You don't know JS (Tiếng Việt)—Khởi đầu - Chương 1: Hiểu về lập trình

Quá trình dịch sẽ không tránh sai sót, vì vậy các bạn nào có phát hiện ra thì xin phản hồi giúp tôi, không thì kệ nó luôn :))

Tôi biết ban đầu sẽ có những đoạn tôi dịch khá lủng củng, nhưng tôi hứa sẽ dịch càng ngày càng dễ đọc hơn.

Code

Một chương trình, thường được gọi là *source code* hoặc *code*, là một tập hợp các hướng dẫn để yêu cầu máy tính cần xử lý một nhiệm vụ nào đó. Thông thường, code được lưu dưới dạng file văn bản, với js bạn có thể gõ code trực tiếp lên console của trình duyệt.

Các nguyên tắc của một định dạng hợp lệ cùng sự kết hợp của các hướng dẫn được gọi là ngôn ngữ máy tính, hay còn gọi là

cú pháp, tương tự như ngôn ngữ giao tiếp chỉ cho bạn cách đọc từ và các tạo ra câu đúng bằng cách sử dụng từ và dấu câu.

Các câu lệnh (Statements)

Trong ngôn ngữ máy tính, một nhóm từ, số, cách thức thực thi từng nhiệm vụ cụ thể được gọi là câu lệnh. Trong JS, câu lệnh có thể như sau:

```
a = b * 2;
```

Ký tự `a` và `b` là các *biến (variable)*, giống như những cái hộp, bạn có thể lưu trữ đồ vật trong đó. Trong lập trình, biến giữ giá trị (ví dụ số `42`) được chương trình sử dụng.

Về mặt tương phản, số `2` bản thân nó chỉ là một giá trị, được gọi là giá trị ký tự (literal value), vì nó độc lập và không lưu trữ trong một biến nào cả.

Dấu `=` và `*` là *toán tử (operators)* (xem "Toán tử") -- nó thực thi các hành động với giá trị và biến như sự phân công và phép toán nhân.

Hầu hết câu lệnh trong JS kết thúc bằng dấu chấm phẩy (`;`) ở cuối câu.

Lệnh `a = b * 2;` báo cho máy tính giá trị được lưu trữ trong biến `b`, nhân giá trị đó với 2, sau đó lưu kết quả lại vào một biến khác gọi là `a`.

Lập trình tương tự như bộ sưu tập của nhiều câu lệnh cùng nhau mô tả tất cả các bước để thực thi mục đích lập trình.

Biểu thức (Expressions)

Các câu lệnh được tạo thành từ một hay nhiều *biểu thức*. Một biểu thức là bất kỳ tham chiếu trên một biến hoặc một giá trị, hoặc tập hợp các giá trị và các biến kết hợp thành toán tử.

Ví dụ:

```
a = b * 2;
```

Câu lệnh này có 4 biểu thức bên trong nó:

- `2` là *giá trị biểu thức trực kiện*.

- `b` là *giá trị biểu thức*, có nghĩa là sẽ lấy giá trị hiện tại của nó
- `b * 2` là *biểu thức toán học*, nghĩa là sẽ làm phép nhân
- `a = b * 2` là một *biểu thức gán*, nghĩa là sẽ gán kết quả của biểu thức `b * 2` cho biến `a` (còn tìm hiểu thêm sau)

Một biểu thức chung đứng một mình còn được gọi là *lệnh biểu thức*(expression statement), như ví dụ sau:

```
b * 2;
```

Kiểu biểu thức này thường không hữu dụng, bởi nó chẳng có tác dụng nào đối với chương trình đang chạy—nó sẽ lấy giá trị của `b` và nhân nó với `2`, nhưng sau đó không làm gì với kết quả đó.

Một loại câu lệnh biểu thức nữa là *biểu thức lệnh gọi* (call expression) (Xem “Functions”), khi toàn câu lệnh là một hàm tự gọi biểu thức:

```
alert( a );
```

Thực thi chương trình

Làm cách nào mà tập hợp các câu lệnh lập trình có thể yêu cầu máy tính phải làm gì? Chương trình cần được thực thi, hay còn được biết đến với tên *chạy chương trình*.

Các lệnh giống như `a = b * 2` rất hữu dụng cho lập trình viên đọc và viết, nhưng nó không hoàn toàn ở dạng để máy tính hiểu trực tiếp. Vì vậy, một trình tiện ích đặc biệt trong máy tính (hoặc là *thông dịch-interpreter* hoặc là *biên dịch-compiler*) được sử dụng để dịch code bạn viết thành các lệnh mà máy tính có thể hiểu.

Đối với một số ngôn ngữ máy tính, mỗi khi chương trình chạy thì bản dịch của các câu lệnh thường được hoàn thành từ trên xuống dưới, từng dòng một, nó thường được gọi là thông dịch mã.

Một số ngôn ngữ khác, bản dịch được hoàn thiện trước, gọi là biên dịch mã, rồi sau đó chương trình mới chạy, những gì đang chạy đã được biên dịch xong để cho máy sẵn sàng chạy.

JavaScript thường được khẳng định nó là ngôn ngữ được *thông dịch*, bởi vì mã nguồn JavaScript được xử lý mỗi lần chạy. Nhưng điều đó không hoàn toàn chính xác, thực ra là

JavaScript engine biên dịch chương trình và sau đó chạy ngay mã đã được biên dịch.

Ghi chú: thông tin thêm về biên dịch JavaScript, xem 2 chương đầu của Phạm vi & Đóng kín trong seri này.

Tự luyện

Chương này sẽ giới thiệu mỗi khái niệm lập trình với những mẫu code đơn giản, tất cả đều viết bằng JavaScript (đương nhiên!!!)

Bạn cần luyện tập mỗi khái niệm bằng cách tự gõ code. Các dễ nhất là mở developer tool console trên trình duyệt.

Mẹo: Thường thì bạn có thể mở developer console với phím tắt hoặc từ menu. Thông tin chi tiết về việc mở và sử dụng console trong trình duyệt ưa thích của bạn, xem "Mastering The Developer Tools Console". Để gõ nhiều dòng trên console cùng lúc, sử dụng `<shift> + <enter>` để chuyển sang dòng mới. Khi bạn nhấn `<enter>`, console sẽ chạy tất cả những gì bạn vừa viết.

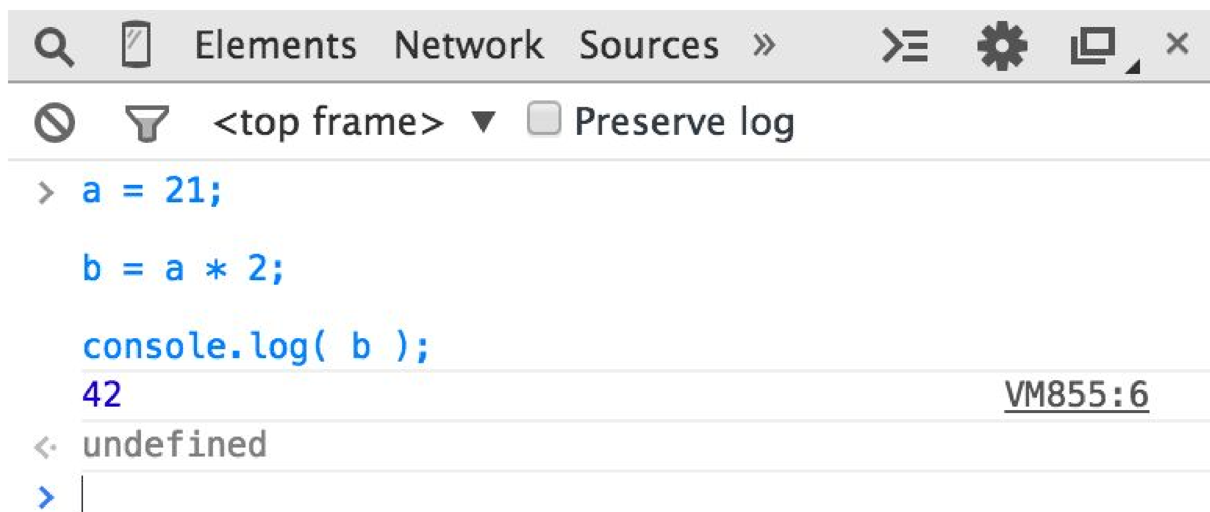
Hãy làm quen với việc chạy code trên console. Trước tiên, tôi đề nghị bạn mở 1 tab trống trên trình duyệt. Tôi thích làm cách

này bằng việc gõ `about:blank` trên thanh địa chỉ. Sau đó chỉ cần chắc chắn developer console được mở như chúng ta đã đề cập.

Và giờ hãy gõ code này và xem chúng chạy:

```
a = 21;
b = a * 2;
console.log( b );
```

Gõ những đoạn code trước trên Chrome console có thể tạo ra gì đó như:



Hãy thử đi, đây là cách học code hay nhất để bắt đầu học lập trình đó!

Output

Trong mẫu code ở trên, chúng ta sử dụng `console.log(..)`. Tóm lại, hãy nhìn vào những dòng code và xem nó là gì.

Bạn có thể đoán, nhưng chính xác làm sao chúng ta có thẻ print text (tức *ouput* cho người dùng) trên dev console. Có hai đặc tính mà chúng ta cần giải thích.

Đầu tiên, phần `log( b )` được đại diện như việc gọi hàm (xem "Functions"). Những gì xảy ra là chúng ta đang giao biến `b` cho hàm đó, hàm sẽ yêu cầu lấy giá trị của `b` và print nó lên console.

Thứ hai, phần `console.` làm một object tham chiếu nơi có hàm`log(...)`. Chúng ta sẽ xem object và các thuộc tính của nó chi tiết hơn ở Chương 2.

Cách khác để tạo ra output để bạn có thể nhìn nó chạy là `alert(...)`. Ví dụ:

```
alert( b );
```

Nếu bạn chạy nó, bạn sẽ thấy rằng thay vì hiển thị output lên console, nó sẽ hiển thị một popup “OK” với nội dung của biến `b`. Tuy nhiên, sử dụng `console.log(..)` giúp cho việc học của bạn

thuận tiện hơn nhiều so với `alert(..)`, bởi vì bạn có thể output nhiều giá trị cùng lúc mà không bị phiền hà bởi trình duyệt.

Với sách này thì chúng ta sẽ dùng `console.log(..)`

Input

Khi chúng ta thảo luận về output, bạn cũng sẽ thắc mắc về *input* (ví dụ như nhận thông tin từ người dùng)

Cách thông thường nhất là trang HTML hiển thị form cho người dùng có thể gõ vào, sau đó sử dụng js để đọc các giá trị vào các biến của chương trình.

Nhưng có một cách đơn giản để lấy input để học và trình bày như bạn học theo sách này, đó là sử dụng function `prompt(...)`

```
age = prompt( "Please tell me your age:" );
console.log( age );
```

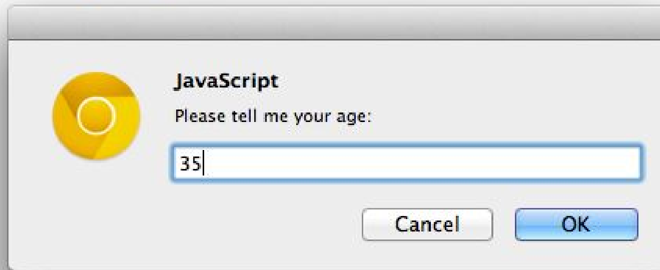
Như bạn có thể đoán, mẩu tin bạn đưa vào `prompt(..)` -- trong trường hợp này, `"Please tell me your age:"` -- được in lên popup.

Điều này cũng tương tự

Elements Network Sources »

<top frame> ▼ Preserve log

```
> age = prompt( "Please tell me your age:" );
  console.log( age );
>
```



Khi bạn nhập input text bằng cách nhấp “OK”, bạn sẽ tuân theo giá trị mà bạn đã gõ được lưu trong biến `age`, cái mà chúng ta *output* với `console.log(..)`:

Elements Network Sources »

<top frame> ▼ Preserve log

```
> age = prompt( "Please tell me your age:" );
  console.log( age );
  35                                          VM848:4
< undefined
>
```

Để giữ cho mọi thứ đơn giản trong quá trình chúng ta học lý thuyết lập trình cơ bản, các ví dụ trong sách này sẽ không cần input. Nhưng giờ bạn đã biết cách sử dụng `prompt(..)`, nếu bạn muốn thử thách bản thân, bạn có thể thử sử dụng input qua khám phá các ví dụ.

Các biểu thức

Toán tử là những gì chúng ta thực thi hành động trên các biến và các giá trị. Chúng ta đã thấy hai kiểu toán tử của JavaScript là `=` và `*`

Toán tử `*` để thực hiện phép toán nhân. Dễ heng.

Dấu `=` được sử dụng để *gán* -- đầu tiên là ta tính toán giá trị ở *phí bên phải*(giá trị nguồn) dấu `=` và sau nó đặt nó vào biến(variable) mà chúng ta xác định ở *phía bên trái* (variable mục tiêu).

Chú ý: Có thể cảm giác hơi bị ngược bằng cách đảo trình tự gán. Thay vì `a = 42`, một số người thích lật ngược trình tự giá trị nguồn ở bên trái và biến mục tiêu ở bên phải, kiểu như `42 -> a` (nó không phải JavaScript!). Không may,`a = 42`là định dạng

được sắp xếp, và tương tự các biến, nó khá phổ biến trong các ngôn ngữ lập trình hiện đại. Nếu có cảm giác không tự nhiên, hãy bỏ chút thời gian để làm quen và tâm trí bạn sẽ tự sắp xếp nó.

Hãy xem:

```
a = 2;

b = a + 1;
```

Tại đây, chúng ta gán giá trị `2` cho biến `a`. Sau đó chúng ta có giá trị của biến `a` (vẫn là `2`), thêm `1` vào nó sẽ có kết quả giá trị `3`, sau đó lưu giá trị đó vào biến `b`.

Trong khi biểu thức kỹ thuật, bạn cũng cần từ khoá `var` trong mọi thể loại lập trình, như cách đầu tiên bạn *khai báo*(aka *khởi tạo*) *các biến* (Xem "Variables")

Bạn có thể luôn khai báo biến bằng tên trước khi sử dụng nó. Nhưng bạn cũng chỉ khai báo một biến một lần trong mỗi *scope* (xem “Scope”): nó có thể được sử dụng nhiều lần khi cần. Ví dụ:

```
var a = 20;
```

```
a = a + 1;
a = a * 2;
console.log( a );    // 42
```

Sau đây là một số biểu thức trong JavaScript:

- Gán: `=` như `a = 2`.
- Toán: `+` (cộng), `-` (trừ), `*` (nhân), và `/` (chia), `a * 3`.
- Tổ hợp gán: `+=`, `-=`, `*=`, và `/=` là các biểu thức tổ hợp mà nó kết hợp giữa toán với các giá trị gán, ví dụ như `a += 2` (tương tự `a = a + 2`).
- Tăng/Giảm: `++` (tăng), `--` (giảm), như `a++` (tương tự `a = a + 1`).
- Tiếp cận object: `.` như `console.log()`. Object là các giá trị chứa các giá trị khác tại các vị trí có tên cụ thể gọi là thuộc tính. `obj.a` nghĩa là một giá trị object gọi `obj` với một đặc tính có tên `a`. Các thuộc tính có thể chuyển đổi cách tiếp cận ví dụ `obj["a"]`. Xem chương 2.
- Tương đương: `==` (bằng tương đối), `===` (bằng tuyệt đối), `!=` (khác tương đối), `!==` (khác tuyệt đối), ví dụ `a == b`. See "Values & Types" and Chapter 2.
- So sánh: `<` (nhỏ hơn), `>` (lớn hơn), `<=` (nhỏ hơn hoặc bằng tương đối), `>=` (lớn hơn hoặc bằng tương đối), như `a <= b`.

Xem “Values & Types” ở 2.

- Tính logic: `&&` (và), `||` (hoặc), ví dụ `a || b` chọn luôn `a` *hoặc* `b`.

Các biểu thức này được sử dụng để diễn tả các điều kiện phức tạp (xem “Điều kiện”), ví như cả `a` *hoặc* `b` đều đúng.

Ghi chú: Để biết chi tiết nhiều hơn, phạm vi của các biểu thức không đề cập ở đây, bạn có thể xem thêm Mozilla Developer Network (MDN)’s “Expressions and Operators” (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators).

Giá trị (Values) & loại (types)

Nếu bạn hỏi nhân viên tại một cửa hàng điện thoại giá của một cái điện thoại gì đó, và họ trả lời “chín chín, chín chín” (ví dụ $99.99), tức họ đã cho bạn một hình dung giá trị tiền mà bạn cần phải trả (bao gồm thuế) để mua nó. Nếu bạn muốn mua 2 cái điện thoại, bạn có thể dễ dàng làm phép toán để gấp đôi giá trị thành $199.98 từ giá gốc.

Nếu người nhân viên đó lấy hai cái điện thoại tương tự và nói “miễn phí”, tức là họ không cho bạn một con số nào, nhưng nó cũng là một dạng đại diện cho mức giá ($0.00)—cho từ “miễn phí”.

Sau đó bạn lại hỏi điện thoại có đồ sạc hay không, câu trả lời có thể là “có” hoặc “không”.

Bằng cách tương tự, khi bạn diễn tả những giá trị trong một chương trình, bạn chọn các kiểu đại diện khác nhau cho các giá trị đó dựa trên những gì bạn kế hoạch với chúng.

Những kiểu đại diện khác nhau cho các giá trị được gọi là *types* trong thuật ngữ lập trình. JavaScript có sẵn các type được gọi là giá trị *nguyên thuỷ* :

- Khi bạn muốn làm toán, bạn muốn `number`.
- Khi bạn muốn in giá trị trên màn hình, bạn cần `string` (một hay nhiều ký tự, từ, câu).
- Khi bạn muốn tạo một quyết định trên chương trình, bạn cần `boolean`(`true` hoặc `false`).

Các giá trị được thêm trực tiếp vào trong source code được gọi là *literals*. `string` literals được bao bằng dấu ngoặc kép `"..."` hoặc dấu ngoặc đơn (`'...'`) -- chỉ là phong cách khác nhau. `number` và `boolean` literals được đại diện như là ví dụ (`42`, `true`, etc.).

Hãy xem:

```
"I am a string";

'I am also a string';
42;
true;
false;
```

Ngoài kiểu giá trị `string`/`number`/`boolean`, các ngôn ngữ lập trình còn thường cung cấp kiểu *arrays*, *objects*, *functions*, và hơn nữa. Chúng ta sẽ tìm hiểu thêm về giá trị và kiểu suốt chương này và chương kế tiếp.

Chuyển đổi giữa các kiểu

Nếu bạn có một `number` nhưng muốn hiển thị trên màn hình, bạn cần chuyển giá trị thành `string`, và trong JavaScript gọi việc chuyển đổi này là "cưỡng bức". Tương tự, nếu ai đó cho

một loạt ký tự số vào trong form của trang thương mại điện tử, đó là `string`, nhưng nếu bạn muốn sử dụng giá trị để thực hiện phép toán, bạn cần *ép* nó thành `number`

JavaScript cung cấp một vài cơ sở khác nhau để ép buộc chuyển đổi các *kiểu*. Ví dụ:

```
var a = "42";


var b = Number( a );
console.log( a );   // "42"
console.log( b );   // 42
```

Sử dụng `Number(..)` (một function có sẵn) như đã thấy là một sự cưỡng bức *minh bạch* từ bất kỳ kiểu nào sang `number`. Nó hơi rõ ràng.

Nhưng có một chủ đề gây tranh cãi rằng những gì xảy ra khi bạn muốn so sánh hai giá trị của cùng một kiểu, nó cần một cưỡng bức *ngầm*.

Khi so sánh chuỗi `"99.99"` với số `99.99`, hầu hết mọi người đồng ý rằng nó bằng nhau. Nhưng nó không chính xác, đúng không? Nó có chung giá trị trong hai kiểu đại diện khác nhau, hai *kiểu* khác nhau. Bạn có thể nói là "bằng tương đối", đúng không?

Để giúp bạn thoát khỏi hoàn cảnh như vậy, JavaScript đôi khi đưa vào một sự cưỡng bức giá trị *ngầm* đến kiểu phù hợp.

Vì vậy nếu bạn sử dụng dấu bằng tương đối `==` để so sánh `"99.99" == 99.99`, JavaScript sẽ chuyển phía bên trái `"99.99"` thành `number` tương đương `99.99`. Sự so sánh trở thành `99.99 == 99.99`, nghĩa là đương nhiên `true`.

Khi được thiết kế để giúp bạn, cưỡng bức ngầm có thể mang đến sự hoang mang nếu bạn không có thời gian để học luật quản lý hành vi của nó. Hầu hết JS dev chưa bao giờ tìm hiểu, vì vậy cảm giác thường tình là sự cưỡng bức ngầm này gây hoang mang và tạo ra bug ngoài ý muốn, vậy nên phải tránh. Nó đôi khi được gọi là lỗ hổng trong thiết kế ngôn ngữ.

Tuy nhiên, cưỡng bức ngầm là một cơ chế *có thể học được*, và hơn thế nữa là *cần được học* bởi bất kỳ ai muốn học lập trình JavaScript một cách nghiêm túc. Nó không chỉ làm bạn hết hoang mang khi bạn học luật này, nó còn làm cho bạn lập trình tốt hơn. Hiệu quả xứng đáng.

Ghi chú: Để biết thêm về sự cưỡng bức, xem Chương 2 của tiêu đề này và Chương 4 của “Kiểu & Ngữ pháp”.

Bình luận trong code

Nhân viên cửa hàng điện thoại có thể liệt kê vài tính năng của loại điện thoại mới hoặc một số kế hoạch của công ty cô ta đưa ra. Những ghi chú đó chỉ dành cho nhân viên—nó không phải dành cho khách hàng. Tuy nhiên, những ghi chú đó đó giúp cô ta làm việc tốt hơn bằng cách tài liệu hoá những gì cô có thể nói với khách hàng.

Một trong những bài học quan trọng là bạn có thể học bằng cách viết code không chỉ cho máy tính, mà còn dành cho lập trình viên cũng như trình biên dịch.

Máy tính của bạn chỉ quan tâm đến mã máy, một chuỗi nhị phân 0s và 1s đến *biên dịch*. Có vô số chương trình mà bạn có thể viết và mang lại kết quả tương tự với chuỗi 0–1. Sự lựa chọn là bạn viết chương trình tế nào—không chỉ riêng bạn, mà còn là thành viên khác trong nhóm và kể cả tương lai của chính bạn.

Bạn có thể phấn đấu không chỉ viết chương trình cho nó chạy đúng, mà còn dễ dàng kiểm định. Bạn có thể mất nhiều thời gian cho việc chọn một cái tên tốt cho biến, cho hàm.

Nhưng phần quan trọng khác là code comment. Có một số văn bản trong chương trình của bạn được đưa vào thuần tuý là để giải thích cho người. Trình thông dịch/ biên dịch sẽ bỏ qua những comment đó.

Có rất nhiều quan điểm về việc điều gì làm nên một comment tốt; chúng ta thực sự không thể xác định một quy tắc bao trùm. Nhưng có một số nhận xét và hướng dẫn có ích:

- Code không có comment là chưa đủ tối ưu.
- Quá nhiều comment là dấu hiệu của code tồi.
- Comment nên giải thích *vì sao* thay vì *cái gì*. Nó có thể giải thích *vì sao*nếu có chút phức tạp.

Trong JavaScript, có hai kiểu comment khả dụng: là comment dòng đơn và comment đa dòng.

Nhận biết:

```
// Đây là comment dòng đơn
/* Đây là
     comment
            đa dòng
                     */
```

`//` Comment dòng đơn thích hợp với việc bạn đặt comment ngay trên một biểu thức hoặc cuối dòng. Những gì trên dòng có dấu `//` đều được coi như là comment (được bỏ qua bởi biên dịch) từ đầu cho đến cuối dòng đó. Không có bất kỳ cá biệt nào có thể xuất hiện trên comment dòn đơn.

Xem:

```
var a = 42;          // 42 là ý nghĩa cuộc đời
```

`/* .. */` Comment đa dòng thích hợp với việc nếu bạn có một vài dòng cần giải thích.

Đây là cách sử dụng comment đa dòng thông thường:

```
/* Giá trị sau đây được sử dụng bởi vì

     nó đã thể hiện rằng nó chính là câu

     trả lời mọi câu hỏi trong vũ trụ  */

var a = 42;
```

Nó có thể xuất hiện trong một dòng, thậm chí giữa dòng, bởi vì có `*/` là kết thúc. Ví dụ:

```
var a = /* arbitrary value */ 42;
console.log( a );   // 42
```

Thứ duy nhất không thể xuất hiện trong comment đa dòng là `*/`, bởi vì nó có thể tạo ra kết comment.

Bạn sẽ muốn bắt đầu học lập trình bằng cách bắt đầu bằng thói quen comment code. Trong suốt phần còn lại của chương này, bạn sẽ thấy bạn muốn dùng comment để giải thích các thứ, hãy làm tương tự với sự luyện tập của bạn. Tin tôi đi, mọi người đọc code của bạn sẽ cảm ơn bạn!

Biến

Hầu hết chương trình hữu ích đều muốn theo dõi một giá trị và sự thay đổi của chúng suốt chương trình, trải qua những biểu thức được gọi theo dự tính của chương trình.

Cách đơn giản nhất để thực hiện điều này là gán một giá trị cho một biểu tượng chứa nó, nó gọi là *biến*—bởi vì giá trị

trong vật chứa này có thể *biến đổi* trong suốt quá trình như mong muốn.

Vài ngôn ngữ lập trình, bạn khai báo một biến để chứa một giá trị có kiểu riêng biệt, ví dụ `number` hoặc `string`. `Static typing` hay còn gọi là *kiểu thực thi* thường được dẫn chứng là một lợi ích cho sự chính xác của chương trình bằng cách bảo vệ nó khỏi sự chuyển đổi giá trị không mong muốn.

Một số ngôn ngữ khác nhấn mạnh kiểu cho giá trị thay bì biến. *Weak typing*, hay còn gọi là *dynamic typing*, cho phép một biến có thể giữ bất kỳ kiểu giá trị tại bất kỳ thời điểm nào. Nó thường được dẫn chứng về lợi ích linh động của chương trình bằng cách cho phép một biến đại diện cho một giá trị cho dù nó là kiểu giá trị gì tại bất kỳ thời điểm nào theo chu trình logic của chương trình.

JavaScript sử dụng cách tiếp cận thứ hai, *dynamic typing*, nghĩa là biến có thể giữ bất kỳ giá trị của bất kỳ *kiểu* nào mà không bắt buộc kiểu thực thi.

Như đã giới thiệu, chúng ta khai báo một biến sử dụng biểu thức `var` -- chú ý rằng không có thông tin *kiểu* trong khai báo. Xem một chương trình đơn giản sau:

```
var amount = 99.99;
amount = amount * 2;
console.log( amount );        // 199.98
// chuyển `amount` sang một string, và thêm "$"
// ở đầu.
amount = "$" + String( amount );
console.log( amount );        // "$199.98"
```

Biến `amount` ban đầu giữ một number `99.99`, và sau đó giữ kết quả `number`kết quả của `amount * 2` tức là `199.98`.

Lệnh `console.log(..)` đầu tiên *ngầm* buộc giá trị `number` thành `string` để in ra.

Tiếp đó lệnh `amount = "$" + String(amount)` *rõ ràng* ép giá trị `199.98`thành một `string` và thêm ký tự `"$"` đằng trước. Tại đây, `amount` có `string` giá trị `"$199.98"`, vì thế `console.log(..)` thứ hai không cần phải cưỡng ép gì cả để in nó ra.

Lập trình viên JavaScript sẽ lưu ý tính linh hoạt của việc sử dụng biến `amount` cho các giá trị `99.99`. `199.98`, và `"$199.98"`. Những người thích static-typing sẽ thích tách biến kiểu như `amountStr` để giữ gía trị `"$199.98"`cuối cùng, bởi vì nó là một kiểu khác.

Dù thế nào, bạn cũng để ý rằng `amount` giữ một giá trị đang chạy mà thay đổi trong quá trình hoạt động của chương trình, minh họa cho mục đích chính của các biến là: quản lý *state* của chương trình.

Nói theo cách khác, *state* là theo dõi các thay đổi của giá trị khi chương trình hoạt động.

Một cách sử dụng phổ biến khác của các biến là tập trung hóa thiết lập giá trị. Thường được gọi là *constants*, khi bạn khai báo một biết với một giá trị xác định *không thay đổi* suốt chương trình.

Bạn khai báo *constants*, thường là ở đầu chương trình, tiện để bạn có một chỗ để thay đổi giá trịnh khi cần. Theo quy ước, các biến JS hằng số thường được viết hoa, hay có gạch dưới _ giữa các liên từ.

Dưới đây là ví dụ:

```
var TAX_RATE = 0.08;      // 8% sales tax
var amount = 99.99;
amount = amount * 2;
amount = amount + (amount * TAX_RATE);
console.log( amount );                    // 215.9784
console.log( amount.toFixed( 2 ) );      // "215.98"
```

Chú ý: Tương tự như làm thế nào`console.log(..)` là một function `log(..)`truy cập như một thuộc tính object trong giá trị `console`, `toFixed(..)` ở đây là một function có thể truy cập các giá trị `number`. Các `number`JavaScript không tự động định dạng cho dollars -- cơ chế không thể biết được ý định của bạn và không có kiểu tiền tệ. `toFixed(..)` cho phép chúng ta chỉ định làm tròn số thập phân `number` như mong muốn, và nó tạo ra `string` như ý.

Biến `TAX_RATE` là một *constant* theo quy ước -- không có gì đặc biệt trong chương trình ngăn cho nó không thay đổi. nhưng nếu thành phố tăng thuế bán lên 9%, chúng ta có thể dễ dàng cập nhật chương trình bằng cách cấu hình lại `TAX_RATE` bằng giá trị `0.09` cùng một nơi, thay vì phải tìm nhiều nhiều giá trị `0.08` rải rác khắp chương trình và cập nhật tất cả.

Phiên bản mới nhất của JS tại thời điểm biết bài này (thường gọi là "ES6") đã có cách mới để khai báo *constants*, bằng cách sửa dụng `const` thay cho `var`:

```
// as of ES6:

const TAX_RATE = 0.08;
```

```
var amount = 99.99;
// ..
```

Const rất hữu dụng giống như var với giá trị không đổi, khác biệt là nó ngăn ngừa sự thay đổi giá trị vô tình xảy ra ở đâu đó sau giá trị khởi tạo. Nếu bạn thử gán giá trị khác cho `TAX_RATE` sau lần khai báo đầu tiên, chương trình của bạn sẽ từ chối thay đổi (trong strict mode, sẽ thất bại với lỗi -- xem "Strict Mode" trong Chương 2).

