

Cho người mới

Javascript

Từ cơ bản tới nâng cao

- Lý thuyết kèm bài tập
- Chỉ dẫn từng bước kèm code minh họa dễ hiểu



MỤC LỤC

LỜI NÓI ĐẦU	5
NỘI DUNG CUỐN SÁCH	7
CUỐN SÁCH NÀY DÀNH CHO AI?	9
Yêu cầu trình độ.....	9
Cách học đúng cách	9
GIỚI THIỆU	11
Lịch sử Javascript.....	12
Tại sao nên học Javascript?.....	12
TỔNG QUAN JAVASCRIPT.....	15
Ưu điểm của Javascript.....	15
Giới hạn của Javascript	17
Công cụ phát triển.....	17
Thực thi chương trình Javascript.....	18
Tạo chương trình Javascript đầu tiên	19
CÚ PHÁP JAVASCRIPT CƠ BẢN	23
Variable - Biến	23
Variable Scope - Phạm vi sử dụng của một biến.....	25
Sự khác nhau giữa <i>var</i> và <i>let</i>	28
Khái niệm và cơ chế Hoisting	29
Kiểu dữ liệu.....	31
Toán tử - Operators.....	34
Toán tử số học	35
Toán tử so sánh.....	36
Toán tử logic.....	37
Toán tử gán	38
Toán tử điều kiện rút gọn	39

Làm việc với điều kiện và cấu trúc có điều kiện	39
<i>if...else</i> Statements.....	39
<i>Switch</i> Statements.....	40
CÚ PHÁP JAVASCRIPT NÂNG CAO.....	42
Function.....	42
Cách định nghĩa một function	44
Tham số và phạm vi.....	45
Nested scope	46
Pure function và non-pure function.....	47
Loop - vòng lặp.....	48
Vòng lặp là gì?	48
Tại sao phải dùng vòng lặp.....	48
Vòng lặp <i>for (...)</i>	49
Vòng lặp <i>while() {...}</i>	51
Vòng lặp <i>do {...} while()</i>	52
Câu lệnh <i>break</i> và <i>continue</i> trong vòng lặp.....	52
DỮ LIỆU CÓ CẤU TRÚC.....	56
Object.....	56
Thuộc tính riêng và thuộc tính kế thừa	58
Cách tạo Object.....	59
Truy xuất thông tin Object.....	63
Truy xuất hàng loạt keys của Object	64
Xóa thuộc tính của Object.....	66
Array	67
Cách khai báo Array.....	67
Truy cập vào phần tử mảng	68
Các thao tác làm việc với mảng.....	69
HIGHER-ORDER FUNCTION.....	80
Khái niệm Higher-Order Functions	80

Functional Programming	81
First-Class Functions.....	81
Higher-Order Functions là gì?.....	81
Ví dụ minh họa Higher-Order function.....	83
Tìm hiểu kỹ hơn về Callback.....	87
Promise	92
Async/Await.....	98
LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VỚI JS	102
Nguyên lý lập trình hướng đối tượng (OOP).....	102
Javascript có hướng đối tượng không?	104
Tính kế thừa.....	105
Tính đóng gói	108
Tính đa hình và trừu tượng.....	109
CÚ PHÁP ES6.....	113
String.....	114
Function.....	118
Class.....	120
Destructuring.....	124
Object Destructuring	124
Array Destructuring	125
Spread operator (...)	126
Modules.....	129
Export.....	129
Import	130
JAVASCRIPT FRAMEWORK.....	132
BÀI TẬP	135
KẾT NỐI VỚI VNTALKING	144
THÔNG TIN TÁC GIẢ	145
CUỐN SÁCH CỦA VNTALKING	146

LỜI NÓI ĐẦU

Hầu hết mọi người khi bắt đầu học lập trình web đều nhận được lời khuyên là bắt đầu từ HTML. Tuy nhiên, bản thân HTML không có nhiều tương tác và logic để bạn có thể học.

Có thể bạn không biết, phần lớn những tương tác giữa trang web với người dùng như: hiện popup, hiệu ứng chuyển động, slideshow ảnh hay gửi dữ liệu lên server... chủ yếu được thực hiện bằng Javascript.

Để phát triển các ứng dụng web tương tác với người dùng như vậy, bạn cần phải biết Javascript.

Việc tự học Javascript không hề khó, chỉ cần bạn hiểu được tư tưởng ban đầu của nó, thì việc học sẽ cực dễ dàng.

Cuốn sách này sẽ giúp bạn giải đáp những câu hỏi, những trăn trở khi tự mày mò tìm hiểu Javascript, cũng như trang bị những kiến thức từ nền tảng tới nâng cao về javascript, đặc biệt phù hợp với những người chưa có kinh nghiệm lập trình thực tế.

Điểm xuất phát của bạn có thể rơi vào 2 trường hợp sau:

- **Javascript là ngôn ngữ lập trình đầu tiên mà bạn tiếp cận.** Có thể bạn vô tình được nghe tới Javascript hoặc được bạn bè giới thiệu "Javascript là ngôn ngữ lập trình dễ học nhất".
- **Bạn đã từng học và làm việc trên các ngôn ngữ lập trình khác như Java, C#, Python...** Do điều kiện ngoại cảnh như chuyển dự án, vì nghe lời quảng cáo thần thánh hóa của các cao nhân trên mạng về Javascript. Hoặc có thể bạn có duyên với Javascript mà yêu thích ngôn ngữ lập trình này, muốn tìm hiểu và kết thân với nó.

Dù xuất phát điểm như thế nào, cuốn sách này cũng sẽ giúp bạn hiểu cặn kẽ, sử dụng thành thạo Javascript như một công cụ để xây dựng ứng dụng web, làm nền tảng vững chắc để bạn tìm hiểu các framework front-end như VueJS, ReactJS... hoặc chuyển sang cả mảng Back-end với NodeJS.

"Với Javascript, con đường trở thành full stack developer ngắn hơn bao giờ hết"

--VNTALKING --

Sau một thời gian dài chuẩn bị, mình cho ra đời cuốn sách này. Với mục tiêu: *"Mang cả thế giới Javascript vào giường của bạn, nhằm, vào sự nghiệp của bạn"* 😊

Bạn đã sẵn sàng đắm chìm vào thế giới "ma thuật" của Javascript chưa?

Còn chờ gì nữa. Hãy tiếp tục đọc và nghiền ngẫm, bạn sẽ cảm thấy yêu thích cuốn sách này.

Mình đảm bảo!

NỘI DUNG CUỐN SÁCH

Javascript là ngôn ngữ lập trình phổ biến nhất hiện nay. Giờ đây, bạn chỉ cần biết một ngôn ngữ lập trình là có thể chinh chiến từ Front-end tới Back-end.

Trước khi bạn nghĩ tới những điều lớn lao như xây dựng ứng dụng web kiểu như Tiki, Shopee... học các framework Front-end như ReactJS, VueJS, Angular... hay như chuyển sang học NodeJS để làm Back-end cho hệ thống, bạn cần phải nắm vững và sử dụng thành thạo công cụ, đó chính là ngôn ngữ lập trình Javascript.

Cuốn sách này ra đời để giúp bất kỳ ai, từ người chưa từng có kinh nghiệm lập trình tới người đã có kinh nghiệm mà muốn chuyển sang học Javascript.

Trong cuốn sách này, bạn sẽ được học và thực hành:

- Cú pháp Javascript cơ bản
 - Biến - variables
 - Kiểu dữ liệu
 - Toán tử operators
 - String
 - Làm việc với điều kiện và cấu trúc có điều kiện
- Cú pháp Javascript nâng cao
 - Hàm và cách sử dụng
 - Vòng lặp
- Dữ liệu có cấu trúc Data structure:
 - Object
 - Array
- Higher-Order Function
- Lập trình hướng đối tượng (Object Oriented Programming)
- Làm quen cú pháp và tính năng mới của ES6
- Bài tập thực hành kèm đáp án.

Để đảm bảo các bạn tập trung và hiểu rõ Javascript, mình sẽ không sử dụng bất kỳ thư viện 3rd nào trong cuốn sách này, không trộn lẫn mã nguồn HTML và Javascript.

CUỐN SÁCH NÀY DÀNH CHO AI?

Cuốn sách này rất phù hợp cho những ai yêu thích lập trình, muốn học kiến thức nền tảng để tiếp tục phát triển các ứng dụng web, mobile hay PC bằng Javascript. Nếu bạn có định hướng sự nghiệp thành full stack developer thì cuốn sách này chính là tài liệu đầu tiên mà bạn cần tới.

Đây là cuốn sách "**No Experience Require**", tức là không yêu cầu người có kinh nghiệm lập trình, chưa từng lập trình. Tất cả sẽ được mình hướng dẫn học từ con số 0.

Yêu cầu trình độ

Javascript là một trong bộ ba kỹ thuật để xây dựng trang web gồm: Javascript, HTML, CSS. Do đó, để có thể học Javascript một cách trơn tru nhất, bạn nên biết:

- Kiến thức cơ bản về HTML.
- Biết sử dụng công cụ debug của trình duyệt

Nếu bạn không biết cả hai thứ trên thì sao? Cũng không sao, đọc xong cuốn sách này bạn cũng sẽ biết chúng thôi.

Cách học đúng cách

Cuốn sách này mình chia nhỏ nội dung thành nhiều phần, mỗi phần sẽ giới thiệu một chủ đề riêng biệt, kèm thực hành. Mục đích là để bạn có thể chủ động lịch học, không bị dồn nén quá nhiều, dễ dẫn tới "tẩu hỏa nhập ma", lúc đó lại oán trách mình 😊

Với mỗi phần lý thuyết, mình đều có ví dụ minh họa. Vì vậy, cách học tốt nhất vẫn là vừa học, vừa thực hành. Bạn nên **tự mình gõ lại từng dòng code** và kiểm tra kết quả trên trình duyệt. Đừng copy cả đoạn code trong sách, điều

này sẽ hạn chế khả năng viết code của bạn, cũng như khiến bạn nhiều khi không hiểu vì sao code bị lỗi.

"Nhớ nhé, đọc đến đâu, tự viết code đến đó, tự build và kiểm tra đoạn code đó chạy đúng không"

Ngoài ra, trong cuốn sách này, kiến thức phần sau được xây dựng từ phần trước. Do vậy, bạn đừng đọc lướt mà bỏ sót đoạn nào nhé.

Trong quá trình bạn đọc sách, nếu code của bạn không chạy hoặc chạy không đúng ý muốn mà vất tay lên trán mấy hôm vẫn chưa giải đáp được thì đừng ngần ngại đặt câu hỏi trên Group: **Hỏi đáp lập trình - VNTALKING**

Liên hệ tác giả

Nếu gặp bất kỳ vấn đề gì trong quá trình đọc sách, code bị lỗi hoặc không hiểu, các bạn có thể liên hệ với mình theo bất kỳ kênh nào dưới đây:

- Website: <https://vntalking.com>
- Fanpage: <https://facebook.com/vntalking>
- Group: <https://www.facebook.com/groups/hoidaplaptrinh.vntalking>
- Email: support@vntalking.com
- Github: <https://github.com/vntalking/Book-Javascript>

PHẦN 1

GIỚI THIỆU

Javascript (thường hay viết tắt là JS) là ngôn ngữ lập trình kịch bản (scripting language) cho client-side, sau này còn cho cả server-side (Nodejs)

Javascript được sử dụng chủ yếu để nâng cao sự tương tác của người dùng với trang web. Nói cách khác, bạn có thể làm cho trang web trở nên sinh động và tăng tính tương tác hơn. Trong các ứng dụng web, người ta hay dùng JS để làm các hiệu ứng đặc biệt như sliders, pop-ups, hoặc xác thực dữ liệu các form (form validations) trước khi gửi dữ liệu lên server .v.v...

Ngày nay, Javascript không chỉ giới hạn trong khuôn khổ xây dựng ứng dụng web, mà còn sử dụng rộng rãi trong phát triển ứng dụng, game trên điện thoại hay các ứng dụng dành cho server.

- Web app: ReactJS, VueJS, Angular...
- Mobile app: React Native, Ionic...
- Game: Phaser, Kiwi.js...
- Server app: Nodejs
- Graphic: two.js (2D), three.js (3D)...
- AI: brain.js...

Và còn nhiều nhiều nữa các lĩnh vực mà Javascript có thể làm được. Các bạn cứ bình tĩnh khám phá nhé.

Lịch sử Javascript

Javascript được tạo bởi lập trình viên kỳ cựu Brendan Eich, giới thiệu lần đầu năm 1995, xuất hiện trên trình duyệt Netscape, một trình duyệt phổ biến thời bấy giờ.

Ban đầu, ngôn ngữ lập trình này được gọi là LiveScript, sau này mới đổi tên thành Javascript. Mới đọc tên thì nhiều người sẽ nhầm tưởng Javascript có "họ hàng" với Java. Nhưng thực tế, hai ngôn ngữ này không hề có liên quan gì tới nhau cả, cây gia phả của chúng không hề chung gốc.

Java là ngôn ngữ lập trình hướng đối tượng phức tạp, còn Javascript là một ngôn ngữ kịch bản (scripting language). Cú pháp của Javascript chủ yếu có hơi hướng ảnh hưởng từ ngôn ngữ C.

Tại sao nên học Javascript?

Trước khi bạn quyết định đầu tư học một ngôn ngữ lập trình, đặc biệt với người chưa từng biết một ngôn ngữ lập trình nào, có thể bạn sẽ đắn đo, băn khoăn liệu mình có nên lao đầu vào ngôn ngữ lập trình này không? Liệu tương lai ngôn ngữ này có phát triển hay không?

Dưới đây là một số lý do để bạn bỏ công sức đầu tư học Javascript.

#1- Là ngôn ngữ lập trình phổ biến nhất

Khi bạn định hướng nghề nghiệp trong tương lai, việc chọn một ngôn ngữ lập trình phổ biến để theo đuổi là lựa chọn không hề tồi chút nào. Cũng giống như bạn đi kinh doanh, bán hàng vậy. Không ai dại gì lại đi bán mặt hàng mà thị trường không có nhu cầu sử dụng cả.

Theo một khảo sát mới nhất của Stackoverflow.com (website hỏi đáp dành cho lập trình viên lớn nhất thế giới) cho thấy, Javascript là ngôn ngữ lập trình phổ biến nhất, được rất nhiều lập trình viên chuyên nghiệp tin tưởng.

Không chỉ front-end, ngay cả các dự án back-end cũng ngày càng lựa chọn Javascript nhiều hơn.

#2- Javascript rất dễ học

Với tính mềm dẻo, linh hoạt, Javascript rất dễ học, đặc biệt là cho người mới học lập trình. Javascript biến các chi tiết phức tạp thành các bản tóm tắt, giúp mọi thứ trở nên dễ dàng hơn với người mới.

Không giống như các ngôn ngữ lập trình bậc cao khác, Javascript mang nhiều cảm giác về ngôn ngữ tự nhiên hơn. Tức là bạn nói sao thì viết như vậy.

#3- Tài nguyên học có sẵn rất nhiều

Khi tiếp cận bất kỳ kỹ thuật mới nào, việc quan trọng đầu tiên phải nghĩ tới đó là tài liệu hướng dẫn có đầy đủ không! Mình từng tham dự một dự án mà sử dụng một framework cổ xưa, tài liệu chính chủ còn không có (chắc tác giả cũng bỏ rơi nó luôn), lúc đó mới thấu hiểu nỗi đau khổ khi không có tài liệu.

Với ngôn ngữ Javascript nói chung, các JS frameworks như React, Vue... nói riêng thì đều có tài liệu hướng dẫn rất chi tiết và đầy đủ (cả chính chủ lẫn của cộng đồng). Do đó, bạn sẽ không gặp phải bất kỳ khó khăn nào trong việc tìm kiếm tài liệu hỗ trợ bạn trong việc học.

Ngoài ra, trên internet còn có hàng ngàn tutorial miễn phí để bạn có thể tham khảo. Tuy nhiên, việc dễ dàng tiếp cận hàng ngàn tài liệu cũng khiến bạn dễ bị bối rối, hỗn loạn kiến thức. Đó là lý do bạn tìm tới cuốn sách này.

#4- Một ngôn ngữ cho tất cả

Nếu trước đây, Javascript được sinh ra chỉ để xây dựng các trang web, thì nay đã khác. Javascript giờ đây có thể xây dựng mọi ứng dụng từ client-side tới back-end, các ứng dụng/game mobile, ứng dụng trên PC, kể cả các ứng dụng trên cloud, AI (Trí tuệ nhân tạo)...

Do đó, thay vì bạn phải đầu tư học rất nhiều ngôn ngữ, giờ bạn chỉ cần tập trung học Javascript cho thật tốt là đủ "cân cả bản đồ".

#5- Tiềm năng phát triển sự nghiệp lớn

Với việc ngày càng có nhiều doanh nghiệp và tổ chức chuyển sang sử dụng Javascript cho sản phẩm của mình. Do đó, nhu cầu tuyển dụng lập trình viên Javascript cũng tăng lên rất nhiều.

Theo một khảo sát của Devskiller.com, 70% các công ty công nghệ muốn tuyển một lập trình viên Javascript.

Ở Việt Nam thì sao? Đảo qua một loạt các trang tuyển dụng lớn như Vietnamworks, ITviec... nhu cầu tuyển lập trình viên React, Angular, Vue, NodeJS... rất nhiều, mức lương cũng rất cao (toàn trên 2k\$ cho một senior developer).

Tóm lại, theo đánh giá của mình, việc chọn Javascript là ngôn ngữ lập trình chính cho sự nghiệp là một lựa chọn đáng giá, xứng đáng với mồ hôi nước mắt.

Ok, giờ là lúc chúng ta cùng nhau chinh phục Javascript thôi!

Nội dung chính

- >> Ưu điểm của ngôn ngữ lập trình Javascript
- >> Giới hạn của Javascript
- >> Công cụ để phát triển ứng dụng với JS
- >> Tạo ứng dụng HelloWorld đầu tiên

PHẦN 2

TỔNG QUAN JAVASCRIPT

Rất nhiều bạn mới học sẽ cảm thấy đôi chút thất vọng khi nghe đâu đó có người nói Javascript là ngôn ngữ lập trình dành cho trẻ con. Thực tế thì họ đã nhầm! Để mình chỉ cho bạn thấy.

Đầu tiên, Javascript là một trong những ngôn ngữ lập trình mạnh nhất hiện nay. Nó là một kỹ năng mà mọi lập trình viên cần phải có nếu muốn theo sự nghiệp web development (ít nhất là vậy).

Cái hay của Javascript nằm ở chỗ cách viết đơn giản để giải quyết một vấn đề phức tạp.

Ưu điểm của Javascript

Để nói về ưu điểm của một ngôn ngữ lập trình thì có thể kể hàng chục trang giấy. Bởi vì, mỗi ngôn ngữ được tạo ra, tác giả đều sẽ cố gắng tối ưu, thiết kế sao cho tốt nhất. Không phải ngẫu nhiên mà Javascript được cộng đồng đón nhận rộng rãi đến như vậy.

Tuy vậy, để bạn hiểu được thế mạnh của Javascript, mình sẽ liệt kê một số nét đặc trưng khiến Javascript nổi bật hơn các ngôn ngữ lập trình khác.

#1- Tích hợp sẵn trong hầu hết các trình duyệt

Không giống như nhiều ngôn ngữ phát triển web khác, ví dụ flash, Java... người dùng muốn sử dụng được thì phải cài đặt thêm plugin cho trình duyệt.

Javascript thì khác, hầu hết trình duyệt hiện đại đều đã tích hợp sẵn. Do đó, việc bạn sử dụng JS để phát triển ứng dụng sẽ rất thuận lợi.

#2- Một ngôn ngữ lập trình vô cùng linh hoạt

Rất nhiều lập trình viên thích trường phái functional programming. Functional Programming một phương pháp lập trình dựa trên các hàm toán học (function), tránh việc thay đổi giá trị của dữ liệu. Nó có nhiều lợi ích như : các khối xử lý độc lập dễ tái sử dụng, thuận lợi cho việc thay đổi logic hoặc tìm lỗi chương trình.

Javascript là ngôn ngữ sinh ra là để dành cho functional programming.

Hai trong số tính năng nổi bật nhất của Javascript là cho phép gán một hàm cho bất kỳ biến nào và tạo một hàm chấp nhận tham số là một hàm khác.

#3- Khả năng tự detect trình duyệt và hệ điều hành

Đôi khi, trong một số ứng dụng, bạn gặp vấn đề và cần phải viết mã nguồn tương thích với từng trình duyệt web hoặc hệ điều hành. Javascript được thiết kế để có thể tự nhận biết được chạy trên trình duyệt gì, hệ điều hành nào.

Điều này, cho phép bạn dễ dàng điều chỉnh mã nguồn để ứng dụng đáp ứng và tương thích với mọi hệ điều hành.

#4- Hỗ trợ cả lập trình hướng đối tượng (OOP)

Lập trình hướng đối tượng cũng là một trường phái lập trình rất phổ biến, khi tất cả mọi thứ của ứng dụng đều xoay quanh đối tượng (Object - Class).

Javascript cung cấp rất nhiều công cụ để bạn làm việc với đối tượng, đồng thời nó cũng dễ học, dễ sử dụng.

