của h2 ra sao

Nếu 1 đối tượng được tạo bằng cách sử dụng kí tự { }, thì hàm tạo của nó là 1 hàm tạo được xây dựng sẵn

```
>>> var o = {};  
>>> o.constructor;  
Object()  
>>> typeof o.constructor;  
"function"
```

2.10. Toán tử instanceof

Bằng cách sử dụng toán tử **instanceof** ta có thể kiểm tra được 1 đối tượng được tạo với 1 hàm tạo riêng biệt nào :

```
>>> function Hero(){ }  
>>> var h = new Hero();  
>>> var o = { };  
>>> h instanceof Hero;  
true  
>>> h instanceof Object;  
false  
>>> o instanceof Object;  
True
```

Chú ý rằng ta không đặt dấu ngoặc đơn sau tên hàm (**không** được sử dụng

h instanceof Hero()). đây là bởi vì ta không thực hiện gọi hàm này, nhưng chỉ tham chiếu tới bằng tên của nó giống như bất cứ biến nào khác

2.11. Các hàm functions mà trả về các đối tượng objects

để thêm vào cách sử dụng hàm tạo và toán tử new để tạo ra các đối tượng objects, thì ta có thể cũng sử dụng 1 hàm thông thường và tạo ra các đối tượng mà không cần sử dụng toán tử **new**.

Ví dụ đây là 1 hàm factory() để tạo ra các đối tượng objects :

```
function factory(name) {
```

```
return {  
  
  name: name  
  
};  
  
}
```

Using the factory():

```
>>> var o = factory('one');  
  
>>> o.name  
  
"one"  
  
>>> o.constructor
```

Object()

Thực tế, ta cũng có thể sử dụng các hàm tạo và trả về các đối tượng objects khác với **this**.điều này có nghĩa là ta có thể chỉnh sửa cách thức xử lý mặc định với hàm tạo

Ví dụ thông thường :

```
>>> function C() {this.a = 1;}  
  
>>> var c = new C();  
  
>>> c.a  
  
1
```

Ví dụ chỉnh sửa hàm tạo :

```
>>> function C2() {this.a = 1; return {b: 2};}  
  
>>> var c2 = new C2();  
  
>>> typeof c2.a  
  
"undefined"
```

```
>>> c2.b
```

```
2
```

Vậy chuyện gì đang xảy ra ?

Câu trả lời là ta thay thế trả về các đối tượng `this` mà chứa thuộc tính `a`, thì hàm tạo sẽ trả về 1 đối tượng khác chứa thuộc tính `b`.điều này chỉ có thể xảy ra nếu giá trị trả về là 1 đối tượng object.

2.12.Các chuyển tiếp các đối tượng

Khi ta sao chép 1 đối tượng object hay chuyển tiếp nó vào 1 hàm function, ta chỉ cần chuyển tiếp 1 tham chiếu tới đối tượng object đó.do vậy, nếu ta tạo ra 1 thay đổi trên tham chiếu này, thì thực chất ta đang thực hiện thay đổi đó trên đối tượng gốc

Đây là 1 ví dụ về cách mà ta có thể gán 1 đối tượng object vào 1 biến khác và sau đó tạo ra sự thay đổi trên phiên bản sao chép,như kết quả ở dưới thì đối tượng gốc cũng thay đổi theo :

```
>>> var original = {howmany: 1};
```

```
>>> var copy = original;
```

```
>>> copy.howmany
```

```
1
```

```
>>> copy.howmany = 100;
```

```
100
```

```
>>> original.howmany
```

```
100
```

Những thứ tương tự cũng được áp dụng vào khi chuyển tiếp các đối tượng objects vào các hàm functions :

```
>>> var original = {howmany: 100};
```

```
>>> var nullify = function(o) {o.howmany = 0;}
```

```
>>> nullify(original);
```

```
>>> original.howmany
```

```
0
```

2.13.Cách so sánh các đối tượng objects

Khi ta so sánh các đối tượng objects, ta sẽ nhận về giá trị **true** nếu và chỉ nếu ta so sánh 2 tham chiếu tới cùng 1 đối tượng.việc so sánh 2 đối tượng riêng biệt dù có cùng chính xác về các phương thức và thuộc tính thì vẫn trả về là **false**

Ta tạo ra 2 đối tượng như sau :

```
>>> var fido = {breed: 'dog'};
```

```
>>> var benji = {breed: 'dog'};
```

Và tiến hành so sánh chúng thì nhận được giá trị **false** :

```
>>> benji === fido
```

```
false
```

```
>>> benji == fido
```

```
False
```

Ta có thể tạo ra 1 biến mới là **mydog** và gán nó thành 1 trong các đối tượng trên, theo cách này thì mydog thực chất chỉ tới cùng 1 đối tượng :

```
>>> var mydog = benji;
```

Trong trường hợp này thì benji là mydog bởi vì chúng cùng là 1 đối tượng (thay đổi thuộc tính mydog sẽ thay đổi thuộc tính của benji).và sự so sánh sau sẽ trả về là true :

```
>>> mydog === benji
```

```
True
```

Và bởi vì fido là 1 đối tượng khác, nên khi so sánh fido với mydog sẽ trả về là false :

```
>>> mydog === fido
```

```
False
```

III. Các đối tượng được xây dựng sẵn

Các đối tượng xây dựng sẵn chia làm 3 nhóm :

- Các đối tượng đóng gói dữ liệu Data wrapper objects – các đối tượng object, mảng array, hàm function, kiểu Boolean, Number, và String. các đối tượng này tương ứng với các kiểu dữ liệu data types khác nhau trong JS. về cơ bản, có 1 đối tượng đóng gói dữ liệu được trả về bởi **typeof** với các ngoại lệ như "undefined" và "null".
- Các đối tượng tiện ích Utility objects – có Math, Date, RegExp
- Các đối tượng lỗi Error objects – đối tượng này cũng giống như các đối tượng khác nó có thể giúp chương trình của ta khôi phục lại được trạng thái hoạt động khi 1 vài thứ gì đó hoạt động không như mong muốn

3.1. Đối tượng object

object là cha của tất cả các đối tượng JS, nó có nghĩa là mọi đối tượng object mà ta tạo ra đều kế thừa từ đối tượng này. để tạo ra 1 đối tượng rỗng mới thì ta có thể sử dụng **{ }** hoặc hàm tạo **Object()**. 2 cách sau là như nhau :

```
>>> var o = { };
```

```
>>> var o = new Object();
```

1 đối tượng rỗng thực chất không phải rỗng hoàn toàn trừ khi nó chứa 1 vài phương thức hay thuộc tính. như sau :

- Thuộc tính o.constructor trả về hàm tạo
- o.toString() là 1 phương thức trả về 1 chuỗi string biểu diễn đối tượng
- o.valueOf() trả về 1 giá trị đơn biểu diễn đối tượng và thường đây chính là đối tượng

ví dụ :

đầu tiên ta tạo ra 1 đối tượng :

```
>>> var o = new Object();
```

Gọi phương thức toString() thì nó trả về 1 chuỗi string biểu diễn đối tượng :

```
>>> o.toString()
```

"[object Object]"

toString() sẽ được gọi bên trong JS, khi 1 đối tượng được sử dụng trong 1 ngữ cảnh string. ví dụ như `alert()` cũng hoạt động chỉ với string, do vậy nếu ta gọi hàm `alert()` chuyển tiếp vào 1 đối tượng object, thì phương thức `toString()` sẽ được gọi. ví dụ 2 dòng sau cùng cho 1 kết quả :

```
>>> alert(o)
```

```
>>> alert(o.toString())
```

Dạng ngữ cảnh string khác là sự ghép nối chuỗi. nếu ta thử ghép nối chuỗi với 1 đối tượng với 1 string, thì phương thức `toString()` sẽ được gọi đầu tiên :

```
>>> "An object: " + o
```

```
"An object: [object Object]"
```

`valueOf()` là 1 phương thức khác mà tất cả các đối tượng JS đều có. ví dụ cho đối tượng đơn giản (mà tất cả hàm tạo của nó đều là `Object()`) thì phương thức `valueOf()` trả về chính là object của chính nó

```
>>> o.valueOf() === o
```

```
True
```

Kết luận :

- ta có thể tạo các đối tượng theo 2 cách `o = { }`; hay `o = new Object();` (**Object** là tên của 1 hàm tạo)
- Bất kỳ đối tượng , không có vấn đề làm thế nào phức tạp , thừa kế từ `Object` object do đó cung cấp các phương pháp như `toString ()` và tài sản như: `constructor`

3.2.Đối tượng Mảng array

`Array()` là 1 hàm được xây dựng sẵn mà ta có thể sử dụng để định nghĩa như 1 hàm tạo để tạo ra mảng arrays :

```
>>> var a = new Array();
```

Dòng trên cũng tương đương với :


```
>>> var a = [];
```

Ta có thể thêm vào các phần tử như sau :

```
>>> a[0] = 1; a[1] = 2; a;
```

```
[1, 2]
```

Khi ta sử dụng hàm tạo array(), thì ta cũng có thể chuyển tiếp các giá trị mà ta sẽ gán vào các phần tử của mảng mới như sau :

```
>>> var a = new Array(1,2,3,'four');
```

```
>>> a;
```

```
[1, 2, 3, "four"]
```

1 ngoại lệ với điều này là khi ta chuyển tiếp 1 số đơn vào hàm tạo. trong trường hợp này , số được chuyển tiếp sẽ được xét như chiều dài của mảng :

```
>>> var a2 = new Array(5);
```

```
>>> a2;
```

```
[undefined, undefined, undefined, undefined, undefined]
```

Bởi vì các mảng arrays được tạo với 1 hàm tạo, điều này có nghĩa là mảng array thực chất là đối tượng object ?

Câu trả lời là đúng và ta có thể kiểm tra lại điều này bằng cách sử dụng toán tử **typeof**

```
>>> typeof a;
```

```
"object"
```

Bởi vì mảng arrays cũng là các đối tượng objects, điều này có nghĩa là chúng kế thừa các thuộc tính và phương thức của đối tượng parent Object.

```
>>> a.toString();
```

```
"1,2,3,four"
```

```
>>> a.valueOf()
```

```
[1, 2, 3, "four"]
```

```
>>> a.constructor
```

```
Array()
```

Các mảng arrays là các đối tượng objects nhưng chúng là 1 dạng đặc biệt vì :

- Tên của các thuộc tính của chúng tự động được gán bằng cách sử dụng các số bắt đầu từ 0
- Chúng có 1 thuộc tính độ dài length – chứa số phần tử có trong mảng
- Chúng có thêm các phương thức được xây dựng sẵn và hơn thế chúng còn được kế thừa từ đối tượng parent object

Xét ví dụ khác biệt giữa mảng array và đối tượng object :

Ta bắt đầu tạo ra 1 mảng array và 1 đối tượng object rỗng :

```
>>> var a = [], o = {};
```

Array có 1 thuộc tính length trong khi đó đối tượng object thông thường thì không :

```
>>> a.length
```

```
0
```

```
>>> typeof o.length
```

```
"undefined"
```

Đều có thể thêm được các thuộc tính kiểu số và không phải số vào cả mảng arrays và objects :

```
>>> a[0] = 1; o[0] = 1;
```

```
>>> a.prop = 2; o.prop = 2;
```

Thuộc tính length luôn được **cập nhật với số thuộc tính kiểu số** và **bỏ qua thuộc tính không phải là số**

```
>>> a.length
```

```
1
```

Thuộc tính length có thể được ta thiết lập. việc thiết lập nó lớn hơn số phần tử thực tại trong mảng array thì tạo ra các phần tử rỗng (hoặc với 1 giá trị là undefined)

```
>>> a.length = 5
```

```
5
```

```
>>> a
```

```
[1, undefined, undefined, undefined, undefined]
```

Các thiết lập length nhỏ hơn các phần tử có trong mảng thì nó sẽ xóa đi các phần tử ở phía đuôi :

```
>>> a.length = 2;
```

```
2
```

```
>>> a
```

```
[1, undefined]
```

3.3.Đối tượng Function

Các hàm functions thực ra cũng là các đối tượng.và nó có 1 hàm tạo được xây dựng sẵn được gọi là function() cho phép 1 cách thay thế (**nhưng không khuyên dùng**) để tạo ra 1 hàm function

Ví dụ 3 cách sau là tương đương :

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

```
>>> sum(1, 2)
```

```
3
```

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

```
>>> sum(1, 2)
```

```
3
```

```
>>> var sum = new Function('a', 'b', 'return a + b;');
```

```
>>> sum(1, 2)
```

```
3
```

Khi ta sử dụng hàm tạo `function()`, thì ta chuyển tiếp tên các tham số vào phần đầu sau đó là phần thân mã code của hàm `function` cũng tương tự như vậy được chuyển tiếp vào giống như 1 chuỗi. và cách sử dụng này nên hạn chế nhất có thể

Ta có thể sử dụng hàm tạo `function` để tạo ra các hàm mà có rất nhiều tham số, nhớ rằng các tham số này được chuyển tiếp giống như 1 danh sách được ngăn cách bởi dấu phẩy như sau :

```
>>> var first = new Function('a, b, c, d', 'return arguments;');
```

```
>>> first(1,2,3,4);
```

```
[1, 2, 3, 4]
```

```
>>> var second = new Function('a, b, c', 'd', 'return arguments;');
```

```
>>> second(1,2,3,4);
```

```
[1, 2, 3, 4]
```

```
>>> var third = new Function('a', 'b', 'c', 'd',
```

```
'return arguments;');
```

```
>>> third(1,2,3,4);
```

```
[1, 2, 3, 4]
```

3.3.1. Các thuộc tính của đối tượng `function`

Giống như bất kì đối tượng nào khác, các hàm `functions` cũng có 1 thuộc tính tạo **constructor** chứa 1 tham chiếu tới hàm tạo `Function()` :

```
>>> function myfunc(a){return a;}
```

```
>>> myfunc.constructor
```

```
Function()
```

Các hàm `functions` cũng có 1 thuộc tính **length** – chứa số các tham số được truyền vào hàm :

```
>>> function myfunc(a, b, c){return true;}
```

```
>>> myfunc.length
```

3

1 thuộc tính nữa không theo chuẩn ECMA nhưng nó lại được thực thi qua các trình duyệt – thuộc tính `caller`. thuộc tính này tham chiếu tới hàm function mà đã gọi hàm function hiện tại. như ví dụ ở dưới 1 hàm A() nhận về lời gọi từ bên trong hàm B(). nếu bên trong A(), ta đặt `A.caller` thì nó sẽ trả về là hàm B()

```
>>> function A(){return A.caller;}
```

```
>>> function B(){return A();}
```

```
>>> B()
```

B()

Điều này có thể hữu dụng nếu ta muốn hàm function của ta phản hồi lại 1 cách khác nhau phụ thuộc vào hàm đã gọi nó. nếu ta gọi hàm A() từ phạm vi toàn cục (nằm ngoài bất cứ hàm function nào) thì `A.caller` sẽ là null

```
>>> A()
```

Null

Thuộc tính quan trọng nhất của 1 đối tượng function là thuộc tính **prototype**. ta sẽ đề cập nó ở phía sau, tuy nhiên ta nói qua về nó :

- Thuộc tính **prototype** của 1 hàm function chứa 1 đối tượng object
- Nó **chỉ được sử dụng** khi ta sử dụng hàm function này **như 1 hàm tạo**
- Tất cả các đối tượng objects được tạo với hàm này đều giữ 1 tham chiếu tới thuộc tính **prototype** và có thể sử dụng các thuộc tính của nó như các thuộc tính của chúng

ta xét nhanh 1 ví dụ để chứng minh thuộc tính prototype. ta bắt đầu với 1 đối tượng đơn giản mà có 1 thuộc tính là **name** và phương thức là **say()**

```
var some_obj = {
```

```
  name: 'Ninja',
```

```
say: function(){  
  
return 'I am a ' + this.name;  
  
}  
  
}
```

Nếu ta tạo ra 1 hàm function rỗng thì ta có thể biến đổi nó 1 cách tự động với 1 thuộc tính **prototype** mà thuộc tính này chứa 1 **đối tượng rỗng**

```
>>> function F(){ }  
  
>>> typeof F.prototype  
  
"object"
```

Sẽ rất thú vị khi ta chỉnh sửa thuộc tính prototype.ta có thể thay thế 1 đối tượng rỗng mặc định thành bất cứ đối tượng nào khác.giờ ta thực hiện gán đối tượng **some_obj** vào thuộc tính **prototype**

```
>>> F.prototype = some_obj // thực hiện nhập các nội dung kiểu mẫu gồm các phương  
thức và thuộc tính trong đối tượng some_obj vào bên trong hàm F
```

Giờ bằng cách sử dụng hàm F() như 1 hàm tạo, ta có thể tạo ra 1 đối tượng mới tên là obj mà nó sẽ nhận các thuộc tính của **F.prototype** như là các thuộc tính của nó

```
>>> var obj = new F();  
  
>>> obj.name  
  
"Ninja"  
  
>>> obj.say()  
  
"I am a Ninja"
```

3.3.2. Các phương thức trong các đối tượng objects function

Các đối tượng function cũng là các con cháu của đối tượng cha parent Object, theo đó mặc định nó cũng có các phương thức mặc định như toString(). khi ta gọi 1 hàm function, thì phương thức toString() trả về mã nguồn bên trong hàm

```
>>> function myfunc(a, b, c) {return a + b + c;}
```

```
>>> myfunc.toString()
```

```
"function myfunc(a, b, c) {
```

```
  return a + b + c;
```

```
}"
```

Nếu ta có thử nhìn hé vào mã nguồn bên trong các hàm xây dựng sẵn, ta sẽ lấy về được các chuỗi [native code]

```
>>> eval.toString()
```

```
"function eval() {
```

```
[native code]
```

```
}"
```

2 phương thức hữu dụng của các đối tượng function là call() và apply(). chúng

cho phép các đối tượng **mượn các phương thức từ các đối tượng khác** và gọi chúng ngay bên trong bản thân chúng. đây là 1 cách rất mạnh mẽ và dễ dàng để giảm thiểu mã code

Ta xét ta có 1 đối tượng **some_obj** object, chứa 1 phương thức **say()**

```
var some_obj = {
```

```
  name: 'Ninja',
```

```
  say: function(who) {
```

```
    return 'Haya ' + who + ', I am a ' + this.name;
```

```
}
```

```
}
```

Ta có thể gọi phương thức say() như sau :

```
>>> some_obj.say('Dude');
```

```
"Haya Dude, I am a Ninja"
```

Giờ ta tạo ra 1 đối tượng đơn giản **my_obj**, mà chỉ có 1 thuộc tính **name** :

```
>>> my_obj = {name: 'Scripting guru'};
```

Bên trong đối tượng **my_obj** không có phương thức say() tuy nhiên ta lại muốn mượn phương thức say() này từ đối tượng **some_obj** vào trong đối tượng **my_obj** thì ta làm như sau :

```
>>> some_obj.say.call(my_obj, 'Dude');
```

Nó đã hoạt động.nhưng chuyện gì sẽ xảy ra ? ta gọi phương thức call() của phương thức say() , 2 tham số được chuyển tiếp là đối tượng my_obj và chuỗi 'Dude'.kết quả khi phương thức say() được gọi, thì có tham chiếu tới đối tượng this, khi này đối tượng this chỉ tới đối tượng my_obj.do vậy this.name không trả về **Ninja** nhưng lại trả về

Scripting guru

Nếu ta có nhiều hơn các tham số được chuyển tiếp vào khi gọi phương thức call(), thì cứ việc chuyển tiếp chúng :

```
some_obj.someMethod.call(my_obj, 'a', 'b', 'c');
```

nếu ta không chuyển tiếp 1 đối tượng như 1 tham số đầu tiên bên trong call() hay là null, thì đối tượng toàn cục sẽ được giả định

phương thức apply() là việc giống với cách của call() nhưng với 1 chút khác biệt là tất cả các tham số mà ta muốn chuyển tiếp vào phương thức của 1 đối tượng khác thì chúng phải được chuyển tiếp như 1 mảng như ví dụ sau :

```
some_obj.someMethod.apply(my_obj, ['a', 'b', 'c']);
```

```
some_obj.someMethod.call(my_obj, 'a', 'b', 'c');
```

hay như làm trong ví dụ trên :

```
>>> some_obj.say.apply(my_obj, ['Dude']);
```

```
"Haya Dude, I am a Scripting guru"
```

IV. Biểu thức chính quy

Biểu thức chính quy cung cấp 1 cách mạnh mẽ để tìm kiếm và thao tác trên các kí tự text

1 biểu thức chính quy bao gồm :

- 1 mẫu mà ta sử dụng để chọn các text cho phù hợp
- Các flags để cung cấp các chỉ dẫn rõ hơn về cách các mẫu được áp dụng

Mẫu có thể đơn giản là các text chuỗi về nguyên văn nhưng rất hiếm và trong trường hợp đó ta nên sử dụng **indexOf()**. đa số thì các mẫu thường rất phức tạp và rất khó hiểu. để làm chủ được các biểu thức chính quy cũng đã là 1 chủ đề lớn

JS cung cấp hàm tạo **RegExp()** cho phép ta tạo ra các đối tượng biểu thức chính quy

```
>>> var re = new RegExp("j.*t");
```

Đây cũng là 1 dạng biểu thức chính quy :

```
>>> var re = /j.*t/;
```

ở ví dụ trên, **j.*t** là 1 mẫu biểu thức chính quy. nó có nghĩa là “ tương ứng với bất kì chuỗi

nào bắt đầu từ **j** với kết thúc là **t** và có hay không có các kí tự nằm ở giữa chúng. dấu ***** có

nghĩa là “không có hay có nhiều hơn ở phía trước “ dấu **.** có nghĩa là “bất cứ kí tự nào”. mẫu này cần đặt vào dấu nháy khi ta sử dụng hàm tạo **RegExp()**

4.1. Các thuộc tính của các đối tượng Objects

các đối tượng biểu thức chính quy có các thuộc tính sau :

- **global**: nếu thuộc tính này là false (theo mặc định) thì tìm kiếm dừng lại khi tương ứng (với mẫu) đầu tiên được tìm thấy. thiết lập là true nếu ta muốn tìm tất cả các tương ứng
- **ignoreCase**: dạng chữ (viết hoa hay thường) có tương ứng hay không, mặc định là false
- **multiline**: tìm kiếm các tương ứng mà có phạm vi trên nhiều hơn 1 dòng, mặc định là false

- lastIndex: vị trí mà bắt đầu tìm kiếm, mặc định là 0
- source: chứa các mẫu regexp

không thuộc tính nào ở trên ngoại trừ lastIndex, có thể được thay đổi khi đối tượng được tạo

3 tham số đầu tiên miêu tả các điều chỉnh regex. nếu ta tạo ra 1 đối tượng regex object bằng cách sử dụng hàm tạo thì ta có thể chuyển tiếp bất kì bộ kí tự sau vào như là tham số thứ 2 :

- 'g' có nghĩa là **global**
- 'I' có nghĩa là **ignoreCase**
- 'm' có nghĩa là **multiline**

Các kí tự này có thể được sắp xếp theo bất cứ thứ tự nào. nếu 1 kí tự được chuyển tiếp, thì các hiệu chỉnh tương ứng được thiết lập là true. trong các trường hợp sau thì tất cả các hiệu chỉnh được thiết lập là true :

```
>>> var re = new RegExp('j.*t', 'gmi');
```

Ta có thể kiểm tra lại :

```
>>> re.global;
```

true

khi thiết lập như sau thì các hiệu chỉnh không thể thay đổi :

```
>>> re.global = false;
```

```
>>> re.global
```

true

để thiết lập các hiệu chỉnh bằng cách sử dụng chuỗi regex, thì ta cần thêm chúng vào sau kí tự đóng dấu gạch chéo /

```
>>> var re = /j.*t/ig;
```

```
>>> re.global
```

True

4.2.Các phương thức của các đối tượng regexp

Các đối tượng regex cung cấp 2 phương thức mà ta có thể sử dụng để tìm kiếm các tương ứng là **test()** và **exec()**. cả 2 phương thức này nhận 1 chuỗi tham số. **test()** trả về kiểu Boolean (true nếu có 1 tương ứng và ngược lại là false) trong khi **exec()** trả về 1 mảng array của các chuỗi tương ứng. hiển nhiên **exec()** là làm việc hiệu quả hơn, do vậy sử dụng **test()** vào những lúc ta không thực sự cần làm 1 vài thứ gì đó với các tương ứng. thông thường ta sử dụng **test()** để kiểm tra tính hợp lệ

Ví dụ :

Không tương ứng do chữ J được viết hoa

```
>>> /j.*t/.test("Javascript")
```

False

Trường hợp bỏ qua dạng kiểu chữ :

```
>>> /j.*t/i.test("Javascript")
```

True

Ta sử dụng **exec()** thì nó trả về 1 mảng array và ta có thể truy cập tới phần tử đầu tiên như sau :

```
>>> /j.*t/i.exec("Javascript")[0]
```

"Javascript"

4.3.Các phương thức string mà nhận các biểu thức chính quy như các tham số

Các đối tượng string cung cấp cho ta các phương thức sau mà nhận các đối tượng biểu thức chính quy như là các tham số :

- **Match()** trả về 1 mảng array của các tương ứng
- **Search()** trả về vị trí của tương ứng đầu tiên
- **Replace()** cho phép ta thế chuỗi tương ứng text với 1 kí tự khác
- **Split()** cũng nhận 1 regexp khi cắt 1 chuỗi string thành các mảng phần tử array

4.3.1.search() và match()

xét 1 vài ví dụ về cách sử dụng phương thức **search()** và **match()**. đầu tiên ta tạo ra 1 đối tượng string :

```
>>> var s = new String('HelloJavaScriptWorld');
```

Bằng cách sử dụng **match()** ta lấy được 1 mảng array chỉ chứa 1 tương ứng đầu tiên

```
>>> s.match(/a/);
```

```
["a"]
```

Bằng cách sử dụng hiệu chỉnh g, ta thực thi tìm kiếm toàn cục, do vậy kết quả mảng array có chứa 2 phần tử :

```
>>> s.match(/a/g);
```

```
["a", "a"]
```

Loại bỏ kiểu chữ :

```
>>> s.match(/j.*a/i);
```

```
["Java"]
```

Phương thức **search()** cho ta biết vị trí của tương ứng đầu tiên :

```
>>> s.search(/j.*a/i);
```

```
5
```

4.3.2.replace()

replace() cho phép ta thay thế các kí tự text tương ứng với 1 vài chuỗi string khác. ví dụ sau xóa tất cả các kí tự viết hoa (nó thay thế chúng với chuỗi rỗng)

```
>>> s.replace(/[A-Z]/g, "");
```

```
"elloavacriptorld"
```

Nếu ta bỏ qua điều chỉnh g, thì ta chỉ thay thế tương ứng đầu tiên :

```
>>> s.replace(/[A-Z]/, "");
```

```
"elloJavaScriptWorld"
```

Khi 1 tương ứng được tìm thấy, nếu ta muốn những chuỗi tương ứng này vào trong chuỗi thay thế, ta có thể nhận được nó bằng cách sử dụng **\$&**. đây là cách thêm vào dấu gạch dưới trước chuỗi tương ứng trong khi vẫn giữ được chuỗi tương ứng

```
>>> s.replace(/[A-Z]/g, "__$&");
```

```
"__Hello__Java__Script__World"
```

Khi biểu thức chính quy chứa các nhóm (biểu diễn bằng dấu ngoặc đơn), thì các chuỗi tương ứng trong mỗi nhóm được biểu diễn như **\$1** cho nhóm đầu tiên, \$2 cho nhóm thứ 2

```
>>> s.replace(/[A-Z]/g, "__$1");
```

```
"_Hello_Java_Script_World"
```

Hình dung rằng ta có 1 chuỗi đăng ký từ trên trang web và yêu cầu địa chỉ email, username, password. người dùng điền vào email của họ và sau đó JS kích hoạt và đưa ra gợi ý về username :

```
>>> var email = "stoyan@phpied.com";
```

```
>>> var username = email.replace(/(.*)@.*/, "$1");
```

```
>>> username;
```

```
"stoyan"
```

4.3.3. Các hàm Replace callbacks

Khi ta định nghĩa chuỗi thay thế, ta cũng có thể chuyển tiếp 1 hàm function mà nó trả về 1 chuỗi string, điều này mang tới cho ta khả năng thực thi bất cứ điều kiện logic đặc biệt nào mà ta cần trước khi định nghĩa các chuỗi thay thế

```
>>> function replaceCallback(match){return "_" +
```

```
match.toLowerCase();}
```

```
>>> s.replace(/[A-Z]/g, replaceCallback);
```

```
"_hello_java_script_world"
```

Hàm callback sẽ nhận về 1 số các tham số :

- Tham số đầu tiên chính là chuỗi tương ứng
- Tham số cuối cùng là chuỗi string đang được tìm kiếm
- Tham số trước tham số cuối cùng là vị trí của chuỗi tương ứng
- Các Tham số còn lại chứa bất cứ chuỗi string nào tương ứng với bất cứ nhóm nào trong mẫu regex

Để kiểm tra điều này, ta tạo ra 1 biến để lưu trữ toàn bộ mảng các tham số được chuyển tiếp qua hàm callback :

```
>>> var glob;
```

Tiếp theo, ta sẽ định nghĩa 1 biểu thức chính quy mà có 3 nhóm và tương ứng với địa chỉ email trong định dạng : something@something.something:

```
>>> var re = /(.*)(.*)\.(.*)/;
```

Cuối cùng ta sẽ định nghĩa 1 hàm callback để lưu trữ các tham số trong glob và sau đó trả về trong chuỗi thay thế

```
var callback = function(){  
  
  glob = arguments;  
  
  return arguments[1] + ' at ' + arguments[2] + ' dot ' +  
  
  arguments[3];  
  
}
```

Sau đó ta có thể gọi hàm như sau :

```
>>> "stoyan@phpied.com".replace(re, callback);
```

```
"stoyan at phpied dot com"
```

```
>>> glob
```

```
["stoyan@phpied.com", "stoyan", "phpied", "com", 0,
```

```
"stoyan@phpied.com"]
```

4.3.4.split()

như ta đã biết về phương thức split(), được dùng để tạo ra 1 mảng array từ 1 chuỗi string đầu vào và 1 kí tự xác định. ví dụ ta cắt chuỗi sau thành các chuỗi con bởi dấu phẩy :

```
>>> var csv = 'one, two,three ,four';
```

```
>>> csv.split(',');
```

```
["one", " two", "three ", "four"]
```

Bởi vì chuỗi đầu vào bao gồm các khoảng không nhất quán trước vào sau dấu phẩy, do đó kết quả mảng array trả về cũng có các khoảng trống. với 1 định thức chính quy, ta có thể sửa lại điều này bằng cách sử dụng **\s***, có nghĩa là “không có hoặc nhiều hơn 1 khoảng trắng”

```
>>> csv.split((\s*,\s*/))
```

```
["one", "two", "three", "four"]
```

4.3.5.Cách chuyển tiếp 1 chuỗi string khi 1 regexp được như kì vọng

1 điều cuối cùng cần phải chú ý tới là 4 phương thức mà ta thấy ở trên (split(), match(), search(), and replace()) cũng có thể có những chuỗi strings đối lập với các biểu thức chính quy. trong trường hợp này, tham số chuỗi string được sử dụng để sản xuất ra 1 đối tượng regex mới nếu nó được chuyển tiếp vào new RegExp()

Ví dụ chuyển tiếp 1 chuỗi vào phương thức replace :

```
>>> "test".replace('t', 'r')
```

```
"rest"
```

Hay như ở trên :

```
>>> "test".replace(new RegExp('t'), 'r')
```


"rest"

Khi ta chuyển tiếp 1 chuỗi string, thì ta có thể không thiết lập các hiệu chỉnh như những gì ta có thể làm với các chuỗi regex thông thường

4.4. Quy tắc mẫu trong biểu thức quan hệ

Mẫu	Thông tin mô tả
[abc]	<p>Tìm kiếm bất cứ kí tự nào nằm trong ngoặc vuông. ví dụ :</p> <pre>>>> "some text".match(/[otx]/g) ["o", "t", "x", "t"]</pre> <p>Nó sẽ tìm tất cả các kí tự o,t,x rồi tách riêng nó ra thành từng phần tử trong chuỗi string</p>
[a-z]	<p>Tìm bất cứ kí tự nào trong khoảng kí tự viết thường từ a đến z. ví dụ như :</p> <ul style="list-style-type: none">• [a-d] thì cũng tương ứng với [abcd]• [a-z] tương ứng với tất cả các kí tự viết thường• [a-zA-Z0-9_] tương ứng với toàn bộ các kí tự (text, số và dấu gạch dưới)• <pre>>>> "Some Text".match(/[a-z]/g) ["o", "m", "e", "e", "x", "t"] >>> "Some Text".match(/[a-zA-Z]/g) ["S", "o", "m", "e", "T", "e", "x", "t"]</pre>
[^abc]	<p>Tìm bất cứ kí tự nào không thuộc các kí tự ở trong ngoặc vuông</p>

	<pre>>>> "Some Text".match(/^[a-z]/g) ["S", " ", "T"]</pre> <p>Tìm kiếm các kí tự không phải là các kí tự viết thường từ a đến z</p>
a b	<p>Tìm kí tự a hoặc b tương tự mệnh đề OR</p> <p>Ví dụ :</p> <pre>>>> "Some Text".match(/t T/g); ["T", "t"] >>> "Some Text".match(/t T Some/g); ["Some", "T", "t"]</pre>
a(?=b)	<p>Tương ứng với a nếu nó được theo sau là b</p> <p>Ví dụ :</p> <pre>>>> "Some Text".match(/Some(?!Tex)/g); null >>> "Some Text".match(/Some(?! Tex)/g); ["Some"]</pre> <p>Tìm kiếm 1 chuỗi Some trong chuỗi string mà theo sau chuỗi đó là “Tex” hay “Tex” (chú ý từ phía sau có chứa 1 khoảng trắng ở phía trước)</p>
a(?!b)	<p>Tương ứng là a mà chuỗi theo sau nó không phải là b</p> <p>Ví dụ :</p> <pre>>>> "Some Text".match(/Some(?! Tex)/g); null</pre>

	>>> "Some Text".match(/Some(?!Tex)/g); ["Some"]
\	Là kí tự thoát được dùng để định dạng các chuỗi kí tự đặc biệt trong mẫu
\n	<p>Sử dụng để tìm kí tự xác định dòng mới</p> <p>Ví dụ :</p> <pre>var str="Visit W3Schools.\nLearn Javascript."; var patt1=/\n/;</pre> <p>Visit W3Schools.\nLearn Javascript.</p> <p>Thường được dùng với hàm search() để trả về số thứ tự của kí tự xuống dòng</p> <p>Ví dụ :</p> <pre>var str="Visit W3Schools.\n Learn JavaScript."; var patt1=/\n/g; document.write(str.search(patt1));</pre> <p>16</p>
\s	<p>Tìm 1 kí tự khoảng trắng hay 5 chuỗi thoát kí tự (\f, \n, \r, \t, \v)</p> <p>Ví dụ :</p> <pre>>>> "R2\n D2".match(/\s/g) ["\n", " "]</pre>
\S	Trái ngược lại với trên, tìm tất cả các kí tự

	<p>không phải là khoảng trắng hay chuỗi thoát kí tự hay nó tương đương với <code>[^\s]</code></p> <p>Ví dụ :</p> <pre>>>> "R2\n D2".match(/S/g) ["R", "2", "D", "2"]</pre>
<code>\w</code>	<p>Tìm kiếm bất cứ kí tự thuộc kiểu chữ, số, dấu gạch ngang</p> <p>Ví dụ :</p> <pre>>>> "Some text!".match(/w/g) ["S", "o", "m", "e", "t", "e", "x", "t"]</pre>
<code>\W</code>	<p>Trái ngược lại <code>\w</code>, tìm kiếm các kí tự đặc biệt</p> <p>Ví dụ :</p> <pre>>>> "Some text!".match(/W/g) [" ", "!", " "]</pre>
<code>\d</code>	<p>Tìm kiếm các kí tự kiểu số như <code>[0-9]</code></p> <p>Ví dụ :</p> <pre>>>> "R2-D2 and C-3PO".match(/d/g) ["2", "2", "3"]</pre>
<code>\D</code>	<p>Trái ngược với <code>\d</code>, tìm kiếm tất cả các kí tự trừ kiểu số hay như <code>^[^0-9]</code> và <code>[^\d]</code></p> <p>Ví dụ :</p> <pre>>>> "R2-D2 and C-3PO".match(/D/g) ["R", "-", "D", " ", "a", "n", "d", " ", "C", "-", "P", "O"]</pre>

<code>\b</code>	<p>Tìm kiếm kí tự ở bắt đầu hay kết thúc của 1 từ</p> <p>Ví dụ :</p> <pre>var str="Visit W3Schools"; var patt1=/\bW3/g;</pre> <p>Visit W3Schools</p> <pre>>>> "R2D2 and C-3PO".match(/[RD]2/g) ["R2", "D2"]</pre> <p>Tìm kiếm các kí tự ở cuối 1 từ</p> <pre>>>> "R2D2 and C-3PO".match(/[RD]2\b/g) ["D2"]</pre> <p>Khi chuỗi đầu vào có dấu gạch ngang ở giữa – thì từ có chứa dấu gạch ngang bị phân tách thành 2 từ riêng biệt</p> <pre>>>> "R2-D2 and C-3PO".match(/[RD]2\b/g) ["R2", "D2"]</pre>
<code>\B</code>	<p>Ngược với <code>\b</code> tìm kiếm các kí tự không nằm ở đầu hay ở cuối của từ</p> <p>Ví dụ :</p> <pre>>>> "R2-D2 and C-3PO".match(/[RD]2\B/g) null >>> "R2D2 and C-3PO".match(/[RD]2\B/g) ["R2"]</pre>

[\\b]	Tìm các kí tự gạch lùi
\\0	Tìm kí tự null
^	<p>Tìm các kí tự ở vị trí bắt đầu chuỗi. nếu ta thiết lập hiệu chỉnh m, thì nó sẽ ở vị trí bắt đầu mỗi dòng</p> <p>Ví dụ :</p> <pre>>>> "regular\\nregular\\nexpression".match(/r/g); ["r", "r", "r", "r", "r"] >>> "regular\\nregular\\nexpression".match(/^r/g); ["r"] >>> "regular\\nregular\\nexpression".match(/^r/mg); ["r", "r"]</pre>
\$	<p>Tìm các kí tự ở cuối chuỗi, nếu sử dụng hiệu chỉnh m, thì nó sẽ ở vị trí kết thúc mỗi dòng</p> <p>Ví dụ :</p> <pre>>>> "regular\\nregular\\nexpression".match(/r\$/g); null >>> "regular\\nregular\\nexpression".match(/r\$/mg); ["r", "r"]</pre>
.	<p>Tìm 1 kí tự đơn ngoại trừ dòng mới hay kết thúc dòng</p> <p>Ví dụ :</p>

	<pre>var str="That's hot!"; var patt1=/h.t/g; That's hot! >>> "regular".match(/r./g); ["re"] >>> "regular".match(/r.../g); ["regu"]</pre>
n*	<p>Tìm kiếm chuỗi kí tự được tìm kiếm theo mẫu n phía trước mà nó xảy ra 0 hay nhiều lần</p> <p>Ví dụ :</p> <p>./.*/ sẽ tương ứng với bất cứ thứ gì ngoại trừ chuỗi rỗng</p> <pre>>>> "".match(/.*/) [""] >>> "anything".match(/.*/) ["anything"] >>> "anything".match(/n.*h/) ["nyth"]</pre>
n?	<p>Tìm kiếm chuỗi kí tự được tìm kiếm theo mẫu n phía trước mà nó xảy ra 0 hay 1 lần</p> <p>Ví dụ :</p> <pre>>>> "anything".match(/ny?/g) ["ny", "n"]</pre>

V.Prototype

5.1.Thuộc tính prototype

Các hàm function trong JS là các đối tượng và chúng chứa các phương thức và thuộc tính. 1 trong các phương thức mà ta đã làm quen là apply() và call() và 1 vài thuộc tính là length và constructor. 1 thuộc tính khác là prototype

Nếu ta định nghĩa 1 hàm đơn giản foo() thì ta có thể truy cập thuộc tính của nó như bất cứ 1 đối tượng nào khác :

```
>>> function foo(a, b){return a * b;}
```

```
>>> foo.length
```

```
2
```

```
>>> foo.constructor
```

```
Function()
```

Prototype là 1 thuộc tính mà được tạo ra vào lúc như ta định nghĩa hàm function. giá trị khởi tạo ban đầu của nó là 1 đối tượng rỗng :

```
>>> typeof foo.prototype
```

```
"object"
```

```
>>> foo.prototype = {}
```

Ta có thể bổ sung đối tượng rỗng này với các phương thức và thuộc tính. chúng sẽ không gây ảnh hưởng gì tới chính hàm foo(); chúng chỉ được sử dụng nếu ta sử dụng hàm foo() như 1 hàm tạo

5.2.Cách thêm các phương thức và thuộc tính bằng cách sử dụng Prototype

Ta xét ví dụ về 1 hàm tạo Gadget() dùng để thêm 2 thuộc tính và 1 phương thức vào các đối tượng mà nó tạo ra :

```
function Gadget(name, color) {
```

```
  this.name = name;
```



```
this.color = color;

this.whatAreYou = function(){

return 'I am a ' + this.color + ' ' + this.name;

}

}
```

Để thêm các phương thức và thuộc tính vào thuộc tính prototype của hàm tạo thì nó là cách khác để bổ sung thêm các đối tượng vào hàm tạo này

Ta thêm 2 thuộc tính price và rating, phương thức getInfo().do prototype chứa 1 đối tượng, thì ta có thể chỉ cần thêm vào nó giống như sau :

```
Gadget.prototype.price = 100;

Gadget.prototype.rating = 3;

Gadget.prototype.getInfo = function() {

return 'Rating: ' + this.rating + ', price: ' + this.price;

};
```

Để thay thế cho việc thêm vào đối tượng prototype, có 1 cách khác cũng nhận được kết quả giống như trên là ta ghi đè hoàn toàn vào thuộc tính prototype, và thay thế nó với 1 đối tượng mà ta chọn :

```
Gadget.prototype = {

price: 100,

rating: 3,

getInfo: function() {

return 'Rating: ' + this.rating + ', price: ' + this.price;

}

}
```

```
};
```

5.3.Cách sử dụng các thuộc tính và phương thức của Prototype

Tất cả các phương thức và thuộc tính mà ta thêm vào prototype thì ngay lập tức sẵn sàng thể hiện trong các đối tượng mới mà ta tạo bằng cách sử dụng hàm tạo. nếu ta tạo 1 đối tượng newtoy object bằng cách sử dụng hàm tạo Gadget(), thì ta có thể nhận tất cả các phương thức và thuộc tính mà ta vừa định nghĩa :

```
>>> var newtoy = new Gadget('webcam', 'black');
```

```
>>> newtoy.name;
```

```
"webcam"
```

```
>>> newtoy.color;
```

```
"black"
```

```
>>> newtoy.whatAreYou();
```

```
"I am a black webcam"
```

```
>>> newtoy.price;
```

```
100
```

```
>>> newtoy.rating;
```

```
3
```

```
>>> newtoy.getInfo();
```

```
"Rating: 3, price: 100"
```

Chú ý rằng : thuộc tính prototype luôn biến đổi. các đối tượng được chuyển tiếp bởi tham chiếu trong JS và do vậy prototype không phải là sao chép của bất cứ thực thể đối tượng mới nào

Điều này thì có nghĩa gì trong thực tế ?