Tuy nhiên, cách “bảo vệ” khỏi các sai sót này cũng tương tự như các ngôn ngữ static-typing, nên bạn có thể thấy tại sao static types trong ngôn ngữ khác hấp dẫn.

Lưu ý: Để có thêm thông tin về sự khác biệt của các biến có thể sử dụng trong chương trình, xem *Kiểu & Ngữ pháp*trong serie này.

Blocks

Nhân viên cửa hàng điện thoại phải đi qua tất cả các khâu để hoàn tất việc thanh toán khi bạn mua điện thoại.

Tương tự, trong code chúng ta thường nhóm các biểu thức với nhau, thường được gọi l2 *block*. Trong JavaScript, một block

được xác định bằng các bao một hoặc nhiều lệnh trong dấu ngoặc `{ .. }`. Ví dụ:

```
var amount = 99.99;
// a general block
{
     amount = amount * 2;
     console.log( amount );   // 199.98
}
```

Kiểu block `{ .. }` chung này hợp lệ, nhưng không thường thấy trong các chương trình JS. Thông thường, block được gắn liền với một lệnh điều khiển, chẳng hạn như biểu thức `if` (xem "Điều kiện") hoặc một vòng lặp (xem "Vòng lặp"). Ví dụ:

```
var amount = 99.99;
// is amount big enough?
if (amount > 10) {            // <-- block attached to `if`
     amount = amount * 2;
     console.log( amount );   // 199.98
}
```

Chúng ta sẽ giải thích `if` trong phần tiếp theo, nhưng như bạn thấy, `{ .. }`block có 2 câu lệnh được gắn với `if (amount > 10)`; câu lệnh bên trong block sẽ chỉ thực hiện khi điều kiện đáp ứng.

Chú ý: Không giống hầu hết các lệnh khác như `console.log(amount);`, một block lệnh không cần dấu chấm phẩy (`;`) để kết thúc.

Điều kiện

"Bạn có muốn thêm bảo vệ màn hình với chỉ $9.99 không?" Nhân viên hỗ trợ cửa hàng điện thoại đã tạo một quyết định cho bạn. Nhưng rõ ràng, đây chỉ là một câu hỏi "yes hoặc no".

Có nhiều cách để bạn diễn đạt *điều kiện* trong chương trình.

Loại thường thấy là lệnh `if`. Về cơ bản, khi bạn nói "*Nếu* điều kiện này đúng, hãy làm theo...". Ví dụ:

```
var bank_balance = 302.13;

var amount = 99.99;
if (amount < bank_balance) {
     console.log( "I want to buy this phone!" );
}
```

Lệnh `if` cần một diễn đạt trong dấu ngoặc đơn `()` được thể hiện như `true` hoặc `false`. Trong ví dụ, chúng ta có biểu thức

amount < bank_balance, nó sẽ xác định true hoặc false tùy thuộc vào giá trị của biến bank_balance.

Bạn cũng có thể cung cấp một sự thay thế điều kiện nếu nó không thỏa, gọi mệnh đề else:

```
const ACCESSORY_PRICE = 9.99;
var bank_balance = 302.13;
var amount = 99.99;
amount = amount * 2;
// can we afford the extra purchase?
if ( amount < bank_balance ) {
     console.log( "I'll take the accessory!" );
     amount = amount + ACCESSORY_PRICE;
}
// otherwise:
else {
     console.log( "No, thanks." );
}
```

Tại đây, nếu amount < bank_balance là true, chúng ta sẽ in ra "I'll take the accessory!" và thêm 9.99 vào biến amount. Nếu không, mệnh đề elsenói ra chúng ta sẽ trả lời lịch sự "No, thanks." và amount không thay đổi.

Như chúng ta đã thảo luận trong "Giá trị & Loại" ở trên, các giá trị không chuẩn bị sẵn cho kiểu mong muốn mà thường bị ép theo kiểu đó, lệnh ifmuốn kiểu boolean, nhưng nếu bạn chèn nó gì đó không phải boolean, sự cưỡng ép sẽ xảy ra.

JavaScript xác định một danh sách các giá trị đặc trưng được coi là “falsy” bởi vì khi bị ép là `boolean`, nó sẽ trở thành `false` -- nó bao gồm các giá trị như `0` và `""`. Bất kỳ giá trị nào không trong danh sách "falsy", sẽ tự động "truthy" -- khi bị ép sang `boolean` nó sẽ trở thành `true`. Các giá trị truthy bao gồm kiểu như `99.99` và `"free"`. Xem "Truthy & Falsy" trong Chương 2.

Các *Điều kiện* hiện hữu trong các dạng khác ngoài `if`. Ví dụ, lệnh `switch`có thể được sử dụng như như là viết tắt của một loạt lệnh `if..else` (xem Chương 2). Vòng lặp sử một *điều kiện* để diễn đạt nếu vòng lặp diễn ra tiếp hay dừng lại.

Chú ý: Thông tin sâu hơn về sự ép buộc có thể xảy ra ngầm trong các biểu thức kiểm tra của các *điều kiện*, xem Chương 4 đề mục *Kiểu & Ngữ pháp*của serie này.

Vòng lặp

Trong suốt quá trình bận rộn, có một danh sách khách hàng cần nói chuyện với nhân viên cửa hàng. Trong khi vẫn còn người trong danh sách đó, cô ta chỉ cần tiếp tục phục vụ khách hàng tiếp theo.

Lập lại một tập hợp hành động cho đến khi có một điều kiện nhất định thất bại—nói cách khác, chỉ lặp lại khi điều kiện thỏa mãn—là công việc của vòng lặp chương trình; vòng lặp có thể có nhiều hình thức khác nhau, nhưng tất cả đều đáp ứng hành vi cơ bản này.

Một vòng lặp bao gồm điều kiện kiểm tra cũng như một block (thường là `{ .. }`). Mỗi lần block vòng lặp được thực hiện, nó được gọi là *sự lặp lại*.

Ví dụ, vòng lặp `while` và dạng `do..while` minh họa khái niệm lặp lại một block câu lệnh cho đến khi điều kiện không còn được đánh giá là `true`:

```
while (numOfCustomers > 0) {


    console.log( "How may I help you?" );
    // help the customer...
    numOfCustomers = numOfCustomers - 1;
}
// versus:
do {
    console.log( "How may I help you?" );
    // help the customer...
    numOfCustomers = numOfCustomers - 1;
} while (numOfCustomers > 0);
```

Sự khác biệt thực tế duy nhất giữa các vòng lặp này là liệu điều kiện được kiểm tra trước lần lặp đầu tiên (`while`) hay sau lần lặp đầu tiên (`do..while`).

Ở một trong hai dạng, nếu điều kiện kiểm tra là `false`, lần lặp kế tiếp sẽ không chạy. Có nghĩa là điều kiện khởi tạo là `false`, vòng lặp `while` sẽ không bao giờ chạy, nhưng một vòng lặp `do..while` sẽ chạy lần đầu.

Đôi lúc bạn thực hiện lặp với mục đính là đếp một nhóm số, như là từ `0`đến `9` (mười số). Bạn có thể thực hiện bằng cách thiết lập một biến lặp `i`ở giá trị `0` và tăng lên `1` ở mỗi lần lặp.

Cảnh báo: Với vài lý do lịch sử, ngôn ngữ lập trình hầu hết luôn đếm mọi thứ từ không có gì, tức là `0` thay vì `1`. Nếu bạn chưa quen với cách tư duy đó, nó có thể hơi khó chịu lúc ban đầu. Dành thời gian để luyện việc đếm từ `0` cho đến khi cảm thấy thoải mái với nó.

Điều kiện được kiểm tra cho mỗi lần lặp, giống như nó có lệnh `if` bên trong vòng lặp.

Chúng ta có thể sử dụng lệnh `break` của JS để ngừng một vòng lặp. Đồng thời, chúng ta có thể thấy rằng thật dễ dàng một cách kinh khủng để tạo ra một vòng lặp chạy liên tục mà không có cơ chế `break`.

Hãy hình dung:

```
var i = 0;
// a `while..true` loop would run forever, right?
while (true) {
     // stop the loop?
     if ((i <= 9) === false) {
          break;
     }
     console.log( i );
     i = i + 1;
}
// 0 1 2 3 4 5 6 7 8 9
```

Cảnh báo: Đây không phải là hình thức thực tế mà bạn muốn sử dụng vòng lặp của mình. Nó chỉ được giới thiệu với mục đích minh họa mà thôi.

Trong khi `while` (hay `do..while`) có thể hoàn thành nhiệm vụ thủ công, có một dạng cú pháp khác được gọi là vòng lặp `for` cho dành mục đích đó.

```
for (var i = 0; i <= 9; i = i + 1) {
```

```
    console.log( i );
}
// 0 1 2 3 4 5 6 7 8 9
```

Như bạn thấy, trong cả hai trường hợp điều kiện `i <= 9` là `true` trong 10 lần lặp đầu tiên của một dạng vòng lặp (các giá trị `i` từ `0` đến `9`) trở thành `false` khi `i` đạt giá trị `10`.

Vòng lặp `for` có ba mệnh đề: mệnh đề khởi tạo (`var i=0`), mệnh đề kiểm tra điều kiện (`i <= 9`), và mệnh đề cập nhật (`i = i + 1`). Vì vậy nếu bạn định đếm với vòng lặp lặp đi lặp lại, `for` là hình thức gọn gàng và dễ hiểu hơn để hiểu và viết.

Có một dạng vòng lặp đặc biệt khác nhằm mục đích lặp các giá trị cụ thể, chẳng hạn như các thuộc tính của một đối tượng (xem Chương 2) trong đó kiểm tra điều kiện nghĩa là tất cả các thuộc tính đã được xử lý. Nguyên lý "lặp cho đến khi điều kiện sai" giữ nguyên bất kể dạng lặp nào.

Hàm (function)

Nhân viên cửa hàng điện thoại có thể không mang theo bàn tính để tính ra số thuế và số lượng mua cuối cùng. Đó là hạng mục cô ta cần xác định một lần và sử dụng nhiều lần. Ngược lại, công ty có phương tiện tính toán thanh toán (máy tính, tablet,...) với các chức năng có sẵn.

Tương tự, chương trình của bạn hầu hết cần tách code thành nhiều phần để sử dụng lại, thay vì lặp đi lặp lại bản thân. Cách làm điều này là xác định một `function`.

Một hàm thường là một phần code được đặt tên và có thể được gọi bằng tên, code bên trong nó sẽ chạy cho mỗi lần gọi.

```
function printAmount() {

     console.log( amount.toFixed( 2 ) );

}
var amount = 99.99;
printAmount(); // "99.99"
amount = amount * 2;
printAmount(); // "199.98"
```

Các hàm có thể tùy ý lấy đối số (aka tham số)—giá trị mà bạn truyền vào. Và chúng có thể tùy ý trả lại giá trị.

```
function printAmount(amt) {

     console.log( amt.toFixed( 2 ) );

}
function formatAmount() {
     return "$" + amount.toFixed( 2 );
}
var amount = 99.99;
printAmount( amount * 2 );          // "199.98"
amount = formatAmount();
console.log( amount );              // "$99.99"
```

Hàm `printAmount(..)` lấy một tham số gọi là `amt`. Hàm `formatAmount()` trả một giá trị. Dĩ nhiên, bạn có thể kết hợp hai kỹ thuật đó trong một hàm.

Các hàm thường được sử dụng với mục đích gọi ra nhiều lần, nhưng nó cũng có thể hữu ích trong việc tổ chức code, kể cả khi bạn chỉ gọi nó một lần.

Ví dụ:

```
const TAX_RATE = 0.08;
function calculateFinalPurchaseAmount(amt) {
     // calculate the new amount with the tax
     amt = amt + (amt * TAX_RATE);
     // return the new amount
     return amt;
}
```

```
var amount = 99.99;
amount = calculateFinalPurchaseAmount( amount );
console.log( amount.toFixed( 2 ) );          // "107.99"
```

Mặc dù `calculateFinalPurchaseAmount(..)` chỉ được gọi một lần, tổ chức hành vi của nó thành những function tách biệt làm cho code trở nên logic rõ ràng hơn (lệnh `amount = calculateFinal...`). Nếu function có nhiều lệnh bên trong nó, lợi ích thậm chí còn rõ ràng hơn.

Scope

Nếu bạn hỏi nhân viên cửa hàng về một mẫu điện thoại mà cô ấy không có, cô ta sẽ không thể bán chiếc điện thoại bạn muốn. Cô ta chỉ có thể bán những chiếc điện thoại có trong kho. Bạn sẽ phải thử ở cửa hiệu khác để tìm chiếc điện thoại bạn muốn.

Lập trình có một thuật ngữ cho khái niệm này: *scope* (kỹ thuật gọi là *lexical scope*). Trong JS, mỗi hàm đều có scope của nó. Scope cơ bản là một bộ tập hợp của các biến cũng như quy tắc cho các biến đó được gọi theo tên. Chỉ có code trong hàm mới có thể tiếp cận được với các biến trong *scope* của hàm đó.

Tên biến bên trong cùng scope phải là duy nhất—nó không thể có hai biến `a` khác nhau tồn tại kế bên. Nhưng biến `a` trùng nhau có thể tồn tại trong các scope khác nhau.

```
function one() {

     // this `a` only belongs to the `one()` function

     var a = 1;

     console.log( a );

}
function two() {
     // this `a` only belongs to the `two()` function
     var a = 2;
     console.log( a );
}
one();          // 1
two();          // 2
```

Ngoài ra, một scope có thể lồng bên trong scope khác, giống như chú hề trong tiệc sinh nhật thổi quả bóng lồng trong quả bóng khác. Nếu một scope được lồng trong scope khác, code bên trong scope sâu nhất có thể tiếp cận với mọi biến ở các phạm vi.

Xem:

```
function outer() {

      var a = 1;
      function inner() {
            var b = 2;
            // we can access both `a` and `b` here
            console.log( a + b );     // 3
      }
      inner();
      // we can only access `a` here
      console.log( a );               // 1
}
outer();
```

Các nguyên tắc của lexical scope cho phép code có thể truy cập các biến của phạm vi bên trong hay bên ngoài scope.

Do đó, code bên trong hàm `inner()` có thể truy cập cả hai biến `a` và `b`, nhưng code trong `outer()` chỉ có thể truy cập `a` -- nó không thể truy cập `b`bởi vì biến đó bên trong `inner()`

Nhắc lại đoạn mã trên:

```
const TAX_RATE = 0.08;
function calculateFinalPurchaseAmount(amt) {
      // calculate the new amount with the tax
      amt = amt + (amt * TAX_RATE);
      // return the new amount
```

```
    return amt;
}
```

Hằng (biến) `TAX_RATE` có thể truy cập từ bên trong function `calculateFinalPurchaseAmount(..)`, vì là lexical scope ta không cần gán nó vào trong.

Ghi chú: Nội dung về lexical scope, xem 3 chương đầu của *Scope & Closures*.

Luyện tập

Không có sự thay thế nào bằng việc thực hành trong lập trình. Cũng không có nội dung rõ ràng nào trong bài viết của tôi biến bạn trở thành lập trình viên.

Bạn hãy ghi nhớ, hãy luyên tập các nguyên lý mà chúng ta đã học trong chương này. Tôi sẽ cho bạn vài “yêu cầu” và bạn thử trước. Sau đó tìm hiểu danh sách code dưới đây để biết cách mà tôi đã tiếp cận chúng.

- Viết một chương trình để tính toán tổng giá điện thoại bạn đã mua. Bạn sẽ tiếp tục mua điện thoại (gợi ý: vòng lặp!) cho đến khi bạn hết tiền trong tài khoản.

Bạn đồng thời cũng sẽ mua phụ kiện cho mỗi cái điện thoại miễn là số tiền dưới ngưỡng chi tiêu.

- Sau khi bạn tính toán tổng tiền, thêm thuế, sau đó in ra tổng cuối cùng với định dạng hoàn chỉnh.
- Cuối cùng, kiểm tra số tiền đối với số dư trong tài khoản ngân hàng để xem bạn có đủ khả năng hay không.
- Bạn nên lập một vài hằng số cho “tax rate,” “phone price,” “accessory price,” and “spending threshold,” cũng như biến cho “bank account balance.””
- Bạn nên xác định các hàm cho việc tính toán thuế và định dạng giá tiền hoàn chỉnh với “$” và làm trong thành hai số thập phân.
- Thách thức tăng cường: Thử kết hợp đầu vào chương trình, có thể là `prompt(..)` được đề cập trong "Input" ở trên. Bạn có thể nhắc người dùng về số dư trong tài khoản, ví dụ vậy. Chúc vui và sáng tạo.

OK, triển thôi. Đừng nhìn code của tôi cho đến khi bạn có gì đó!

Ghi chú: Bởi đây là cuốn sách JavaScript, Tôi sẽ giải bài tập bằng JavaScript. Nhưng bạn có thể thực hiện nó với ngôn ngữ khác nếu bạn cảm thấy thoải mái hơn.

Đây là giải pháp của tôi:

```
const SPENDING_THRESHOLD = 200;


const TAX_RATE = 0.08;


const PHONE_PRICE = 99.99;


const ACCESSORY_PRICE = 9.99;
var bank_balance = 303.91;
var amount = 0;
function calculateTax(amount) {
     return amount * TAX_RATE;
}
function formatAmount(amount) {
     return "$" + amount.toFixed( 2 );
}
// còn tiền thì mua tiếp
while (amount < bank_balance) {
     // mua điện thoại mới!
     amount = amount + PHONE_PRICE;
     // chúng ta có thể mua phụ kiện được không?
     if (amount < SPENDING_THRESHOLD) {
          amount = amount + ACCESSORY_PRICE;
     }
}
// đừng quên trả tiền cho chính phủ
amount = amount + calculateTax( amount );
console.log(
```

```
    "Your purchase: " + formatAmount( amount )
);
// Bạn dùng hết: $334.76
// bạn trả tiền nổi không?
if (amount > bank_balance) {
    console.log(
        "You can't afford this purchase. :("
    );
}
// Bạn không trả nổi. :(
```

Ghi chú: Cách đơn giản nhất để chạy chương trình JavaScript là gõ trên console trình duyệt mà bạn quen thuộc.

Bạn đã làm như thế nào? Cũng không khổ sở gì để thử lại nếu bạn đã thấy code của tôi. Có thể chơi một chút bằng cách thay đổi vài hằng số và xem chương trình nó chạy với những giá trị khác nhau.

Ôn lại

Học lập trình không cần thiết phải theo tiến trình phức tạp và nặng nề. Nó chỉ có vài nguyên lý cơ bản bạn cần ghi nhớ trong đầu.

Việc học cũng như xây nhà. Để tạo ra một tòa tháp cao, bạn phải bắt đầu đặt những viên gạch này lên trên những viên gạch khác. Đây là vài chương trình cần thiết để xây tháp:

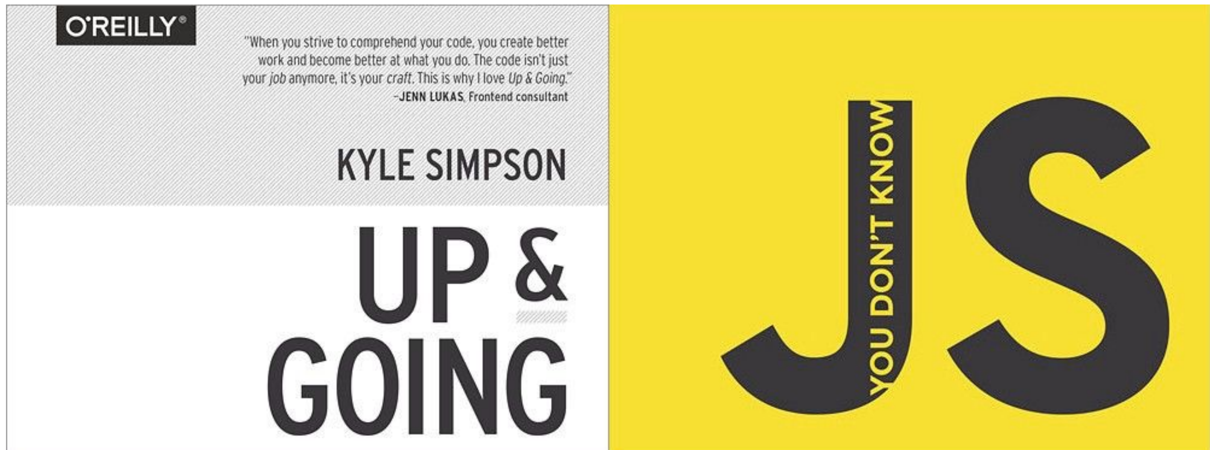
- Bạn cần *toán tử* để thực hiện hành động trên các giá trị.
- Bạn cần các giá trị và *kiểu* để thực hiện vài kiểu tác vụ khác nhau như toán với `number` hoặc xuất đầu ra với `string`.
- Bạn cần *biến* để lưu trữ dữ liệu (cũng như *state*) trong suốt quá trình thực thi chương trình.
- Bạn cần *điều kiện* như câu lệnh `if` để ra quyết định.
- Bạn cần *vòng lặp* để lặp lại tác vụ cho đến khi điều kiện ngừng xác thực.
- Bạn cần *hàm* để tổ chức code của bạn thành những thành phần logic và tái sử dụng.

Bình luận code là một trong những các hiệu quả để giúp cho code dễ đọc, làm cho chương trình của bạn dể hiểu, dễ bảo trì và sửa chữa sau đó nếu có vấn đề.

Cuối cùng, đừng bỏ qua sức mạnh của thực hành. Cách tốt nhất để học lập trình là lập trình.

You Don’t Know JS (tiếng Việt)—Khởi

đầu—Chương 2: Hiểu về JavaScript



Trong chương trước, tôi giới thiệu những cụm căn bản của lập trình như là biến, vòng lặp, điều kiện, hàm. Tất nhiên, tất cả các code đều trình diễn theo ngôn ngữ JS. Nhưng trong chương này, tôi muốn tập trung đặc biệt vào những điều cần biết về JS để bạn khởi đầu trở thành một lập trình viên JS.

Chúng ta sẽ giới thiệu một vài nguyên lý trong chương này một cách không đầy đủ cho đến những phần tiếp theo bộ sách *YDKJS*. Bạn có thể nghĩ chương này là một mục khái quát các chuyên đề trong series.

Đặc biệt nếu JS còn mới mẻ đối với bạn, bạn cần bỏ ít thời gian để xem lại các lý thuyết và ví dụ nhiều lần. Bất kỳ một nền tảng tốt nào cũng phải đặt từng viên gạch một, cho nên đừng hy vọng bạn ngay lập tức hiểu liền trong lần đầu.

Cuộc phiêu lưu của bạn để học sâu về JS bắt đầu từ đây.

Ghi chú: Như tôi đề cập ở Chương 1, bạn phải tự mình thử tất cả các code khi bạn học trong suốt chương này. Chú ý rằng một vài chỗ phù hợp với phiên bản mới nhất của JS tại thời điểm viết bài này (thường gọi là "ES6" cho phiên bản 6 của ECMAScript—tên chính thức của JS). Nếu bạn định sử dụng phiên bản trình duyệt cũ hơn trước thời ES6, code có thể không hoạt động. Bạn cần nâng cấp trình duyệt.

Giá trị & Kiểu

Như chúng ta đã khẳng định trong Chương 1, JS có kiểu của giá trị, không có kiểu của biến. Các kiểu có sẵn là:

- `string`
- `number`
- `boolean`

- `null` and `undefined`
- `object`
- `symbol` (new to ES6)

JavaScript có một biểu thức `typeof` có thể kiểm tra giá trị và cho bạn biết kiểu của nó là gì.

```
var a;

typeof a;                // "undefined"
a = "hello world";
typeof a;                // "string"
a = 42;
typeof a;                // "number"
a = true;
typeof a;                // "boolean"
a = null;
typeof a;                // "object" -- weird, bug
a = undefined;
typeof a;                // "undefined"
a = { b: "c" };
typeof a;                // "object"
```

Giá trị trả lại từ biểu thức `typeof` luôn là một trong sáu kiểu ở dạng giá trị string (ES6 là 7 -- thêm kiểu "symbol"). Đó là, `typeof "abc"` trả lại `"string"`, không phải `string`.

Chú ý rằng đoạn code này, biến `a` giữ mọi kiểu của giá trị, và mặc dù là có bề ngoài, `typeof a` sẽ không hỏi "kiểu của `a`", mà là

"kiểu của giá trị hiện tại trong `a`". Trong JS chỉ có giá trị mới có kiểu; biến chỉ đơn giản là vật chứa các giá trị đó.

`typeof null` là một trường hợp thú vị, bởi vì nó trả sai thành `"object"`, trong khi bạn mong muốn nó trả `"null"`.

Cảnh báo: Đây là một lỗi trường kỳ của JS và dường như sẽ không bao giờ được sửa. Quá nhiều code trên web liên quan đến lỗi mà cố sửa nó thì nó sẽ tạo ra nhiều lỗi hơn.

Đồng thời, chú ý `a = undefined`. Chúng ta thiết lập rõ ràng `a` là giá trị `undefined`, nhưng nó cũng có hành vi không khác với biến chưa có giá trị, ví dụ như `var a;` ở đoạn code bên trên. Một biến có thể nhận trạng thái "undefined" này bằng nhiều cách khác nhau, bao gồm hàm trả lại không có giá trị và cách sử dụng `void`.

Objects

Kiểu `object` đề cập đến một giá trị phức hợp mà bạn có thể lập các thuộc tính, mỗi cái đều có thể có giá trị của riêng chúng với bất kỳ kiểu nào. Đây có lẽ là một trong nhưng kiểu hữu dụng nhất trong JS.

```
var obj = {

     a: "hello world",

     b: 42,

     c: true

};
obj.a;          // "hello world"
obj.b;          // 42
obj.c;          // true
obj["a"]; // "hello world"
obj["b"]; // 42
obj["c"]; // true
```

Tốt hơn là xem giá trị của `obj` một cách trực quan:

obj

a: "hello world"	b: 42	c: true

Thuộc tính có thể truy cập bằng *dấu chấm* (vd: `obj.a`) hay *dấu ngoặc* (vd: `obj["a"]`). Dấu chấm ngắn hơn và dễ đọc hơn, vì vậy nó được ưa thích hơn.

Dấu ngoặc hữu dụng nếu bạn có một tên thuộc tính có ký tự đặc biệt trong nó, `obj["hello world!"]` -- các thuộc tính như vậy thường được là *chìa khóa* khi truy cập thông qua dấu ngoặc. Dấu ngoặc `[ ]` đòi hỏi phải có một biến (giải thích sau) hoặc một `string` *nguyên bản* (được bao bởi `" .. "`hoặc `' .. '`).

Đương nhiên, dấu ngoặc cũng hữu dụng nếu bạn muốn tiếp cận một thuộc tính/chìa khóa nhưng tên được lưu trữ ở biến khác, như là:

```
var obj = {

     a: "hello world",

     b: 42

};
var b = "a";
obj[b];             // "hello world"
obj["b"];      // 42
```

Ghi chú: Xem thêm tập *this & Object Prototypes* đặc biệt ở Chương 3 để biết thêm về `object`.

Có một vài kiểu giá trị mà bạn có thể tương tác với chương trình JS: *array*và *function*. Nhưng thay vì là các tích hợp phù hợp, nó phải được coi như là các kiểu thứ cấp—là phiên bản đặc biệt của kiểu `object`.

Mảng (array)

Trong mảng là `object` giữ các giá trị (của bất kỳ kiểu nào) có vị trí theo chỉ số, chứ không phải theo trong một khóa/ thuộc tính được đặt tên. Ví dụ:

```
var arr = [

     "hello world",

     42,

     true

];
arr[0];              // "hello world"
arr[1];              // 42
arr[2];              // true
arr.length;          // 3
typeof arr;          // "object"
```

Ghi chú: Các ngôn ngữ thường được đếm từ 0, JS cũng dùng `0` như là chỉ mục của giá trị đầu tiên trong mảng.

Để dễ hình dung về `arr` hãy xem hình dưới:

arr

0: "hello world"	1: 42	2: true

Bởi vì mảng là một object đặc biệt (như `typeof` đã ngụ ý), nó có thể có thuộc tính, bao gồm thuộc tính `length` cũng được tự động cập nhật.

Về mặt lý thuyết, bạn có thể sử dụng mảng như một object bình thường, hoặc bạn có thể sử dụng một `object` nhưng chỉ cho các thuộc tính số (`0`, `1`, ...) tương tự như một mảng. Tuy nhiên, điều này thường được coi là sử dụng không đúng loại tương ứng.

Cách tự nhiên và tốt nhất là sử dụng mảng cho các giá trị được xác định vị trí theo số và sử dụng `object` cho các thuộc tính có tên.

Hàm

Một kiểu `object` con khác mà bạn sẽ sử dụng trong suốt chương trình JS là hàm:

```
function foo() {

     return 42;

}
foo.bar = "hello world";
typeof foo;              // "function"
typeof foo();       // "number"
typeof foo.bar;          // "string"
```

Một lần nữa, hàm là kiểu con của `objects` -- `typeof` trả kết quả `"function"` để hàm ý rằng `function` là một kiểu chính -- và có thể có các thuộc tính, nhưng bạn chỉ thường sử dụng thuộc tính đối tượng hàm trong trường hợp giới hạn (ví dụ `foo.bar`).

Chú ý: Thông tin chi tiết về giá trị và kiểu của JS, xem hai chương đầu của phần *Kiểu & Ngữ pháp*.

Các phương thức kiểu dựng sẵn

Những kiêu dựng sẵn và kiểu con mà chúng ta vừa bàn luận có hành vi rõ ràng như các thuộc tính và phương thức của nó rất mạnh mẽ, hữu dụng.

Ví dụ:

```
var a = "hello world";


var b = 3.14159;
a.length;                   // 11
a.toUpperCase();            // "HELLO WORLD"
b.toFixed(4);               // "3.1416"
```

Việc “làm cách nào” đằng sau lệnh `a.toUpperCase()` phức tạp hơn chỉ là phương thức hiện tại.

Tóm tắt thì nó là một dạng bao lấy object `String` (viết hoa `S`), thường được gọi là "native", cặp với `string` nguyên thủy; đó là object bao ngoài định nghĩa phương thức `toUpperCase()` trên nguyên mẫu của nó.

Khi bạn sử dụng giá trị ban đầu `"hello world"` như một `object` bằng cách tham chiếu một thuộc tính hay phương thức (vd: `a.toUpperCase()` trong đoạn code trên), JS tự động "đóng hộp" giá trị cho đối tượng bao ngoài của nó.

Một giá trị `string` có thể được bao bởi một `String` object, một `number` có thể được bao bởi `Number` object, và `boolean` được bao bởi `Boolean` object. Phần lớn, bạn không cần phải lo lắng hoặc trực tiếp sử dụng những dạng bao object của giá trị (trong thực tế bạn sử dụng định dạng giá trị nguyên thủy, JS sẽ lo phần còn lại).