Nói một cách chính xác hơn, có thể coi Javascript là ngôn ngữ dựa trên đối tượng, vì lý do:

- Không hỗ trợ đầy đủ các đặc điểm của OOP như: đa hình, kế thừa
- Có sẵn kiểu dữ liệu đối tượng. Ví dụ: JavaScript có sẵn đối tượng window...

Trong cuốn sách này, chúng ta cũng sẽ tìm hiểu cách [lập trình hướng đối tượng bằng Javascript](#). Đừng bỏ qua nhé.

#5- Học một ngôn ngữ dùng mọi nơi

Phần này mình chỉ nhắc lại thôi. Nếu trước đây, Javascript được tạo ra chỉ để phát triển các ứng dụng front-end chạy trên trình duyệt thì giờ đây mọi chuyện đã khác. Có thể bạn chưa biết, trước đây một full stack developer cần phải học rất nhiều ngôn ngữ lập trình. Có thể kể tên nhẹ nhàng như: front-end thì có javascript, back-end thì có PHP, JAVA, Ruby, Golang..., ứng dụng mobile thì có Java, Kotlin, Swift... rất nhiều ngôn ngữ phải học.

Nhưng ngày nay, bạn chỉ cần học duy nhất Javascript là đủ. Biết Javascript, bạn có thể xây dựng các ứng dụng web (cái này tất nhiên rồi), xây dựng ứng dụng phía back-end (nhờ Node.JS), xây dựng ứng dụng mobile (React Native, Ionic...)

Giới hạn của Javascript

Ưu điểm thì nhiều, nhưng không phải là không có nhược điểm gì cả. Javascript thiếu một số tính năng cần thiết như:

- Vì lý do bảo mật, Javascript client-side không thể đọc và ghi file.
- Javascript chỉ xử lý đơn luồng.
- Javascript không hỗ trợ cho xử lý trên nhiều nhân của CPU. Dù CPU có nhiều nhân thì JS cũng chỉ dùng tới một nhân.

Ngoài ra, còn một số nhược điểm liên quan tới bảo mật nữa, nhưng chúng ta không đề cập nhiều ở cuốn sách này, nó thuộc phạm trù khác rồi.

Công cụ phát triển

Có lẽ JS là ngôn ngữ dành cho anh em nhà nghèo thì phải. Không cần phải đầu tư nhiều tiền để sắm IDE, tool toy gì cả.

Để viết code Javascript, bạn chỉ cần một trình soạn thảo văn bản như Notepad, Notepad++. Sang hơn thì có Visual Studio Code, Sublime Text... Và một trình duyệt để hiển thị trang web bạn phát triển, ví dụ Firefox, Chrome, Edge.v.v... Tất cả những công cụ này đều miễn phí. Ngoài ra, bạn có thể code trực tiếp trên các Text Editor online như: [Codesandbox.io](https://codesandbox.io/), [Playcode.io](https://playcode.io/)...

Trong cuốn sách này, mình sẽ chủ yếu sử dụng hai trang này để viết code minh họa.

Thực thi chương trình Javascript

Các ngôn ngữ lập trình có hai trường phái thực thi chương trình, đó là:

- Biên dịch - compiled
- Thông dịch - interpreted

Nhóm ngôn ngữ kiểu biên dịch gồm có: C++, Java, C Fortran và COBOL,v.v...

Ngôn ngữ lập trình được xác định là biên dịch khi mã nguồn được viết bởi lập trình viên, sau đó mã này sẽ được chạy thông qua một chương trình đặc biệt, gọi là **compiler**. Nhiệm vụ của trình compiler là dịch toàn bộ mã nguồn thành mã máy (ngôn ngữ mà máy tính có thể hiểu được) sau đó mới thực thi chương trình.

Javascript thuộc vào nhóm ngôn ngữ thứ 2, đó là thông dịch. Nhóm này gồm có: Javascript, PHP, Ruby, Haskell and Perl,v.v...

Với ngôn ngữ kiểu thông dịch, mã nguồn vẫn được viết bởi lập trình viên nhưng nó không cần phải chạy qua trình compiler, thay vào đó, mã nguồn được dịch thẳng sang mã máy để máy tính thực thi. Khi chương trình chạy đến dòng lệnh nào sẽ chuyển thành mã máy đến đó để máy tính có thể thực thi.

Ưu điểm của cách chạy chương trình kiểu thông dịch là quá trình chạy ngắn hơn, không cần qua trung gian là trình compiler. Ngoài ra, lập trình viên có khả năng cập nhập và thực hiện các thay đổi trong chương trình bất kỳ lúc nào, bất kỳ chỗ nào mà không cần chờ đợi compile lại cả chương trình.

Trên thực tế, trình duyệt sẽ chịu trách nhiệm thực thi mã nguồn Javascript. Khi người dùng yêu cầu một trang HTML có javascript, các tập lệnh JS sẽ được gửi tới trình duyệt và trình duyệt sẽ thực thi nó.

Tạo chương trình Javascript đầu tiên

Trong toàn bộ nội dung cuốn sách này, chúng ta sẽ chỉ sử dụng Javascript là ngôn ngữ xây dựng ứng dụng client (front-end) trên trình duyệt. Còn Javascript để tạo ứng dụng trên server (sử dụng NodeJS) sẽ đề cập trong cuốn sách khác. Mặc dù, cơ bản cú pháp là như nhau nhưng môi trường để chạy JS sẽ có khác nhau đôi chút.

Để chạy một chương trình Javascript, bạn cần đưa mã JS vào trong một trang HTML. Có hai cách để thêm mã vào trang HTML:

- Cách 1: Viết mã lệnh JavaScript trực tiếp vào trang web.
- Cách 2: Viết mã lệnh JavaScript vào tập tin JavaScript (*tập có phần đuôi là .js*)

Với các ứng dụng web thực tế, chủ yếu sử dụng cách 2 để thêm JS. Tuy nhiên, mình vẫn sẽ giới thiệu cả hai cách trong cuốn sách này.

Cách 1: Viết mã lệnh trực tiếp vào trang web

Bạn đặt toàn bộ mã Javascript vào trong thẻ `<script>`. Điều này giúp trình duyệt phân biệt mã JS với phần còn lại. Ngoài ra, vì có các ngôn ngữ kịch bản cho client khác nữa (ví dụ: VBScript...), do đó, bạn nên chỉ định ngôn ngữ kịch bản mà bạn sử dụng trong thẻ `<script>`, kiểu như sau:

```
<script type="text/javascript">
```

Bạn tạo mới một tệp html đặt tên là `index.html`, có nội dung như sau:

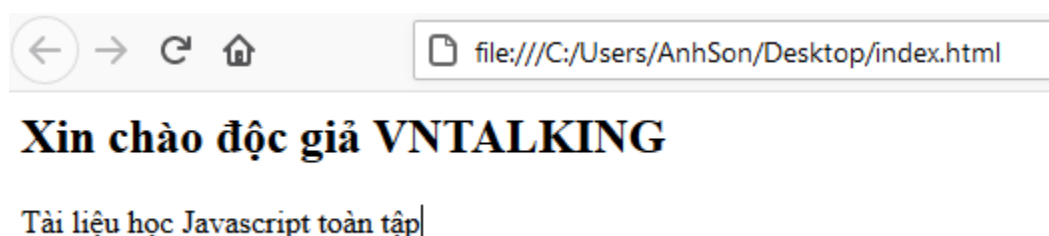
```
<html lang='en'>
  <head>
    <meta charset='UTF-8' />
    <title>Sách học lập trình Javascript - by VNTALKING</title>
  </head>
  <body>
```

```

<h2>Xin chào độc giả VNTALKING</h2>
<span id='content'></span>
</body>
<script type='text/javascript'>
    document.getElementById("content").innerHTML = "Tài liệu học Javascript t
oàn
    tập";
</script>
</html>;

```

Bạn lưu lại và mở tệp lên bằng trình duyệt. Đây là kết quả thu được.



Hình 2.1: Chương trình hello world bằng javascript

Cách 2: Viết mã lệnh JavaScript vào tệp tin JavaScript

Với cách viết Javascript trực tiếp vào trong HTML sẽ khiến cho mã nguồn trở nên "rối rắm", đặc biệt khi dự án ngày càng trở nên phức tạp. Đặc biệt là trong trường hợp chúng ta muốn xây dựng các plugin/module bằng javascript, tách biệt với HTML.

Chúng ta sẽ tách mã JS ra khỏi HTML, và chỉ nhúng đường dẫn trỏ tới tệp .js trong HTML mà thôi.

```

<html lang='en'>
<head>
    <meta charset='UTF-8' />
    <title>Sách học lập trình Javascript - by VNTALKING</title>
</head>
<body>
    <h2>Xin chào độc giả VNTALKING</h2>
    <span id='content'></span>
</body>

```

```
<script src='src/index.js'></script>
</html>;
```

Còn nội dung tệp *index.js* như sau:

```
document.getElementById("content").innerHTML =
"Tài liệu học Javascript toàn tập";
```

Cách 3 (Optional): Sử dụng IDE online.

Để học Javascript, thay vì sử dụng hai cách trên, mình thường tìm tới các Text Editor Online để viết và chạy thử chương trình Javascript được nhanh và trực quan nhất.

Một số Text Editor Online gợi ý cho bạn:

- [Codesandbox.io](https://codesandbox.io)
- [Playcode.io](https://playcode.io)
- [Jsfiddle.net](https://jsfiddle.net)

Ưu điểm của những trình Text Editor Online này là dễ sử dụng, được tích hợp sẵn mọi thứ, không cần cài đặt thêm gì cả. Việc của bạn chỉ là gõ code và chạy.

Trong cuốn sách này, mình chọn codesandbox.io để chạy các đoạn code minh họa. Lý do đơn giản là bởi vì IDE online này nó thiết kế rất giống với Visual Studio Code, các tính năng cũng tương tự, có thể kết nối tới Github để lưu và lấy code.



Hình 2.2: Cách chạy code Javascript trực tiếp trên các Text Editor Online

Hai đoạn code minh họa trên, mình có tạo trên codesandbox, bạn có thể tham khảo:

- <https://codesandbox.io/s/js-hello-world-cach-1-ijw66>
- <https://codesandbox.io/s/kind--b3upj>

Đọc đến đây, mình hi vọng bạn đã hiểu được phần nào về ngôn ngữ lập trình Javascript, biết cách tạo một ứng dụng đơn giản với Javascript. Phần tiếp theo, chúng ta sẽ cùng nhau tìm hiểu các cú pháp từ cơ bản tới nâng cao của Javascript nhé.

PHẦN 3

CÚ PHÁP JAVASCRIPT CƠ BẢN

Theo các hiểu thông thường, một chương trình máy tính là một tập danh sách các hướng dẫn (có thể gọi là statements) để máy tính thực thi. Và Javascript cũng vậy, chỉ khác một điều là các tập lệnh này cho trình duyệt thực hiện thay vì máy tính.

Sau này, khi tìm hiểu các ngôn ngữ lập trình khác, bạn cũng sẽ thấy về cơ bản thì chúng cũng na ná như nhau mà thôi. Tất cả cũng chỉ xoay quanh về biến, hàm, kiểu dữ liệu, các toán tử tính toán và so sánh, mảng, vòng lặp.v.v... Có chăng, chỉ khác nhau đôi chút về cách viết mà thôi, hay còn gọi là cú pháp (syntax).

Đó là lý do tại sao người ta vẫn truyền tai nhau rằng: chỉ cần bạn nắm vững một ngôn ngữ lập trình, sau đó muốn chuyển sang một ngôn ngữ khác sẽ rất dễ dàng.

Chúng ta sẽ cùng nhau tìm hiểu cú pháp cơ bản của Javascript ngay bây giờ.

Variable - Biến

Hiểu đơn giản, variable - biến là tên nơi lưu trữ dữ liệu. Chúng ta sử dụng biến để lưu trữ giá trị (name = "anh sơn") hoặc biểu thức (sum = x + y).

Trước khi có thể sử dụng được biến, bạn cần phải khai báo nó. Bạn có thể sử dụng từ khóa *var* hoặc *let* để khai báo một biến.

Đoạn code dưới đây, chúng ta khai báo một biến có tên là message:

```
let message;
```

Giờ bạn có thể đưa thông tin vào biến bằng cách sử dụng toán tử "="

```
let message;  
message = "Xin chào độc giả VNTALKING quý mến";
```

Giờ đây thì giá trị của string được lưu trữ trong bộ nhớ và liên kết với biến. Bạn có thể sử dụng nó thông qua tên biến.

```
let message;  
message = "Xin chào độc giả VNTALKING quý mến";  
  
alert(message);
```

Nếu bạn không thích dài dòng, bạn có thể gán giá trị ngay khi khai báo biến.

```
let message = "Xin chào độc giả VNTALKING quý mến";  
alert(message);
```

Ngoài ra, bạn cũng có thể khai báo nhiều biến trên cùng một dòng cũng được.

```
let user = "Son Duong", age = 25, message = "Xin chào";
```

Nhìn có vẻ ngắn gọn hơn đấy, nhưng mình không khuyến khích cách viết này. Nên viết khai báo mỗi biến trên một dòng để dễ đọc hơn.

```
let user = "Son Duong";  
let age = 25;  
let message = "Xin chào";
```

Hoặc viết thế này cũng được:

```
let user = "Son Duong",  
    age = 25,  
    message = "Xin chào";
```

Như mình đã đề cập ở trên, bạn có thể sử dụng *var* để khai báo biến.

```
var message = "Xin chào độc giả VNTALKING quý mến";
```


Vậy sự khác nhau giữa *var* và *let* như thế nào? Để thấy được sự khác biệt giữa *var* và *let*, chúng ta cần hiểu một khái niệm khác nữa, đó là phạm vi của một biến: global và local variables. Sau khi tìm hiểu xong khái niệm này, mình chỉ quay trở lại vấn đề của *var* và *let* nhé.

Bạn có thể tham khảo code online tại đây: <https://codesandbox.io/s/variable-4vq2v>

Variable Scope - Phạm vi sử dụng của một biến

Trong Javascript, scope hay hiểu là phạm vi truy cập của một biến. Scope là phạm vi mà biến có thể truy cập được qua tên trực tiếp. Ở ngoài scope, biến đó sẽ không thể nhìn và truy cập được một cách trực tiếp nữa.

Một biến có 2 loại scope:

- **Local variable:** Các biến không thể truy cập được ở bất kỳ nơi nào khác trong dự án, ngoại trừ trong hàm mà biến đó được khai báo.
- **Global variable:** là các biến được khai báo bên ngoài hàm, có thể được truy xuất ở bất cứ đâu trong dự án.

Mình lấy một ví dụ như sau: Bạn tưởng tượng toàn bộ mã nguồn dự án là một khách sạn với rất nhiều phòng.

- Local variable tương ứng là các đồ dùng trong mỗi phòng.
- Global variable tương ứng là các đồ dùng ở khu vực chung như máy giặt, bể bơi, nhà ăn...

Mỗi khách thuê phòng nào thì chỉ được sử dụng đồ đạc trong phòng đó và các đồ dùng ở khu vực chung mà thôi. Họ không thể sử dụng các đồ đạc ở các phòng bên cạnh (động vào chết ngay 😊)

Tương tự, các khách thuê phòng khác, kể cả nhân viên của khách sạn cũng không có quyền truy cập, sử dụng đồ đạc trong các phòng khách sạn.

Chúng ta cùng nhau xem ví dụ bên dưới đây:

```
var x = 10; //Đây là global variable bởi vì nó không khai báo trong hàm
var turnToZero = function (number) {
```

```
var y = 5; // Đây là local variable bởi vì nó khai báo trong hàm
number = 0;
console.log("Giá trị của biến x là: " + x);
console.log("Giá trị của biến y là: " + y);
};
```

```
turnToZero(x);
console.log("Bên ngoài hàm, giá trị của biến x là: " + x);
```

Chạy thử và kiểm tra kết quả nhé.



Hình 3.1: Variable scope

Bạn có thấy, biến "x" vẫn có thể truy cập bên trong hàm `turnToZero()`. Còn biến "y" thì sao? Bạn không thể nào truy cập biến này bên ở bên ngoài hàm `turnToZero()`. Không tin? Chúng ta thử nhé.

```
var x = 10; // Đây là global variable bởi vì nó không khai báo trong hàm
var turnToZero = function (number) {
var y = 5; // Đây là local variable bởi vì nó khai báo trong hàm
number = 0;
console.log("Giá trị của biến x là: " + x);
console.log("Giá trị của biến y là: " + y);
};
```

```
turnToZero(x);
console.log("Bên ngoài hàm, giá trị của biến x là: " + x);
console.log("Bên ngoài hàm, giá trị của biến y là: " + y);
```

Chạy thử và xem kết quả.



Hình 3.2: Truy xuất local variable bên ngoài phạm vi

Ở đây có một điểm khiến nhiều bạn bị nhầm lẫn: Biến "x" được truyền vào trong hàm thì sau khi kết thúc hàm, giá trị của nó có bị thay đổi không?

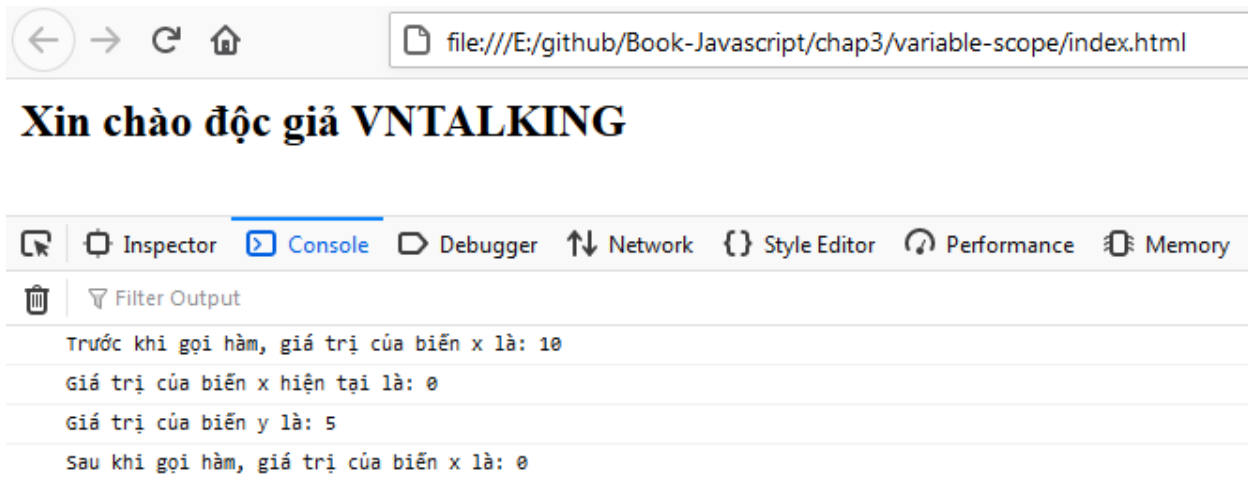
```
turnToZero(x);
```

Thực chất là bạn chỉ copy giá trị của biến "x" vào trong hàm, đặt nó vào biến tạm thời "number". Điều này có nghĩa là, khi bạn thực hiện thay đổi giá trị biến "number", bạn thực không làm gì đến biến "x" cả.

Do đó, bạn không cần phải truyền một biến global vào trong một hàm (thông qua tham số) để thay đổi giá trị của nó. Thay vào đó, bạn làm như sau:

```
var x = 10; // Đây là global variable bởi vì nó không khai báo trong hàm
var turnToZero = function () {
var y = 5; // Đây là local variable bởi vì nó khai báo trong hàm
  x = 0;
  console.log("Giá trị của biến x hiện tại là: " + x);
  console.log("Giá trị của biến y là: " + y);
};
console.log("Trước khi gọi hàm, giá trị của biến x là: " + x);
turnToZero();
console.log("Sau khi gọi hàm, giá trị của biến x là: " + x);
```

Kết quả thu được như hình dưới đây:



Hình 3.3: Cách thay đổi giá trị biến global

Đến đây, bạn đã hiểu scope của một biến rồi đúng không?

Bạn có thể xem code online tại đây: <https://codesandbox.io/s/variable-scope-9u8tf>

Sự khác nhau giữa *var* và *let*

Quay trở lại câu hỏi: "*Sự khác nhau giữa var và let như thế nào?*"

Câu trả lời: sự khác nhau chính giữa *var* và *let* là phạm vi truy cập của chúng.

- Phạm vi của biến số sử dụng *var* là phạm vi hàm số hoặc bên ngoài hàm số, toàn cục.
- Phạm vi của biến số sử dụng *let* là phạm vi một khối, xác định bởi cặp {}

Chúng ta cùng xem xét ví dụ sau (code online tại đây:

<https://codesandbox.io/s/frosty-https-zu9de>):

```
function viduvar() {  
  for (var i = 0; i < 3; i++) {  
    console.log("i bên trong vòng lặp: ", i); // kết quả: 0, 1, 2  
  }  
  
  console.log("i bên ngoài vòng lặp ", i); //Kết quả: 3  
}  
  
function vidulet() {
```

```
for (let j = 0; j < 3; j++) {
  console.log("j bên trong vòng lặp: ", j); //Kết quả: 0, 1, 2
}
console.log("j bên ngoài vòng lặp: ", j); //ReferenceError: j is not defined
}

viduvar();
vidulet();
```

Đến đây, bạn đã hiểu được sự khác nhau giữa chúng rồi đúng không?