Nó có nghĩa là ta có thể chỉnh sửa prototype vào bất cứ lúc nào và tất cả các đối tượng (ngay cả các đối tượng đã được tạo ra trước khi chỉnh sửa) thì **sẽ kế thừa những sự thay đổi này**

Giờ ta thực hiện tiếp với ví dụ trên, thêm vào 1 phương thức mới vào prototype:

```
Gadget.prototype.get = function(what) {  
  
    return this[what];  
  
};
```

Mặc dù **newtoy** được tạo ra trước khi phương thức get() được định nghĩa, thì newtoy sẽ vẫn nhận phương thức mới :

```
>>> newtoy.get('price');
```

```
100
```

```
>>> newtoy.get('color');
```

```
"black"
```

5.4.Các thuộc tính chính gốc so với các thuộc tính được thêm vào qua prototype

Trong ví trên thì getInfo() được sử dụng **this** nội tại để xác định đối tượng.nó cũng có thể sử dụng **Gadget.prototype** để nhận về kết quả như trên :

```
Gadget.prototype.getInfo = function() {  
  
    return 'Rating: ' + Gadget.prototype.rating + ', price: ' + Gadget.  
    prototype.price;  
  
};
```

Vậy điểm khác biệt ở đây là gì ?

Để trả lời câu hỏi này, ta xét cách thức hoạt động của prototype 1 cách chi tiết hơn

```
>>> var newtoy = new Gadget('webcam', 'black');
```

Khi ta thử truy cập 1 thuộc tính của newtoy, thì **newtoy.name** sẽ tìm kiếm thông qua tất cả các thuộc tính của đối tượng và tìm kiếm 1 thuộc tính được gọi là **name** nếu nó tìm thấy nó thì nó sẽ trả về giá trị :

```
>>> newtoy.name
```

```
"webcam"
```

Vậy khi ta truy cập thuộc tính **rating** ?

Thì JS sẽ xem xét tất cả các thuộc tính của newtoy và nếu nó không tìm được cái nào gọi là **rating** thì JS sẽ xác định thuộc tính prototype trong hàm tạo để sử dụng tạo ra đối tượng này (tương tự như ta viết newtoy.constructor.prototype).nếu thuộc tính được tìm thấy trong prototype, thì thuộc tính được sử dụng :

```
>>> newtoy.rating
```

```
3
```

Điều này cũng giống như ta truy cập prototype 1 cách trực tiếp.mọi đối tượng đều có thuộc tính tạo constructor mà nó tham chiếu tới 1 hàm function dùng để tạo ra đối tượng

```
>>> newtoy.constructor
```

```
Gadget(name, color)
```

```
>>> newtoy.constructor.prototype.rating
```

```
3
```

Giờ ta nhìn vào các bước tiếp theo.mọi đối tượng có 1 hàm tạo.thuộc tính prototype cũng là 1 đối tượng, do vậy nó cũng phải có 1 hàm tạo

```
>>> newtoy.constructor.prototype.constructor
```

```
Gadget(name, color)
```

```
>>> newtoy.constructor.prototype.constructor.prototype
```

```
Object price=100 rating=3
```

5.5. Ghi đè thuộc tính của prototype với thuộc tính chính gốc

Như ví dụ minh họa ở trên, nếu 1 trong các đối tượng của ta không có 1 thuộc tính đã biết của chính nó, thì nó có thể sử dụng 1 thuộc tính (nếu nó tồn tại) ở bên trong prototype.

Chuyện gì nếu đối tượng có thuộc tính của chính nó và prototype cũng có 1 thuộc tính trùng tên với thuộc tính đó ?

Câu trả lời là thuộc tính của chính nó được ưu tiên hơn so với thuộc tính trong prototype

Ta xét ví dụ khi 1 thuộc tính cũng tồn tại ở cả thuộc tính chính gốc và thuộc tính của đối tượng prototype :

```
function Gadget(name) {
```

```
  this.name = name;
```

```
}
```

```
Gadget.prototype.name = 'foo';
```

```
"foo"
```

Tạo ra 1 đối tượng và truy cập vào **thuộc tính name** thì nó sẽ lấy ra **thuộc tính name** chính gốc

```
>>> var toy = new Gadget('camera');
```

```
>>> toy.name;
```

```
"camera"
```

Nếu ta xóa đi thuộc tính này, thì thuộc tính của prototype có cùng tên mới được biểu hiện ra

```
>>> delete toy.name;
```

```
true
```

```
>>> toy.name;
```

```
"foo"
```

Tất nhiên, ta có thể luôn luôn tạo lại thuộc tính chính gốc :

```
>>> toy.name = 'camera';
```

```
>>> toy.name;
```

```
"camera"
```

5.6.Các liệt kê các thuộc tính

Nếu ta muốn liệt kê tất cả các thuộc tính của 1 đối tượng thì ta có thể sử dụng 1 vòng lặp **for-in**

```
var o = {p1: 1, p2: 2};
```

```
for (var i in o) {
```

```
    console.log(i + '=' + o[i]);
```

```
}
```

Kết quả :

```
p1=1
```

```
p2=2
```

có 1 vài chi tiết ta phải để ý tới là :

- Không phải tất cả các thuộc tính được hiển thị ra trong 1 vòng lặp **for-in**. ví dụ như thuộc tính `length` (cho các đối tượng mảng arrays) và thuộc tính `constructor` sẽ không được hiển thị. các thuộc tính mà được hiển thị ra gọi là **enumerable (đếm được)**. ta có thể kiểm tra những thuộc tính là **enumerable (đếm được)** với sự giúp đỡ của phương thức **propertyIsEnumerable()** mà mọi đối tượng đều hỗ trợ
- Các thuộc tính trong prototype cũng được hiển thị như thể chúng cũng là các **enumerable (đếm được)** .ta có thể kiểm tra nếu 1 thuộc tính là thuộc tính chính gốc với thuộc tính của prototype bằng cách sử dụng phương thức **hasOwnProperty()**
- Phương thức **propertyIsEnumerable()** sẽ trả về `false` cho tất cả các thuộc tính của prototype, mặc dù chúng vẫn là các **enumerable (đếm được)** và được hiển thị trong vòng lặp **for-in**

Ví dụ :

```
function Gadget(name, color) {  
  
    this.name = name;  
  
    this.color = color;  
  
    this.someMethod = function(){return 1;}  
  
}
```

```
Gadget.prototype.price = 100;
```

```
Gadget.prototype.rating = 3;
```

Tạo ra 1 đối tượng mới :

```
var newtoy = new Gadget('webcam', 'black');
```

giờ đây ta sử dụng vòng lặp for-in, ta có thể thấy tất cả các thuộc tính của đối tượng, bao gồm cả các thuộc tính chứa trong prototype :

```
for (var prop in newtoy) {  
  
    console.log(prop + ' = ' + newtoy[prop]);  
  
}
```

Kết quả cũng bao gồm các phương thức của đối tượng (xét cho cùng các phương thức cũng chỉ là các đối tượng mà nó thực thi giống như các hàm function)

```
name = webcam
```

```
color = black
```

```
someMethod = function () { return 1; }
```

```
price = 100
```

```
rating = 3
```

nếu ta muốn phân biệt rõ giữa các thuộc tính chính gốc với thuộc tính của prototype, thì ta phải sử dụng phương thức **hasOwnProperty()** :

```
>>> newtoy.hasOwnProperty('name')
```

true

```
>>> newtoy.hasOwnProperty('price')
```

False

Ta chỉnh sửa lại vòng lặp và chỉ hiện thị ra các thuộc tính chính gốc :

```
for (var prop in newtoy) {  
  
  if (newtoy.hasOwnProperty(prop)) {  
  
    console.log(prop + '=' + newtoy[prop]);  
  
  }  
  
}
```

Kết quả :

name=webcam

color=black

someMethod=function () { return 1; }

giờ ta thử với phương thức `propertyIsEnumerable()`. phương thức này sẽ trả về true cho các thuộc tính chính gốc mà không phải được xây dựng sẵn :

```
>>> newtoy.propertyIsEnumerable('name')
```

True

Đa số các thuộc tính được xây dựng sẵn không phải là **enumerable (đếm được)**

```
>>> newtoy.propertyIsEnumerable('constructor')
```

False

Bất cứ thuộc tính nào trong prototype cũng không phải là **enumerable (đếm được)** :

```
>>> newtoy.propertyIsEnumerable('price')
```

False

Chú ý, tuy nhiên các thuộc tính như vậy là đếm được nếu ta gọi đối tượng chứa trong prototype và gọi **propertyIsEnumerable()** của nó

```
>>> newtoy.constructor.prototype.propertyIsEnumerable('price')
```

True

5.7.isPrototypeOf()

mọi đối tượng objects đều có phương thức **isPrototypeOf()**. phương thức này thông báo cho ta đối tượng cụ thể được sử dụng như 1 thuộc tính prototype của đối tượng khác :

nhìn vào 1 đối tượng đơn giản sau :

```
var monkey = {  
  
  hair: true,  
  
  feeds: 'bananas',  
  
  breathes: 'air'  
  
};
```

Giờ ta tạo ra 1 hàm tạo **Human()** và thiết lập thuộc tính **prototype** chỉ tới **monkey**

```
function Human(name) {  
  
  this.name = name;  
  
}
```

```
Human.prototype = monkey;
```

Giờ nếu ta tạo ra 1 đối tượng **Human** được gọi là **George** và đặt ra câu hỏi “ prototype của **George** có phải là **monkey** “

Câu trả lời là đúng vậy

```
>>> var george = new Human('George');
```

```
>>> monkey.isPrototypeOf(george)
```

true

5.8.Bí mật về __proto__ Link

như ta đã biết, thuộc tính prototype sẽ được quan tâm đến khi ta thử truy cập 1 thuộc tính mà nó không tồn tại trong đối tượng hiện tại

như ví dụ trước ta có 1 đối tượng được gọi là **monkey** và sử dụng nó giống như là 1 thuộc tính prototype khi ta tạo ra các đối tượng objects với hàm tạo **Human()**

```
var monkey = {  
  feeds: 'bananas',
```

```
  breathes: 'air'
```

```
};
```

```
function Human() {}
```

```
Human.prototype = monkey;
```

Giờ ta tạo ra 1 đối tượng **developer** và đưa vào nó 1 vài thuộc tính :

```
var developer = new Human();
```

```
developer.feeds = 'pizza';
```

```
developer.hacks = 'JavaScript';
```

giờ ta xét 1 vài thuộc tính.như **hacks** là 1 thuộc tính của đối tượng **developer**

```
>>> developer.hacks
```

```
"JavaScript"
```

Thuộc tính feeds cũng là thuộc tính của đối tượng **developer**

```
>>> developer.feeds
```

```
"pizza"
```

Thuộc tính breathes không phải là 1 thuộc tính đã tồn tại trong đối tượng developer, do vậy prototype được tìm kiếm và có 1 đường link đặc biệt chỉ tới đối tượng prototype này :

```
>>> developer.breathes
```

"air"

Điều này chỉ ra rằng link bí mật liên kết tới prototype vẫn tồn tại. link bí mật này được lộ ra trong firefox là phần **__proto__ property** (từ “proto” với 2 gạch dưới trước và sau 2 từ any)

```
>>> developer.__proto__
```

Object feeds=bananas breathes=air

Ta có thể sử dụng thuộc tính bí mật này cho các mục đích học tập nhưng nó không phải là ý tưởng hay để sử dụng trong các đoạn mã thực tế bởi vì nó không tương thích với Internet Explorer, do vậy đoạn mã trên không linh hoạt. ví dụ, ta nói rằng ta đã tạo ra 1 số các đối tượng objects với monkey như là 1 prototype và giờ ta muốn thay đổi 1 vài thứ trong tất cả các đối tượng. ta có thể thay đổi monkey và tất cả các thực thể sẽ kế thừa sự thay đổi này :

```
>>> monkey.test = 1
```

1

```
>>> developer.test
```

1

__proto__ không giống với thuộc tính prototype. **__proto__** là 1 thuộc tính trong các **thực thể**, trái lại **prototype** là 1 thuộc tính trong **các hàm tạo**

```
>>> typeof developer.__proto__
```

"object"

```
>>> typeof developer.prototype
```

"undefined"

Developer là 1 thực thể của hàm tạo **Human**

Nhắc lại 1 lần nữa, ta chỉ nên sử dụng **__proto__** cho việc học và sửa lỗi

5.9.Cách làm gia tăng các đối tượng được xây dựng sẵn

Các đối tượng được xây dựng sẵn như các đối tượng Array, String, Object và Function cũng có thể được gia tăng thông qua các prototypes của chúng, điều này có nghĩa là ta có thể thêm 1 phương thức mới vào Array prototype và theo cách này làm nó có tác dụng tới toàn bộ mảng arrays.

Trong php có 1 hàm gọi là **in_array()** nó sẽ thông báo cho ta biết 1 giá trị value có tồn tại trong 1 array hay không. trong JS, không có phương thức **inArray()** nào như vậy, do vậy ta thực thi và thêm nó vào **Array.prototype**

```
Array.prototype.inArray = function(needle) {  
  
  for (var i = 0, len = this.length; i < len; i++) {  
  
    if (this[i] === needle) {  
  
      return true;  
  
    }  
  
  }  
  
  return false;  
  
}
```

Giờ tất cả các mảng arrays sẽ có 1 phương thức mới

```
>>> var a = ['red', 'green', 'blue'];
```

```
>>> a.inArray('red');
```

```
true
```

```
>>> a.inArray('yellow');
```

```
False
```

Đây là cách rất hay và dễ dàng ! thực hiện nó lại 1 lần nữa, và hình dung rằng ứng dụng của ta thường cần đảo lại chuỗi strings và ta nhận thấy nên xây dựng sẵn 1 phương thức reverse() cho các đối tượng string. sau tất cả, arrays có reverse(), ta có thể dễ dàng thêm phương thức reverse() này vào trong String prototype bằng cách vay mượn Array.prototype.reverse()

```
String.prototype.reverse = function() {  
  
return Array.prototype.reverse.apply(this.split('')).join("");  
  
}
```

Đoạn mã code sử dụng split() để tạo ra 1 mảng array từ 1 chuỗi string, sau đó gọi phương thức reverse() trên mảng này và phương thức này sẽ tạo ra 1 mảng đảo ngược. kết quả được trả về dạng string bằng cách sử dụng join()

```
>>> "Stoyan".reverse();  
  
"nayotS"
```

5.10. Một vài ghi chú về Prototype

Có 2 cách thực thi thú vị được xét tới khi ta làm việc với prototype :

- Thuộc tính prototype sống với 1 ngoại lệ là khi ta **hoàn toàn thay thế đối tượng prototype**
- prototype.constructor là không đảm bảo

cách tạo 1 hàm tạo đơn giản và 2 đối tượng :

```
>>> function Dog(){this.tail = true;}  
  
>>> var benji = new Dog();  
  
>>> var rusty = new Dog();
```

Ngay sau khi ta tạo ra các đối tượng trên, ta vẫn có thể thêm vào các thuộc tính vào trong prototype và các đối tượng sẽ nhận các thuộc tính mới

```
>>> Dog.prototype.say = function(){return 'Woof!';}
```

Cả 2 đối tượng đều nhận phương thức mới :

```
>>> benji.say();  
  
"Woof!"  
  
>>> rusty.say();
```

"Woof!"

Tính tới thời điểm này nếu ta để ý tới các đối tượng, và yêu cầu xem hàm tạo được sử dụng để tạo ra chúng, thì chúng sẽ được báo cáo 1 cách chính xác

```
>>> benji.constructor;
```

Dog()

```
>>> rusty.constructor;
```

Dog()

1 điều phải chú ý rằng nếu ta đòi hỏi cái gì là hàm tạo của đối tượng prototype thì ta cũng sẽ lấy về là Dog(), điều này là không chính xác bởi prototype chỉ là 1 đối tượng thông thường được tạo bởi Object(). nó không có bất cứ thuộc tính nào của 1 đối tượng được tạo bởi Dog()

```
>>> benji.constructor.prototype.constructor
```

Dog()

```
>>> typeof benji.constructor.prototype.tail
```

"undefined"

Giờ ta xét việc ghi đè hoàn toàn đối tượng prototype với 1 đối tượng mới (nghĩa là ta gán prototype là luôn 1 đối tượng object):

```
>>> Dog.prototype = { paws: 4, hair: true };
```

Nó chỉ ra rằng **các đối tượng cũ không nhận** được các thuộc tính của **prototype mới** mà chúng vẫn giữ link bí mật chỉ tới đối tượng prototype cũ

```
>>> typeof benji.paws
```

"undefined"

```
>>> benji.say()
```

"Woof!"

```
>>> typeof benji.__proto__.say
```

"function"

```
>>> typeof benji.__proto__.paws
```

"undefined"

Bất kỳ đối tượng mới nào được tạo ra vào thời điểm hiện tại **sẽ được cập nhật prototype** :

```
>>> var lucy = new Dog();
```

```
>>> lucy.say()
```

TypeError: lucy.say is not a function

```
>>> lucy.paws
```

4

Khi này The secret __proto__ link chỉ tới đối tượng prototype mới :

```
>>> typeof lucy.__proto__.say
```

"undefined"

```
>>> typeof lucy.__proto__.paws
```

"number"

Giờ thuộc tính tạo constructor của các đối tượng mới sẽ **không còn được báo cáo đúng nữa.thay vì nó phải chỉ tới Dog() thì nó lại chỉ tới Object().**

```
>>> lucy.constructor
```

Object()

```
>>> benji.constructor
```

Dog()

Phần ta hay nhầm lẫn nhiều nhất là khi ta tìm kiếm hàm tạo của prototype :

```
>>> typeof lucy.constructor.prototype.paws
```

"undefined"

```
>>> typeof benji.constructor.prototype.paws
```

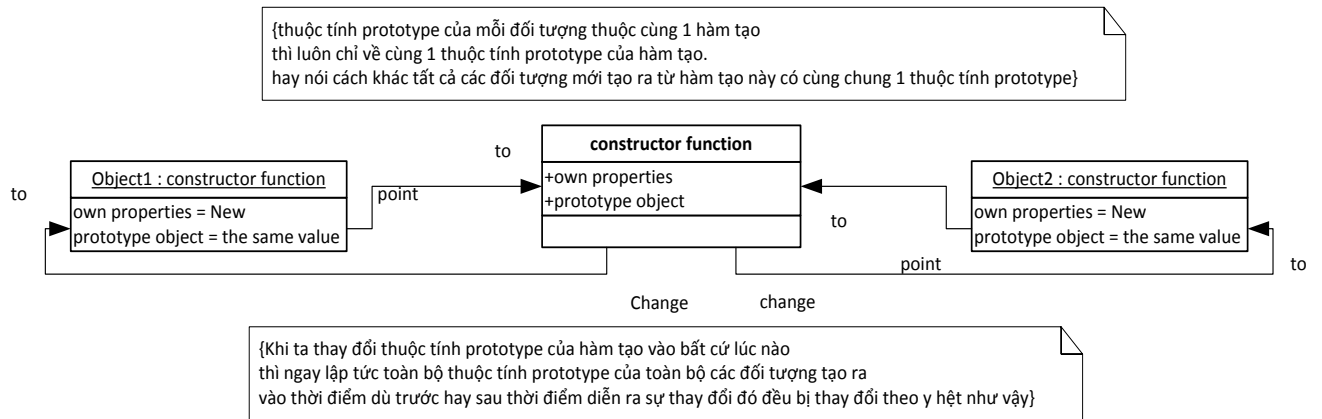
```
"number"
```

Theo đó ta có thể sửa lại các trạng thái xử lý không như mong đợi ở trên bằng cách :

```
>>> Dog.prototype = {paws: 4, hair: true};
```

```
>>> Dog.prototype.constructor = Dog;
```

Chú ý : khi ta ghi đè lên prototype, thì đây là ý tưởng hay để reset lại thuộc tính tạo constructor



VI. Sự kế thừa

Như ta đã biết rằng không có phương pháp kiểu class nào trong JS, mặc dù vậy ta có thể giả lập chúng với các hàm tạo constructor functions.

Sự đóng gói encapsulation ?

Đúng vậy, các đối tượng objects gói gọn cả dữ liệu và phương thức được sử dụng thao tác trên dữ liệu.

Sự kết tụ Aggregation ?

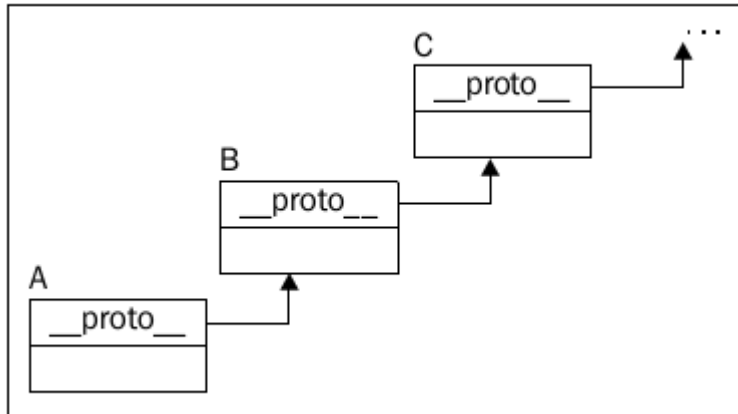
Chắc chắn, 1 đối tượng có thể chứa các đối tượng khác. thực tế, thì đây luôn là trường hợp mà trong đó các phương thức chính là các hàm functions và các hàm functions cũng là các đối tượng objects.

6.1. Các chuỗi móc nối prototype

Ta bắt đầu với cách mặc định chính là nhờ cách thực thi sự kế thừa – chuỗi móc nối kế thừa được thể hiện thông qua prototype

Như ta đã biết, mọi hàm function đều có 1 thuộc tính prototype thuộc tính mà chứa 1 đối tượng trong nó. khi hàm function này được gọi bằng cách sử dụng 1 toán tử new, thì 1 đối tượng object được tạo ra và đối tượng này có 1 secret link chỉ tới đối tượng prototype. secret link (hay còn được gọi là __proto__) cho phép các phương thức và thuộc tính của đối tượng prototype được sử dụng như thể chúng thuộc về đối tượng mới vừa mới tạo

Đối tượng prototype chỉ là 1 đối tượng chính qui như thông thường và do đó nó cũng bao gồm 1 liên kết link tới thuộc tính prototype của nó và do vậy 1 chuỗi móc nối được tạo và nó được gọi là 1 chuỗi móc nối prototype



Trong ví dụ minh họa này, 1 đối tượng A chứa 1 các thuộc tính. 1 trong các thuộc tính là thuộc tính ẩn `__proto__` - thuộc tính mà chỉ tới 1 đối tượng khác như đối tượng B. thuộc tính `__proto__` của B chỉ tới C. chuỗi móc nối này kết thúc với đối tượng Object object, đây là đối tượng có mức cha cao nhất, và mọi đối tượng khác được kế thừa từ nó

Về mặt thực tiễn thì điều này có nghĩa, khi 1 đối tượng A thiếu 1 thuộc tính và đối tượng B có nó, thì đối tượng A vẫn có thể truy cập tới thuộc tính này như chính thuộc tính của nó

Ta sẽ xem xét 2 ví dụ về cách sử dụng sự kế thừa như sau : 1 vật thể cha chung được kế thừa bởi vật thể 2D - (vật thể 2 chiều như hình tam giác, hình chữ nhật)

6.2. Ví dụ về chuỗi móc nối prototype

Để thực thi sự phân cấp, ta định nghĩa 3 hàm tạo :

```
function Shape(){

this.name = 'shape';

this.toString = function() {return this.name;};

}

function TwoDShape(){

this.name = '2D shape';

}

function Triangle(side, height) {
```

```
this.name = 'Triangle';  
  
this.side = side;  
  
this.height = height;  
  
this.getArea = function(){return this.side * this.height / 2;};  
  
}
```

Đoạn mã sau thực thi sự phân cấp :

```
TwoDShape.prototype = new Shape();  
  
Triangle.prototype = new TwoDShape();
```

Chuyện gì xảy ra ở đây ?

ta lấy đối tượng chứa trong thuộc tính prototype của TwoDShape và để thay thế việc thêm vào các thuộc tính riêng lẻ vào bên trong đối tượng này, ta hoàn toàn có thể ghi đè nó với các đối tượng khác, được tạo bởi các gọi hàm tạo shape() với toán tử new.

Tương tự với Triangle : prototype của nó được thay thế bằng 1 đối tượng được tạo ra bởi new TwoDShape().ta cần tạo ra 1 thực thể bằng cách sử dụng hàm tạo new Shape() và sau đó ta có thể kế thừa các thuộc tính của đối tượng này (ta không kế thừa trực tiếp từ hàm tạo Shape())

Thêm vào đó, sau khi kế thừa, ta có thể chỉnh sửa Shape(), ghi đè nó hay ngay cả xóa bỏ nó và điều này sẽ không gây ảnh hưởng tới TwoDShape, bởi vì tất cả những gì ta cần chỉ là 1 thực thể để kế thừa từ thực thể đó

Như ta đã biết ở chương trước khi ta ghi đè hoàn toàn prototype, **điều này có 1 tác động phụ tiêu cực lên thuộc tính tạo constructor**.do đó, ý tưởng tốt để reset lại constructor sau sự kế thừa là :

```
TwoDShape.prototype.constructor = TwoDShape;  
  
Triangle.prototype.constructor = Triangle;
```

Giờ ta kiểm tra những gì ta đã làm.tạo ra 1 đối tượng Triangle object và gọi nó từ phương thức getArea() :

```
>>> var my = new Triangle(5, 10);
```

```
>>> my.getArea();
```

Mặc dù đối tượng không có phương thức chính gốc là `toString()`, nó được kế thừa phương thức này. chú ý rằng phương thức được kế thừa này là `toString()`, nó được kết hợp với đối tượng `this`

```
>>> my.toString()
```

```
"Triangle"
```

1 điều thú vị cần phải chú ý ở đây là nền tảng JS thực hiện như sau khi ta gọi `my.toString()` :

- nó thực hiện lặp qua tất cả các thuộc tính của `my` và nó không tìm thấy 1 phương thức nào gọi là **`toString()`**
- sau đó, nó bắt đầu tìm trong đối tượng **`my.__proto__`**, đối tượng này là 1 thực thể **`new TwoDShape()`** được tạo ra trong suốt quá trình thực thi của thực thể
- giờ nền tảng JS lặp thông qua thực thể của **`TwoDShape`** và không tìm thấy 1 phương thức **`toString()`**.sau đó nó kiểm tra **`__proto__`** của đối tượng đó.thời điểm này, **`__proto__`** chỉ tới thực thể được tạo bởi **`new Shape()`**
- thực thể của **`new Shape()`** được xem xét và **`toString()`** cuối cùng cũng được tìm ra
- phương thức này được gọi trong ngữ cảnh của **`my`**, có nghĩa là `this` chỉ tới **`my`**

ta đặt ra câu hỏi “ vậy hàm tạo nó là ai ? ”

nó sẽ báo cáo 1 cách chính xác bởi vì thuộc tính tạo constructor được reset mà ta đã thực hiện sau sự kế thừa

```
>>> my.constructor
```

```
Triangle(side, height)
```

Bằng cách sử dụng toán tử `instanceof`, ta có thể hợp thức hóa rằng `my` là 1 thực thể của tất cả 3 hàm tạo

```
>>> my instanceof Shape
```

```
true
```

```
>>> my instanceof TwoDShape
```

true

```
>>> my instanceof Triangle
```

true

```
>>> my instanceof Array
```

False

Cũng giống như trên khi ta gọi phương thức `isPrototypeOf()` của các hàm tạo chuyển tiếp qua `my` :

```
>>> Shape.prototype.isPrototypeOf(my)
```

true

```
>>> TwoDShape.prototype.isPrototypeOf(my)
```

true

```
>>> Triangle.prototype.isPrototypeOf(my)
```

true

```
>>> String.prototype.isPrototypeOf(my)
```

false

Ta cũng có thể tạo ra các đối tượng bằng cách sử dụng 2 hàm tạo khác. các đối tượng được tạo với `new TwoDShape()` cũng có phương thức `toString()`, được kế thừa từ `Shape()`

```
>>> var td = new TwoDShape();
```

```
>>> td.constructor
```

TwoDShape()

```
>>> td.toString()
```

"2D shape"

```
>>> var s = new Shape();
```

```
>>> s.constructor
```

```
Shape()
```

6.3.Cách chuyển các thuộc tính được chia sẻ vào bên trong prototype

Khi ta tạo ra các đối tượng objects bằng cách sử dụng 1 hàm tạo constructor function thì các thuộc tính của chính nó được thêm vào bằng sử dụng toán tử this.điều này có thể không có tác dụng trong trường hợp mà các thuộc tính không thay đổi thông qua việc thực thể hóa.trong trường hợp trên, Shape() được định nghĩa như sau :

```
function Shape(){  
  this.name = 'shape';  
}
```

Điều này có nghĩa là mọi lúc ta tạo ra 1 đối tượng mới bằng cách sử dụng new Shape() thì 1 thuộc tính name mới sẽ được tạo và lưu trữ ở 1 vài nơi nào đó trong bộ nhớ.có 1 lựa chọn khác là ta có thể thêm thuộc tính name vào trong prototype và chia sẻ nó ở toàn bộ các thực thể :

```
function Shape(){ }  
Shape.prototype.name = 'shape';
```

Giờ mọi lúc ta tạo ra 1 đối tượng mới bằng cách sử dụng new Shape() thì đối tượng này sẽ không có thuộc tính name chính gốc nhưng sẽ sử dụng thuộc tính này khi nó được thêm vào trong prototype,điều này mang lại hiệu quả tốt hơn, nhưng ta chỉ nên sử dụng nó cho các thuộc tính mà không bị thay đổi khi được thực thể hóa từ đối tượng này tới đối tượng khác.