Ghi chú: Để biết thêm về các bản chất của JS và "đóng hộp", xem Chương 3 *Kiểu & Ngữ pháp*. Để hiểu hơn về prototype và object, xem Chương 5 *this & Object Proptotypes*

So sánh các giá trị.

Có hai kiểu so sánh giá trị chính mà bạn cần thực hiện trong chương trình JS: *bằng nhau* và không *bằng nhau*. Kết quả của bất kỳ sự so sánh nào đều là `boolean` (`true` hoặc `false`), bất kể kiểu già trị nào được so sánh.

Sự ép buộc

Chúng ta đã nói ngắn gọn về sự ép buộc ở Chương 1, nhưng chúng ta sẽ thảo luận thêm ở đây.

Sự ép buộc xuất hiện trong 2 dạng của JavaScript: *minh bạch* và *tiềm ẩn*. Sự ép buộc minh bạch đơn giản là bạn thấy rõ ràng trong code có một sự chuyển đổi từ dạng này sang dạng khác, trong khi đó sự ép buộc tiềm ẩn là khi chuyển đổi dạng có thể xảy ra nhiều hơn, là một hiệu ứng phụ của một số hoạt động khác.

Bạn có thể từng nghe ý kiến rằng "sự ép buộc là ma quỷ" được nêu ra trong một số sự kiện tại một vị trí rõ ràng nào đó, nơi mà sự ép buộc có thể tạo nên những kết quả bỡ ngỡ. Có lẽ đối với các developer thì không có gì thất vọng hơn khi một ngôn ngữ làm họ bỡ ngỡ.

Sự ép buộc không phải là ma quỷ hay sự bất ngờ. Thực tế, phần lớn các trường hợp chính bạn có thể xây dựng với kiểu ép buộc hợp lý và dễ hiểu, thậm chí còn có thể sử dụng để *cải thiện* khả năng đọc code. Nhưng chúng ta không đi sâu vào tranh luận điều này.—Chương 4 *Kiểu & Ngữ pháp* sẽ giải quyết hết các vấn đề.

Dưới đây là ví dụ của ép buộc *minh bạch*::

```
var a = "42";
var b = Number( a );
```

```
a;                    // "42"
b;                    // 42 -- số!
```

Và dưới đây là ví dụ của ép buộc *không minh bạch*:

```
var a = "42";
var b = a * 1; // "42" ép buộc ngầm 42 tại đây
a;                    // "42"
b;                    // 42 -- số!
```

Đúng và Sai

Trong chương 1, chúng ta đã đề cập ngắn gọn đến tính "truthy" và "falsy" tự nhiên của giá trị: khi một giá trị không phải `boolean` bị ép thành `boolean`, nó có trở thành tính `true` hay `false` tương ứng?

Danh sách cụ thể của giá trị "falsy" trong JS:

- `""` (chuỗi rỗng)
- `0`, `-0`, `NaN` (`number` không hợp lệ)
- `null`, `undefined`
- `false`

Bất kỳ giá trị nào không phải "falsy" thì là "truthy". Đây là ví dụ:

- `"hello"`

- `42`
- `true`
- `[ ]`, `[ 1, "2", 3 ]` (arrays)
- `{ }`, `{ a: 42 }` (objects)
- `function foo() { .. }` (functions)

Điều quan trọng là phải nhớ rằng giá trị `boolean` chỉ theo sự cưỡng ép "truthy"/"falsy" nếu nó bị ép theo `boolean`. Cũng không phải khó khăn gì để tự nhầm lẫn với một tình huống có vẻ như nó đang buộc một giá trị thành `boolean`khi nó không phải.

Đẳng thức

Có bốn loại đẳng thức: `==`, `===`, `!=`, và `!==`. Dạng `!` tất nhiên là bản "không bình đẳng" đối xứng với các đối chiếu của nó; không nên nhầm lẫn giữa *không bằng tuyệt đối* và *không bằng tương đối*.

Sự khác biệt giữa `==` và `===` thường được đặc trưng hóa rằng `==` kiểm tra bằng nhau của giá trị và `===` kiểm tra bằng nhau cả giá trị lẫn kiểu. Tuy nhiên, điều này không đúng. Cách thích hợp để đặc trưng hóa chúng là `==`kiểm tra bằng nhau của giá trị với việc ép buộc được cho phép, và `===`không cho phép cưỡng bức;

`===` thường được gọi là "bình đẳng nghiêm ngặt" (strict equality) vì lý do này.

Xem sự ép buộc ngầm cho phép bằng cách so sánh bình đẳng `==` và không được phép với sự ép buộc nghiêm ngặt `===`:

```
var a = "42";


var b = 42;
a == b;             // true
a === b;        // false
```

Trong việc so sánh `a == b`, JS nhận thấy rằng kiểu không trùng khớp, nên nó phải qua một loạt các bước theo trình tự ép buộc một hoặc cả hai giá trị khác kiểu cho đến khi chúng phù hợp, tại điểm này một phép so sánh giá trị đơn giản có thể được kiểm tra.

Nếu bạn nghĩ, có hai cách để `a == b` có thể `true` thông qua ép buộc. Hoặc nó được so sánh là `42 == 42` hoặc nó là `"42" == "42"`, thì nó là cái nào?

Câu trả lời: `"42"` trở thành `42`, để so sánh `42 == 42`. Trong ví dụ đơn giản, nó cũng không thực sự quan trọng tiến trình nào xảy ra, vì kết quả cũng như nhau. Nhưng đối với trường hợp phức

tạp hơn thì nó quan trọng bởi vì nó không chỉ là kết quả của việc so sánh, mà là *làm thế nào* để dẫn tới điều này.

`a === b` tạo nên `false` bởi vì sự ép buộc không cho phép, vì vậy sự so sánh đơn giản của giá trị đương nhiên sai. Nhiều lập trình viên cảm thấy `===`có thể dự đoán được, vậy nên họ hay dùng mẫu này và tránh xa `==`. Tôi nghĩ cách nhìn này khá ngắn. Tôi tin `==` là một công cụ quan trong để giúp chúng ta lập trình, *nếu bạn bỏ thời gian học cách nó hoạt động thế nào*

Chúng ta sẽ không đi chi tiết hết nền tảng về phương thức so sánh `==` ép buộc như thế nào. Hầu hết khá là hợp lý, nhưng cũng có một vài trường hợp góc cạnh quan trọng cũng nên cẩn thận. Bạn có thể đọc phần 11.9.3 của đặc tính ES5 (http://www.ecma-international.org/ecma-262/5.1/) để biết luật chính xác, và bạn sẽ ngạc nhiên cơ chế này đơn giản thế nào khi so sánh với những cường điệu trái ngược xung quanh nó.

Để làm rõ toàn bộ những chi tiết của vài điều cần ghi nhớ đơn giản, giúp bạn biết được khi nào thì dùng `==` hay `===`, tôi liệt kê một số nguyên tắc sau:

- Nếu một trong hai bên trong phép so sánh là có thể là giá trị `true` hoặc `false`, tránh dùng `==` mà dùng `===`.
- Nếu một trong hai bên trong phép so sánh là có thể một giá trị cụ thể (`0`, `""`, hoặc `[]` -- array rỗng), tránh `==`mà dùng `===`.
- Trong *tất cả* các trường hợp khác, bạn yên tâm dùng `==`. Không chỉ vì nó an toàn, đơn giản là giúp code của bạn dễ đọc hơn trong nhiều trường hợp.

Những nguyên tắc trên là để yêu cầu bạn suy nghĩ nghiêm túc về code của bạn và về những loại giá trị nào có thể đi qua các biến để được so sánh bằng. Nếu bạn chắc chắn về giá trị, và `==` an toàn, hãy dùng nó! Nếu bạn không thể chắc chắn về giá trị, sử dụng `===`. Đơn giản vậy thôi.

Ký hiệu không bằng `!=` đi cặp với `==`, và `!==` đi với `===`. Tất cả các quy tắc và quan sát chúng ta vừa thảo luận đều cũng được áp dụng với phép so sánh không bằng này.

Bạn nên chú ý đặc biệt về các so sánh `==` và `===` khi so sánh hai giá trị không phải là nguyên thủy, như là `object` (bao gồm `function` và `array`). Bởi vì các giá trị đó là tham chiếu, cả `==` và

=== đều kiểm tra khi nào các tham chiếu phù hợp chứ không phải các giá trị bên trong.

Ví dụ, array được mặc định ép buộc sang string bằng cách đơn giản gộp tất cả các giá trị với dấu phẩy (,) ở giữa. Bạn có thể nghĩ là hai array với nội dung như nhau có thể là ==, nhưng không phải:

```
var a = [1,2,3];

var b = [1,2,3];

var c = "1,2,3";
a == c;        // true
b == c;        // true
a == b;        // false
```

Chý ý: Để biết thêm về quy tắc so sánh ==, xem ES5 specification (section 11.9.3) và cũng có bàn bạc trong Chương 4 của *Kiểu & Ngữ pháp* trong bộ này; xem Chương 2 để biết thêm về giá trị với tham chiếu.

Bất bình đẳng

Toán tử <, >, <=, và >= sử dụng cho bất bình đẳng, được đề cập trong đặc là như là "so sánh quan hệ". Thông thường chúng

được sử dụng với các giá trị so sánh theo thứ tự như `number`. Như `3 < 4` cũng dễ hiểu.

Nhưng giá trị JavaScript `string` cũng có thể được so sánh bất bình đẳng, sử dụng nguyên tắc alphabe thông thường (`"bar" < "foo"`).

Về sự ép buộc thì sao? Tương tự như nguyên tắc so sánh `==` (mặc dù không giống hệt vậy) áp dụng trong toán tử bất bình đẳng. Đáng chú ý, không có toán tử "bất bình đẳng nghiêm ngặt" không cho phép ép buộc như `===` như trong "bình đẳng nghiêm ngặt".

Ví dụ:

```
var a = 41;

var b = "42";

var c = "43";
a < b;          // true
b < c;          // true
```

Chuyện gì đang xảy ra? ở phần 11.8.5 của đặc tính kỹ thuật ES5, nó nói rằng nếu cả hai giá trị trong so sánh `<` đều là `string`, như ví dụ `b < c` trên, việc so sánh được thực hiện bằng từ điển

học (wtf??). Nhưng nếu một trong hai giá trị không phải là `string`, như so sánh `a < b`, thì cả hai giá trị bị ép thành `number`, và phép so sánh số thông thường diễn ra.

Hãy nhớ rằng là không có “bất bình đẳng nghiêm ngặt” để sử dụng, cái hay nhất là bạn có thể sử dụng phép so sánh với nhiều kiểu giá trị khác nhau khi một trong các giá trị không phải là một giá trị số hợp lệ, ví dụ:

```
var a = 42;


var b = "foo";
a < b;          // false
a > b;          // false
a == b;         // false
```

Khoan, cách nào mà tất cả ba phép so sánh trên có thể `false`? Bởi vì giá trị `b` bị ép buộc thành "một giá trị number không hợp lệ" `NaN` trong `<` và `>`, và đặc điểm kỹ thuật cho biết `NaN` có thể là lớn hoặc nhỏ hơn bất kỳ một giá trị khác.

Phép so sánh `==` thất bại vì một lý do khác. `a == b` có thể thất bại kể cả được thông dịch ở cả hai cách `42 == NaN`hay `"42" == "foo"` -- như chúng ta đã giải thích lúc trước.

Chú ý: Để biết thêm thông tin về luật của phép so sánh bất bình đẳng, xem phần 11.8.5 của đặc tính kỹ thuật ES5, đồng thời tham khảo đề mục *Kiểu & Ngữ pháp* Chương 4 của bộ sách này.

Biến

Trong JavaScript, tên biến (bao gồm tên hàm) phải là *nhận diện* hợp lệ. Sự nghiêm ngặt và hoàn chỉnh của các nguyên tắc cho các ký tự hợp lệ trong việc định danh hơi phức tạp một chút khi bạn xem xét các ký tự không phổ biến như là Unicode. Nếu bạn chỉ xem xét các ký tự ASCII tiêu biểu thì các quy tắc lại trở nên đơn giản.

Một định danh nên bắt đầu với `a-z`, `A-Z`, `$`, hay _. Và nó có thể chứa bất kỳ các ký tự như vậy cùng với số từ `0-9`. Nói chung, các quy tắc tương tự áp dụng cho tên thuộc tính như là một biến số nhận diện. Tuy nhiên, một số từ nhất định không thể được sử dụng như các biến, nhưng cũng OK khi đặt tên thuộc tính. Những từ này gọi là từ "dành riêng", và bao gồm các từ khóa JS (`for`, `in`, `if`, v.v...) cũng như `null`, `true`, và `false`.

Ghi chú: Để biết thêm thông tin về từ dành riêng, xem Phụ lục A của tập *Kiểu & ngữ pháp*

Function Scopes (Phạm vi hàm)

Bạn sử dụng từ khóa `var` để khai báo biến cho phạm vi chức năng gần nhất, hoặc là toàn cục nếu nó nằm ở tầng trên cùng ngoài tất cả các hàm.

Hoisting

Khi bất kỳ một `var` xuất hiện bên trong phạm vi, việc khai báo có thể thực hiện mọi nơi trong toàn bộ phạm vi đó.

Một cách ẩn dụ, hành vi này gọi là *hoisting*, khi khai báo `var` "di chuyển" lên trên đầu phạm vi của chính nó. Về mặt kỹ thuật, quá trình này được giải thích chính xác hơn bằng code được biên dịch như thế nào, nhưng tạm thời chúng ta bỏ qua chi tiết.

Ví dụ:

```
var a = 2;
foo();                        // hoạt động nhờ khai báo
`foo()` được "dời lên"
function foo() {
     a = 3;
```

```
    console.log( a );   // 3
    var a;                  // khai báo được "dời" lên trên
đầu của `foo()`
}
console.log( a );   // 2
```

Chú ý: Đây là một ý không hay khi dựa vào biến *hoisting* để sử một biến trước đó trong phạm vi của nó hơn là `var`được khai báo bởi chính nó, điều này có thể gây bối rối. Cách thông thường và được chấp nhận để sử dụng hàm *hoisted*, là gọi trước khi nó được khai báo như chúng ta làm với `foo()`.

Phạm vi (scope) lồng nhau.

Khi bạn khai báo một biến, nó có hiệu lực ở toàn bộ trong phạm vi đó, kể cả phạm vi con. Ví dụ:

```
function foo() {

    var a = 1;
    function bar() {
        var b = 2;
        function baz() {
            var c = 3;
            console.log( a, b, c );  // 1 2 3
        }
        baz();
        console.log( a, b );         // 1 2
    }
    bar();
    console.log( a );                // 1
}
foo();
```

Chú ý là `c` không có bên trong `bar()`, bởi vì nó chỉ được khai báo bên trong scope `baz()`, và tương tự `b` không có trong `foo()`.

Nếu bạn muốn tiếp cận giá trị của một biến trong một scope không có nó, bạn sẽ gặp lỗi `ReferenceError`. Nếu bạn cố lập một biến chưa được khai báo, bạn cũng sẽ vô tình tạo một biến ở tầng cao nhất - toàn cục (bad!) hoặc gặp lỗi, tùy vào "strict mode" (xem "Strict Mode"). Hãy xem ví dụ:

```
function foo() {

    a = 1;    // `a` không được khai báo thông thường

}
foo();
a;            // 1 -- oops, tự động trở thành biến toàn cục
:(
```

Đây là một trường hợp vô cùng tệ. Đừng làm điều này, bạn phải luôn khai báo biến một cách bình thường.

Trong phương thức khai báo biến cho function, ES6 *cho phép* bạn khai báo các biến thuộc về các khối (block) riêng biệt `{...}` sử dụng từ khóa `let`. Bên cạnh các sắc thái chi tiết, nguyên tắc

của scope sẽ hoạt động y với những gì chúng ta đã thấy ở các function:

```
function foo() {

	var a = 1;
	if (a >= 1) {
		let b = 2;
		while (b < 5) {
			let c = b * 2;
			b++;
			console.log( a + c );
		}
	}
}
foo();
// 5 7 9
```

Bởi vì sử dụng `let` thay vì `var`, `b` chỉ thuộc về biểu thức `if` chứ không phải toàn bộ scope của function `foo()`. Tương tự, `c` chỉ thuộc về vòng lặp `while`. Cụm phạm vi hóa rất hữu ích để quản lý biến scope theo phương thức tốt hơn, giúp cho code của bạn dễ dàng bảo trì.

Ghi chú: Xem thêm phần *Scope & Closures* để nắm rõ hơn. Đối với phạm vi hóa `let`, xem phần *ES6 & Beyond*.

Điều kiện

Thêm một số thông tin về biểu thức `if` chúng ta đã giới thiệu ngắn gọn trong Chương 1, Javascript cung cấp một vài cơ chế mà chúng ta cần xem qua.

Đôi khi bạn sẽ thấy mình viết một loạt điều kiện `if..else..if` như sau:

```
if (a == 2) {

    // làm gì đó

}

else if (a == 10) {

    // làm gì đó khác

}

else if (a == 42) {

    // làm gì đó khác nữa

}

else {
```

```
    // không thì sẽ là
}
```

Cấu trúc này hoạt động, nhưng hơi bị rườm rà vì bạn cần kiểm thử `a` cho mỗi trường hợp. Đây là một phương thức khác với biểu thức `switch`:

```
switch (a) {
    case 2:
        // làm gì đó
        break;
    case 10:
        // làm gì đó khác
        break;
    case 42:
        //  làm gì đó khác nữa
```

```
        break;
    default:
        // không thì sẽ là
}
```

`break` quan trọng nếu bạn muốn mỗi biểu thức trong một `case` hoạt động. Nếu bạn bỏ sót `break` ở một `case`, và `case` đó chạy, sự thực thi sẽ tiếp tục đến biểu thức `case` tiếp theo vì `case` kia đã khớp. Cái này được gọi là "thất bại" đôi khi lại hữu dụng:

```
switch (a) {
    case 2:
    case 10:
        // làm gì đó
        break;
    case 42:
        // làm gì đó khác
```

```
        break;

    default:

        // không thì sẽ là

}
```

Ở đây, nếu `a` là `2` hoặc `10`, nó sẽ sẽ thực thi biểu thức "làm gì đó".

Một dạng khác của điều kiện trong JavaScript là “điều hành điều kiện”, thường được gọi là “toán tử bậc 3”. Nó như là một dạng rút gọn của biểu thức `if..else`, ví dụ:

```
var a = 42;
var b = (a > 41) ? "hello" : "world";
// tương tự với:
// if (a > 41) {
//    b = "hello";
// }
// else {
//    b = "world";
// }
```

Nếu biểu thức (`a > 41` ở đây) thỏa `true`, kết qủa là mệnh đề đầu tiên(`"hello"`), ngược lại là kết quả (`"world"`), và cho dù kết quả là gì thì đều gán vào `b`.

Điều hành điều kiện không nhất thiết phải gán như vậy, nhưng tất nhiên đây là cách được dùng nhiều nhất.

Chi ghú: Để biết thêm kiểm tra điều kiện và mẫu khác của `switch` và `? :`, xem phần *Kiểu & Ngữ pháp*.

Chế độ "nghiêm ngặt" (strict)

ES5 bổ sung "strict mode" cho ngôn ngữ, giúp các hành vi nhất định có nguyên tắc chặt chẽ hơn. Tổng quan thì sự strict được xem như là giữ cho code an toàn và phù hợp hơn. Đồng thời, tôn trong chế độ strict giúp cho bạn được tối ưu hóa hơn bởi cơ chế. Chế độ strict là bàn thắng lớn của code, và bạn nên sử dụng nó cho toàn bộ chương trình của mình.

Bạn có thể tham gia chế độ strict cho một hàm riêng biệt hay toàn bộ file, tùy thuộc bạn đặt chế độ strict đó ở đâu:

```
function foo() {

     "use strict";
     // Đoạn code này theo chế độ strict
     function bar() {
          // Đoạn code này theo chế độ strict
     }
}
```

```
// Đoạn code này không có chế độ strict
```

So sánh với:

```
"use strict";
function foo() {
	// Đoạn code này theo chế độ strict
	function bar() {
		// Đoạn code này theo chế độ strict
	}
}
// Đoạn code này theo chế độ strict
```

Điểm khác biệt mấu chốt (cải tiến!) ở chế độ strict là không cho phép tự động tiềm ẩn khai báo biến toàn cục khi bỏ qua `var`:

```
function foo() {

	"use strict";  // đặt chế độ strict

	a = 1;              // `var` thiếu, ReferenceError

}
foo();
```

Bạn sẽ gặp lỗi nếu bạn chuyển sang chế độ strict, hoặc code sẽ dính bug, bạn có thể bị cám dỗ việc né tránh strict. Nhưng bản năng đó là một ý tưởng tồi để lạm dụng. Nếu chế độ strict gây

ra các vấn đề trong chương trình, gần như chắc chắn nó là dấu hiệu rằng chương trình của bạn cần khắc phục.

Chế độ strict không chỉ giúp cho code của bạn theo một lối an toàn hơn, tối ưu hóa hơn, mà nó cũng là đại diện cho tương lai của ngôn ngữ. Bắt đầu với chế độ này sẽ dễ dàng hơn là sau mới chuyển qua.

Ghi chú: Xem thêm Chương 5 của phần *Kiểu & Ngữ pháp*.

Hàm là giá trị

Tới giờ, chúng ta đã đề cập hàm như là một cơ chế đầu tiên của *scope* trong JavaScript. Cú pháp khai báo hàm thông thường như sau:

```
function foo() {

	// ..

}
```

Cơ bản `foo` là một biến trong phạm vi bao quanh bên ngoài được tham chiếu với `function` khai báo, mặc dù dựa trên cú

pháp điều này không rõ ràng. Vậy nên, `function` bản thân nó là một giá trị, như là `42` hay `[1,2,3]`.

Mới nghe có vẻ lạ, nên có thể bạn cần một phút suy ngẫm chuyện này. Bạn không chỉ truyền giá trị (tham số) *vào* một hàm, mà *một hàm tự nó có thể là một giá trị* được gán vào biến, hoặc được truyền hay trả từ hàm khác.

Như vậy, một giá trị hàm có thể coi là một biểu thức, giống như các biểu thức hay giá trị khác.

Ví dụ:

```
var foo = function() {

	// ..

};
var x = function bar(){
	// ..
};
```

Biểu thức hàm đầu tiên gán vào biến `foo` được gọi là *anonymous(ẩn danh)*bởi vì không có `name`.

Biểu thức hàm tiếp theo được *đặt tên* (`bar`), ngay cả khi một tham chiếu đến nó cũng được gán vào biến `x`. *Hàm biểu thức được đặt tên* thường thích hợp hơn, mặc dù *hàm biểu thức vô danh* được sử dụng nhiều hơn.

Để tìm hiểu thêm, xem phần *Scope & Closures*

Immediately Invoked Function Expressions (IIFEs)

Trong đoạn code trên, muốn biểu thức function được thực thi—chúng ta phải có thêm `foo()` hoặc `x()`.

Có một cách khác để thực hi một biểu thức function, nó thường được gọi là *immediately invoked function expression—(tạm dịch) Hàm biểu thức thực hiện ngay lập tức* (IIFE):

```
(function IIFE(){

    console.log( "Hello!" );

})();
```

```
// "Hello!"
```

`( .. )` bao ngoài `(function IIFE(){ .. })` function là một sắc thái cần thiết trong ngữ pháp JS để bảo vệ nó khỏi bị hành xử giống như function thông thường.

Dấu `()` cuối cùng của biểu thức `})();` là chính xác cái gì sẽ thực thi tức thì trước nó.

Cái này có vẻ lạ, nhưng không phải xa lạ với cái nhìn đầu tiên. Xem ví dụ tương tự giữa `foo` và `IIFE`:

```
function foo() { .. }
// biểu thức function đại diện `foo`,
// sau đó `()` thực thi nó
foo();
// biểu thức function `IIFE`,
// sau đó `()` thực thi nó
(function IIFE(){ .. })();
```

Như bạn thấy, liệt kê `(function IIFE(){ .. })` trước khi nó thực thi `()` là cần thiết tương tự `foo` trước khi thực thi nó bằng `()`; trong cả hai trường hợp, function đại diện thực thi ngay tức thì sau dấu `()`.

Bởi vì IIFE chỉ là một function và function thì tạo *phạm vi* biến, sử dụng IIFE theo cách này thường là để khai báo biến không ảnh hưởng đến code bên ngoài IIFE:

```
var a = 42;
(function IIFE(){
     var a = 10;
     console.log( a );   // 10
})();
console.log( a );        // 42
```

IIFEs có thể trả kết quả:

```
var x = (function IIFE(){

     return 42;

})();
x;   // 42
```

Giá trị `42` được `return` từ `IIFE`- thực thi function được đặt tên theo `x`.

Closure (đóng kín)

Closure là một trong những khái niệm JS quan trọng nhất, và ít được hiểu nhất. Tôi sẽ không đi sâu ở đây, và sẽ phân tích ở *Scope & Closures*. Nhưng tôi sẽ nêu một vài vấn đề để bạn có cái nhìn tổng quan về nó. Đây sẽ là một trong nhưng kỹ thuật quan trọng nhất của bạn.

Bạn có thể nghĩ closure là một cách để “nhớ” và tiếp tục tiếp cận scope của hàm (biến) kể cả khi hàm đã hoàn tất.

Xem:

```
function makeAdder(x) {

     // tham số `x` là một biến bên trong hàm `add()`, vì vậy
nó là một "closure" thông qua nó.

     function add(y) {

          return y + x;

     };
     return add;
}
```

Mối tương quan giữa hàm `add(...)` được trả với mỗi lần gọi hàm `makeAdder(..)` trên nó là ghi nhớ giá trị `x` được truyền vào `makeAdder(..)`. Giờ hãy sử dụng `makeAdder(..)`:

```
// `plusOne` có một mối quan hệ khép kín với hàm `add(..)`

// thông qua tham số `x`

// của hàm `makeAdder(..)` trên nó

var plusOne = makeAdder( 1 );
// `plusTen` tương tự
var plusTen = makeAdder( 10 );
plusOne( 3 );       // 4  <-- 1 + 3
plusOne( 41 );      // 42 <-- 1 + 41
plusTen( 13 );      // 23 <-- 10 + 13
```

Code hoạt động như sau:

1. Khi gọi `makeAdder(1)`, chúng ta có được quy chiếu với `add(..)` bên trong nó là `x` bằng `1`. Chúng ta gọi là hàm tham chiếu `plusOne(..)`.
2. Tương tự khi ta gọi `makeAdder(10)`, chúng ta lại có một quy chiếu khác đến `add(..)` rằng `x` là `10`. Chúng ta gọi là hàm tham chiếu `plusTen(..)`.

3. Khi chúng ta gọi `plusOne(3)`, nó cộng `3` (`y` bên trong) với `1` (được ghi nhớ bởi `x`), và chúng ta có kết quả là `4`.
4. Khi chúng ta gọi `plusTen(13)`, nó cộng `13` (`y` bên trong) với `10` (được ghi nhớ bởi `x`), và chúng ta có kết quả là `23`.

Đừng lo lắng nếu nó có thể xa lạ và hơi bối rối lúc ban đầu (có thể!) Ta sẽ có nhiều bài tập để hiểu nó đầy đủ hơn. Hãy tin tôi, khi bạn đã hiểu, nó là một trong những những kỹ thuật bá đạo và hữu dụng nhất trong tất cả chương trình. Nó tất nhiên là đáng để cho não bạn căng lên chút. Trong phần tiếp theo, tôi sẽ có một ít bài tập với closure.

Modules

Cách sử dụng closure trong JS nhiều nhất là module pattern (mẫu mô-đun). Không giống như API công khai có thể tiếp cận từ phía ngoài, module cho phép bạn xác định các chi tiết (biến, hàm) thực hiện khép kín theo cá thể, ẩn khỏi các yếu tố bên ngoài.

Ví dụ:

```
function User(){
```

```
	var username, password;
	function doLogin(user,pw) {
		username = user;
		password = pw;
		// bên trong là các phương thức login
	}
	var publicAPI = {
		login: doLogin
	};
	return publicAPI;
}
// Tạo ngay một module `User`
var fred = User();
fred.login( "fred", "12Battery34!" );
```

Hàm `User()` thực hiện với vai trò là scope ngoài cùng chứa biến `username`và `password`, trong khi hàm `doLogin()`bên trong với các nội dung của module `User` đều là cục bộ và không thể tiếp cận từ bên ngoài.

Chú ý: Chúng ta không gọi `new User()` mặc dù nó có vẻ thông thường với nhiều người.`User()` chỉ là một hàm, không phải một class để khởi tạo, nên chỉ gọi nó bình thường. Sử dụng `new` là không thích hợp và đương nhiên là lãng phí tài nguyên.

Thực thi `User()` tạo ra một module `User` tức thì -- toàn bộ scope mới được tạo ra, và do đó một bản sao hoàn toàn mới của mỗi biến/hàm bên trong. Chúng ta gán nó với `fred`. Nếu chúng ta

chạy tiếp `User()`, chúng ta sẽ có một trường hợp mới tách biệt với `fred`.

Hàm `doLogin()` bên trong có một closure đối với `username` và `password`, nghĩa là nó vẫn giữ nguyên khả năng truy cập đến chúng sau khi hàm `User()` đã chạy xong.

`publicAPI` là một object với một thuộc tính/phương thức `login` trong nó. Khi chúng ta trả `publicAPI` từ `User()`, nó trở nên tức thì mà ta gọi là `fred`.

Lúc này, hàm `User()` đã hoàn tất thực thi. Thông thường, chúng ta nghĩ biến bên trong như là `username` và `password` đã biến mất. Nhưng không, bởi có một closure bên trong hàm `login()` giữ nó tồn tại.

Đó là lý do tại sao chúng ta gọi `fred.login(..)` cũng như khi gọi hàm `doLogin(..)` và nó vẫn truy cập biến `username` và `password` bên trong.

Tuy một vài vấn đề trong nó vẫn còn gây bối rối nhưng vậy tạm ổn rồi! Cũng cần vài thứ để nạp vào đầu bạn.

Từ chỗ này, xem phần *Scope & Closures* sẽ có sự khám phá sâu hơn.

Xác định this

Một concept khác của Javascript hay bị hiểu sai là xác định `this`. Chúng ta cũng có hẳn một phần nói về nó *this & Object Prototypes*, ở đây ta chỉ giới thiệu sơ qua thôi.

Trong mọi người thường nghĩ `this` liên quan đến "mẫu hướng đối tượng", trong JS `this` lại là cơ chế khác.

Nếu một hàm có `this` bên trong nó, sự quy chiếu của `this` này thường chỉ đến một `object`. Nhưng `object` nào thì tùy vào hàm nó gọi cái gì.