Khái niệm và cơ chế Hoisting

JS là ngôn ngữ lập trình kiểu thông dịch. Mỗi khi chạy ứng dụng, bộ thông dịch trong JS engine sẽ thực thi các câu lệnh từ trên xuống dưới. Tuy nhiên, trong Javascript tồn tại thêm một cơ chế rất đặc biệt nữa, đó là khái niệm “Hoisting” (hay “hoisted”)

Hoisting là cơ chế chỉ có trong Javascript, nó khiến cho nhiều bạn mới tiếp cận JS cảm thấy bối rối, khó hiểu khi gặp lỗi liên quan đến hoisting.

Khái niệm

Bất cứ khi nào chương trình của bạn chạy, Javascript sẽ phân tích cú pháp mã nguồn trước khi thực thi. Trong giai đoạn phân tích này, trình phân tích sẽ kiểm tra từng dòng mã, nếu phát hiện bất kỳ lỗi lầm nào, chương trình sẽ dừng.

Giả sử mã nguồn của bạn OK, không có sai sót nào cả, trình thông dịch Javascript sẽ di chuyển các hàm, biến được khai báo lên đỉnh của mã nguồn. Theo cách này, chúng ta có thể sử dụng các biến hay hàm trước khi đến đoạn khai báo nó trong mã nguồn.

Vậy cơ chế hoisting là gì? Hoisting là cơ chế mà trình thông dịch di chuyển tất cả biến, hàm được khai báo lên đầu mã nguồn. Bất kể phạm vi của chúng là toàn cục (global) hay cục bộ (local), chúng đều được trình thông dịch đưa lên đầu (trong phạm vi của chúng).

Cơ chế hoạt động của hoisting

Chúng ta sẽ cùng tìm hiểu sự khác nhau của cơ chế hoisting giữa *var* và *let*.

1. Sử dụng từ khóa *var*:

```
language = 'Cơ chế hoisting trong javascript';  
console.log(language);  
var language;  
//kết quả in ra dòng: Cơ chế hoisting trong javascript
```

Cơ chế hoisting sẽ đưa các biến được khai báo bằng từ khóa *var* lên đầu và khởi tạo giá trị *undefined*. Giá trị *undefined* sẽ tồn tại cho tới khi trình thông dịch gặp dòng lệnh gán giá trị cho biến đó.

Đoạn code trên sẽ tương tự như sau:

```
var language = undefined;  
language = 'Cơ chế hoisting trong javascript';  
console.log(language);  
// Kết quả in ra dòng: Cơ chế hoisting trong javascript
```

2. Sử dụng từ khóa *let*

Trong trường hợp bạn khai báo một biến nhưng sử dụng từ khóa *let* thì sao?

Chúng ta cùng xem xét đoạn code sau:

```
console.log(name);  
let name = 'vntalking';  
//error: ReferenceError: can't access lexical declaration 'name' before initialization
```

Chuyện gì đang xảy ra vậy? từ khóa let không hỗ trợ hoisting sao?

Câu trả lời là: các biến được khai báo bằng *let* vẫn được cơ chế hoisting đưa lên đầu mã, nhưng chúng không được khởi tạo. Đây chính là điểm khác biệt.

Các biến được khai báo bằng từ khóa *var* sẽ được khởi tạo thành *undefined*. Còn với *let* thì không.

Kiểu dữ liệu

Không giống như "người anh em" nhìn tên có vẻ giống nhưng chẳng họ hàng gì - JAVA, kiểu dữ liệu trong Javascript khá là mềm dẻo và mơ hồ.

Tại sao mình lại nói như vậy? Bạn sẽ biết ngay sau đây.

Nếu như JAVA, một biến khi khai báo, bắt buộc phải thuộc một kiểu dữ liệu nào đó (kiểu int, double, String, hay Object...). Trong Javascript thì khác, lúc khai báo không phải biết kiểu dữ liệu của nó là gì.

Có một điểm đặc biệt ở Javascript mà bạn phải luôn nhớ: **biến không có kiểu, nhưng giá trị của biến thì có kiểu dữ liệu.**

Chính vì điều đó mà không chỉ vì một biến ban đầu được gán trị là một chuỗi ký tự mà sau này vĩnh viễn nó chỉ có thể chứa các chuỗi ký tự.

Ví dụ:

```
let foo = "bar";  
// biến `foo` giờ đang có giá trị là một chuỗi ký tự (string)  
  
foo = false;  
// Giờ biến `foo` lại có giá trị là kiểu boolean  
  
foo = 1337;  
// Bây giờ thì biến `foo` có giá trị là kiểu number
```

Đoạn mã trên hoàn toàn hợp lệ, vì trong Javascript, các biến không có kiểu. Các biến có thể chứa giá trị tùy ý.

Cứ hình dung các biến như là cái hộp được gán nhãn mà nội dung của nó có thể thay đổi theo thời gian (nó có thể chứa nước, hoặc bột gạo, sắt thép...).



Tóm lại, chúng ta chỉ có khái niệm kiểu dữ liệu của giá trị, chứ không có kiểu dữ liệu của biến.

Về cơ bản, mỗi giá trị có một trong 7 kiểu dữ liệu sau:

- number,
- string,
- boolean
- object
- null
- undefined
- symbol (được thêm từ bản ECMAScript 2015 hay còn biết tới với tên quen thuộc ES6)

Chúng ta sẽ cùng nhau tìm hiểu kỹ hơn các kiểu dữ liệu này.

Numbers

Javascript chỉ chứa một kiểu số, gọi chung là number (không có phân chia thành *int*, *float*, *double* như JAVA). Kiểu number này có thể là số nguyên, số thập phân, số âm, số dương...

```
var x1 = 56.00; // có dấu phẩy
var x2 = 56; // không có dấu phẩy
```

Để khai báo một số rất lớn hoặc rất nhỏ, bạn có thể sử dụng dấu phẩy động

```
var y = 234e5; // 23400000
var z = 234e-5; // 0.00234
```

String

String là chuỗi các ký tự đặt trong dấu ngoặc kép hoặc ngoặc đơn.

Ví dụ:

```
var carName = "Ford XS80"; // OK
var bikeName = 'SH XS80'; // OK
```

Ngoài ra, dấu ngoặc kép hay đơn có thể sử dụng kết hợp với nhau.

```
var answer = "It's ok"; // một dấu ngoặc đơn trong ngoặc kép
var answer = "He is called 'Billy'"; // Cặp ngoặc đơn trong ngoặc kép
var answer = 'He is called "Billy"'; // Cặp ngoặc kép trong ngoặc đơn
```


Boolean

Kiểu dữ liệu boolean rất đơn giản, chỉ có hai giá trị: *true* hoặc *false*.

```
var x = true;
var y = false;
```

Object

Object là kiểu dữ liệu kết hợp các kiểu dữ liệu bên trên, được khai báo bằng dấu ngoặc {}, trong đó các thuộc tính được khai báo theo cặp *<name: value>*

Ví dụ:

```
var person = {
  firstName: "Sơn",
  lastName: "Dương Anh",
  age: 30,
  eyeColor: "black",
  hasChild: true
};
```

Để truy xuất vào giá trị của mỗi thuộc tính trong Object, bạn đơn giản chỉ cần gọi nó như sau:

```
console.log(person.firstName); // Sơn
console.log(person.age); // 30
console.log(person.hasChild); // true
```

Riêng Object, chúng ta sẽ tìm hiểu kỹ hơn ở phần sau của cuốn sách ([Phần 5: Dữ liệu có cấu trúc](#)).

Undefined và null

Chắc hẳn bạn sẽ cảm thấy khá bối rối về hai giá trị đặc biệt này đúng không? Nhìn qua thì có vẻ cả hai đều ám chỉ đến giá trị "không có gì cả".

Đúng như bạn nghĩ, thực tế hai giá trị này khá tương đồng nhau, có chăng chỉ khác nhau về khái niệm thôi.

Undefined có nghĩa là không xác định, khi một biến được khai báo mà chưa gán giá trị thì mặc định giá trị của nó là *Undefined*.

```
var person;  
console.log(person); // undefined
```

Bạn muốn reset một biến bất kỳ thì cứ thiết lập nó về *undefined* là được.

```
var person = undefined;  
console.log(person); // undefined
```

Null là giá trị rỗng hoặc giá trị không tồn tại.

```
var person = null;  
console.log(person); // null
```

Vậy sự khác nhau giữa undefined và null là gì?

Chúng ta cần đọc lại về biến và giá trị thì bạn sẽ hiểu sự khác nhau mà mình nói ở đây.

- **undefined đại diện cho biến**, một biến được khai báo nhưng không trở đến bất kì giá trị nào cả.
- **null đại diện cho giá trị**, ám chỉ một giá trị không tồn tại trong vùng nhớ.

Khi bạn thử sử dụng từ khóa *typeof* để kiểm tra sẽ thấy:

```
console.log(typeof undefined) // undefined  
console.log(typeof null) // object
```

Nếu bạn so sánh hai giá trị này thì sao?

```
console.log(null === undefined) // false  
console.log(null == undefined) // true
```

Toán tử - Operators

Về cơ bản, các toán tử trong Javascript hoàn toàn giống với các ngôn ngữ lập trình bậc cao khác thôi.

Cú pháp:

`<toán hạng trái> toán tử <toán hạng phải>`
`<toán hạng trái> toán tử`

Có thể phân loại các toán tử trong javascript thành 5 loại sau:

- Toán tử số học
- Toán tử so sánh
- Toán tử logic
- Toán tử gán
- Toán tử điều kiện rút gọn

Toán tử số học

Là những toán tử để thực hiện các phép tính toán giữa các số học.

Toán tử	Giải thích
+	Phép tính cộng
-	Phép tính trừ
*	Phép nhân
/	Phép chia
%	Phép chia lấy phần dư
++	Tăng giá trị toán hạng lên 1.
--	Giảm giá trị toán hạng đi 1.

Chúng ta cùng xem ví dụ minh họa bên dưới đây để hiểu rõ hơn.

```
var x = 5, y = 10, z = 15;
```

```
console.log(x + y); //returns 15  
console.log(y - x); //returns 5  
console.log(x * y); //returns 50  
console.log(y / x); //returns 2
```

```
console.log(x % 2); //returns 1
console.log(x++); //returns 6
console.log(x--); //returns 4
```

Riêng toán tử `+`, bạn có thể thực hiện trên các kiểu dữ liệu khác nhau. Chính xác thì toán tử này sẽ tiến hành cộng chuỗi *string* nếu trong số các toán hạng mà có toán hạng có kiểu *string*.

```
var a = 5, b = "Xin chào ", c = "đọc giả!", d = 10;
```

```
console.log(a + b); // "5Xin chào "
console.log(b + c); // "Xin chào đọc giả!"
console.log(a + d); // 15
```

Toán tử so sánh

Javascript có một số toán tử để so sánh hai toán hạng, **trả về giá trị kiểu boolean (true/false)**

Toán tử	Giải thích
<code>==</code>	So sánh hai toán hạng (chỉ xét giá trị), bỏ qua kiểu dữ liệu
<code>===</code>	So sánh hai toán hạng xét cả kiểu dữ liệu.
<code>!=</code>	So sánh khác nhau nhau. Nếu hai toán hạng khác nhau thì trả về true.
<code>></code>	So sánh lớn hơn
<code><</code>	So sánh nhỏ hơn
<code>>=</code>	So sánh lớn hơn hoặc bằng
<code><=</code>	So sánh nhỏ hơn hoặc bằng

Dưới đây là các ví dụ minh họa:

```

var a = 5, b = 10, c = "5";
var x = a;

console.log(a == c); // returns true
console.log(a === c); // returns false
console.log(a == x); // returns true
console.log(a != b); // returns true
console.log(a > b); // returns false
console.log(a < b); // returns true
console.log(a >= b); // returns false
console.log(a <= b); // returns true
console.log(a >= c); // returns true
console.log(a <= c); // returns true

```

Toán tử logic

Toán tử logic là toán tử được sử dụng để kết hợp hai hay nhiều điều kiện. Trong javascript có các toán tử logic sau:

Toán tử	Giải thích
&&	Toán tử && hay còn gọi là toán tử AND. Toán tử này chỉ trả kết quả true khi tất cả các toán hạng đều là true.
	Toán tử hay còn gọi là toán tử OR. Toán tử này chỉ trả về false khi tất cả các toán hạng đều false.
!	Toán tử này còn gọi là toán tử NOT. Nó đảo ngược giá trị của toán hạng.

Dưới đây là các ví dụ minh họa.

```

var a = 5, b = 10;

console.log((a != b) && (a < b)); // returns true
console.log((a > b) || (a == b)); // returns false
console.log((a < b) || (a == b)); // returns true
console.log(!(a < b)); // returns false
console.log(!(a > b)); // returns true

```

Toán tử gán

Hiểu đơn giản là các toán tử ngoài việc để gán giá trị cho biến còn khuyến mãi thêm tiện ích.

Toán tử	Giải thích
=	Gán giá trị của toán hạng phải cho toán hạng trái.
+=	Thực hiện tính tổng giá trị hai toán hạng trước, xong rồi mới gán giá trị tổng đó cho toán hạng trái.
-=	Thực hiện trừ (toán hạng trái - toán hạng phải), xong rồi mới gán giá trị hiệu đó cho toán hạng trái.
*=	Tương tự hai toán tử trên, thực hiện nhân trước rồi mới gán trị.
/=	Tương tự, thực hiện chia lấy phần nguyên trước rồi mới gán giá trị
%=	Tương tự, thực hiện chia lấy phần dư trước rồi mới gán giá trị.

Sau đây là ví dụ minh họa.

```
var x = 5, y = 10, z = 15;
```

```
console.log(x = y); //x would be 10
console.log(x += 1); //x would be 6
console.log(x -= 1); //x would be 4
console.log(x *= 5); //x would be 25
console.log(x /= 5); //x would be 1
console.log(x %= 2); //x would be 1
```

Toán tử điều kiện rút gọn

Thực ra mình cũng không biết gọi toán tử này là gì nữa, chỉ biết là cách dùng nó giống với câu lệnh *if - else* nên tạm gọi là toán tử điều kiện rút gọn. Lưu ý là loại này chỉ sử dụng cho 1 câu lệnh, thường dùng để gán giá trị trả về

Cú pháp:

<Điều kiện> ? <trả về giá trị 1 nếu điều kiện là true> : <trả giá trị 2 nếu điều kiện là false>;

ví dụ:

```
let a = 5;
```

```
let result = (a > 10) ? "a lớn hơn 10":"a nhỏ hơn 10"  
console.log(result);//a nhỏ hơn 10
```

Đoạn code trên tương đương:

```
let a = 5;  
let result;  
if(a > 0) {  
    result = "a lớn hơn 10";  
} else {  
    result = "a nhỏ hơn 10";  
}  
console.log(result);//a nhỏ hơn 10
```

Làm việc với điều kiện và cấu trúc có điều kiện

if...else Statements

Tương tự với các ngôn ngữ lập trình khác, Javascript cũng sử dụng khối lệnh *if...else* để điều khiển luồng của chương trình.

Với khối lệnh *if...else*, chương trình có thể rơi vào 3 trường hợp:

```
if(điều kiện 1){
```

```
// code ở đây được thực thi nếu điều kiện 1 đúng (true)
} else if ( điều kiện 2 ) {
    // code ở đây được thực thi nếu điều kiện 2 đúng (true)
    ...
} else {
    // Các trường hợp còn lại
}
```



Các câu lệnh trong khối điều kiện bắt buộc phải trả về một trị boolean (true/false). Nếu bạn thực hiện tính toán gì đó, thì cuối cùng phải tiến hành phép so sánh để trả về giá trị boolean, có như vậy khối lệnh if...else mới hoạt động

Ngoài ra, trong khối điều kiện, bạn có thể sử dụng các toán tử logic để kết hợp nhiều điều kiện.

Ví dụ:

```
if(điều kiện 1&& điều kiện 2){
    // code ở đây được thực thi nếu điều kiện 1 VÀ điều kiện 2 cùng đúng (true)
} else {
    // Các trường hợp còn lại
}
```

Switch Statements

Trong trường hợp khối lệnh *if...else* quá dài, bạn có thể nghĩ tới sử dụng câu lệnh *switch* để điều khiển luồng chương trình.

switch rất phù hợp khi bạn cần thực hiện một trong nhiều khối lệnh dựa trên điều kiện là giá trị trả về của một biểu thức được chỉ định.

Cú pháp:

```
switch(biểu thức){
    case 1:
        //code ở đây được thực thi
```



```

        break;
    case 2:
        //code ở đây được thực thi
        break;
    case n:
        //code ở đây được thực thi
        break;
    default:
        //Các trường hợp còn lại nếu tất cả trường hợp trên không được
        //thực thi
}

```

Điểm mạnh của *switch* so với câu lệnh *if...else* đó là giá trị trả về của biểu thức điều kiện không nhất thiết phải là boolean. Câu lệnh *switch* hỗ trợ giá trị điều kiện có kiểu dữ liệu: *boolean, number, string*



*Nhớ sử dụng câu lệnh **break** hoặc **return** để dừng chương trình và kết thúc câu lệnh **switch***

Sau đây là một ví dụ:

```

let dieukien = 'hello';
switch (dieukien) {
    case 'hello':
        console.log('OK, ' + dieukien);
        break;
    case 'xin chao':
        console.log('OK, ' + dieukien);
        break;
    default:
        console.log('OK, không chào ai hết');
        break;
}

```

>> Làm việc với function, định nghĩa và cách sử dụng

>> Tìm hiểu chi tiết các loại vòng lặp trong JS

>> Sự khác nhau giữa câu lệnh break và continue

PHẦN 4

CÚ PHÁP JAVASCRIPT NÂNG CAO

Sau khi hoàn thành xong phần 3 - cú pháp cơ bản của Javascript, bạn đã hiểu phần nào về ngôn ngữ lập trình đầy mê hoặc này rồi đúng không?😊

Nhưng Javascript không chỉ dừng lại có thế, nó còn rất nhiều điều hay ho phía trước nữa. Đảm bảo bạn sẽ cảm thấy thích thú hơn nữa cho mà xem.

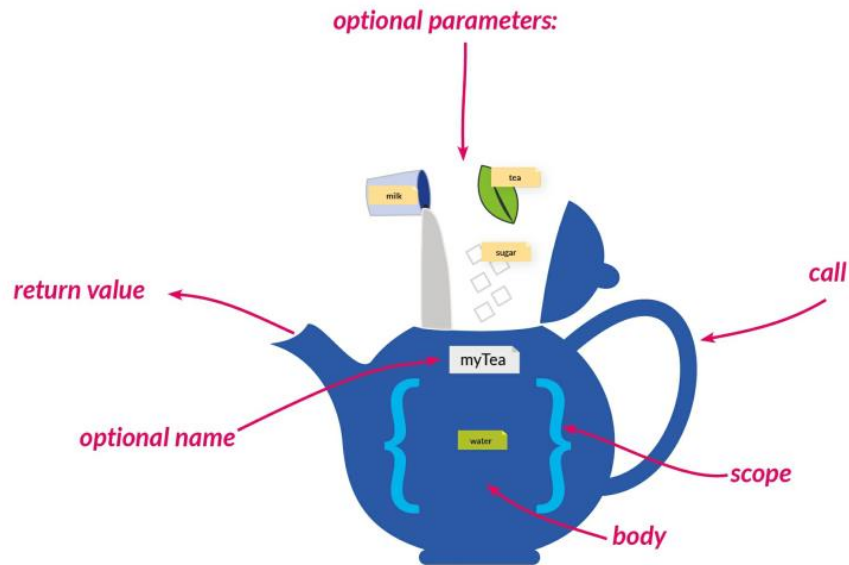
Trong phần cú pháp nâng cao, chúng ta sẽ tập trung tìm hiểu hai khái niệm rất phổ biến: *function* và vòng lặp (*loop*).

Function

Function hay còn gọi là hàm có thể được coi là một chương trình con, thực hiện một nhiệm vụ nào đó.

Cũng tương tự một chương trình hoàn chỉnh, một function gồm một tập hợp nhiều khối lệnh bên trong, gọi là "body" của function. Ngoài ra, function có thể nhận giá trị truyền vào qua các tham số và trả về kết quả sau khi kết thúc.

Có thể hình dung function như một cái máy chế biến, nhận nguyên liệu và trả về thành phẩm sau khi đã chế biến xong.



Hình 4.1: Minh họa cơ chế hoạt động của function

Function sinh ra là để giải quyết bài toán tái sử dụng mã nguồn. Người ta có thể gọi function ra để thực hiện những công việc giống nhau ở mọi nơi, thay vì phải viết lại toàn bộ code thực hiện nhiệm vụ đó, tránh trùng lặp code.

Javascript coi function là một first-class Object, vì chúng có các method và properties trong các programming object khác. Đây cũng là điểm khác biệt giữa function và object thông thường.



*Trong khoa học máy tính, một ngôn ngữ lập trình hỗ trợ các function có dạng first-class objects có nghĩa là ngôn ngữ đó hỗ trợ việc xây dựng mới các function trong quá trình thực thi chương trình, lưu trữ chúng trong cấu trúc dữ liệu, truyền chúng như là đối số cho các function khác, và trả về chúng như là các giá trị của function khác - **Theo wikipedia***

Có sự khác nhau giữa function và procedure là: function nên trả về một giá trị còn procedure thì không nhất thiết.

Để trả về một giá trị trong function, bạn sử dụng câu lệnh *return*. Trong trường hợp bạn không có câu lệnh *return*, giá trị mặc định trả về của function sẽ là "undefined".