Để cải tiến ví dụ ở trên bằng cách thêm vào tất cả các phương thức và thuộc tính phù hợp vào trong prototype.trong trường hợp Shape() và TwoDShape() thì mọi thứ có nghĩa là được chia sẻ :

```
function Shape(){ }  
// augment prototype  
Shape.prototype.name = 'shape';  
Shape.prototype.toString = function() {return this.name;};  
function TwoDShape(){ }
```

```
// take care of inheritance
TwoDShape.prototype = new Shape();
TwoDShape.prototype.constructor = TwoDShape;
// augment prototype
TwoDShape.prototype.name = '2D shape';
```

Như ta có thể chú ý tới sự phân cấp đầu tiên trước khi tham số hóa prototype, hay theo cách khác bất cứ thứ gì ta thêm vào TwoDShape.prototype sẽ bị xóa khi ta kế thừa

Hàm tạo Triangle constructor có 1 chút khác biệt bởi vì mọi đối tượng nó tạo ra là 1 đối tượng triangle mới – những triangle thường có kích thước khác biệt, do đó cách tốt nhất là giữ side và height là các thuộc tính gốc và chia sẻ phần còn lại. phương thức getArea() trong ví dụ này là như nhau bất chấp các chiều khác nhau của mỗi đối tượng triangle. do đó 1 lần nữa, ta kế thừa 1 vài cái đầu tiên và sau đó tham số hóa prototype

```
function Triangle(side, height) {
  this.side = side;
  this.height = height;
}
// take care of inheritance
Triangle.prototype = new TwoDShape();
Triangle.prototype.constructor = Triangle;
// augment prototype
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function(){return this.side * this.height
/ 2;};
```

Đoạn mã code ở trên sẽ hoạt động chính xác theo cùng với cách mà ta làm ở trên :

```
>>> var my = new Triangle(5, 10);
>>> my.getArea()
25
>>> my.toString()
"Triangle"
```


Đây chỉ là 1 sự hơi khác biệt khi ta gọi `my.toString()`. điều khác biệt là có nhiều hơn 1 sự tìm kiếm được thực hiện trước khi phương thức được tìm thấy trong `Shape.prototype`, điều này khác với trong thực thể `new Shape()` mà ta đã làm ở ví dụ trước

Ta cũng có thể dùng `hasOwnProperty()` để thấy sự khác biệt giữa thuộc tính chính gốc với 1 thuộc tính đến từ chuỗi prototype

```
>>> my.hasOwnProperty('side')
true
>>> my.hasOwnProperty('name')
false
```

gọi tới toán tử `isPrototypeOf()` và `instanceof` :

```
>>> TwoDShape.prototype.isPrototypeOf(my)
true
>>> my instanceof Shape
True
```

6.4.Cách thừa kế chỉ từ prototype

Như đã giải thích từ trước, vì lí do liên quan tới hiệu năng mà ta nên xem xét việc thêm các thuộc tính được tái sử dụng và các phương thức được tái sử dụng vào prototype. nếu ta làm như vậy thì nó hầu như là 1 ý tưởng tốt dùng để kế thừa chỉ từ prototype, bởi vì tất cả mã code được tái sử dụng lại nằm ở đó. điều này có nghĩa là việc kế thừa đối tượng obj được chứa trong `Shape.prototype` là hiệu quả hơn việc kế thừa đối tượng được tạo ra với `new Shape()`. sau tất cả thì `new Shape()` sẽ chỉ mang lại cho ta các thuộc tính shape properties của chính nó và dĩ nhiên điều này là không phải là được tái sử dụng, ta làm gia tăng hiệu quả lên 1 chút bởi :

- không tạo ra 1 đối tượng obj mới với mục đích kế thừa 1 cách đơn độc
- giảm thiểu thời gian tìm kiếm trong quá trình thực thi (ví dụ khi tìm kiếm `toString()`)

đây là đoạn code được cập nhật và các thay đổi được làm nổi bật lên :

```
function Shape(){ }
```

```
// augment prototype
Shape.prototype.name = 'shape';
Shape.prototype.toString = function() {return this.name;};
function TwoDShape(){ }
// take care of inheritance
TwoDShape.prototype = Shape.prototype;
TwoDShape.prototype.constructor = TwoDShape;
// augment prototype
TwoDShape.prototype.name = '2D shape';
function Triangle(side, height) {
  this.side = side;
  this.height = height;
}
// take care of inheritance
Triangle.prototype = TwoDShape.prototype;
Triangle.prototype.constructor = Triangle;
// augment prototype
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function(){return this.side * this.height
/ 2;}

```

Đoạn mã code dùng để kiểm tra cũng mang lại cùng kết quả :

```
>>> var my = new Triangle(5, 10);
>>> my.getArea()
25
>>> my.toString()
"Triangle"
```

Sự khác biệt nào trong việc tìm kiếm khi ta gọi my.toString() ?đầu tiên, như thông lệ thì JS tìm 1 phương thức có tên là toString() trong chính đối tượng obj đó.js không tìm thấy được phương thức nào như vậy thì nó sẽ kiểm tra trong prototype. Prototype chỉ tới cùng 1 đối tượng obj mà prototype của TwoDShape và prototype của Shape.prototype cùng chỉ tới.nhớ rằng các đối tượng objects không được sao chép bởi giá trị mà chỉ được tham chiếu.do đó

việc tìm kiếm chỉ là 1 tiến trình xử lý 2 bước ngược lại với tiến trình xử lý 4 bước ở ví dụ trước và tiến trình xử lý 3 bước ở ví dụ đầu tiên

Việc đơn giản sao chép prototype mang hiệu quả cao hơn nhưng nó cũng có **tác dụng phụ** :
bởi vì tất cả các con và cha **chỉ tới cũng 1 đối tượng object**, thì khi 1 con hiệu chỉnh prototype thì lớp cha của nó cũng bị thay đổi theo và các anh em của nó cũng bị như vậy

Nhìn vào dòng sau :

```
Triangle.prototype.name = 'Triangle';
```

Nó thay đổi thuộc tính name do vậy nó cũng thay đổi thuộc tính Shape.prototype.name. nếu ta tạo ra 1 thực thể bằng cách sử dụng new Shape() thì thuộc tính name sẽ là "Triangle"

```
>>> var s = new Shape()
```

```
>>> s.name
```

```
"Triangle"
```

6.5.Hàm tạo Constructor tạm thời - new F()

1 cách giải quyết vấn đề ở trên là vấn đề mà tất cả các prototypes đều chỉ tới cùng 1 đối tượng và đối tượng cha lấy cả thuộc tính của đối tượng con , đó là cách sử dụng 1 trung gian để phá vỡ chuỗi liên kết này. đối tượng trung gian nằm trong 1 hàm tạo tạm thời temporary constructor function. bằng cách tạo ra 1 hàm rỗng F() và thiết lập prototype của nó là prototype của hàm tạo cha, thì ta gọi new F() và tạo ra các đối tượng mà không có chứa thuộc tính bên trong nhưng lại kế thừa mọi thứ từ prototype của đối tượng cha

Ta hiệu chỉnh lại đoạn code như sau :

```
function Shape(){ }  
// augment prototype  
Shape.prototype.name = 'shape';  
Shape.prototype.toString = function() {return this.name;};  
function TwoDShape(){ }
```

```
// take care of inheritance
var F = function({});
F.prototype = Shape.prototype;
TwoDShape.prototype = new F();
TwoDShape.prototype.constructor = TwoDShape;

// augment prototype
TwoDShape.prototype.name = '2D shape';
function Triangle(side, height) {
  this.side = side;
  this.height = height;
}

// take care of inheritance
var F = function({});
F.prototype = TwoDShape.prototype;
Triangle.prototype = new F();
Triangle.prototype.constructor = Triangle;

// augment prototype
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function(){return this.side * this.height
/ 2;};
```

Ta vẫn dùng đoạn mã sau để test :

```
>>> var my = new Triangle(5, 10);
>>> my.getArea()
25
>>> my.toString()
"Triangle"
```

Bằng cách sử dụng cách tiếp cận này thì ta giữ chuỗi prototype ở nơi mà các thuộc tính của đối tượng cha không bị ghi đè bởi đối tượng con

```
>> my.__proto__.__proto__.__proto__.constructor
Shape()
>>> var s = new Shape();
```

```
>>> s.name  
"shape"
```

Vào cùng thời điểm, thì cách tiếp cận này hỗ trợ ý tưởng là chỉ có các thuộc tính và phương thức được thêm vào prototype mới được thừa kế và các thuộc tính chính gốc thì không nên được thừa kế. yếu tố gốc rễ của điều này là các thuộc tính chính gốc quá riêng biệt để có thể tái sử dụng

6.6.Uber – truy cập đối tượng cha từ đối tượng con

Các ngôn ngữ OOP cổ điển thường có 1 cấu trúc ngữ pháp đặc biệt dùng để cho ta khả năng truy cập tới lớp cha, hay như tham chiếu tới 1 lớp superclass.điều này rất thuận tiện khi 1 đối tượng con muốn có 1 phương thức để có thể làm được mọi thứ trong phương thức của đối tượng cha.trong những trường hợp như vậy thì đối tượng con gọi tới phương thức của cha với cùng 1 tên và hoạt động với cùng 1 kết quả

Trong JS, thì không có cấu trúc đặc biệt nào như vậy nhưng có thể dễ dàng tạo ra cái có cùng chức năng như vậy.ta viết lại ví dụ trên và chú ý tới sự kế thừa, tạo ra 1 thuộc tính uber dùng để chỉ tới đối tượng prototype của lớp cha

```
function Shape(){ }  
// augment prototype  
Shape.prototype.name = 'shape';  
Shape.prototype.toString = function(){  
var result = [];  
if (this.constructor.uber) {  
result[result.length] = this.constructor.uber.toString();  
}  
result[result.length] = this.name;  
return result.join(', ');  
};  
function TwoDShape(){ }  
// take care of inheritance  
var F = function(){ };  
F.prototype = Shape.prototype;  
TwoDShape.prototype = new F();
```

```
TwoDShape.prototype.constructor = TwoDShape;
TwoDShape.uber = Shape.prototype;
// augment prototype
TwoDShape.prototype.name = '2D shape';
function Triangle(side, height) {
  this.side = side;
  this.height = height;
}
// take care of inheritance
var F = function(){};
F.prototype = TwoDShape.prototype;
Triangle.prototype = new F();
Triangle.prototype.constructor = Triangle;
Triangle.uber = TwoDShape.prototype;
// augment prototype
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function(){return this.side * this.height
/ 2;}

```

Những thứ mới có ở đây :

- cách mà thuộc tính uber được thiết lập để chỉ tới prototype của lớp cha
- toString() được cập nhật

ở ví dụ trước thì toString() chỉ trả về this.name. hiện tại để bổ sung vào điều đó thì có 1 kiểm tra xem this.constructor.uber đã tồn tại chưa nếu có thì nó gọi tới toString() của nó đầu tiên. this.constructor là chính là hàm function và this.constructor.uber chỉ tới prototype của lớp cha. kết quả ở đây là khi ta gọi tới toString() đối với 1 thực thể Triangle, tất cả các phương thức toString() phía trên chuỗi prototype sẽ được gọi :

```
>>> var my = new Triangle(5, 10);
>>> my.toString()
"shape, 2D shape, Triangle"
```

Tên của thuộc tính uber có thể là "superclass" nhưng nó sẽ gợi ý rằng JS có các lớp classes.

6.7.Cách phân tách phần kế thừa vào trong 1 hàm function

Ta sẽ chuyển mã code cần cho toàn bộ sự kế thừa vào trong 1 hàm có tên là extend() có khả năng tái sử dụng :

```
function extend(Child, Parent) {  
  var F = function(){};  
  F.prototype = Parent.prototype;  
  Child.prototype = new F();  
  Child.prototype.constructor = Child;  
  Child.uber = Parent.prototype;  
}
```

Bằng cách sử dụng hàm này thì nó sẽ trợ giúp ta giữa các mã code được rõ ràng đối với các nhiệm vụ lặp đi lặp lại sự kế thừa.bằng cách này ta có thể đơn giản kế thừa bằng cách sử dụng :

extend(TwoDShape, Shape); và extend(Triangle, TwoDShape);

cách tiếp cận này chính là cách mà thư viện YUI thực thi sự kế thừa thông qua phương thức extend().ví dụ nếu ta sử dụng YUI và ta muốn Triangle thừa kế Shape thì ta sử dụng như sau :

YAHOO.lang.extend(Triangle, Shape)

VII.Các kiểu mẫu tạo đối tượng object

7.1.Các phương thức và thuộc tính riêng tư

JS không có cú pháp đặc biệt nào cho việc kí hiệu private, protected, hay public cho các thuộc tính và phương thức.tất cả các đối tượng objects đều là public :

```
var myobj = {  
  myprop: 1,  
  getProp: function () {  
    return this.myprop;  
  }  
};  
  
console.log(myobj.myprop); // `myprop` is publicly accessible  
console.log(myobj.getProp()); // getProp() is public too
```

điều này cũng đúng khi ta sử dụng các hàm tạo để tạo ra đối tượng objects; tất cả thành viên đều vẫn là public :

```
function Gadget() {  
  this.name = 'iPod';  
  this.stretch = function () {  
    return 'iPad';  
  };  
}  
  
var toy = new Gadget();  
console.log(toy.name); // `name` is public  
console.log(toy.stretch()); // stretch() is public
```

7.1.2.Các thành viên riêng tư

mặc dù ngôn ngữ không có cú pháp đặc biệt nào cho các thành viên riêng tư nhưng ta có thể thực thi chứng bằng cách sử dụng 1 bao đóng.các hàm khởi tạo của ta tạo ra 1 bao đóng và bất cứ biến nào nằm trong phạm vi bao đóng đều không lộ ra bên ngoài hàm khởi tạo.**tuy nhiên các biến riêng tư này luôn được thể hiện trong các phương thức public** – các phương thức được định nghĩa bên trong hàm tạo và được lộ ra như phần của các đối tượng được trả

về,ta xét 1 ví dụ với **name** là tên thành viên riêng tư – không truy cập được từ ngoài hàm khởi tạo :

```
function Gadget() {  
  // private member  
  var name = 'iPod';  
  
  // public function  
  this.getName = function () {  
    return name;  
  };  
}  
  
var toy = new Gadget();  
// `name` is undefined, it's private  
console.log(toy.name); // undefined  
// public method has access to `name`  
console.log(toy.getName()); // "iPod"
```

như ta thấy, rất dễ để tạo ra sự riêng tư trong JS.tất cả những gì mà ta cần là thực đóng gói dữ liệu mà ta muốn giữ riêng tư vào trong 1 hàm function và đảm bảo rằng nó là địa phương trong hàm function

7.1.3.Các phương thức được ưu tiên

kí hiệu privileged methods cũng không bao hàm bất cứ cú pháp đặc biệt nào, nó chỉ là 1 tên được đặt cho các phương thức public mà có thể truy cập tới các thành viên riêng tư

trong ví dụ trên thì **getName()** là 1 phương thức được ưu tiên privileged method bởi vì nó đặc biệt dùng để truy cập thuộc tính riêng tư name

7.1.4. Các thiếu sót quyền riêng tư

Có một số trường hợp khi ta lo ngại về sự riêng tư :

- ở 1 vài phiên bản trước của firefox cho phép 1 tham số thứ 2 được chuyển tiếp vào hàm eval() mà là 1 đối tượng ngữ cảnh cho phép ta lên vào phạm vi riêng tư của hàm function. tương tự với thuộc tính __parent__ trong Mozilla Rhino cho phép ta truy cập vào phạm vi riêng tư. những trường hợp này không còn được áp dụng vào các trình duyệt được sử dụng rộng rãi ngày nay
- khi ta trực tiếp trả về 1 biến riêng tư từ 1 phương thức ưu tiên và biến này là 1 đối tượng hay 1 mảng array thì mã code bên ngoài có thể hiệu chỉnh biến riêng tư này bởi vì nó được chuyển tiếp bởi tham chiếu

```
function Gadget() {  
  // private member  
  
  var specs = {  
  
    screen_width: 320,  
    screen_height: 480,  
    color: "white"  
  
  };  
  
  // public function  
  this.getSpecs = function () {  
    return specs;  
  };  
}
```

Vấn đề ở đây là **getSpecs()** trả về 1 tham chiếu tới đối tượng **specs** object. điều này cho phép người dùng của Gadget hiệu chỉnh **specs** riêng tư và dường như bị ẩn đi

```
var toy = new Gadget(),  
specs = toy.getSpecs()  
specs.color = "black";  
specs.price = "free";
```

```
console.dir(toy.getSpecs());
```

kết quả này được in ra trong firebug như hình 5-2 :



color	"black"
price	"free"
screen_height	480
screen_width	320

Figure 5-2. The private object was modified

Giải pháp cho trạng thái xử lý không mong muốn này là cần thận không chuyển tiếp tham chiếu tới đối tượng object và mảng array mà ta muốn giữ riêng tư. 1 cách để thực hiện điều này là getSpecs() trả về 1 đối tượng object mới chỉ chứa 1 vài dữ liệu mà có thể được dùng trong đối tượng. điều này được biết đến như là nguyên tắc tối thiểu hóa quyền hạn (POLA) – nó phát biểu là ta không bao giờ đưa mức quyền hạn cao hơn mức cần. trong trường hợp này nếu người dùng Gadget chỉ quan tâm tới kích thước. do vậy thay thế cho việc đưa ra mọi thứ thì ta có thể tạo ra getDimensions() – phương thức trả về 1 đối tượng mới chỉ chứa width và height. ta rất có thể không cần thực thi getSpecs()

Cách tiếp cận khác, khi ta cần chuyển tiếp tất cả dữ liệu thì ta tạo ra 1 bản copy của đối tượng specs object bằng cách sử dụng 1 hàm general-purpose object-cloning function. chương sau sẽ có 2 hàm functions như vậy – 1 hàm được gọi là extend() có chức năng thực hiện 1 bản sao chép shallow copy của 1 đối tượng đã biết, 1 hàm được gọi là extendDeep() dùng thực hiện sao chép sâu, sao chép 1 cách đệ quy tất cả các thuộc tính và các thuộc tính lồng nhau của chúng

7.1.5.Object Literals và quyền riêng tư

ở trên ta xét ví dụ được sử dụng tạo ra đối tượng bằng cách sử dụng hàm tạo để tạo ra sự riêng tư. nhưng thế trong trường hợp ta tạo ra các đối tượng object bằng cách trực tiếp thì làm thế nào ? nó vẫn có thể tạo ra các thành viên riêng tư chứ ?

như ta đã nói tới từ trước, tất cả những gì mà ta cần là 1 hàm function dùng để đóng gói lại dữ liệu riêng tư. vì vậy trong trường hợp tạo ra các đối tượng bằng cách trực tiếp thì ta có thể sử dụng bao đóng được tạo ra bởi cách bổ sung thêm 1 hàm vô danh trực tiếp. đây là ví dụ :

var **myobj**; // this will be the object

(function () {

// private members

var **name** = "my, oh my";

// implement the public part

// note -- no `var`

myobj = {

// privileged method

getName: function () {

return name;

}

};

}());

myobj.**getName**(); // "my, oh my"



Ta khởi tạo 1 đối tượng rỗng **myobj** bên ngoài hàm tức thời. trong hàm tức thời ta thực hiện gán đối tượng này. trong đối tượng chứa phương thức ưu tiên **getName**

cũng cùng ý tưởng như trên với cách thực thi khác đôi chút như ví dụ sau :

var **myobj** = (function () {

// private members

var **name** = "my, oh my";

// implement the public part

return {

getName: function () {

return name;

}

};

}());



Ta gán đối tượng với giá trị được trả về trong hàm tức thời. giá trị được trả về này là 1 đối tượng, bên trong đối tượng này chứa phương thức ưu tiên **getName**

```
myobj.getName(); // "my, oh my"
```

ví dụ này cũng là khung xương của kiểu mẫu được biết tên là “module pattern,”

7.1.6.Các Prototypes và quyền riêng tư

hạn chế của các thành viên riêng tư là khi sử dụng với các hàm khởi tạo thì chúng được tạo ra mỗi lần hàm khởi tạo được gọi để tạo ra đối tượng mới

đây là 1 vấn đề thực tiễn với bất cứ thành viên nào khi ta thêm this vào bên trong hàm khởi tạo. để tránh sự dư thừa và tiết kiệm bộ nhớ thì ta có thể thêm có thuộc tính và phương thức chung vào thuộc tính prototype của hàm khởi tạo. với cách này thì các phần dùng chung được chia sẻ với toàn bộ các thực thể được tạo ra bởi cùng 1 hàm khởi tạo. ta cũng có thể chia sẻ các thành viên riêng tư ẩn đi với các thực thể này. để làm được như vậy ta có thể sử dụng 1 bộ đôi kiểu mẫu : private properties bên trong hàm khởi tạo và private properties bên trong đối tượng nguyên bản. bởi vì thuộc tính prototype cũng là 1 đối tượng nên nó có thể được tạo với đối tượng nguyên bản

```
function Gadget() {  
  // private member  
  
  var name = 'iPod';  
  
  // public function  
  this.getName = function () {  
    return name;  
  };  
}  
  
Gadget.prototype = (function () {  
  // private member  
  
  var browser = "Mobile Webkit";  
  
  // public prototype members  
  return {  
    getBrowser: function () {  
      return browser;  
    }  
  }  
})()
```

```
};
```

```
}());
```

```
var toy = new Gadget();
```

```
console.log(toy.getName()); // privileged "own" method
```

```
console.log(toy.getBrowser()); // privileged prototype method
```

7.1.7.Cách phát hiện các hàm riêng như các phương thức public

kiểu mẫu revelation pattern là về cách có các phương thức riêng tư – những phương thức này ta cũng muốn để lộ ra giống các phương thức public. điều này có thể hữu dụng khi tất cả các chức năng trong 1 đối tượng có tính quyết định đối với việc làm việc với đối tượng và ta muốn bảo vệ nó nhiều nhất có thể. nhưng cũng cùng thời điểm đó ta muốn cung cấp 1 truy cập public tới 1 vài trong các chức năng này bởi vì nó có thể hữu dụng. khi ta để lộ ra các phương thức 1 cách công khai thì ta khiến chúng có khả năng bị tấn công : 1 vài người dùng của API public này có thể hiệu chỉnh nó 1 cách vô tình. trong ECMAScript 5 thì ta có 1 lựa chọn để đóng băng 1 đối tượng nhưng nó không có trong các phiên bản về trước. áp dụng vào kiểu mẫu revelation pattern (cụm từ này được tạo ra bởi Christian Heilmann)

ta xét 1 ví dụ về cách xây dựng :

```
var myarray;
```

```
(function () {
```

```
var astr = "[object Array]",
```

```
toString = Object.prototype.toString;
```

```
function isArray(a) {
```

```
return toString.call(a) === astr;
```

```
}
```

```
function indexOf(haystack, needle) {
```

```
var i = 0,
```

```
max = haystack.length;
```

```
for (; i < max; i += 1) {  
  if (haystack[i] === needle) {  
    return i;  
  }  
}  
return -1;  
}
```

```
myarray = {  
  isArray: isArray,  
  indexOf: indexOf,  
  inArray: indexOf  
};  
}());
```

ở ví dụ này ta có 2 biến riêng tư và 2 hàm riêng tư - `isArray()` và `indexOf()`. xem xét phần cuối của hàm tức thời thì đối tượng `myarray` được đưa dẫn đến với chức năng mà ta quyết định tạo ra sự sẵn sàng công khai. trong trường hợp này thì `indexOf()` riêng tư được lộ ra

7.2. Cách thành viên cố định static

các phương thức và phương thức static là những cái mà không thay đổi từ đối tượng này tới đối tượng khác. trong ngôn ngữ hướng class thì các thành viên static được tạo bằng cách sử dụng cú pháp đặc biệt và sau đó được sử dụng như thể chúng là các thành viên của chính lớp class. ví dụ 1 phương thức static `max()` của lớp `MathUtils` sẽ được gọi như sau

`MathUtils.max(3, 5)`. đây là 1 ví dụ về 1 thành viên public – có thể được sử dụng mà không phải tạo ra 1 thực thể của class. đây cũng có thể là các thành viên riêng tư static – không hiện thị đối với người dùng class nhưng vẫn được chia sẻ thông qua tất cả các thực thể của lớp class. chúng sẽ xem cách thực thi cả các thành viên public và private static trong JS

7.2.1. Các thành viên Public Static

trong JS thì không có cú pháp đặc biệt nào để ký hiệu các thành viên static. nhưng ta có thể có cú pháp tương tự như trong ngôn ngữ hướng class bằng cách sử dụng 1 hàm khởi tạo và thêm các thuộc tính vào trong nó. điều này có hoạt động bởi vì các hàm khởi tạo – giống với tất cả các hàm function khác đều là các đối tượng object và chúng cũng có các thuộc tính. kiểu mẫu tối ưu hóa bộ nhớ được bàn tới trong chương trước cũng thực hiện cùng 1 ý tưởng – thêm các thuộc tính vào 1 hàm function

ví dụ sau định nghĩa 1 hàm khởi tạo Gadget với 1 phương thức static **isShiny()** và 1 phương thức chính quy **setPrice()**. phương thức **isShiny()** là phương thức static bởi vì nó không cần 1 đối tượng gadget object cụ thể nào để hoạt động. và mặt khác phương thức **setPrice()** cần 1 đối tượng bởi vì gadgets có thể định giá khác nhau :

```
// constructor
var Gadget = function () {};

// a static method
Gadget.isShiny = function () {

return "you bet";

};

// a normal method added to the prototype
Gadget.prototype.setPrice = function (price) {

this.price = price;

};
```

Giờ ta gọi các phương thức này. phương thức tĩnh **isShiny()** được gọi trực tiếp từ hàm khởi tạo trái ngược với phương thức chính quy cần 1 thực thể mới gọi được :

```
// calling a static method
Gadget.isShiny(); // "you bet"

// creating an instance and calling a method
var iphone = new Gadget();
```



```
iphone.setPrice(500);
```

việc cố thử gọi phương thức chính quy từ 1 hàm khởi tạo thì sẽ không hoạt động, kết quả cũng vậy khi cố gọi 1 phương thức static từ đối tượng iphone object:

```
typeof Gadget.setPrice; // "undefined"
```

```
typeof iphone.isShiny; // "undefined"
```

thình thoảng cũng khá là tiện lợi khi ta có các phương thức static hoạt động được với 1 thực thể.điều này dễ dàng làm được đơn giản thêm 1 phương thức mới vào prototype – nó giống với 1 con trỏ chỉ tới phương thức static gốc :

```
Gadget.prototype.isShiny = Gadget.isShiny;
```

```
iphone.isShiny(); // "you bet"
```

trong nhiều trường hợp ta cần cẩn thận nếu ta sử dụng this bên trong phương thức static.khi ta thực hiện Gadget.isShiny() thì this nằm bên trong isShiny() sẽ tham chiếu tới hàm tạo Gadget.nếu ta thực hiện gọi iphone.isShiny() thì this sẽ chỉ tới iphone

ở ví dụ trước chỉ ra cho ta cách mà ta có thể các cùng 1 phương thức được gọi 1 cách cố định và không cố định và trạng thái thực thi có đôi chút khác biệt, phụ thuộc vào kiểu mẫu invocation pattern.tại đây **instanceof** trợ giúp xác định cách mà phương thức được gọi :

```
// constructor
```

```
var Gadget = function (price) {  
  this.price = price;  
};
```

```
// a static method
```

```
Gadget.isShiny = function () {
```

```
// this always works
```

```
var msg = "you bet";
if (this instanceof Gadget) {
  // this only works if called non-statically- this chỉ hoạt động khi lời gọi không phải là lời gọi cố định
  msg += ", it costs $" + this.price + '!';
}
return msg;
};
```

// a normal method added to the prototype

```
Gadget.prototype.isShiny = function () {
  return Gadget.isShiny.call(this);
};
```

Kiểm tra lời gọi 1 phương thức static :

```
Gadget.isShiny(); // "you bet"
```

Kiểm tra lời gọi không cố định từ 1 thực thể :

```
var a = new Gadget('499.99');
a.isShiny(); // "you bet, it costs $499.99"
```

7.2.2.Các thành viên Private Static

giờ ta xét tới cách mà ta có thể thực thi các thành viên Private Static – các thành viên này là :

- được chia sẻ bởi tất cả các đối tượng được tạo ra trong cùng 1 hàm khởi tạo
- không thể truy cập ngoài hàm khởi tạo

giờ ta xét 1 ví dụ với **counter** là 1 thuộc tính private static trong hàm tạo Gadget.ta cần 1 hàm function để thực hiện như là 1 bao đóng và đóng gói xung quanh các thành viên private.sau đó ta có 1 hàm bao đóng giống vậy để thực thi 1 cách trực tiếp và trả về 1 hàm function mới.giá trị hàm function được trả về được gán vào biến Gadget và trở thành 1 hàm khởi tạo mới :

```
var Gadget = (function () {  
  
    // static variable/property  
  
    var counter = 0;  
  
    // returning the new implementation  
    // of the constructor  
    return function () {  
        console.log(counter += 1);  
    };  
  
})(); // execute immediately
```

Hàm tạo Gadget mới đơn giản là cộng thêm 1 vào biến riêng tư counter. tiến hành kiểm tra với 1 vài thực thể ta có thể thấy rằng counter thực tế được chia sẻ với tất cả các thực thể :

```
var g1 = new Gadget(); // logs 1  
var g2 = new Gadget(); // logs 2  
var g3 = new Gadget(); // logs 3
```

bởi vì ta tăng 1 đơn vị trong counter với mọi đối tượng thì thuộc tính static này trở thành 1 ID mà được xác định độc nhất trong từng phần tử được tạo ra với cùng hàm tạo Gadget. xác định duy nhất này có thể hữu dụng vì vậy tại sao không để lộ nó thông qua 1 phương thức ưu tiên ?

ví dụ dưới xây dựng dựa trên ví dụ trước và thêm vào 1 phương thức ưu tiên **getLastId()** để truy cập tới thuộc tính static private :

```
// constructor  
  
var Gadget = (function () {  
  
    // static variable/property  
  
    var counter = 0,  
  
    NewGadget;
```

```
// this will become the
// new constructor implementation
NewGadget = function () {
  counter += 1;
};

// a privileged method
NewGadget.prototype.getLastId = function () {
  return counter;
};

// overwrite the constructor
return NewGadget;

})(); // execute immediately
```

Kiểm tra thực thi mới :

```
var iphone = new Gadget();
iphone.getLastId(); // 1
var ipod = new Gadget();
ipod.getLastId(); // 2
var ipad = new Gadget();
ipad.getLastId(); // 3
```

thuộc tính static (bao gồm cả private và public) có thể hoàn toàn dễ dàng điều khiển. chúng có thể chứa các phương thức và dữ liệu mà không phải là 1 định nghĩa thực thể và không phải tạo lại với mọi thực thể. trong chương sau, khi ta đề cập tới kiểu mẫu singleton pattern thì ta có thể thấy 1 ví dụ về thực thi mà sử dụng các thuộc tính static để thực thi các hàm khởi tạo singleton giống với hướng class

7.3.Các hằng số đối tượng

không tồn tại các hằng số trong JS, mặc dù nhiều môi trường hiện đại rất có thể đưa ra cho ta cú pháp const để tạo ra hằng số

1 cách tiếp cận thông thường là sử dụng 1 quy ước đặt tên và khiến các biến mà không được thay đổi.quy ước này thực tế được sử dụng trong đối tượng được xây dựng sẵn trong JS :

```
Math.PI; // 3.141592653589793
```

```
Math.SQRT2; // 1.4142135623730951
```

```
Number.MAX_VALUE; // 1.7976931348623157e+308
```

Để tạo ra với các hằng số của chính mình thì ta có thể chấp nhận quy ước đặt tên như vậy và thêm chúng vào như các thuộc tính static vào bên trong hàm khởi tạo :

```
// constructor
var Widget = function () {
// implementation...
};
// constants
Widget.MAX_HEIGHT = 320;
Widget.MAX_WIDTH = 480;
```

Quy ước như trên có thể được áp dụng vào các đối tượng được tạo bởi với cách tạo đối tượng nguyên bản, các hằng số có thể là các thuộc tính thông thường với tên là chữ viết in hoa

Nếu ta muốn có 1 giá trị không biến đổi thì ta có thể tạo ra 1 thuộc tính private và cung cấp 1 phương thức getter nhưng không cho phương thức setter.đây dường như là 1 khả năng quá mức cần thiết trong nhiều trường hợp khi ta có thể lấy về bằng với 1 quy ước đơn giản nhưng nó vẫn còn là 1 lựa chọn

Ví dụ sau là 1 thực thi của các đối tượng constant object – đối tượng này được cung cấp các phương thức :

```
set(name, value)
```

để định nghĩa 1 hằng số mới

```
isDefined(name)
```

kiểm tra xem hằng số đã tồn tại chưa

get(name)

lấy về giá trị trong hằng số

trong thực thi này, chỉ các giá trị cơ bản mới được là các hằng số.

```
var constant = (function () {  
  var constants = {},  
  ownProp = Object.prototype.hasOwnProperty,  
  allowed = {  
    string: 1,  
    number: 1,  
    boolean: 1  
  },  
  prefix = (Math.random() + "_").slice(2);  
  return {  
    set: function (name, value) {  
      if (this.isDefined(name)) {  
        return false;  
      }  
      if (!ownProp.call(allowed, typeof value)) {  
        return false;  
      }  
      constants[prefix + name] = value;  
      return true;  
    },  
    isDefined: function (name) {  
      return ownProp.call(constants, prefix + name);  
    },  
    get: function (name) {  
      if (this.isDefined(name)) {  
        return constants[prefix + name];  
      }  
      return null;  
    }  
  }  
})
```

```
}  
};  
}());
```

Kiểm tra thực thi :

```
// check if defined  
constant.isDefined("maxwidth"); // false  
// define  
constant.set("maxwidth", 480); // true  
// check again  
constant.isDefined("maxwidth"); // true  
// attempt to redefine  
constant.set("maxwidth", 320); // false  
// is the value still intact?  
constant.get("maxwidth"); // 480
```

7.4.Kiểu mẫu chuỗi hóa - Chaining Pattern

Chaining Pattern cho phép ta gọi các phương thức trên 1 đối tượng từng cái này sau cái khác mà không gán các giá trị trả về của thao tác phía trước vào các biến và không phải cắt lời gọi của ta ra làm nhiều dòng :

```
myobj.method1("hello").method2().method3("world").method4();
```

khi ta tạo ra các phương thức mà không có ý nghĩa là phải trả về giá trị thì ta **có thể trả về this** – thực thể của đối tượng mà chúng đang được thực thi.điều này sẽ cho phép người dùng của đối tượng đó gọi tới phương thức tiếp theo tạo nên 1 chuỗi như ví dụ trước :

```
var obj = {  
  value: 1,  
  increment: function () {  
    this.value += 1;  
    return this;  
  },  
  add: function (v) {
```

→

Thể hiện sau phương thức này có thể nối chuỗi hay phương thức khác có thể được thêm vào ngay sau phương thức này

```
this.value += v;  
return this;  
  
},  
shout: function () {  
  alert(this.value);  
}  
};  
  
// chain method calls  
obj.increment().add(3).shout(); // 5  
  
// as opposed to calling them one by one  
obj.increment();  
obj.add(3);  
obj.shout(); // 5
```

7.4.1.Ưu điểm và nhược điểm của kiểu mẫu chuỗi hóa - Chaining Pattern

1 ưu điểm của việc sử dụng kiểu mẫu Chaining Pattern là ta có thể tiết kiệm 1 vài lần gõ và tạo mã code ngắn gọn hơn

1 ưu điểm khác là nó trợ giúp cho ta nghĩ về việc cắt các hàm functions của ta và tạo ra thành những cái nhỏ hơn, những hàm functions chuyên biệt hơn.điều này cải tiến khả năng sửa chữa trong thời gian chạy lâu

1 nhược điểm là nó sẽ khó để tìm lỗi trong mã code.ta rất có thể biết 1 lỗi xảy ra trên 1 dòng cụ thể nhưng lại có quá nhiều thứ trên dòng này.khi 1 trong các phương thức đáp ứng mà ta đã chuỗi hóa gặp lỗi 1 cách âm thầm thì ta khó lòng nhận ra nó

Trong bất kì trường hợp nào thì cách tốt để tổ chức lại kiểu mẫu này và khi 1 phương thức mà ta viết không rõ ràng và không có ý nghĩa trả về giá trị thì ta có thể luôn luôn trả về this.kiểu mẫu này được sử dụng khá rộng rãi trong JQUERY.và nếu ta nhìn vào DOM API thì nó chú ý tới nó cũng có cấu trúc được chuỗi hóa như sau :

```
document.getElementsByTagName('head')[0].appendChild(newnode);
```


7.5.Phương thức method()

JS có thể gây khó hiểu đối với nhiều lập trình viên – những người thường suy nghĩ tới các cụm từ trong các lớp classes.đây là lí do tại sao mà 1 vài người lập trình chọn lựa khiến JS giống với hướng class hơn.1 trong những cố gắng như vậy là ý tưởng về phương thức method() được giới thiệu bởi Douglas Crockford.

Bằng cách sử dụng các hàm khởi tạo giống như sử dụng các lớp classes trong java.chúng cũng cho phép ta thêm vào các thuộc tính thực thể vào this bên trong thân hàm khởi tạo.tuy nhiên cách thêm vào các phương thức vào this là không hiệu quả bởi vì chúng kết thúc quá trình tạo lại mọi thực thể và sử dụng nhiều bộ nhớ hơn.đây là lí do tại sao các phương thức có khả năng được tái sử dụng nên được thêm vào thuộc tính prototype trong hàm khởi tạo.thuộc tính prototype có thể trông rất lạ lẫm với nhiều nhà lập trình do vậy ta có thể ẩn nó bên dưới 1 phương thức

Chú ý : việc thêm các chức năng tiện lợi vào trong 1 ngôn ngữ thông thường được xem như là syntactic sugar hay đơn giản là sugar.trong trường hợp này ta có thể gọi phương thức method() là “sugar method.”

Cách định nghĩa 1 lớp class bằng cách sử dụng phương thức method() sẽ như sau :


```
var Person = function (name) {  
  this.name = name;  
  
}•
```

```
method('getName', function () {  
  return this.name;  

```

```
});•
```

```
method('setName', function (name) {  
  
  this.name = name;  
  return this;  
  
});
```



Thể hiện sau phương thức này có thể nối chuỗi hay phương thức khác có thể được thêm vào ngay sau phương thức này.trong trường hợp này phương thức setName có khả năng nối chuỗi

Chú ý cách hàm khởi tạo được chuỗi hóa để gọi phương thức method().điều này cho phép kiểu mẫu chuỗi hóa ở phần trước trợ giúp ta định nghĩa toàn bộ lớp class với 1 cú pháp đơn

Phương thức method() cần 2 tham số :

- tên của phương thức mới
- mã thực thi của phương thức

phương thức mới này được thêm vào lớp Person và mã thực thi cũng giống như hàm function khác và nằm bên trong thực thi function thì this chỉ tới đối tượng được tạo bởi hàm tạo

Person

đây là cách mà ta có thể sử dụng Person() để tạo ra và sử dụng 1 đối tượng mới :

```
var a = new Person('Adam');  
a.getName(); // 'Adam'  
a.setName('Eve').getName(); // 'Eve'
```

1 lần nữa chú ý tới kiểu mẫu chaining pattern

Và cuối cùng đây là cách mà phương thức **method()** được thực thi :

```
if (typeof Function.prototype.method !== "function") {  
  Function.prototype.method = function (name, implementation) {  
    this.prototype[name] = implementation;  
    return this;  
  };  
}
```

Trong thực thi của method(), đầu tiên ta kiểm tra xem phương thức này chưa được thực thi.nếu chưa ta xử lý bằng cách thêm hàm function này chuyển tiếp như 1 tham số implementation vào prototype của hàm tạo.trong trường hợp này this tham chiếu tới hàm tạo và prototype của nó được tham số hóa

VIII. Các kiểu mẫu có khả năng tái sử dụng - Code Reuse Patterns

Việc tái sử dụng lại mã code là 1 chủ đề rất thú vị và khá quan trọng đơn giản là vì nó là 1 điều rất tự nhiên trong lập trình, nó được sử dụng cho việc viết mã code ít đi và tái sử dụng lại nhiều nhất có thể từ đoạn mã code đã có mà ta hay 1 vài người khác đã viết trước đó. về bản chất điều này rất tốt cho việc thử nghiệm, có khả năng sửa chữa, mở rộng và mã code hướng tài liệu

Khi đang bàn luận về việc tái sử dụng mã code thì thứ đầu tiên đến trong suy nghĩ của ta là sự kế thừa, và nó là 1 chủ đề lớn ở trong chương sau. ta sẽ thấy 1 vài cách để làm sự kế thừa này theo hướng class hóa hoặc không. nhưng điều quan trọng để giữ tới cuối cùng mục tiêu trong đầu ta là ta muốn tái sử dụng mã code. sự kế thừa là 1 cách để với tới mục đích này. và nó không chỉ là cách duy nhất. ta sẽ thấy cách mà ta có thể biên soạn các đối tượng objects từ các đối tượng khác, cách sử dụng đối tượng mix-ins và cách mà ta có thể vay mượn và tái sử dụng chỉ với những chức năng mà ta cần mà không phải kế thừa 1 cách kỹ thuật từ bất cứ gì vĩnh cửu

Khi tiếp cận 1 nhiệm vụ tái sử dụng lại mã code thì hãy nhớ trong đầu về định luật Gang of Four để đưa ra cách tạo đối tượng object “**ưu tiên sự kết tụ đối tượng hơn là sự phân cấp lớp class**”

8.1. Kiểu mẫu hướng class so với các kiểu mẫu phân cấp hiện đại

Ta thường nghe được cụm từ “sự kế thừa hướng class - classical inheritance” trong các bàn luận trên các chủ đề về sự kế thừa trong JS vì vậy đầu tiên ta sẽ làm rõ “classical” nghĩa là gì. từ này không được sử dụng để miêu tả 1 cái gì đó cổ xưa, đã chìm vào dĩ vãng. từ này chỉ là 1 cách chơi chữ của từ “class.”