Điều quan trọng là `this` đó *không* tham chiếu đến bản thân hàm của nó, và đây là đây chính là điều hay bị hiểu lầm.

Đây là một minh họa nhanh:

```
function foo() {

    console.log( this.bar );
```

```
}
var bar = "global";
var obj1 = {
     bar: "obj1",
     foo: foo
};
var obj2 = {
     bar: "obj2"
};
// --------
foo();                    // "global"
obj1.foo();               // "obj1"
foo.call( obj2 );    // "obj2"
new foo();                // undefined
```

Có 4 nguyên tắc cho việc lập `this` ra sao, và nó được thể hiện bằng bốn dòng cuối của đoạn code.

1. `foo()` kết thúc việc thiết lập `this` bằng một object toàn cục ở chế độ không nghiêm ngặt (strict mode) -- với chế độ nghiêm ngặt thì `this`sẽ là `undefined` và bạn sẽ gặp lỗi khi truy vấn thuộc tính `bar` -- vì vậy `"global"`là giá trị tìm thấy cho `this.bar`.
2. `obj1.foo()` đặt `this` cho object `obj1`.
3. `foo.call(obj2)` đặt `this` cho object `obj2`.
4. `new foo()` đặt `this` cho một object rỗng mới.

Dòng cuối: để hiểu `this` trỏ đi đâu, bạn phải xem xét hàm được gọi như thế nào. Nó sẽ là một trong 4 cách vừa được thể hiện, và nó sẽ trả lời `this`là gì.

Ghi chú: Để biết thêm về `this`, xem Chương 1 và 2 của phần *this & Object Prototypes*.

Prototypes (nguyên mẫu)

Cơ chế nguyên mẫu trong JavaScript khá là phức tạp. Chúng ta chỉ xem qua nó ở đây. Bạn sẽ dành nhiều thời gian với nó hơn trong Chương 4–6 của phần *this & Object Prototypes*.

Khi bạn tham chiếu một thuộc tính bên trong object, nếu thuộc tính đó không tồn tại, JS sẽ tự động sử dụng tham chiếu nguyên mẫu bên trong để tìm object khác để tìm thuộc tính. Bạn có thể cho rằng đây gần như là dự phòng nếu thuộc tính bị thiếu.

Các mối liên kết tham chiếu nội bộ nguyên mẫu từ một object đến dự phòng của nó xảy ra tại thời điểm object được tạo. Cách đơn giản nhất để minh họa nó là hàm có sẵn

`Object.create(..)`.

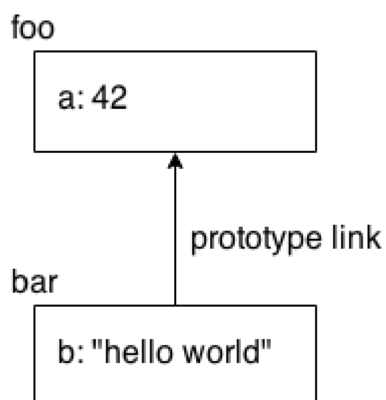
Ví dụ:

```
var foo = {

     a: 42

};
// tạo `bar` và liên kết với `foo`
var bar = Object.create( foo );
bar.b = "hello world";
bar.b;         // "hello world"
bar.a;         // 42 <-- ủy thác đến `foo`
```

Đây là sơ đồ mối quan hệ giữa `foo` và `bar`:



Thuộc tính `a` không thực sự tồn tại trong object `bar`, nhưng vì liên kết nguyên mẫu `bar` với `foo`, JavaScript tự động tìm `a` trong `foo` object.

Sự liên kết có lẽ là một đặc trưng đặc biệt khác lạ của ngôn ngữ. Cách thông dụng nhất của tính năng này được sử dụng nhiều nhất là cố giả lập/giả tạo một cơ chế "class" với "sự thừa kế".

Nhưng cách tự nhiên hơn để áp dụng nguyên mẫu là một pattern gọi là "hành vi ủy quyền (delegation)", nơi bạn cố ý thiết kế các đối tương liên kết có thể *ủy thác* từ một đến nhiều các phần cần hành vi đó.

Ghi chú: Để biết thêm về nguyên mẫu và hành vi ủy quyền, xem Chương 4–6 của phần *this & Object Prototypes*.

Cũ & Mới

Một số tính năng của JS chúng ta cũng đã biết, nhiều tính năng còn lại chúng ta sẽ tìm hiểu trong các phần còn lại, có những tính năng không phù hợp với trình duyệt cũ. Trên thực tế, có cả một số tính năng đặc trưng còn chưa được triển khai trên các trình duyệt ổn định.

Vì vậy, bạn phải làm gì với hàng mới? Bạn có phải chờ hàng năm hay cả thập kỷ để để cho các trình duyệt cũ chìm vào trong bóng tối?

Đó là cách nhiều người nghĩ vậy, nhưng đó là ý nghĩ không "lành mạnh" đối với việc tiếp cận JS.

Có hai kỹ thuật chính để đem đồ chơi mới của JavaScript chiến trên trình duyệt cũ: polyfilling và transpiling.

Polyfilling

Từ "polyfill" là một thuật ngữ được phát kiến bởi (Remy Sharp) (https://remysharp.com/2010/10/08/what-is-a-polyfill) được dùng để mô tả việc xác định một tính năng mới và tái tạo lại đoạn code có thể chạy trên môi trường JS cũ với những hành vi tương tự.

Ví dụ, ES6 định nghĩa một tiện ích gọi là `Number.isNaN(..)` để cung cấp một phương thức kiểm tra chính xác không lỗi cho giá trị `NaN` thay cho tiện ích gốc `isNaN(..)`. Nhưng cũng dễ dàng

để polyfill tiện ích đó nên bạn có thể bắt đầu sử dụng nó bất kể người dùng cuối có dùng trình duyệt ES6 hay không:

```
if (!Number.isNaN) {

    Number.isNaN = function isNaN(x) {

        return x !== x;

    };

}
```

Câu lệnh `if` bảo vệ việc áp dụng polyfill trong trình duyệt ES6. Nếu nó chưa có, chúng ta định nghĩa `Number.isNaN(..)`.

Ghi chú: Việc kiểm tra mà chúng ta thực hiện tận dụng lợi thế của giá trị quái đản `NaN`, nó là giá trị duy nhất trong toàn bộ ngôn ngữ không bằng với chính nó. Ví vậy giá trị `NaN` là loại duy nhất có thể làm cho `x !== x` trở thành `true`.

Không phải tất cả các tính năng mới để có thể polyfill được hoàn toàn. Đôi khi hầu hết các hành vi có thể được polyfill, nhưng cũng có những sai lệch nhỏ. Bạn phải rất rất cẩn thận

khi tự thực hiện polyfill, để đảm bảo rằng bạn đang tuân thủ các đặc điểm kỹ thuật càng chặt chẽ càng tốt.

Hoặc tốt hơn, sử dụng một bộ polyfill tin cậy đã được cung cấp bởi ES5-Shim (https://github.com/es-shims/es5-shim) và ES6-Shim (https://github.com/es-shims/es6-shim).

Transpiling

Không có cách nào để polyfill cú pháp mới khi đã được thêm vào ngôn ngữ. Cú pháp mới sẽ báo lỗi không nhận/không hợp pháp trong JS engine cũ.

Vì vậy phương án tiếp tốt hơn là sử dụng công cụ để chuyển đoạn code của bạn sang đoạn code cũ tương ứng. Các này thường được gọi là “transpiling”, cụm từ của transforming + compiling.

Về bản chất, mã nguồn của bạn được tạo theo dạng cú pháp mới, nhưng những gì bạn triển khai trên trình duyệt là mã nguồn cũ. Bạn thường cho transpiler trong quá trình build, tương tự như linter (kiểm lỗi cú pháp) & minifier (nén code).

Bạn có thể thắc mắc vì sao bạn phải cú pháp mới chỉ để chuyển nó theo cách cũ—tại sao không viết luôn code kiểu cũ cho rồi?

Có vài lý do quan trọng để bạn quan tâm đến transpiling:

- Cú pháp mới được thiết kế để đưa vào ngôn ngữ làm cho code của bạn dễ đọc dễ bảo trì hơn. Code cũ tương ứng thường phức tạp hơn rất nhiều. Bạn sẽ muốn viết theo cách mới với cú pháp rõ ràng hơn, không chỉ cho riêng bạn mà còn cho thành viên khác trong nhóm của bạn.
- Nếu bạn chỉ transpile cho trình duyệt cũ, sử dụng cú pháp mới cho trình duyệt mới, bạn sẽ tận dụng được hiệu suất của cú pháp mới. Nó đồng thời giúp người tạo ra trình duyệt có nhiều thực tế để kiểm tra và tối ưu trình duyệt của họ.
- Sử dụng cú pháp mới sớm hơn cho phép thử nghiệm mạnh mẽ hơn trong thực tế, đưa ra phản hồi cho ủy ban JavaScript (TC39). Nếu phát hiện sớm các vấn đề, họ có thể thay đổi/sửa chữa trước khi lỗi thiết kế ngôn ngữ trở nên vĩnh viễn.

Đây là một ví dụ nhanh cho transpiling. ES6 thêm một tính năng gọi là “tham chiếu mặc định” như thế này:

```
function foo(a = 2) {

     console.log( a );

}
foo();          // 2
foo( 42 );      // 42
```

Đơn giản phải không? Cũng rất hữu dụng nữa! Nhưng mà cú pháp mới này không tương thích với ES engine cũ. Vì vậy nó sẽ transpiler sang code cũ để phục vụ môi trường cũ.

```
function foo() {

     var a = arguments[0] !== (void 0) ? arguments[0] : 2;

     console.log( a );

}
```

Như bạn thấy, nó kiểm tra nếu giá trị `arguments[0]` là `void 0` (tức `undefined`), thì sẽ là giá trị mặc định `2`; ngược lại, nó sẽ chỉ định vào bất kỳ gì truyền vào.

Bây giờ ngoài việc có thể sử dụng cú pháp đẹp hơn cho trình duyệt cũ, nhìn vào mã transpile có thể giải thích hành vi rõ ràng hơn.

Bạn có thể không nhận ra rằng `undefined` là giá trị duy nhất không thể truyền vào thông qua việc cho một tham số mặc định khi nhìn vào bản ES6, nhưng ở bản transpile thì nó rõ ràng hơn.

Chi tiết quan trọng cuối cùng nhấn mạnh vào transpiler là không nên cho nó là một phần tiêu chuẩn của cộng đồng phát triển JS. JS sẽ tiếp tục phát triển nhanh chóng hơn bao giờ hết, nên chỉ vài tháng là có vài tính năng mới được bổ sung.

Nếu bạn mặc định sử dụng transpiler, bạn phải luôn cho phép nó chuyển qua cú pháp mới khi bạn thấy nó hữu dụng thay vì phải chờ đợi nhiều năm cho trình duyệt hiện tại lu mờ.

Có vài transpiler để bạn chọn, dưới đây là 2 cách để bạn xem xét:

- Babel (https://babeljs.io) (formerly 6to5): Transpiles ES6+ into ES5

- Traceur (https://github.com/google/traceur-compiler): Transpiles ES6, ES7, and beyond into ES5

Non-JavaScript

Tới giờ, điều duy nhất chúng ta quan tâm là bản thân JS. Thực tế thì hầu hết JS được viết để chạy và tương tác với môi trường như trình duyệt. Nhưng nói nghe hơi lạ là những gì bạn viết, thẳng ra là vậy, không được điều khiển bởi JavaScript.

Ví dụ đoạn code mà bạn hay dùng nhất trong DOM API không phải là JavaScript

```
var el = document.getElementById( "foo" );
```

Biến `document` tồn tại như một biến toàn cục khi code của bạn chạy trên trình duyệt. Nó không được cung cấp bởi JS engine hay được đặc biệt điều khiển bởi đặc tính JavaScript. Nó lấy dạng của gì đó trông kinh hơn JS `object` thông thường, mà cũng không chính xác là nó. Nó là một `object`đặc biệt, thường được gọi là "host object".

Hơn nữa, phương thức `getElementById(..)` ở `document` trông như hàm JS thông thường, nhưng nó chỉ là một giao diện mỏng được lộ ra với phương thức tích hợp bởi DOM từ trình duyệt. Trong một vài trình duyệt mới, lớp này có thể ở trong JS, nhưng bản chất thì DOM và hành vi của nó được thực hiện bởi gì đó như là C/C++.

Một ví dụ khác là với input/output (I/O).

Mọi người ưa thích pop up `alert(..)` trên cửa sổ trình duyệt. `alert(..)`được cung cấp cho JS của bạn thông qua trình duyệt, không phải JS engine. Khi gọi hàm, nó gửi nội dung đến bên trong trình duyệt và nó sẽ vẽ và hiển thị khung nội dung đó.

Tương tự với `console.log(..)`; trình duyệt của bạn cung cấp cơ chế và nối chúng với công cụ phát triển (developer tools).

Cuốn sách này tập trung vào ngôn ngữ JavaScript. Vì vậy bạn sẽ không thấy để cập non-JavaScript nhiều. Tuy nhiên bạn cần phải có nhận thức về chúng, nó ngay trong mọi chương trình JS bạn viết.

Ôn tập

Bước đầu tiên để học JavaScript là hiểu cơ bản về cơ chế cốt lõi của nó như: value, type, closure, `this`, và prototypes.

Đương nhiên, mỗi đề mục được đáng được cụ thể hơn bạn thấy ở đây, đó là lý do tại sao nó được đề cập suốt trong các phần bộ sách. Sau khi bạn cảm thấy thoải mái với các khái niệm và mẫu code trong chương này, phần còn lại của bộ sách chờ bạn đào sâu vào ngôn ngữ hơn.

Chương cuối của cuốn sách sẽ tóm tắt từng mỗi tiêu đề và các khái niệm khác bên cạnh những gì chúng ta khám phá.

You Don’t Know JS (tiếng Việt)—Phạm vi & Đóng kín— Chương 1: Scope (phạm vi) là gì?

O'REILLY®

"Kyle's way of critically thinking about every tiny bit of the language will creep into your mindset and general workflow."
—**SHANE HUDSON**, freelance frontend website developer

KYLE SIMPSON

SCOPE & CLOSURES

YOU DON'T KNOW JS

Một trong những mô hình cơ bản của các ngôn ngữ lập trình là khả năng lưu trữ giá trị trong biến và sau đó gọi các giá trị đó hoặc thay đổi chúng. Trên thực tế, khả năng lưu trữ giá trị và lấy giá trị đó ra khỏi biến là cách tạo ra *state* của chương trình.

Không có khái niệm này, một chương trình sẽ cực kỳ giới hạn và thậm chí không thú vị mặc dù nó có thể thực thi một số nhiệm vụ.

Nhưng việc đưa các biến vào trong chương trình của chúng ta đặt ra câu hỏi thú vị nhất mà chúng ta sẽ giải quyết: những biến đó *sống* ở đâu? Hay nói cách khác, nó được lưu ở đâu? Và quan trọng nhất là làm cách nào chương trình có thể tìm thấy nó khi cần?

Những câu hỏi nói lên sự cần thiết của các quy tắc được xác định rõ ràng của việc lưu trữ các biến ở đâu đó, cùng việc tìm các biến sau đó. Chúng ta gọi quy tắc đó là *Scope*

Nhưng, các quy tắc *Scope* được thiết lập ở đâu và như thế nào?

Lý thuyết trình biên dịch

Tùy thuộc vào mức độ bạn tương tác với các ngôn ngữ khác nhau, nó có thể là hiển nhiên hoặc lạ lẫm, mặc dù thực tế JS là một ngôn ngữ nằm trong hạng mục "động" hoặc "giải nghĩa", thực tế nó là ngôn ngữ được biên dịch. Cũng như nhiều ngôn ngữ được biên dịch truyền thống khác, nó không được biên dịch trước hay cũng không phải là kết quả của sự kết hợp giữa các hệ thống phân phối khác nhau.

Tuy nhiên JS engine thực hiện nhiều bước đồng thời, theo những cách phức tạp hơn mà chúng ta từng biết, kể cả bất trình biên dịch truyền thống nào.

Trong tiến trình biên dịch của ngôn ngữ truyền thống, mã nguồn sẽ thông qua ba bước trước khi thực thi chương trình, gọi là biên dịch:

1. **Tokenizing/Lexing**: Tách các chuỗi ký tự thành các khối có ý nghĩa (đối với ngôn ngữ), được gọi là token (thẻ xác thực). Ví dụ: `var a = 2;`. Chương trình sẽ được phân thành các token sau: `var`, `a`, `=`, `2`, và `;`. Khoảng trắng có thể có hoặc không duy trì như một token, tùy thuộc vào nó có nghĩa hay không.

2. **Ghi chú**: Sự khác biệt giữa tokenizing và lexing là tinh tế và học thuật (subtle and academic), nhưng nó tập trung vào việc các token này được xác định theo cách là có *trạng thái* hay *không trạng thái*. Nói đơn giản, nếu tokenizer gọi các quy tắc phân tích trạng thái để xác định liệu `a` có được xem là một token riêng biệt hay chỉ là một phần của token, *đó*chính là lexing.
3. **Phân tích cú pháp (parsing)**: lấy một mảng (array) token và chuyển nó thành các phần tử cây lồng nhau, đại diện cho cấu trúc ngữ pháp của chương trình. Cây này được gọi là "AST" (**A**bstract **S**yntax **T**ree).
4. Cây của `var a = 2;` có thể bắt đầu với một nút cao nhất được gọi là `VariableDeclaration`, với một nút con được gọi là `Identifier` (giá trị `a`), và nút con khác được gọi là `AssignmentExpression` với bản thân nó có một con được gọi là `NumericLiteral` (giá trị `2`).
5. **Xử lý mã (Code-Generation)**: là quá trình lấy AST và biến nó thành mã thực thi. Phần này khác nhau rất nhiều tùy thuộc vào ngôn ngữ, nền tảng mà nó nhắm đến, ...

Vì vậy, thay vì đắm chìm vào chi tiết, chúng ta chỉ phẩy tay và nói rằng có cách để lấy AST được mô tả ở trên cho `var a = 2;`

và chuyển nó thành một tập mã máy để thực sự *tạo* ra một biến gọi là `a` (bao gồm cả việc lưu trữ bộ nhớ, v.v...), và sao lưu trữ một giá trị vào `a`.

Ghi chú: Chi tiết về quản lý hệ thống của máy sâu hơn những gì chúng ta sẽ tìm hiểu, vì vậy chúng ta chỉ cần biết rằng máy có thể tạo và lưu trữ biến theo mong muốn.

Cũng như những ngôn ngữ biên dịch khác, JavaScript engine phức tạp và mênh mông hơn là chỉ có ba bước trên. Ví dụ, trong tiến trình phân tích và xử lý mã, trong đó đã có các bước tối ưu hiệu suất thực thi, bao gồm gộp các phần tử thừa,...

Vì vậy, tôi chỉ tô vẽ những nét lớn ở đây. Nhưng tôi nghĩ bạn sẽ mau nhận ra vì sao những chi tiết đó chúng ta cần biết, thậm chí là ở mức cao.

Có một điều là JS engine không có nhiều thời gian tối ưu hóa sang chảnh như các trình biên dịch ngôn ngữ khác, bởi việc biên dịch JS không xảy ra trước trong bước xây dựng như những ngôn ngữ khác.

Với JavaScript, trong nhiều trường hợp, sự biên dịch được xảy ra trong vài mili giây (hoặc ít hơn) trước khi code được thực

thi. Để đảm bảo hiệu suất nhanh nhất, JS engine sử dụng tất cả các mẹo vượt qua luôn cả phạm vi chúng ta thảo luận ở đây (như JITs, biên dịch chậm và thậm chí biên dịch lại, ...).

Nói cho đơn giản, bất kỳ đoạn code JavaScript cũng phải được biên dịch trước khi nó thực thi (thường là *ngày trước khi*). Vậy, trình biên dịch JS sẽ lấy `var a = 2;` và biên dịch nó *trước*, sau đó sẽ thực thi nó, thường là ngay tức thì.

Hiểu Scope

Cách chúng ta tiếp cận học scope là nghĩ về quá trình của một cuộc hội thoại. Nhưng *ai* có cuộc hội thoại đó?

Vai diễn

Hãy gặp vai diễn của các nhân vật tương tác để tiếp cận chương trình `var a = 2;` để hiểu đoạn hội thoại mà chúng ta sẽ được xem:

1. *Engine*: chịu trách nhiệm biên soạn và thực thi từ đầu đến kết thúc chương trình JavaScript của chúng ta.
2. *Compiler*: Một trong những người bạn của *Engine*; phụ trách tất cả các việc nặng nề của của phân tích cú pháp và xử lý mã (xem phần trên).

3. *Scope*: người bạn khác của *Engine*; tập hợp và duy trì một danh sách các định danh đã khai báo (biến), và thực hiện một tập hợp các quy tắc nghiêm ngặt về cách thức truy cập các mã vừa thực thi.

Để bạn *hiểu đầy đủ* hơn JavaSript làm việc như thế nào, bạn cần bắt đầu *nghĩ* như *Engine* (và bạn của nó), đặt những câu hỏi chúng hỏi, đồng thời trả lời những câu hỏi đó.

Trước và sau

Khi bạn thấy chương trình `var a = 2;` bạn hầu như nghĩ nó như một câu lệnh. Nhưng mà anh bạn mới *Engine* của chúng ta không thấy vậy. Chính xác là *Engine* thấy hai câu lệnh riêng biệt, một do *Compiler* sẽ phụ trách trong quá trình biên dịch, và một do *Engine* sẽ phụ trách trong quá trình thực thi.

Vậy, hãy phân tích cách *Engine* và bạn bè đã tiếp cận `var a = 2;`

Đầu tiên *Compiler* sẽ thực hiện phân tích để tách nó ra thành các token, các token này sẽ tách thành cây. Nhưng khi *Compiler* thực hiện xử lý mã, nó sẽ coi chương trình hơi khác so với giả định.

Một giả định hợp lý là *Compiler* sẽ tạo ra mã có thể được tóm tắt bởi mã giả: “Phân bổ bộ nhớ cho biến, gắn nhãn `a`cho nó, sau đó gán giá trị `2` cho biến đó". Thật không may là nó không hẳn chính xác.

Compiler sẽ làm như sau:

1. Gặp `var a`, *Compiler* hỏi *Scope* xem nếu biến `a` có tồn tại trong bộ scope cụ thể đó hay không. Nếu có, *Compiler* bỏ qua khai báo và tiếp tục. Ngược lại, *Compiler* yêu cầu *Scope* khai báo một biến mới tên `a`.
2. *Compiler* sau đó tạo ra code cho *Engine* thực thi sau đó, thực hiện việc gán `a = 2`. Khi *Engine* chạy sẽ hỏi *Scope*rằng có biến nào trong bộ sưu tập tên là `a` có thể truy cập được hay không. Nếu có, *Engine* sử dụng biến đó, nếu không thì *Engine* sẽ tìm *ở đâu đó* (xem phần *Scope* lồng nhau bên dưới).

Nếu *Engine* cuối cùng tìm thấy một biến, nó gán giá trị `2` vào đó. Nếu không, *Engine* sẽ hô lên là có lỗi!

Túm lại: Hai hành động riêng biệt được thực hiện cho một phép gán biến: Một là *Compiler* khai báo một biến (nếu chưa

có trong scope hiện tại), và thứ hai là khi thực thi, *Engine* tìm biến trong *SCope* và nếu tìm thấy thì sẽ gán nó vào.

Lời của Compiler

Chúng ta cần một chút thuật ngữ của compiler để tiếp tục hiểu sâu hơn.

Khi *Engine* thực thi code do *Compiler* tạo ra cho bước (2), nó phải tra cứu biến `a` để xem nó đã khai báo hay chưa, và việc tra cứu này là cố vấn *Scope*. Nhưng kiểu tra cứu *Engine* thực hiện ảnh hưởng đến kết quả của việc tra cứu.

Trong trường hợp của chúng ta, *Engine* thực hiện một tra cứu “LHS” cho biến `a`. Một kiểu tra cứu khác là "RHS".

Tôi cá là bạn có thể đoán “L” và “R” nghĩa là gì. Cụm từ để chỉ cho “Left-hand Side (Phía bên trái)” và “Right-hand Side (Phía bên phải)”.

Phía… của cái gì?

Của một phép gán

Nói cách khác, tra cứu LHS được hoàn tất khi một biến xuất hiện ở bên trái của phép gán, và một tra cứu RHS hoàn tất khi một biến xuất hiện ở bên phải của phép gán.

Chính xác hơn một chút, tìm kiếm RHS không thể phân biệt được với mục đích của chúng ta, đơn giản chỉ là tra cứu của một giá trị của biến nào đó, trong khi tra cứu LHS để tìm vùng chứa biến để gán. Bằng cách này, RHS không *thực sự* nghĩa là "phía bên phải của một phép gán" cho chính nó, mà chính xác chỉ là "không phải phía bên trái".

Ngắn gọn hơn, bạn có thể nghĩ "RHS" thay vì "lấy đi nguồn (giá trị) của nó" thì là "đi lấy giá trị của..."

Đào sâu hơn chút. Khi tôi nói:

```
console.log(a);
```

Tham chiếu đến `a` là một tham chiếu RHS, bởi vì không có gì được gán cho `a` cả. Thay vào đó, chúng ta đang tìm kiếm để lấy giá trị của `a` để truyền vào `console.log(..)`.

Ngược lại:

```
a = 2;
```

Tham chiếu của a ở đây là tham chiếu LHS, bởi vị chúng ta không cần quan tâm giá trị gần nhất là gì, chúng ta chỉ đơn giản muốn tìm biến mục tiêu của phép gán =a

Ghi chú: LHS và RHS có nghĩa "phép gán bên trái/phải" không nhất thiết có phải chính xác là "phía bên trái/phải của phép gán =". Có vài cách khác mà phép gán đó có thể xảy ra, do đó tốt hơn là nghĩ khái niệm đó như sau: "cái gì là mục tiêu của phép gán (LHS)" và "cái gì là nguồn của phép gán (RHS)"

Ví dụ chương trình có cả LHS và RHS dưới đây:

```
function foo(a) {

     console.log( a ); // 2

}
foo( 2 );
```

Dòng cuối cùng gọi hàm foo (..) yêu cầu một tham chiếu RHS đến foo, có nghĩa là,"hãy tra cứu giá trị của foo, và đưa nó cho

tôi". Hơn nữa, `(..)` có nghĩa là giá trị của `foo` cần được thực hiện, vì vậy nó thực sự là một hàm!

Bạn có phát hiện ra được là có một phép gán tinh tế quan trọng ở đây không?

Bạn có thể quên ngụ ý của `a = 2` trong đoạn code này. Nó xảy ra khi giá trị `2` được truyền như một đối số trong hàm `foo(..)`, tức `2` được gán cho tham số `a`. Phép tra cứu LHS đã được thực hiện để (ngầm) gán đến tham số `a`.

Đồng thời cũng có một quan hệ RHS của giá trị `a`, và kết quả được chuyển đến `console.log(..)`. `console.log(..)` cần một tham chiếu để thực thi, đó là tra cứu RHS cho object `console`, sau đó xử lý thuộc tính thực hiện để xem có phương thức nào gọi là `log` hay không.

Cuối cùng, chúng ta có thể khái niệm hóa rằng có một trao đổi LHS / RHS chuyển giá trị `2` (bằng cách tra cứu RHS biến `a`) vào `log (..)`. Bên trong việc thực hiện `log (..)`, chúng ta có thể giả định rằng nó có các tham số, phần đầu (chắc gọi là `arg1`) có một tham chiếu LHS trước khi gán `2` vào nó.

Ghi chú: Bạn có thể bị cám dỗ khái niệm hóa khai báo hàm `function foo(a) {...` như một khai báo biến và gán bình thường như `var foo` và `foo = function(a){...`. Làm như vậy thì nó sẽ hướng suy nghĩ rằng việc khai báo hàm này như một tra cứu LHS.

Tuy nhiên, sự khác biệt tinh tế quan trọng ở đây là *Compiler* đảm nhiệm cả việc khai báo và xác định giá trị trong quá trình tạo code. Chẳng hạn như khi *Engine* thực thi code, không cần phải thêm bước "gán" một giá trị hàm cho `foo`.

Hội thoại Engine/Scope

```
function foo(a) {

     console.log( a ); // 2

}
foo( 2 );
```

Hãy tưởng tượng sự trao đổi xử lý đoạn mã ở trên như một đoạn hội thoại. Đoạn hội thoại sẽ trở nên gì đó giống như vầy:

Engine: Ê *Scope*, tao có một tham chiếu RHS cho `foo`. Có thấy bao giờ chưa?

Scope: Có có, có chớ. *Compiler* mới khai báo một giây trước. Nó là hàm, đây nè.

Engine: Tuyệt, cảm ơn. OK, tao đang xử `foo`

Engine: Ê *Scope*, tao cũng có tham chiếu LHS cho `a` nè, mày biết ko?

Scope: Hả, có luôn. Đây nè, *Compiler* mới báo nó như một tham số cho `foo`.

Engine: Vẫn ngon như mọi khi, *Scope* ạ. Cảm ơn nhiều. Giờ thì gán `2` cho `a` thôi.

Engine: Ờ... *Scope*, làm phiền mày tiếp. Tao cần tra cứu RHS để kiếm `console`. Có không?

Scope: Không sao *Engine*, tao làm mấy chuyện này suốt ngày. Tao có `console` chớ, nó là hàm dựng sẵn mà.*Engine*: Perfect. Looking up `log(..)`. OK, great, it's a function.

Engine: Yo, *Scope*, mày có thể giúp tao một tham chiếu RHS cho `a` không? Tao nghĩ là tao nhớ nó rồi, nhưng mà kiểm lại lần nữa.

Scope: Ờ đúng rồi, cũng là nó đó, không đổi gì hết.

Engine: Ngon rồi. Chuyển giá trị `2` cho `a` vào trong `log(..)`.

...

Câu đố

Giờ kiểm tra khả năng hiểu của bạn. Mà nhớ là chơi trò này giống như là một phần của “hội thoại” của hai thằng *Engine* với *Scope*:

```
function foo(a) {

    var b = a;

    return a + b;

}
var c = foo( 2 );
```

1. Xác định tất cả tra cứu LHS (có 3!).
2. Xác định các tra cứu RHS (có 4!).

Ghi chú: Xem lời giải ở Ôn tập!

Scope lồng nhau

Chúng ta đã nói *Scope* là một tập hợp các quy tắc để tìm kiếm biến được nhận diện theo tên. Tuy nhiên thì thường có nhiều hơn một *Scope*.

Cũng như một cụm hàm được lồng vào cụm hàm khác, scope này cũng lồng trong scope kia. Nên nếu một biến không tìm thấy trong scope ngay liền, *Engine* sẽ tham khảo scope bên ngoài, tiếp tục kiếm cho tới khi nào ra tới ngoài cùng nhất (cũng là toàn cục).