Các tham số (đôi khi mọi người còn gọi là đối số) của một function có tác dụng là để truyền giá trị và sử dụng trong body của function. Do đó, nếu trong body của function mà có sửa đổi giá trị của đối số, thì phần sửa đổi đó chỉ có tác dụng trong phạm vi body của hàm đó mà thôi.

Cách định nghĩa một function

Chúng ta có nhiều cách để định nghĩa một function. Nhưng nhìn chung là phải sử dụng từ khoá *function* để định nghĩa một function. Từ phiên bản ES6, chúng ta có cách khác nữa để định nghĩa một function, đó là Arrow function. Phần 8 của cuốn sách, mình sẽ giới thiệu kỹ hơn cách này.

Cách 1: Định nghĩa kiểu truyền thống

```
function functionName([param1], [param2],...){  
  // body hàm  
  statement1;  
  statement2;  
  ...  
}
```

Cách 2: Định nghĩa function rồi gán cho biến

Vì function trong JS được coi là một first-class Object nên function cũng có thể được gán cho một biến và có thể truyền *function* vào một *function* khác với vai trò như một tham số (sau này bạn sẽ gặp nhiều cái này với tên gọi "thân thương" là callback).

```
var functionName = function([param1], [param2],...){  
  statement1;  
  statement2;  
  ...  
};
```

Dưới đây là một số ví dụ:

```
// Định nghĩa hàm  
function sayHello () {  
  console.log("Xin chào các bạn độc giả VNTALKING")  
}
```

```

var sayHi = function() {
  console.log("Xin chào các bạn đọc giả VNTALKING")
}

const square = function (number) {
  return number * number;
}

// gọi hàm
sayHello();
sayHi();
console.log(square(5));

```

Mình sẽ tóm tắt lại những đặc điểm chính của function như sau:

- Một function có thể có 1 tham số, nhiều tham số hoặc không có tham số.
- Các tham số truyền vào function có thể là biến hoặc một *function* khác.
- Body của một *function* bắt buộc phải đặt trong dấu ngoặc nhọn { }
- *Function* có thể có giá trị trả về hoặc không.

Tham số và phạm vi

Tham số của hàm giống như là một biến số với giá trị của nó được khởi tạo khi *function* được gọi.

Tất cả những tham số/biến số được khởi tạo trong hàm số đều có phạm vi cục bộ (local variable). Do vậy, nó chỉ được truy cập và tồn tại trong phạm vi của hàm số đó mà thôi.

Chúng ta cùng xem ví dụ sau nhé:

```

function increaseNumber(number){
  var a = 5;
  number += a;
  console.log('number bên trong function:' + number);
}

var number = 10;
increaseNumber(number);
// => number bên trong function: 15

console.log(number);

```

```
// => 10 (không thay đổi)

console.log(a);
// => Uncaught ReferenceError: a is not defined
```

Như ở ví dụ trên, bạn có thể thấy rằng: Tham số *number* ở trong *function* có giá trị là 15, nhưng khi ra ngoài *function* thì nó vẫn có giá trị là 10. Biến *a* được khai báo trong *function* sẽ chỉ truy cập ở trong *function* mà thôi, ở bên ngoài mà gọi biến đó là bị lỗi ngay.

Đặc điểm trên gọi là tính đóng của chương trình. Tức là mỗi *function* được coi là một chương trình hoàn chỉnh, độc lập với phần còn lại. Khi bạn khai báo biến bên trong *function*, bạn không cần phải quan tâm bên ngoài biến này đã khai báo rồi hay chưa.

Nested scope

Chúng ta khai báo một *function* bên trong một *function* thì được gọi là hàm lồng nhau. Vậy phạm vi của các biến khai báo trong các hàm lồng nhau (nested function) như thế nào?

Nếu bạn sử dụng hàm nested, thì những hàm này cũng có phạm vi sử dụng cục bộ trong hàm chứa nó. Bạn không thể gọi những hàm này ở global.

Chúng ta xem ví dụ sau:

```
function findMax(a, b, c, d) {
  var max = function (m, n) {
    if (m > n) return m;
    return n;
  };

  var t1 = max(a, b);
  var t2 = max(c, d);
  var t3 = max(t1, t2);

  console.log(t3);
}
```

```
findMax(3, 5, 4, 10); // => 10
```

```
var t = max(10, 11); // => Uncaught ReferenceError: max is not defined
```

Ở ví dụ trên, trong hàm *findMax* có chứa hàm *max*. Do đó, hàm *max* có thể được gọi bên trong hàm *findMax*. Nhưng khi mình gọi hàm *max* ở phạm vi bên ngoài hàm đó thì nó báo lỗi hàm *max* không được định nghĩa.

Pure function và non-pure function

Theo một góc độ nào đó thì có thể chia function ra làm hai loại:

- Pure function
- Non-pure function

Pure function là luôn trả về kết quả giống nhau khi giá trị tham số truyền vào giống nhau. Nó không bị phụ thuộc bởi bất cứ trạng thái, dữ liệu nào khác khi chương trình chạy mà chỉ phụ thuộc duy nhất vào tham số truyền vào.

Ngược lại với pure function là non-pure function, là hàm phụ thuộc vào biến toàn cục hay môi trường thực thi. Do đó, khi bạn gọi cùng một hàm số với cùng một đối số, nhưng kết quả thu được lại khác nhau.

Chúng ta cùng xem xét ví dụ về pure function sau:

```
function calculatePrice(productPrice) {  
  return productPrice * 0.1 + productPrice;  
}
```

Còn ví dụ sau là non-pure function:

```
var tax = 10;  
  
function calculateTax(productPrice) {  
  return productPrice * (tax / 100) + productPrice;  
}
```

Như ở ví dụ trên, giá trị trả về ngoài việc phụ thuộc vào giá trị tham số truyền vào, nó còn phụ thuộc vào biến toàn cục ở bên ngoài nữa (biến *tax*)

Ưu điểm của pure function

Có một ưu điểm quan trọng của pure function đó là tính dễ test và reafactoring (tái cấu trúc lại mã nguồn). Đặc tính luôn trả về một kết quả với cùng một đầu vào giúp cho việc test trở nên dễ dàng hơn bao giờ hết.

Trong khi đó, đặc tính không làm ảnh hưởng đến môi trường bên ngoài cũng giúp việc refactor code thuận lợi hơn, bởi vì bạn luôn chắc chắn rằng những thay đổi bên trong pure function sẽ chẳng phương hại gì đến những phần code khác trong dự án.

Ngoài ra, nếu sau này bạn làm việc với ReactJS, bạn sẽ bị bắt buộc phải sử dụng pure function trong redux (thư viện quản lý state phổ biến nhất). Phần này, mình sẽ nói chi tiết trong cuốn [ReactJS thật đơn giản](#).

Loop - vòng lặp

Vòng lặp (tiếng anh là loop) là một trong những khái niệm cơ bản không chỉ trong Javascript - mà các ngôn ngữ lập trình khác cũng thế.

Trước khi sử dụng vòng lặp, chúng ta cần phải biết vòng lặp là gì? Tác dụng của chúng là gì? Tại sao vòng lặp lại có ích cho bạn trong dự án.

Vòng lặp là gì?

Vòng lặp là đoạn mã cho phép bạn lặp lại một đoạn mã với số lần nhất định. Nếu bạn để lặp mà không xác định được số lần lặp, người ta gọi đó là lặp vô hạn.

Tại sao phải dùng vòng lặp

Để trả lời cho câu hỏi này, mình sẽ lấy ví dụ: Bạn cần in ra màn hình 10 lần dòng chữ:

"Xin chào các bạn đọc giả VNTALKING".

Chắc hẳn, trong đầu bạn sẽ nghĩ: đơn giản quá, viết 10 lần dòng code in ra đoạn chữ trên là được chứ gì! Có phải bạn định viết thế này không!?

```
document.write("Xin chào các bạn đọc giả VNTALKING!<br>");  
document.write("Xin chào các bạn đọc giả VNTALKING!<br>");  
document.write("Xin chào các bạn đọc giả VNTALKING!<br>");  
document.write("Xin chào các bạn đọc giả VNTALKING!<br>");
```



```
document.write("Xin chào các bạn đọc giả VNTALKING!<br>");
document.write("Xin chào các bạn đọc giả VNTALKING!<br>");
document.write("Xin chào các bạn đọc giả VNTALKING!<br>");
document.write("Xin chào các bạn đọc giả VNTALKING!<br>");
document.write("Xin chào các bạn đọc giả VNTALKING!<br>");
document.write("Xin chào các bạn đọc giả VNTALKING!<br>");
```

Nếu xét khía cạnh chỉ để chương trình chạy được thì có lẽ bạn đã làm đúng. Nhưng thực tế không ai làm như vậy, vì code sẽ bị trùng lặp rất nhiều - đây là điều tối kị trong lập trình. Mà đây còn chưa kể nếu phải in ra 100, 1000 lần thì sao? Bạn copy code mỗi tay luôn.

Thay vào đó, người ta sử dụng vòng lặp, ý tưởng như sau:

```
Do this block 10 times {
    document.write("Xin chào các bạn đọc giả VNTALKING!<br>");
}
```

Tất nhiên, bạn sẽ cần phải sử dụng câu lệnh lặp của Javascript để thay thế cho đoạn chữ "*Do this block 10 times*". Chúng ta sẽ tìm hiểu tiếp.

Để xem các vòng lặp thực sự hữu ích như thế nào trong ứng dụng, chúng ta cùng nhau xem các cấu trúc vòng lặp mà bạn có thể sử dụng trong Javascript.

Trong Javascript, chúng ta có 3 kiểu cấu trúc lặp:

- *for(...)*
- *while*
- *do...while*

Vòng lặp *for (...)*

Vòng lặp *for* trong Javascript được biết đến như một vòng lặp với số lần lặp biết trước. Ý tưởng của vòng lặp *for* là chúng ta sẽ có:

- Một biến đếm ở dạng số nguyên (thường được khởi tạo là 0)
- Một điều kiện để vòng lặp kết thúc (thường là biến đếm < số lần lặp)
- Sau mỗi lần lặp, biến đếm sẽ thay đổi giá trị.

Như bài toán ở trên, chúng ta có thể sử dụng vòng *for* như sau:

```

    }
    document.write("Xin chào các bạn đọc giả VNTALKING!<br>");
}

```

Biến đếm để xác định
số lần lặp

Trong thân vòng lặp, bạn hoàn toàn có thể sử dụng biến đếm (hoặc bất kỳ biến nào được khai báo trong vòng lặp) để làm điều đó gì đó.

Ví dụ:

```

document.write("=====START=====<br>");
for (var count = 1; count < 11; count++) {
    document.write(count + ". Xin chào các bạn đọc giả VNTALKING!<br>");
}
document.write("=====END=====");

```

Biến đếm count
được sử dụng để
tạo chỉ số dòng

Ngoài ra, bạn hoàn toàn có thể thêm các câu lệnh khác vào trong vòng lặp để xử lý yêu cầu nào đó.

Sau đây là ví dụ mình kết hợp câu lệnh *if/else* vào trong vòng lặp.

```

for (var count = 1; count < 11; count++) {
    if (count % 2) {
        document.write(
            count +
            '. Xin chào các bạn đọc giả VNTALKING!<br>Đây là số lần chào chẵn <br>'
        );
    } else {
        document.write(
            count +
            '. Xin chào các bạn đọc giả VNTALKING!<br>Đây là số lần chào lẻ <br>'
        );
    }
}

```

Vòng lặp `while()` {...}

Khác với vòng lặp `for`, vòng lặp `while` là vòng lặp với số lần không biết trước. Ý tưởng của vòng lặp `while` cũng đơn giản hơn, chúng ta chỉ cần chỉ cho vòng lặp `while` một điều kiện. Nếu điều kiện còn đúng, thì nó còn thực hiện lặp, nếu sai thì vòng lặp sẽ dừng.

Như vậy, với vòng lặp `while` có thể vòng lặp không được thực hiện nếu điều kiện kiểm tra sai ngay từ đầu.

```
var count = 1;  ← Khai báo một biến làm biến đếm

while (count < 6) {  ← Kiểm tra điều kiện trước khi vào lặp. Điều kiện
    //JavaScript code xử lý khi vào lặp    đúng thì vào vòng lặp. Không thì thoát luôn
    count++;  ← Biến đếm được cập nhập để tránh vòng lặp vô hạn
}
```

Trong vòng lặp `while`, bạn phải nhớ thay đổi giá trị của biến đếm, nếu không bạn sẽ dính vào một vòng lặp vô hạn. Trong phần lớn trường hợp, vòng lặp vô hạn sẽ làm treo ứng dụng.

Với đoạn code minh họa trên, nếu bỏ qua đoạn tăng biến đếm `count ++`, vòng lặp sẽ lặp vô thời hạn, chắc chắn trình duyệt sẽ bị treo ngay sau ít phút.

Vì vậy, điều quan trọng nhất mà bạn phải nhớ với vòng lặp `while` là cung cấp cho một biến đếm với giá trị ban đầu trước khi vào vòng lặp và nhớ phải thay đổi giá trị của biến trong chính vòng lặp.

Quay trở lại ví dụ in ra màn hình 10 dòng chữ, chúng ta hoàn toàn có thể thay thế vòng lặp `for` bằng vòng lặp `while` như sau:

```
var count = 1;
while (count < 11) {
    document.write(count + ". Xin chào các bạn đọc giả VNTALKING!<br>");
    count++;
}
```

Kết quả thu được hoàn toàn giống với vòng lặp `for`. Trong nhiều trường hợp, việc lựa chọn giữa vòng lặp `for` hay `while` chỉ dựa trên sở thích cá nhân của developer mà thôi.

Vòng lặp *do {...} while()*

Ý tưởng cơ bản của vòng lặp *do...while* là: cứ thực hiện lặp đi đã, kiểm tra điều kiện sau. Nếu điều kiện đúng, tiếp tục lặp, còn không thì thoát vòng lặp.

Như vậy, so với vòng lặp *while*, vòng lặp *do...while* có một đặc điểm khác biệt sau:

- Vòng lặp *while* có thể không lặp bất kỳ lần nào nếu điều kiện lặp sai ngay từ đầu. Nghĩa là nó sẽ kiểm tra điều kiện rồi mới lặp.
- Vòng lặp *do ... while* luôn lặp ít nhất một lần, nghĩa là nó sẽ chạy xong rồi mới kiểm tra điều kiện.

```
var count = 1;
do {
  //JavaScript code xử lý khi vào lặp
  count++;
} while (count < 6);
```

Câu lệnh *break* và *continue* trong vòng lặp

Câu lệnh *break*:

- Câu lệnh *break* thường được đặt vào bên trong các vòng lặp như: *for*, *while*, *do while* hoặc trong câu lệnh *switch...case*
- Nếu câu lệnh *break* không có label, nó sẽ kết thúc khối lệnh vòng lặp hoặc câu lệnh *switch* và chuyển sang khối lệnh tiếp theo.
- Nếu câu lệnh *break* có label, nó sẽ kết thúc câu lệnh có label đó.

Nói cách khác là "*lệnh break được dùng để thoát khỏi vòng lặp trước khi vòng lặp kết thúc*".

Chúng ta cùng nhau xem xét ví dụ sau.

```
for (let i = 1; i <= 10; i++) {
  if (i === 5) {
    break; // Thoát khỏi vòng lặp
  }
  console.log(i + ". Xin chào các bạn đọc giả VNTALKING!<br>");
}
```

Kết quả thu được như sau:

1. Xin chào các bạn đọc giả VNTALKING!
2. Xin chào các bạn đọc giả VNTALKING!
3. Xin chào các bạn đọc giả VNTALKING!
4. Xin chào các bạn đọc giả VNTALKING!

Như bạn thấy trên kết quả, ngay khi gặp câu lệnh, ngay lập tức thoát vòng lặp, chúng ta không còn thấy in ra màn hình các giá trị khi biến đếm *i* từ 6 đến 10 nữa.

Với trường hợp khối lệnh *switch...case*

```
var season = 'ha';
switch (season) {
  case 'xuan':
    document.write('Mùa Xuân');
    break;
  case 'ha':
    document.write('Mùa Hạ');
    break;
  case 'thu':
    document.write('Mùa Thu');
    break;
  case 'dong':
    document.write('Mùa Đông');
    break;
  default:
    break;
}
```

Kết quả hiển thị dòng chữ:

Mùa Hạ

Với trường hợp câu lệnh *break* có label.

```
document.write('1.Xin chào các bạn đọc giả VNTALKING!<br>');
document.write('2.Xin chào các bạn đọc giả VNTALKING!<br>');
document.write('3.Xin chào các bạn đọc giả VNTALKING!<br>');
number: {
  document.write('4.Xin chào các bạn đọc giả VNTALKING!<br>');
```

```

document.write('5.Xin chào các bạn đọc giả VNTALKING!<br>');
break number;
document.write('6.Xin chào các bạn đọc giả VNTALKING!<br>');
document.write('7.Xin chào các bạn đọc giả VNTALKING!<br>');
}
document.write('8.Xin chào các bạn đọc giả VNTALKING!<br>');
document.write('9.Xin chào các bạn đọc giả VNTALKING!<br>');

```

Kết quả:

```

1.Xin chào các bạn đọc giả VNTALKING!
2.Xin chào các bạn đọc giả VNTALKING!
3.Xin chào các bạn đọc giả VNTALKING!
4.Xin chào các bạn đọc giả VNTALKING!
5.Xin chào các bạn đọc giả VNTALKING!
8.Xin chào các bạn đọc giả VNTALKING!
9.Xin chào các bạn đọc giả VNTALKING!

```

Câu lệnh *continue*:

- Câu lệnh *continue* thường được đặt vào bên trong các vòng lặp như: *for*, *while*, *do...while*
- Khi lệnh *continue* được thực thi, những câu lệnh bên dưới của lần lặp hiện tại sẽ bị bỏ qua.

Cũng với đoạn code trên, chúng ta đổi lệnh *break* thành *continue* xem thế nào nhé.

```

for (let i = 1; i <= 10; i++) {
  if (i === 5) {
    continue; // Thoát khỏi lần lặp này, và chuyển sang bước lặp tiếp theo.
  }
  document.write(i + '. Xin chào các bạn đọc giả VNTALKING!<br>');
}

```

Kết quả:

```

1. Xin chào các bạn đọc giả VNTALKING!
2. Xin chào các bạn đọc giả VNTALKING!
3. Xin chào các bạn đọc giả VNTALKING!
4. Xin chào các bạn đọc giả VNTALKING!
6. Xin chào các bạn đọc giả VNTALKING!

```

7. Xin chào các bạn đọc giả VNTALKING!
8. Xin chào các bạn đọc giả VNTALKING!
9. Xin chào các bạn đọc giả VNTALKING!
10. Xin chào các bạn đọc giả VNTALKING!

Tóm lại:

Cả hai câu lệnh này đều có ý nghĩa bỏ qua - không thực hiện đoạn code bên dưới nó trong vòng lặp. Điểm khác biệt duy nhất là hành vi chương trình sau đó. Cụ thể:

- *Break*: Thoát luôn toàn bộ vòng lặp.
- *Continue*: Chỉ bỏ qua bước lặp hiện tại và tiếp tục lặp bước tiếp theo.

>> Khái niệm, các thao tác làm việc với Object

>> Cách khai báo và làm việc với mảng dữ liệu. Tìm hiểu những hàm built-in hữu ích cho Array.

PHẦN 5

DỮ LIỆU CÓ CẤU TRÚC

Giống với các ngôn ngữ lập trình bậc cao khác, Javascript cũng hỗ trợ các loại dữ liệu có cấu trúc, là những dữ liệu kết hợp các kiểu dữ liệu khác.

Trong phần này, chúng ta sẽ cùng tìm hiểu hai loại dữ liệu có cấu trúc phổ biến nhất:

- Object
- Array

Có một quan điểm cho rằng: chỉ cần bạn hiểu rõ và làm việc thành thục với Object là bạn đã làm chủ được Javascript.

Mình cũng hoàn toàn đồng ý với quan điểm đó. Bởi vì, khi bạn làm việc với Javascript, bạn sẽ làm việc rất nhiều với Object. Sau này, khi bạn học xong cuốn sách này, đi làm dự án, hãy thử ngẫm lại và cảm nhận xem quan điểm trên có đúng không nhé!?

Object

Object hay còn gọi là đối tượng, là một kiểu dữ liệu rất đặc biệt, nói không quá khi nó là kiểu dữ liệu quan trọng nhất, tạo thành nền tảng của Javascript hiện đại.

Kiểu dữ liệu Object là sự kết hợp của nhiều kiểu dữ liệu, cả nguyên thủy (primitive data-types) như: *Number*, *String*, *Boolean*, *null*, *undefined*... và cả Object khác trong đó (Object lồng nhau).

Thật khó để đưa ra một định nghĩa chính xác về kiểu dữ liệu Object. Mình xin định nghĩa một cách dễ hiểu nhất như sau: Kiểu dữ liệu Object trong Javascript là một tập hợp các kiểu dữ liệu liên quan không theo thứ tự dưới dạng cặp “*key: value*”. Key ở đây có thể là biến hoặc hàm và được gọi là thuộc tính hoặc method, tùy ngữ cảnh của Object đó.

Một Object được định nghĩa bằng dấu ngoặc nhọn {...} và chứa bên trong danh sách các thuộc tính.