Nhiều ngôn ngữ lập trình có khái niệm về các lớp classes – là các bản thiết kế dành cho đối tượng object. trong những ngôn ngữ đó thì mọi đối tượng là 1 thực thể của 1 lớp class cụ thể nào đó và 1 đối tượng không thể được tạo ra nếu lớp class cho nó không tồn tại. trong JS bởi vì không có khái niệm gì về các lớp classes nên khái niệm về thực thể của lớp class không có ý nghĩa gì cả. các đối tượng trong JS đơn giản là các cặp key-value, cái mà ta có thể tạo ra và thay đổi bất cứ lúc nào ta muốn

Nhưng JS có các hàm khởi tạo và cú pháp toán tử new giống với đa số cú pháp khi sử dụng lớp classes

Trong java thì ta có thể làm 1 vài thứ như sau :

```
Person adam = new Person();
```

Trong js thì ta có thể làm :

```
var adam = new Person();
```

1 điểm khác biệt là trong java thì hướng xác định kiểu dữ liệu là rất mạnh mẽ và ta cần khai báo adam là 1 kiểu dữ liệu Person. lời gọi hàm khởi tạo trong JS như thể Person là 1 lớp class nhưng quan trọng là nhớ ở trong đây là Person vẫn còn là 1 hàm function. sự tương đồng trong cú pháp đã khiến cho nhiều nhà lập trình nghĩ về JS trong cụm từ classes để phát triển các ý tưởng và các kiểu mẫu kế thừa để giả lập các lớp classes. các thực thi giống như vậy ta có thể gọi là “classical.” ta cũng phát biểu rằng cụm từ “modern” là bất cứ kiểu mẫu nào khác mà nó không bắt buộc ta phải nghĩ về các lớp classes

Khi nói đến việc áp dụng 1 kiểu mẫu kế thừa cho các project thì ta có 1 số các lựa chọn. ta có thể luôn luôn cố gắng lựa chọn 1 kiểu mẫu modern pattern, trừ khi nhóm của ta không thực sự cảm thấy thoải mái nếu project không bao hàm các lớp class trong đó

8.2. Kết quả như mong muốn khi sử dụng sự kế thừa theo hướng classical

Mục tiêu của sự kế thừa theo hướng classical là phải có các đối tượng objects được tạo bởi 1 hàm tạo Child() lấy các thuộc tính trong 1 hàm tạo khác là Parent().

Chú ý : mặc dù đã bàn luận về kiểu mẫu classical patterns nên chúng ta tránh sử dụng từ “class.”. việc nói “hàm khởi tạo - constructor function” hay “constructor” vẫn còn nhưng nó chính xác và không nhập nhằng. theo thông thường thì ta cố gắng tránh sử dụng từ “constructor” khi bàn bạc trong nhóm bởi vì khi sử dụng nó trong JS thì từ này rất có thể mang ý nghĩa khác nhau đối với từng người

Đây là 1 ví dụ về cách định nghĩa 2 hàm khởi tạo Parent() và Child() :

```
// the parent constructor
function Parent(name) {
  this.name = name || 'Adam';
}
// adding functionality to the prototype
```

```
Parent.prototype.say = function () {  
    return this.name;  
};  
// empty child constructor  
function Child(name) {}  
// inheritance magic happens here  
inherit(Child, Parent);
```

ở đây ta có các hàm khởi tạo parent và child, 1 phương thức say() được thêm vào prototype của hàm khởi tạo Parent() và gán đối tượng này tới prototype của Child().đây là thực thi đầu tiên của hàm có chức năng tái sử dụng **inherit()** :

```
function inherit(C, P) {  
    C.prototype = new P();  
}
```

Điều quan trọng cần nhớ là thuộc tính prototype chỉ nên **chỉ tới 1 đối tượng** chứ **không phải là 1 hàm function**. vì vậy nó phải chỉ tới 1 thực thể (1 đối tượng) được tạo ra bởi hàm tạo parent chứ không phải hàm tạo của chính nó.theo cách khác chú ý tới toán tử new bởi vì ta cần nó cho kiểu mẫu này có thể hoạt động

Sau đó trong ứng dụng của ta khi ta sử dụng new Child() để tạo ra 1 đối tượng thì nó lấy chức năng từ thực thể Parent() thông qua prototype như ví dụ sau :

```
var kid = new Child();  
kid.say(); // "Adam"
```

8.3.Việc kế tiếp chuỗi prototype

bằng cách sử dụng kiểu mẫu này thì ta kế thừa cả thuộc tính chính gốc (thuộc tính được định nghĩa cho từng thực thể được thêm vào this giống như name) và các thuộc tính và phương thức trong prototype (giống như say())

ta xem xét lại cách chuỗi prototype hoạt động trong kiểu mẫu kế thừa này.ta nghĩ về các đối tượng như các khối blocks ở 1 vài nơi nào đó trong bộ nhớ - chúng có thể chứa dữ liệu vào

các tham chiếu tới các blocks khác.khi ta tạo ra 1 đối tượng bằng cách sử dụng new Parent() thì ta tạo ra 1 khối giống như block (block được đánh dấu #2 trong hình 6-1).nó giữ dữ liệu cho thuộc tính name.nếu ta có gắng truy cập vào phương thức say() mặc dù block #2 không chứa phương thức này.nhưng bằng cách sử dụng liên kết ẩn __proto__ để chỉ tới thuộc tính prototype của hàm tạo Parent() thì ta khuếch đại truy cập tới đối tượng object #1 (Parent.prototype) mà nó chứa phương thức say().tất cả xảy ra đằng sau ngữ cảnh này và ta không cần lo lắng về nó nhưng điều quan trọng phải biết cách mà nó hoạt động và nơi dữ liệu mà ta đang truy cập hay rất có thể hiệu chỉnh nó.chú ý rằng __proto__ được sử dụng ở đây để giải thích chuỗi prototype; thuộc tính này không có sẵn trong ngôn ngữ mặc dù nó được cung cấp trong 1 vài môi trường (ví dụ như Firefox).

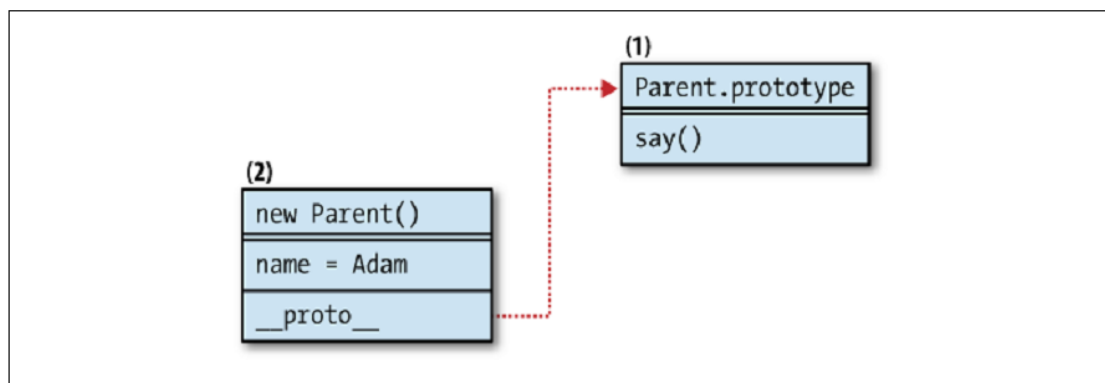


Figure 6-1. Prototype chain for the Parent() constructor

Giờ ta xem những gì xảy ra khi 1 đối tượng mới được tạo bằng cách sử dụng kid = new Child() ngay sau sử dụng hàm inherit()function.biểu đồ được chỉ ra như hình 6-2 :

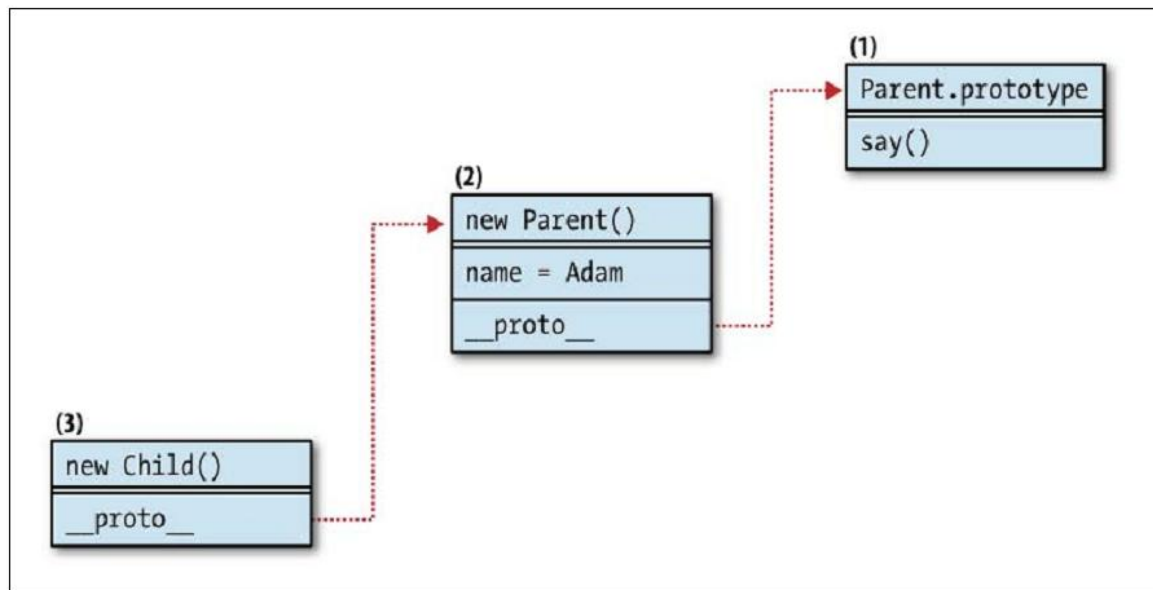


Figure 6-2. Prototype chain after inheritance

Hàm tạo Child() là rỗng và không có thuộc tính được thêm vào Child.prototype; do đó bằng cách sử dụng new Child() tạo ra các đối tượng đẹp đẽ hơn nhiều đối tượng rỗng ngoại trừ liên kết ẩn __proto__. trong trường hợp này __proto__ chỉ tới đối tượng new Parent() object được tạo trong hàm inherit() function.

Giờ cái gì xảy ra khi ta thực hiện kid.say()? Đối tượng Object #3 không có phương thức nào như vậy vì vậy nó tìm kiếm trong #2 thông qua chuỗi prototype. đối tượng #2 cũng không có cái nào như vậy do đó nó đi theo chuỗi prototype lên #1 – đối tượng mà chứa phương thức đó. sau đó trong say() có 1 tham chiếu tới this.name mà cần phải giải quyết. sự tìm kiếm bắt đầu lại 1 lần nữa. trong trường hợp này this chỉ tới object #3 - đối tượng không có name. Object #2 được thăm dò và nó có chứa thuộc tính name với giá trị là “Adam.”

Cuối cùng ta cần xét thêm 1 bước nữa :

```
var kid = new Child();
kid.name = "Patrick";
kid.say(); // "Patrick"
```

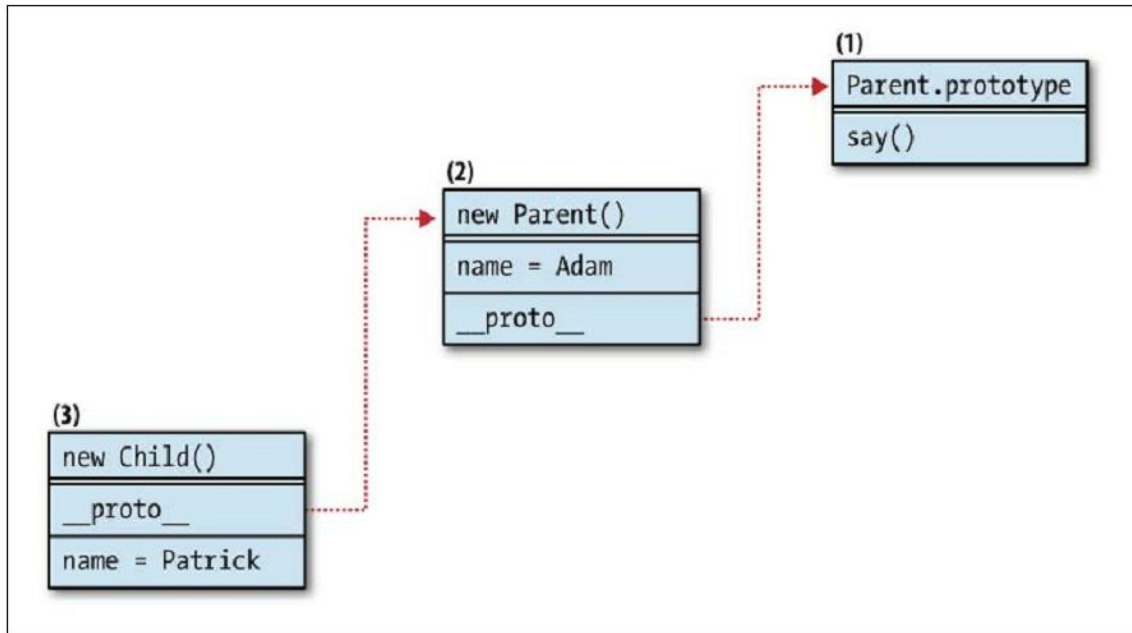


Figure 6-3. Prototype chain after inheritance and adding a property to the child object

Việc thiết lập `kid.name` không thay đổi thuộc tính `name` của object #2, nhưng nó tạo ra 1 thuộc tính chính gốc trực tiếp trong đối tượng `kid` object #3. khi ta thực hiện `kid.say()`, thì phương thức `say` được tìm kiếm trong object #3, sau đó là trong #2 và cuối cùng được tìm thấy trong #1 giống như trước đó. nhưng lúc này việc tìm kiếm `this.name` (hay là `kid.name`) là nhanh bởi vì thuộc tính được tìm thấy ngay tại object #3

Nếu ta xóa thuộc tính mới bằng cách sử dụng `delete kid.name` thì thuộc tính `name` của đối tượng #2 sẽ được lộ ra và được tìm thấy trong sự tìm kiếm nối tiếp

8.4. Nhược điểm khi sử dụng kiểu mẫu #1

1 nhược điểm của kiểu mẫu này là ta kế thừa cả thuộc tính chính gốc được thêm vào `this` và các thuộc tính trong prototype. đa số thời gian ta không muốn thuộc tính chính gốc bởi vì chúng được định nghĩa riêng cho từng thực thể và không có khả năng tái sử dụng

Chú ý : quy luật chung ngón tay cái với hàm tạo là các thành viên có khả năng tái sử dụng nên được thêm vào prototype

1 thứ khác về cách sử dụng 1 hàm `inherit()` function chung là nó không cho phép ta chuyển tiếp các tham số vào hàm tạo `child`, với `child` sau khi được chuyển tiếp vào `parent`. xét ví dụ sau :


```
var s = new Child('Seth');
```

```
s.say(); // "Adam"
```

đây là cái mà ta không mong muốn.điều này có thể cho child chuyển tiếp tham số vào hàm khởi tạo parent nhưng sau đó thì ta phải thực hiện kế thừa mỗi lần mà ta cần 1 đối tượng child mới – cách này không mang lại hiệu năng tốt bởi vì ta kết thúc việc tạo lại các đối tượng parent hơn và hơn nữa

8.5.Kiểu mẫu Classical Pattern #2 - Rent-a-Constructor – kiểu mẫu vay mượn hàm tạo

kiểu mẫu kế tiếp giải quyết vấn đề của việc chuyển tiếp các tham số từ child tới parent.nó vay mượn hàm tạo parent và chuyển tiếp đối tượng child bị ràng buộc với this và cũng đẩy mạnh bất kì tham số nào :

```
function Child(a, c, b, d) {  
  Parent.apply(this, arguments);  
}
```

Cách này ta chỉ có thể kế thừa các thuộc tính được thêm vào this bên trong hàm tạo parent.ta không kế thừa được các thành viên đã được thêm vào trong prototype.

Cách sử dụng kiểu mẫu khởi tạo vay mượn thì các đối tượng children lấy được các **Sao**

chép của các thành viên được kế thừa không giống với kiểu mẫu classical #1 pattern nơi chúng chỉ lấy được các tham chiếu.ví dụ sau sẽ cho ta thấy sự khác biệt :

```
// a parent constructor  
function Article() {  
  this.tags = ['js', 'css'];  
}  
  
var article = new Article();
```

```
// 1 blog post kế thừa từ 1 đối tượng article object
// thông qua kiểu mẫu classical pattern #1
function BlogPost() {}
BlogPost.prototype = article;
var blog = new BlogPost();
//chú ý rằng ở trên ta không cần `new Article()`
//bởi vì ta đã có 1 biến thực thể
//1 trang static page kế thừa từ article
//thông qua kiểu mẫu rented constructor pattern – kiểu mẫu vay mượn hàm khởi tạo
function StaticPage() {
  Article.call(this); // vay mượn toàn bộ mã thực thi trong hàm khởi tạo của hàm Article,
  không vay mượn được bên trong thuộc tính prototype của Article
}
var page = new StaticPage();
alert(article.hasOwnProperty('tags')); // true
alert(blog.hasOwnProperty('tags')); // false
alert(page.hasOwnProperty('tags')); // true
```

trong đoạn mã này thì cha Article() được kế thừa theo 2 cách .kiểu mẫu mặc định gây ra việc đối tượng blog object khuếch đại truy cập thuộc tính tags thông qua prototype, vì vậy nó không có thuộc tính chính gốc và hasOwnProperty() trả về false.đối tượng page object có 1 thuộc tính tags chính gốc bởi vì cách sử dụng hàm tạo vay mượn nên đối tượng mới lấy về được bản sao chép của thành viên tags của hàm tạo cha (sao chép chứ không phải tham chiếu)

chú ý sự khác biệt khi ta hiệu chỉnh thuộc tính tags được kế thừa :

```
blog.tags.push('html');
page.tags.push('php');
alert(article.tags.join(', ')); // "js, css, html"
```

trong ví dụ trên thì đối tượng con blog object hiệu chỉnh thuộc tính tags và cùng cách này nó cũng hiệu chỉnh đối tượng cha bởi vì về bản chất cả blog.tags và article.tags đều chỉ tới cùng 1 mảng array.việc thay đổi trong **page.tags** không có tác dụng với **article** bởi vì **page.tags** là 1 bản sao chép riêng biệt được tạo trong quá trình kế thừa

8.5.1.Chuỗi prototype

ta cần xem xét cách mà chuỗi prototype trông như thế nào khi ta sử dụng kiểu mẫu này và các hàm khởi tạo Parent() và Child() quen thuộc. Child() sẽ được hiệu chỉnh 1 cách nhẹ nhàng như sau :

```
// the parent constructor
```

```
function Parent(name) {
```

```
  this.name = name || 'Adam';
```

```
}
```

```
// adding functionality to the prototype
```

```
Parent.prototype.say = function () {
```

```
  return this.name;
```

```
};
```

```
// child constructor
```

```
function Child(name) {
```

```
  Parent.apply(this, arguments); // sao chép hàm khởi tạo trong hàm Parent và có khả
```

năng truyền thêm vào tham số, tuy nhiên sự sao chép này không xảy ra trong thuộc tính prototype của hàm Parent

```
}
```

```
var kid = new Child("Patrick");
```

```
kid.name; // "Patrick"
```

```
typeof kid.say; // "undefined" // kết quả chứng minh cho sự sao chép không xảy ra tại prototype
```

nếu ta nhìn vào hình 6-4.ta sẽ chú ý rằng không còn 1 liên kết giữa đối tượng new Child object và Parent.đây là bởi vì Child.prototype không được sử dụng ở tất cả và nó đơn giản chỉ tới 1 đối tượng rỗng.bằng cách sử dụng kiểu mẫu này thì kid lấy được thuộc tính chính gốc

Tham số của hàm Parent và hàm Child đều là **name**.tham số giữa 2 hàm này có sự tương đồng

name nhưng phương thức say() không bao giờ được kế thừa và 1 sự cố gắng gọi phương thức này thì kết quả sẽ là 1 lỗi. sự kế thừa là 1 hành động dùng để sao chép các thuộc tính chính gốc của cha như các thuộc tính chính gốc của con và không có liên kết `__proto__` nào được lưu giữ

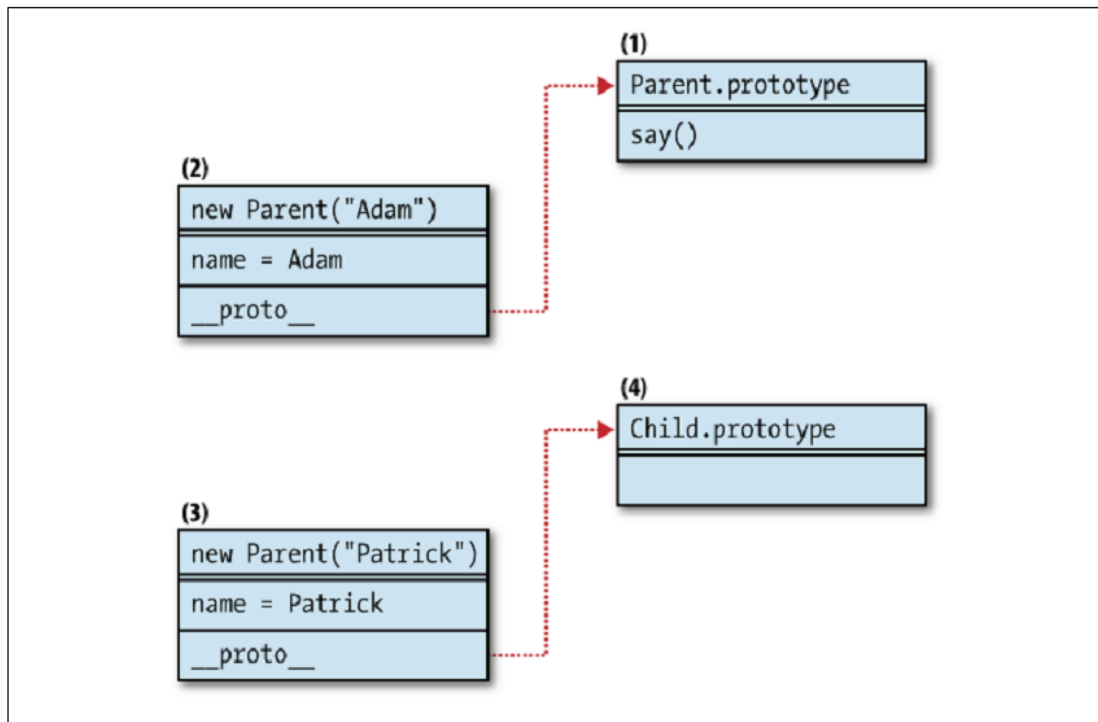


Figure 6-4. The broken chain when using the borrowing constructor pattern

8.5.2. Sự đa kế thừa bằng các hàm tạo vay mượn

Bằng cách sử dụng kiểu mẫu hàm khởi tạo vay mượn thì có thể thực thi sự đa kế thừa đơn giản bằng cách vay mượn từ 1 hay nhiều hơn 1 hàm khởi tạo :

```
function Cat() {
  this.legs = 4;
  this.say = function () {
    return "meowww";
  }
}

function Bird() {
  this.wings = 2;
  this.fly = true;
}
```

```
function CatWings() {  
  Cat.apply(this);  
  
  Bird.apply(this);  
}  
  
var jane = new CatWings();  
console.dir(jane);
```

kết quả được hiện thị trong hình 6-5. bất cứ các thuộc tính di thừa nào sẽ được quyết định bởi cái nào có thuộc tính cuối cùng sẽ thắng



fly	true
legs	4
wings	2
say	function()

Figure 6-5. A CatWings object inspected in Firebug

8.5.3. Ưu điểm và nhược điểm của kiểu mẫu hàm khởi tạo vay mượn

Nhược điểm của kiểu mẫu này là chẳng có thứ gì trong prototype được kế thừa và prototype là nơi dùng để thêm vào các phương thức và thuộc tính có khả năng được tái sử dụng – chúng sẽ không phải tạo lại cho mỗi thực thể

1 ưu điểm là ta lấy về các bản sao chép đúng của các thành viên chính gốc trong hàm cha và không có nguy cơ mà 1 con có thể ngẫu nhiên ghi đè lên 1 thuộc tính của cha

Vì vậy cách nào có thể kế thừa được cả các thuộc tính trong prototype như trong ví dụ trước và cách nào có thể để kid truy cập được phương thức say() ? kiểu mẫu tiếp theo sẽ trả lời cho câu hỏi này

8.6. Kiểu mẫu Classical Pattern #3 - Rent and Set Prototype – kiểu mẫu vay mượn và thiết lập prototype

Kết hợp 2 kiểu mẫu trước bằng cách đầu tiên ta vay mượn hàm tạo và sau đó cũng thiết lập prototype của con chỉ tới 1 thực thể mới của hàm tạo :

```
function Child(a, c, b, d) {
```

```
Parent.apply(this, arguments);
```

```
}
```

```
Child.prototype = new Parent();
```

Ưu điểm là các đối tượng sao chép được các thành viên chính gốc của cha và các tham chiếu tới chức năng có khả năng tái sử dụng của cha (các thành viên trong prototype).child cũng có thể chuyển tiếp bất cứ tham số nào tới hàm khởi tạo cha.cách thức xử lý này hầu như giống nhất với những gì ta mong đợi trong Java; ta kế thừa mọi thứ ở đó trong parent và cũng thời điểm đó ta có thể an toàn hiệu chỉnh các thuộc tính chính gốc mà không gặp nguy cơ nào về sự hiệu chỉnh parent

1 nhược điểm là hàm khởi tạo parent được gọi tới 2 lần vì vậy nó có thể không mang lại hiệu quả về mặt hiệu năng.và cuối cùng các thuộc tính chính gốc (giống như name trong trường hợp của ta) được kế thừa tới 2 lần

Ta xem xét mã code sau và làm 1 vài thử nghiệm :

```
// the parent constructor
function Parent(name) {
  this.name = name || 'Adam';
}
// adding functionality to the prototype
Parent.prototype.say = function () {
  return this.name;
};
// child constructor
function Child(name) {
```

```
Parent.apply(this, arguments);
```

```
}
```

```
Child.prototype = new Parent();
```

```
var kid = new Child("Patrick");
```

```
kid.name; // "Patrick"
```

```
kid.say(); // "Patrick"
delete kid.name;
kid.say(); // "Adam"
```

không giống với kiểu mẫu trước, giờ say() được kế thừa 1 cách chính xác. ta cũng có thể chú ý tới name được kế thừa tới 2 lần và sau khi ta xóa đi 1 cái sao chép thì cái ở chuỗi prototype sẽ lộ ra

hình 6-6 chỉ ra cách mà mối quan hệ giữa các đối tượng objects hoạt động. mỗi quan hệ này tương tự với những cái trong hình 6-3 nhưng nó có đôi chút sự khác biệt

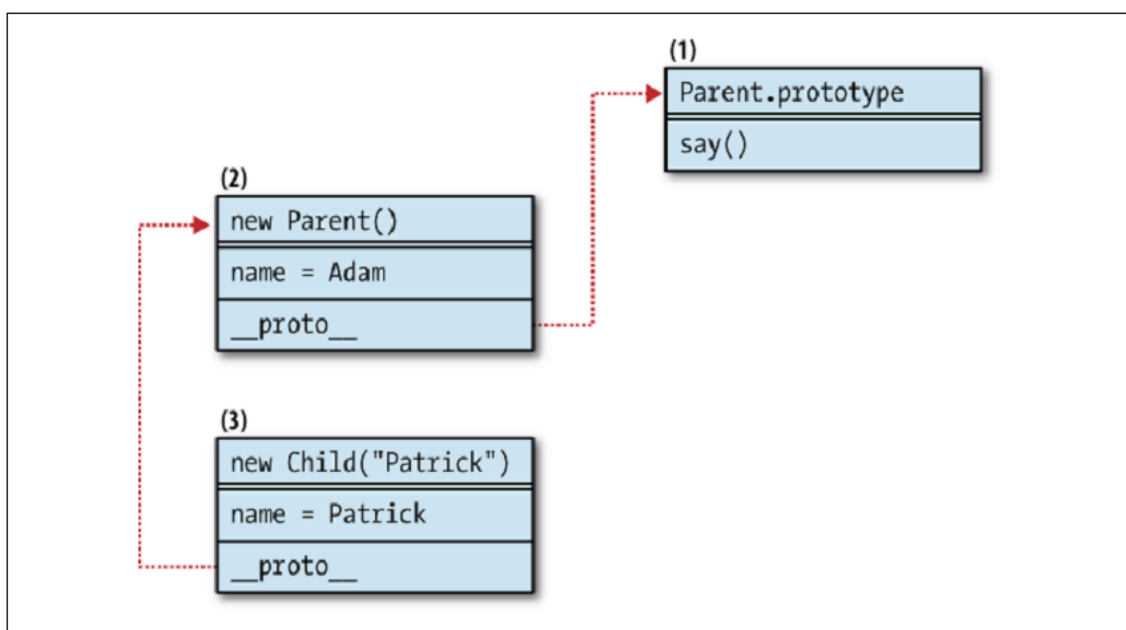


Figure 6-6. The prototype chain is kept in addition to the inherited own members

8.7. Kiểu mẫu Classical Pattern #4 – chia sẻ prototype

Không giống với kiểu mẫu kế thừa hướng classical ở trên – kiểu mẫu đòi hỏi 2 lời gọi tới hàm tạo parent, kiểu mẫu tiếp theo này không thực hiện gọi hàm tạo parent

Định luật ngón tay cái là các thành viên có khả năng tái sử dụng nên được thêm vào prototype và không phải thêm vào this. do vậy với mục đích kế thừa nên bất cứ thứ gì có sự kế thừa nên ở trong prototype. vì vậy ta chỉ cần thiết lập prototype của child là prototype của parent :

```
function inherit(C, P) {
```

```
c.prototype = P.prototype;
```

```
}
```

Điều này mang tới cho ta việc tìm chuỗi prototype ngắn gọn và nhanh bởi vì tất cả các đối tượng thực chất chia sẻ cùng 1 prototype. nhưng cũng có 1 nhược điểm bởi vì nếu 1 child hay 1 grandchild ở 1 vài nơi nào đó phía dưới của chuỗi kế thừa hiệu chỉnh prototype thì nó ảnh hưởng tới tất cả các parents và grandparents.

Như hình 6-7 chỉ ra cả 2 đối tượng child và parent objects chia sẻ cùng 1 prototype và có khả năng truy cập ngang nhau tới phương thức say(). tuy nhiên các đối tượng children không kế thừa thuộc tính name

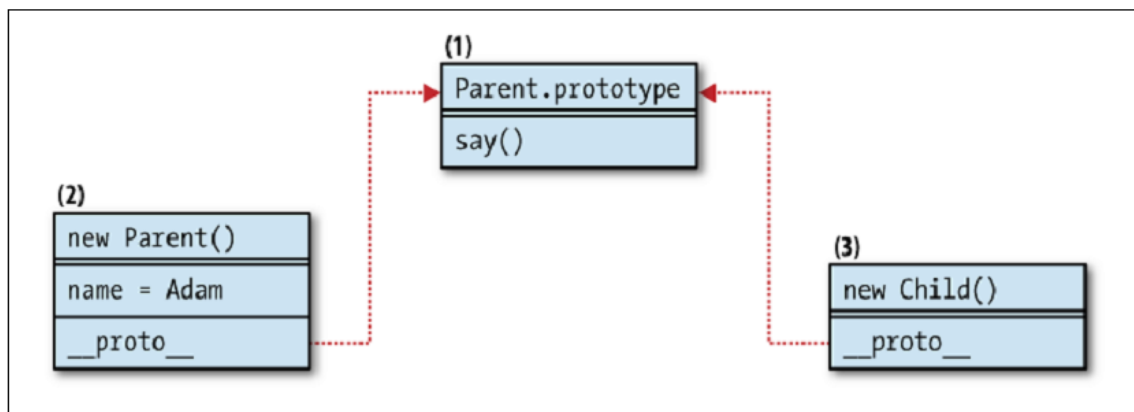


Figure 6-7. Relationships when sharing the same prototype

8.8.Kiểu mẫu Classical Pattern #5 – 1 hàm tạo tạm thời

Kiểu mẫu kế tiếp này giải quyết vấn đề cùng 1 prototype như ở trên bằng cách phá vỡ các liên kết trực tiếp giữa prototype của child và prototype của parent và cùng lúc thu lợi từ chuỗi prototype

Bên dưới là 1 thực thi của kiểu mẫu này nơi mà ta có 1 hàm F() rỗng được dùng phục sự như 1 sự ủy quyền giữa parent và child. thuộc tính prototype của F() chỉ tới thuộc tính prototype của parent. prototype của child là 1 thực thể của 1 hàm rỗng :

```
function inherit(C, P) {  
    var F = function () {};  
    F.prototype = P.prototype;  
    C.prototype = new F();  
}
```



```
}
```

Kiểu mẫu này có 1 thực thi hơi khác với kiểu mẫu mặc định (classical pattern #1) bởi vì tại đây child chỉ kế thừa các thuộc tính của prototype (hình 6-8)

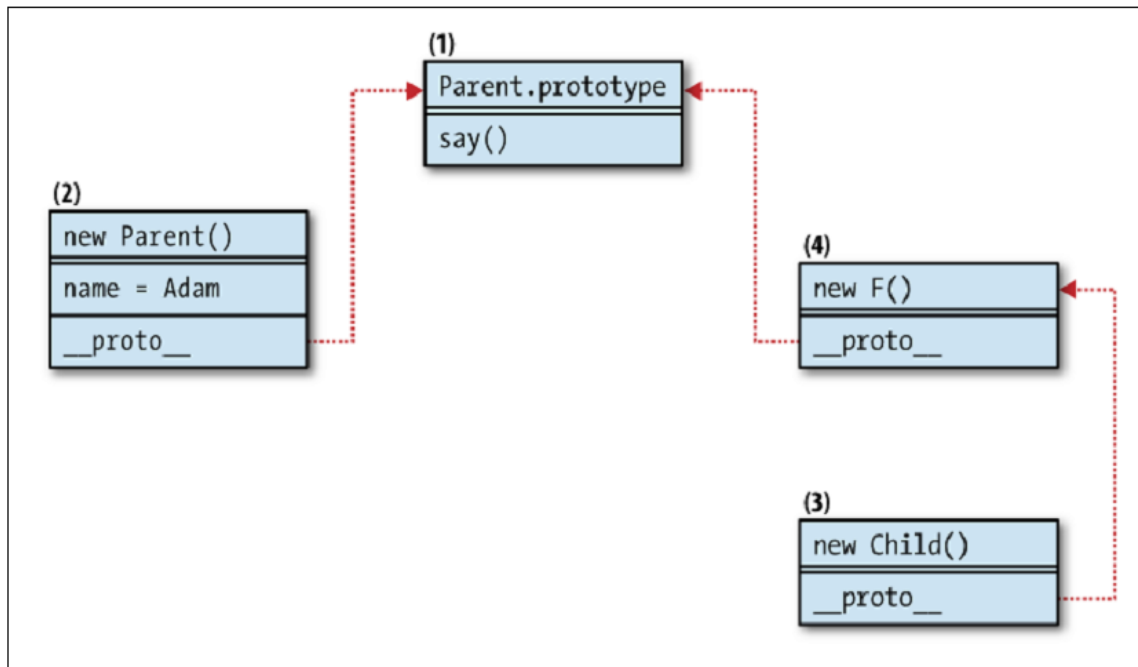


Figure 6-8. Classical inheritance by using a temporary (proxy) constructor F()

Điều này thường là tốt và thực chất rất được ưa thích bởi vì prototype là nơi đặt các chức năng tái sử dụng lại. trong kiểu mẫu này thì bất cứ thành viên nào mà hàm tạo parent thêm vào this sẽ không được kế thừa

Ta tạo ra 1 đối tượng child object mới và kiểm tra cách thực thi của nó:

```
var kid = new Child();
```

nếu ta truy cập vào `kid.name` thì kết quả là `undefined`, trong trường hợp này `name` là 1 thuộc tính chính gốc của parent và trong khi kế thừa chúng ta thực chất không bao giờ gọi `new Parent()` do đó thuộc tính này sẽ không bao giờ được tạo ra. khi ta truy cập vào `kid.say()`, thì nó không có sẵn trong object #3 vì vậy chuỗi prototype sẽ được tìm kiếm. Object #4 không có phương thức này nhưng object #1 lại có và đây là cùng 1 vị trí trong memory – sẽ được sử dụng lại trong tất cả các hàm tạo khác mà kế thừa `Parent()` và được kế thừa trong tất cả các đối tượng được tạo ra bởi tất cả hàm tạo children

8.8.1.Cách lưu trữ Superclass

cách xây dựng phần đầu của kiểu mẫu trước thì ta có thể thêm 1 tham chiếu tới parent gốc.điều này giống với cách truy cập tới superclass trong những ngôn ngữ lập trình khác và có thể nó là 1 cơ hội rất thuận tiện

thuộc tính được gọi là uber bởi vì “super” là 1 từ dành riêng và “superclass” có thể chỉ dẫn những nhà lập trình 1 cách không ngờ vực về việc định hướng nghĩ JS có các lớp classes.đây là 1 cách thực thi cải tiến của kiểu mẫu classical pattern này :

```
function inherit(C, P) {  
  var F = function () {};  
  F.prototype = P.prototype;  
  C.prototype = new F();  
  C.uber = P.prototype;  
}
```

8.8.2.Cách cài đặt lại con trỏ hàm khởi tạo

Điều cuối cùng cần thêm vào hàm kế thừa theo hướng classical gần như hoàn hảo này là cài đặt lại con trỏ chỉ tới hàm khởi tạo trong trường hợp ta cần nó để tìm lại


Nếu ta không reset con trỏ tới hàm khởi tạo thì tất cả các đối tượng con sẽ báo cáo rằng Parent() là hàm khởi tạo của chúng và điều này không hữu dụng.vì vậy sử dụng thực thi inherit() ở trước thì ta có thể phải tuân theo cách thức xử lý sau :

```
// parent, child, inheritance  
function Parent() {}  
function Child() {}  
inherit(Child, Parent);  
  
// testing the waters  
var kid = new Child();  
  
kid.constructor.name; // "Parent"  
  
kid.constructor === Parent; // true
```

thuộc tính tạo constructor hiếm khi được sử dụng nhưng có thể trở nên tiện lợi cho thời gian thực thi của các đối tượng. ta có thể reset nó chỉ tới hàm tạo constructor như mong đợi mà không gây ảnh hưởng gì tới chức năng bởi vì thuộc tính này đa phần là sự truyền tin

phiên bản cuối cùng của kiểu mẫu kế thừa sẽ giống như sau :

```
function inherit(C, P) {  
  var F = function () {};  
  F.prototype = P.prototype;  
  C.prototype = new F();  
  C.uber = P.prototype;  
  C.prototype.constructor = C;  
}
```