Ví dụ:

```
function foo(a) {

     console.log( a + b );

}
var b = 2;
foo( 2 ); // 4
```

Tham chiếu RHS cho `b` khộng thể tìm thấy trong hàm `foo`, nhưng nó có thể được thấy trong *Scope* bao ngoài (trong trường hợp này là toàn cục).

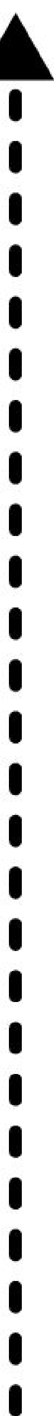
Vì vậy, quay lại đoạn hội thoại giữa *Engine* và *Scope*, chúng ta sẽ nghe:

Engine: "Ừầy, *Scope* của `foo`, có `b` không? Có tham chiếu cho nó nè"
Scope: "Không thấy, chưa nghe bao giờ, thôi lượn đi"
Engine: "Này, *Scope* bên ngoài`foo`, à ờ mày là *Scope* toàn cục à, cũng được . Nghe thấy `b` bao giờ chưa? có tham chiếu RHS cho nó nè."
Scope: "Có, chắc cú. Đầy nè."

Nguyên tắc đơn giản để đi qua *Scope* lồng nhau là: *Engine* bắt đầu từ xử lý *Scope* gần nhất, tìm biến trong đó, nếu không có, lên tầng trên, và cứ tiếp tục như vậy. Tới khi ra tận phạm vi toàn cục, tìm thấy biến hay không thì việc tìm kiếm cũng dừng lại.

Xây dựng trên phép ẩn dụ.

Để hình dung chi tiết *Scope* lồng nhau, bạn hãy nghĩ nó như một tòa nhà cao tầng.


Global Scope
Current Scope

Tòa nhà đại diện cho một tập hợp *Scope* lồng nhau của chương trình. Tầng trệt của tòa nhà đại diện cho *Scope* thực thi gần nhất. Tầng cao nhất là *Scope* toàn cục.

Chúng ta giải quyết tham chiếu LHS và RHS bằng việc nhìn vào tầng hiện tại của bạn, và nếu bạn không tìm thấy nó, leo lên tầng tiếp theo rồi tìm, rồi tiếp tục như vậy. Cho đến khi bạn đến tầng thượng (*Scope* toàn cục), bạn có thể tìm thấy cái mình cần hoặc không. Bất kể sao thì bạn vẫn dừng lại.

Lỗi

Tại sao lại quan trọng khi chúng ta gọi nó là LHS hay RHS?

Bởi vì có hai kiểu tra cứu hành xử khác nhau trong trường hợp biến chưa được khai báo (không tìm thấy trong bất *Scope* nào).

Ví dụ:

```
function foo(a) {

     console.log( a + b );

     b = a;
```

```
}
foo( 2 );
```

Khi tra cứu RHS tìm `b` lần đầu xảy ra, nó sẽ không được tìm thấy. Đây được gọi là "chưa khai báo biến" bởi vì nó không tìm thấy trong phạm vi.

Nếu một tra cứu RHS thất bại khi tìm biến, kết quả `ReferenceError` sẽ được đưa ra bởi *Engine*. Việc báo lỗi `ReferenceError` quan trọng.

Ngược lại, nếu *Engine* thực hiện tra cứu LHS và đến tầng trên cùng rồi mà vẫn không thấy nó, và nếu chương trình không đang chạy ở "strict mode", *Scope* toàn cục sẽ tạo ra một biến mới theo tên đó ở phạm vi toàn cục và chuyển đến *Engine*

"Không, chưa bao giờ có biến đó cả, nhưng tao sẽ hỗ trợ tạo một cái cho mày."

"Strict Mode" được thêm vào từ ES5, có một số hành vi khác nhau từ chế độ bình thường/thoải mái/lười, một trong những hành vi như vậy sẽ không cho phép tự động tạo biến toàn cầu hay ngầm định. Trong trường hợp đó, sẽ không có biến nào

của *Scope* toàn cục để trả cho tra cứu LHS, và *Engine*sẽ đưa ra lỗi `ReferenceError` tương tự trường hợp RHS.

Giờ đây, nếu một biến được tìm thấy cho một tra cứu RHS, nhưng nếu cố làm gì đó với giá trị của nó là không thể, chẳng hạn như cố thực thi hàm cho một giá trị phi hàm hoặc tham chiếu một thuộc tính của giá trị `null` hay `undefined`, *Engine* sẽ đưa ra một loại lỗi khác, gọi là `TypeError`.

`ReferenceError` là lỗi liên quan của chi tiết *Scope* trong khi `TypeError` lại ngụ ý rằng chi tiết *Scope* đúng, nhưng lại có một hành động không hợp lệ/bất khả thi cố gắng đi ngược kết quả.

Ôn tập (TL;DR)

Phạm vi là một tập hợp các nguyên tắc xác định biến được tìm ở đâu và như thế nào. Việc tra cứu có thể nhằm mục đích gán biến, là tham chiếu từ LHS, hay nhằm mục đích truy xuất dữ liệu của nó dựa trên tham chiếu RHS.

JavaScript *Engine* biên dịch code trước khi thực thi, và khi làm như vậy, nó chia các lệnh, ví dụ `var a = 2;`, thành hai bước riêng biệt:

1. Đầu tiên, `var a` để khai báo nó trong *Scope* đó. Việc này được thực hiện từ đầu, trước khi code được thực thi.
2. Sau đó, `a = 2` để tìm biến (LHS) và gán cho nó nếu nó được tìm thấy.

Cả tra cứu LHS và RHS đều được bắt đầu ở *Scope* thực thi gần nhất, và nếu cần thiết (tức là nó không tìm thấy cái nó đang tìm trong đó), nó sẽ hoạt động bằng cách đi lên trên *Scope* lồng nhau, trong một thời điểm việc tìm kiếm thực hiện cho đến khi nó đi lên tầng cuối cùng toàn cục rồi dừng lại, kể cả có tìm thấy hay không.

Kết quả tham chiếu RHS không hoàn thành sẽ là lỗi `ReferenceError`. Kết quả tham chiếu LHS sẽ tự động tạo ra một biến toàn cục theo tến đó nếu không ở "Strict Mode" hoặc lỗi `ReferenceError` nếu đang ở "strict mode".

Trả lời câu đố

```
function foo(a) {

    var b = a;
```

```
    return a + b;

}
var c = foo( 2 );
```

1. Nhận diện tất cả các tra cứu LHS.
2. `c = ..`, `a = 2` (ngầm gán) và `b = ..`
3. Nhận diện tất cả các tra cứu RHS.
4. `foo(2..`, `= a;`, `a + ..` và `.. + b`

*MDN: Strict Mode

You Don’t Know JS (tiếng Việt)—Phạm vi & Đóng kín—Chương 2: Phạm vi từ vựng (Lexical Scope)

Trong Chương 1, chúng ta đã tìm hiểu “phạm vi” như một tập hợp quy tắc cách vận hành *Engine*, tra cứu biến với tên nhận diện có thể là trong *Scope* gần nhất hoặc bất kỳ *Scope lồng nhau*.

Có hai mô hình nổi bật về cách hoạt động của phạm vi. Cái đầu tiên gọi là Lexical Scope, là cái thông thường nhất, được sử dụng phần lớn bởi các ngôn ngữ lập trình, chúng ta sẽ tìm hiểu sâu hơn. Mô hình khác gọi là Dynamic Scope vẫn được thường sửa dụng ở vài ngôn ngữ (ví dụ như Bash, vài mô hình trong Pearl).

Dynamic Scope được đề cập ở Phụ lục A. Tôi gợi ý nó chỉ để cung cấp một sự tương phản với Lexical scope, một mô hình mà JavaScript sử dụng.

Lex-time

Theo thảo luận ở Chương 1—phần 1, giai đoạn truyền thống đầu tiên của trình dịch ngữ được gọi là lexing (hay tonkenizing). Nếu bạn nhớ lại, quá trình lexing kiểm tra một chuỗi các ký tự mã nguồn và gán ngữ nghĩa cho các token như là kết quả của một số trạng thái phân tích cú pháp.

Chính khái niệm này cung cấp nền tảng để hiểu được phạm vi từ vựng là gì và cái tên đến từ đâu.

Lexical scope là một phạm vi được định nghĩa trong thời điểm lexing (lexing time). Nói cách khác, lexical scope dựa trên các biến và khối phạm vi do bạn tạo tại thời điểm viết, do đó hầu hết viên gạch được xây ra từ ngay thời điểm lexer xử lý code của bạn.

Ghi chú: Chúng ta sẽ thấy có vài cách để đánh lừa lexical scope bằng cách sửa đổi nó sau khi lexer đã truyền qua, nhưng việc này không hay. Phương thức tốt nhất vẫn là hành xử lexical scope với vài trò lexical thôi, mọi thứ đều theo tự nhiên.

Xem đoạn code dưới đây:

```
function foo(a) {
     var b = a * 2;
     function bar(c) {
          console.log( a, b, c );
     }
     bar(b * 3);
}
foo( 2 ); // 2 4 12
```

Có ba phạm vi lồng nhau ở đoạn code ở trên. Để dễ hình dung thì cứ nghĩ các phạm vi đó như bong bóng bên trong từng cụm.

```
function foo(a) {
	var b = a * 2;

	function bar(c) {
		console.log( a, b, c );
	}

	bar(b * 3);
}

foo( 2 ); // 2, 4, 12
```

Bong bóng 1 bao gồm phạm vi toàn cục, và chỉ có một đối tượng trong nó: `foo`.

Bong bóng 2 bao gồm phạm vi của `foo`, bao gồm ba đối tượng: `a`, `bar`and `b`.

Bong bóng 3 bao gồm phạm vi của `bar`, và chỉ có một đối tượng: `c`.

Phạm vi của các bong bóng được xác định ở tại các cụm scope được viết, cái này chồng cái kia,... Trong chương tiếp theo,

chúng ta sẽ thảo luận các đơn vị khác của scope nhưng giờ chỉ cần giả định mỗi hàm tạo một scope bong bóng mới.

Bong bóng `bar` được chứa hoàn toàn trong bong bóng `foo`, bởi vì (và chỉ vì) đó là nơi chúng ta chọn để xác định hàm `bar`.

Lưu ý rằng những bong bóng lồng nhau này được lồng một cách chặt chẽ. Chúng ta không nói về biểu đồ Venn, nơi các bong bóng có thể vượt qua ranh giới. Nói khác đi, không có bong bóng cho một số hàm đồng thời tồn tại bên trong hai bong bóng bên ngoài khác, cũng như không có hàm nào có thể một phần bên trong hai hàm cha.

Tra cứu

Cấu trúc và vị trí tương quan của các bong bóng phạm vi này giải thích đầy đủ cho cho *Engine* tất cả các nơi nó cần đến để tìm nhận diện.

Ở đoạn code ở trên, *Engine* thực thi biểu thức `console.log(..)` và tìm ba biến tham chiếu `a`, `b`, và `c`. Đầu tiên nó sẽ tìm ở phạm vi bong bóng bên trong nhất là hàm `bar(...)`. Nếu nó không tìm thấy `a` ở đó, nó sẽ đi lên trên một bậc là phạm vi `foo(...)`. Nếu

nó tìm thấy `a` ở đó, thì nó sẽ dùng `a`. Tương tự với `b`. Nhưng `c` được thấy bên trong `bar(...)`

Nếu đã có `c` bên trong `bar (..)` và bên trong `foo (.. )`, lệnh `console.log (..)` đã tìm và sử dụng trong `bar ( ..)` rồi, nên không bao giờ là cái trong `foo (..)`.

Tra cứu phạm vi dừng lại một khi nó "khớp lệnh". Tên định danh giống nhau có thể được đặt ở nhiều lớp khác nhau trong phạm vi lồng nhau, được gọi là "đổ bóng" (nhận dạng bên trong "che bóng" nhận dạng bên ngoài). Bất kể là che bóng, phạm vi luôn bắt đầu tra cứu từ phạm vi trong nhất, và làm việc từ trong ra ngoài, từ dưới lên trên cho đến khi nó khớp lệnh và dừng lại.

Ghi chú: Các biến toàn cục cũng tự động là thuộc tính của đối tượng toàn cục (`window` của các trình duyệt...) vì thế có thể tham chiếu đến một biến toàn cục không trực tiếp bằng tên từ vựng của nó, thay vào đó là gián tiếp bằng một đại diện thuộc tính toàn cục.

```
window.a
```

Kỹ thuật này cho phép truy cập vào một biến toàn cục, nếu không thì sẽ bị che bóng. Tuy nhiên, không phải biến toàn cục mà không bị che bóng thì cũng không truy cập được.

Cho dù một hàm được gọi ra từ đâu, hay kể cả được gọi như thế nào, phạm vi từ vựng của nó chỉ xác định tại nơi hàm được khai báo.

Quy trình tra cứu phạm vi từ vựng chỉ áp dụng cho lớp nhận dạng đầu tiên, ví dụ như `a`, `b`, và `c`. Nếu bạn có một tham chiếu đến `foo.bar.baz` trong một mẩu code, phạm vi từ vựng sẽ tra cứu đến nhận diện `foo`, một khi đã định vị biến đó rồi, các quy tắc truy cập thuộc tính đối tượng sẽ tiếp tục giải quyết thuộc tính `bar` và `baz` tương ứng.

Ăn gian từ vựng

Nếu phạm vi từ vựng chỉ được xác định tại nơi hàm được khai báo, hoàn toàn là một quyết định của author-time (tạm không dịch—phân biệt author-time & run-time), làm cách nào có thể có cách để "chỉnh sửa" (ăn gian) phạm vi từ vựng tại run-time?

JavaScript có hai cơ chế như vậy. Cả hai đều khó chịu như nhau như là một trải nghiệm xấu trong code của bạn. Nhưng những lập luận phản đối đều thường thiếu điểm quan trọng nhất: ăn gian phạm vi từ vựng dẫn đến hiệu suất nghèo nàn.

Trước khi tôi giải thích về vấn đề hiệu suất, hãy xem hai cơ chế đó như thế nào.

eval

Hàm `eval(..)` trong JS lấy một chuỗi như là một đối số và xử lý các nội dung của chuỗi như thể nó thực sự là tác giả code (authored) tại thời điểm đó. Nói cách khác, bạn có thể lập trình tạo code bên trong authored code, và chạy code mới tạo đó như thể là nó đã ở đó tại thời điểm author time.

Đánh giá (chơi chữ) `eval(..)` ở đây, cần làm rõ `eval(..)` cho phép bạn sửa đổi môi trường phạm vi từ vựng bằng cách đánh lừa và giả vờ rằng author-time code đó đã tồn tại như thế nào.

Trên các dòng mã tiếp theo sau khi `eval(..)` được thực thi, *Engine* sẽ không biết hoặc quan tâm rằng code trước đó đang được diễn dịch động và do đó đã thay đổi môi trường phạm vi

từ vựng. *Engine* chỉ đơn giản thực hiện tra cứu phạm vi từ vựng theo cách nó luôn hoạt động.

Xem ví dụ sau:

```
function foo(str, a) {

    eval( str ); // ăn gian!

    console.log( a, b );

}
var b = 2;
foo( "var b = 3;", 1 ); // 1, 3
```

Chuỗi `"var b = 3;"` được xử lý như là nó đã tồn tại tại hàm `eval(..)`. Bởi vì code đó xảy ra để khai báo biến `b` mới, nó sửa đổi phạm vi từ vựng hiện tại của `foo(..)`. Thực tế như đã đề cập ở trên thì code đã tạo biến `b` trong `foo(..)` đã che bóng `b` khai báo ở phạm vi bên ngoài (toàn cục).

Khi thực thi `console.log(..)`, nó tìm cả `a` và `b` trong phạm vi của `foo(..)`, và không bao giờ tìm `b` bên ngoài. Vì vậy, chúng ta in ra "1, 3" thay vì "1, 2" như trường hợp thông thường.

Ghi chú: Trong ví dụ này, cho đơn giản, chuỗi của “code” chúng ta truyền vào là một chuỗi trực kiện cố định. Nhưng nó có thể dễ dàng tạo ra bằng cách thêm các ký tự dựa trên logic chương trình. `eval(..)` thường được sử dụng để thực thi tạo code động, nhưng việc định lượng động code tĩnh từ một chuỗi trực kiện thường không mang một lợi ích thực tế nào thay cho chỉ cần tạo code trực tiếp.

Mặc định một chuỗi code mà `eval(..)` thực thi chứa một hay nhiều khai báo (bao gồm biến và hàm), hoạt động này sửa đổi phạm vi từ vựng hiện hữu tại nơi `eval(..)` tồn tại. Về mặt kỹ thuật, `eval(..)` có thể được gọi "gián tiếp" bằng một vài thủ thuật (ngoài phạm trù thảo luận) làm cho nó thay vì thực thi ở bối cảnh toàn cục, thì lại sửa đổi nó. Nhưng trong cả hai trường hợp, `eval(..)` có thể sửa đổi theo runtime một phạm vi từ vựng author-time.

Ghi chú: `eval(..)` khi được sử dụng ở strict-mode nó hoạt động trong phạm vi từ vựng của chính nó, nghĩa là các khai báo được tạo bên trong `eval()` không thực sự sửa đổi phạm vi.

```
function foo(str) {
```

```
  "use strict";

  eval( str );

  console.log( a ); // ReferenceError: a không xác định

}
foo( "var a = 2" );
```

Có một điều kiện dễ dàng khác trong JavaScript mang lại hiệu ứng tương tự như `eval(..)`. `setTimeout(..)` và `setInterval(..)` *có thể* lấy một chuỗi cho đối số đầu tiên của nó, nội dung được định lượng giống như code của hàm tự tạo ra. Đừng sử dụng cách này, từ lâu người ta đã không tán thành.

Phương thức khởi tạo `new Function(..)` cũng lấy một chuỗi của code trong đối số cuối trong nó để biến thành một hàm tự tự động (các đối số đầu tiên), nếu có,là các tham số cho hàm mới). Cú pháp hàm khởi tạo này an toàn hơn `eval(..)`, nhưng cũng nên tránh sử dụng.

Việc sử dụng tự tạo code động trong chương trình thực sự rất hiếm, vì nó làm xuống cấp hiệu suất và hầu như không bao giờ đáng.

with

Tính năng khác nữa (đã hết sử dụng) trong JS để lừa phạm vi từ vựng là `with`. Có vài cách hợp lệ để giải thích cho `with`, nhưng tôi chọn hướng giải thích theo góc độ cách nó tương tác và ảnh hưởng đến phạm vi từ vựng.

`with` được giải thích như là một cách ngắn gọn để tạo nhiều tham chiếu đối với một object mà bản thân nó không phải lặp lại tham chiếu object nhiều lần.

Ví dụ:

```
var obj = {

	a: 1,

	b: 2,

	c: 3

};
// sự lặp lại "obj" một cách "tẻ nhạt"
obj.a = 2;
obj.b = 3;
obj.c = 4;
// cách giản tiện hơn
```

```
with (obj) {
	a = 3;
	b = 4;
	c = 5;
}
```

Tuy nhiên, có nhiều thứ xảy ra hơn đơn giản chỉ là cách giản tiện cho cách try cập object:

```
function foo(obj) {

	with (obj) {

		a = 2;

	}

}
var o1 = {
	a: 3
};
var o2 = {
	b: 3
};
foo( o1 );
console.log( o1.a ); // 2
foo( o2 );
console.log( o2.a ); // không xác định
console.log( a ); // 2 -- Oops, bị rò ra ngoài toàn cục!
```

Trong ví dụ, hai object `o1` và `o2` được tạo ra. Một có thuộc tính `a`, còn cái khác thì không có. Hàm `foo(...)` lấy tham chiếu `obj` như

là một tham số, và gọi `with (obj) { .. }` về tham chiếu đó. Bên trong khối `with`, ta làm cho những gì hiện hữu trở thành tham chiếu từ vựng thông thường cho biến `a`, tham chiếu LHS (xem Chương 1) xảy ra, gán giá trị `2` cho nó.

Khi chúng ta truyền vào `o1`, phép gán `a = 2` tìm thuộc tính `o1.a` và gán giá trị `2` như phản hồi của `console.log(o1.a)`. Tuy nhiên, khi chúng ta truyền vào `o2`, nó không có thuộc tính `a`, chẳng có gì được tạo ra nên `o2.a` giữ nguyên `undefined`.

Nhưng chúng cần lưu ý một hiệu ứng phụ đặc biệt, đó là biến toàn cục `a`được tạo bởi phép gán `a = 2`. Sao lại như vậy?

Lệnh `with` lấy một object không hoặc có các thuộc tính, và **xử lý object như thể nó là một phạm vi từ vựng riêng biệt hoàn toàn**, và do đó các thuộc tính của object được coi như là các định danh từ vựng trong "phạm vi".

Ghi chú: Mặc dù một khối `with` xử lý đối tượng như một phạm vi từ vựng, khai báo `var` thông thường bên trong `with` sẽ không được mở rộng đến khối `with`, thay vào đó là chứa phạm vi hàm.

Trong khi hàm `eval(..)` có thể thay đổi phạm vi từ vựng hiện tại nếu nó có một chuỗi của code với một hay nhiều khai báo

bên trong, lệnh `with` lại thực sự tạo ra **một phạm vi hoàn toàn mới** từ object mà bạn đưa vào nó.

Hiểu theo cách này, “phạm vi” được khia báo bởi lệnh `with` khi ta đưa vào `o1` là `o1`, và "phạm vi" này có "nhận diện" trong đó tương ứng thuộc tính `o1.a`. Nhưng khi chúng ta dùng `o2` như là "phạm vi", nó không có nhận diện `a`bên trong, và nguyên tắc của tra cứu nhận diện LHS được thực hiện.

Không phải “phạm vi” của `o2`, hay phạm vi của `foo(..)`, hay kể cả phạm vi toàn cục, có nhận diện `a` được tìm thấy, vì vậy khi `a = 2` được thực thi, kết quả của nó là sự tự tạo của toàn cục (dùng ở non-strict mode).

Muốn thấy `with` chuyển hướng một object và thuộc tính của nó thành một phạm vi với các nhận diện tại runtime là kiểu suy nghĩ lệch lạc lạ lùng. Nhưng nó là cái giải thích rõ ràng nhất mà tôi có thể trình bày kết quả.

Ghi chú: Cộng thêm đây là ý tưởng tồi để sử dụng, cả `eval(..)` và `with`đều bị ảnh hưởng (cản trở) bởi Strict Mode. `with` thì hoàn toàn không được phép, trong khi một vài dạng gián tiếp hay không an toàn như `eval(..)` không được phép khi giữ chức năng cốt lõi.

Hiệu suất

Cả `eval(..)` và `with` gian lận để author-time làm trái việc xác định phạm vi từ vựng bằng cách sửa đổi hay tạo phạm vi từ vựng mới tại runtime.

Nếu bạn hỏi, có lợi hại gì? nếu người cho thêm nhiều tính năng phức tạp và linh hoạt, chẳng phải là tốt hay sao? **Không**

JavaScript *Engine* có một lượng tối ưu hiệu suất được thực hiện trong suốt quá trình biên dịch. Một số đã giản gọn để có thể phân tích tĩnh cần thiết và xác định trước chỗ tất cả các biến và hàm khai báo, để nó có thể giảm tải việc giải quyết các định danh trong quá trình thực thi.

Nhưng nếu *Engine* tìm thấy một `eval(..)` hoặc `with` trong code, về cơ bản nó phải thừa nhận tất cả nhận thức về vị trí định danh không hợp lệ, bởi vì nó không thể biết tại lexing time (tạo từ vựng) có chính xác bao nhiêu code được đưa vào `eval(..)` để sửa đổi phạm vi từ vựng, hoặc nội dung của object bạn có thể chuyển vào `with` để tạo ra phạm vi từ vựng mới.

Nói theo nghĩa bi quan, hầu hết những tối ưu hóa mà nó tạo ra đều vô nghĩa nếu `eval(..)` hay `with` tồn tại, vì nó đơn giản không thực hiện tối ưu hóa nào cả.

Code của bạn sẽ hầu như chắc chắn chạy chậm hơn bởi việc bạn thêm `eval(..)` hay `with` bất kỳ đâu trong code. Dù cho *Engine* có thông minh cỡ nào không thể giới hạn những hiệu ứng phụ của những giả định bi quan đó.

Ôn tập (TL;DR)

Phạm vi từ vựng nghĩa là phạm vi được xác định bởi author-time tại nơi hàm được khai báo. Giai đoạn lexing của biên dịch cần xảy ra để biết tất cả các định danh được khai báo ở đâu và ra sao, và cả dự đoán việc chúng sẽ được tìm như thế nào trong quá trình thực thi.

Hai cơ chế trong JavaScript có thể "đánh lừa" phạm vi từ vựng: `eval(..)`và `with`. Dạng này có thể sửa đổi phạm vi từ vựng (tại runtime) bằng cách đánh giá một chuỗi các mã được khai trong đó, hay là tạo ra toàn bộ phạm vi từ vựng mới tại (runtime) bằng cách hành xử tham chiếu object như là "phạm

vi" và các thuộc tính của object đó như là định danh của phạm vi.

Nhược điểm của cơ chế này là đánh bạnh khả năng thực hiện tối ưu trong quá trình biên dịch của *Engine* liên quan đến phạm vi tra cứu, bởi *Engine*phải giả định rằng tối ưu hóa như vậy sẽ không hợp lệ. Code sẽ chạy chậm hơn khi sử dụng những tính năng này. **Đừng dùng chúng.**

You Don’t Know JS (tiếng Việt)—Phạm vi & Đóng kín—Chương 3: Hàm vs. Block scope

Như chúng ta đã khám phá trong chương 2, scope bao gồm một tập hợp các “bong bóng”, mỗi scope hoạt động như vật chứa trong đó xác định các định danh (biến, function). Các bong bóng tổ hợp (nesting) gọn gàng bên trong bong bóng khác, và tổ hợp này được xác định tại author-time.

Nhưng chính xác là cái gì tạo ra bong bóng mới? Có phải chỉ có hàm? Có cấu trúc nào khác tạo ra các bong bóng trong phạm vi?

Phạm vi từ các Hàm

Câu trả lời phổ biến nhất cho các câu hỏi trên là JavaScript có scope nền function (function-based scope). Nghĩa là, mỗi function bạn khai báo sẽ tự tạo bong bóng, và không có bất kỳ cấu trúc nào khác tự tạo bong bóng. Nếu chúng ta để ý thì điều này không hoàn toàn đúng.

Trước hết, ta hãy khám phá function scope & các hàm ý của nó:

```
function foo(a) {


	var b = 2;
	// some code
	function bar() {
		// ...
	}
	// more code
	var c = 3;
}
```

Trong đoạn trích này, bong bóng phạm vi của `foo(..)` bao gồm nhận dạng `a`, `b`, `c` và `bar`. Một khai báo xuất hiện *ở đâu* trong phạm vi **không quan trọng**, bất kể biến hoặc bong bóng phạm vi hàm chứa nó.

`bar(..)` có bong bóng của riêng nó. Phạm vi toàn cũng chỉ có một định danh gắn liền: `foo`

Bởi vì `a`, `b`, `c`, và `bar` đều thuộc về bong bóng phạm vi của `foo(..)`, nó sẽ không thể tiếp cận bên ngoài `foo(..)`. Dòng code dưới đây đều có kết quả lỗi `ReferenceError`, khi các định danh đều không có sẵn tại phạm vi toàn cục:

```
bar(); // thất bại
console.log( a, b, c ); // cả 3 đều thất bại
```

Tuy vậy, các định danh (`a`, `b`, `c`, `foo`, và `bar`) đều có thể tiếp cận được bên trong `foo(..)`, và cũng hợp lệ bên trong `bar(..)` (giả định không có khai báo che bóng nào bên trong `bar(..)`).

Phạm vi hàm khuyến khích ý tưởng tất cả các biến thuộc về hàm, và có thể sử dụng/tái sử dụng xuyên suốt trong hàm đó (tiếp cận được trong phạm vi lồng nhau). Các thiết kế này khá

là hữu dụng, và mang lại các biến JS sự "linh hoạt" khi lấy giá trị với các kiểu khác nhau khi cần.

Mặt khác nếu bạn không cẩn thận, biến đã tồn tại bên trong phạm vi có thể sẽ dẫn đến kết quả không mong muốn.

Ẩn bên trong phạm vi

Theo cách nghĩ thông thường thì bạn khai báo hàm, viết code bên trong. Nếu nghĩ ngược lại thì sẽ thấy lợi thế: lấy một đoạn code đã viết bất kỳ, bao nó bằng một hàm khai báo mới, nó sẽ tạo hiệu ứng ẩn code.

Kết quả thực tế là tạo một phạm vi bong bóng quanh đoạn code, bất kỳ khai báo nào bên trong được gắn với phạm vi của hàm mới, tức là bạn có thể dấu code bên trong phạm vi của một hàm.

Lý do hữu ích của việc giấu code?

Xuất phát từ nguyên lý thiết kế phần mềm "Nguyên tắc ưu tiên thấp nhất", có khi gọi là "Quyền tối thiểu", hay "Phơi bày thấp nhất", thường dùng trong thiết kế API cho một module/object,

chúng ta chỉ lộ những phần tối thiểu cần thiết và giấu những phần còn lại.

Nguyên tắc này dẫn đến việc lựa chọn phạm vi nào sẽ chứa biến và hàm. Nếu tất cả các biến và hàm đều ở toàn cục, nó sẽ truy cập được ở bất kỳ phạm vi lồng nhau. Tức là vi phạm nguyên tắc “tối thiểu ...”

Ví dụ:

```
function doSomething(a) {

     b = a + doSomethingElse( a * 2 );
     console.log( b * 3 );
}
function doSomethingElse(a) {
     return a - 1;
}
var b;
doSomething( 2 ); // 15
```

Trong đoạn code trên, biến `b` và hàm `doSomethingElse(..)` ẩn đi chi tiết cách `doSomething(..)` hoạt động ra sao. Tạo phạm vi bao vây việc truy cập vào `b` và `doSomethingElse(..)` không chỉ không cần thiết mà còn có thể "nguy hiểm", nó có thể có kết quả không mong muốn, dù vô tình hay cố ý thì điều này cũng vi phạm nguyên tắc tiền giả định của `doSomething(..)`.

Thiết kế hợp lý hơn sẽ ẩn chi tiết bên trong phạm vi của `doSomething(..)`:

```
function doSomething(a) {

     function doSomethingElse(a) {

          return a - 1;

     }
     var b;
     b = a + doSomethingElse( a * 2 );
     console.log( (b * 3) );
}
doSomething( 2 ); // 15
```

Giờ đây, `b` và `doSomethingElse(..)` không thể tiếp cận từ bên ngoài mà chỉ được điều khiển bởi `doSomething(..)`. Chi tiết được thiết kế biệt lập, hoạt động của hàm và kết quả không bị ảnh hưởng, nhưng thiết kế giữ cho chi tiết được ẩn đi, điều này sẽ làm cho ứng dụng tốt hơn.