Thuộc tính là một cặp “*key:value*”, trong đó key phải là một *string* hoặc *number* (hay còn gọi là tên thuộc tính), còn value thì có thể là bất kì thứ gì (có thể là giá trị, hàm hoặc một object khác...)

Đọc định nghĩa xong lại cảm thấy khó hiểu hơn phải không? Haha, trước mình cũng thế 😊

Để dễ hiểu hơn định nghĩa này, chúng ta cùng xem xét ví dụ sau nhé.

```
let website = {  
  name : "VNTALKING.COM",  
  location : "Hà Nội - Việt Nam",  
  established : "2018"  
}
```

Trong ví dụ trên, “name”, “location”, “established” đều là “key”.

Còn “VNTALKING.COM”, “Hà Nội – Việt Nam”, “2018” là các value, giá trị tương ứng.

Tóm lại:

- Mỗi cặp “*key:value*” này được gọi là thuộc tính (properties) của Object.
- Với thuộc tính nào mà value lại là một hàm thì lúc này người ta gọi đây là method của Object đó.

Nếu Object trên mà có method thì sẽ như thế nào? Cùng xem ví dụ sau đây:

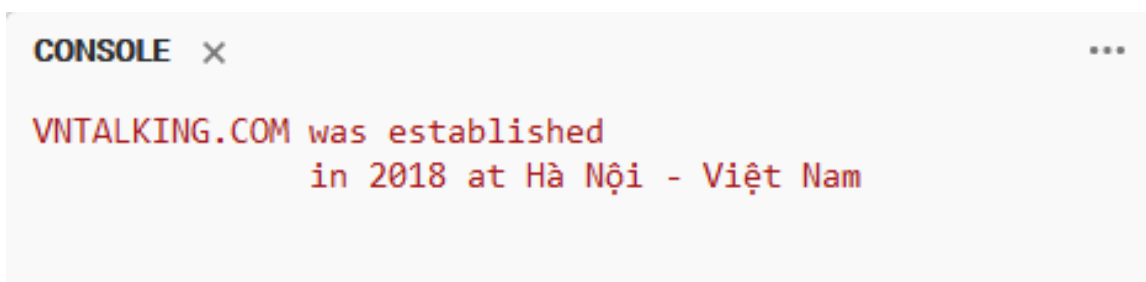
```
let website = {  
  name : "VNTALKING.COM",  
  location : "Hà Nội - Việt Nam",  
  established : "2018",  
  displayInfo : function(){  
    console.log(` ${website.name} was established  
      in ${website.established} at ${website.location} `);  
  }  
}
```

← Đây gọi là thuộc tính

→ Đây gọi là method

`website.displayInfo();`

Kết quả khi chạy đoạn code trên:



Chúng ta có thể hiểu nôm na như sau: Object để mô tả đặc điểm – hành vi của một đối tượng, sự vật nào đó. Ví dụ: chúng ta dùng Object để mô tả một ai đó, trong đó thuộc tính là đặc điểm về chiều cao, cân nặng, màu da... còn method là hành vi của người đó như cách người đi lại, cách ăn uống, cách người đó thể hiện bản thân...

Đến đây, bạn đã hiểu rõ định nghĩa về Object rồi đúng không!?

Thuộc tính riêng và thuộc tính kế thừa

Trong Object, có hai kiểu thuộc tính:

- Thuộc tính riêng (own property) là thuộc tính được định nghĩa tại bản thân của đối tượng (object).

- Thuộc tính kế thừa (inherited property) là những thuộc tính được kế thừa từ đối tượng prototype của object đó. Đối tượng prototype là đối tượng đặc biệt, được thêm vào tự động khi một đối tượng được tạo. Khái niệm prototype là khái niệm đặc biệt, có ý nghĩa quan trọng khi bạn định lập trình hướng đối tượng bằng Javascript (Phần sau cuốn sách mình sẽ trình bày kỹ về hướng đối tượng).

Để xác định một thuộc tính có phải là thuộc tính riêng hay không, chúng ta có thể sử dụng hàm *hasOwnProperty()*

```
// hasOwnProperty code in js
const object = new Object();
object.property1 = 42;
console.log("property1 là own property: " +
object.hasOwnProperty("property1")); //kết quả là true
```

Kết quả khi chạy đoạn code trên:



Để kiểm tra một thuộc tính có thuộc object hay không (kể cả thuộc tính riêng và thuộc tính kế thừa), ta dùng toán tử *in*:

```
var person = {name: 'Duong Anh Son'};
//kiểm tra thuộc tính
console.log('name' in person);    //kết quả in ra là: true
console.log('age' in person);    //kết quả in ra là: false
```

Cách tạo Object

Trong phần kiểu dữ liệu, mình cũng đã trình bày sơ lược về Object, và cách tạo một Object. Đến phần này, mình sẽ trình bày đầy đủ hơn, chi tiết về tất cả các cách để bạn khởi tạo một Object.

Nhìn chung, chúng ta có 4 cách để khởi tạo một Object:

- Dùng Object literals
- Dùng Object constructor
- Sử dụng Object Constructor functions
- Sử dụng prototype pattern.

Chúng ta sẽ cùng lần lượt tìm hiểu từng cách.

Cách 1: Dùng Object literals

Mình cũng không biết dịch từ literal chính xác là gì, nhưng bạn cứ hiểu đại khái như sau: Trong OOP, Object literals được biểu diễn bằng dấu phẩy ngăn cách giữa các cặp ***name-value*** nằm trong ngoặc nhọn { }

Về cơ bản, cú pháp như sau:

```
var obj = {  
    member1 : value1,  
    member2 : value2,  
};
```

Trong đó, member có thể là bất cứ thứ gì, như *strings, numbers, functions, arrays...* thậm chí là một object khác.

//Ví dụ 1: đối tượng có 1 thuộc tính

```
var myBook = {name: 'Học lập trình Javascript thật đơn giản'};
```

//Ví dụ 2: đối tượng có 1 thuộc tính và 1 phương thức (method) và một object.

```
var myCar = {  
    brand: 'Toyota',  
    run: function(){  
        console.log('xe ô tô đang chạy bon bon trên đường làng');  
    },  
    wheel: {  
        color: "black",  
        size: 20  
    }  
};
```

Cách 2: Sử dụng Object Contructor

Một cách khác để tạo đối tượng, đó là sử dụng hàm khởi tạo (constructor). Hàm khởi tạo này là một hàm để tạo ra các object mới, đi kèm với từ khoá *new*. Ngoài ra, với hàm khởi tạo, bạn có thể khởi tạo luôn những giá trị mặc định của đối tượng được tạo.

```
const website = new Object();
website.name = 'VNTALKING.COM';
website.location = 'Hà Nội';
website.established = 2018;

website.displayInfo = function(){
    console.log(`${website.name} được thành lập từ ${website.established} tại
    ${website.location}`);
}

website.displayInfo();
```

Hai cách tạo object được đề cập ở trên không phù hợp lắm với các chương trình yêu cầu tạo nhiều đối tượng cùng loại. Vì bạn sẽ phải lặp lại code rất là nhiều.

Để giải quyết vấn đề này, chúng ta có thể sử dụng hai phương pháp tạo đối tượng khác, chính là cách 3 và 4 mà mình sẽ trình bày dưới đây.

Cách 3: Sử dụng Object Contructor functions

Tương tự như cách 2, nhưng cải tiến hơn một chút và cũng giống với hầu hết các ngôn ngữ lập trình OOP. Tức là, với cách này, chúng ta sẽ định nghĩa một danh sách các thuộc tính và phương thức chung cho tất cả các đối tượng.

Chúng ta cùng xem đoạn code là hiểu rõ ngay.

```
function Vehicle (name, maker) {
    this.name = name;
    this.maker = maker;
}
```

```
let car1 = new Vehicle('Fiesta', 'Ford');
let car2 = new Vehicle('Santa Fe', 'Hyundai')

console.log(car1.name);    // In ra màn hình console: Fiesta
console.log(car2.name);    // In ra màn hình console: Santa Fe
```

Trong trường hợp bạn không muốn viết kiểu function thì có thể viết kiểu class (kiểu viết theo class có vẻ giống với các ngôn ngữ thuần OOP như Java hơn, từ khóa này chỉ xuất hiện từ bản ES6 trở lên)

```
class people {
  constructor() {
    this.name = 'Dương Anh Sơn';
  }
}

let person = new people();

// In ra màn hình console là: : Dương Anh Sơn
console.log(person.name);
```



Trong một class, chỉ được phép có duy nhất một hàm có tên là `constructor()`, nếu không sẽ có lỗi compile.

Cách 4: Sử dụng prototype

Trong Javascript, bản thân function cũng được coi là 1 object. Đã là object thì sẽ có thuộc tính. Mặc định, khi function được tạo, nó sẽ được thêm vào một thuộc tính đặc biệt, đó là đối tượng prototype. Mặc định đối tượng prototype là đối tượng rỗng.

Chúng ta có thể thêm method hoặc thuộc tính vào đối tượng prototype này. Mô tả chi tiết về những đặc điểm của Prototype nằm ngoài phạm vi của phần này, chúng ta sẽ tìm hiểu kỹ hơn về prototype trong phần sau (lập trình hướng đối tượng OOP với javascript)

```
// Tạo một hàm constructor function rỗng
function Person(name){
}
```

```
// Thêm thuộc tính name, age, address cho prototype property của hàm Person constructor
Person.prototype.name = "Dương Anh Sơn" ;
Person.prototype.age = 28;
Person.prototype.address = "Hà Nội";
Person.prototype.displayName = function(){
    console.log(this.name);
}

// Khởi tạo object sử dụng hàm khởi tạo của Person
var person = new Person();

// Truy cập tới thuộc tính name sử dụng đối tượng person
person.displayName() // Output "Dương Anh Sơn"
```

Truy xuất thông tin Object

Để truy xuất vào member của một Object, cụ thể là các thuộc tính và method của Object, chúng ta có hai cách:

- **Sử dụng dấu chấm (.)**

Đây có lẽ là cách phổ biến và được nhiều người sử dụng nhất. Cách này giống với hầu hết các ngôn ngữ lập trình bậc cao khác như Java, C#, Python...

Cú pháp: *(objectName.memberName)*

Ví dụ đoạn code minh họa nhé.

```
let website = {
    domain : "VNTALKING.COM",
    location : "Hà Nội",
    established : 2018,
    displayinfo : function() {
        console.log("Website học lập trình hàng đầu VN");
    }
}

console.log(website.domain);
```

```
website.displayinfo();
```

- **Sử dụng dấu ngoặc vuông []**

Nếu bạn đã từng lập trình bằng ngôn ngữ khác như Java, C#... bạn sẽ thấy cách viết này khá giống với cách bạn truy xuất vào phần tử của mảng.

Cú pháp: *objectName["memberName"]*

Vẫn đối tượng trên, chúng ta thử cách truy cập kiểu này xem sao.

```
let website = {  
  domain : "VNTALKING.COM",  
  location : "Hà Nội",  
  established : 2018,  
  displayinfo : function() {  
    console.log("Website học lập trình hàng đầu VN");  
  }  
}  
  
console.log(website["domain"]);  
console.log(website["location"]);
```

Tuy nhiên, với cách viết này, bạn không thể truy xuất tới method của Object được.

Ví dụ, bạn viết như này sẽ bị lỗi:

```
website[displayinfo()] //error: ReferenceError: displayinfo is not defined
```

Truy xuất hàng loạt keys của Object

Trong một số trường hợp, bạn cần phải truy xuất tất cả thuộc tính của Object, nhưng vấn đề là Object đó có rất nhiều thuộc tính thì bạn phải làm sao? Bạn có đủ kiên nhẫn để gõ từng key khi muốn truy xuất thông tin không? Thậm chí bạn còn không biết tên key của thuộc tính đó là gì nữa.

Trường hợp này phổ biến nhất khi bạn parser JSON, bạn sẽ gặp phải vấn đề này.

Để minh họa cho vấn đề trên, bạn thử truy cập API này sẽ rõ:

<https://api.exchangeratesapi.io/latest>

```
{  
  "CAD": 1.5453,  
  "HKD": 9.3911,  
  "ISK": 157,  
  "PHP": 58.27,  
  "DKK": 7.4377,  
  "HUF": 360.9,  
  "CZK": 26.025,  
  "AUD": 1.5744,  
  "RON": 4.8753,  
  "SEK": 10.1139,  
  "IDR": 17123.2,  
  "INR": 88.438  
}
```

Để giải quyết bài toán này, chúng ta sử dụng vòng lặp *for...in*

```
let rates= {  
  "CAD": 1.5453,  
  "HKD": 9.3911,  
  "ISK": 157,  
  "PHP": 58.27,  
  "DKK": 7.4377,  
  "HUF": 360.9,  
  "CZK": 26.025,  
  "AUD": 1.5744,  
  "RON": 4.8753,  
  "SEK": 10.1139,  
  "IDR": 17123.2,  
  "INR": 88.438  
}  
for (let key in rates) {  
  console.log(key + ": " + rates[key]);  
}
```

Kết quả như dưới đây:

```
CONSOLE x ...
CAD: 1.5453
HKD: 9.3911
ISK: 157
PHP: 58.27
DKK: 7.4377
HUF: 360.9
CZK: 26.025
AUD: 1.5744
RON: 4.8753
SEK: 10.1139
IDR: 17123.2
INR: 88.438
```



Với cách sử dụng vòng lặp `for..in`, bạn chỉ có thể truy xuất vào các thuộc tính được phép. Ví dụ, với các thuộc tính được kế thừa từ `Object.prototype` thì bạn sẽ không sử dụng cách này để truy xuất thông tin được.

Xóa thuộc tính của Object

Để xóa một thuộc tính của Object, chúng ta có thể sử dụng toán tử `delete`. Cách làm như sau:

```
let persion = {
  name: "Duong Anh Son"
}
//Output: Duong Anh Son
console.log(persion.name);
delete persion.name
```

```
//Sau khi xóa, bạn sẽ không thể truy xuất thông tin của thuộc tính đó nữa. Đơn
giản là vì nó không còn tồn tại.
//Output : undefined
console.log(persion.name);
```

Array

Trong Javascript, Array – hay còn gọi là mảng, là kiểu dữ liệu mà giá trị của nó chứa nhiều giá trị khác. Mỗi giá trị của mảng được gọi là một phần tử.

Không giống với nhiều ngôn ngữ lập trình khác khi mà array tham chiếu đến nhiều biến, trong javascript, array là một biến duy nhất lưu trữ nhiều phần tử.

Cách khai báo Array

Về cơ bản, bạn có 2 cách để khai báo một Array.

Cách 1: Dùng Array literals [...]:

Được biểu diễn bằng dấu ngoặc vuông [...], bạn có thể khai báo luôn là mảng rỗng hoặc có giá trị, các giá trị được ngăn cách nhau bằng dấu phẩy.

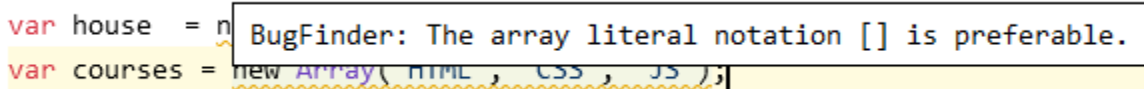
```
let house = []; // mảng rỗng
var webLanguages = ["HTML", "CSS", "JS"]; // mảng có 3 phần tử
```

Cách 2: Dùng hàm khởi tạo constructor

Với cách này, chúng ta sử dụng toán tử *new* để khởi tạo.

```
var house = new Array();
var courses = new Array("HTML", "CSS", "JS");
```

Trong hai cách khai báo trên, tùy từng trường hợp mà cách 1 được thích hơn và cũng được các chuyên gia khuyến khích sử dụng.



```
var house = []
var courses = new Array('HTML', 'CSS', 'JS');
```

Hình 5.1: Cảnh báo của trình BugFinder

Chúng ta sẽ cùng nhau phân tích những điểm khác biệt giữa hai cách khai báo này để từ đó tự rút ra lý do tại sao nhé.

Cùng nhau xem xét hai ví dụ sau:

```
//Sử dụng cách 1:
// Khởi tạo giá trị trong khi khai báo
var house = ["1BHK", "2BHK", "3BHK", "4BHK"];
//=====

//Sử dụng cách 2:
// Khởi tạo giá trị trong khi khai báo
// Tạo một mảng có 5 phần tử: 10, 20, 30, 40, 50
var house = new Array(10, 20, 30, 40, 50);

//Khởi tạo một mảng có 5 phần tử là undefined
var house1 = new Array(5);

//Khởi tạo một mảng có 1 phần tử là !BHK
var home = new Array("!BHK");
```

Như ở đoạn code trên, khi bạn sử dụng cách 2, mảng *house* có 5 phần tử (10, 20, 30, 40, 50) trong khi mảng *house1* lại chứa 5 phần tử *undefined* thay vì có một phần tử có giá trị là 5.

Do đó, khi làm việc với mảng mà có phần tử là các số thì cách 2 không được ưu thích cho lắm, nhưng nó lại khá ổn khi các phần tử là String hoặc Boolean. Như ví dụ mình khởi tạo mảng *home* chỉ có một phần tử là ("!BHK")

Truy cập vào phần tử mảng

Mỗi phần tử của mảng sẽ được đánh số tự động, bắt đầu từ số 0. Do đó, để truy cập (lấy ra hoặc cập nhật) vào một phần tử của mảng, bạn làm như sau:

```
//Lấy giá trị
tên_mảng[chỉ_số_của_phần_tử]
//Cập nhật
tên_mảng[chỉ_số_của_phần_tử] = <giá_trị_mới>
```

Chúng ta cùng xem ví dụ dưới đây:

```
var house = ["1BHK", "2BHK", "3BHK", "4BHK"];
```

```
console.log(house[0]); //kết quả: 1BHK
//cập nhật
house[0] = "ABC";
console.log(house[0]); //kết quả: ABC
```

Để truy cập vào tất cả các phần tử của Array, chúng ta đơn giản là dùng một vòng lặp.

Trong trường hợp bạn truy cập vào phần tử không tồn tại thì nó trả về giá trị *undefined* thôi.

```
var house = ["1BHK", "2BHK", "3BHK", "4BHK"];
console.log(house[10]); //kết quả: undefined
```

Mảng chứa những phần tử có kiểu khác nhau

Trong Javascript, có một điểm khá thú vị, đó là Array có thể chứa nhiều phần tử có những kiểu dữ liệu khác nhau. Đây có lẽ là điểm mà những ngôn ngữ như Java có mơ cũng không làm được 😊

```
//lưu trữ các phần tử có kiểu number, boolean, strings trong một Array
var house = ["1BHK", 25000, "2BHK", 50000, "Rent", true];
```

Các thao tác làm việc với mảng

Sau khi chúng ta đã hiểu và biết cách khai báo một Array. Với Javascript, bạn có rất nhiều công cụ hỗ trợ để bạn làm việc trên Array thuận tiện và hiệu quả.

Dưới đây là những hàm hỗ trợ bạn làm việc với Array phổ biến nhất, mặc dù không phải là tất cả nhưng đủ để bạn tha hồ vùng vẫy.

1. Thêm phần tử vào mảng

Phương thức *push()* được dùng để thêm một phần tử mới vào vị trí cuối mảng.

Cú pháp:

```
Array.push(item1, item2 ...);
```



Phần tử để thêm vào mảng. Những phần tử này sẽ được thêm vào cuối mảng.

Ví dụ minh họa:

```
// Khai báo và khởi tạo giá trị các phần tử
var number_arr = [ 10, 20, 30, 40, 50 ];
var string_arr = [ "HTML", "CSS", "JAVASCRIPT", "NODEJS" ];
console.log(number_arr);
console.log(string_arr);
//=====//
console.log("=====")
//1. Thêm một phần tử 60 vào mảng số
number_arr.push(60); //kết quả: number_arr chứa [10, 20, 30, 40, 50, 60]

//2. Thêm cùng lúc nhiều phần tử
number_arr.push(70, 80, 90); //kết quả: [10, 20, 30, 40, 50, 60, 70, 80, 90]

//3. Thêm phần tử cho mảng string
string_arr.push("JAVA", "PHP"); // [ "HTML", "CSS", "JAVASCRIPT", "NODEJS", "J
AVA", "PHP" ]

//Hiển thị kết quả
console.log(number_arr);
console.log(string_arr);
```

Như bạn đã thấy kết quả trong ví dụ trên, các phần tử thêm mới đều được đẩy xuống vị trí cuối của mảng. Nhưng trong một số trường hợp, bạn muốn thêm các phần tử vào đầu của mảng thì sao? Giải pháp đơn giản là bạn sử dụng hàm *unshift()*.

```
Array.unshift(item1, item2 ...);
```



Phần tử để thêm vào mảng. Những phần tử này sẽ được thêm vào đầu mảng.

Vẫn đoạn code trên, chúng ta thay thế hàm *push()* bằng hàm *unshift()* và kiểm tra kết quả:

```
// Khai báo và khởi tạo giá trị các phần tử
```

```

var number_arr = [ 10, 20, 30, 40, 50 ];
var string_arr = [ "HTML", "CSS", "JAVASCRIPT", "NODEJS" ];
console.log(number_arr);
console.log(string_arr);
//=====//
console.log("=====")
//1. Thêm một phần tử 60 vào mảng số
number_arr.unshift(60); //kết quả: number_arr chứa [60, 10, 20, 30, 40, 50]

//2. Thêm cùng lúc nhiều phần tử
number_arr.unshift(70, 80, 90); //kết
quả: [70, 80, 90, 60 ,10 ,20 ,30 ,40 ,50 ]

//3. Thêm phần tử cho mảng string
string_arr.unshift("JAVA", "PHP"); // [ "HTML", "CSS", "JAVASCRIPT", "NODEJS"
,"JAVA", "PHP" ]

//Hiển thị kết quả
console.log(number_arr);
console.log(string_arr);

```

2. Xóa phần tử khỏi mảng

Để xóa một phần tử ở cuối mảng, chúng ta sử dụng hàm *pop()*. Kết quả của hàm này là giá trị của phần tử vừa xóa xong.