Chú ý : kiểu mẫu này cũng được ám chỉ như là 1 kiểu mẫu sử dụng 1 hàm proxy

function hay 1 hàm tạo proxy constructor thay thế cho 1 hàm tạo tạm thời temporary constructor bởi vì hàm tạo tạm thời được sử dụng như 1 proxy để lấy prototype của parent

1 kiểu mẫu tối ưu thường dùng khác tránh việc tạo ra 1 hàm tạo tạm thời mỗi lần ta cần kế thừa. điều này có khả năng chỉ tạo ra hàm tạo tạm thời 1 lần và chỉ thay đổi prototype của hàm này. ta có thể sử dụng 1 hàm tức thời và lưu trữ proxy function ở trong bao đóng của nó :

```
var inherit = (function () {  
  var F = function () {};  
  return function (C, P) {  
    F.prototype = P.prototype;  
    C.prototype = new F();  
    C.uber = P.prototype;  
    C.prototype.constructor = C;  
  }  
})();
```



Quay lại ví dụ về hàm tức thời đã nói đến từ trước thì. F ở đây giống như 1 thuộc tính của inherit. khi inherit được gọi ra lần đầu tiên thì nó tạo ra thuộc tính F ở bên trong. và sau đó khi sử dụng inherit ở các lần tiếp theo thì nó không phải khởi tạo lại nữa và vẫn giữ được liên kết tới thuộc tính F

8.9.Klass

Rất nhiều thư viện JS mô phỏng các classes, thông qua việc giới thiệu về cú pháp sugar syntax mới. các thực thi này khác nhau nhưng chúng thường có 1 vài điểm tương đồng như sau :

- có 1 quy tắc dựa trên cách đặt tên 1 phương thức – được xét tới sự khởi tạo của lớp class ví dụ như initialize, _init, hay 1 vài thứ tương tự và chúng được gọi 1 cách tự động
- các lớp classes kế thừa được từ các lớp classes khác
- có khả năng truy cập được class cha (superclass) từ trong phạm vi lớp class con

ta không đi sâu vào chi tiết mà ta chỉ xét 1 thực thi của các giả lập classes trong JS. đầu tiên, giải pháp được sử dụng từ viễn cảnh của client sẽ như thế nào ?

```
var Man = klass(null, {  
  __construct: function (what) {  
    console.log("Man's constructor");  
    this.name = what;  
  },  
  getName: function () {  
    return this.name;  
  }  
});
```

Cú pháp syntax sugar trong định dạng của 1 hàm function tên là klass(). trong 1 vài thực thi ta rất có thể thấy nó như là khởi tạo Klass() constructor hay như Object.prototype được tham số hóa nhưng trong ví dụ này ta giữ nó là 1 hàm function đơn giản

Hàm function này cần 2 tham số : 1 lớp class cha dùng để được kế thừa và thực thi của lớp class mới bằng cách cung cấp 1 đối tượng nguyên bản. ảnh hưởng bởi PHP, ta thiết lập quy ước này – hàm tạo của lớp class phải là 1 phương thức có tên là __construct. trong đoạn mã trước thì 1 lớp class mới có tên là Man được tạo ra và nó không kế thừa bất cứ thứ gì. lớp Man có 1 thuộc tính chính gốc name được tạo bên trong __construct và 1 phương thức getName(). lớp class này là 1 hàm tạo vì vậy đoạn mã sau vẫn sẽ hoạt động :

```
var first = new Man('Adam'); // logs "Man's constructor"
```

```
first.getName(); // "Adam"
```

giờ ta mở rộng lớp class này và tạo ra 1 lớp Superman :

```
var Superman = klass(Man, {  
  __construct: function (what) {  
    console.log("SuperMan's constructor");  
  },  
  getName: function () {  
    var name = Superman.uber.getName.call(this);  
    return "I am " + name;  
  }  
});
```

Tại đây tham số đầu tiên trong klass() là lớp Man – lớp này sẽ được kế thừa. chú ý trong getName() thì hàm getName() của lớp cha được gọi đầu tiên vì sử dụng thuộc tính static là uber của Superman. ta thực hiện kiểm tra :

```
var clark = new Superman('Clark Kent');  
clark.getName(); // "I am Clark Kent"
```

dòng đầu tiên ghi lại console “Man’s constructor” và sau đó là “Superman’s constructor.” trong 1 vài ngôn ngữ thì khởi tạo của cha được gọi 1 cách tự động mỗi lần khởi tạo của con được gọi, vậy tại sao không giả lập điều đó 1 cách thật tốt ?

kiểm tra bằng toán tử instanceof thì kết quả trả về như mong đợi :

```
clark instanceof Man; // true  
clark instanceof Superman; // true
```

cuối cùng ta xem cách mà hàm klass() được thực thi :

```
var klass = function (Parent, props) {  
  var Child, F, i;  
  // 1.  
  // new constructor
```

```
Child = function () {  
  if (Child.uber && Child.uber.hasOwnProperty("__construct")) {  
    Child.uber.__construct.apply(this, arguments);  
  }  
  if (Child.prototype.hasOwnProperty("__construct")) {  
    Child.prototype.__construct.apply(this, arguments);  
  }  
};  
// 2.  
// inherit  
Parent = Parent || Object;  
F = function () {};  
F.prototype = Parent.prototype;  
Child.prototype = new F();  
Child.uber = Parent.prototype;  
Child.prototype.constructor = Child;  
// 3.  
// add implementation methods  
for (i in props) {  
  if (props.hasOwnProperty(i)) {  
    Child.prototype[i] = props[i];  
  }  
}  
// return the "class"  
return Child;  
};
```

Thực thi `klass()` có 3 sự thú vị và 3 phần tách rời :

1. hàm tạo `Child()` được tạo ra. đây là hàm function được trả về ở cuối và sẽ được sử dụng như là 1 lớp class. trong hàm này thì phương thức `__construct` được gọi nếu nó tồn tại và trước khi được gọi thì phương thức `__construct` của cha được gọi trước (nếu nó tồn tại) bằng cách sử dụng thuộc tính static là `uber`. rất có thể trong nhiều

trường hợp thì uber không được định nghĩa – khi ta kế thừa từ 1 đối tượng object chứ không phải là lớp định nghĩa class

- phần thứ 2 chú trọng vào sự kế thừa. nó đơn giản sử dụng kiểu mẫu kế thừa hướng classical ở trên. và chỉ có 1 thứ mới là cách thiết lập Parent thành Object nếu không có Parent được chuyển tiếp vào
- phần cuối là 1 vòng lặp thông qua tất cả các phương thức thực thi (giống như __construct và getName trong ví dụ) – đây là định nghĩa thực tế của lớp class và thêm chúng vào prototype của con

khi nào ta sử dụng 1 kiểu mẫu giống như trên ? câu trả lời là nó thực sự tốt hơn nếu ta tránh sử dụng nó bởi vì nó mang tới toàn bộ sự lẫn lộn của các kí hiệu lớp classes – và những thứ đó không tồn tại trong JS. nó thêm vào cú pháp mới và các luật mới để phải học và phải nhớ. như đã nói nếu ta hay nhóm của ta cảm thấy thoải mái với các lớp classes và không thoải mái với prototypes thì điều này có thể được dùng. kiểu mẫu này cho phép ta quên hoàn toàn về prototype và điều tốt là ta có thể sử dụng 1 cách vận dụng cú pháp và quy ước giống với những ngôn ngữ lập trình khác

8.10. Sự kế thừa hướng Prototypal

nếu đã đề cập từ trước về các kiểu mẫu “modern” không động tới việc giả lập class thì những kiểu mẫu như thế này được gọi là prototypal inheritance. tại đây các đối tượng objects được kế thừa từ các đối tượng objects khác. ta có thể nghĩ về nó theo cách này : ta có 1 đối tượng object mà ta muốn sử dụng lại và ta muốn tạo ra 1 đối tượng thứ 2 để lấy các chức năng từ đối tượng thứ nhất

đây là cách mà ta dễ hình dung ra về nó như sau :

```
// object to inherit from
var parent = {
  name: "Papa"
};
// the new object
var child = object(parent);
// testing
alert(child.name); // "Papa"
```

trong đoạn mã trước, ta có 1 đối tượng đã có sẵn tên là parent được tạo ra bởi theo cách tạo đối tượng nguyên bản và ta muốn tạo ra 1 đối tượng khác gọi là child muốn có cùng các phương thức và thuộc tính như parent. đối tượng child được tạo ra với 1 hàm function được gọi là object().hàm này không có thật trong JS (đừng hiểu nhầm sang hàm tạo Object())

trong tự kiểu mẫu hướng classical ta có thể sử dụng 1 hàm tạo tạm thời rỗng là F().sau đó ta thiết lập prototype của F() là đối tượng parent.cuối cùng ta trả về 1 thực thể mới trong hàm tạo tạm thời :

```
function object(o) {  
  
    function F() {}  
    F.prototype = o;  
    return new F();  
}
```

Hình 6-9 chỉ ra chuỗi prototype khi sử dụng kiểu mẫu kế thừa hướng prototypal.tại đây child luôn luôn bắt đầu như là 1 đối tượng rỗng – đối tượng không có thuộc tính của chính nó nhưng cùng thời điểm đó nó có tất cả các chức năng trong cha của nó bởi ưu điểm của __proto__

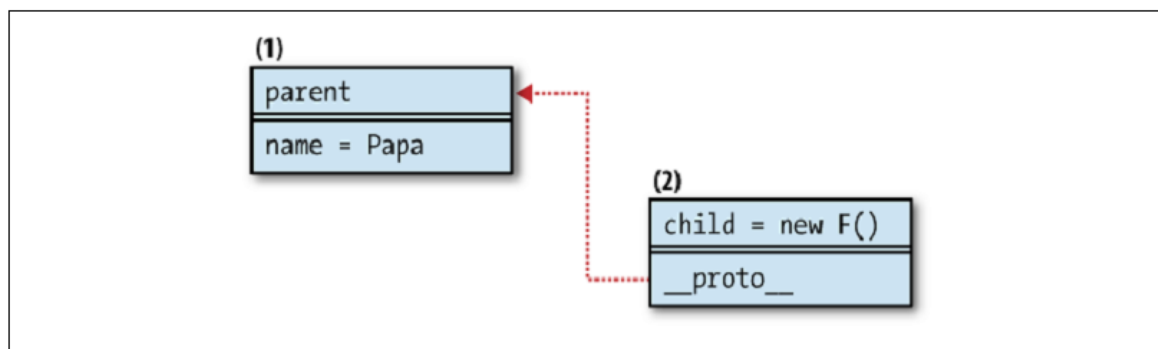


Figure 6-9. Prototypal inheritance pattern

8.10.1.Thảo luận

Trong kiểu mẫu kế thừa hướng prototypal này thì cha parent không cần được tạo ra với kiểu tạo nguyên văn.ta có thể có các hàm tạo để tạo ra parent.chú ý rằng nếu ta làm như vậy thì cả thuộc tính chính gốc và thuộc tính prototype của hàm tạo sẽ đều được kế thừa :

```
// parent constructor
```



```
function Person() {  
  // an "own" property  
  this.name = "Adam";  
}  
// a property added to the prototype  
Person.prototype.getName = function () {  
  return this.name;  
};  
// create a new person  
var papa = new Person();  
// inherit  
var kid = object(papa);  
  
// test that both the own property  
// and the prototype property were inherited  
kid.getName(); // "Adam"
```

1 dạng biến đổi khác của kiểu mẫu này là ta có lựa chọn để kế thừa chỉ đối tượng prototype của 1 hàm tạo đã có. nhớ rằng, các đối tượng kế thừa từ đối tượng khác bất chấp cách mà đối tượng cha parent được tạo. đây là 1 minh họa sử dụng ví dụ trước nhưng có 1 chút thay đổi :

```
// parent constructor  
function Person() {  
  // an "own" property  
  this.name = "Adam";  
}  
// a property added to the prototype  
Person.prototype.getName = function () {  
  return this.name;  
};  
// inherit  
var kid = object(Person.prototype);  
  
typeof kid.getName; // "function", because it was in the prototype
```

`typeof kid.name; // "undefined", because only the prototype was inherited`

8.10.2. Việc bổ sung ECMAScript 5

trong ECMAScript 5 thì kiểu mẫu kế thừa hướng prototypal trở thành 1 phần chính thức của ngôn ngữ. kiểu mẫu này được thực thi thông qua phương thức `Object.create()`. theo cách khác ta không cần làm hàm function giống với `object()` vì nó sẽ được xây dựng sẵn trong ngôn ngữ :

```
var child = Object.create(parent);
```

`Object.create()` nhận 1 tham số bổ sung là 1 đối tượng `object`. thuộc tính trong đối tượng bổ sung này sẽ được thêm vào như thuộc tính chính gốc của đối tượng `child` mới. đây là 1 quy ước cho phép ta kế thừa và xây dựng nên đối tượng `child` với 1 lời gọi phương thức. ví dụ :

```
var child = Object.create(parent, {  
  age: { value: 2 } // ECMA5 descriptor  
});  
child.hasOwnProperty("age"); // true
```

ta cũng có thể tìm thấy kiểu mẫu kế thừa hướng prototypal được thực thi trong các thư viện JS như YUI3 có phương thức `Y.Object()` như sau :

```
YUI().use('*', function (Y) {  
  var child = Y.Object(parent);  
});
```

8.10.3. Sự kế thừa bằng cách sao chép các thuộc tính

Ta xem xét 1 kiểu mẫu kế thừa khác - kế thừa bằng cách sao chép các thuộc tính. trong kiểu mẫu này thì 1 đối tượng `object` lấy các chức năng từ 1 đối tượng khác đơn giản là sao chép chúng. đây là 1 thực thi mẫu của hàm `extend()` ;

```
function extend(parent, child) {  
  var i;  
  child = child || {};
```

```
for (i in parent) {
```

```
  if (parent.hasOwnProperty(i)) {  
    child[i] = parent[i];  
  }  
}  
return child;  
}
```

Đây là 1 thực thi đơn giản và nó chỉ là lặp thông qua các thành viên của parent và sao chép chúng. trong thực thi này thì child là không bắt buộc; nếu ta không chuyển tiếp 1 đối tượng có sẵn là tham số thì 1 đối tượng mới được tạo và trả về.

```
var dad = {name: "Adam"};  
var kid = extend(dad);  
kid.name; // "Adam"
```

Không cần truyền thêm số child vào hàm extend. hàm này tự động trả về 1 đối tượng {...} và nó được gán vào ngay trong kid

thực thi như thế này gọi là “shallow copy” của 1 đối tượng. mặt khác 1 sao chép chuyên sâu deep copy có nghĩa là tiến hành kiểm tra nếu thuộc tính mà ta sẽ copy xem có là 1 đối tượng hay 1 mảng array, và nếu vậy thì lập 1 cách đệ quy thông qua các thuộc tính và sao chép chúng. với shallow copy (bởi vì các đối tượng được chuyển tiếp như là tham chiếu trong JS) nên nếu ta thay đổi 1 thuộc tính của child và **thuộc tính này là 1 đối tượng object** thì nó sẽ bị thay đổi trong parent. ta xét ví dụ sau :

```
var dad = {  
  counts: [1, 2, 3],  
  reads: {paper: true}  
};  
var kid = extend(dad);  
kid.counts.push(4);  
dad.counts.toString(); // "1,2,3,4"  
dad.reads === kid.reads; // true
```

giờ ta hiệu chỉnh lại hàm `extend()` để khiến nó tạo ra deep copies. tất cả những gì ta cần là kiểm tra xem 1 thuộc tính có là 1 đối tượng hay không và nếu đúng là vậy thì sao chép thuộc tính của nó 1 cách đệ quy. hay 1 kiểm tra khác là ta cần kiểm tra xem đối tượng là đối tượng object thật sự hay là 1 mảng array. do đó phiên bản deep copy của `extend()` sẽ như sau :

```
function extendDeep(parent, child) {
  var i,
  toStr = Object.prototype.toString,
  astr = "[object Array]";
  child = child || {};
  for (i in parent) {
    if (parent.hasOwnProperty(i)) {
      if (typeof parent[i] === "object") {
        child[i] = (toStr.call(parent[i]) === astr) ? [] : {}; // kiểm tra xem thuộc tính là 1 đối tượng
        hay 1 mảng array
        extendDeep(parent[i], child[i]); // thực hiện đệ quy
      } else {
        child[i] = parent[i];
      }
    }
  }
  return child;
}
```

Giờ ta kiểm tra thực thi mới có cho ta các bản copies đúng của các đối tượng objects vì vậy các đối tượng `child` không hiệu chỉnh lại các cha của chúng :

```
var dad = {
  counts: [1, 2, 3],
  reads: {paper: true}
};
var kid = extendDeep(dad);
```

```
kid.counts.push(4);
kid.counts.toString(); // "1,2,3,4"
dad.counts.toString(); // "1,2,3"
dad.reads === kid.reads; // false
kid.reads.paper = false;
kid.reads.web = true;
dad.reads.paper; // true
```

kiểu mẫu property copying pattern này đơn giản và được sử dụng rộng rãi ví dụ như Firebug có 1 thuộc tính tên là extend() cho phép tạo ra các shallow copies và extend() của jQuery tạo ra 1 bản deep copy. YUI3 đưa ra 1 phương thức tên là Y.clone(), dùng để tạo ra 1 bản deep copy và có thể copies cả những function được liên kết với chúng vào child object

chú ý rằng : không có bao hàm prototypes trong kiểu mẫu này mà nó chỉ là các đối tượng và các thuộc tính chính gốc của chúng

8.10.4.Mix-ins

Để thay thế cho việc sao chép từ 1 đối tượng object ta có thể sao chép từ bất cứ số objects nào và kết hợp tất cả chúng lại trong 1 đối tượng mới

Thực thi này cũng khá đơn giản và chỉ cần lặp thông qua các tham số và sao chép mọi thuộc tính của mọi đối tượng được chuyển tiếp vào hàm function :

```
function mix() {
  var arg, prop, child = {};
  for (arg = 0; arg < arguments.length; arg += 1) { // thực hiện lặp thông qua tất cả các tham số
    arguments được truyền vào hàm function này
    for (prop in arguments[arg]) {
      if (arguments[arg].hasOwnProperty(prop)) { // mỗi tham số này đều là 1 kiểu object
        child[prop] = arguments[arg][prop];
      }
    }
  }
  return child;
}
```

Giờ ta có 1 hàm mix-in function tổng quát và ta có thể chuyển tiếp bất cứ số lượng đối tượng objects nào vào trong nó và kết quả sẽ là 1 đối tượng object mới mà chứa tất cả các thuộc tính của các đối tượng objects tài nguyên

```
var cake = mix(  
  { eggs: 2, large: true },  
  { butter: 1, salted: true },  
  { flour: "3 cups" },  
  { sugar: "sure!" }  
);
```

Hình 6-10 chỉ ra kết quả của sự hiện thị các thuộc tính của đối tượng cake mới đã được mixed-in bằng cách thực thi console.dir(cake) trong Firebug console.



butter	1
eggs	2
flour	"3 cups"
large	true
salted	true
sugar	"sure!"

Figure 6-10. Examining the cake object in Firebug

Chú ý : nếu ta sử dụng khái niệm mix-in từ các ngôn ngữ mà trong đó khái niệm này là 1 phần chính thức thì ta có thể mong muốn rằng việc thay đổi 1 hay nhiều parents sẽ gây ảnh hưởng tới đối tượng child nhưng điều này không đúng với thực thi đã cho này, tại đây, ta đơn giản chỉ lập và sao chép các thuộc tính chính gốc và phá vỡ liên kết với các cha parent(s)

8.10.5. Vay mượn các phương thức Methods

1 vài lúc rất có thể điều này sẽ xảy ra, ta có thể chỉ vay mượn 1 hay 2 phương thức từ 1 đối tượng đã có. ta muốn sử dụng lại chúng nhưng ta không thực sự muốn định dạng 1 mối quan hệ parent-child với đối tượng đó. ta muốn sử dụng chỉ những phương thức mà ta muốn mà không phải kế thừa lại toàn bộ các phương thức khác mà ta sẽ không bao giờ sử dụng. điều

này là có thể với việc sử dụng kiểu mẫu vay mượn borrowing methods pattern – sử dụng ưu điểm của việc sử dụng phương thức call() và apply().ta đã thấy kiểu mẫu này được sử dụng trong hàm extendDeep()

Như ta biết thì các hàm functions trong JS đều là các đối tượng objects và chúng có sẵn 1 vài phương thức thú vị giống như call() và apply().ta có thể sử dụng cả 2 phương thức này để vay mượn các chức năng từ 1 đối tượng có sẵn :

```
// call() example
notmyobj.doStuff.call(myobj, param1, p2, p3);
// apply() example
notmyobj.doStuff.apply(myobj, [param1, p2, p3]);
```

tại đây ta có 1 đối tượng được gọi là myobj và ta biết rằng 1 đối tượng khác tên là notmyobj sử dụng phương thức hữu dụng này là doStuff().thay thế cho việc kế thừa và kế thừa 1 số các phương thức mà đối tượng myobj sẽ không bao giờ cần đến, ta có thể đơn giản vay mượn phương thức doStuff() 1 cách tạm thời

ta chuyển tiếp đối tượng của ta và các bất cứ tham số nào và các phương thức được vay mượn kết hợp với đối tượng của ta giống như thuộc tính this của chính nó.1 cách cơ bản, đối tượng của ta giả vờ là 1 đối tượng khác để thu được ưu điểm từ phương thức mà ta muốn.nó giống như lấy 1 sự kế thừa nhưng không trả phí cho cước thuê kế thừa (với cước thuê ở đây là các thuộc tính và phương thức bổ sung mà ta không cần)

8.10.6.Ví dụ : vay mượn từ 1 mảng array

1 cách sử dụng thông thường cho kiểu mẫu này là vay mượn các phương thức của mảng array.

Mảng array có các phương thức hữu dụng mà các đối tượng objects kiểu mảng array như arguments thì không có.vì vậy arguments có thể vay mượn các phương thức của mảng array giống như phương thức slice().đây là 1 ví dụ :

```
function f() {
var args = [].slice.call(arguments, 1, 3);
return args;
}
// example
```

```
f(1, 2, 3, 4, 5, 6); // returns [2,3]
```

trong ví dụ này 1 mảng array rỗng được tạo ra chỉ vì mục đích sử dụng phương thức của nó. 1 cách hơi dài khác để làm điều tương tự là mượn phương thức trực tiếp từ prototype của Array, bằng cách sử dụng `Array.prototype.slice.call(...)`, cách này dài hơn 1 tí nhưng ta sẽ tiết kiệm được việc phải tạo ra 1 mảng array rỗng

8.10.7.Vay mượn và ràng buộc

khi vay mượn các phương thức thông qua `call()` hoặc `apply()` hay đơn giản thông qua phép gán thì đối tượng mà `this` chỉ tới bên trong phương thức được vay mượn được xác định dựa vào biểu thức gọi. nhưng thỉnh thoảng cách tốt nhất có giá trị của `this` bị khóa hay bị giới hạn trong 1 đối tượng cụ thể và được xác định trước

ta xét 1 ví dụ. đây là 1 đối tượng object gọi là `one` – đối tượng có 1 phương thức `say()` :

```
var one = {  
  name: "object",  
  say: function (greet) {  
    return greet + ", " + this.name;  
  }  
};  
// test  
one.say('hi'); // "hi, object"
```

giờ 1 đối tượng khác là `two` không có phương thức `say()` nhưng nó có thể vay mượn phương thức này từ `one` :

```
var two = {  
  name: "another object"  
};  
one.say.apply(two, ['hello']); // "hello, another object"
```

trong trường hợp trên thì `this` bên trong phương thức `say()` chỉ tới `two` và `this.name` do vậy sẽ là “another object.”. nhưng kịch bản sẽ như thế nào nếu ta gán hàm function chỉ tới 1 biến

toàn cục hay ta chuyển tiếp hàm function như 1 **hàm callback** ? trong lập trình hướng client thì có nhiều sự kiện events và hàm callbacks do vậy điều này xảy ra như sau :

```
// assigning to a variable
```

```
// `this` will point to the global object
```

```
var say = one.say;
```

```
say('hoho'); // "hoho, undefined" →
```

```
// passing as a callback
```

```
var yetanother = {
```

```
  name: "Yet another object",
```

```
  method: function (callback) {
```

```
    return callback('Hola');
```

```
  }
```

```
};
```

```
yetanother.method(one.say); // "Holla, undefined"
```

one.say là 1 hàm function – hay nó chỉ là 1 phương thức của đối tượng one mà thôi, chứ không phải là cả đối tượng one, do đó thuộc tính name có trong đối tượng one thì one.say không có do đó this trong one.say chỉ tới đối tượng global object → this.name = **undefined** – xem lại trang 12

trong trường hợp này this bên trong say() chỉ tới đối tượng toàn cục global object và toàn bộ đoạn mã không hoạt động như mong đợi. để sửa lại (ràng buộc) 1 đối tượng với 1 phương thức thì ta có thể sử dụng 1 hàm đơn giản như sau :

```
function bind(o, m) {
```

```
  return function () {
```

```
    return m.apply(o, [].slice.call(arguments));
```

```
  };
```

```
}
```

Hàm bind() này nhận 1 đối tượng object là o và 1 phương thức là m, ràng buộc chúng với nhau và sau đó trả về 1 hàm function khác. hàm function được trả về truy cập o và m thông qua 1 bao đóng. do vậy ngay sau khi bind() trả về thì hàm function bên trong sẽ truy cập o và m – những thứ mà luôn chỉ tới đối tượng và phương thức gốc. ta tạo ra 1 hàm function mới bằng cách sử dụng bind() :

```
var twosay = bind(two, one.say);
```

```
twosay('yo'); // "yo, another object"
```

như ta có thể thấy mặc dù `twosay()` được tạo như là 1 hàm toàn cục thì this cũng không chỉ tới đối tượng toàn cục mà nó chỉ tới đối tượng `two` – đối tượng mà được chuyển tiếp vào `bind()`. bất chấp cách mà ta gọi `twosay()` thì this luôn chỉ vào `two`.

Cái giá mà ta trả cho sự xa xỉ này là có 1 ràng buộc là 1 bao đóng bổ sung

8.10.8.Function.prototype.bind()

ECMAScript 5 thêm 1 phương thức `bind()` vào `Function.prototype`, và nó cũng sử dụng dễ dàng như sử dụng `apply()` và `call()`. do vậy ta có thể thực hiện biểu thức giống như sau :

```
var newFunc = obj.someFunc.bind(myobj, 1, 2, 3);
```

điều này có nghĩa ràng buộc `someFunc()` và `myobj` với nhau và cũng điền trước 3 tham số đầu tiên mà `someFunc()` mong chờ. đây cũng là 1 ví dụ về ứng dụng hàm tách rời `partial function application` như đề cập ở phần trước

ta xét cách mà ta có thể thực thi `Function.prototype.bind()` khi chương trình của ta chạy trong môi trường pre-ES5 :

```
if (typeof Function.prototype.bind === "undefined") {  
  Function.prototype.bind = function (thisArg) {  
    var fn = this,  
        slice = Array.prototype.slice,  
        args = slice.call(arguments, 1);  
    return function () {  
      return fn.apply(thisArg, args.concat(slice.call(arguments)));  
    };  
  };  
}
```

Thực thi này hầu như tương tự và nó sử dụng ứng dụng tách rời và móc nối vào danh sách các tham số - các tham số này được chuyển tiếp vào `bind()` ngoại trừ tham số đầu tiên và chúng được chuyển tiếp khi 1 hàm function mới được trả về bởi `bind()` được gọi sau đó. đây là 1 ví dụ :

```
var twosay2 = one.say.bind(two);  
twosay2('Bonjour'); // "Bonjour, another object"
```

trong ví dụ trên, ta không chuyển tiếp bất cứ tham số vào nào bind() hơn là đối tượng được giới hạn. trong ví dụ tiếp theo, ta chuyển tiếp 1 tham số được áp dụng 1 cách từng phần :

```
var twosay3 = one.say.bind(two, 'Enchanté');  
twosay3(); // "Enchanté, another object"
```

IX. Kiểu mẫu thiết kế

9.1.Singleton – kiểu mẫu duy nhất

ý tưởng của kiểu mẫu singleton pattern là chỉ có duy nhất 1 thực thể của 1 lớp class cụ thể.điều này có nghĩa là ở lần thứ 2, ta sử dụng cùng 1 lớp class để tạo ra 1 đối tượng object mới thì ta chỉ nhận về cùng 1 đối tượng object như đã tạo trong lần đầu tiên

và cách áp dụng điều này vào JS như thế nào ? trong JS, không có các lớp classes chỉ có các đối tượng objects.khi ta tạo 1 đối tượng object mới thì thực chất không có đối tượng nào giống như vậy và đối tượng mới vừa được tạo chính là 1 singleton (duy nhất).việc tạo 1 đối tượng đơn giản bằng cách sử dụng object literal cũng là 1 ví dụ của 1 singleton:

```
var obj = {  
  myprop: 'my value'  
};
```

Trong JS, thì các đối tượng không bao giờ là tương đương trừ khi chúng là cùng 1 đối tượng vì vậy ngay cả nếu ta tạo 1 đối tượng tương đồng với cùng chính xác các thành viên thì nó cũng **không phải** là cái đầu tiên :

```
var obj2 = {  
  myprop: 'my value'  
};  
obj === obj2; // false  
obj == obj2; // false
```

do vậy ta có thể nói rằng mọi lúc ta tạo ra 1 đối tượng bằng cách sử dụng object literal thì thực chất ta tạo ra 1 singleton mà không cần phải có cú pháp đặc biệt nào

chú ý : thỉnh thoảng khi mọi người nói tới “singleton” trong ngữ cảnh JS thì cũng có nghĩa là kiểu mẫu module pattern

9.1.1.Cách sử dụng new

Js không có các lớp classes do vậy định nghĩa nguyên văn cho Singleton không có ý nghĩa 1 cách kỹ thuật. nhưng JS có cú pháp new cho việc tạo ra các đối tượng bằng cách sử dụng hàm tạo và thỉnh thoảng ta có thể muốn 1 thực thi Singleton sử dụng cú pháp này. ý tưởng là khi ta sử dụng new để tạo ra 1 vài đối tượng bằng cách sử dụng cùng 1 hàm khởi tạo thì ta chỉ lấy duy nhất các con trỏ mới chỉ tới chính xác cùng 1 đối tượng

Đoạn mã sau chỉ ra cách thức xử lý như mong đợi (giả định rằng ta gạt bỏ ý tưởng của Multiverse và nhận ra rằng chỉ có duy nhất 1 Universe ở đó) :

```
var uni = new Universe();  
var uni2 = new Universe();  
uni === uni2; // true
```

trong ví dụ này thì uni được tạo vào thời điểm đầu tiên hàm tạo được gọi. vào thời điểm thứ 2 (thứ 3, 4,...) thì đối tượng cùng uni được trả về. đây là lí do tại sao uni === uni2. bởi vì chúng về bản chất là 2 tham chiếu chỉ tới chính xác cùng 1 đối tượng object và cách để đặt được điều này trong JS như thế nào ?

t cần hàm tạo Universe cache thực thể đối tượng this khi nó được tạo và sau đó trả về chính nó ở thời điểm thứ 2 hàm tạo được gọi. ta có 1 vài sự chọn lựa để thu được kết quả như vậy :

- ta có thể sử dụng 1 biến toàn cục để lưu trữ thực thể. điều này không được khuyến dùng bởi vì theo nguyên tắc chung thì toàn cục là tồi tệ. thêm vào đó bất cứ ai cũng có thể ghi đè lên biến toàn cục này và nó có thể dẫn tới tai họa. vì vậy ta không bàn tới lựa chọn này thêm nữa
- ta có thể cache trong 1 thuộc tính static của hàm tạo. các hàm functions trong JS đều là các đối tượng vì vậy chúng có thể có các thuộc tính. ta có thể có 1 vài thứ giống như Universe.instance dùng để cache đối tượng ở đó. đây là 1 cách làm tốt, 1 giải pháp rõ ràng với chỉ duy nhất 1 nhược điểm là thuộc tính instance được truy cập công khai và mã code bên ngoài có thể thay đổi nó do vậy ta có thể mất thực thể này
- ta có thể đóng gói nó vào 1 bao đóng. điều này giữ thực thể riêng tư và không hiện ra cho sự hiệu chỉnh ở bên ngoài hàm tạo

ta xét 1 ví dụ thực thi theo lựa chọn 2 và 3

9.1.2. Thực thể trong 1 thuộc tính static

đây là 1 ví dụ về việc caching 1 thực thể chính quy trong 1 thuộc tính static của hàm tạo

Universe :

```
function Universe() {  
  // do we have an existing instance?  
  // kiểm tra xem thực thể đã tồn tại hay chưa  
  if (typeof Universe.instance === "object") {  
    return Universe.instance;  
  }  
  // proceed as normal  
  this.start_time = 0;  
  this.bang = "Big";  
  // cache  
  Universe.instance = this;  
  // implicit return:  
  // return this;  
}  
// testing  
var uni = new Universe();  
var uni2 = new Universe();  
uni === uni2; // true
```

ở lần tạo thứ 2, sau mệnh đề if kiểm tra nhận thấy **Universe.instance** đã là 1 đối tượng rồi. do đó ta gán đối tượng uni2 thành **Universe.instance** – trong thuộc tính này chứa tham chiếu tới uni. do vậy uni2 tham chiếu tới uni. nên uni2 và uni chính là 1 đối tượng



Sau lần khởi tạo đầu tiên thì đối tượng uni được thực thể hóa từ hàm tạo Universe sẽ được gán vào bên trong 1 thuộc tính static của chính hàm tạo (do là static nên nó không bị tạo mới mỗi lần ta thực thể hóa). thuộc tính **Universe.instance** sau khi được gán sẽ chứa 1

như ta thấy, đây là 1 giải pháp hiển nhiên với chỉ duy nhất 1 nhược điểm là instance là 1 public.