Tránh trùng lặp

Lợi ích khác của việc ẩn biến và hàm trong phạm vi là tránh được sự trùng lặp không chủ định giữa hai định danh cùng tên

nhưng khác phương thức sử dụng. Việc trùng lặp sẽ ghi đè giá trị không mong muốn.

Ví dụ:

```
function foo() {

     function bar(a) {

          i = 3; // thay đổi `i` bên trong phạm vi vòng lặp
for

          console.log( a + i );

     }
     for (var i=0; i<10; i++) {
          bar( i * 2 ); // oops, lặp vô định!
     }
}
foo();
```

Phép gán `i = 3` bên trong `bar(..)` ghi đè, nhưng `i` lại được khai báo trong `foo(..)` trong vòng lặp for. Trường hợp này kết quả là vòng lặp vô tận, vì `i` được đặt giá trị "cứng" `3` và luôn luôn có giá trị `< 10`.

Phép gán bên trong `bar(..)` cần được khai bái một biến nội bộ để sử dụng, bất kể thên nào được chọn. `var i = 3;`sẽ sửa lỗi (

"biến đổ bóng" `i`). Phương án (không thay thế) khác là chọn định danh khác, chẳng hạn như `var j = 3;`. Nhưng việc thiết kế phần mềm tự nhiên sẽ có tên định danh giống nhau, cho nên việc ẩn khai báo bên trong là cách tốt nhất trong trường hợp này.

"Namespaces" toàn cục

Một ví dụ rõ ràng về việc va chạm biến xảy ra trong phạm vi toàn cục là khi nhiều thư viện nạp vào chương trình có thể dễ dàng va chạm nhau nếu bạn không ẩn biến và hàm.

Các thư viện như vậy thường tạo ra một biến duy nhất, thường là object, với một tên duy nhất trong toàn cục. Object này sử dụng như là một "không gian tên" cho thư viện đó, các chức năng phơi bày như các thuộc tính của object.

Ví dụ:

```
var MyReallyCoolLibrary = {

    awesome: "stuff",

    doSomething: function() {

        // ...
```

```
        },

        doAnotherThing: function() {

            // ...

        }

};
```

Quản lý module

Phương án khác hiện đại hơn để tránh va chạm là "module", sử dụng bất kỳ quản lý phụ thuộc. Theo cách này thì không có thư viện nào thêm vào định danh toàn cục, thay vào đó là phải nhập các nhận dạng vào một phạm vi cụ thể thông qua cơ chế khác nhau của bộ phận quả lý module phụ thuộc (dependecy manager).

Cần lưu ý là các công cụ này không có phương thức "thần thánh" nào để miễn quy tắc phạm vi từ vựng. Nó chỉ đơn giản sử dụng quy tắc đã giải thích là không có bất kỳ định danh chung nào trong phạm vi, thay vào đó là giữ cá thể, không va chạm, ngăn ngừa bất kỳ sự va chạm vô tình nào.

Xem Chương 5 để biết thêm về module pattern.

Hàm đóng vai trò phạm vi

Chúng ta đã thấy rằng có thể lấy bất kỳ đoạn code và bao nó bằng một hàm, điều này tạo ra hiệu quả “giấu” bất kỳ biến hoặc hàm đã khai báo với phía bên ngoài phạm vi hoặc hàm khác nằm bên trong phạm vi.

```
var a = 2;
function foo() { // <-- chèn
     var a = 3;
     console.log( a ); // 3
} // <--
foo(); // <-- và
console.log( a ); // 2
```

Đầu tiên là chúng ta phải khai báo một tên hàm `foo()`, nghĩa là bản thân định danh `foo` "làm bẩn" phạm vi bên trong, chúng ta đồng thời gọi tên hàm để nó thực thi.

Lý tưởng hơn nếu hàm không cần tên (hoặc tên không làm dơ phạm vi bên trong), và nếu hàm tự động thực thi.

May mắn là JavaScript có cả giải pháp cho hai vấn đề.

```
var a = 2;
(function foo(){ // <-- thêm cái này
     var a = 3;
     console.log( a ); // 3
})(); // <-- và này
console.log( a ); // 2
```

Phân tích.

Đầu tiên, để ý rằng bao lệnh hàm được bắt đầu bằng `(function...` đối lập với `function...`. Trong khi nhìn nó là chi tiết phụ nhưng nó lại là thay đổi chính. Thay vì ứng xử hàm như một khai báo chuẩn, thì hàm lại coi như là một biểu thức của hàm.

Ghi chú: Cách đơn giản để phân biết khai báo hay biểu thức là vị trí của từ "function" trong câu lệnh (không chỉ khác một dòng, mà là cả câu lệnh). Nếu "function" là cái trước nhất của biểu thức, thì nó là khai báo. Ngược lại, nó là một biểu thức hàm.

Mấu chốt khác biệt mà ta thấy ở đây là một khai báo hàm và biểu thức hàm liên quan đến nơi tên của nó được ràng buộc như một định danh.

So sánh hai đoạn code trên. Nếu đoạn code đầu, cái tên `foo` được ràng buộc với phạm vi của nó, và ta gọi trực tiếp `foo()`. Trong đoạn thứ hai, cái tên `foo` không ràng buộc với phạm vi của nó, mà chỉ ràng buộc với bên trong chính hàm của nó.

Nói khác đi, `(function foo(){ .. })` là một biểu thức nghĩa là định danh `foo` chỉ được tìm thấy trong phạm vi `..` chỉ ra, không phải phạm vi bên ngoài. Ẩn `foo` bên trong chính nó nghĩa là không làm ảnh hưởng phạm vi của nó.

Vô danh vs. Có tên

Bạn đã quen thuộc với biểu thức hàm là tham chiếu callback, như là:

```
setTimeout( function(){

     console.log("I waited 1 second!");

}, 1000 );
```

Cái này được gọi là "biểu thức hàm vô danh", bởi vì `function()...` không có tên định danh. Biểu thức hàm có thể vô danh, nhưng khai báo hàm phải có tên.

Hàm vô danh nhanh và tiện gõ, và nhiều thư viện và công cụ có xu hướng khuyến khích điều này, nhưng có vài vấn đề cần phải nắm rõ:

1. Hàm vô danh sẽ không có tên trong truy dấu, khó debug.
2. Không có tên, nếu hàm muốn tham chiếu đến nó, hoặc đệ quy, ..., **tham chiếu đã bị bỏ** `arguments.callee` lại cần thiết. Ví dụ khác nữa là khi một hàm điều khiển sự kiện cần unbind chính nó sau khi chạy.
3. Hàm vô danh làm code khó đọc hơn. Một cái tên còn chính là document của code.

Biểu thức hàm trực tiếp rất mạnh và hữu dụng—câu hỏi giữa hàm ẩn danh vs. hàm có tên cũng không ảnh hưởng. Đặt tên tất nhiên là tốt hơn, nhưng không có nhược điểm. Dù gì tốt nhất vẫn phải luôn đặt tên hàm:

```
setTimeout( function timeoutHandler(){ // <-- coi nè, tui có tên đó!

    console.log( "I waited 1 second!" );

}, 1000 );
```

Invoking Function Expressions Immediately (IIFE—Gọi biểu thức hàm tức thì)

```
var a = 2;
(function foo(){
     var a = 3;
     console.log( a ); // 3
})();
console.log( a ); // 2
```

Ta có một hàm như một biểu thức được đặt trong `( )`, chúng ta có thể xử lý hàm đó bằng thêm `()` vào phía cuối `(function foo(){ .. })()`. `( )` đằng trước tạo ra một biểu thức cho hàm, và `()` thực thi hàm.

Mẫu này rất thông thường, một vài năm trước cộng đồng đã đồng ý cho cụm từ: IIFE, đại diện cho Immediately Invoked Function Expression.

Đương nhiên, IIFE không cần tên—dạng thông thường của IIFE được sử dụng theo cách vô danh. Việc đặt tên IIFE không phổ biến nhưng lợi ích đối với hàm ẩn danh nói trên thì đây cũng là việc tốt để thực hành.

```
var a = 2;
(function IIFE(){
     var a = 3;
     console.log( a ); // 3
})();
console.log( a ); // 2
```

Có một biến tấu nhỏ trong dạng IIFE truyền thống, một số người thích: `(function(){ .. }())`. Nhìn kỹ để thấy sựa khác biệt. Trong dạng đầu tiên, biểu thức hàm được bao trong `( )`, và sau đó `()` gọi hàm nằm ngay bên ngoài. Trong dạng thứ 2, `()` lại được bỏ vào trong `( )`.

Hai dạng này giống hệt nhau. **Tùy theo phong cách bạn chọn thôi.**

Biến tấu khác của IIFE cũng hay thấy là sử dụng sự kiện trong sự kiện, chỉ gọi hàm, và truyền vào tham số.

Ví dụ:

```
var a = 2;
(function IIFE( global ){
     var a = 3;
     console.log( a ); // 3
     console.log( global.a ); // 2
})( window );
console.log( a ); // 2
```

Ta gọi tham chiếu `window` nhưng đặt tên tham số là `global`, nên ta có một cách mô tả rõ ràng giữa đại diện toàn cục vs. không toàn cục. Đương nhiên bạn có thể truyền bất cứ gì vào phạm vi bên trong bạn cần, và bạn có thể đặt tên tham số bất kỳ. Nó hầu như cũng chỉ là phong cách.

Ứng dụng khác của mẫu này cũng giải quyết một hiệu ứng phụ là định danh `undefined` mặc định có thể có giá trị không hợp lệ ghi đè dẫn đến kết quả không mong muốn. Bằng cách đặt tên tham số `undefined`, nhưng không truyền bất kỳ giá trị đối số nào, ta có thể đảm bảo giá trị không xác định ở trong khối code:

```
undefined = true; // đây là đặt mìn ngầm cho code! cần tránh!
(function IIFE( undefined ){
     var a;
     if (a === undefined) {
          console.log( "Undefined an toàn ở đây!" );
     }
})();
```

Và biến thể khác của IIFE là đảo ngược thứ tự các thứ, hàm có thể thực thi sau khi viện dẫn tham số để truyền vào nó. Mẫu này được sử dụng trong dự án UDM (Universal Module Definition). Dù nó hơi dài dòng, một số người lại thấy vậy lại dễ hiểu.

```
var a = 2;
(function IIFE( def ){
     def( window );
})(function def( global ){
     var a = 3;
     console.log( a ); // 3
     console.log( global.a ); // 2
});
```

Hàm `def` được xác định trong phần thứ hai của đoạn code, và sau đó truyền như một tham số (gọi là `def`) vào hàm `IIFE` ở phần đầu của đoạn code. Cuối cùng, tham số `def` (hàm) được gọi, truyền `window` vào với tham số `global`.

Block scope

Trong khi hàm là đơn vị thường thấy của phạm vi, và quy mô của nó trong thiết kế JS cũng lan rộng khắp chương trình, các đơn vị khác thì lại có thể dẫn đến sự rõ ràng, sạch sẽ để bảo trì code.

Nhiều ngôn ngữ khác JavaScript hỗ trợ Block Scope, và người lập trình các ngôn ngữ đó thường quen với khái niệm này, còn dân chỉ chơi JavaScript có thể thấy khái niệm này hơi lạ lẫm.

Nhưng mặc dù bạn chưa bao giờ viết bất kỳ dòng code nào theo lối block-scoped, bạn vẫn quen với kiểu này như một thành ngữ trong JavaScript:

```
for (var i=0; i<10; i++) {

     console.log( i );

}
```

Ta khai báo biến `i` trực tiếp trong đầu vòng lặp for, bởi vì ý định của chúng ta là chỉ sử dụng `i` cho ngữ cảnh của vòng lặp này, và cơ bản bỏ qua ảnh hưởng của phạm vi ngoài (hàm hay toàn cục).

Đó là tất cả nội dung của block scope: khai báo biến tại nơi nó được sử dụng càng cục bộ càng tốt. Ví dụ khác:

```
var foo = true;
if (foo) {
     var bar = foo * 2;
     bar = something( bar );
     console.log( bar );
}
```

Ta dùng biến `bar` chỉ trong ngữ cảnh của lệnh if, nên nó tạo ra cảm giác rằng ta có thể khai báo nó bên trong khối if. Tuy nhiên, nơi khai báo lại không liên quan khi sử dụng `var`, vì nó luôn thuộc về scope bên trong nó. Đoạn code này cơ bản là "giả" block-scoping, và nên dựa vào việc tự thực thi chứ đừng vô tình sử dụng `bar` ở chỗ khác trong phạm vi.

Block scope là công cụ để mở rộng "Principle of Least Privilege Exposure" (Nguyên tắc tối thiểu) để ẩn thông tin trong hàm thành ẩn thông tin trong khối của code.

Xem lại đoạn code:

```
for (var i=0; i<10; i++) {

     console.log( i );

}
```

Vì sao ảnh hưởng toàn bộ phạm vi hàm với biến `i` lại chỉ sử dụng (và chỉ nên sử dụng) cho vòng lặp for?

Điều quan trọng nhất là nhà phát triển muốn nó tự kiểm để tránh vô tình sử dụng biến ngoài mục đích, chẳng hạn như báo

lỗi biến không xác định nếu bạn sử dụng biến sai chỗ. Block scope cho biến `i` làm cho `i` chỉ khả dụng cho vòng lặp for, sẽ lỗi nếu `i` truy cập chỗ khác trong hàm. Việc này chắc chắn biến không được tái sử dụng nhầm lẫn và khó bảo trì.

Thực tế đáng buồn là xét ở bề mặt thì JavaScript không có cơ sở cho block scope, bạn khai thác thêm mới có.

with

Chúng ta đã tìm hiểu `with` ở Chương 2. Dù nó là một cấu trúc kỳ quặc nhưng nó chính là ví dụ của block scope, trong đó, phạm vi được tào từ object chỉ tồn tại trong vòng đời của `with`, và không ở trong toàn bộ phạm vi.

try/catch

Một chi tiết nhỏ rằng từ JavaScript ES3 đã chỉ định khai báo biến trong mệnh đề `catch` của `try/catch` để block-scoped cho khối `catch`.

Ví dụ:

```
try {
```

```
    undefined(); // thực thi bất hợp lệ nhằm được chấp nhận!
}
catch (err) {
    console.log( err ); // hoạt động!
}
console.log( err ); // ReferenceError: `err` không tìm thấy
```

Như bạn thấy, `err` chỉ tồn tại trong mệnh đề `catch`, và báo lỗi khi bạn muốn thao chiếu nó đâu đó.

Ghi chú: Trong khi hành vi này được xác định và đúng với tất cả môi trường JS tiêu chuẩn (có thể ngoại trừ một số trình duyệt IE cũ), nhiều linter có vẻ vẫn khó chịu với nhiều hơn hai mệnh đề `catch` trong cùng một phạm vi mà mỗi khai báo biến lỗi cùng với tên định danh, mặc dù việc này không cần định nghĩa lại vì các biến đã block-scoped an toàn.

Để tránh những cảnh báo không cần thiết, một số lập trình viên sẽ đặt tên biến `catch` với `err1`, `err2`... Một số khác chỉ đơn giản tắt báo trùng tên biên của linter.

Bản chất của `catch` block-scoping trông có vẻ vô dụng, nhưng trong Phụ Lục B sẽ giải thích vì sao nó hữu dụng.

let

Chúng ta đã thấy JavaScript cũng chỉ có vài hành vi lạ phơi bày chức năng block scope. Nếu đó là những gì ta có (điều đã xảy ra trong nhiều năm), thì block scoping chẳng có lợi ích cho lập trình viên JavaScript.

May mắn là ES6 đã thay đổi điều này, từ khóa `let` ra như một cách khai báo biến khác bên cạnh `var`.

Từ khóa `let` gắn liền việc khai báo với phạm vi của bất kỳ khối nào (thường là trong `{ .. }`) chứa nó.

```
var foo = true;
if (foo) {
     let bar = foo * 2;
     bar = something( bar );
     console.log( bar );
}
console.log( bar ); // ReferenceError
```

Sử dụng `let` để gắn một biến vào một block hiện hữu có gì đó hơi ngầm. Nó có thể làm bạn nhầm nếu bạn không để ý block

nào có biến nào suốt quá trình phát triển code bằng việc di chuyển block, bao nó trong block khác...

Tạo các block biệt lập cho block-scoping có thể giải quyết một số mối lo, cho thấy rõ nó có được gắn liền hay không. Thông thường, đoạn code ngầm được ưa dùng hơn code biệt lập, nhưng kiểu tách block-scoping này dễ diễn đạt, và tự nhiên phù hợp hơn cách block-scoping hoạt động như trong các ngôn ngữ khác:

```
var foo = true;
if (foo) {
	{ // <-- khối biệt lập
		let bar = foo * 2;
		bar = something( bar );
		console.log( bar );
	}
}
console.log( bar ); // ReferenceError
```

Ta có thể tạo một block ngẫu nhiên cho `let` đơn giản bằng cách thêm cặp `{ .. }` bất cứ chỗ nào trong một cú pháp hợp lệ. Trong trường hợp này, bạn tạo một block biệt lập trong lệnh if, nó giúp dễ dàng hơn khi di chuyển trong quá trình refactor mà không ảnh hưởng vị trí và ngữ nghĩa của lệnh if đi kèm.

Ghi chú: Cách khác để diễn đạt block scope rõ ràng có thể xem phụ lục B.

Trong Chương 4, ta sẽ tìm hiểu hoisting, việc khai báo được đưa ra trước cho toàn bộ phạm vi chứa nó.

Tuy nhiên, khai báo được tạo ra với `let` sẽ *không* đưa lên trong toàn bộ block nó xuất hiện. Bởi việc khai báo sẽ không "tồn tại" cho đến khi có biểu thức khai báo.

```
{

  console.log( bar ); // ReferenceError!

  let bar = 2;

}
```

Gom rác

Lý do khác cho block-scoping là sự hữu ích liên quan đến closures và gom rác để lấy lại bộ nhớ. Tôi sẽ minh họa ngắn ở đây, cơ chế closure sẽ được giải thích chi tiết trong Chương 5.

```
function process(data) {
```

```
    // làm gì đó thú vị

}
var someReallyBigData = { .. };
process( someReallyBigData );
var btn = document.getElementById( "my_button" );
btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

Hàm callback điều khiển `click` không cần biến `someReallyBigData` gì hết. Nghĩa là về mặt lý thuyết, sau khi `process(..)` chạy, cấu trúc dữ liệu hao bộ nhớ được gom lại. Tuy nhiên, nó kiểu giống như JS engine vẫn giữ cấu trúc đâu đó, khi hàm `click` có một closure trên toàn bộ scope.

Block-scoping làm cho engine hiểu rõ nó không cần giữ `someReallyBigData`xung quanh:

```
function process(data) {

    // làm gì đó

}
// bất kỳ khai báo trong khối này sẽ "ra đi" sau đó!
{
    let someReallyBigData = { .. };
    process( someReallyBigData );
}
```

```
var btn = document.getElementById( "my_button" );
btn.addEventListener( "click", function click(evt){
     console.log("button clicked");
}, /*capturingPhase=*/false );
```

Khai báo các khối riêng biệt cho biến ràng buộc cục bộ là một công cụ mạnh mẽ.

Vòng lặp let

Một trường hợp đặc biệt mà `let` tỏa sáng là trong vòng lặp for.

```
for (let i=0; i<10; i++) {

     console.log( i );

}
console.log( i ); // ReferenceError
```

Không chỉ có `let` trong đầu vòng lặp ràng buộc `i` vào thân của vòng lặp, mà còn tái ráng buộc nó vào mỗi lần lặp, đảm bảo gán lại giá trị của nó từ cuối của lần lặp trước.

Đây là cách khác để minh họa hành vi mỗi lần lặp:

```
{

     let j;
```

```
	for (j=0; j<10; j++) {

		let i = j; // chuyển qua mỗi lần lặp!

		console.log( i );

	}

}
```

Lý do tại sao ràng buộc từng lần lặp lại hay sẽ thảo luận rõ ở Chương 5.

Vì khai báo `let` đính kèm khối tùy ý chứ không phải phạm vi hàm bao quanh (hay toàn cục), nó có thể tạo được tại nơi code hiện hữu có một khai báo ẩn `var` trong function scoped, việc thay thế `var` bằng `let` có thể cần chú ý khi refactor code.

ví dụ:

```
var foo = true, baz = 10;
if (foo) {
	var bar = 3;
	if (baz > bar) {
		console.log( baz );
	}
	// ...
}
```

Code này có thể dễ dàng refactor như sau:

```
var foo = true, baz = 10;
if (foo) {
     var bar = 3;
     // ...
}
if (baz > bar) {
     console.log( baz );
}
```

Nhưng cẩn thận với các thay đổi khi sử dụng biến trong block scope:

```
var foo = true, baz = 10;
if (foo) {
     let bar = 3;
     if (baz > bar) { // <-- đừng quên `bar` khi thay đổi!
          console.log( baz );
     }
}
```

Xem phụ lục B cho kiểu khác (cụ thể hơn) của block-scoping, có thể cung cấp một hướng để bảo trì/refactor code dễ dàng hơn.

const

Bên cạnh `let`, ES6 cũng giới thiệu `const`, cũng là tạo ra biến trong block-scoped, nhưng giá trị được cố định (constant - hằng số). Bất kỳ ý định thay đổi giá trị của nó sẽ bị lỗi.

```
var foo = true;
if (foo) {
     var a = 2;
     const b = 3; // block-scoped trong mệnh đề `if`
     a = 3; // just fine!
     b = 4; // error!
}
console.log( a ); // 3
console.log( b ); // ReferenceError!
```

Ôn tập (TL;DR)

Hàm là một trong những đơn vị thông thường của phạm vi (scope) trong JavaScript. Biến và hàm được khai báo bên trong hàm khác sẽ "ẩn" đối với "scope" bao ngoài nó, đây là một nguyên tắc thiết kế phần mềm tốt.

Nhưng hàm không phải là đơn vị duy nhất của scope. Block-scope đề cập đến ý tưởng rằng biến và hàm có thể thuộc một block bất kỳ (chính xác là `{...}`) chứ không chỉ là hàm trên nó.

Bắt đầu từ ES3, cấu trúc `try/catch` có block-scope trong mệnh đề `catch`.

Trong ES6, từ khóa `let` (bà con của `var`) được giới thiệu cho phép khai báo biến trong block bất kỳ. `if (..) { let a = 2; }` sẽ khai báo biến `a`rằng nó gắn liền chính nó trong `{...}` của mệnh đề `if`.

Mặc dù vậy, block scope cũng không nên được thực hiện như là thay thế hoàn toàn cho `var` trong phạm vi hàm. Cả hai đều có thể cùng tồn tại, và người lập trình có thể sử dụng cả kỹ thuật function-scope và block-scope để phù hợp với việc bảo trì, đọc code.

You Don’t Know JS (tiếng Việt)—Scope & closure—Chương 4: Hoisting

O'REILLY®

"Kyle's way of critically thinking about every tiny bit of the language will creep into your mindset and general workflow."
—**SHANE HUDSON**, freelance frontend website developer

KYLE SIMPSON

SCOPE & CLOSURES

YOU DON'T KNOW JS

Đọc tới chương này là coi như bạn đã thoải mái với phạm vi, cách biến được gắn với các phạm vi ở tầng khác nhau, cách nó được khai báo. Cả phạm vi hàm (function scope) và phạm vi khối (block scope) đều có chung nguyên tắc: biến nào khai báo ở phạm vi nào thì thuộc về phạm vi đó.

Có một chi tiết tinh tế về cách thức đính kèm scope hoạt ra sao với khai báo xuất hiện ở các vị trí khác trong scope.

Gà hay trứng?

Có thể bạn nghĩ rằng JavaScript được thông dịch theo từng dòng, từ trên xuống dưới khi thực thi chương trình. Đúng là vậy, nhưng lại có một phần giả định đó sẽ dẫn đến suy nghĩ sai.

Xem đoạn code sau:

```
a = 2;
var a;
console.log( a );
```

Bạn nghĩ nó sẽ log ra cái gì trong lệnh `console.log(..)`?

Nhiều lập trình viên sẽ nghĩ là `undefined` vì `var a` xuất hiện sau `a = 2`, và đương nhiên biến đó sẽ được định nghĩa lại theo tự nhiên, và sẽ là `undefined`. Nhưng kết quả là `2`.

Xem đoạn code dưới đây:

```
console.log( a );
var a = 2;
```

Dựa theo đoạn code ở trên kia thì có thể bạn sẽ nghĩ rằng `2` sẽ được in ra, kiểu như hành vi tìm kiếm từ trên xuống được tối thiểu hóa. Một số người thì nghĩ rằng `a` được gọi trước khi khai báo, nên kết quả chắc chắn là `ReferenceError`.

Không may, cả hai đều sai, kết quả là `undefined`.

Vậy chuyện gì đã xảy ra? Câu chuyện gà và trứng xuất hiện. Cái nào có trước? Trứng hay gà?

Phản biện của trình biên dịch

Để trả lời câu hỏi này, chúng ta cần ngược lại Chương 1, và câu chuyện trình biên dịch của chúng ta. Gọi lại *Engine* sẽ biên dịch code trước khi diễn giải nó. Một phần của giai đoạn biên dịch

là tìm và kết hợp với tất cả các khai báo trong phạm vi phù hợp. Chương 2 đã chỉ cho ta thấy đó chính là trái tim của lexical scope (phạm vi từ vựng).

Vì vậy, cách tốt nhất để nghĩ về hoisting là: các khai báo biến và hàm đó được xử lý trước khi thực thi.

Khi bạn thấy `var a = 2;`, bạn có thể nghĩ nó là một câu lệnh. Nhưng thực tế thì JavaScript hiểu nó là hai lệnh: `var a;` và `a = 2;`. Câu lệnh đầu tiên, khai báo sẽ được xử lý trong giai đoạn dịch. Câu lệnh thứ hai, phép gán , được đặt đúng chỗ cho giai đoạn thực thi.

Đoạn code đầu của chúng ta có thể được viết như thế này:

```
var a;
a = 2;
console.log( a );
```

...phần đầu là biên dịch và phần sau là thực thi.

Tương tự, đoạn code thứ 2 có thể ghi như sau:

```
var a;
console.log( a );
a = 2;
```

Do đó, một cách suy nghĩ về tiến trình này theo cách ẩn dụ rằng khai báo biến và hàm được “chuyển” đến nơi chúng xuất hiện trong luồng code đến đầu code. Điều này dẫn đến cái tên “hoisting”

Nói cách khác, **trứng có trước gà**.

Ghi chú: Chỉ có khai báo tự nó được hoist, trong khi các phép gán hay logic thực thi khác được đặt vào chỗ. Nếu hoisting tự sắp xếp logic thực thi cho code, nó sẽ dẫn đến sự tàn phá.

```
foo();
function foo() {
     console.log( a ); // undefined
     var a = 2;
}
```

Khai báo hàm `foo` được hosting (trường hợp này bao gồm giá trị ngụ ý của nó như một hàm thực tế), sao cho việc gọi hàm ở dòng đầu tiên có thể thực hiện được.

Đồng thời cũng rất quan trọng để chú ý rằng hoisting theo từng scope. Đoạn code trước của ta đơn giản là nó một scope toàn cục, hàm `foo(..)` mà chúng ta thực hiện ở đây chính nó cho thấy

`var a` được chuyển lên trên cùng của `foo(..)`. Vì vậy chương trình có lẽ giải thích chính xác như sau:

```
function foo() {

	var a;
	console.log( a ); // undefined
	a = 2;
}
foo();
```

Khai báo hàm được đưa lên trên nhưng biểu thức hàm thì không.

```
foo(); // not ReferenceError, but TypeError!
var foo = function bar() {
	// ...
};
```

Định danh biến `foo` được đưa lên và gắn với scope của toàn bộ chương trình (toàn cục), vì vậy `foo()` không thất bại với lỗi `ReferenceError`. Nhưng `foo` lại chưa có giá trị nào (mà nó phải có vì nó là khai báo hàm chứ không phải biểu thức). Vì vậy, khi `foo()` có gắng tìm giá trị `undefined`, thì lại bị `TypeError`.

Và cũng nhắc lại rằng cho dù là một biểu thức hàm được đặt tên, tên định danh của nó cũng không có trong toàn bộ scope:

```
foo(); // TypeError


bar(); // ReferenceError
var foo = function bar() {
     // ...
};
```

Đoạn này cần diễn đạt chính xác (có hoisting) hơn như sau:

```
var foo;
foo(); // TypeError
bar(); // ReferenceError
foo = function() {
     var bar = ...self...
     // ...
}
```

Hàm trước

Cả khai báo và biến đều được hoist. Nhưng có một chi tiết tinh vi (có thể xuất hiện trong mã với nhiều khai báo "trùng lặp") rằng hàm được đưa lên trước, sau mới tới biến.

Xem:

```
foo(); // 1
var foo;
function foo() {
     console.log( 1 );
}
```

```
foo = function() {
     console.log( 2 );
};
```

`1` được in ra thay vì `2`! Đoạn code này được diễn dịch bởi *Engine* như sau:

```
function foo() {


     console.log( 1 );


}
foo(); // 1
foo = function() {
     console.log( 2 );
};
```

Để ý rằng khai báo `var foo` đã trùng lặp (và bị bỏ qua), mặc dù nó đến trước khai báo `function foo()...`, bởi khia báo hàm được đưa lên trước các biến thông thường.

Trong khi khai báo `var` trùng lặp nhiều lần được bỏ qua thì khai báo hàm tiếp theo lại ghi đè lên hàm trước.

```
foo(); // 3
function foo() {
     console.log( 1 );
}
var foo = function() {
     console.log( 2 );
```

```
};
function foo() {
     console.log( 3 );
}
```

Mặc dù có vẻ nó không có gì thú vị hơn về mặt học thuật, nhưng nó làm nổi bật sự trùng lặp trong cùng một scope và là một ý tồi dẫn đến kết quả khó hiểu.

Khai báo hàm xuất hiện trong block bình thường thường đưa lên trên scope bao nó hơn là trong điều kiện như đoạn code dưới.