Cú pháp:

`Array.pop()` ← Không cần tham số. Vì mặc định là xóa phần tử cuối cùng.

Ví dụ minh họa:

```

// Khai báo và khởi tạo giá trị
var number_arr = [ 20, 30, 40, 50 ];
var string_arr = [ "Dương Anh Sơn", "Trần Huyền Trang", "Nguyễn Văn A" ];
console.log(number_arr);
console.log(string_arr);
//=====
console.log("=====");
// xóa phần tử ở phía cuối cùng của mảng

```

```
number_arr.pop(); //kết quả: [ 20, 30, 40 ]
```

```
// xóa phần tử ở phía cuối cùng của mảng
```

```
string_arr.pop(); //kết quả: [ "Dương Anh Sơn", "Trần Huyền Trang" ]
```

```
//In kết quả
```

```
console.log(number_arr);
```

```
console.log(string_arr);
```

Đây là bạn muốn xóa phần tử cuối cùng, thế trong trường hợp muốn cho “bay màu” phần tử đầu tiên thì sao? Bạn dùng hàm *shift()* là được.

Chỉnh sửa lại đoạn code trên và xem kết quả như nào nhé.

```
// Khai báo và khởi tạo giá trị
```

```
var number_arr = [ 20, 30, 40, 50 ];
```

```
var string_arr = [ "Dương Anh Sơn", "Trần Huyền Trang", "Nguyễn Văn A" ];
```

```
console.log(number_arr);
```

```
console.log(string_arr);
```

```
//=====
```

```
console.log("=====");
```

```
// xóa phần tử ở phía cuối cùng của mảng
```

```
number_arr.shift(); //kết quả: [ 30, 40, 50 ]
```

```
// xóa phần tử ở phía cuối cùng của mảng
```

```
string_arr.shift(); //kết quả: [ "Trần Huyền Trang", "Nguyễn Văn A" ]
```

```
//In kết quả
```

```
console.log(number_arr);
```

```
console.log(string_arr);
```

Đến đây, hẳn bạn sẽ bật ra ngay thắc mắc là: *Từ lúc này đến giờ, tôi chỉ thấy thêm/xóa phần tử ở đầu hoặc cuối của mảng, thế tôi muốn thêm hoặc xóa một phần tử ở bất kỳ vị trí nào trong mảng thì làm thế nào? Chẳng nhẽ bó tay à!*

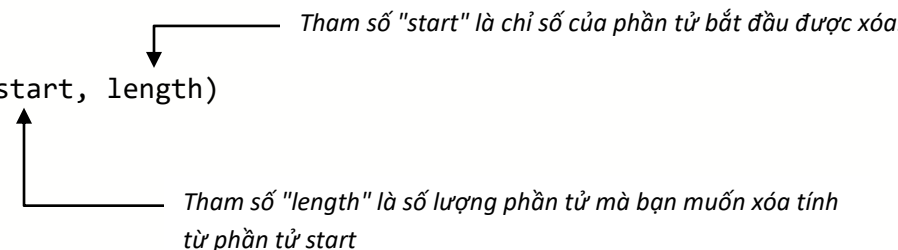
Tất nhiên là không bó tay được rồi. Câu trả lời là sử dụng hàm *splice()*.

Cách sử dụng hàm này, mình chia là 2 trường hợp:

- **Xóa “một hoặc nhiều phần tử” ở vị trí bất kỳ trong mảng**

Cú pháp:

`array.splice(start, length)`



Tham số "start" là chỉ số của phần tử bắt đầu được xóa.

Tham số "length" là số lượng phần tử mà bạn muốn xóa tính từ phần tử start

Ví dụ:

```
let mobiles = ["HTC", "Nokia", "Apple", "LG", "Honda", "SYM", "Suzuki"];
mobiles.splice(1,4);
console.log(mobiles);
//=> Kết quả: ["HTC", "SYM", "Suzuki"];
```

- **Thêm “một hoặc nhiều phần tử” vào vị trí bất kỳ trong mảng**

Cú pháp:

`array.splice(start, số_phần_tử_sẽ_xóa_khi_thêm, phần_tử_mới_1, phần_tử_mới_2,);`

Ví dụ:

```
let mobiles = ["HTC", "Nokia", "SamSung", "LG", "Apple"];
mobiles.splice(2, 0, " a ", " b ", " c ");
console.log(mobiles)
//=> kết quả: ["HTC" ,"Nokia" ," a " ," b " ," c ","SamSung" ,"LG" ,"Apple"]
```

Bạn có thắc mắc vai trò của tham số `số_phần_tử_sẽ_xóa_khi_thêm` mà bạn truyền vào là gì không? Trong ví dụ trên thì mình truyền số 0, nếu truyền số khác thì sao?

Ý nghĩa của tham số này là **trước khi thêm phần tử mới** vào mảng, hàm `splice()` sẽ xóa số phần tử bằng với giá trị bạn truyền vào tính từ phần tử có chỉ số `start`. Để hiểu rõ hơn, chúng ta xem ví dụ dưới đây:

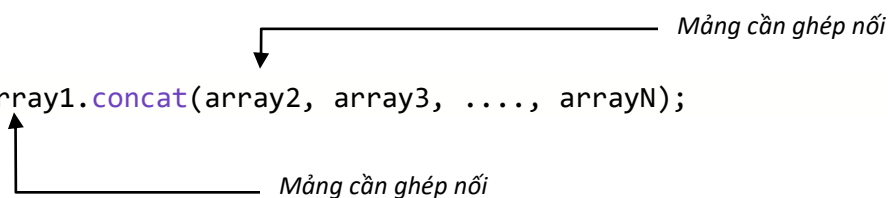
```
let mobiles = ["HTC", "Nokia", "SamSung", "LG", "Apple"];
mobiles.splice(2, 3, " a ", " b ", " c ");
console.log(mobiles)
//==> kết quả: ["HTC" ,"Nokia" ," a " ," b " ," c "]
```

3. Ghép nhiều mảng thành một mảng

Trong nhiều trường hợp thực tế, bạn cần phải hợp nhất hai mảng thành một mảng duy nhất. Có nhiều cách để làm được điều này, thủ công nhất là bạn dùng vòng loop. Nhưng với JS, bạn có cách làm đơn giản hơn rất nhiều, đó là sử dụng hàm *concat()*

Cú pháp:

```
var tên_mảng_mới = array1.concat(array2, array3, ...., arrayN);
```



Ví dụ minh họa:

```
const array1 = [1, 2, 3, 4, 5]
const array2 = [10, 20, 30, 40, 50]
let arrayConbined = array1.concat(array2)
console.log(arrayConbined)
//-----> Kết quả : [1, 2, 3, 4, 5, 10, 20, 30, 40, 50]
```

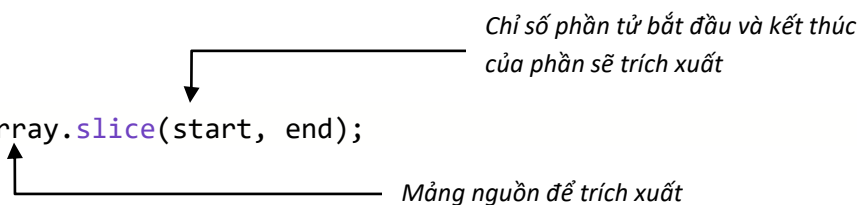
4. Trích xuất một phần của mảng

Nếu muốn tạo một mảng bằng cách trích xuất một phần từ một mảng đã có, bạn có thể sử dụng hàm *slice()*.

Cú pháp:

```
var tên_mảng_mới = array.slice(start, end);
```

Ví dụ:



```
let mobiles = ["HTC", "Nokia", "Apple", "LG", "Honda", "SYM", "Suzuki"];
var myMobile = mobiles.slice(1,5);
console.log(myMobile);
```

```
//==> Kết quả: ["Nokia", "Apple", "LG", "Honda"];
```

Trong trường hợp bạn không truyền giá trị *end*, hàm này sẽ trích xuất mảng từ phần tử *start* cho tới hết mảng.

```
let mobiles = ["HTC", "Nokia", "Apple", "LG", "Honda", "SYM", "Suzuki"];  
var myMobile = mobiles.slice(1);  
console.log(myMobile);  
//==> Kết quả: ["Nokia", "Apple", "LG", "Honda", "SYM", "Suzuki"];
```



Hai tham số "*start*" và "*end*" đều chấp nhận giá trị là số âm. Nếu bạn truyền số âm, thì chỉ số của các phần tử mảng sẽ được xác định theo chiều ngược lại (phần tử đầu tiên là -1).

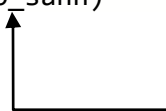
5. Sắp xếp mảng

Sắp xếp lại các phần tử của mảng là một thao tác rất phổ biến, đặc biệt trong các trường hợp như tìm kiếm, tìm phần tử lớn nhất, nhỏ nhất.v.v...

Bạn có thể sử dụng hàm *sort()* để sắp xếp các phần tử trong một mảng, các phần tử có thể được sắp xếp theo bảng chữ cái hoặc theo chữ số theo thứ tự tăng dần hoặc giảm dần.

Cú pháp:

```
array.sort(hàm_so_sánh)
```



Là tham số không bắt buộc. Nó là một hàm định nghĩa thứ tự sắp xếp, hàm này nên được trả về giá trị âm, 0 hoặc dương tùy thuộc vào tham số đầu vào của nó. Khi phương thức *sort()* tiến hành so sánh 2 giá trị, nó sẽ gửi các giá trị đó đến hàm này, và sắp xếp chúng dựa vào kết quả trả về của hàm này.

Ví dụ:

```
const myArray = [5, 4, 3, 2, 1]
```

```
// Sắp xếp từ Nhỏ nhất đến Lớn nhất
```

```
myArray.sort(function(a, b) {
  return (a-b);
})

console.log(myArray);
//-----> Kết quả : [1, 2, 3, 4, 5]

// Sắp xếp từ Lớn nhất đến Nhỏ nhất
myArray.sort(function(a, b) {
  return (b-a);
})
console.log(myArray);
//-----> Kết quả : [5, 4, 3, 2, 1]
```



Khác với các hàm thao tác với mảng mà mình giới thiệu ở trên, hàm `sort()` làm thay đổi giá trị mảng ban đầu thay vì trả về một mảng mới.

6. Tìm kiếm phần tử trong mảng

Tương tự như hàm sắp xếp, tìm kiếm cũng là một thao tác rất phổ biến và cách viết code cũng tương tự.

Để tìm kiếm, bạn sử dụng hàm `find()`. Hàm này sẽ trả về giá trị của phần tử đầu tiên trong mảng thỏa mãn được điều kiện kiểm tra (được truyền vào như một hàm). Nếu không tìm thấy phần tử nào thỏa mãn, nó trả về giá trị *undefined*.

Cú pháp:

```
array.find(hàm_kiểm_tra);
```

Ví dụ:

```
const persons = [
  { id: 1, name: "Trang" },
  { id: 2, name: "Doanh" },
  { id: 3, name: "Sơn" },
]
```

```

let person = persons.find(function(person) {
    return person.id === 3;
});
console.log(person);
//-----> Kết quả : {id: 3, name: "Sơn"}

let p = persons.find(function(person){
    return person.id === 7;
});
console.log(p);
//-----> Kết quả : undefined

```

7. Các hàm làm việc với mảng hữu ích khác

Trên đây là những thao tác, hàm hỗ trợ làm việc với Array phổ biến và hay dùng nhất. Trong Javascript, bạn còn được cung cấp rất nhiều công cụ hay ho khác nữa, chắc chắn sẽ khiến bạn “mất chữ A, miệng chữ O” về độ tiện lợi của nó, cũng như giải quyết được rất nhiều bài toán trong dự án của bạn.

Dưới đây là một trong số đó:

- **Hàm *map()*:** Thường được dùng để biến đổi các phần tử trong một mảng thành một kiểu khác. Tất nhiên, mảng mới trả về phải luôn cùng số lượng phần tử với mảng ban đầu.

Ví dụ:

```

const myArray = [5, 4, 3, 2, 1]
// Tạo mảng mới mà mỗi phần tử có giá trị gấp 10 lần phần tử trong mảng cũ
const newArray = myArray.map(function(x){
    return x * 10;
})

console.log(newArray);
//=>Kết quả: [50 ,40 ,30 ,20 ,10]

```

- **Hàm *filter()***: Đúng như tên gọi, hàm này có chức năng là lọc những phần tử thỏa mãn điều kiện. Để thực hiện lọc, hàm này sẽ duyệt từng phần tử và kiểm tra điều kiện. Nếu thỏa mãn điều kiện, nó sẽ nhét phần tử này vào mảng mới.

Ví dụ:

```
//Lọc những độ tuổi được phép xem phim 18+
const ages = [5, 12, 13, 24, 33, 16, 40];

let canWatch = ages.filter(function(age){
  return age >= 18;
});

console.log(canWatch);
//=>Kết quả: [24 ,33 ,40]
```

- **Hàm *every()***: Hàm này có chức năng giúp bạn kiểm tra xem trong mảng mà **tất cả** các phần tử có thỏa mãn điều kiện nào đó hay không. Kết quả trả về là *true* hoặc *false*. Chỉ cần một phần tử trong mảng không thỏa mãn điều kiện là hàm *every()* sẽ trả về kết quả *false*.

Ví dụ:

```
//Kiểm tra xem mảng này có phải có tất cả phần tử là số chẵn hay không?
const arr = [56, 92, 18, 88, 12];

var result = arr.every(function(element) {
  return (element % 2 === 0);
});

console.log(result);
//=>Kết quả: true
```

- **Hàm *some()***: hàm này tương tự với hàm *every()*, chỉ khác một chút là nó trả về *true* khi **chỉ cần ít nhất một phần tử thỏa mãn điều kiện**. Hai hàm *some()* và *every()* làm mình liên tưởng tới hai toán tử so sánh AND và OR. Hàm *every()* tương đương với phép AND, còn *some()* tương đương với OR.

Ví dụ:

```
//Kiểm tra xem mảng này có phải chứa ít nhất một số chẵn hay không?  
const arr = [55, 92, 18, 88, 12];  
  
var result = arr.some(function(element) {  
    return element % 2 === 0;  
});  
  
console.log(result);  
//==>Kết quả: true
```

PHẦN 6

HIGHER-ORDER FUNCTION

Khi bạn đọc đến dòng này, chứng tỏ bạn cũng khá kiên trì đọc sách đấy ^_^.
Với 5 phần đầu của cuốn sách, bạn đã bước đầu hiểu và làm chủ những kỹ thuật cơ bản của ngôn ngữ lập trình Javascript, là nền tảng để tiếp tục nghiên cứu những phần nâng cao về Javascript, những điều mà sẽ khiến bạn thêm yêu mến ngôn ngữ lập trình này.

Phần này, chúng ta sẽ cùng nhau tìm hiểu một khái niệm vô cùng quan trọng và ảnh hưởng rất nhiều tới tư duy lập trình khi làm việc với Javascript. Đó là khái niệm Higher-Order Functions.

Với việc hỗ trợ Higher-Order Functions khiến cho Javascript phù hợp với triết lý lập trình hướng hàm (Functional Programming). Tại sao lại như vậy? Mình sẽ giải thích ngay ở dưới đây.

Khái niệm Higher-Order Functions

Để làm rõ khái niệm Higher-Order Functions này, chúng ta lần lượt tìm hiểu thêm các khái niệm:

- Functional Programming
- First-Class Functions

Functional Programming

Trong giới lập trình hiện nay, có 2 trường phái lập trình phổ biến nhất là: lập trình hướng đối tượng (OOP) và lập trình hướng hàm (FP).

Trong đó, lập trình hướng hàm là phương pháp lập trình lấy hàm làm trọng tâm, là đơn vị nhỏ nhất và cơ bản nhất của chương trình.

Với phương pháp lập trình này, chúng ta chỉ quan tâm tới hành vi, hành động để giải quyết vấn đề thay vì tập trung vào đối tượng, dữ liệu, trạng thái.

Trong phương pháp lập trình FP, có một điểm nhỏ nhưng lại rất quan trọng đó là **bạn có thể truyền các hàm dưới dạng tham số của hàm khác**.

JavaScript, Haskell, Clojure, Scala, và Erlang là một trong số các ngôn ngữ lập trình hỗ trợ phương pháp lập trình Functional Programming.

First-Class Functions

Ở phần 4, khi tìm hiểu về function, mình cũng đã đề cập đến khái niệm này. Mình chỉ nhắc lại thôi, trong Javascript, một hàm cũng được coi là 1 object. Tức là hàm cũng mang các đặc điểm giống với một kiểu dữ liệu thông thường như *Number*, *String*, *Object*.v.v... Cụ thể như chúng ta có thể gán hàm vào một biến, tạo một hàm bên trong một hàm, và đặc biệt có thể return về một hàm như cách mà chúng ta return một giá trị.

Kết hợp hai khái niệm trên, chúng ta tiến tới khái niệm Higher-Order Functions.

Higher-Order Functions là gì?

Chúng ta hiểu đơn giản như sau, Higher-Order Functions là một hàm:

- Chấp nhận truyền một hàm qua tham số
- Hoặc hàm đó có thể return về một hàm.

Ví dụ:

```
// phần định nghĩa hàm higher-order
function higherOrder (number, sayLog) {
  let result = number * 10;
  sayLog(result)
}

function sayLog(x) {
  console.log(x);
}

// Cách sử dụng
higherOrder(10, sayLog)
//==>kết quả: 100
```

Hàm này gọi là higher-order function

Hàm này gọi là callback function

Hoặc chúng ta có thể truyền thẳng hàm ẩn danh (anonymous function) làm callback cũng được. Ví dụ như dưới đây:

```
// phần định nghĩa hàm higher-order
function higherOrder (number, callback) {
  let result = number * 10;
  callback(result)
}

//Cách sử dụng
higherOrder(10, function(result) {
  console.log(result)
});
//==>kết quả: 100
```

Tóm lại, chúng ta phân biệt hai khái niệm:

- **Hàm Higher-Order** là hàm nhận một hàm khác là tham số hoặc return về một hàm. Một hàm mà có một trong hai yếu tố trên đều được gọi là higher-order function.
- **Hàm callback**: là hàm truyền vào hàm higher-order.

Higher-order functions được sử dụng rộng rãi trong JavaScript. Lúc đầu, khi mới tiếp cận với Javascript, bản thân mình cũng gặp khá nhiều khó khăn với khái niệm này. Nhưng cứ thực hành nhiều, dần dần bạn sẽ tự vỡ ra.

Tất nhiên, higher-order function phải có nhiều ưu điểm thì nó mới được sử dụng rộng rãi đến vậy. Mình cô đọng lại thì ưu điểm lớn nhất đó là tối ưu code thỏa mãn tiêu chí: **Single Responsibility**. Tức là mỗi hàm chỉ nên làm duy nhất một nhiệm vụ.

Chúng ta có thể tạo các hàm chỉ xử lý một logic. Sau đó, kết hợp chúng để tạo ra các hàm phức tạp hơn, giải quyết các logic lớn hơn. Kỹ thuật này giúp giảm lỗi, dễ unit test, và làm cho code dễ đọc và dễ hiểu hơn.

Ví dụ minh họa Higher-Order function

Sau khi đã hiểu khái niệm, chúng ta sẽ cùng nhau xem xét các ví dụ về higher-order function để hiểu rõ hơn nhé,

Trường hợp 1: Truyền một function như là tham số vào một hàm

Bài toán: Giả sử bạn cần phải xây dựng một hàm tổng quát về tính toán hai số, phép tính có thể là cộng, trừ, nhân, chia. Lúc bạn định nghĩa hàm này, bạn cũng không biết hàm này sử dụng phép tính nào, chỉ khi nào sử dụng, lập trình viên người ta truyền hàm tính thì mới biết được.

Hàm này có 3 tham số:

- Number1
- Number2
- Phép tính

Bạn hiểu yêu cầu của đầu bài rồi đúng không?

Dưới đây là code cho ví dụ trên:

```
// Định nghĩa hàm tính tổng quát. operation có thể là cộng, trừ, nhân chia
function doOperation(number1, number2, operation) {
    return operation(number1, number2)
}
```

```

// Phép tính tổng
function sum(a, b) {
    return a + b;
}

//Phép tính trừ
function minus (a, b) {
    return a - b;
}

//Phép tính nhân
function multiplier (a, b) {
    return a * b;
}
// phép tính chia
function divide(a, b) {
    if(b !== 0) {
        return a / b;
    }
}

// Lúc lập trình viên gọi
let result = doOperation(3, 5, sum)
console.log(result);
//==> Kết quả: 8

```

Bình thường, nếu được yêu cầu viết hàm tính toán hai số thì có lẽ bạn chỉ dừng lại việc viết hàm *sum()*, *minus()*... nhưng trong trường hợp này nó lại chỉ là tham số cho hàm *doOperation()*.