9.1.3. Thực thể trong 1 bao đóng

Cách khác để tạo ra singleton theo hướng class là sử dụng 1 bao đóng để bảo vệ thực thể đơn nhất. ta có thể thực thi điều này bằng cách sử dụng 1 thành viên riêng tư private static như đã nhắc đến ở chương trước điều thú vị bí mật ở đây là ghi đè hàm tạo :

```
function Universe() {  
  // the cached instance  
  var instance = this; // thuộc tính riêng tư
```

```
// proceed as normal
this.start_time = 0;
this.bang = "Big";
// ghi đè lại hàm khởi tạo
```

```
Universe = function () {
```

```
return instance;
```

```
};
```

```
}
```

```
// testing
```

```
var uni = new Universe();
```

```
var uni2 = new Universe();
```

```
uni === uni2; // true
```

Ý tưởng cách làm giống hệt như trên. cũng là gán 1 tham chiếu tới đối tượng đầu tiên vào 1 biến, sau đó thực hiện gán đối tượng thứ 2 vào biến đó. do vậy khi này 2 đối tượng cùng tham chiếu tới cùng 1 đối tượng

hàm tạo gốc được gọi vào thời điểm đầu tiên và trả về như bình thường. sau đó lần thứ 2, lần thứ 3 thì hàm tạo bị ghi đè. hàm tạo bị ghi đè có thể truy cập biến instance riêng tư thông qua bao đóng và đơn giản trả về chính nó

thực thi này thực chất là 1 ví dụ khác của kiểu mẫu hàm tự định nghĩa self-defining function pattern. nhược điểm của kiểu mẫu này như ta đã nói đến từ trước là hàm function được ghi đè lại sẽ mất đi bất kì thuộc tính nào được thêm vào nó giữa thời điểm định nghĩa ban đầu và được định nghĩa lại. trong ví dụ riêng any thì bất cứ thứ gì ta thêm vào prototype của Universe() thì sẽ không có 1 liên kết tồn tại chỉ tới thực thể mà được tạo bởi thực thi gốc

đây là cách ta có thể nhìn ra vấn đề với vài mẫu thử nghiệm :

```
// thêm vào prototype
```

```
Universe.prototype.nothing = true; // do prototype này được thêm vào trước khi
```

```
khởi tạo đối tượng đầu tiên nên nó là prototype được thêm vào hàm Universe gốc
```

```
var uni = new Universe();
```

```
// 1 lần nữa thêm vào prototype
```

```
// sau khi đối tượng ban đầu được tạo
```

`Universe.prototype.everything = true;` // prototype này được thêm vào giữa thời điểm khởi tạo xong đối tượng đầu tiên và trước khi khởi tạo đối tượng thứ 2. do đó nó sẽ không tồn tại

```
var uni2 = new Universe();
```

ta thử nghiệm :

```
//chỉ có prototype gốc được liên kết với objects
```

```
uni.nothing; // true
```

```
uni2.nothing; // true
```

```
uni.everything; // undefined
```

```
uni2.everything; // undefined
```

```
// điều này có vẻ đúng
```

```
uni.constructor.name; // "Universe"
```

```
// tuy nhiên lại không phải vậy
```

```
uni.constructor === Universe; // false
```

nguyên nhân là vì `uni.constructor` không còn là hàm tạo `Universe()` constructor là bởi vì `uni.constructor` vẫn chỉ tới hàm tạo gốc (hàm tạo ban đầu) chứ không phải chỉ tới hàm tạo đã được định nghĩa lại

nếu việc lấy về prototype và con trỏ hàm tạo phải như đúng mong muốn thì ta có thể thực hiện điều này với 1 vài mẹo sau :

```
function Universe() {
```

```
// the cached instance
```

```
var instance;
```

```
// ghi đè lại hàm tạo
```

```
Universe = function Universe() {
```



```
return instance;
};
// mang các thuộc tính prototype sang
Universe.prototype = this;
// the instance
instance = new Universe();
// reset lại con trỏ hàm khởi tạo
instance.constructor = Universe;
// all the functionality
instance.start_time = 0;
instance.bang = "Big";
return instance;
}
```

Giờ tất cả các bài kiểm tra của ta sẽ hoạt động như mong đợi :

```
// cập nhật prototype và tạo ra thực thể
Universe.prototype.nothing = true; // true
var uni = new Universe();
Universe.prototype.everything = true; // true
var uni2 = new Universe();
// it's the same single instance
uni === uni2; // true
// tất cả thuộc tính prototype đều hoạt động
// không có vấn đề gì khi chúng được định nghĩa
uni.nothing && uni.everything && uni2.nothing && uni2.everything; // true
// các thuộc tính bình thường vẫn hoạt động
uni.bang; // "Big"
// thuộc tính tạo constructor chính xác
uni.constructor === Universe; // true
```

1 cách giải quyết khác cũng sẽ đóng gói hàm tạo và thực thể bên trong 1 hàm tức thời. lần đầu tiên, hàm tạo được gọi, thì nó tạo ra 1 đối tượng và cũng định hướng instance riêng tư chỉ tới

nó.từ lần gọi thứ 2 trở đi thì hàm tạo đơn giản trả về biến riêng tư này.tất cả các kiểm tra từ đoạn mã trước sẽ vẫn hoạt động tốt :

```
var Universe;
```

```
(function () {
```

```
var instance;
```

```
Universe = function Universe() {
```

```
  if (instance) {
```

```
    return instance;
```

```
  }
```

```
  instance = this;
```

```
  // all the functionality
```

```
  this.start_time = 0;
```

```
  this.bang = "Big";
```

```
};
```

```
})();
```

9.2.Factory – kiểu mẫu sản xuất đối tượng

Mục đích của kiểu mẫu factory là tạo ra các đối tượng objects, điều này thông thường được thực thi trong 1 lớp class hay 1 phương thức static – chúng có những mục đích sau :

- xử lý lặp lại các thao tác khi thiết lập các đối tượng tương tự nhau
- đưa ra 1 cách cho các khách hàng của factory là khả năng tạo ra các đối tượng objects mà không cần biết tới kiểu cụ thể (class) trong thời gian biên dịch

điểm thứ 2 quan trọng hơn trong các ngôn ngữ class static là nó tầm thường hóa việc tạo ra các thực thể của các lớp classes mà người lập trình không cần hiểu rõ các lớp này làm gì .trong JS thì phần thực thi của này trở nên rất đơn giản

các đối tượng được tạo bởi phương thức factory (hay lớp class factory) được thiết kế để kế thừa từ cùng 1 đối tượng cha parent; chúng được định rõ các lớp con subclasses để thực thi các chức năng được chuyên biệt hóa. thỉnh thoảng parent thông thường chính là lớp class mà nó có chứa phương thức factory

xét ví dụ như sau :

- 1 parent chung là hàm tạo CarMaker
- 1 phương thức static của hàm tạo CarMaker gọi là factory() – dùng để tạo ra các đối tượng car
- Các hàm tạo được chuyên biệt hóa CarMaker.Compact, CarMaker.SUV, và CarMaker.Convertible được kế thừa từ CarMaker. tất cả chúng sẽ được định nghĩa như là các thuộc tính static của cha parent vì vậy ta vẫn giữ được namespace toàn cục sạch sẽ và do vậy ta cũng biết nơi để tìm ra chúng khi ta cần

Đầu tiên nhìn vào cách thực thi sau khi hoàn thành sẽ được sử dụng như sau :

```
var corolla = CarMaker.factory('Compact');
var solstice = CarMaker.factory('Convertible');
var cherokee = CarMaker.factory('SUV');
corolla.drive(); // "Vroom, I have 4 doors"
solstice.drive(); // "Vroom, I have 2 doors"
cherokee.drive(); // "Vroom, I have 17 doors"
```

xem xét phần sau :

```
var corolla = CarMaker.factory('Compact');
```

đây hầu như chắc chắn là dạng dễ nhận ra nhất của kiểu mẫu factory. ta có 1 phương thức nhận 1 kiểu cho trước làm tham số (kiểu string) trong thời gian thực thi và sau đó nó tạo ra và trả về các đối tượng theo kiểu dữ liệu yêu cầu. như thấy ở trên thì không có hàm tạo nào được sử dụng với new hay bất cứ đối tượng object literals nào mà chỉ có 1 hàm function dùng để tạo ra các đối tượng dựa trên 1 kiểu được xác định bằng 1 chuỗi string

đây là 1 ví dụ về thực thi của kiểu mẫu factory – đoạn mã phía dưới cần thiết cho đoạn mã phía trên có thể thực thi được :

// parent constructor

```
function CarMaker() {}  
// a method of the parent  
CarMaker.prototype.drive = function () {  
  return "Vroom, I have " + this.doors + " doors";  
};
```

// the static factory method

```
CarMaker.factory = function (type) {
```

```
  var constr = type, newcar;
```

```
  // thông báo lỗi nếu hàm tạo không tồn tại
```

```
  if (typeof CarMaker[constr] !== "function") {
```

```
    throw {
```

```
      name: "Error",
```

```
      message: constr + " doesn't exist"
```

```
    };
```

```
  }
```

```
  // tại đây là khi hàm tạo đã biết chắc là tồn tại
```

```
  // ta cho nó kế thừa cha parent duy nhất 1 lần
```

```
  if (typeof CarMaker[constr].prototype.drive !== "function") {
```

```
    CarMaker[constr].prototype = new CarMaker();
```

```
  }
```

```
  // tạo ra 1 thực thể mới
```

```
  newcar = new CarMaker[constr]();
```

```
  // tùy chọn, có thể gọi 1 vài phương thức và sau đó trả về
```

```
  return newcar;
```

```
};
```

```
// định nghĩa các car makers riêng biệt
```

Kiểm tra xem đã có sự kế thừa hay chưa. nếu đã có sự kế thừa thì phương thức drive ở hàm tạo cha đã được thêm vào prototype của hàm tạo con

```
CarMaker.Compact = function () {
```

```
  this.doors = 4;
```

```
};
```

```
CarMaker.Convertible = function () {
```

```
  this.doors = 2;
```

```
};
```

```
CarMaker.SUV = function () {
```

```
  this.doors = 4;
```

```
};
```

Không có gì là quá khó hiểu về cách thực thi của kiểu mẫu factory. tất cả những gì ta cần làm là tìm kiếm hàm tạo dùng để tạo ra 1 đối tượng theo kiểu được yêu cầu. trong trường hợp này thì 1 quy ước đặt tên đơn giản được sử dụng để xác định các kiểu đối tượng tới hàm tạo dùng để tạo ra chúng. phần kế thừa chỉ là 1 ví dụ mẫu của 1 phần chúng được lặp lại của mã code do vậy nó nên được đặt vào phương thức factory thay cho việc lặp lại mỗi lần cho mỗi dạng hàm tạo

9.2.1. Đối tượng Object Factory được xây dựng sẵn

Xét tới hàm khởi tạo Object() toàn cục được xây dựng sẵn. nó cũng có cách thức xử lý như 1 factory bởi vì nó tạo ra các đối tượng khác nhau dựa vào input. nếu ta chuyển tiếp vào nó 1 kiểu số cơ bản thì nó tạo ra 1 đối tượng với hàm tạo Number(). cũng tương tự vậy khi ta chuyển tiếp vào chuỗi string và kiểu Boolean. bất kì giá trị khác không bao gồm giá trị input thì nó sẽ tạo ra 1 đối tượng thông thường

Đây là 1 vài ví dụ và mẫu kiểm tra của cách thức xử lý này. chú ý rằng Object có thể được gọi với hay không với new :

```
var o = new Object(),
```

```
n = new Object(1),
```

```
s = Object('1'),
b = Object(true);
// test
o.constructor === Object; // true
n.constructor === Number; // true
s.constructor === String; // true
b.constructor === Boolean; // true
```

thực tế thì Object() cũng là 1 factory nhưng ít được sử dụng trong thực hành

9.3.Iterator – kiểu mẫu biến lặp

Mẫu Iterator cung cấp khả năng truy cập và duyệt các thành phần của một tập hợp không cần quan tâm đến cách thức biểu diễn bên trong

ở kiểu mẫu biến lặp iterator pattern thì ta có 1 đối tượng chứa 1 vài kiểu dữ liệu kết tụ. kiểu dữ liệu này có thể được lưu trữ bên trong 1 cấu trúc dữ liệu phức tạp và ta muốn cung cấp 1 cách truy cập dễ dàng tới từng phần tử của cấu trúc dữ liệu đó. những người sử dụng đối tượng object loại này không cần phải biết cách mà ta cấu trúc hóa dữ liệu như thế nào, tất cả những gì mà họ cần là làm việc với các phần tử riêng lẻ

trong kiểu mẫu biến lặp iterator pattern, thì các đối tượng của ta cần cung cấp 1 phương thức next() method. việc gọi next() trong dãy tuần tự phải trả về phần tử nối tiếp phía sau và nó hiểu “tiếp theo”, ở đây là tiếp theo trong cấu trúc dữ liệu cụ thể nào đó

giả sử rằng đối tượng của ta có tên là agg, ta có thể truy cập từng phần tử dữ liệu bằng cách đơn giản gọi phương thức **next()** trong 1 vòng lặp như sau :

```
var element;
while (element = agg.next()) {
// do something with the element ...
console.log(element);
}
```

Trong kiểu mẫu biến lặp này, đối tượng kết tụ thông thường cũng cung cấp 1 sự tiện lợi là có phương thức `Next()`, vì vậy những người sử dụng của đối tượng có thể xác định xem chúng được chạy tới cuối dữ liệu của ta hay không. theo cách khác để truy cập tới tất cả các phần tử 1 cách liên tục thì ta sử dụng **hasNext()** và nó giống như sau :

```
while (agg.hasNext()) {  
  // do something with the next element...  
  console.log(agg.next());  
}
```

Giờ ta sử dụng các ví dụ mẫu trên và xem cách mà chúng thực thi giống như là
Đối tượng kết tụ

Khi thực thi 1 kiểu mẫu biến lặp `iterator pattern` thì có nghĩa là ta lưu trữ 1 cách riêng tư dữ liệu và 1 con trỏ (index) chỉ tới phần tử tiếp theo phía sau. để minh họa, ta giả định dữ liệu chỉ là 1 mảng array thông thường và điều kiện logic đặc biệt cho việc lấy về phần tử kế tiếp phía sau, sẽ được trả về mọi phần tử array khác :

```
var agg = (function () {  
  var index = 0,  
  data = [1, 2, 3, 4, 5],  
  length = data.length;
```

```
  return {
```

```
    next: function () {
```

```
      var element;
```

```
      if (!this.hasNext()) {
```

```
        return null;
```

```
      }
```

```
      element = data[index];
```

```
      index = index + 2;
```

```
      return element;
```

```
    },
```

Sử dụng hàm tức thời và hàm tức thời này trả về 1 đối tượng

{...} đối tượng được trả về này có 2 phương thức cần có trong kiểu mẫu biến lặp là

next và **hasNext**

Phương thức **next** dùng để trả về phần tử tiếp theo sau phần tử hiện tại 2 vị trí. (chỉ trả về khi phần tử hiện tại có phần tử tiếp theo bằng cách kiểm tra điều kiện thông qua phương thức

hasNext

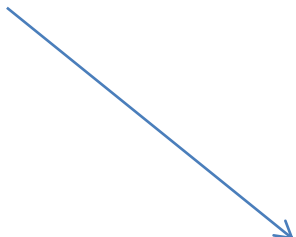
hasNext: function () {

return index < length;

}

};

})();



Phương thức **hasNext** trả về giá trị true hay false. bằng cách so sánh giá trị index với độ dài length của cấu trúc dữ liệu

Để cung cấp cách truy cập dễ dàng hơn và khả năng lặp 1 vài lần trên dữ liệu thì đối tượng của ta có thể cung cấp bổ sung thêm các phương thức tiện lợi khác như :

- **rewind()** dùng để reset lại con trỏ trở về vị trí ban đầu
- **current()** trả về phần tử hiện thời bởi vì ta không thể thực hiện this với next() mà không phải phức tạp hóa con trỏ

Thực thi các phương thức này sẽ không có gì khó khăn :

```
var agg = (function () {
```

```
// [snip...]
```

```
return {
```

```
// [snip...]
```

rewind: function () {

index = **0**;


```
},  
  
current: function () {  
    return data[index];  
}  
};  
}());
```

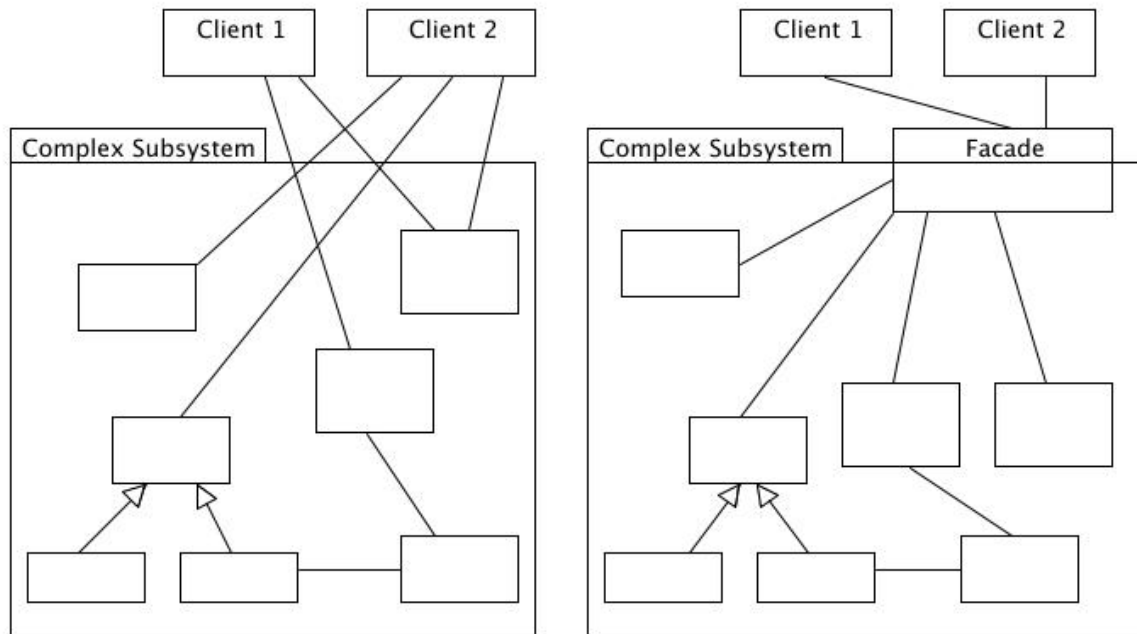
Giờ ta tiến hành lặp :

```
// this loop logs 1, then 3, then 5  
while (agg.hasNext()) {  
    console.log(agg.next());  
}  
// go back  
agg.rewind();  
console.log(agg.current()); // 1
```

kết quả ghi trên màn hình console là 1, 3, 5 (từ trong vòng lặp) và cuối cùng là 1

9.4.Façade

Khi cấu trúc hóa một hệ thống thành các hệ thống con sẽ giúp làm giảm độ phức tạp của hệ thống. Một mục tiêu thiết kế thông thường là tối thiểu hóa sự giao tiếp và phụ thuộc giữa các hệ thống con. Một cách để đạt được mục tiêu này là đưa ra đối tượng facade, đối tượng này cung cấp một giao diện đơn giản để dễ dàng tổng quát hóa cho một hệ thống con hơn.



Mẫu Facade cung cấp một giao diện thống nhất cho một tập các giao diện trong một hệ thống con. Facade định nghĩa một giao diện ở mức cao hơn, giao diện này làm cho hệ thống con được sử dụng dễ dàng hơn.

Facade là 1 kiểu mẫu đơn giản và nó cung cấp duy nhất chỉ 1 giao diện chung interface tới 1 đối tượng object. đây là 1 cách thiết kế trong thực hành tốt và nó giữ cho các phương thức của ta ngắn gọn và không quá mất công sức để hiệu chỉnh. theo sau kiểu thực hành này thì ta sẽ kết thúc với lượng lớn các phương thức hơn là ta sử dụng phương thức uber methods với 1 lượng lớn tham số. thỉnh thoảng 2 hay nhiều hơn các phương thức có thể thường được gọi cùng với nhau. trong trường hợp như vậy thì ta nên tạo ra 1 phương thức khác mà đóng gói các lời gọi phương thức được lặp lại.

Ví dụ khi hiệu chỉnh các sự kiện events thì ta có các phương thức sau :

- **stopPropagation()** bẫy sự kiện và không để nó sủi bọt lên các nút cha
- **preventDefault()** không cho trình duyệt thực hiện các thao tác mặc định

đây là 2 phương thức tách biệt được sử dụng với các mục đích khác nhau và chúng nên được giữ tách biệt nhưng tại cùng 1 thời điểm thì chúng thường được gọi cùng với nhau. vì vậy để thay thế sự dư thừa tạo ra 2 lời gọi phương thức thì ta có thể tạo ra 1 phương thức facade mà gọi tới cả 2 phương thức đó như sau:

```
var myevent = {
```

```
// ...  
stop: function (e) {  
  e.preventDefault();  
  e.stopPropagation();  
}  
  
// ...  
};
```

Kiểu mẫu façade pattern cũng phù hợp với các kịch bản trình duyệt mà với các điểm khác nhau giữa các trình duyệt có thể bị cho ẩn đi phía sau 1 façade. tiếp đoạn ví dụ ở trên thì ta có thể thêm vào mã code dùng để hiệu chỉnh những điểm khác biệt trong hàm event API của IE như sau :

```
var myevent = {  
  // ...  
  stop: function (e) {  
    // others  
    if (typeof e.preventDefault === "function") {  
      e.preventDefault();  
    }  
    if (typeof e.stopPropagation === "function") {  
      e.stopPropagation();  
    }  
    // IE  
    if (typeof e.returnValue === "boolean") {  
      e.returnValue = false;  
    }  
    if (typeof e.cancelBubble === "boolean") {  
      e.cancelBubble = true;  
    }  
  }  
  // ...  
};
```

};

Kiểu mẫu façade pattern cũng rất hữu dụng với việc thiết kế lại và sản xuất lại các kết quả đã đạt được. khi ta muốn thay thế 1 đối tượng với 1 thực thi khác thì ta phải thực hiện mất 1 thời gian trong khi vào cùng thời điểm đoạn mã mới đang được viết để sử dụng đối tượng này. ta có thể bắt đầu với việc nghĩ về các hàm API của đối tượng mới và sau đó thực hiện tạo ra 1 façade ở phía trước đối tượng cũ để cho phép các API mới. theo cách này khi ta thay thế toàn diện đối tượng cũ thì ta sẽ có ít hơn các mã client code cần được hiệu chỉnh lại bởi vì bất cứ client code nào gần đây cũng sử dụng hàm API mới

9.5.Proxy

Trong kiểu mẫu proxy design pattern thì 1 đối tượng được xem như là 1 kênh giao diện chung interface của đối tượng khác. nó khác với kiểu mẫu façade pattern. proxy nằm ở giữa client của 1 đối tượng và chính đối tượng đó và nó bảo vệ truy cập tới đối tượng đó

Kiểu mẫu này rất có thể như 1 loại chi phí phụ nhưng nó rất hiệu quả cho mục đích của nâng cao hiệu năng xử lý. Proxy phục vụ như 1 người bảo vệ của đối tượng (cũng có thể gọi là “real subject”) và **có gắng để chủ đề thực real subject này làm càng ít có thể càng tốt**

Ví dụ sử dụng giống như vậy ta có thể gọi là lazy initialization. để dễ hình dung thì việc khởi tạo chủ đề thực real subject có chi phí khá đắt đỏ và điều này xảy ra khi client khởi tạo nó nhưng không bao giờ thực sự sử dụng nó. trong trường hợp như thế thì proxy có thể giúp bằng cách nó như là 1 kênh giao diện chung interface của chủ đề thực real subject. Proxy nhận các yêu cầu khởi tạo nhưng không bao giờ thông qua (chuyển tiếp) nó cho tới khi nó rõ ràng mà real subject thực sự được sử dụng

Hình 7-2 là biểu đồ minh họa với client tạo ra 1 yêu cầu khởi tạo và proxy trả lời rằng tất cả đều tốt nhưng không thực sự chuyển thông điệp cho tới khi rõ ràng là client cần 1 vài công việc được thực hiện bởi subject. sau đó chỉ thực hiện thông qua proxy để chuyển tiếp cả 2 thông điệp cùng nhau

Client gửi yêu cầu khởi tạo init tới proxy, sau đó proxy trả lời là tốt ok và không chuyển tiếp bất cứ thứ gì tới real subject. vì yêu cầu init này không cần thiết để real subject thực sự làm việc gì cả

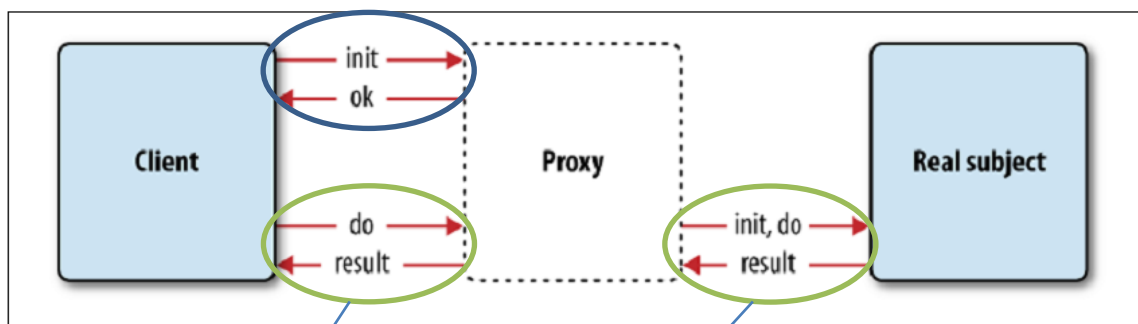


Figure 7-2. Relationship between client and real subject when going through a proxy

Client gửi yêu cầu do (cần làm 1 công việc gì đó) tới proxy, sau đó proxy nhận thấy yêu cầu này là cần thiết và nó gửi yêu cầu init, do tới real subject. và real subject nhận được yêu cầu và xử lý gửi lại kết quả result tới proxy và proxy chuyển tiếp kết quả result này trở lại client

9.5.1.Một ví dụ mẫu

Kiểu mẫu proxy pattern là rất hữu dụng khi real subject làm 1 vài điều gì đó có chi phí đắt đỏ.trong các ứng dụng web thì 1 trong các thực thi có chi phí đắt đỏ nhất mà ta có thể phải thực hiện là 1 yêu cầu network vì vậy ta nên kết hợp với các yêu cầu HTTP càng nhiều có thể càng tốt.ta sẽ xem xét 1 ví dụ để hình dung rõ kiểu mẫu proxy pattern thực hiện

9.5.1.1.A video expando

Đây là 1 ứng dụng nhỏ dùng để chơi video từ 1 ca sĩ đã được chọn (hình 7-3), ta có thể xem 1 ví dụ trực tiếp ngay tại <http://www.jspatterns.com/book/7/proxy.html>.

Ta có 1 danh sách video trên 1 trang page,khi người dùng ấn vào 1 tiêu đề của video thì vùng phía dưới tiêu đề được mở rộng ra và hiện thị thông tin chi tiết hơn về video và cũng cho phép video được bật.thông tin chi tiết của video và URL của video không phải là 1 phần của trang page và chúng cần được lấy về bởi việc tạo ra 1 lời gọi web service call. web service call nhận nhiều video IDs vì vậy ta có thể tăng tốc ứng dụng bằng cách tạo ra nhiều yêu cầu HTTP requests hơn vào bất cứ khi nào có thể và nhận về dữ liệu cho 1 vài videos vào cùng 1 thời điểm

ứng dụng cho phép 1 vài (hay tất cả) videos được mở rộng ra cùng 1 thời điểm vì vậy đây đúng là 1 cơ hội hoàn hảo để kết hợp với các yêu cầu web service requests.

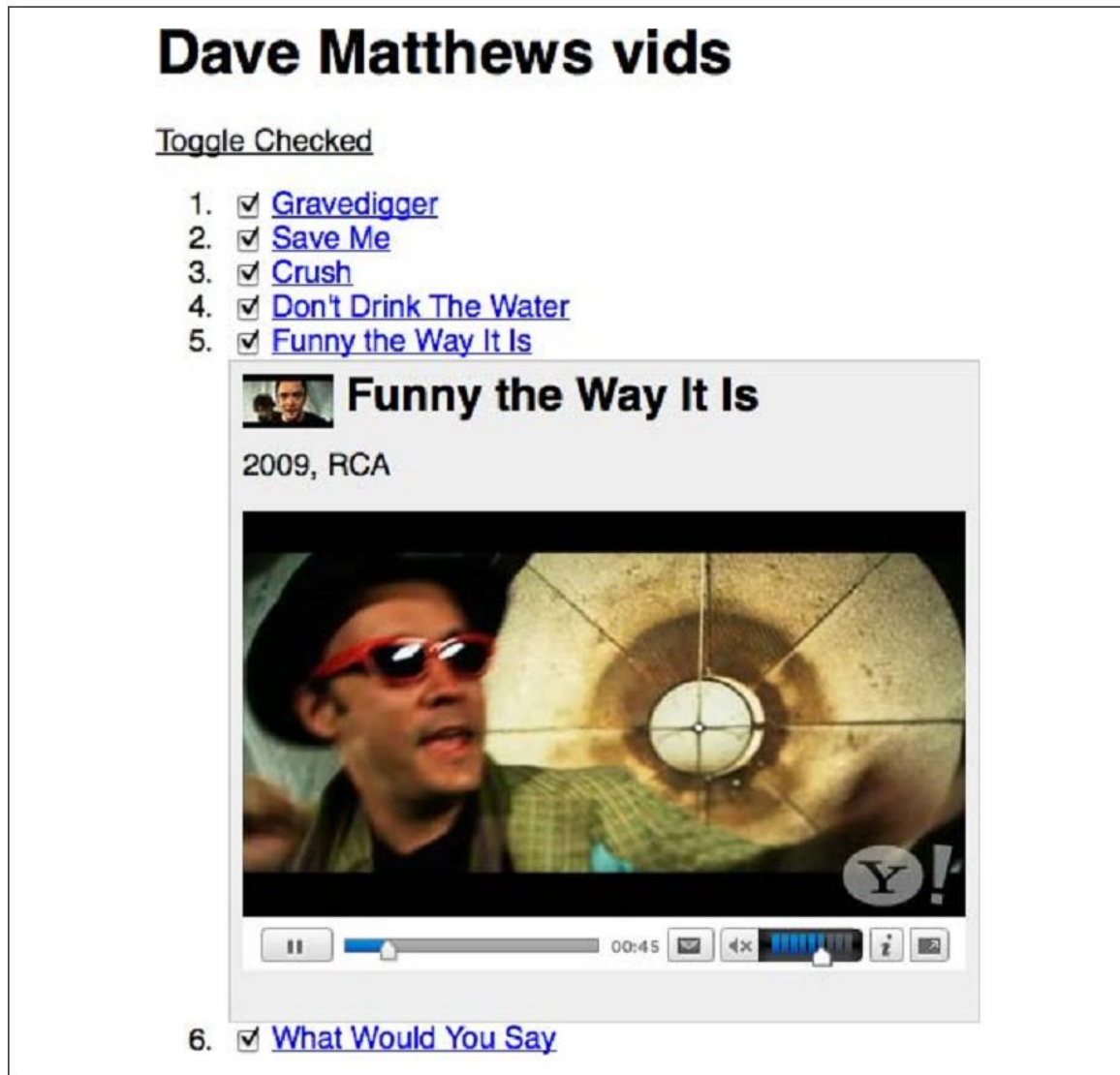


Figure 7-3. Video expando in action

9.5.1.2. Không với 1 proxy

Hành động chính trong ứng dụng là 2 đối tượng :

- videos phụ trách về vùng thông tin được mở ra hay co lại (phương thức videos.getInfo()) và chơi videos (phương thức videos.getPlayer())
- http phụ trách về việc giao tiếp với server thông qua phương thức http.makeRequest()

khi không có proxy thì videos.getInfo() sẽ gọi tới http.makeRequest() một lần với mọi video. khi ta thêm vào 1 proxy thì nó trở thành 1 thành phần mới được gọi là proxy và nó sẽ nằm ở giữa videos và http và ủy quyền lời gọi tới makeRequest() và kết hợp chúng khi có thể

ta xem xét mã code với không proxy đầu tiên và sau đó thêm proxy vào để cải tiến độ phản hồi của ứng dụng

9.5.1.3.HTML

mã HTML code chỉ là 1 danh sách các links :

```
<p><span id="toggle-all">Toggle Checked</span></p>
<ol id="vids">
<li><input type="checkbox" checked><a
href="http://new.music.yahoo.com/videos/--2158073">Gravedigger</a></li>
<li><input type="checkbox" checked><a
href="http://new.music.yahoo.com/videos/--4472739">Save Me</a></li>
<li><input type="checkbox" checked><a
href="http://new.music.yahoo.com/videos/--45286339">Crush</a></li>
<li><input type="checkbox" checked><a
href="http://new.music.yahoo.com/videos/--2144530">Don't Drink The Water</a></li>
<li><input type="checkbox" checked><a
href="http://new.music.yahoo.com/videos/--217241800">Funny the Way It Is</a></li>
<li><input type="checkbox" checked><a
href="http://new.music.yahoo.com/videos/--2144532">What Would You Say</a></li>
</ol>
```

9.5.1.4.Các hàm điều chỉnh sự kiện Event handlers

Giờ ta xét tới các hàm điều chỉnh sự kiện Event handlers.đầu tiên ta định nghĩa 1 quy ước ngắn gọn sau :

```
var $ = function (id) {
return document.getElementById(id);
};
```

Bằng cách sử dụng sự ủy thác sự kiện event delegation thì ta hiệu chỉnh tất cả các sự kiện clicks xảy ra trong danh sách có thứ tự id="vids" với 1 hàm function duy nhất :

```
$('vids').onclick = function (e) {
```



```
var src, id;
e = e || window.event;
src = e.target || e.srcElement;
if (src.nodeName !== "A") {
return;
}
if (typeof e.preventDefault === "function") {
e.preventDefault();
}
e.returnValue = false;
id = src.href.split('--')[1];
if (src.className === "play") {
src.parentNode.innerHTML = videos.getPlayer(id);
return;
}
src.parentNode.id = "v" + id;
videos.getInfo(id);
};
```

Trong hàm điều chỉnh bất toàn bộ sự kiện click thì ta sẽ thấy thích thú với 2 cú clicks : 1 để mở hay co lại phần thông tin (gọi getInfo()) và 1 để chơi video (khi mục tiêu có class tên là play) – khi chơi video thì có nghĩa là phần thông tin sẽ được mở ra và ta có thể gọi getPlayer(). các IDs của videos được trích ra từ thuộc tính hrefs

Hàm điều chỉnh click khác tác động trở lại để ẩn click vào công tắc bật tắt của toàn bộ vùng thông tin, về bản chất nó chỉ là lời gọi tới phương thức getInfo() 1 lần nữa nhưng trong 1 vòng lặp :

```
$('#toggle-all').onclick = function (e) {
var hrefs,
i,
max,
id;
hrefs = $('#vids').getElementsByTagName('a');
for (i = 0, max = hrefs.length; i < max; i += 1) {
```

```
// skip play links
if (hrefs[i].className === "play") {
  continue;
}
// skip unchecked
if (!hrefs[i].parentNode.firstChild.checked) {
  continue;
}
id = hrefs[i].href.split('--')[1];
hrefs[i].parentNode.id = "v" + id;
videos.getInfo(id);
}
};
```

9.5.1.5. Đối tượng *videos object*

Đối tượng *videos object* có 3 phương thức :

- *getPlayer()* trả về HTML cần thiết cho việc chơi Flash video
- *updateList()* hàm callback dùng để nhận tất cả dữ liệu từ web service và tạo ra mã HTML code được dùng trong phần thông tin đã được mở ra.
- *getInfo()* là phương thức dùng để bật tắt sự hiện thị của vùng thông tin và cũng tạo ra lời gọi tới http được chuyển tiếp vào *updateList()*

đây là đoạn mã của đối tượng :

```
var videos = {
  getPlayer: function (id) { ... },
  updateList: function (data) { ... },
  getInfo: function (id) {
    var info = $('info' + id);
    if (!info) {
      http.makeRequest([id], "videos.updateList");
    }
    return;
  }
};
```

```
}  
if (info.style.display === "none") {  
    info.style.display = "";  
} else {  
    info.style.display = 'none';  
}  
}  
};
```

9.5.1.5.http object

http object chỉ có duy nhất 1 phương thức – dùng để tạo ra yêu cầu JSON tới Yahoo!’s

YQL web service:

```
var http = {  
    makeRequest: function (ids, callback) {  
        var url = 'http://query.yahooapis.com/v1/public/yql?q=',  
            sql = 'select * from music.video.id where ids IN ("%ID%")',  
            format = "format=json",  
            handler = "callback=" + callback,  
            script = document.createElement('script');  
            sql = sql.replace('%ID%', ids.join(", "));  
            sql = encodeURIComponent(sql);  
            url += sql + '&' + format + '&' + handler;  
            script.src = url;  
            document.body.appendChild(script);  
        }  
};
```

Khi 6 videos được bật tắt thì 6 yêu cầu dành riêng sẽ được gửi tới web service như sau :

```
select * from music.video.id where ids IN ("2158073")
```

9.5.1.6. Thêm vào proxy

mã code ở trên hoạt động tương đối tốt nhưng ta có thể khiến nó tốt hơn. đối tượng proxy object được thêm vào kịch bản này và nó chịu trách nhiệm kết nối http với video. nó có gắng kết hợp các yêu cầu bằng cách sử dụng 1 điều kiện logic đơn giản : 1 bộ đệm 50ms buffer. đối tượng videos object không cần gọi HTTP service 1 cách trực tiếp nhưng gọi thông qua proxy. proxy sau đó đợi trước khi nó gửi yêu cầu đi. nếu các lời gọi khác từ videos tới trong khoảng thời gian 50ms thì chúng sẽ được hợp nhất là 1. 1 sự trì hoãn 50ms là khoảng thời gian mà người dùng không thể cảm nhận được nhưng có thể giúp kết hợp các yêu cầu và tăng tốc trải nghiệm khi ấn vào “toggle” và mở rộng hơn 1 video trong cùng 1 thời điểm. nó cũng giảm thiểu tải trọng của server 1 cách đáng kể khi mà web server khi này điều khiển 1 số lượng yêu cầu ít hơn

chuỗi truy vấn kết hợp YQL query cho 2 video giống như sau :

```
select * from music.video.id where ids IN ("2158073", "123456")
```

trong phiên bản đã được chỉnh lại của mã code thì chỉ có duy nhất 1 sự thay đổi là videos.getInfo() giờ gọi tới proxy.makeRequest() thay thế cho gọi tới http.makeRequest() như sau :

```
proxy.makeRequest(id, videos.updateList, videos);
```

proxy thiết lập 1 hàng đợi để thu thập các IDs của videos được nhận về trong 50ms và sau đó làm đầy hàng đợi bằng cách gọi http và cung cấp 1 hàm callback bởi vì hàm videos.updateList() callback có thể hiệu chỉnh duy nhất 1 bảng ghi dữ liệu đơn

đây là mã code cho proxy:

```
var proxy = {  
  ids: [],  
  delay: 50,  
  timeout: null,  
  callback: null,  
  context: null,  
  makeRequest: function (id, callback, context) {  
    // add to the queue  
    this.ids.push(id);
```

```
this.callback = callback;
this.context = context;
// set up timeout
if (!this.timeout) {
  this.timeout = setTimeout(function () {
    proxy.flush();
  }, this.delay);
},
flush: function () {
  http.makeRequest(this.ids, "proxy.handler");
// clear timeout and queue
  this.timeout = null;
  this.ids = [];
},
handler: function (data) {
  var i, max;
  // single video
  if (parseInt(data.query.count, 10) === 1) {
    proxy.callback.call(proxy.context, data.query.results.Video);
    return;
  }
  // multiple videos
  for (i = 0, max = data.query.results.Video.length; i < max; i += 1) {
    proxy.callback.call(proxy.context, data.query.results.Video[i]);
  }
};
```

Proxy cung cấp khả năng kết hợp nhiều yêu cầu web service requests vào thành 1 với chỉ 1 thay đổi đơn giản trong mã code gốc

Hình 7-4 và 7-5 minh họa biểu đồ của việc tạo ra 3 roundtrips tới server (với không proxy) với 1 roundtrip khi sử dụng 1 proxy

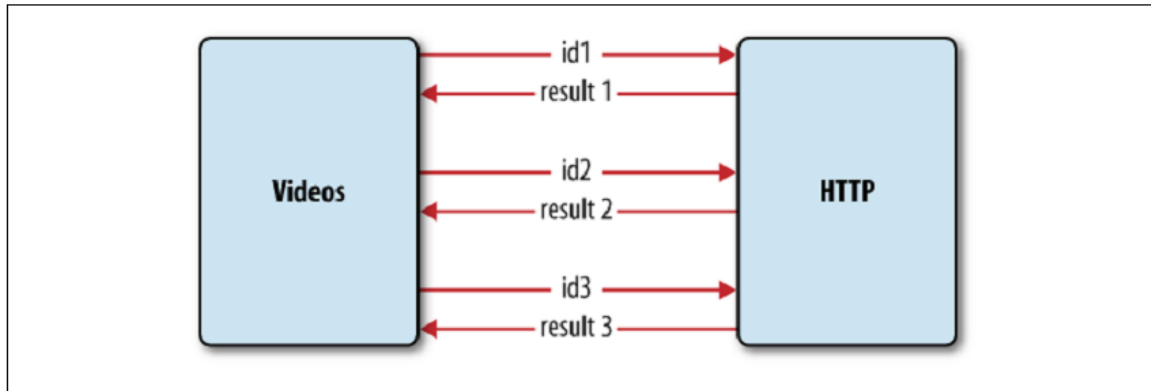


Figure 7-4. Three roundtrips to the server

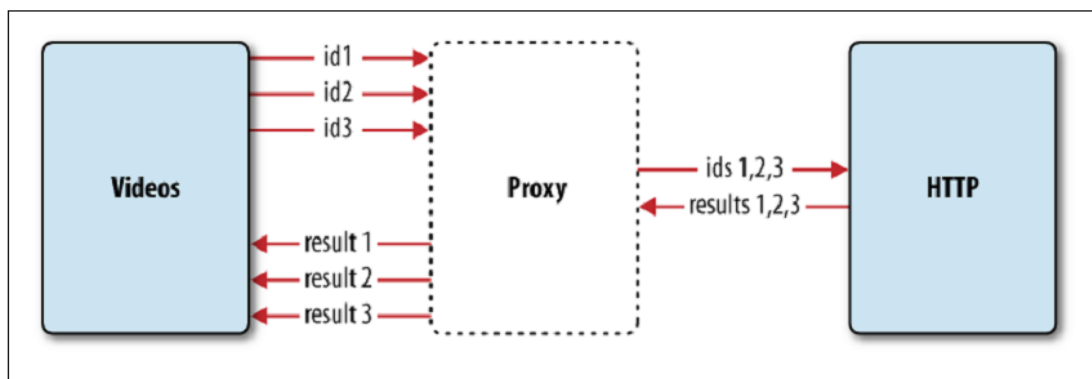


Figure 7-5. Using a proxy to combine and reduce the number of roundtrips to the server

9.5.1.7. Sử dụng proxy làm cache

Trong ví dụ này thì đối tượng client object (videos) không đủ thông minh để yêu cầu thông tin video từ cùng 1 video.proxy có thể tiến xa hơn trong việc bảo vệ real subject http bằng cách caching kết quả từ các yêu cầu trước đó trong 1 thuộc tính cache (như hình 7-6).sau đó nếu đối tượng videos object được yêu cầu thông tin về cùng 1 video ID trong lần thứ 2 thì proxy có thể lấy nó ra khỏi cache và tiết kiệm network roundtrip.