```
foo(); // "b"
var a = true;
if (a) {
  function foo() { console.log( "a" ); }
}
else {
  function foo() { console.log( "b" ); }
}
```

Tuy nhiên, cũng quan trọng để chú ý rằng hành vi này không tin cậy và có thể thay đổi trong phiên bản tiếp theo của JavaScript, vì vậy tốt nhất là tránh khia báo chức năng trong khối.

Ôn tập (TL;DR)

Chúng ta có thể thấy `var a = 2;` là một câu lệnh nhưng JavaScript *Engine*lại không. Nó thấy `var a` và `a = 2` như là hai câu lệnh riêng biệt, cái đầu là ở giai đoạn biên dịch và cái thứ hai là giai đoạn thực thi.

Điều này dẫn đến tất cả các khai báo trong scope dù nó ở đâu, cũng được xử lý trước khi đoạn code được thực thi. Bạn có thể hình dung được điều này khi các khai báo (biến và hàm) đừa "dời" lên đầu của phạm vi tương ứng, gọi là "hoisting".

Bản thân khai báo đã được hoist, nhưng phép gán, kể cả gán biểu thức hàm, thì không được hoist.

Cần cẩn thận cho khai báo trùng lặp, đặc biệt là sự pha trộn giữa khai báo var thông thường và khai báo hàm,nguy hiểm luôn rình rập.

You Don't Know JS (tiếng Việt) — Scope & closure — Chương 5: Scope Closure

O'REILLY®

"Kyle's way of critically thinking about every tiny bit of the language will creep into your mindset and general workflow."
—**SHANE HUDSON**, freelance frontend website developer

KYLE SIMPSON

SCOPE & CLOSURES

YOU DON'T KNOW JS

Tới chương này thì ta đã nắm rõ scope hoạt động như thế nào rồi.

Giờ chúng ta chuyển qua chú ý đến một vấn đề vô cùng quan trọng, nhưng cũng rất khó nắm bắt, là một phần của ngôn ngữ và *gần như là truyền thuyết*: **closure**. Theo như những gì ta tìm hiểu về lexical scope cho đến nay, thì lợi ích của closure là hiển nhiên. Nhưng *có một người đằng sau bức màn bí mật, chúng ta sẽ phải tìm anh ta*. Không, anh ta không phải Crockford đâu(cha đẻ JS—người dịch)!

Tuy nhiên, nếu bạn còn thắc mắc về lexical scope, tốt nhất là nên quay về Chương 2 trước khi tiếp tục.

Khai sáng

Với những người có kinh nghiệm với JavaScript, nhưng có lẽ chưa bao giờ nắm bắt đầy đủ khái niệm của closure, việc *hiểu closure* có thể coi như đạt cảnh giới niết bàn cần phải có sự phấn đấu và hy sinh mới đạt được.

Tôi nhớ vài năm trước, khi tôi đã nắm vững JavaScript, nhưng vẫn không hiểu closure là gì. Có một gợi ý châm chọc rằng ở

phía bên kia của ngôn ngữ, có nhiều hứa hẹn hơn những gì tôi có. Tôi nhớ tôi đã đọc hết source code của một framework và cố gắng hiểu nó hoạt động ra sao. Tôi nhớ lần đầu tiên cái gì đó như "module pattern" đã gợi lên trong tâm trí. Tôi nhớ những khoảnh khắc *a-ha!*.

Những gì tôi đã không biết sau đó, cái gì đã khiến tôi cả năm để hiểu, và những gì tôi hy vọng truyền đặt được cho bạn là bí mất: **closure luôn ở xung quanh JavaScript, bạn phải nhận ra và nắm bắt lấy nó**. Closure không phải công cụ đặc biệt mà bạn phải học thêm về cú pháp và pattern. Không, closure thậm chí cũng không phải vũ khí mà bạn phải học cách làm chủ như Luke đã luyện trong The Force (xem Star wars—người dịch).

Closure xảy ra như là kết quả của viết code dựa trên lexical scope. Đôi khi là nó chỉ xảy ra. Thậm chí bạn còn không thực sự có ý định tạo closure để tận dụng lợi thế của chúng. Closure được tạo ra suốt quá trình code. Những gì bạn *đang thiếu* là bối cảnh để nhận ra, nắm bắt và dùng như đòn bẩy cho ý riêng.

Giây phút khai sáng hẳn là: **oh, closure thực sự xuất hiện suốt trong code của mình, giờ mình có thể thấy chúng**. Hiểu closure giống như Neo thấy Ma trận lần đầu.

Thực chất của vấn đề

OK, tán phét đủ rồi.

Giờ là những gì bạn cần để biết, hiểu, nắm bắt closure:

Closure là khi một hàm có khả năng nhớ và truy cập lexical scope của nó ngay cả khi hàm đó được thực thi bên ngoài lexical scope của nó.

Hãy xem vài đoạn code minh họa.

```
function foo() {


     var a = 2;
function bar() {
          console.log( a ); // 2
     }
bar();
}
foo();
```

Đoạn code này nhìn tương tự như trong phần Nested Scope (phạm vi lồng nhau). Hàm `bar()` *truy cập* đến biến `a` ở trong phạm vi bao ngoài vì quy tắc tìm kiếm lexical scope (trường hợp này là một tìm kiếm RHS)

Đây phải "closure" không?

Chà, về kỹ thuật thì... *có lẽ*. Nhưng những gì bạn cần biết như ở trên thì...*không chính xác*. Tôi nghĩ cách chính xác nhất để giải thích `bar()` tham chiếu `a` thông qua quy tắc tìm kiếm lexical scope, và các quy tắc (quan trọng!) đó *chỉ* là **một phần** của closure.

Từ góc độ hàn lâm thuần túy, hàm `bar()` ở trên được giải thích là nó có *closure* trong phạm vi của `foo()` (và thậm chí thực ra là trong phần con lại của scope nó có quyền truy cập đến, phạm vi toàn cục chẳng hạn). Còn nói hơi khác thì `bar()` đóng kín trong phạm vi của `foo()`. Vì sao? Vì `bar()`xuất hiện lồng bên trong `foo()`, đơn giản thẳng đuột ruột ngựa.

Nhưng, xác định closure theo cách này không trực tiếp *observable (quan sát được)*, và chúng ta cũng không thấy

closure *thể hiện* gì trong đoạn code trên. Ta thấy rõ lexical scope, nhưng closure vẫn là gì đó ẩn sâu bên trong.

Hãy xem đoạn code mang closure ra ngoài ánh sáng:

```
function foo() {

     var a = 2;
function bar() {
          console.log( a );
     }
return bar; // lấy hàm bar như một giá trị và trả về cho hàm
foo
}
var baz = foo();
baz(); // 2 -- Whoa, thấy closure rồi nha các mẹ.
```

Hàm `bar()` có lexical scope truy cập vào scope của `foo()`. Nhưng sau đó, ta lấy chính bản thân hàm `bar()`, và truyền nó đi như một giá trị. Trường hợp này, ta `return` chính hàm `bar` như một tham chiếu.

Sau khi thực thi `foo()`, chúng ta gán giá trị trả về (hàm `bar()` bên trong) cho một biến gọi là `baz`, sau đó chúng ta gọi `baz()`, và dĩ nhiên nó gọi hàm bên trong `bar()`, chẳng qua là theo cách nhận diện tham chiếu khác mà thôi.

Chắc chắn `bar()` được thực thi. Nhưng trong trường hợp này, nó lại thực thi *bên ngoài* lexical scope mà nó đã khai báo.

Sau khi `foo()` thực thi, thông thường chúng ta cho rằng toàn bộ scope bên trong của `foo()` sẽ ra đi, vì *Engine* sử dụng công cụ *Gom rác* đi kèm và giải phóng bộ nhớ khi nó không còn sử dụng. Vì dường như nội dung của `foo()` đã không còn sử dụng, nó có thể coi là *mất*.

Nhưng "điều kỳ diệu" của closusre không để điều này xảy ra. Nghĩa là phần bên trong scope vẫn đang "sử dụng", không đi đâu hết. Ai dùng nó? **Chính hàm `bar()`.**

Vì `bar()` đã được khai báo nên nó có lexical scope closure qua phạm vi bên trong của `foo()`, việc này đã giúp cho `bar()` tồn tại trong việc tham chiếu về sau.

`bar()` vẫn có một tham chiếu đến phạm vi, và điều này được gọi là closure.

Vì vậy, vài micro giây sau đó, khi biến `baz` được gọi (gọi hàm `bar` bên trong), nó có quyền truy cập đến author-time của lexical scope, nên nó có thể tiếp cận biến `a` như ta mong muốn.

Hàm được gọi ngon lành cành đào từ bên ngoài của author-time lexical scope. **Closure** cho phép hàm tiếp tục truy cập lexical scope đã xác định tại author-time.

Tất nhiên bất kỳ phương cách nào mà hàm có thể *truyền đi xung quanh*như một giá trị, và được viện dẫn ở chỗ khác, đều là mô tả của việc observe/thực hiện closure.

```
function foo() {


     var a = 2;
function baz() {
          console.log( a ); // 2
     }
bar( baz );
}
function bar(fn) {
     fn(); // cha mạ ơi, tui thấy closure !
}
```

Ta truyền hàm `baz` bên trong cho `bar`, và gọi hàm đó (tức là `fn`), và như vậy closure của nó qua phạm vi bên trong `foo()` được observe bằng cách truy cập `a`.

Cách truyền hàm như vậy có thể theo cách gián tiếp.

```
var fn;
function foo() {
```

```
     var a = 2;
function baz() {
          console.log( a );
     }
fn = baz; // gán `baz` cho biến toàn cục
}
function bar() {
     fn(); // closure đây nè!
}
foo();
bar(); // 2
```

Dù chúng ta sử dụng phương tiện gì để *vận chuyển* hàm bên trong ra ngoài lexical scope, nó vẫn giữ một tham chiếu phạm vi tại nơi nó được khai báo, và dù ta thực thi nó ở đâu, closure sẽ được thực hiện.

Giờ tôi đã thấy

Các đoạn code ở trên cũng chỉ để minh họa cấu trúc của việc sử dụng closure. Nhưng tôi hứa là nó còn hơn cả một đứa trẻ có đồ chơi mới. Tôi hứa là closure nó luôn hiện hữu xung quanh code. Hãy xem đoạn code sau:

```
function wait(message) {
setTimeout( function timer(){
          console.log( message );
     }, 1000 );
}
wait( "Hello, closure!" );
```

Ta lấy hàm bên trong (`timer`) và truyền nó vào `setTimeout(..)`. Nhưng `timer` có một phạm vi closure qua `wait()`, và giữ một tham chiếu đến biến `message`.

Một ngàn mili giây sau khi ta thực thi `wait()`, scope bên trong nó sẽ thay vì mất đi, nhưng hàm bên trong `timer` vẫn có closure trong phạm vi.

Đi sâu vào mấu chốt của *Engine*, hàm tiện ích dựng sẵn `setTimeout(..)`tham chiếu đến vài tham số, tạm gọi là `fn`hay `func` hay đại loại tương tự vậy. *Engine* sẽ gọi hàm đó, nghĩa là hàm `timer` bên trong được gọi, và tham chiếu lexical scope vẫn nguyên vẹn.

Closure.

Nếu bạn đã dùng jQuery (hay bất kỳ JS framework nào):

```
function setupBot(name,selector) {

    $( selector ).click( function activator(){

        console.log( "Activating: " + name );
```

```
	} );
}
setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

Tôi không chắc bạn viết code kiểu nào, còn tôi thì thường viết code có trách nhiệm điều khiển toàn bộ "trang bị vũ trang" của closure, nên ví dụ trên là thực tiễn!

Về cơ bản, bất kỳ *khi nào* và *tại đâu* bạn xử lý hàm (hàm truy vập vào lexical scope riêng của chúng) như một giá trị hạng nhất và truyền chúng đâu đó, bạn có thể thấy hàm đó thực hiện closure. Là timers, event handler, gọi Ajax, web worker, cửa sổ thông báo, hay bất kỳ nhiệm vụ động bộ, bất đồng bộ, khi bạn truyền trong một hàm *callback*, tức là đã dùng closure.

Ghi chú: Chương 3 đã giới thiệu mẫu IIFE. Có người thường hay bảo bản thân IIFE là một ví dụ cho thực hiện closure, dựa theo xác định ở trên thì tôi có ý kiến không đồng ý lắm.

```
var a = 2;
(function IIFE(){
	console.log( a );
})();
```

Code này hoạt động, nhưng nó hoàn toàn không phải là một observation của closure. Vì sao? bởi vì hàm `IIFE` không được thực thi bên ngoài lexical scope. Nó vẫn gọi ngay trong cùng một scope mà nó đã được khai báo (scope bên ngoài có biến `a`). `a` được tìm thế theo cách tra cứu lexical scope thông thường, không phải là qua closure.

Về mặt kỹ thuật, trong khi closure xảy ra ở thời điểm khai báo, IIFE không phải như vậy.

Mặc dù bản thân IIFE không phải là ví dụ của closure, nó lại tạo ra scope, và nó là một trong những công cụ thông thường nhất để ta tạo ra scope. Vì vậy IIFE có mối liên hệ mật thiết với closure, mặc dù bản thân nó không thực hiện closure.

Và giờ thì dừng đọc đi mấy bạn, tôi có nhiệm vụ cho bạn đây. Giờ bạn mở code của bạn lên, coi coi có hàm nào là closure hay giá trị là hàm không.

Giờ thì rõ rồi nhé!

Vòng lặp + Closure

Ví dụ điển hình để minh họa closure là vòng lặp for.

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

Ghi chú: Linter (trình dò lỗi JavaScript—người dịch) thường cảnh báo khi bạn đưa hàm vào trong vòng lặp do nhiều lập trình viên chưa nắm được closure. Tôi sẽ giải thích làm thế nào tận dụng toàn bộ sức mạnh của closure ở đây.

Linh hồn của đoạn code trên là điều chúng ta *mong muốn* là các số “1”, “2”, .. “5” sẽ được in ra sau mỗi giây tương ứng.

Khi bạn chạy đoạn code này, bạn sẽ có kết quả là “6” được in ra 5 lần theo mỗi giây.

Hả?

Trước tiên, tôi sẽ giải thích `6` ở đâu ra. Điều kiện chấm dứt vòng lặp khi `i`*không* `<=5`. Trường hợp đầu tiên của việc này là `i` bằng 6. Vì vậy đầu ra phản ánh giá trị cuối cùng của `i` sau khi vòng lặp kết thúc.

Điều này có vẻ rõ ràng, hàm timeout callback đều chạy ngon sau khi hoàn thành vòng lặp. Trong thực tế, khi bộ đếm thực hiện, kể cả nếu `setTimeout(.., 0)` cho mỗi lần lặp, tất cả các hàm callback đó đều chạy đúng sau khi hoàn thành vòng lặp, do đó nó in `6` cho mỗi lần.

Đi vào sâu hơn, code của ta thiếu cái gì mà đáng lẽ nó phải thực hiện đúng như ta muốn?

Cái thiếu là cái chúng ta đang cố thực hiện rằng mỗi vòng lặp bắt 1 bản sao của `i` tại thời điểm lặp. Nhưng theo cách hoạt động của scope, thì ta có 5 hàm được xác định riêng biệt theo mỗi lần lặp, tất cả đều được **đóng kín bởi cùng một scope toàn cục**, và chỉ có một `i` trong nó.

Theo cách này, tất nhiên tất cả các hàm đều có chung một tham chiếu đến cùng một `i`. Có gì đó về cấu trúc vòng lặp làm ta bối rối và nghĩ rằng có gì đó phức tạp, nhưng không phải vậy,

chẳng có gì khác biệt với việc khai báo các hàm callback đó 5 lần tuần tự cái này tiếp cái kia mà không dùng vòng lặp.

Ok, vậy, quay lại câu hỏi của chúng ta. Cái gì thiếu? Ta cần thêm closure scope. Đặc biệt là chúng cần một closure scope mới cho mỗi lần lặp.

Chúng ta đã học trong Chương 3 rằng IIFE tạo scope bằng cách khai báo một hàm và thực thi nó ngay lập tức.

Thử xem:

```
for (var i=1; i<=5; i++) {
    (function(){
        setTimeout( function timer(){
            console.log( i );
        }, i*1000 );
    })();
}
```

Nó có chạy không?

Không. Nhưng tại sao? Rõ ràng là có lexical scope. Mỗi hàm timeout callback đóng qua mỗi lần lặp scope của riêng nó được tạo bởi IIFE tương ứng.

Bởi vì hàm IIFE của ta là một scope rỗng chẳng làm gì cả. Nó cần có gì đó hữu dụng hơn, tức nó cần một biến riêng, là bản sao của giá trị `i` sau mỗi lần lặp.

```
for (var i=1; i<=5; i++) {

     (function(){

          var j = i;

          setTimeout( function timer(){

               console.log( j );

          }, j*1000 );

     })();

}
```

Eureka! Đã chạy!

Phiên bản được ưa thích là:

```
for (var i=1; i<=5; i++) {

     (function(j){

          setTimeout( function timer(){

               console.log( j );

          }, j*1000 );

     })( i );

}
```

Cách dùng IIFE trong mỗi lần lặp tạo ra một scope mới cho mỗi lần lặp, cho phép hàm timeout call back cơ hội đóng thông qua scope mới cho mỗi lần lặp, và nó có biến trong mỗi lần lặp giúp ta try cập.

Vấn đề được giải quyết!

Gặp lại block scoping

Xem xét cẩn thận các phân tích ở giải pháp trên. Chúng ta sử dụng IIFE để tạo ra scope cho mỗi lần lặp. Nói cách khác, chúng ta cần *block-scope* của mỗi mỗi lần lặp. Chương 3 cho ta thấy khai báo `let` chiếm một block và khai báo một biến tại đó.

Cơ bản nó tạo một block trong scope để chúng ta có thể đóng nó. Vì vậy đoạn code dưới đây chạy ngon:

```
for (var i=1; i<=5; i++) {

    let j = i; // vâng, block-scope cho closure!

    setTimeout( function timer(){

        console.log( j );

    }, j*1000 );

}
```

Nhưng chưa hết! Nó có một hành vi đặc biệt được xác định cho khai báo `let` được sử dụng trên đầu của vòng lặp for. Hành vi này cho biết rằng biến sẽ được khai báo không chỉ một lần cho

vòng lặp, mà còn là mỗi lần lặp. Và nó được khởi tạo tại mỗi lần lặp với giá trị từ cuối cho đến lần lặp trước. (mới thấy `var` và `let` khác biệt rõ rệt nhé - người dịch)

```
for (let i=1; i<=5; i++) {

     setTimeout( function timer(){

          console.log( i );

     }, i*1000 );

}
```

Block scoping và closure đã tay trong tay hoạt động cùng nhau, giải quyết mọi thứ trên thế giới. Tôi không biết bạn sao chớ nó làm cho tôi vui với JavaScript.

Module

Có những mẫu code sử dụng sức mạnh của closure nhưng không xuất hiện trên bề mặt mà thường là callback. Hãy kiểm tra một trong những kiểu mạnh nhất trong đó: *mô-đun*.

```
function foo() {
```

```
    var something = "cool";

    var another = [1, 2, 3];
function doSomething() {
        console.log( something );
    }
function doAnother() {
        console.log( another.join( " ! " ) );
    }
}
```

Hiện tại, đoạn code trên chưa có observable closure. Chúng ta chỉ có biến riêng `something`, `another`, và hàm `doSomething()` và `doAnother()` bên trong, cả biến và hàm đều có lexical scope bên trong phạm vi của `foo()`.

```
function CoolModule() {

    var something = "cool";

    var another = [1, 2, 3];
function doSomething() {
        console.log( something );
    }
function doAnother() {
        console.log( another.join( " ! " ) );
    }
return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}
```

```
var foo = CoolModule();
foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

Đây là mẫu trong JavaScript gọi là *module*. Cách thông thường nhất để viết một mẫu module thường gọi là "Revealing Module", cách viết như ở trên.

Hãy kiểm xem có gì trong đoạn code trên.

Trước hết, `CoolModule()` chỉ là một hàm, nhưng nó phải được gọi để khởi tạo một module. Nếu không thực thi hàm bên ngoài, việc tạo scope bên trong và closure sẽ không xảy ra.

Tiếp theo, hàm `CoolModule()` trả về một object, biểu thị bởi cú pháp object-literal `{ key: value, ... }`. Object mà ta trả có tham chiếu trong nó đến các hàm phía trong, giữ cho chúng ẩn và riêng biệt. Bạn nên nghĩ object này trả giá trị như một **API công khai của module**.

Việc trả giá trị cho object cuối cùng gán cho biến bên ngoài `foo`, sau đó ta có thể truy cập những phương thức theo API, ví dụ `foo.doSomething()`.

Ghi chú: Không nhất thiết là ta phải trả một object (literal) thực tế từ module. Ta chỉ cần trả trực tiếp hàm bên trong. jQuery là một ví dụ, dấu `$`là một API công khai của jQuery module, nhưng bản thân nó cũng là một hàm (có thuộc tính, và các hàm đều là object).

Hàm `doSomething()` và `doAnother()` có closure thông qua scope bên trong của module (có được nhờ gọi `CoolModule()`). Khi ta chuyển hàm trong lexical scope ra ngoài, ta thiếu lập một điều kiện rằng closure nào có thể được observe và thực hiện bằng cách tham chiếu vào object chúng ta trả.

Mô tả đơn giản hơn, có hai yêu cầu cho một module pattern thực hiện:

1. Nó phải là hàm bao quanh bên ngoài, và phải được gọi ít nhất một lần (mỗi lần gọi tạo nên một module instance).
2. Hàm bên ngoài phải trả ít nhất một hàm bên trong, nhờ vậy hàm bên trong có closure thông qua scope riêng biệt, và có thể truy cập để thay đổi trạng thái riêng biệt đó.

Bản thân một object với thuộc tính hàm bên trong nó không thực sự là một module. Trong bối cảnh observable, một object được trả từ một hàm chỉ có các thuộc tính dữ liệu bên trong nó và không có hàm closure nào thì *không thực sự* là một module.

Đoạn code ở trên cho thấy một trình tạo module độc lập gọi là `CoolModule()`, có thể gọi bao nhiêu lần cũng được, mỗi lần gọi thì nó tạo một module tức thì. Một thay đổi nhỏ với mẫu này là khi bạn quan tâm đến việc chỉ có một lần tạo, đây là một dạng "sigleton":

```
var foo = (function CoolModule() {

     var something = "cool";

     var another = [1, 2, 3];
function doSomething() {
          console.log( something );
     }
function doAnother() {
          console.log( another.join( " ! " ) );
     }
return {
          doSomething: doSomething,
          doAnother: doAnother
     };
})();
foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

Chúng ta đã chuyển hàm module thành IIFE và gọi nó ngay lập tức và gán giá trị trả lại của nó trực tiếp vào module instance đơn `foo`.

```
function CoolModule(id) {

     function identify() {

          console.log( id );

     }
return {
          identify: identify
     };
}
var foo1 = CoolModule( "foo 1" );
var foo2 = CoolModule( "foo 2" );
foo1.identify(); // "foo 1"
foo2.identify(); // "foo 2"
```

Một biến thể mạnh mẽ của module pattern là đặt tên object và bạn trả lại như một API công khai:

```
var foo = (function CoolModule(id) {

     function change() {

          // sửa đổi API công khai
```

```
            publicAPI.identify = identify2;

      }
function identify1() {
            console.log( id );
      }
function identify2() {
            console.log( id.toUpperCase() );
      }
var publicAPI = {
            change: change,
            identify: identify1
      };
return publicAPI;
})( "foo module" );
foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE
```

Bằng cách giữ lại tham chiếu bên trong object API công khai bên trong module instance, bạn có thể sửa đổi module instance đó **từ phía trong**, bao gồm phương thức thêm và bớt thuộc tính, và thay đổi giá trị.

Module hiện đại

Vài trình tải/quản lý module phụ thuộc bao lấy mẫu module thành một API gần gũi hơn. Thay vì kiểm tra bất kỳ một thư viện cụ thể, tôi sẽ giới thiệu một chứng minh đơn giản cho khái niệm này nhằm **mục đích minh họa**:

```
var MyModules = (function Manager() {

     var modules = {};
function define(name, deps, impl) {
          for (var i = 0; i < deps.length; i++) {
               deps[i] = modules[deps[i]];
          }
          modules[name] = impl.apply( impl, deps );
     }
function get(name) {
          return modules[name];
     }
return {
          define: define,
          get: get
     };
})();
```

Mấu chốt của đoạn code trên là `modules[name] = impl.apply(impl, deps)`. Nó sẽ chạy hàm xác định bên ngoài để tạo module, và lưu giá trị trả lại (API của module) thành một danh sách các module được theo dõi theo tên.

Và đây là cách tôi có thể sử dụng để tạo module:

```
MyModules.define( "bar", [], function(){

     function hello(who) {

          return "Let me introduce: " + who;
```

```
        }
return {
            hello: hello
        };
} );
MyModules.define( "foo", ["bar"], function(bar){
     var hungry = "hippo";
function awesome() {
          console.log( bar.hello( hungry ).toUpperCase() );
     }
return {
          awesome: awesome
     };
} );
var bar = MyModules.get( "bar" );
var foo = MyModules.get( "foo" );
console.log(
     bar.hello( "hippo" )
); // Let me introduce: hippo
foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Cả module "foo" và "bar" đều được xác định bằng một hàm trả một API công khai. "foo" thậm chí nhận được instance của "bar" như một tham số phụ thuộc, và có thể sử dụng nó tùy ý.

Bạn hãy dành thời gian để kiểm tra đoạn code trên cho đến khi hiểu hoàn toàn sức mạnh của closure để phục vụ cho mục đích của mình. Không có "ma thuật" nào trong trình quản lý module, nó đáp ứng hai đặc tính của module pattern mà tôi đã liệt kê ở trên: gọi một hàm xác định bao ngoài, và trả giá trị như là API của module đó.

Nói cách khác, module là module, kể cả khi bạn tạo một công cụ trên nó.

Module tương lai

ES6 bổ sung cú pháp cao cấp cho khái niệm module. Khi được tải bởi hệ thống module, ES6 xử lý một file như một module riêng lẻ. Mỗi module có thể vừa nhập các module khác hoặc thành viên API cụ thể, đồng thời xuất các thành viên API công khai của chính nó.

Ghi chú: Module nền hàm không phải là nhận dạng mẫu tĩnh (cái mà trình biên dịch hiểu rõ), vì vậy API của nó sẽ không được nhận thấy cho đến run-time. Vì thế, bạn có thể thay đổi một API của module trong quá trình run-time (xem thảo luận về `publicAPI` ở trên).

Ngược lại, ES6 API của module là tĩnh (API không thay đổi tại run-time). Do đó, trình biên dịch kiểm tra trong quá trình tải và biên dịch có một tham chiếu đến thành của một API module đã nhập có *thực sự tồn tại*. Nếu tham chiếu API không tồn tại, trình biên dịch báo lỗi "trước" trong thời gian biên dịch

(compiler-time) thay vì chờ cho run-time (báo lỗi, nếu có) theo thông thường.

ES6 module không có định dạng “trực tiếp”, nó cần được xác định theo các file riêng lẻ (với mỗi module). Trình duyệt/engine có một trình tải module mặc định (có thể ghi đè, không thuộc vấn đề nói tới ở đây) đồng bộ tải module file khi nó được nhập (import).

Ví dụ:

bar.js

```
function hello(who) {

     return "Let me introduce: " + who;

}
export hello;
```

foo.js

```
// nhập mình `hello()` từ module "bar"

import hello from "bar";
var hungry = "hippo";
function awesome() {
```

```
	console.log(
		hello( hungry ).toUpperCase()
	);
}
export awesome;
// nhập toàn bộ "foo" và "bar"
module foo from "foo";
module bar from "bar";
console.log(
	bar.hello( "rhino" )
); // Let me introduce: rhino
foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Ghi chú: Các file riêng lẻ "**foo.js**" và "**bar.js**" cần được tạo, với nội dung như hai đoạn code trên. Sau đó chương trình của bạn có thể tải/nhập các module đó để sử dụng chúng như đã trình bày ở đoạn code thứ ba.

`import` một hay nhiều thành viên từ API của module trong phạm vi gần nhất, mỗi cái là một biến (`hello` trong trường hợp của ta). `module` nhập toàn bộ API module thành các biến (`foo`, `bar` trong trường hợp ví dụ). `export`xuất một định danh (biến, hàm) thành API công khai cho module gần nhất. Các biểu thức này có thể sử dụng nhiều lần trong việc xác định module nếu cần thiết.

Nội dung bên trong *file module* được xử lý như một scope closure bên ngoài, cũng như hàm closure đã xem ở trên.

Ôn tập (TL;DR)

Closure được xem như sự giác ngộ về thế giới bí ẩn bên trong JavaScript. Nhưng nó lại là một tiêu chuẩn và rõ ràng khi chúng ta viết code trong môi trường lexical scope, hàm và giá trị có thể truyền xung quanh theo ý muốn.

Khi một hàm có thể ghi nhớ và truy cập lexical scope của nó kể cả khi nó được gọi ngoài lexical scope được gọi là closure.

Closure có thể làm chúng ta sai sót, ví dụ với loop, nếu ta không cẩn thận khi nhận biết chúng và cách chúng hoạt động. Nhưng nó cũng là một công cụ mạnh mẽ vô hạn, cho phép triển khai các mẫu (pattern) như *module*theo nhiều dạng khác nhau.

Module đòi hỏi hai đặc điểm chính:

1. Hàm bao ngoài được gọi để tạo scope bao ngoài.
2. Giá trị trả về của hàm bao ngoài phải bao gồm tham chiếu đến ít nhất một hàm bên trong để sau đó nó có closure thông qua phạm vi riêng (private) bên trong của hàm đó.

Và ta cũng thấy closure xuất hiện liên tục ở trong code của mình, đây là khả năng để nhận biết và tận dụng chúng cho mục đích của mình!

You Don’t Know JS (tiếng Việt)—Scope & closure—Phụ lục A: Scope động (dynamic scope)

O'REILLY®

"Kyle's way of critically thinking about every tiny bit of the language will creep into your mindset and general workflow."
—**SHANE HUDSON**, freelance frontend website developer

KYLE SIMPSON

SCOPE & CLOSURES

YOU DON'T KNOW JS

Trong chương 2, ta đã nói về “Dynamic Scope” là tương phản của “Lexical Scope”, đó là cách phạm vi hoạt động trong JavaScript (thực tế là nhiều ngôn ngữ khác cũng vậy).

Chúng ta sẽ xem xét một chút nhẹ dynamic scope để coi tương phản đó ra sao. Nhưng quan trọng hơn là dynamic scope thực tế là con cô con cậu của cơ chế khác (`this`) trong JavaScript, ta sẽ coi nó trong phần "*this & Nguyên mẫu đối tượng*".

Như ta đã biết trong Chương 2, lexical scope là một tập hợp các quy tắc về cách tìm kiếm biến như thế nào và ở đâu của *Engine*. Đặc điểm chính của lexical scope là nó được xác định tại author-time khi code được viết ra (giả định bạn không dùng `eval()` hoặc `with` để đánh lừa).

Dynamic scope dường như là ngụ ý rằng có một mô hình mà theo đó phạm vi có thể được mô tả linh hoạt tại runtime hơn là thụ động tại author-time. Xem ví dụ minh họa:

```
function foo() {

    console.log( a ); // 2

}
```

```
function bar() {
     var a = 3;
     foo();
}
var a = 2;
bar();
```

Lexical scope giữ tham chiếu RHS đến `a` trong `foo()` sẽ được xử lý thành biến toàn cục `a`, và sẽ có kết quả xuất ra là `2`.