Nếu nhìn ở góc độ khái quát, hàm *doOperation()* có vẻ cao hơn mấy hàm *sum()*, *minus()*... một bậc. Có lẽ chính vì thế mà người ta gọi hàm *doOperation()* là “higher-order – xếp hạng cao hơn” chẳng?! 😊

Trường hợp 2: Return một function

Tiếp theo, chúng ta thử tạo một hàm higher-order trong đó return lại về một hàm khác. Cụ thể, hàm có tên là *multiplyBy()*, hàm này có tham số nhận vào một số và trả về một hàm sẽ nhân số nhận được với một hằng số nào đó.

Giải thích dài dòng vậy thôi, chứ lúc viết code ra thì rất dễ hiểu. Nhìn bên dưới này nhé.

```
function multiplyBy(multiplier) {  
    return function result(num) {  
        return num * multiplier  
    }  
}
```

```
//Lúc sử dụng  
multiplyByThree = multiplyBy(3);  
let result = multiplyByThree(4);  
console.log(result)  
//==> kết quả: 12
```

Với hai cách sử dụng higher-order function, cá nhân mình chỉ hay dùng theo cách như mô tả trong trường hợp 1 mà thôi, tức là truyền hàm vào tham số một hàm chứ rất ít khi dùng kiểu return về một hàm cả.

Một số hàm higher-order built-in (được xây dựng sẵn trong Javascript)

Các hàm higher-order được sử dụng rất nhiều trong javascript, bản thân các hàm tiện ích được tích hợp sẵn trong javascript cũng được viết theo kiểu higher-order.

Mình lấy ví dụ các hàm làm việc với mảng như:

- `Array.prototype.map`
- `Array.prototype.filter`
- `Array.prototype.reduce`

Đều là những ví dụ điển hình nhất về sử dụng hàm higher-order. Các bạn có thể xem lại tác dụng của các hàm này trong phần trước. [Phần 5 - Array](#)

Tại phần này, mình sẽ chỉ giải thích thêm sự tiện lợi của hàm higher-order qua cách viết code trong trường hợp sử dụng hàm higher-order và không sử dụng nó.

Mình sẽ sử dụng hàm *Array.prototype.map* làm minh họa với yêu cầu đầu bài như sau: Chúng ta có một mảng các số, yêu cầu tạo một mảng mới mà mỗi phần tử có giá trị gấp đôi mỗi giá trị của mảng ban đầu.

- **Không Higher-Order function**

```
const array = [1, 2, 3];
const newArray = [];
let length = array.length
let i = 0
while (i < length){
  newArray.push(array[i] * 2);
  i++;
}

console.log(newArray);
//Kết quả: [2, 4, 6]
```

- **Có Higher-Order function**

Các nhà phát triển Javascript đã sử dụng cách này để tạo ra hàm *map()* mà bạn vẫn hay sử dụng đó. Các viết cũng dễ đọc hơn rất nhiều phải không.

```
const array = [1, 2, 3];
const newArray = array.map(function(item) {
  return item * 2;
});
console.log(newArray);
//Kết quả: [2, 4, 6]
```

Qua ví dụ trên, bạn có thể tự cảm nhận được lợi ích của higher-order function, có lẽ mình sẽ không phải đưa ra ý kiến ở đây nữa.

Trong phần này, khi tìm hiểu về higher-order function, thật không phải khi không nói kỹ hơn về callback, một đặc sản của Javascript.

Tìm hiểu kỹ hơn về Callback

Về định nghĩa callback, bạn có thể xem lại trong phần [định nghĩa của higher-order](#). Tóm lại, callback là hàm được truyền vào một hàm khác qua tham số của hàm đó. Có lẽ do mọi người hay sử dụng callback để thực hiện việc gì đó sau khi một hàm khác đã thực hiện xong (nên tên gọi mới gọi là callback – “gọi lại”).

Vậy người ta hay sử dụng callback để làm gì?

Chắc hẳn bạn cũng đã từng nghe nói, trình duyệt hay như Node.JS sử dụng javascript như một ngôn ngữ lập trình hướng sự kiện. Tại đây có khái niệm xử lý bất đồng bộ.

Mình sẽ giải thích đơn giản: Một **mã chạy đồng bộ** tức là chương trình sẽ chạy lần lượt các dòng lệnh từ trên xuống dưới như cách bạn viết code.

Ví dụ như dưới đây:

```
console.log("thực hiện dòng lệnh 1");
console.log("thực hiện dòng lệnh 2");
console.log("thực hiện dòng lệnh 3");
```

```
//Kết quả:
//thực hiện dòng lệnh 1
//thực hiện dòng lệnh 2
//thực hiện dòng lệnh 3
```

Ngược lại, một chương trình chạy bất đồng bộ tức là các lệnh sẽ được đưa hết vào trong bộ nhớ, lệnh nào chạy xong trước sẽ trả kết quả trước, do đó thứ tự kết quả trả về có thể không giống với thứ tự như lúc bạn viết code.

Ví dụ:

```
console.log("thực hiện dòng lệnh 1");
setTimeout(function(){
  console.log("thực hiện dòng lệnh 2");
```

```
}, 1000)

console.log("thực hiện dòng lệnh 3");

//Kết quả:
//thực hiện dòng lệnh 1
//thực hiện dòng lệnh 3
//thực hiện dòng lệnh 2
```

Như bạn thấy ở trên, kết quả trả về đã có sự xáo trộn. Trong khuôn khổ cuốn sách, mình sẽ không trình bày quá sâu về cơ chế lập trình bất đồng bộ hay Event-Loop.

Khi code chạy bất đồng bộ có ưu điểm là tận dụng được sự nhàn rỗi của CPU, nhờ đó mà tăng hiệu năng của ứng dụng. Đặc biệt là khi cần thao tác với tài nguyên từ bên ngoài như đọc/ghi database, lấy dữ liệu qua network.v.v...

Tuy nhiên, trong một số trường hợp, bạn cần phải để code chạy đồng bộ, bạn cần phải chờ kết quả trả về từ một hàm (ví dụ: chờ hàm lấy dữ liệu từ database trả kết quả) để tiếp tục xử lý logic tiếp theo. Đây chính là lúc bạn cần tới callback.

Callback được sử dụng để biến code chạy không đồng bộ thành đồng bộ. Callback là một cách để đảm bảo đoạn code nào đó không thực thi cho đến khi code khác thực hiện xong.

Bạn có thắc mắc liệu có phải đồng nhất khái niệm callback với higher-order function không?

Thực ra, callback chỉ là một tác dụng, một ứng dụng cụ thể của higher-order function mà thôi.

Ok, giờ thì đã rõ về callback rồi đúng không? Chúng ta sẽ thử viết code tạo một callback nhé.

Bài toán: Bạn viết một chương trình làm bài tập về nhà, khi nào làm xong thì thông báo ra màn hình.


```

function doHomework(subject, callback) {
  console.log(`Bắt đầu làm bài tập ${subject}.`)
  console.log('Đang làm...')
  //Dành 1s để làm bài tập
  setTimeout(function(){
    //đã làm xong
    callback();
  }, 1000)
}

doHomework('Toán cao cấp', function() {
  console.log('Làm bài tập xong!');
});
//Kết quả=>
//Bắt đầu làm bài tập Toán cao cấp.
//Đang làm...
//Làm bài tập xong!

```

Thêm một ví dụ nữa nhé: Giả sử chúng ta cần tải một ảnh từ một URL, sau khi tải ảnh về xong thì tiến hành xử lý ảnh đó. Theo logic thông thường, bạn sẽ cần tiến hành gọi hàm tải ảnh trước, tải xong thì gọi hàm xử lý ảnh.

Có phải bạn sẽ định viết như bên dưới đây đúng không!?

```

// Hàm tải ảnh từ một url
function download(url) {
  // Mình dùng hàm timeout để giả lập việc tải ảnh cần thời gian
  setTimeout(() => {
    // đoạn code tiến hành download tại đây.
    console.log(`Downloading ${url} ...`);
    // Download xong thì trả kết quả
    return picture;
  }, 3000);
}
// Hàm xử lý ảnh
function process(picture) {
  // Giả lập việc xử lý ảnh

```

```

    console.log(`Processing ${picture}`);
}

// Gọi hàm
let url = 'https://vntalking.com/wp-content/uploads/2018/05/tech-logo-white.png';
//Step 1: tải ảnh
let picture = download(url);
//Step 2: Xử lý ảnh tải về
process(picture);

```

Bạn thử đoán xem kết quả sẽ như nào? Có phải chương trình sẽ xử lý lần lượt logic như bạn mong muốn không? Tải ảnh xong rồi mới xử lý ảnh được tải đó!?

Bùm! Đây là kết quả mà bạn thu được:

```

// Kết quả ==>
// Processing undefined
// Downloading https://vntalking.com/wp-content/uploads/2018/05/tech-logo-white.png ...

```

Như bạn thấy đấy, mặc dù hàm *process(image)* được gọi sau nhưng lại cho ra kết quả trước, thế là bạn nhận được giá trị *undefined*. Nguyên nhân là *download(url)* là hàm bất đồng bộ, chương trình sẽ không đợi nó trả kết quả rồi mới gọi tới hàm tiếp theo. Đây chính là lý do chúng ta cần tới callback.

Bạn có thể tham khảo cách chúng ta sử dụng callback trong trường hợp này.

```

function download(url, callback) {
    setTimeout(() => {
        // đoạn code tiến hành download tại đây.
        console.log(`Downloading ${url} ...`);
        // Download xong thì gọi hàm tiến hành xử lý ảnh
        callback(url);
    }, 3000);
}

function process(picture) {
    console.log(`Processing ${picture}`);
}

```

```
// Gọi hàm
let url = 'https://vntalking.com/wp-content/uploads/2018/05/tech-logo-white.png';
// tải ảnh và xử lý ảnh
download(url, process);
// Kết quả ==>
// Downloading https://vntalking.com/wp-content/uploads/2018/05/tech-logo-white.png ...
// Processing https://vntalking.com/wp-content/uploads/2018/05/tech-logo-white.png
```

OK rồi nhỉ! Bạn hiểu cách thức hoạt động của callback rồi đúng không! Khi làm việc với Javascript, chúng ta sử dụng callback rất nhiều. Tuy nhiên, nếu bạn sử dụng callback một cách bừa bãi sẽ làm code trở lên khó đọc, đặc biệt là khi callback lồng nhau.

Khi callback lồng nhau quá sâu, nhiều tầng, người ta gọi đây là callback hell. Như đoạn code minh họa dưới đây chẳng hạn.

```
asyncFunction(function(){
  asyncFunction(function(){
    asyncFunction(function(){
      asyncFunction(function(){
        asyncFunction(function(){
          //....
        });
      });
    });
  });
});
```

Để khắc phục lỗi callback hell, bạn có thể sử dụng **Promise** hoặc **Async/Await**. Chúng ta sẽ tiếp tục tìm hiểu hai công cụ quan trọng này trong phần tiếp theo.

Còn bây giờ, hãy ngồi nghỉ ngơi và thưởng thức tách cafe trước khi tiếp tục nhé.

Promise

Trước hết, chúng ta cùng xem định nghĩa promise là gì, hiểu được cái này thì bạn mới có thể hiểu được bản chất của promise. Theo định nghĩa của nhà phát hành:

Promise là một đối tượng được sử dụng cho tính toán bất đồng bộ. Một promise đại diện cho một tiến trình hay một tác vụ chưa thể hoàn thành ngay được. Trong tương lai, promise sẽ trả về giá trị hoặc là đã được giải quyết (resolve) hoặc là không (reject).

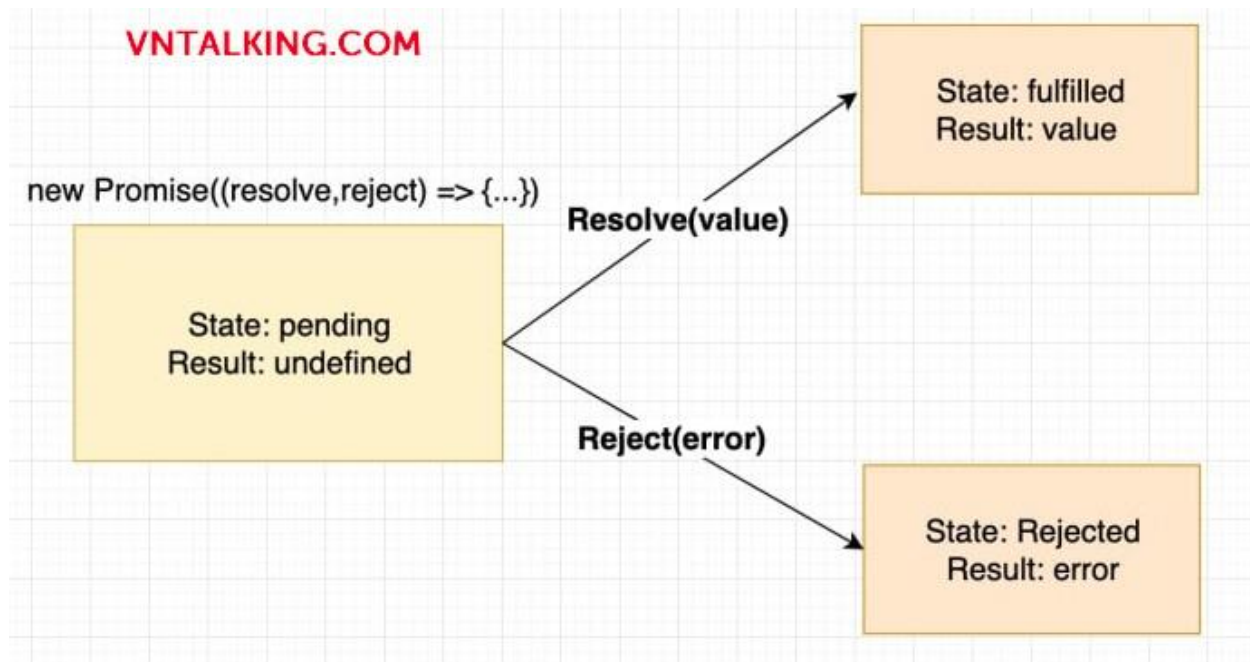
Đọc định nghĩa có vẻ khó hiểu vậy thôi. Chứ thực ra, có thể promise đúng như tên gọi của nó: **Lời hứa**. Tức là một lời hứa có thể thực hiện (resolve) hoặc thất hứa (reject).

Để mình lấy một ví dụ về Promise Javascript: Hãy quay trở lại tuổi thơ dữ dội của mình. Một hôm đẹp trời, bố bảo: “Nếu con ngoan, bố hứa sẽ mua cho một con robot siêu to khổng lồ vào tuần tới”.

Nếu nhìn theo góc độ kỹ thuật, đây chính là một promise. Và promise này có thể có 3 trạng thái:

- **Pending:** Hiện lời hứa vẫn đang là hứa suông. Không biết có thành hiện thực hay không?
- **Fulfilled:** Bố bạn cảm thấy vui và mua cho bạn một con robot thật.
- **Rejected:** Rất tiếc, bố trượt con đề và lời hứa kia đã thành lời nói gió bay.

Tóm lại, một promise sẽ có 3 trạng thái: Pending, Fulfilled, và Rejected.



Hình 6.1: Các trạng thái của Promise

Cú pháp sử dụng promise cơ bản như sau:

```
var promise = new Promise (function a(resolve, reject) {  
  if(// task complete) {  
    resolve(value);  
  } else {  
    reject(new Error());  
  }  
});
```

Cú pháp là vậy, chúng ta cùng nhau viết đoạn code cho bài toán ví dụ về lời hứa của bố ở trên.

```
// Trạng thái tâm lý của bố: true -> bố đang vui  
const isDadHappy = true;  
  
// Định nghĩa một promise  
let buyNewToy = new Promise(function(resolve, reject) {  
  if (isDadHappy) {  
    let toy = {
```

```

        name: "Ô tô Camry 2.0 mới nhất."
    }
    resolve(toy); // Fulfilled
} else {
    reject('Bố trượt lô, nên khỏi mua đồ chơi luôn');
}
});

// Cách sử dụng promise trên
let askDad = function (isDadHappy) {
    buyNewToy
        .then(function (fulfilled) {
            console.log('Mình nhận được một món đồ chơi từ bố, là: ' +
fulfilled.name);
        })
        .catch(function (reject) {
            console.log('Chả có đồ chơi nào cả.');
```

Cũng dễ sử dụng nhỉ!

Trong các dự án Javascript, người ta cũng khuyến khích bạn sử dụng promise thay thế cho callback, đơn giản vì promise có nhiều ưu điểm làm cho code của bạn sạch đẹp hơn, xử lý error tốt hơn so với callback truyền thống. Ngoài ra, trong phần trước, mình có đề cập tới việc promise giúp chúng ta giải quyết được “vấn nạn” callback hell. Chúng ta sẽ cùng nhau xem xét ví dụ minh họa bên dưới nhé.

Bài toán: Giả sử, mỗi sáng thức dậy, bạn cần phải làm một công việc thiết yếu như: đánh răng, ăn sáng,... sau đó thì tới trường. Các công việc này đều phải làm tuần tự, việc này làm xong thì mới tiến hành làm việc khác được. Bạn phải thức dậy đã rồi mới đánh răng được, chứ không ai đang ngủ mà vẫn đánh răng cả.

Giờ tiến hành viết code nhé.

Đầu tiên, chúng ta sẽ viết code sử dụng callback:

```
//Định nghĩa các hành động
function wakeUp(callback) {
  console.log("Thức dậy");
  setTimeout(function () {
    //Làm xong, tiến hành việc nào đó khác
    callback();
  }, 1000);
}

function brushTeeth(callback) {
  console.log("Đánh răng");
  setTimeout(function () {
    //Làm xong, tiến hành việc nào đó khác
    callback();
  }, 2000);
}

function eatBreakfast(callback) {
  console.log("Ăn sáng");
  setTimeout(function () {
    //Làm xong, tiến hành việc nào đó khác
    callback();
  }, 3000);
}

function gotoSchool(callback) {
  console.log("Đi học");
  setTimeout(function () {
    //Làm xong, tiến hành việc nào đó khác
    callback();
  }, 5000);
}

// Trình tự mỗi sáng bạn sẽ làm lần lượt:
```

```
// Thức dậy => đánh răng => ăn sáng => đi học
wakeUp(function () {
  brushTeeth(function () {
    eatBreakfast(function () {
      gotoSchool(function () {
        // Đang ở trên trường vừa học vừa trên gái.
      });
    });
  });
});
//Kết quả =>
// Thức dậy
// Đánh răng
// Ăn sáng
// Đi học
```

Code giờ đã “phẳng”, không còn lồng nhau nữa. Đây chính là biểu hiện của callback hell

Bạn thấy đoạn code trên có “xấu” không? Nhìn rất khó hiểu vì nó lồng nhau quá nhiều.

Nếu dùng promise để xử lý callback hell thì sẽ thế nào!? Chúng ta xem cách giải quyết như dưới đây nhé.

```
//Định nghĩa các hành động
function wakeUp() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      console.log("Thức dậy ");
      //Làm xong, tiến hành việc nào đó khác
      resolve();
    }, 1000);
  });
}

function brushTeeth() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      console.log("Đánh răng");
      //Làm xong, tiến hành việc nào đó khác
      resolve();
    }, 1000);
  });
}
```



```

    }, 2000);
  });
}
function eatBreakfast() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      console.log("Ăn sáng");
      //Làm xong, tiến hành việc nào đó khác
      resolve();
    }, 1000);
  });
}
function gotoSchool() {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      console.log("Đi học");
      //Làm xong, tiến hành việc nào đó khác
      resolve();
    }, 3000);
  });
}
// Trình tự mỗi sáng bạn sẽ làm lần lượt:
// Thức dậy => đánh răng => ăn sáng => đi học
wakeUp()
  .then(function (fulfilled) {
    return brushTeeth();
  })
  .then(function (fulfilled) {
    return eatBreakfast();
  })
  .then(function (fulfilled) {
    return gotoSchool();
  });
//Kết quả =>
// Thức dậy
// Đánh răng
// Ăn sáng
// Đi học

```

Code giờ đã “phẳng”, không còn lồng nhau nữa.

Về cơ bản, dùng promise đã khá thú vị rồi, những dòng code đã trở nên đẹp hơn rất nhiều. Nhưng Javascript chưa dừng lại ở đó, từ phiên bản ES8 trở lên, Javascript còn giới thiệu thêm tính năng mới cũng hay ho không kém, đó là *Async/await*.

Nhờ *Async/Await* mà những dòng code bất đồng bộ nhìn như đồng bộ, rất phù hợp với các bạn đã từng học các ngôn ngữ kiểu đồng bộ như PHP, Java... chuyển sang Javascript.

Async/Await

Async/Await là một tính năng giúp làm việc với các chức năng không đồng bộ đơn giản hơn. Nó được xây dựng dựa trên Promise và tương thích với tất cả các API dựa trên Promise hiện có.

Cú pháp của *Async/Await* rất đơn giản, chúng ta sử dụng từ khóa *async* trước một hàm. Từ khóa *await* chỉ được sử dụng bên trong một hàm được định nghĩa bằng khóa *async*.