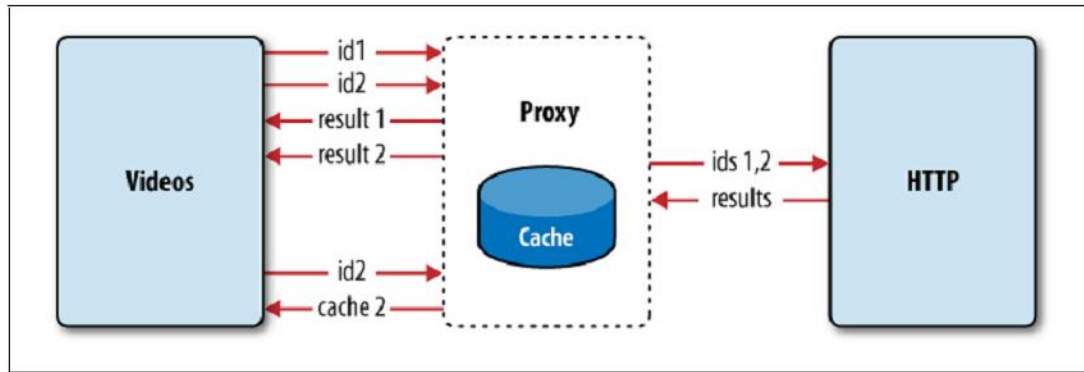


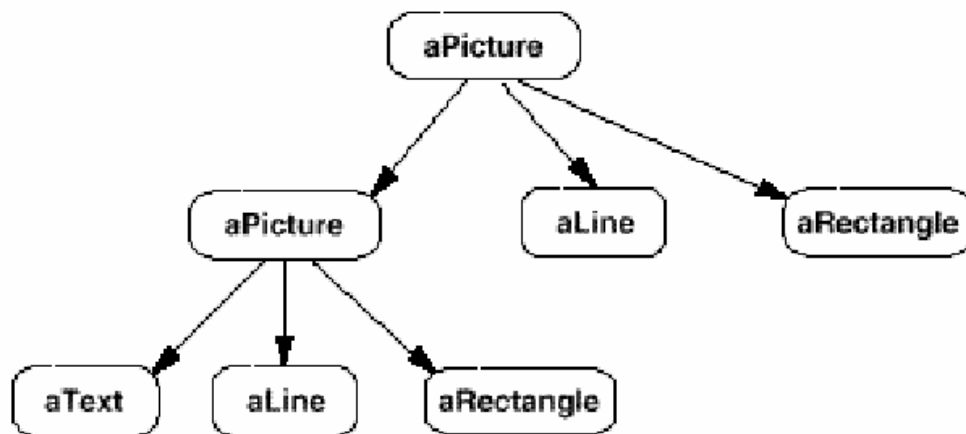
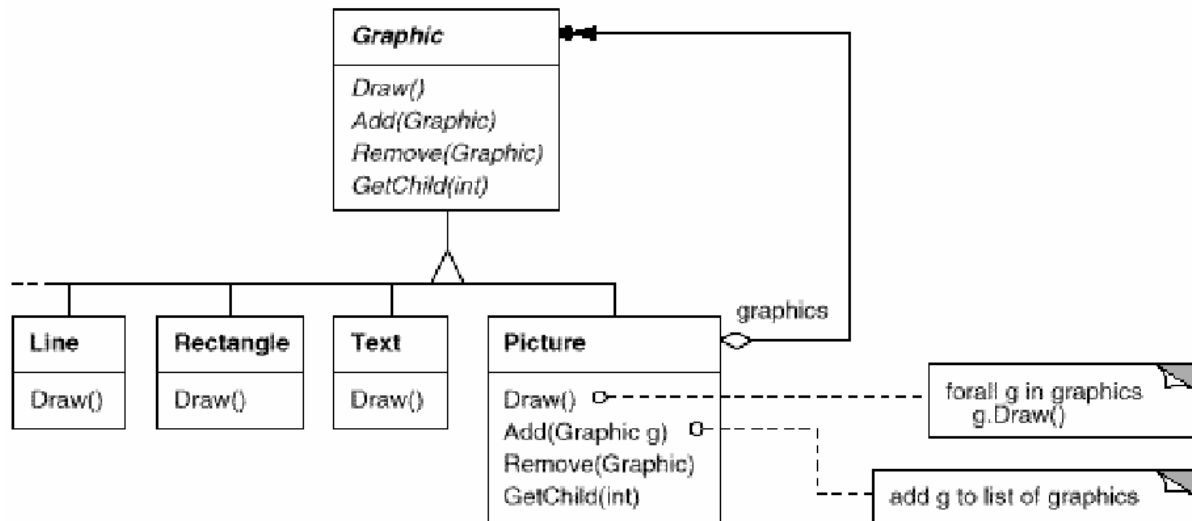
Figure 7-6. The proxy cache

9.6.Composite

9.6.1.Vấn đề đặt ra

Các ứng dụng đồ họa như bộ soạn thảo hình vẽ và các hệ thống lưu giữ biểu đồ cho phép người sử dụng xây dựng lên các lược đồ phức tạp khác xa với các thành phần cơ bản, đơn giản. Người sử dụng có thể nhóm một số các thành phần để tạo thành các thành phần khác lớn hơn, và các thành phần lớn hơn này lại có thể được nhóm lại để tạo thành các thành phần lớn hơn nữa. Một cài đặt đơn giản có thể xác định các lớp cho các thành phần đồ họa cơ bản như Text và Line, cộng với các lớp khác cho phép hoạt động như các khuôn chứa các thành phần cơ bản đó. Nhưng có một vấn đề với cách tiếp cận này, đó là, mã sử dụng các lớp đó phải tác động lên các đối tượng nguyên thủy (cơ bản) và các đối tượng bao hàm các thành phần nguyên thủy ấy là khác nhau ngay cả khi hầu hết thời gian người sử dụng tác động lên chúng là như nhau. Có sự phân biệt các đối tượng này làm cho ứng dụng trở nên phức tạp hơn. Composite pattern đề cập đến việc sử dụng các thành phần đệ quy để làm cho các client không tạo ra sự phân biệt trên.

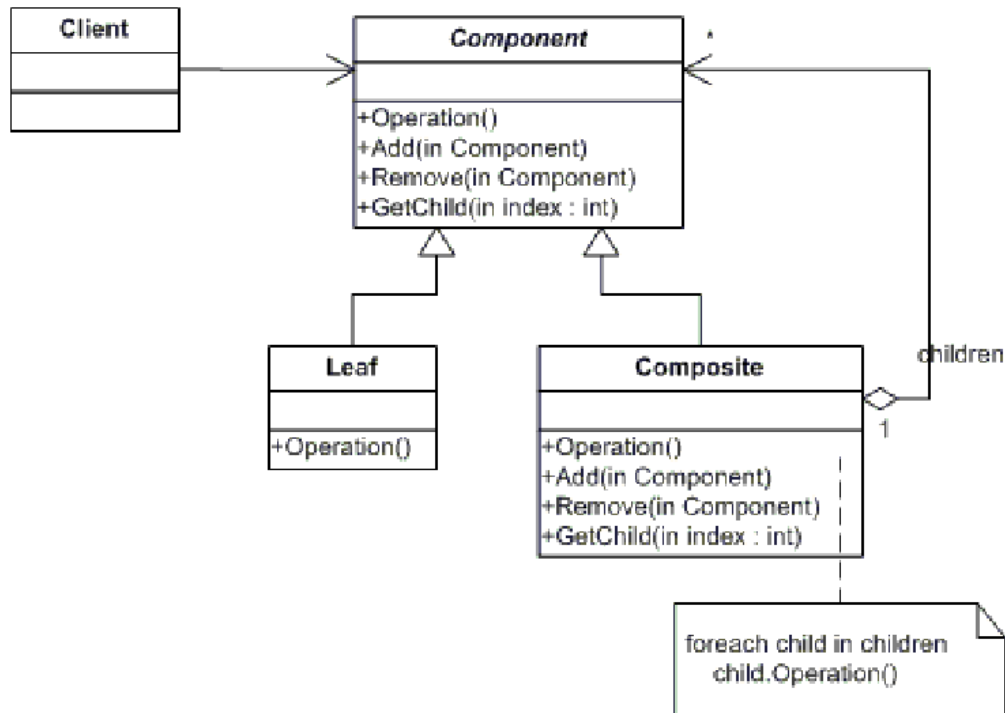
Giải pháp của Composite pattern là một lớp trừu tượng biểu diễn cả các thành phần cơ bản và các lớp chứa chúng. Lớp này cũng xác định các thao tác truy nhập và quản lý các con của nó.



Composite được áp dụng trong các trường hợp sau :

- Ta muốn biểu diễn hệ thống phân lớp bộ phận – toàn bộ của các đối tượng
- Ta muốn các client có khả năng bỏ qua sự khác nhau giữa các thành phần của các đối tượng và các đối tượng riêng lẻ. Các client sẽ “đối xử” với các đối tượng trong cấu trúc composite một cách thống nhất.

Composite là mẫu thiết kế dùng để tạo ra các đối tượng trong các cấu trúc cây để biểu diễn hệ thống phân lớp: bộ phận – toàn bộ. Composite cho phép các client tác động đến từng đối tượng và các thành phần của đối tượng một cách thống nhất.



Component (DrawingElement)

- Khai báo giao diện cho các đối tượng trong một khối kết tập.
- Cài đặt các phương thức mặc định cho giao diện chung của các lớp một cách phù hợp.
- Khai báo một giao diện cho việc truy cập và quản lý các thành phần con của nó
- Định nghĩa một giao diện cho việc truy cập các đối tượng cha của các thành phần theo một cấu trúc đệ quy và cài đặt nó một cách phù hợp nhất.

Leaf (PrimitiveElement)

- Đại diện cho một đối tượng nút lá trong khối kết tập. Một lá là một nút không có con.
- Định nghĩa hành vi cho các đối tượng nguyên thủy trong khối kết tập

Composite (CompositeElement)

- Định nghĩa hành vi cho các thành phần mà có con.
- Lưu trữ các thành phần con
- Cài đặt toán tử quan hệ giữa các con trong giao diện của thành phần

Client (CompositeApp)

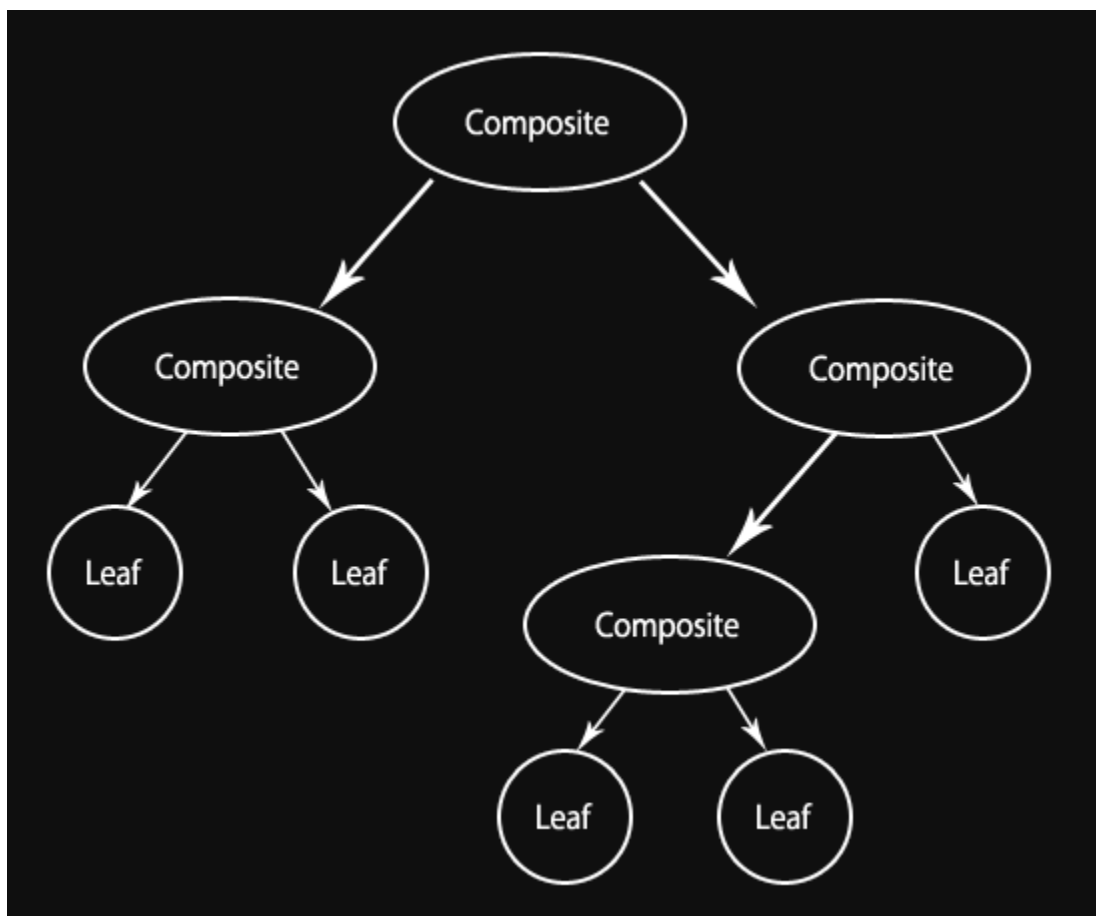
- Vận dụng các đối tượng trong khối kết tập thông qua giao diện của thành phần

Có 2 ưu điểm chính mà kiểu mẫu Composite pattern cung cấp :

- Ta có thể thực thi xử lý toàn bộ tập hợp của các đối tượng theo cùng cách mà ta thực thi xử lý bất kì 1 đối tượng riêng biệt nào trong tập hợp đối tượng đó, các hàm functions thực thi trên kết tập composite thì được chuyển tiếp xuống tới từng con của nó. ở các tập hợp lớn thì điều này rất có lợi (mặc dù cũng có thể là sai lầm bởi vì ta rất có thể không hiểu rõ tập hợp này lớn như thế nào và do vậy không hiểu tồn tại về hiệu năng xử lý là bao nhiêu)
- Nó tổ chức các đối tượng vào trong 1 cấu trúc hình cây và mỗi đối tượng kết tụ composite object chứa đựng 1 phương thức để lấy về các con của nó do đó ta có thể ẩn đi cách thực thi và thiết lập lại các con theo bất cứ cách nào mà ta muốn

9.6.2. Cấu trúc của kiểu mẫu composite pattern

Trong cây phân cấp của kiểu mẫu composite pattern thì có 2 kiểu đối tượng : lá leaf và kết tụ composite. hình dưới chỉ ra 1 mẫu ví dụ về cấu trúc của composite. nó là đệ quy. điểm khác biệt giữa các đối tượng kết tụ composite objects và các lá leaves là các lá leaves thì chúng không có bất kì con nào trong khi về bản chất thì composite là đối tượng này có chứa con



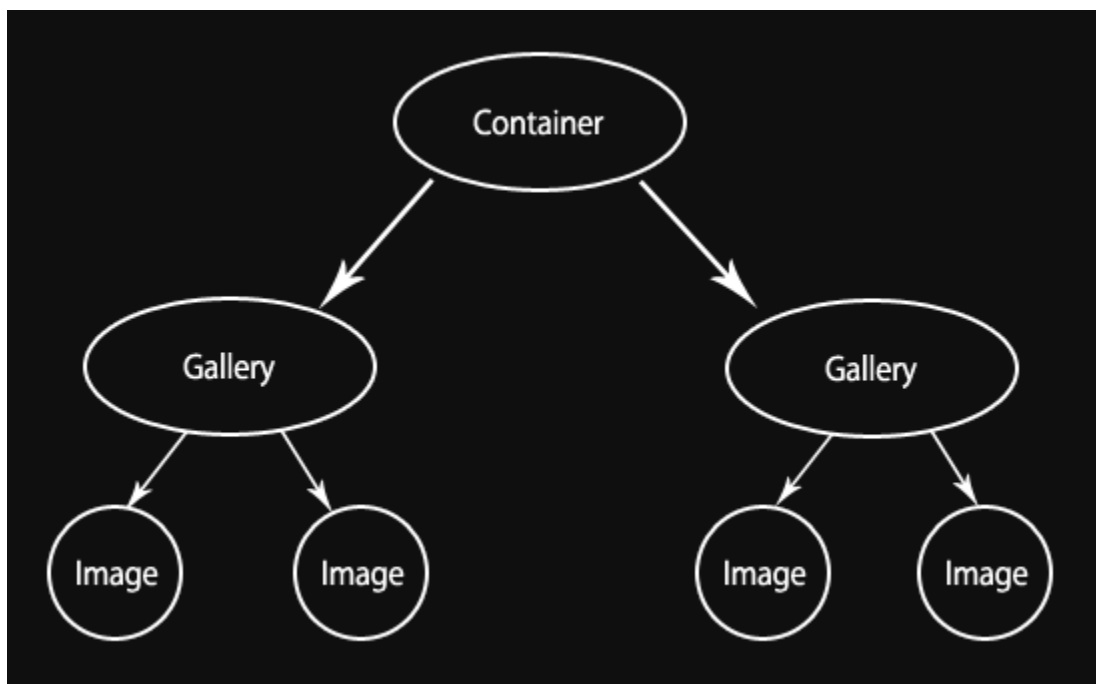
9.6.3.Các mẫu ví dụ về kiểu mẫu Composite Pattern

Đây là 1 số các mẫu thông dụng của kiểu mẫu Composite Pattern.nếu ta đã từng sử dụng PC thì ta sẽ hầu như sẽ đã từng sử dụng thực thi rất phổ biến của kiểu mẫu này: đó là cấu trúc file.xem xét mọi disk/drive và thư mục folder thì chúng đều là các composite object và mọi file đều là lá leaf.khi ta có gắng xóa 1 thư mục thì nó không chỉ xóa thư mục đó đi mà còn mọi thư mục khác và file được chứa trong thư mục đó cũng bị xóa đi.

1 mẫu ví dụ khác là 1 dạng đặc biệt của composite là cây nhị phân.nếu ta không biết nó là gì thì ta có thể xem nó trên http://en.wikipedia.org/wiki/Binary_search_tree.đây là trường hợp đặc biệt bởi vì mỗi nút node có thể chứa nhiều nhất 2 con.và cây nhị phân cũng có lá leaf và các composite. Composites biểu diễn 1 giá trị kết thúc cũng giống các lá leaves và có lá leaves này cũng có thể trở thành các composites vào bất cứ lúc nào mà ta đưa vào chúng các con.nó cũng được tối ưu hóa được chủ yếu sử dụng trong tìm kiếm thông qua các dữ liệu có khả năng sắp xếp

9.6.4.Mẫu ví dụ trong JS sử dụng kiểu mẫu Composite Pattern

Để minh họa kiểu mẫu Composite pattern bằng cách sử dụng JS thì ta sẽ không bao giờ sử dụng mẫu ví dụ ở trên mà thay thế vào đó thì ta tạo ra 1 image gallery.ví dụ này rất giống với ví dụ về cấu trúc file ở trên ngoại trừ rằng không có cách nào để ánh xạ tới nơi mà ảnh được lưu trữ hay được tổ chức trên ổ đĩa và nó thực sự chỉ hướng tới 1 số cấp nào đó.nhìn ảnh phía dưới :



Chú ý rằng trong ảnh image thì tất cả các ảnh images đều được chứa trong phạm vi composites ở mức độ “Gallery” level.

Mỗi đối tượng trong sự phân cấp cần thực thi 1 giao diện chung interface nào đó. do JS không có gì gọi là interface nên ta chỉ cần đảm bảo rằng ta thực thi các phương thức xác định nào đó. phía dưới là danh sách các phương thức và chức năng của chúng :

add	Thêm 1 nút con child node vào composite
remove	Xóa bỏ 1 nút con child node từ composite
getChild	Trả về 1 đối tượng con child object
hide	Ẩn đi composite và toàn bộ con của nó
show	Hiện ra composite và toàn bộ con của nó
getElement	Lấy về phần tử HTML của node

Đầu tiên ta sẽ đưa ra mã JS cho việc thực thi GalleryComposite bằng cách sử dụng giao diện chung interface này. chú ý tới việc sử dụng thư viện JQuery để trợ giúp 1 vài việc liên quan tới DOM :

```
var GalleryComposite = function (heading, id) {  
  
    this.children = [];  
  
    this.element = $('<div id="' + id + '" class="composite-gallery"></div>')  
        .append('<h2>' + heading + '</h2>');  
  
}
```

```
GalleryComposite.prototype = {  
  
    add: function (child) {  
  
        this.children.push(child);  
        this.element.append(child.getElement());  
  
    },  
  
}
```

```
remove: function (child) {  
    for (var node, i = 0; node = this.getChild(i); i++) {  
        if (node == child) {  
            this.children.splice(i, 1);  
            this.element.detach(child.getElement());  
            return true;  
        }  
  
        if (node.remove(child)) {  
            return true;  
        }  
    }  
  
    return false;  
},
```

```
getChild: function (i) {  
    return this.children[i];  
},
```

```
hide: function () {  
    for (var node, i = 0; node = this.getChild(i); i++) {  
        node.hide();  
    }  
  
    this.element.hide(0);  
},
```

```
show: function () {  
    for (var node, i = 0; node = this.getChild(i); i++) {  
        node.show();  
    }  
}
```

```
this.element.show(0);  
},
```

```
getElement: function () {  
    return this.element;  
}  
}
```

Tiếp đến, ta sẽ thực thi xử lý các images bằng cách sử dụng GalleryImage:

```
var GalleryImage = function (src, id) {  
  
    this.children = [];  
  
    this.element = $('<img />')  
        .attr('id', id)  
        .attr('src', src);  
}
```

```
GalleryImage.prototype = {  
    // Due to this being a leaf, it doesn't use these methods,  
    // but must implement them to count as implementing the  
    // Composite interface  
    add: function () { },  
  
    remove: function () { },  
  
    getChild: function () { },  
  
    hide: function () {  
        this.element.hide(0);  
    },  
}
```

```
show: function () {  
    this.element.show(0);  
},
```

```
getElement: function () {  
    return this.element;  
}  
}
```

Ta để ý rằng GalleryImage không làm bất cứ thứ gì trong các phương thức add, remove và getChild.do nó là 1 lá leaf và nó không chứa bất cứ con nào vì vậy nó không làm gì trong các phương thức này.tuy nhiên ta cần phải nhúng các phương thức này để tuân theo với giao diện chung Interface mà ta đã thiết lập.sau tất cả thì các đối tượng composite objects không biết rằng nó là 1 lá leaf và rất có thể thử gọi phương thức này

Giờ các lớp classes đã được xây dựng và ta sẽ viết 1 vài mã JS để dùng các lớp classes này để tạo và hiện thị ra 1 gallery thật sự.mặc dù ví dụ này chỉ chỉ ra cách dùng 3 phương thức như trên nhưng ta có thể mở rộng mã code này để tạo ra 1 gallery linh hoạt hơn – nơi mà người dùng có thể thêm và xóa các images và các thư mục folders 1 cách dễ dàng

```
var container = new GalleryComposite("", 'allgalleries');  
var gallery1 = new GalleryComposite('Gallery 1', 'gallery1');  
var gallery2 = new GalleryComposite('Gallery 2', 'gallery2');
```

```
var image1 = new GalleryImage('image1.jpg', 'img1');
```

```
var image2 = new GalleryImage('image2.jpg', 'img2');
```

```
var image3 = new GalleryImage('image3.jpg', 'img3');
```

```
var image4 = new GalleryImage('image4.jpg', 'img4');
```

```
gallery1.add(image1);
```

```
gallery1.add(image2);
```

```
gallery2.add(image3);
```

```
gallery2.add(image4);
```

```
container.add(image1);
```

```
container.add(image2);
```

```
// Make sure to add the top container to the body,
```

```
// otherwise it'll never show up.
```

```
container.getElement().appendTo('body');
```

```
container.show();
```

9.6.5.Ưu điểm và nhược điểm của kiểu mẫu Composite Pattern

ta có thể thấy những ưu điểm đáng ngạc nhiên là luôn luôn gọi 1 hàm function ở các đối tượng thuộc top level và có các kết quả ở bất cứ hay tất cả các nút nodes trong cấu trúc composite .mã code dễ dàng được trong sử dụng hơn.các đối tượng trong composite được ghép cặp lồng bởi vì tất cả chúng đều tuân theo cùng 1 giao diện interface chung.cuối cùng composite đưa ra 1 cấu trúc đẹp dễ cho các đối tượng hơn là việc giữ tất cả chúng trong những biến riêng biệt hay trong 1 mảng array

tuy nhiên thì kiểu mẫu composite pattern có thể dễ gây nhầm lẫn.tạo ra 1 lời gọi tới 1 hàm function đơn có thể rất dễ dàng và do đó ta rất có thể không hình dung ra rõ những hậu quả bất lợi về hiệu năng xử lý nếu composite quá lớn

kiểu mẫu Composite pattern là 1 công cụ mạnh mẽ “với tính năng mạnh mẽ đồng nghĩa phải gánh vác nặng hơn”.sử dụng kiểu mẫu Composite pattern 1 cách khôn ngoan thì ta sẽ tiến đến gần hơn để trở thành 1 bậc thầy JS.

Trong JQuery thì khi ta áp dụng các phương thức tới 1 phần tử hay 1 tập các phần tử thì ta có thể thực thi xử lý cả 2 tập trong cùng 1 cách thống nhất như các selections trả về 1 đối tượng jquery object.

Ta minh họa mã code bằng cách sử dụng jQuery selector ở phía dưới. ở dưới ta có thể thêm 1 lớp class tên active vào cả 2 selections cho 1 phần tử đơn (ví dụ như phần tử với ID độc nhất) hay 1 nhóm các phần tử với cùng tên tag hay lớp class mà không phải bổ sung thêm 1 kết quả nào :

```
// Single elements
```

```
$( "#singleItem" ).addClass( "active" );
```

```
$( "#container" ).addClass( "active" );
```

```
// Collections of elements
```

```
$( "div" ).addClass( "active" );
```

```
$( ".item" ).addClass( "active" );
```

```
$( "input" ).addClass( "active" );
```

Trong JQuery thì addClass() có thể sử dụng trực tiếp các vòng lặp for để lặp thông qua 1 tập hợp các phần tử để áp dụng phương thức tới các phần tử đơn hay các nhóm phần tử. nhìn vào mã code :

```
addClass: function( value ) {  
    var classNames, i, l, elem,  
        setClass, c, cl;  
  
    if ( jQuery.isFunction( value ) ) {  
        return this.each(function( j ) {  
            jQuery( this ).addClass( value.call(this, j, this.className) );  
        });  
    }  
  
    if ( value && typeof value === "string" ) {  
        classNames = value.split( rspace );  
  
        for ( i = 0, l = this.length; i < l; i++ ) {  
            elem = this[ i ];
```

```
if ( elem.nodeType === 1 ) {
    if ( !elem.className && classNames.length === 1 ) {
        elem.className = value;

    } else {
        setClass = " " + elem.className + " ";

        for ( c = 0, cl = classNames.length; c < cl; c++ ) {
            if ( !~setClass.indexOf( " " + classNames[ c ] + " " ) ) {
                setClass += classNames[ c ] + " ";
            }
        }
        elem.className = jQuery.trim( setClass );
    }
}

return this;
}
```

9.7.Observer – người quan sát

Kiểu mẫu observer pattern được sử dụng 1 cách rộng rãi trong lập trình JS hướng client. tất cả các sự kiện events của trình duyệt (mouseover, keypress, ...) đều là các mẫu của kiểu mẫu này. 1 tên gọi khác cho kiểu mẫu là là custom events – điều này có nghĩa là các sự kiện events mà ta tạo ra theo cách tự định nghĩa thì chống lại với những sự kiện events do trình duyệt kích hoạt ra. tên gọi khác nữa là subscriber/publisher pattern

Động cơ thúc đẩy chính nằm trong kiểu mẫu này chính là đẩy mạnh sự ghép cặp không chặt. thay thế cho việc 1 đối tượng gọi 1 phương thức của đối tượng khác thì đối tượng đặt mua công việc được định nghĩa của đối tượng khác và lấy về các thông báo. người đặt mua subscriber cũng được gọi là người quan sát observer, trong khi đối tượng đang được quan sát

được gọi là publisher hay subject. Publisher thông báo (hay gọi) tất cả các subscribers khi 1 sự kiện event quan trọng xảy ra và thường chuyển tiếp 1 thông điệp trong dạng là 1 đối tượng event object

9.7.1.Mẫu ví dụ #1: việc đặt mua tạp chí

Để hiểu cách thực thi kiểu mẫu này thì ta xem xét 1 ví dụ cụ thể.ta có 1 nhà xuất bản publisher là paper – xuất bản những tờ báo mới ra hàng ngày và tạp chí ra hàng tháng.1 người đặt mua subscriber là joe sẽ được thông báo bất cứ khi nào có chuyện gì xảy ra

Đối tượng paper object cần có 1 thuộc tính subscribers – là 1 mảng array dùng để lưu trữ toàn bộ người đặt mua subscribers.thao tác của việc đặt mua là đơn thuần là thêm nó vào mảng array.khi 1 sự kiện event xảy ra thì paper lập thông qua danh sách người đặt mua subscribers và thông báo tới tất cả chúng.sự thông báo này có nghĩa là nó thực hiện gọi 1 phương thức của đối tượng đặt mua subscriber object.do vậy khi đặt mua thì người đặt mua subscriber cung cấp 1 trong các phương thức của nó tới phương thức subscribe() của paper

Paper cũng có thể cung cấp 1 phương thức unsubscribe() – dùng để xóa người đặt mua ra khỏi mảng array.phương thức quan trọng cuối cùng của paper là publish() – dùng để gọi tới các phương thức của người đặt mua subscribers.khái quát lại thì 1 đối tượng publisher object cần các thành viên sau :

- subscribers 1 mảng array
- subscribe() thêm 1 người đặt mua subscriber vào mảng array
- unsubscribe() xóa phần tử từ mảng subscribers array
- publish() lập thông qua các người đặt mua subscribers và gọi các phương thức mà chúng cung cấp khi chúng đăng kí

tất cả 3 phương thức trên cần 1 tham số type bởi vì 1 người xuất bản publisher có thể gây ra 1 vài sự kiện events (xuất bản cả 2 loại : 1 tạp chí và 1 tờ báo) và người đặt mua subscribers có thể chọn đặt mua tới 1 cái và không được đặt mua tới cái còn lại

bởi vì các thành viên này đều có đặc điểm chung cho bất kì đối tượng publisher object nào nên nó có thể thực thi xử lý chúng như 1 phần của 1 đối tượng riêng biệt.sau đó ta có thể sao chép chúng (mix-in pattern) tới bất kì đối tượng nào và biến đổi bất cứ đối tượng đã biết nào thành 1 nhà xuất bản publisher.

Đây là mã thực thi của 1 mẫu đối tượng publisher khái quát – nó định nghĩa tất cả các thành viên bắt buộc mà đã liệt kê ra ở trên và thêm vào đó phương thức phụ trợ **visitSubscribers()** method:

```
var publisher = {  
  subscribers: {
```

Subscribers là 1 đối tượng mà nó có chứa 1 thuộc tính tên là **any**

```
    any: [] // event type: subscribers
```

```
  },
```

```
  subscribe: function (fn, type) {
```

Phương thức dùng để thêm người đặt mua **subscribe** vào thuộc tính mảng **any** của đối tượng **subscribers**

```
    type = type || 'any';
```

```
    if (typeof this.subscribers[type] === "undefined") {  
      this.subscribers[type] = [];  
    }
```

```
    this.subscribers[type].push(fn);
```

```
  },
```

```
  unsubscribe: function (fn, type) {
```

```
    this.visitSubscribers('unsubscribe', fn, type);
```

```
  },
```

```
  publish: function (publication, type) {
```

```
    this.visitSubscribers('publish', publication, type);
```

```
  },
```

```
  visitSubscribers: function (action, arg, type) {
```

```
    var pubtype = type || 'any',
```

```
    subscribers = this.subscribers[pubtype],
```

```
    i,
```

```
    max = subscribers.length;
```

```
    for (i = 0; i < max; i += 1) { // duyệt các thành viên thông qua 1 vòng lặp
```

```
      if (action === 'publish') { // nếu thao tác là publish
```

```
        subscribers[i](arg);
```

```
      } else { // nếu là thao tác khác publish , ví dụ như là thao tác unsubscribe
```

Phương thức duyệt tất cả các thành viên **subscribers** chứa trong thuộc tính **any** của đối tượng **subscribe**. sử dụng mệnh đề điều kiện để phân loại các thao tác **action** lên các thành viên

```
if (subscribers[i] === arg) {  
  subscribers.splice(i, 1);  
}  
}  
}  
}  
};
```

Đây là 1 hàm function cần 1 đối tượng object và biến đổi nó hành 1 publisher bằng cách sao chép đơn giản trên các phương thức chung :

```
function makePublisher(o) {  
  var i;  
  for (i in publisher) {  
    if (publisher.hasOwnProperty(i) && typeof publisher[i] === "function") {  
      o[i] = publisher[i];  
    }  
  }  
  o.subscribers = {any: []};  
}
```

Giờ ta thực thi đối tượng paper object. tất cả những gì nó có thể làm là xuất bản theo ngày và theo tháng :

```
var paper = {  
  daily: function () {  
    this.publish("big news today");  
  },  
  monthly: function () {  
    this.publish("interesting analysis", "monthly");  
  }  
};
```

Cách tạo paper như là 1 publisher:

```
makePublisher(paper);
```

giờ ta có 1 publisher, và ta xem xét đối tượng subscriber object là joe – đối tượng có 2 phương thức sau :

```
var joe = {  
  drinkCoffee: function (paper) {  
    console.log('Just read ' + paper);  
  },  
  sundayPreNap: function (monthly) {  
    console.log('About to fall asleep reading this ' + monthly);  
  }  
};
```

Giờ joe đặt mua paper :

```
paper.subscribe(joe.drinkCoffee);  
paper.subscribe(joe.sundayPreNap, 'monthly');
```

như ta nhìn thấy joe cung cấp 1 phương thức được gọi theo sự kiện mặc định any và 1 phương thức khác được gọi khi kiểu sự kiện “monthly” xảy ra. giờ ta kích hoạt 1 vài sự kiện :

```
paper.daily();  
paper.daily();  
paper.monthly();
```

tất cả các ấn phẩm này gọi tới các phương thức tương ứng của joe và kết quả trong màn hình console là :

```
Just read big news today  
Just read big news today  
About to fall asleep reading this interesting analysis
```

Phần hay nhất ở đây là đối tượng paper không gắn mã cứng joe và joe không gắn cứng với paper. các đối tượng tham gia được ghép cặp không chặt và không phải hiệu chỉnh lại tất cả

chúng khi ta thêm vào nhiều hơn subscribers vào paper và cũng như vậy joe có thể unsubscribe bất cứ khi nào

Giờ ta xem xét ví dụ này ở xa hơn và làm joe thành 1 publisher. vì vậy joe trở thành 1 publisher và có thể post status trên Twitter :

```
makePublisher(joe);  
joe.tweet = function (msg) {  
  this.publish(msg);  
};
```

Giờ hình dung rằng quan hệ bộ phận công khai của paper quyết định đọc những gì người đọc tweet và đặt mua tới joe, bằng cách cung cấp phương thức readTweets():

```
paper.readTweets = function (tweet) {  
  alert('Call big meeting! Someone ' + tweet);  
};  
joe.subscribe(paper.readTweets);
```

giờ khi joe tweet thì paper sẽ hiện lên thông báo :

```
joe.tweet("hated the paper today");
```

kết quả trong thông báo là “Call big meeting! Someone hated the paper today.”