Ngược lại, dynamic scope không quan tâm đến việc khi nào, ở đâu hàm và scope được khai báo, mà đúng hơn là **chúng được gọi từ đâu**. Nói cách khác, chuỗi phạm vi dựa trên call-stack, không phải scope lồng nhau trong code.

Vì vậy, nếu JS có dynamic scope, khi `foo()` được thực thi, **về mặt lý thuyết** thì đoạn code dưới đây sẽ ra kết quả là `3`.

```
function foo() {

     console.log( a ); // 3  (không phải 2!)

}
function bar() {
     var a = 3;
     foo();
}
var a = 2;
bar();
```

Sao lại vậy? Bởi vì khi `foo()` không thể xử lý tham chiếu biến cho `a`, thay vì nó đi tiếp lên trên đến chuỗi scope được lồng (lexical) tiếp theo, nó đi tiếp tới call-stack để tìm xem `foo()` được gọi gọi từ đâu. Khi `foo()` được gọi từ `bar()`, nó kiểm tra biến trong scope của `bar()` và tìm thấy `a` có giá trị bằng `3`.

Trong giây phút thì có thể bạn nghĩ rằng nó kỳ lạ.

Thực ra là bởi bạn chỉ mới làm việc với (ít nhất là tìm hiểu sâu) code được lexical scope, nên dynamic scope thấy có vẻ xa lạ. Nếu bạn đã từng viết code ở ngôn ngữ dynamic scope, thì đây là bình thường, còn lexical scope mới là xa lạ.

Nói thẳng đuột, JavaScript **không có dynamic scope** mà chỉ có lexical scope. Nhưng mà cơ chế `this` lại là một kiểu dynamic scope.

Mâu thuẫn then chốt: l**exical scope là write-time/author-time trong khi dynamic scope (và `this`!) là runtime**. Lexical scope quan tâm *hàm được khai báo ở đâu*, nhưng dynamic scope quan tâm hàm được *gọi từ đâu*.

Còn `this` thì lại quan tâm việc *hàm được gọi như thế nào*, đây là mối liên hệ của cơ chế `this` với ý tưởng dynamic scope.

Homepage

SeanDangFollow

@Doda. I'm just a Product manager, senior UI/UX, Frontend developer.

Feb 27

You Don’t Know JS: Scope & Closures -Phụ lục B: Polyfilling Block Scope

O'REILLY®

"Kyle's way of critically thinking about every tiny bit of the language will creep into your mindset and general workflow."
—**SHANE HUDSON**, freelance frontend website developer

KYLE SIMPSON

SCOPE & CLOSURES

YOU DON'T KNOW JS

Trong Chương 3, chúng ta khám phá Block Scope. Ta thấy rằng mệnh đề `with` và `catch` đều là hai ví dụ nhỏ của block scope tồn tại trong JS từ ES3.

Khi ES6 giới thiệu `let` cuối cùng cũng cung cấp đầy đủ khả năng mở rộng của block-scoping cho code của chúng ta. Có rất nhiều điều hấp dẫn, cả hàm và phong cách code mà block scope sẽ cho phép.

Nhưng nếu như chúng ta sử dụng block scope trong môi trường trước ES6.

Xem đoạn code sau:

```
{
    let a = 2;
    console.log( a ); // 2
```

```
}
console.log( a ); // ReferenceError
```

Với môi trường ES6 thì đoạn này chơi ngon, nhưng với bản trước ES6 `catch` sẽ là câu trả lời.

```
try {

     throw 2

} catch(a) {

     console.log( a ); // 2

}
console.log( a ); // ReferenceError
```

Whoa! Code nhìn xấu vãi cả dị. Ta thấy `try/catch` xuất hiện để ép việc ném ra lỗi, nhưng "error" chỉ quăng ra giá trị `2`, và sau

đó khai báo biến sẽ nhận giá trị bên trong mệnh đề `catch(a)`. Bể não.

Đúng vậy, mệnh đề `catch` có block-scoping của nó, nghĩa là nó có thể sử dụng như một polyfill cho block scope trong môi trường tiền ES6.

“Nhưng...chẳng ai muốn viết code xấu vậy!” Đúng vậy, cũng giống việc chẳng ai viết code theo phong cách code được phiên dịch từ CoffeeScript cả. Nhưng đây không phải vấn đề.

Vấn đề là công cụ này có thể chuyển code ES6 có thể làm việc với môi trường tiền ES6. Bạn có thể viết code theo block-scoping với những ích lợi từ nó, rồi để cho công cụ ở bước built lo phần chuyển đổi giúp code có thể làm việc ngon lành khi triển khai.

E hèm, đây là lối đi ưa thích cho mọi người trong việc chuyển đổi ES6: sử dụng một trình phiên dịch code để chuyển code ES6 thành code tương thích ES5 trong quá trình (các trình duyệt—người dịch) hoán chuyển từ tiền ES6 sang ES6.

Traceur

Google bảo trì một dự án tên là "Traceur" Google Traceur, với nhiệm vụ chính xác là chuyển các tính năng ES6 sang tiền ES6 (hầu hết là ES5, nhưng không phải tất cả!) để sử dụng chung. Ủy ban TC39 dựa vào công cụ này (và các công cụ khác) để kiểm tra ngữ nghĩa của các tính năng họ chỉ định.

Traceur tạo gì trong đoạn code của ta? Bạn đoán rồi đó!

```
{
    try {
        throw undefined;
    } catch (a) {
        a = 2;
```

```
        console.log( a );

    }

}
console.log( a );
```

Với việc sử dụng các công cụ như vậy, chúng ta bắt đầu khai thác lợi thế của block scope bất kể ta có nhắm đến ES6 hay không, bởi vì `try/catch` đã tồn tại và làm việc theo cách này từ hồi ES3.

Khối ngầm vs. minh bạch

Trong Chương 3, chúng ta nhận diện vài cạm bẫy tiềm tàng với tính bảo trì/refactor của code khi giới thiệu về block-scoping. Có cách nào khác để tận dụng lợi thế của block scope mà giảm thiểu nhược điểm này?

Xem xét hình thức thay thế của `let`, gọi là "let block" hay "câu lệnh let" (tương phản với "khai báo let").

```
let (a = 2) {

    console.log( a ); // 2

}
console.log( a ); // ReferenceError
```

Thay vì ngầm chiếm lấy block hiện hữu, lệnh let tạo một block rõ ràng cho ràng buộc scope của nó. Không chỉ làm nổi bật các block minh bạch hơn, mà có lẽ thiết thực hơn trong việc refactor code, nó tạo ra gì đó sạch hơn về mặt ngữ pháp, buộc tất cả các khai báo lên trên đầu block. Nó làm cho bấy kỳ block nào cũng dễ thấy và scope gì của nó hay không.

Là một khuôn mẫu, nó phản ánh cách tiếp cận của nhiều người trong việc scope hóa hàm khi họ tự dời/hoist tất cả khai báo `var` lên trên đầu hàm. Lệnh let đặt chúng lên đầu block theo ý

định, và nếu bạn không sử dụng khai báo `let` rải rác, khai báo block scoping của bạn sẽ dễ dàng nhận diện để bảo trì hơn.

Nhưng lại có một vấn đề, dạng lệnh let không có trong ES6. Và cả Traceur cũng không chấp nhận dạng code này.

Ta có hai lựa chọn. Ta có thể bắng bằng cách sử dụng cú pháp ES6 hợp lệ và một chút khuôn phép:

```
/*let*/ { let a = 2;

     console.log( a );

}
console.log( a ); // ReferenceError
```

Nhưng các công cụ tồn tại để giải quyết vấn đề của chúng ta. Vì vậy lựa chọn khác là viết các khối lệnh let rõ ràng và để công cụ chuyển chúng thành code hợp lệ.

Vì vậy tôi đã xây dựng một công cụ gọi là let-er để xử lý vấn đề này. *let-er* là một trình phiên dịch code ở bước build, nhiệm vụ duy nhất của nó là tìm dạng lệnh let và phiên dịch , những dạng code khác nó vẫn để nguyên, kể cả khai báo let. Bạn có thể sử dụng *let-er* một cách an toàn như là bước phiên dịch ES6 ban đầu, sau đó chuyển code của bạn qua gì đó như Traceur nếu cần thiết.

Hơn nữa, *let-er* có một cờ cấu hình (configuration flag) `--es6`, có thể bật (mặc định là tắt), thay đổi kiểu code được tạo. Thay vì dùng `try/catch`, *let-er* sẽ chuyển đoạn code tuân thủ ES6 đầy đủ mà không phải hack gì cả:

```
{
    let a = 2;
    console.log( a );
}
console.log( a ); // ReferenceError
```

Bạn có thể dùng *let-er* ngay bây giờ và quan trọng nhất là, **bạn có thể sử dụng dạng lệnh let một cách thích hợp và rõ ràng** mặc dù nó không (chưa) phải là phần chính thức của ES.

Hiệu suất

Tôi nêu nhanh về hiệu suất `try/catch` một chút để giải thích câu hỏi "tại sao sử dụng IIFE để tạo scope?"

Trước tiên, hiệu suất của `try/catch` chậm hơn nhưng lại không có một giả định hợp lý rằng phải dùng cách này, hay nó vốn phải vậy. Từ khi TC39 chính thức chấp nhận trình phiên dịch ES6 sử dụng `try/catch`, nhóm Traceur đã yêu cầu Crome cải thiện hiệu suất của `try/catch`, và rõ ràng là họ có động cơ để đòi hỏi.

Tiếp đến, IIFE không phải là cách so sánh ngang bằng với `try/catch`, vì bất kỳ đoạn code nào được bao bởi hàm đều thay đổi ý nghĩa bên trong code, ý nghĩa của `this`, `return`, `break`, và

`continue`. IIFE không phải là một thay thế phù hợp. Nó chỉ có thể dùng thủ công tùy trường hợp.

Câu hỏi cụ thể là: bạn có thực sự muốn block-scoping hay không? Nếu có thì công cụ cho bạn một phương án. Nếu không, hãy cứ dùng `var` và tiếp tục code thôi.

You Don’t Know JS: Scope & Closures — Phụ lục C: Lexical-this

O'REILLY®

"Kyle's way of critically thinking about every tiny bit of the language will creep into your mindset and general workflow."
—**SHANE HUDSON**, freelance frontend website developer

KYLE SIMPSON

SCOPE & CLOSURES

YOU DON'T KNOW JS

Dù tiêu đề này không nhắm đến cơ chế `this` gì cả, nhưnf đây là một đề tài khá quan trọng của ES6 mà `this` có liên quan đến lexical scope.

ES6 bổ sung một dạng cú pháp quan trọng cho việc khai báo hàm gọi là “hàm mũi tên” (arrow function). Trông như sau:

```
var foo = a => {

     console.log( a );

};
foo( 2 ); // 2
```

Nó còn được gọi là “mũi tên bự”, thường được gợi ý như là dạng viết tắt của từ khóa `function` *thừa thải rườm rà.*

Nhưng có vấn đề quan trọng hơn khi dùng hàm mũi tên ngoài việc tiết kiệm gõ phím khi khai báo. Ngắn gọn thì đoạn code dưới đây có trục trặc:

```
var obj = {

     id: "chuẩn",
```

```
	cool: function coolFn() {

		console.log( this.id );

	}

};
var id = "chưa chuẩn";
obj.cool(); // chuẩn
setTimeout( obj.cool, 100 ); // chưa chuẩn
```

Vấn đề ở đây là mất ràng buộc `this` tại hàm `cool()`. Có vài cách để xử lý vụ này, nhưng thường thì dùng giải pháp `var self = this;`.

```
var obj = {

	count: 0,

	cool: function coolFn() {

		var self = this;
		if (self.count < 1) {
			setTimeout( function timer(){
				self.count++;
				console.log( "chuẩn chưa?" );
			}, 100 );
		}
	}
};
```

```
obj.cool(); // chuẩn chưa?
```

Không phải dùng cỏ (cần sa) quá nhiều vì quá ảo, “giải pháp” `var self = this` chỉ phân phối cho toàn bộ việc hiểu và dùng đúng ràng buộc `this`, và quay về cái gì đó mà chúng ta cảm thấy thoải mái: lexical scope, `self` trở thành một định danh có thể giải quyết thông qua lexical scope và closure, không quan tâm những gì xảy ra với ràng buộc `this` trên đường đi.

Người ta không thích viết cái gì dài dòng, đặc biệt là khi cứ phải lặp lại mãi. Bởi vậy, một động lực của ES6 là hỗ giảm bớt hoàn cảnh này, thực tế là *sửa*một vấn đề bản chất, như là cái này.

Giải pháp hàm mũi tên của ES6 giới thiệu một hành vi gọi là “lexical this”.

```
var obj = {
     count: 0,
     cool: function coolFn() {
          if (this.count < 1) {
               setTimeout( () => { // gì vậy, hàm mũi tên?
```

```
                    this.count++;

                    console.log( "chuẩn chưa?" );

               }, 100 );

          }

     }

};
obj.cool(); // chuẩn chưa?
```

Giải thích ngắn gọn ở trên cho thấy hàm mũi tên không hành xử tất cả như hàm thông thường khi nói đến ràng buộc `this`. Nó loại bỏ tất cả các quy tắc thông thường của ràng buộc `this`, và thay vào đó lấy ngay giá trị `this`của lexical scope bao ngoài, bất kể nó là gì.

Vì vậy, trong đoạn code trên, hàm mũi tên không nhận được loại bỏ ràng buộc `this` của nó theo một số cách không đoán được, nó chỉ "kế thừa" ràng buộc `this` của hàm `cool()` (điều này đúng nếu ta gọi nó như đã trình bày).

Trong khi nó làm cho code ngắn hơn, quan điểm của tôi là hàm mũi tên chỉ thực sự chỉ làm cho code thành một cú pháp gây sai lầm thường thấy cho lập trình viên nhầm lẫn và pha trộn quy tắc "this binding" với quy tắc "lexical scope".

Ngược lại với quan điểm trên: tại sao lại đi gặp rắc rối và rườm rà bằng cách sử dụng mô hình `this`, chỉ cần cắt bớt và trộn nó với tham chiếu lexical. Việc nắm bắt một trong những cách tiếp cận cho bất kỳ đoạn code mà không phải trộn chúng trong cùng một đoạn code có vẻ tự nhiên hơn.

Ghi chú: một nhược điểm khác của hàm mũi tên là chúng vô danh. Xem **Chương 3** để biết lý do tại sao hàm vô danh ít được chú trọng bằng hàm định danh.

Theo quan điểm của tôi với vấn đề này thì nên nắm bắt và sử dụng cơ chế `this` một cách chính xác.

```
var obj = {

    count: 0,

    cool: function coolFn() {
```

```
        if (this.count < 1) {

            setTimeout( function timer(){

                this.count++; // `this` an toàn nhờ
`bind(..)`

                console.log( "càng chuẩn" );

            }.bind( this ), 100 ); // xem kìa, `bind()`!

        }

    }

};
obj.cool(); // càng chuẩn
```

Dù bạn thích hàm mũi tên với hành vi lexical-this hơn, hay bạn thích sử dụng `bind()` hơn, điều quan trọng cần ghi chú là hàm mũi tên *không chỉ*đơn thuần là đỡ mất công gõ chữ "function". Nó có một *sự khác biệt về hành vi có chủ đích* mà chúng ta cần học và hiểu, nếu ta chọn.

Giờ bạn đã hiểu lexical scope (và closure), hiểu lexical-this cũng không phải là thừa!

You Don’t Know JS: this & Nguyên mẫu đối tượng—Chương 1: this hay là That?

Một trong những cơ chế hãi hùng nhất của JavaScript là từ khoá `this`. Nó là một từ khoá nhận dạng đặc biệt tự động xác định trong scope của mọi hàm, nhưng ngay cả lập trình viên JavaScript dày dặn cũng phải điêu đứng với câu chuyện chính xác nó là gì.

Công nghệ *tiên tiến* với phép thuật không phân biệt được.—Arthur C. Clarke

Cơ chế của `this` không hẳn ghê gớm như câu trên, nhưng các lập trình viên thường diễn giải trong tâm trí bằng cách thêm từ "phức tạp" hoặc "mơ hồ", `this` có thể xem là sự huyền diệu trong cái hoang mang.

Tại sao lại là this?

Nếu cơ chế `this` quá phức tạp nên ngay cả đối với lập trình viên JavaScript dày dạn vẫn có thể thắc mắc tại sao nó lại hữu dụng? Hay nó chỉ đem lại rắc rối? Trước khi tìm hiểu *như thế nào*, chúng ta nên xác định *vì sao*

Hãy thử minh hoạ động lực và tiện ích của `this`:

```
function identify() {

	return this.name.toUpperCase();

}
function speak() {
	var greeting = "Hello, I'm " + identify.call( this );
	console.log( greeting );
}
var me = {
	name: "Kyle"
};
var you = {
	name: "Reader"
};
identify.call( me ); // KYLE
identify.call( you ); // READER
speak.call( me ); // Hello, I'm KYLE
speak.call( you ); // Hello, I'm READER
```

Nếu *như thế nào* của đoạn code này vẫn còn làm bạn bối rối, đừng lo! Chúng ta sẽ tìm hiểu nó cụ thể ngay sau đây. Chỉ cần đặt câu hỏi này qua một bên, như vậy ta mới có thể nhìn rõ *tại sao* hơn.

Đoạn trích này cho phép hàm `identify()` và `speak()` được tái sử dụng cho object trong nhiều *ngữ cảnh* (`me` và `you`) thay vì sử dụng nhiều phiên bản khác nhau của hàm cho mỗi object.

Thay vì sử dụng `this`, bạn có thể thông qua object một cách rõ ràng trong ngữ cảnh cho cả `identify()` và `speak()`.

```
function identify(context) {

     return context.name.toUpperCase();

}
function speak(context) {
     var greeting = "Hello, I'm " + identify( context );
     console.log( greeting );
}
identify( you ); // READER
speak( me ); // Hello, I'm KYLE
```

Tuy nhiên, cơ chế `this` cung cấp phương thức tao nhã hơn theo kiểu ngầm truyền một đối tượng tham chiếu, hướng tới thiết kế API rõ ràng hơn và dễ sử dụng hơn.

Mô hình bạn sử dụng càng phức tạp, bạn sẽ càng thấy rõ việc dùng một tham số cụ thể tên tuổi thường rối rắm hơn dùng `this`. Khi bạn khám phá object & prototype, bạn sẽ thấy sự hữu dụng của một tập hợp hàm có thể tự động tham chiếu với object ngữ cảnh thích hợp.

Các nhầm lẫn

Chúng ta sẽ sớm giải thích `this` *thực sự* hoạt động ra sao, nhưng trước tiên chúng ta phải xoá tan những hiểu nhầm về những gì nó không thực sự đúng.

Cái tên "this" tạo ra một số nhầm lẫn khi các lập trình viên cố gắng nghĩ về nó theo nghĩa đen quá. Có hai ý nghĩa thường đưa ra nhưng cả hai đều sai.

Chính nó

Cám dỗ phổ biến đầu tiên là giả định bản thân `this` như một hàm. Ít nhất, đó là một suy luận có lý về mặt ngữ pháp.

Tại sao bạn muốn tham chiếu một hàm bên trong chính nó? Một số lý do phổ biến là thứ gì đó như kiểu đệ quy (gọi hàm từ bên trong nó) hoặc có một hàm xử lý sự kiện (event handler) mà có thể tự huỷ trong lần gọi đầu tiên.

Các lập trình viên mới làm quen với cơ chế của JS thường tham chiếu hàm như một object (vì tất cả các hàm trong JS đều là object!) cho phép bạn lưu trữ *state* (giá trị bên trong thuộc tính) khi gọi hàm. Mặc dù điều này là chắc chắn và có một số hạn chế sử dụng, phần còn lại của cuốn sách sẽ mô tả trên nhiều mô hình khác cho *vị trí* tốt hơn để lưu trữ state bên cạnh mô hình object hàm.

Ở đây chúng ta tìm hiểu vấn đề này một chút, để minh hoạ `this` không cho một hàm tham chiếu lên chính nó như chúng ta giả định như thế nào.

Xem đoạn code dưới đây, chúng ta thử theo dõi bao nhiêu lần hàm `foo`được gọi:

```
function foo(num) {
```

```
    console.log( "foo: " + num );
    // theo dõi bao nhiêu lần `foo` được gọi
    this.count++;
}
foo.count = 0;
var i;
for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9
// bao nhiêu lần `foo` được gọi?
console.log( foo.count ); // 0 -- WTF?
```

`foo.count` vẫn là `0`, mặc dù qua bốn lần `console.log` đã chỉ ra rõ ràng `foo(..)` là sự kiện được gọi bốn lần. Sự thất vọng bắt nguồn từ sự diễn dịch *quá trực nghĩa* của `this` (`this.count++`).

Khi `foo.count = 0` được thực thi, thực chất là nó thêm một thuộc tính `count` vào hàm `foo`. Nhưng với `this.count`tham chiếu trong hàm, `this`không trỏ đến hàm chút nào, và mặc dù tên thuộc tính giống nhau, object gốc lại khác nhau, và sự nhầm lẫn xảy ra.

Ghi chú: Một lập trình viên có trách nhiệm sẽ *phải* hỏi ở chỗ này "nếu tôi đã thêm một thuộc tính `count` nhưng nó lại không

phải cái tôi mong muốn, vậy `count` mà tôi đã thêm là cái nào?" Thực tế, nếu đào sâu thêm, bạn sẽ thấy rằng vô tình bạn đã tạo một biến toàn cục `count` (xem Chương 2 để biết nó đã xảy ra *như thế nào*), và hiện tại nó có giá trị `NaN`. Tất nhiên, một khi bạn đã nhận thấy kết quả đặc biệt này, câu ta sẽ có một câu hỏi tổng quan hơn: "Nó toàn cục như thế nào? và tại sao nó lại kết thúc bằng `NaN`thay cho một giá trị đếm được?" (Xem Chương 2)

Thay vì dừng tại điểm này và đào sâu vào lý do tại sao tham chiếu `this` có vẻ không thoả *kỳ vọng*, và trả lời của những câu hỏi khó nhưng quan trọng này, nhiều lập trình viên đơn giản tránh lỗi này, và khám phá giải pháp khác, ví dụ như tạo một object để giữ giá trị `count`:

```
function foo(num) {

	console.log( "foo: " + num );
	// theo dõi `foo` được gọi bao nhiêu lần
	data.count++;
}
var data = {
	count: 0
};
var i;
for (i=0; i<10; i++) {
	if (i > 5) {
		foo( i );
	}
}
// foo: 6
```

```
// foo: 7
// foo: 8
// foo: 9
// bao nhiêu lần `foo` được gọi?
console.log( data.count ); // 4
```

Trong khi rõ ràng cách tiếp cận này "giải quyết" vấn đề, không may nó lại bỏ qua một vấn đề thực sự—thiếu hiểu biết ý nghĩa của `this` và cách nó làm việc -- thay vào đó là rút lui vào một cơ chế gần gũi hơn: lexical scope

Chú ý: Lexical scope là một cơ chế tốt và hững dụng; Trên mọi phương diện thì tôi không coi thường việc sử dụng nó (Xem *"Scope & Closures"*). Nhưng lúc nào cũng *đoán* sử dụng `this` như thế nào, hoặc thường xuyên hiểu *sai*thì đây không phải là lý do tốt để chuyển qua sử dụng lexical scope và không bao giờ học *vì sao* `this` lẩn tránh bạn.

Để tham chiếu một hàm bên trong chính nó, bản thân `this` thường không đầy đủ, bạn cần có một tham chiếu đến hàm thông qua một định danh gốc (biến) để trỏ vào nó.

Xem 2 hàm dưới đây:

```
function foo() {

	foo.count = 4; // `foo` tham chiếu đến chính nó
```

```
}
setTimeout( function(){
     // hàm vô danh, không thể tham chiếu đến chính nó.
}, 10 );
```

Trong hàm thứ nhất, gọi là “hàm định danh”, `foo` là một tham chiếu có thể sử dụng để tham chiếu đến hàm bên trong nó.

Nhưng trong ví dụ thứ 2, hàm callback qua `setTimeout(..)` không có nhận dạng tên (còn gọi là "hàm vô danh"), do đó không có cách nào tham chiếu đến object hàm bởi chính nó.

Ghi chú: Có một phương thức cũ bị bãi bỏ là `arguments.callee` tham chiếu bên trong một hàm *cũng* trỏ về object hàm của hàm đang được thực thi. Các tham chiếu này thường là cách duy nhất để tiếp cận một object hàm vô danh bên trong nó. Tuy nhiên, cách tiếp cận tốt nhất là tránh sử dụng chức năng ẩn danh hoàn toàn, ít nhất là đối với các hàm cần một tham chiếu chính nó, thay vào đó là tạo hàm có tên (biểu thức). `arguments.callee`không được sử dụng nữa và cũng không nên sử dụng.

Do đó một giải pháp khác cho ví dụ của chúng ta là sử dụng nhận diện `foo`như một tham chiếu object hàm tại mỗi vị trí, và không sử dụng `this` nữa, nó sẽ hoạt động.

```
function foo(num) {

	console.log( "foo: " + num );
	// Kiểm tra bao nhiêu lần `foo` được gọi
	foo.count++;
}
foo.count = 0;
var i;
for (i=0; i<10; i++) {
	if (i > 5) {
		foo( i );
	}
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9
// Bao nhiêu lần `foo` được gọi?
console.log( foo.count ); // 4
```

Tuy nhiên, cách tiếp cận này tương tự các bước *thực sự* hiểu `this` và dựa hoàn toàn vào lexical scope của biến `foo`.

Cách khác để tiếp cận vấn đề là buộc `this` thực sự trỏ vào `foo` object hàm:

```
function foo(num) {
```

```
	console.log( "foo: " + num );
	// theo dõi bao nhiêu lần `foo` được gọi
	// Ghi chú: giờ `this` thực sự là `foo`, dựa vào
	// `foo` được gọi như thế nào (xem bên dưới)
	this.count++;
}
foo.count = 0;
var i;
for (i=0; i<10; i++) {
	if (i > 5) {
		// sử dụng `call(..)`, chúng ta chắc chắn `this`
		// tự nó trỏ vào object hàm (`foo`)
		foo.call( foo, i );
	}
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9
// bao nhiêu lần `foo` được gọi?
console.log( foo.count ); // 4
```

Thay vì tránh né `this`, chúng ta bao lấy nó. Chúng ta sẽ giải thích các kỹ thuật này hoạt động tốt hơn như thế nào, vì vậy đừng lo lắng nếu bạn vẫn còn một chút băn khoan!

Phạm vi (Scope) của nó

Quan niệm sai lầm phổ biến tiếp theo về ý nghĩa của `this` là bằng cách nào đó, nó tham chiếu đến scope của hàm. Đây là

một câu hỏi khéo, vì xét về mặt nào đó là đúng, nhưng ở khía cạnh khác nó là sai.

Một cách rõ ràng thì `this` không tham chiếu đến một lexical scope của hàm. Đúng là xét về nội bộ, scope là một dạng giống như object có thuộc tính cho mỗi object định danh sẵn có. Nhưng scope "object" không thể truy cập vào JavaScript code. Nó là phần bên trong thực thi của *Engine*.

Xem đoạn code đã cố (và thất bại!) vượt qua ranh giới và sử dụng `this` để ngầm tham chiếu vào lexical scope của hàm:

```
function foo() {

     var a = 2;

     this.bar();

}
function bar() {
     console.log( this.a );
}
foo(); //undefined
```

Có nhiều sai sót trong đoạn code này. Đoạn code bạn thấy là sự trích tách từ một đoạn code thực tế được trao đổi trong các diễn đàn cộng đồng hỗ trợ. Nó là một minh hoạ tuyệt vời (nếu

không đáng buồn) cho việc hiểu sai `this` cho giả định được đặt ra.

Đầu tiên, một sự cố gắng được tạo ra để tham chiếu hàm `bar()` thông qua `this.bar()`. Hầu như chắc chắn rằng *vô tình* nó hoạt động, nhưng chúng ta sẽ giải thích *vì sao* sau. Cách tự nhiên nhất để gọi `bar()` là bỏ qua đầu mối `this` và chỉ cần tạo một tham chiếu lexical đến hàm định danh.

Tuy nhiên, lập trình viên nào đã viết đoạn code trên đang cố gắng sử dụng `this` để tạo ra một cây cầu giữa các lexical scope của `foo()` và `bar()`, từ đó `bar()` có thể tiếp cận biến `a` trong scope của `foo()`. **Chẳng có cái cầu nào xảy ra**. Bạn không thể sử dụng `this` để tham chiếu và tìm một thứ gì đó trong lexical scope. Hoàn toàn không thể.

Mỗi khi bạn cảm thấy mình muốn pha trộn việc tìm kiếm với `this`, luôn nhớ rằng: *không có cây cầu nào hết*

this là gì?

Với các giả định sai sót khác nhau, giờ chúng ta chuyển qua việc chú ý cơ chế hoạt động thực sự của `this`.

Chúng ta đã nhắc `this` không phải là một ràng buộc author-time mà là một ràng buộc runtime. Nó dựa trên ngữ cảnh của điều kiện gọi hàm. Ràng buộc `this` không liên quan với nơi mà hàm được khai báo, thay vào đó, mọi thứ liên quan đến cách gọi hàm.

Khi một hàm được gọi ra, một bản ghi kích hoạt hay còn gọi là ngữ cảnh thực thi được tạo ra. Bản ghi này chứa thông tin về nơi mà hàm được gọi (call-stack), *cách* hàm đã được gọi, tham số nào được truyền,v.v...Một trong như thuộc tính của tham chiếu của bản ghi này là `this`, là cái được sử dụng trong suốt quá trình thực thi hàm.

Trong chương tiếp theo, ta sẽ học cách tìm một call-site của hàm để xác định việc thực thi của nó sẽ ràng buộc `this`như thế nào.

Ôn tập (TL;DR)

Ràng buộc `this` là một đề tài gây bối rối cho nhiều lập trình viên JavaScript chưa bỏ thời gian tìm hiểu cơ chế của nó thực chất hoạt động như thế nào. Việc đoán, thử và sai, sao chép mù

quáng từ các câu trả lời từ Stack Overflow không phải là cách hiệu quả để tận dụng cơ chế `this` quan trọng này.

Cho dù bất kỳ giả định hay quan niệm sai lầm nào dẫn lối, để học `this`, bạn phải học cái không phải là `this`. Bản thân `this` không phải là một tham chiếu đến hàm, hay là một tham chiếu đến lexical scope của hàm.

`this` chỉ thực sự là một ràng buộc được tạo ra cho đến khi hàm được gọi, và cái gì nó tham chiếu đều được xác định bởi call-site tại nơi hàm được gọi.