X.Asynchronous patterns - Kiểu mẫu không đồng bộ

Kiểu mẫu không đồng bộ đang trở nên phổ biến hơn và quan trọng hơn để nâng cao hiệu suất lập trình web lên phía trước. chúng đang là vấn đề thách thức khi làm việc trong JS, giúp tạo ra các kiểu mẫu không đồng bộ (hay đồng bộ) 1 cách dễ dàng hơn, các thư viện JS (như JQuery và Dojo) đã thêm vào 1 khái niệm mới được gọi là promises (hay còn được gọi là deferreds). với các thư viện này thì người lập trình có thể sử dụng promises trong bất cứ trình duyệt nào có hỗ trợ ECMAScript 5

10.1.Lợi ích và thách thức với lập trình không đồng bộ

Ta xét ví dụ 1 trang web page bắt đầu 1 thao tác xử lý không đồng bộ giống như [XMLHttpRequest2](#) (XHR2) hay [Web Workers](#). Có 1 ưu điểm là 1 vài công việc được xử lý theo cách song song. điều này gây ra sự phức tạp cho người lập trình để giữ trang page phản hồi tới người dùng và không làm trở ngại sự tương tác trong khi vẫn điều phối những gì mà trang web page đang thực hiện với công việc không đồng bộ. điều này là phức tạp bởi vì chương trình thực thi xử lý không còn được tuân theo cách xử lý tuyến tính đơn giản nữa

Khi ta tạo ra 1 lời gọi không đồng bộ thì ta cần điều chỉnh sự hoàn thành thành công của công việc và các lỗi có khả năng xảy ra trong quá trình thực thi. với lời gọi sự hoàn thành thành công của 1 công việc không đồng bộ thì rất có thể ta muốn chuyển tiếp kết quả để tạo ra các yêu cầu Ajax request khác. điều này có thể dẫn tới sự phức tạp thông qua các hàm callbacks lồng nhau :

```
function searchTwitter(term, onload, onerror) {  
  
    var xhr, results, url;  
  
    url = 'http://search.twitter.com/search.json?rpp=100&q=' + term;  
  
    xhr = new XMLHttpRequest();  
    xhr.open('GET', url, true);  
  
    xhr.onload = function (e) {  
        if (this.status === 200) {  
            results = JSON.parse(this.responseText);  
        }  
    }  
}
```



```
onload(results);  
}  
};
```

```
xhr.onerror = function (e) {  
onerror(e);  
};  
  
xhr.send();  
}
```

```
function handleError(error) {  
/* handle the error */  
}
```

```
function concatResults() {  
/* order tweets by date */  
}
```

```
function loadTweets() {  
var container = document.getElementById('container');
```

```
searchTwitter('#IE10', function (data1) {
```

```
searchTwitter('#IE9', function (data2) {
```

```
/* Reshuffle due to date */
```

```
var totalResults = concatResults(data1.results, data2.results);
```

```
totalResults.forEach(function (tweet) {  
var el = document.createElement('li');  
el.innerText = tweet.text;  
container.appendChild(el);
```

searchTwitter(term, **onload**, **onerror**)

searchTwitter('#IE10', function (data1) { ... },

searchTwitter('#IE9', function (data2) { ... }, handleError)

```
});  
, handleError);  
}, handleError);  
  
}
```

Các hàm callback lồng nhau khiến đoạn mã code trở nên khó hiểu – khó phân biệt được code nào là business logic dùng để định nghĩa ứng dụng app và cái nào là code mẫu được yêu cầu làm việc với lời gọi không đồng bộ ?thêm vào đó, tương xứng với các hàm callbacks lồng nhau thì sự hiệu chỉnh lỗi trở thành bị phân mảnh.ta phải kiểm tra 1 vài nơi để xem xét nếu có 1 lỗi xảy ra

Việc giảm thiểu sự phức tạp trong điều phối cách thức thực thi không đồng bộ thì khiến người lập trình có cái nhìn thực thi xử lý nhất quán hơn và dễ để hiểu các hiệu chỉnh lỗi với những cái khác trong các hàm callbacks lồng nhau

10.2.Promises

Kiểu mẫu được gọi là *promise* khi nó biểu diễn kết quả của 1 thao tác có khả năng xử lý trong 1 thời gian dài và thao tác vẫn chưa kết thúc.thay thế cho việc cản trở và đợi cho việc tính toán xử lý trong thời gian dài kết thúc thì kiểu mẫu này trả về 1 đối tượng dùng để biểu diễn kết quả promised

1 ví dụ của kiểu mẫu này có thể tạo ra 1 yêu cầu tới hệ thống của bên thứ 3 với độ trễ về network là biến đổi.thay thế cho việc ngăn cản toàn bộ ứng dụng trong khi chờ đợi – trong lúc chờ đợi này thì ứng dụng đang được tự do để có thể làm những thứ khác cho tới khi giá trị là cần thiết.1 promise thực thi xử lý 1 phương thức để đăng ký hàm callback danh cho các thông báo thay đổi trạng thái, thông thường nó có tên là **then** :

```
var results = searchTwitter(term).then(filterResults);  
  
displayResults(results);
```

vào bất cứ thời điểm nào thì promises có thể có 1 trong 3 trạng thái sau :

unfulfilled (chưa được hoàn thành), resolved (đã được giải quyết) hay rejected (bị bác bỏ).

Phương thức then trên đối tượng promise object thêm vào các hàm điều chỉnh handlers cho các trạng thái) resolved (đã được giải quyết) hay rejected (bị bác bỏ).hàm function này trả về 1 đối tượng promise object khác để cho phép sự xử lý tiên lệnh promise, cho phép người lập trình trói buộc các thao tác 1 cách đồng bộ với nhau,với kết quả của thao tác đầu tiên sẽ được chuyển tiếp vào thao tác thứ 2

then(resolvedHandler, rejectedHandler);

hàm *resolvedHandler* callback được gọi khi promise đang ở trạng thái chưa hoàn thành, được chuyển tiếp kết quả của xử lý tính toán.hàm *rejectedHandler* được gọi khi promise rơi vào trạng thái thất bại

ta xem lại ví dụ trên bằng cách sử dụng giả mã của 1 promise để tạo ra yêu cầu AJAX tới search Twitter, dẫn đến màn hình với hiện thị các dữ liệu và các hiệu chỉnh lỗi.ta bắt đầu 1 ví dụ về 1 thư viện promise sẽ trông giống như thế nào nếu ta thiết kế nó từ 1 vài đoạn mã hết sức cơ bản.đầu tiên ta sẽ cần 1 vài dạng của đối tượng để lưu trữ promise.

```
var Promise = function () {  
  /* initialize promise */  
};
```

Giờ ta cần 1 cặp phương thức để thực thi xử lý 1 trạng thái chuyển tiếp giữa trạng thái unfulfilled (chưa hoàn thành) và trạng thái resolved (đã được giải quyết) hay trạng thái rejected (bị bác bỏ)

```
Promise.prototype.resolve = function (value) {  
  /* move from unfulfilled to resolved */  
};
```

```
Promise.prototype.reject = function (error) {  
  /* move from unfulfilled to rejected */  
};
```

Giờ ta có 1 vài bản mẫu dành cho 1 đối tượng Promise object có thể là cái gì, giờ ta xem xét thông qua ví dụ ở phía trên – ví dụ về truy vấn Twitter với thẻ tag #IE10. đầu tiên ta sẽ tạo ra 1 phương thức cho việc tạo ra 1 yêu cầu Ajax GET bằng cách sử dụng XMLHttpRequest2 tới 1 URL đã biết nào đó và đóng gói nó vào trong 1 promise, tiếp theo ta tạo ra 1 phương thức đặc biệt dành cho Twitter bằng cách gọi 1 phương thức đóng gói Ajax với 1 cụm từ tìm kiếm đã biết. cuối cùng ta sẽ gọi hàm search function và hiển thị kết quả trong 1 danh sách không thứ tự

```
function searchTwitter(term) {  
  
    var url, xhr, results, promise;  
    url = 'http://search.twitter.com/search.json?rpp=100&q=' + term;  
    promise = new Promise();  
    xhr = new XMLHttpRequest();  
    xhr.open('GET', url, true);  
  
    xhr.onload = function (e) {  
        if (this.status === 200) {  
            results = JSON.parse(this.responseText);  
            promise.resolve(results);  
        }  
    };  
  
    xhr.onerror = function (e) {  
        promise.reject(e);  
    };  
  
    xhr.send();  
  
    return promise;  
}  
  
function loadTweets() {  
    var container = document.getElementById('container');
```

```
searchTwitter('#IE10').then(function (data) {
  data.results.forEach(function (tweet) {
    var el = document.createElement('li');
    el.innerText = tweet.text;
    container.appendChild(el);
  });
}, handleError);
}
```

Giờ ta có thể tạo ra 1 yêu cầu Ajax đơn giống như 1 đối tượng Promise. giờ ta bàn tiếp về 1 kịch bản khi ta muốn tạo ra nhiều hơn 1 yêu cầu AJAX và điều phối các kết quả. để điều khiển được kịch bản này thì ta sẽ tạo ra 1 phương thức when trên đối tượng Promise object để xếp hàng các promises cho việc gọi về. khi promises di chuyển từ trạng thái unfulfilled tới 1 trong 2 trạng thái resolved hay rejected thì 1 hiệu chỉnh tương ứng sẽ được gọi trong phương thức then. phương thức when về bản chất là 1 thao tác 3 chắc dùng để đợi sự hoàn thành của tất cả các thao tác trước khi tiếp tục xử lý

```
Promise.when = function () {
  /* handle promises arguments and queue each */
};
```

Giờ ta có thể xếp hàng nhiều promises 1 cách đồng thời, ví dụ như việc tìm kiếm cả 2 #IE10 và #IE9 trên twitter :

```
var container, promise1, promise2;

container = document.getElementById('container');
```

```
promise1 = searchTwitter('#IE10');
```

```
promise2 = searchTwitter('#IE9');
```

```
Promise.when(promise1, promise2).then(function (data1, data2) {
  /* Reshuffle due to date */
})
```

```
var totalResults = concatResults(data1.results, data2.results);

totalResults.forEach(function (tweet) {
    var el = document.createElement('li');
    el.innerText = tweet.text;
    container.appendChild(el);
});
}, handleError);
```

Điều quan trọng cần nhớ là mã code trong những ví dụ này là không có gì ngoài các mã JS thông thường. các nhà lập trình web tất nhiên có thể tạo ra các thư viện giống promise của chính họ nhưng để cho tiện và co động thì ta có thể tận dụng được kiểu mẫu promises patterns có trong các thư viện JS phổ biến

10.3. Khám phá Promises trong bộ công cụ JQuery

Jquery giới thiệu về 1 khái niệm mới trong phiên bản 1.5 gọi là *Deferred* – nó cũng là 1 hệ thống xử lý bất nguồn từ Promises/A proposal. đối tượng *Deferred* object được bộc lộ ra 1 phương thức then – cho phép người lập trình hiệu chỉnh các trạng thái fulfillment (đã hoàn thành) và error. đối tượng này có 2 phương thức *resolve* và *reject*. người lập trình có thể tạo ra 1 đối tượng Deferred object trong JQuery bằng cách gọi hàm **\$.Deferred** function.

```
function xhrGet(url) {
    var xhr, results, def;
    def = $.Deferred();
    xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);

    xhr.onload = function (e) {
        if (this.status === 200) {
            results = JSON.parse(this.responseText);
            def.resolve(results);
        }
    };

    xhr.onerror = function (e) {
        def.reject(e);
    };
}
```

```
};

xhr.send();

return def;
}

function searchTwitter(term) {
    var url, xhr, results, def;
    url = 'http://search.twitter.com/search.json?rpp=100&q=' + term;
    def = $.Deferred();
    xhr = new XMLHttpRequest();
    xhr.open('GET', url, true);

    xhr.onload = function (e) {
        if (this.status === 200) {
            results = JSON.parse(this.responseText);
            def.resolve(results);
        }
    };

    xhr.onerror = function (e) {
        def.reject(e);
    };

    xhr.send();

    return def;
}

$(document).ready(function () {
    var container = $('#container');

    searchTwitter('#IE10').then(function (data) {
        data.results.forEach(function (tweet) {
            container.append('<li>' + tweet.text + '</li>');
        });
    });
});
```

```
});  
});
```

JQuery không trả về 1 promise nào khác từ phương thức `then`. thay vào đó, JQuery cung cấp 1 phương thức *pipe* để trói buộc các thao tác lại với nhau. bổ sung thêm, JQuery đưa ra 1 phương thức tiện ích khác cho phép sự kết tụ chặt hơn bao gồm việc lọc thông qua phương thức *pipe*

JQuery 1.5 cũng sửa đổi lại các phương thức Ajax giờ chúng trả về đối tượng jqXHR object – thực thi xử lý 1 cách trực tiếp từ giao diện chung promise interface.

```
$.ajax({  
  url: 'http://search.twitter.com/search.json',  
  dataType: 'jsonp',  
  data: { q: '#IE10', rpp: 100 }  
}).then(function (data) {  
  /* handle data */  
}, function (error) {  
  /* handle error */  
});
```

Để chặt chẽ hơn, phương thức `jQuery.ajax` cung cấp các phương thức *success*, *error*, and *complete* methods

```
$.ajax({  
  url: 'http://search.twitter.com/search.json',  
  dataType: 'jsonp',  
  data: { q: '#IE10', rpp: 100 }  
}).success(function (data) {  
  /* handle data */  
}).error(function (error) {  
  /* handle error */  
});
```


XI.ASYNCH JS : sức mạnh của đối tượng \$.DEFERRED

1 trong những khía cạnh quan trọng về cách xây dựng các ứng dụng HTML5 mượt mà và có sự đáp ứng nhanh là sự đồng bộ giữa tất cả các thành phần khác biệt của ứng dụng như việc lấy dữ liệu, cách xử lý, các hiệu ứng, và các thành phần giao diện người dùng

Sự khác biệt chính với 1 môi trường Desktop hay 1 môi trường bản địa là các trình duyệt không đưa ra khả năng truy cập theo mô hình threading model mà chỉ cung cấp 1 thread duy nhất cho mọi thứ đang truy cập vào giao diện người dùng (ví dụ như DOM). Điều này có nghĩa là tất cả điều kiện application logic đang truy cập và hiệu chỉnh các thành phần giao diện người dùng luôn trong cùng 1 thread, do đó việc quan trọng để giữ tất cả ứng dụng hoạt động theo các đơn vị càng nhỏ và càng hiệu quả càng tốt và nắm bắt ưu điểm của các khả năng không đồng bộ mà trình duyệt đưa ra càng nhiều càng tốt

11.1.Các hàm APIS không đồng bộ của trình duyệt

May mắn thay là trình duyệt cung cấp 1 số các hàm APIs không đồng bộ giống như XHR (XMLHttpRequest or 'AJAX') APIs mà ta vẫn thường dùng, hay các IndexedDB, SQLite, HTML5 Web workers, và HTML5 Geolocation APIs. ngay cả 1 vài DOM liên quan đến thao tác cũng được bộc lộ ra là không đồng bộ giống như CSS3 animation thông các sự kiện transitionEnd events.

Cách mà trình duyệt trình bày cách lập trình không đồng bộ tới application logic là thông qua sự kiện events và các hàm callbacks. trong các hàm APIs không đồng bộ dựa trên sự kiện event thì người lập trình đăng kí 1 điều chỉnh event handler cho 1 đối tượng đã biết nào đó (ví dụ như phần tử HTML hay các đối tượng DOM objects) và sau đó gọi các thao tác xử lý. trình duyệt sẽ thực thi xử lý thao tác này thông thường trong 1 thread khác và kích hoạt sự kiện trong thread chính

Ví dụ : mã code sử dụng XHR API, 1 sự kiện dựa trên API không đồng bộ như sau :

```
// Create the XHR object to do GET to /data resource
var xhr = new XMLHttpRequest();
xhr.open("GET","data",true );
// register the event handler
xhr.addEventListener('load',function (){
if (xhr.status === 200){
alert("We got data: " + xhr.response);
```

```
}  
,false )  
// perform the work  
xhr.send();
```

sự kiện CSS3 transitionEnd event cũng là 1 ví dụ khác của 1 sự kiện event dựa vào API không đồng bộ

```
// get the html element with id 'flyingCar'  
var flyingCarElem = document.getElementById("flyingCar");  
// register an event handler  
// ('transitionEnd' for FireFox, 'webkitTransitionEnd' for webkit)  
flyingCarElem.addEventListener("transitionEnd",function () {  
// will be called when the transition has finished.  
alert("The car arrived");  
});  
// add the CSS3 class that will trigger the animation  
// Note: some browsers delegate some transitions to the GPU , but  
// developer does not and should not have to care about it.  
flyingCarElem.classList.add('makeItFly')
```

APIS khác của trình duyệt như SQLi te và HTML5 Geolocation đầu dựa trên các hàm callback, điều này có nghĩa là người lập trình chuyển tiếp 1 hàm function giống như 1 tham số và sẽ dùng để gọi lại bởi thực thi nằm dưới với giải pháp tương ứng

Ví dụ như HTML5 Geolocation mã code giống như sau :

```
// call and pass the function to callback when done.  
navigator.geolocation.getCurrentPosition(function(position){  
alert('Lat: ' + position.coords.latitude + ' ' +  
'Lon: ' + position.coords.longitude);  
});
```

Trong trường hợp này ta chỉ gọi 1 phương thức và chuyển tiếp 1 hàm function dùng để gọi lại với kết quả đã được yêu cầu.điều này cho phép trình duyệt thực thi xử lý tính năng này 1 cách

đồng bộ và không đồng bộ và đưa ra 1 hàm API đơn tới người lập trình bất chấp thông tin thực thi chi tiết

11.2.Cách tạo các ứng dụng 1 cách không đồng bộ

Vượt xa ngoài các hàm APIS không đồng bộ được xây dựng sẵn thì 1 ứng dụng có kiến trúc tốt nên trình bày các hàm APIS ở mức thấp trong 1 kiểu không đồng bộ, về bản chất khi chúng làm việc với bất cứ kiểu I/O nào hay bất kì kiểu tiến trình tính toán nặng nề nào.ví dụ APIS lấy dữ liệu thì nên tuân theo sự không đồng bộ và không nên giống như đoạn mã sau :

```
// Sai lầm : điều này sẽ khiến UI đóng băng khi tiến hành thực hiện lấy dữ liệu
```

```
var data = getData();  
alert("We got data: " + data);
```

thiết kế API này đòi hỏi getData() để thực hiện cản trở - sẽ đóng băng giao diện người dùng tới khi dữ liệu được lấy về.nếu dữ liệu là địa phương trong ngữ cảnh JS thì điều này có thể không gây ra hậu quả nào tuy nhiên nếu dữ liệu cần lấy về ở 1 mạng network hay ngay cả địa phương trong 1 SQLite hay 1 index store thì nó cũng tác động mạnh tới trải nghiệm của người dùng

thiết kế đúng đắn là tại ra tất cả các hàm API của ứng dụng mà cần 1 khoảng thời gian nào đó để thực thi xử lý thành không đồng bộ hóa

ví dụ như hàm getData() API đơn giản như sau :

```
getData(function (data) {  
    alert("We got data: " + data);  
});
```

1 điều tốt về cách tiếp cận này là nó ép buộc UI của ứng dụng trở thành không đồng bộ từ lúc bắt đầu và cho phép các hàm APIS bên dưới quyết định nếu chúng cần sự không đồng bộ hay không trong 1 giai đoạn sau đó

Chú ý rằng không phải tất cả API của ứng dụng cần hay nên trở thành không đồng bộ.quy tắc ngón tay cái là bất cứ API nào mà làm việc với bất cứ kiểu I/O nào hay thực thi xử lý nặng nề

(bất cứ thứ gì cần nhiều hơn 15ms) thì nên được trình bày theo sự không đồng bộ từ khi bắt đầu ngay cả khi thực thi xử lý đầu tiên của nó là đồng bộ

11.3.Cách hiệu chỉnh sự thất bại

Việc nắm bắt trong lập trình không đồng bộ theo kiểu truyền thống là try/catch dùng để hiệu chỉnh sự thất bại thì giờ cách này không thực sự hoạt động nữa – các lỗi thông thường xảy ra ở trong thread khác.do đó callee cần phải có 1 cách có cấu trúc để thông báo tới caller khi 1 vài thứ gì đó trở nên sai lầm trong quá trình xử lý

Trong 1 API không đồng bộ dựa trên sự kiện event thì nó thường được hoàn thành bởi mã ứng dụng truy vấn tới sự kiện event hay đối tượng object khi nhận được sự kiện event.đối với các APIS không đồng bộ dựa trên các hàm callback thì cách thực hành tốt nhất là có 1 tham số thứ 2 – là 1 hàm function sẽ được gọi trong trường hợp 1 thất bại với 1 thông tin lỗi tương ứng như là tham số

getData sẽ trở thành thể này :

```
// getData(successFunc,failFunc);

getData(function (data){
  alert("We got data: " + data);
}, function (ex) {
  alert("oops, some problem occurred: " + ex);
});
```

11.4.\$@DEFERRED

1 sự giới hạn của cách tiếp cận theo hàm callback ở trên là nó có thể thực sự trở thành 1 gánh nặng ngay cả khi viết 1 điều kiện logic đồng bộ chuyên sâu vừa phải

Ví dụ nếu ta cần đợi 2 API không đồng bộ được thực hiện xong trước khi làm điều thứ 3 thì đoạn mã code có thể trở nên phức tạp lên rất nhanh

```
// first do the get data.

getData(function (data) {
```

```
// then get the location

getLocation(function (location) {

    alert("we got data: " + data + " and location: " + location);

}, function (ex){

    alert("getLocation failed: " + ex);

});

}, function (ex){

    alert("getData failed: " + ex);

});
```

Mọi thứ có thể trở nên phức tạp hơn khi ứng dụng cần tạo ra cùng 1 lời gọi từ nhiều phần của ứng dụng, cũng như mọi lời gọi sẽ phải được thực thi những lời gọi này theo nhiều bước hay ứng dụng sẽ phải thực thi chính cơ chế caching của nó

May mắn thay, có 1 kiểu mẫu tương đối cũ có tên là Promises và 1 thực thi xử lý hiện đại và rất thiết thực trong JQuery có tên là \$.Deferred – cung cấp 1 giải pháp đơn giản và mạnh mẽ để giải quyết vấn đề lập trình không đồng bộ

Để khiến nó đơn giản thì kiểu mẫu Promises pattern định nghĩa API không đồng bộ trả về 1 đối tượng Promise object. theo giải pháp này thì caller lấy về đối tượng Promise object và gọi tới done(successFunc(data)) – điều này sẽ nói cho đối tượng Promise object gọi tới hàm successFunc khi dữ liệu được giải quyết xong

Vì vậy getData ở ví dụ trên giống như sau :

```
// get the promise object for this API
var dataPromise = getData();

// register a function to get called when the data is resolved
dataPromise.done(function (data){
    alert("We got data: " + data);
});

// register the failure function
dataPromise.fail(function (ex){
```

```
alert("oops, some problem occurred: " + ex);
});
// Note: we can have as many dataPromise.done(...) as we want.
dataPromise.done(function (data){
alert("We asked it twice, we get it twice: " + data);
});
```

Giờ ta lấy về được đối tượng dataPromise object ở đầu tiên và sau đó gọi tới phương thức .done để đăng kí 1 hàm function mà ta muốn gọi lại khi dữ liệu được giải quyết xong. ta cũng có thể gọi tới phương thức .fail để hiệu chỉnh lại sự thất bại khi nó xảy ra. chú ý rằng ta có thể có nhiều lời gọi . done hay . fail mà ta cần bởi vì hệ thống xử lý bên dưới Promise sẽ hiệu chỉnh đăng ký và các hàm callbacks

Với kiểu mẫu này, thì rất dễ dàng để thu thập và xử lý mã code đồng bộ cao cấp hơn và JQuery cung cấp 1 phương thức được dùng phổ biến là \$.when

Ví dụ, hàm callback getData/getLocation lồng nhau ở trên sẽ trở thành như sau :

```
// assuming both getData and getLocation return their respective Promise
var combinedPromise = $.when(getData(), getLocation())
// function will be called when both getData and getLocation resolve
combinedPromise.done(function (data, location){
alert("We got data: " + dataResult + " and location: " + location);
});
```

Và những sự tiện lợi của jQuery.Deferred sẽ làm cho người lập trình dễ dàng thực thi xử lý hàm function không đồng bộ. ví dụ getData sẽ như sau :

```
function getData(){
// 1) create the jQuery Deferred object that will be used
var deferred = $.Deferred ();
// ---- AJAX Call ---- //
XMLHttpRequest xhr = new XMLHttpRequest ();
xhr.open("GET", "data", true );
// register the event handler
```

```
xhr.addEventListener('load', function () {  
  
  if (xhr.status === 200){  
  
    // 3.1) RESOLVE the DEFERRED (this will trigger all the done()...)  
  
    deferred.resolve(xhr.response);  
  
  } else {  
  
    // 3.2) REJECT the DEFERRED (this will trigger all the fail()...)  
  
    deferred.reject("HTTP error: " + xhr.status);  
  
  }  
  
  }, false )  
  
  // perform the work  
  xhr.send();  
  
  // Note: could and should have used jQuery.ajax.  
  // Note: jQuery.ajax return Promise, but it is always a good idea to wrap it  
  // with application semantic in another Deferred/Promise  
  // ---- /AJAX Call ---- //  
  
  // 2) return the promise of this deferred  
  
  return deferred.promise();  
  
}
```

Vì vậy khi `getData()` được gọi thì đầu tiên nó tạo ra 1 đối tượng `jQuery.Deferred` object

(1) và sau đó trả về `Promise`(2) của nó do đó caller có thể đăng kí các hàm `done` và `fail`

functions. sau đó khi lời gọi `XHR` call được trả về thì nó sẽ giải quyết xong `deferred`

(3.1) hay từ bỏ nó (3.2). việc thực hiện `deferred.resolve` sẽ kích hoạt tất cả các hàm

`done(...)` functions và các hàm `promise` functions khác như (e.g., `then` và `pipe`) và cách gọi `deferred.reject` sẽ gọi toàn bộ hàm `fail()` functions

11.5. Các trường hợp mẫu

Đây là 1 vài trường hợp mẫu trong đó Deferred được sử dụng nhiều

- **Data Access:** việc trình bày data access APIs giống như là \$.Deferred là 1 thiết kế đúng. rõ ràng là với dữ liệu được truy cập từ xa thì các lời gọi remote calls sẽ có thể hủy hoại hoàn toàn trải nghiệm của người dùng và nó cũng đúng với dữ liệu địa phương ở mức độ thấp như SQLite và IndexedDB. phương thức \$.when và .pipe của Deferred API thực sự rất mạnh mẽ để đồng bộ và nối tiếp các câu truy vấn con không đồng bộ
- **UI Animations:** việc phối hợp 1 hay nhiều hiệu ứng với các sự kiện transitionEnd events có thể tạo ra sự buồn tẻ và về bản chất khi các hiệu ứng là 1 sự kết hợp giữa các CSS3 animation và JS. việc đóng gói các hàm animation functions giống như Deferred có thể giảm thiểu đáng kể sự phức tạp của mã code và cải thiện tính mềm dẻo. ngay cả 1 hàm đơn giản như cssAnimation(className) thì sẽ trả về đối tượng Promise mà sẽ giải quyết transitionEnd có thể là 1 sự trợ giúp lớn
- **UI Component Display:** điều này có 1 chút nâng cao hơn nhưng các khung thành phần HTML nâng cao nên sử dụng Deferred. không cần đi sâu quá nhiều vào chi tiết, khi 1 ứng dụng cần hiển thị các thành phần khác nhau của giao diện người dùng thì việc có vòng đời của các thành phần này được tóm gọn trong Deferred sẽ cho phép sự điều khiển to lớn hơn về mặt thời gian
- **Any browser asynchronous API:** với mục đích tầm thường hóa thì đây thông thường là 1 ý tưởng tốt để đóng gói lời gọi browser API calls trong Deferred. Điều này cần 4 tới 5 dòng code mỗi lần nhưng nó sẽ đơn giản hóa bất cứ mã ứng dụng nào. giống như đã chỉ ra ở trên trong giả mã getData/getLocation pseudo thì điều này cho phép mã ứng dụng có 1 mô hình không đồng bộ thông qua tất cả các dạng API
- **Caching:** đây là 1 lợi ích do tác dụng phụ nhưng có thể hữu dụng trong 1 vài cơ hội. bởi vì Promise APIs như done(..) và .fail(..) có thể được gọi trước hay sau lời khi lời gọi không đồng bộ được xử lý, đối tượng Deferred object có thể được sử dụng như 1 hiệu chỉnh caching cho 1 lời gọi không đồng bộ. ví dụ 1 CacheManager có thể chỉ cần lưu giữ dấu vết của Deferred dành cho các yêu cầu đã biết và trả về Promise của Deferred tương ứng nếu nó không bắt hợp lệ. điều hay ở đây là caller không cần

phải biết nếu lời gọi đã được giải quyết hay đang trong tiến trình được giải quyết do vậy hàm callback của nó sẽ được gọi chính xác theo trong cùng 1 cách

11.6.JQuery.Deferred và Promise

jQuery 1.5 giới thiệu 1 khái niệm mới là “Deferred”.ta sẽ nhận thấy rằng khái niệm mới này sẽ rất mạnh mẽ khi ta làm việc với JS và ajax và nó sẽ là 1 cơ hội lớn trong cách ta thay đổi viết các mã không đồng bộ trong JS

Thông thường, cách mà ta sử dụng để làm việc với mã không đồng bộ trong JS là chuyển tiếp hàm callback giống như 1 tham số tới 1 hàm function như sau :

```
$.ajax({
  url: "/echo/json/",
  data: {json: JSON.stringify({firstName: "Jose", lastName:
    "Romaniello"})},
  type: "POST",
  success: function(person) {
    alert(person.firstName + " saved.");
  },
  error: function() {
    alert("error!");
  }
});
```

Như ta có thể thấy ở đây thì ta chuyển tiếp vào 2 hàm callbacks (success và error) tới phương thức \$.ajax method

Điều này có hiệu quả nhưng nó không phải là 1 thực thể tiêu chuẩn và nó đòi hỏi 1 vài công việc bên trong phương thức POST và ta không thể đăng ký được nhiều hàm callbacks

11.6.1.\$.Deferred

deferred object có 2 phương thức quan trọng :

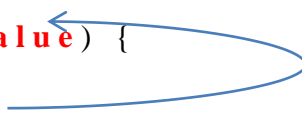
- Resolve
- Reject

Và nó có 3 sự kiện events quan trọng hay 3 cách để đính thêm vào 1 hàm callback :

- Done
- Fail
- Always

1 ví dụ hết sức đơn giản như sau :


```
var deferred = $.Deferred();
deferred.done(function( value ) {
    alert(value);
});
deferred.resolve( "hello world" ); // thông báo đối tượng deferred đã
được giải quyết xong và nó truyền vào 1 tham số « hello world » - tham số này giống
như kết quả của việc hoàn thành xử lý và nó được truyền vào bên trong các hàm function
trong phương thức deferred.done
```



nếu ta gọi phương thức "reject" thì hàm **fail callback** được đính thêm vào sẽ được xử lý, hàm **always callback** luôn được thực thi xử lý bất cứ khi nào đối tượng deferred được resolved hay rejected.

1 điểm thú vị khác là nếu ta đính thêm vào 1 hàm callback và 1 đối tượng deferred đã được resolved từ trước đó thì nó cũng được thực thi xử lý ngay lập tức :

```
var deferred = $.Deferred();
deferred.resolve( "hello world" );
deferred.done(function( value ) {
    alert(value);
});
```



11.6.2. Phương thức `deferred.resolve()`

`deferred.resolve(args)` trả về [Deferred](#)

giải quyết 1 đối tượng Deferred object và gọi bất kì hàm doneCallbacks nào với các tham số **args** đã cho trước

với **args** là các tham số không bắt buộc được dùng để chuyển tiếp vào các hàm doneCallbacks.

Khi Deferred được giải quyết thì bất cứ hàm doneCallbacks nào được thêm vào bởi [deferred.then](#) hay [deferred.done](#) sẽ được gọi. các hàm callbacks được thực thi xử lý theo thứ tự mà chúng được thêm vào. mỗi hàm callback được chuyển tiếp vào tham số **args** từ phương thức `.resolve()`. bất cứ hàm doneCallbacks được thêm vào sau khi Deferred đi vào trong thái resolved state thì sẽ được thực thi ngay lập tức khi chúng được thêm vào, bằng cách sử dụng các tham số mà được chuyển tiếp bởi lời gọi `.resolve()`

11.6.3. Phương thức `deferred.reject()`

`deferred.reject(args)` trả về [Deferred](#)

loại bỏ 1 đối tượng Deferred object và gọi tới bất cứ hàm failCallbacks nào với các tham số **args** cho trước

với **args** là các tham số không bắt buộc được dùng để chuyển tiếp vào các hàm doneCallbacks.

Thông thường, chỉ có bộ tạo lập của 1 Deferred thì mới nên gọi phương thức này; ta có thể cản trở các mã code khác từ việc thay đổi trạng thái của Deferred bằng cách trả về 1 đối tượng Promise object bị giới hạn thông qua [deferred.promise\(\)](#).

Khi Deferred bị loại bỏ rejected thì bất cứ hàm failCallbacks nào được thêm vào bởi [deferred.then](#) hay [deferred.fail](#) đều được gọi. các hàm callbacks được thực thi xử lý theo thứ tự chúng được thêm vào. mỗi hàm callback được chuyển tiếp vào tham số args từ lời gọi `deferred.reject()`. bất cứ hàm failCallbacks được thêm vào sau khi Deferred đi vào trong thái resolved state thì sẽ được thực thi ngay lập tức khi chúng được thêm vào, bằng cách sử dụng các tham số mà được chuyển tiếp bởi lời gọi `.reject ()`

11.6.4. Phương thức Promise()

Đối tượng Deferred object có 1 phương thức quan trọng khác có tên là Promise(). phương thức này trả về 1 đối tượng với cùng 1 giao diện chung interface hơn là 1 đối tượng Deferred đơn thuần, nhưng nó chỉ có các phương thức để đính thêm vào hàm callbacks và không có phương thức dùng để resolve và reject

Điều này rất hữu dụng khi ta muốn đưa ra cách gọi API để cho 1 vài thứ có thể đăng ký nhận nó nhưng lại không có khả năng resolve hay reject deferred. đoạn mã code dưới sẽ thất bại bởi vì promise không có phương thức "resolve" :

```
function getPromise() {  
    return $.Deferred().promise();  
}  
try {  
    getPromise().resolve("a");  
}  
catch(err) {  
    alert(err);  
}
```

Phương thức \$.ajax() trong JQuery trả về 1 Promise, do vậy ta có thể làm như sau :

```
var post = $.ajax({  
  
    url: "/echo/json/",
```

```
data: {json: JSON.stringify({firstName: "Jose", lastName: "Romaniello"})} ,
```

```
type: "POST"
```

```
});
```

```
post.done(function(p) {  
    alert(p.firstName + " saved.");  
});
```

```
post.fail(function() {  
    alert("error!");  
});
```

P là tham số tự động được nhận về khi \$.ajax() được thực hiện xong và ta không phải truyền bằng tay như **deferred.resolve(n)**

Post là 1 promise nên nó có sẵn 2 phương thức **post.done** và **post.fail** và bản thân nó tự biết khi nào nó resolved và rejected mà không cần phải resolved và rejected bằng tay nữa

Đoạn mã code này cũng thực hiện giống với đoạn mã đầu tiên với chỉ 1 sự khác biệt là ta có thể thêm vào nhiều hàm callback nếu ta muốn, và cú pháp này rất rõ ràng bởi vì ta không cần 1 tham số bổ sung vào trong phương thức. theo cách khác, **ta có thể sử dụng promise để trả về 1 giá trị được trả về của hàm function khác** và đây là 1 trong những điều thú vị nhất mà ta có thể làm 1 số thao tác với promises

11.6.4.1. Chi tiết về *deferred.promise()*

deferred.promise([target]) trả về **Promise**

trả về 1 đối tượng Promise của Deferred

với **target** là đối tượng mà ở trên nó phương thức promise phải được đính thêm vào

phương thức deferred.promise() cho phép 1 hàm không đồng bộ ngăn cản mã code khác được can thiệp từ 1 tiến trình xử lý hay trạng thái của các yêu cầu bên trong nó. Promise bộc lộ duy nhất các phương thức Deferred cần thiết được đính thêm vào các hàm điều chỉnh handlers - bổ sung hay xác định trạng thái (then, done, fail, always, pipe, progress, và state) nhưng

không phải những phương thức dùng để thay đổi trạng thái (resolve, reject, progress, resolveWith, rejectWith, và progressWith).

Nếu target được sử dụng thì deferred.promise() sẽ đính thêm các phương thức vào nó và trả về đối tượng này hơn là tạo ra 1 đối tượng mới. điều này có thể hữu dụng khi đính thêm trạng thái xử lý Promise vào 1 đối tượng đã tồn tại

Nếu ta đang tạo ra 1 Deferred thì việc giữ 1 tham chiếu tới Deferred và do vậy nó có thể resolved hay rejected tại 1 vài thời điểm. việc trả về duy nhất 1 đối tượng Promise object thông qua deferred.promise() do vậy các mã code khác có thể đăng kí hay quan sát trạng thái hiện tại

11.6.4.2. Ví dụ :

Tạo ra 1 **Deferred** và thiết lập 2 hàm function dựa vào timer để resolve hay reject **Deferred**

Sau 1 khoảng thời gian ngẫu nhiên. bất cứ cái nào kích hoạt đầu tiên sẽ thắng “wins” và sẽ gọi 1 hàm callbacks. **Timeout** thứ 2 không có hiệu ứng nào bởi vì **Deferred** vừa mới hoàn thành (trong trạng thái resolve hay reject) từ thao tác timeout đầu tiên. ta cũng thiết lập tiến trình xử lý dựa vào timer để thông báo hàm function, và gọi 1 hiệu chỉnh tiến trình **progress handler** bằng cách thêm vào "**working...**" vào thân document

```
function asyncEvent(){
    var dfd = new jQuery.Deferred();

    // Resolve after a random interval
    setTimeout(function(){
        dfd.resolve("hurray");
    }, Math.floor(400+Math.random()*2000));

    // Reject after a random interval
    setTimeout(function(){
        dfd.reject("sorry");
    }, Math.floor(400+Math.random()*2000));

    // Show a "working..." message every half-second
    setTimeout(function working(){
```

```
if ( dfd.state() === "pending" ) {
    dfd.notify("working... ");
    setTimeout(working, 500);
}
}, 1);
// Return the Promise so caller can't change the Deferred
// trả về Promise do đó caller không thể thay đổi Deferred
return dfd.promise();
}

// Attach a done, fail, and progress handler for the asyncEvent
$.when( asyncEvent() ).then(
    function(status){
        alert( status+', things are going well' );
    },
    function(status){
        alert( status+', you fail this time' );
    },
    function(status){
        $("body").append(status);
    }
);
```

dfd.promise(); được gọi thì mọi xử lý của đối tượng \$.Deferred() trước đó được lưu lại cố định, tức là trạng thái resolved được ghi nhận bởi phương thức dfd.resolve() và trạng thái rejected được ghi nhận bởi dfd.reject() sẽ được lưu cố định lại. ngay sau đó thì ta không thể thay đổi trạng thái resolved hay rejected nữa. ta có thể hiểu rằng đối tượng \$.Deferred() chỉ có 1 trạng thái resolved hay rejected mà ta đã thiết lập cho chúng

Cách sử dụng tham số target để thúc đẩy **1 đối tượng có sẵn** trở thành **promise**

```
// Existing object
var obj = {
    hello: function( name ) {
        alert( "Hello " + name );
    }
},
// tạo ra 1 Deferred
```

```
defer = $.Deferred();
```

```
// thiết lập đối tượng có sẵn thành 1 promise
```

```
defer.promise( obj );
```

```
// Resolve the deferred
```

```
defer.resolve( "John" );
```

```
// sử dụng đối tượng object như là 1 Promise
```

```
obj.done(function( name ) {
```

```
    obj.hello( name ); // will alert "Hello John"
```

```
}).hello( "Karl" ); // will alert "Hello Karl"
```