Ví dụ:

```
async function doSomething() {  
  let a = await callAsyncFunction1();  
  let b = await callAsyncFunction2();  
  return a + b;  
}
```

Một vài điểm lưu ý khi sử dụng *Async/Await*:

- Từ khóa *await* chỉ được sử dụng bên trong hàm được định nghĩa với từ khóa *async*. *Await* không thể nằm trong hàm không được khai báo từ khóa *async* phía trước
- *async/await* thực chất là một promises. Do đó, tất cả những api mà trước đó bạn vẫn quen sử dụng với promise, thì nay đều dùng được với *async/await*.

Ok, giờ chúng ta quay trở lại với ví dụ ngủ dậy mỗi buổi sáng. Nếu chúng ta sử dụng *async/await* sẽ như thế nào nhé.

Phần định nghĩa các hàm như *wakeUp()*, *brushTeeth()*... không cần phải thay đổi, có thay đổi chỉ là lúc chúng ta gọi và sử dụng chúng thôi.

//Định nghĩa các hành động

```
function wakeUp() {  
    return new Promise(function (resolve, reject) {  
        setTimeout(function () {  
            console.log("Thức dậy ");  
            //Làm xong, tiến hành việc nào đó khác  
            resolve();  
        }, 1000);  
    });  
}
```

```
function brushTeeth() {  
    return new Promise(function (resolve, reject) {  
        setTimeout(function () {  
            console.log("Đánh răng");  
            //Làm xong, tiến hành việc nào đó khác  
            resolve();  
        }, 2000);  
    });  
}
```

```
function eatBreakfast() {  
    return new Promise(function (resolve, reject) {  
        setTimeout(function () {  
            console.log("Ăn sáng");  
            //Làm xong, tiến hành việc nào đó khác  
            resolve();  
        }, 1000);  
    });  
}
```

```
function gotoSchool() {  
    return new Promise(function (resolve, reject) {  
        setTimeout(function () {  
            console.log("Đi học");  
            //Làm xong, tiến hành việc nào đó khác
```

```

        resolve();
    }, 3000);
});
}

```

```

// Trình tự mỗi sáng bạn sẽ làm lần lượt:
// Thức dậy => đánh răng => ăn sáng => đi học

```

```

async function main() {
    await wakeUp();
    await brushTeeth();
    await eatBreakfast();
    await gotoSchool();
}

```

Code nhìn đẹp như cách bạn vẫn hay
viết kiểu đồng bộ như Java hay PHP vậy

```

// Gọi hàm
main();
//Kết quả =>
// Thức dậy
// Đánh răng
// Ăn sáng
// Đi học

```

Bạn thấy chưa, code giờ nhìn quá đẹp luôn!

Với Promise, bạn bắt lỗi bằng cách sử dụng `reject()` sau đó lúc gọi thì lỗi (nếu có) sẽ nhảy vào hàm `catch()`. Còn với `async/await`, chúng ta bắt lỗi bằng cách sử dụng cặp từ khóa `try{...} catch{...}`, rất quen thuộc phải không!?

Mình sẽ lấy ví dụ về cách bắt lỗi và xử lý lỗi với `async/await`

```

//Định nghĩa các hành động
function wakeUp(isWakeUpStatus) {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            console.log("Ê, dậy đi! đến giờ đi làm rồi");
            //Làm xong, tiến hành việc nào đó khác
            if (isWakeUpStatus) {
                resolve("Mình đã thức dậy thành công! Thật tuyệt vời");
            } else {

```

```

        reject("zzzzz, ngủ tiếp!");
    }
    resolve();
}, 1000);
});
}

async function main() {
    try {
        let result = await wakeUp(false);
        console.log(result);
    } catch (error) {
        console.log(error);
    }
}

// Gọi hàm
main();
// Kết quả =>
// zzzzz, ngủ tiếp

```

Bạn thấy cách viết theo kiểu Async/Await hay Promise hay hơn? Thực tế thì các dự án đang có xu hướng viết theo kiểu async/await hơn. Quan điểm cá nhân mình thì tùy bạn, tùy sở thích và yêu cầu cụ thể mà bạn chọn cách viết nào cho hợp lý.

>> Tìm hiểu 4 nguyên lý của lập trình OOP

>> Tìm câu trả lời cho câu hỏi: JS có phải ngôn ngữ lập trình OOP không?

>> Tìm hiểu xem JS hỗ trợ các nguyên lý OOP như thế nào?

PHẦN 7

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG VỚI JS

Từ đầu cuốn sách đến giờ, bạn có nhận ra rằng mọi thứ trong Javascript đều là đối tượng, từ các biến, function, Array... (trừ duy nhất các dữ liệu kiểu nguyên thủy).

Như mình từng đề cập, chỉ cần bạn nắm chắc được Object là bạn sẽ học Javascript rất nhanh chóng và cảm thấy nó đơn giản hơn bao giờ hết.

Trước đây, mọi người nói tới Javascript là nghĩ ngay đây là ngôn ngữ lập trình kịch bản, kiểu script, viết là chạy theo từng block riêng lẻ, chỉ dùng để xử lý logic, tạo hiệu ứng cho các website. Nhưng hiện nay, Javascript đã khác trước rất nhiều, với sự hỗ trợ làm việc mạnh mẽ với Object, liệu Javascript có thực sự phù hợp với triết lý lập trình hướng đối tượng (OOP) hay không? Phần 7 của cuốn sách, chúng ta cùng nhau tìm hiểu chủ đề này.

Để hiểu được trọn vẹn phần này, mình khuyên bạn nên đọc lại và hiểu thật rõ về Object. Mời bạn ôn lại [kiến thức về Object](#)

Nguyên lý lập trình hướng đối tượng (OOP)

Lập trình hướng đối tượng là một trong những mô hình lập trình phổ biến nhất hiện nay. Hầu hết các ngôn ngữ lập trình “kinh điển” như Java, C#, Ruby... đều hỗ trợ mô hình này. Một trong những điểm mạnh của mô hình lập trình

này là nó đẩy mạnh việc module hóa và tái sử dụng mã nguồn – hai tính năng quan trọng trong các dự án xây dựng phần mềm lớn.

Tuy nhiên, không có một tài liệu hay tiêu chuẩn chính thức nào mô tả chính xác OOP là gì. Định nghĩa OOP chủ yếu dựa trên nhận thức, kinh nghiệm truyền miệng là chính.

Nhưng dù sao, định nghĩa dưới đây được đông đảo giới lập trình viên công nhận: Lập trình hướng đối tượng là mô hình lập trình thỏa mãn hai yêu cầu:

- Khả năng mô hình hóa một vấn đề thông qua các đối tượng
- Cho phép module hóa và tái sử dụng mã nguồn.

Để đáp ứng 2 yêu cầu trên, các ngôn ngữ hỗ trợ OOP phải thỏa mãn 4 nguyên tắc:

- **Tính đóng gói:** Đây là tính năng đóng gói dữ liệu vào một đối tượng. Các đối tượng khác không thể tác động trực tiếp đến dữ liệu bên trong và làm thay đổi trạng thái của đối tượng mà bắt buộc phải thông qua các phương thức công khai do đối tượng đó cung cấp.
- **Tính kế thừa:** Là tính chất thể hiện rõ nét nhất của việc tái sử dụng mã nguồn. Đây là cơ chế cho phép một đối tượng có được một số hoặc tất cả các tính năng của một đối tượng khác.
- **Tính đa hình:** Tính đa hình trong lập trình OOP cho phép các đối tượng khác nhau thực thi chức năng giống nhau theo những cách khác nhau.
- **Tính trừu tượng:** Tính trừu tượng giúp loại bỏ những thứ phức tạp, không cần thiết của đối tượng và chỉ tập trung vào những gì cốt lõi, quan trọng.

Một ngôn ngữ lập trình đáp ứng được các tiêu chí trên thì được coi là một ngôn ngữ lập trình hướng đối tượng.

Trong khuôn khổ cuốn sách này, mình sẽ không giải thích và đề cập quá sâu vào nguyên lý OOP. Nếu bạn chưa có nền tảng về OOP thì có thể tìm hiểu thêm trên blog của mình: VNTALKING.COM

Phần này, chúng ta sẽ chỉ tập trung tìm hiểu xem cách chúng ta có thể lập trình OOP bằng Javascript như thế nào.

Javascript có hướng đối tượng không?

Sau khi đã xác định được các nguyên tắc để một ngôn ngữ lập trình được coi là ngôn ngữ lập trình hướng đối tượng. Vậy liệu Javascript có được coi là ngôn ngữ lập trình OOP?

Cho tới thời điểm hiện tại, rất nhiều cuộc tranh luận để trả lời cho câu hỏi trên. Nhiều developer cho rằng, Javascript không có khái niệm Class nên không thể coi Javascript là ngôn ngữ OOP được. Nghe có vẻ hợp lý đúng không?

Nhưng nếu đọc kỹ, thực sự trong các định nghĩa về OOP thì abstract Class không phải nguyên tắc bắt buộc, có chăng là do nhiều người quá quen thuộc với Java hay C# nên mới nhầm lẫn như vậy.

Với cá nhân mình, mặc dù Javascript không có abstract classes như các ngôn ngữ OOP khác, nhưng đó không phải yếu tố tiên quyết để coi Javascript không phải là ngôn ngữ OOP.

OK nhỉ! Lý thuyết về OOP và Javascript đến đây thôi nhé. Chúng ta thử so sánh một ví dụ đơn giản nhưng được viết theo hai cách khác nhau: theo kiểu thủ tục và theo kiểu OOP.

```
// Cách viết theo kiểu thủ tục
let luongCoBan = 30000;
let gioLamThem = 10;
let rate = 20;

function nhanLuong(luongcoban, gioLamThem, rate) {
    return luongcoban + (gioLamThem * rate);
}
console.log('PROCEDURAL => ' + nhanLuong(luongCoBan, gioLamThem, rate));
```


Còn đây là cách viết theo kiểu OOP.

```
// OOP
let nhanvien = {
  luongCoBan: 30000,
  gioLamThem: 10,
  rate: 20,
  nhanLuong: function() {
    return this.luongCoBan + (this.gioLamThem * this.rate);
  }
};
console.log('OOP => ' + nhanvien.nhanLuong());
```

Cả hai cách đều cho một kết quả như nhau, chỉ là cách tiếp cận và giải quyết vấn đề khác nhau mà thôi.

Phần tiếp theo của cuốn sách, chúng ta sẽ cùng nhau khám phá cách Javascript hỗ trợ tính trừu tượng và các nguyên tắc của OOP như thế nào nhé.

Tính kế thừa

Như mình đã nói, tính kế thừa một tính chất được sử dụng rất nhiều trong mô hình OOP. Không giống với các ngôn ngữ OOP khác, khi mà class kế thừa từ một class khác. Trong Javascript, thay vì class kế thừa mà là các Object kế thừa các tính năng của nhau (method và property).

Để sử dụng tính kế thừa trong Javascript, chúng ta có 2 phương pháp:

- **Cách 1:** Sử dụng từ khóa *extends* để kế thừa các Object được định nghĩa bằng từ khóa *class*.
- **Cách 2:** Sử dụng Prototype



Trong Javascript cũng có từ khóa class, nhưng khái niệm class trong Javascript hơi khác so với các ngôn ngữ lập trình như Java mà mọi người đã biết từ trước. Class trong javascript bản chất là “hàm đặc biệt”, mà đã là hàm thì suy cho cùng vẫn là Object. Bạn có thể định nghĩa Class giống như định nghĩa một hàm.

Chúng ta sẽ cùng nhau xem ví dụ sau để hiểu rõ hơn nhé.

```
class Person{
  constructor(name){
    this.name = name;
  }
  //method to return the string
  eat (){
    return (`${this.name} đang ăn`);
  }
}

class Student extends Person{
  constructor(name,id){
    //super keyword dùng để gọi hàm khởi tạo của class cha
    super(name);
    this.id = id;
  }
  eat(){
    return (`${super.eat()},\nMã sinh viên là: ${this.id}`);
  }
}

let sinhvien = new Student('Sơn Dương',22);
console.log(sinhvien.eat());
// Kết quả ==>
//Sơn Dương đang ăn,
//Mã sinh viên là: 22
```

Trong ví dụ trên, chúng ta định nghĩa một đối tượng *Person*, với một thuộc tính là *name* và một phương thức *eat()* để mô tả về hành vi ăn uống. Sau đó, chúng ta định nghĩa tiếp một đối tượng *Student* kế thừa đối tượng *Person*, nên đối tượng *Student* có đầy đủ các thuộc tính và phương thức của đối tượng *Person*, ngoài ra, đối tượng *Student* còn có những thuộc tính của riêng nó (ví dụ như thuộc tính *id*)

Tiếp theo, chúng ta sẽ xem cách kế thừa bằng prototype sẽ thế nào nhé.

```
function Person(firstName, lastName) {
  this.FirstName = firstName || 'unknown';
  this.LastName = lastName || 'unknown';
}
```

```

}
Person.prototype.getFullName = function () {
    return this.FirstName + ' ' + this.LastName;
};

```

Ở ví dụ này, mình định nghĩa một *function Person* với 2 thuộc tính là *FirstName* và *LastName*, và một hàm *getFullName()* - Hàm này được thêm vào class thông qua prototype object.

Giờ chúng ta sẽ tạo một *function Student*, kế thừa từ *function Person*.

```

function Student(firstName, lastName, schoolName, grade) {
    Person.call(this, firstName, lastName);
    this.SchoolName = schoolName || 'unknown';
    this.Grade = grade || 0;
}
//Student.prototype = Person.prototype;
Student.prototype = new Person();
Student.prototype.constructor = Student;

```

Các bạn để ý rằng, mình đã đặt *Student.prototype* là một *Person*.

Từ khóa *new* tạo một đối tượng *Person* và cũng gán *Person.prototype* cho prototype object của đối tượng mới. Và cuối cùng gán đối tượng được tạo cho *Student.prototype*

Bây giờ bạn có thể tạo một đối tượng *student* sử dụng các thuộc tính và hàm của lớp *Person* như sau:

```

function Person(firstName, lastName) {
    this.FirstName = firstName || 'unknown';
    this.LastName = lastName || 'unknown';
}
Person.prototype.getFullName = function () {
    return this.FirstName + ' ' + this.LastName;
};
function Student(firstName, lastName, schoolName, grade) {
    Person.call(this, firstName, lastName);
    this.SchoolName = schoolName || 'unknown';
}

```

```

    this.Grade = grade || 0;
}
//Student.prototype = Person.prototype;
Student.prototype = new Person();
Student.prototype.constructor = Student;
var std = new Student('Dương Anh', 'Sơn', 'XYZ', 10);

console.log(std.getFullName()); // James Bond
console.log(std instanceof Student); // true
console.log(std instanceof Person); // true

```

Nhìn cũng giống với kế thừa như các ngôn ngữ kinh điển (Java, C#) nhỉ!

Tính đóng gói

Tính đóng gói là tính chất cho phép bạn đưa tất cả thông tin, dữ liệu quan trọng vào bên trong một đối tượng (object). Các đối tượng khác muốn truy cập vào dữ liệu được đóng gói bắt buộc phải thông qua các hàm được cung cấp bởi đối tượng đó.

Mục đích chính của tính đóng gói là bảo vệ dữ liệu bên trong của đối tượng. Dữ liệu đó hoàn toàn không nên bị sửa đổi một cách bất ngờ bởi những mã lệnh bên ngoài, từ những thành phần khác của chương trình.

```

const Book = function(t, a) {
    let title = t;
    let author = a;

    return {
        summary : function() {
            console.log(`${title} được viết và biên tập bởi ${author}.`);
        }
    }
}

// Khởi tạo đối tượng
const book = new Book('Cuốn sách Javascript từ cơ bản tới nâng cao', 'VNTALKI NG');

//Gọi hàm

```

```
book.summary();  
// Kết quả =>  
// Cuốn sách Javascript từ cơ bản tới nâng cao được viết và biên tập bởi VNTA  
LKING.
```

Trong đoạn code trên, hai thuộc tính *title* và *author* chỉ khả dụng trong phạm vi của hàm *Book* (đối tượng bên ngoài không thể truy cập được) và hàm *summary()* được dùng cho các đối tượng khác gọi. Do đó, người ta gọi *title* và *author* được đóng gói bên trong *Book*

Kỹ thuật hỗ trợ tính chất đóng gói này chính là kiến thức phạm vi của một biến, hàm. Bạn có thể đọc lại: [Phạm vi của biến](#)

Tính đa hình và trừu tượng

Với hầu hết các ngôn ngữ lập trình, để hỗ trợ tính đa hình sẽ phải hỗ trợ các tính năng sau:

- Overloading (nạp chồng): Tức là cho phép tạo nhiều hàm có tên giống nhau nhưng tham số khác nhau: khác nhau về số lượng tham số, kiểu dữ liệu.
- Hỗ trợ các kiểu dữ liệu chung chung, không biết trước được nó thuộc kiểu dữ liệu nào.

Với tính nạp chồng (Overloading), với các ngôn ngữ có kiểu dữ liệu tường minh như Java, C#... thì sẽ rất dễ hiểu. Ví dụ như C# chẳng hạn.

```
// Ví dụ Overloading bằng C#  
public int CountItems(int x) {  
    return x.ToString().Length;  
}  
public int CountItems(string x) {  
    return x.Length;  
}
```

Bạn thấy đấy, cùng một tên hàm nhưng tham số đầu vào có kiểu dữ liệu khác nhau. Hoặc có thể khác nhau về số lượng tham số.

```
// Ví dụ Overloading bằng C#
public int Sum(int x, int y) {
    return Sum(x, y, 0);
}
public int Sum(int x, int y, int z) {
    return x + y + z;
}
```

Trong trường hợp trên, hàm *sum()* có thể tính tổng của 2 hoặc 3 số. Khi bạn sử dụng, bạn có thể gọi bất kỳ hàm nào, chương trình sẽ tự động phát hiện chính xác hàm tùy vào tham số bạn truyền vào.

Với Javascript, khi mà biến không có kiểu dữ liệu thì phải làm sao? Tính Overloading được hỗ trợ như nào nhỉ?

Với trường hợp hàm *CountItems()*, Javascript sẽ viết như sau:

```
function countItems(x) {
    return x.toString().length;
}
```

```
// lúc sử dụng
console.log(countItems(2))
// Kết quả: 1

console.log(countItems("VNTALKING"))
// Kết quả: 9
```

Bạn thấy đấy, vì biến trong javascript không có kiểu, nên bạn truyền vào là số hay chuỗi đều được, bạn không cần phải viết code chia tách làm 2 hàm nữa.

Tương tự, với hàm *sum()* sẽ như sau:

```
function sum(x, y, z) {
    x = x ? x : 0;
    y = y ? y : 0;
    z = z ? z : 0;
    return x + y + z;
}
```

```
}
```

```
// Lúc sử dụng  
console.log(sum(1,2))  
//==> kết quả: 3  
  
console.log(sum(1,2,3))  
//Kết quả: 6
```

Hàm này cũng vậy, lúc sử dụng, bạn gọi hàm và truyền 1 tham số cũng được, truyền 2 tham số cũng được... bạn không nhất thiết phải định nghĩa 3 hàm tương ứng mà khác nhau về số lượng tham số truyền vào.

Tính ra Javascript lại khá tiện lợi, hơn hẳn so với C# đúng không?!

Tính trừu tượng được hiểu là cách chúng ta ẩn cách giải quyết một vấn đề, chỉ hiển thị các tính năng cần thiết cho người sử dụng. Nói cách khác, tính chất này giúp chúng ta ẩn những chi tiết không liên quan và chỉ hiển thị những gì cần thiết cho người dùng.

Việc áp dụng tính chất trừu tượng trong Javascript là không rõ ràng. Như ví dụ dưới đây, bạn có thể hiểu là phần thực hiện để in ra màn hình tên cuốn sách và tác giả đã được ẩn đi, người gọi không biết được chương trình làm như nào, chỉ biết là được phép gọi hàm `giveSummary` để sử dụng mà thôi. *(Về cơ bản thì mình thấy tính chất này nó tương đối giống với tính đóng gói ở trên)*

```
const Book = function (getTitle, getAuthor) {  
  // Private properties  
  let title = getTitle;  
  let author = getAuthor;  
  
  // Public method  
  this.giveTitle = function() {  
    return title;  
  }  
  
  // Private method  
  const summary = function() {
```

```

        return `${title} được viết bởi ${author}.`
    }
    // Public method that has access to private method.
    this.giveSummary = function() {
        return summary()
    }
}
const book = new Book('Javascript từ cơ bản tới nâng cao', 'VNTALKING');
console.log(book.giveTitle());
// Kết quả: Javascript từ cơ bản tới nâng cao

console.log(book.summary());
// Báo lỗi: Uncaught TypeError: book1.summary is not a function

console.log(book.giveSummary());
// Kết quả: Javascript từ cơ bản tới nâng cao được viết bởi VNTALKING.

```

Trong phần này, chúng ta đã khám phá các nguyên tắc cơ bản của lập trình hướng đối tượng, cách mà Javascript hỗ trợ OOP như thế nào. Nếu có một thang điểm để đánh giá mức độ hỗ trợ OOP của Javascript thì cá nhân mình cho 5/10 điểm.

Với khả năng hỗ trợ OOP chỉ ở mức trung bình, bạn vẫn có thể xây dựng ứng dụng theo mô hình OOP. Tuy nhiên, sẽ có những khó khăn nhất định trong quá trình phát triển. Để có thể hỗ trợ OOP tốt hơn, bạn có thể xem xét lựa chọn sử dụng TypeScript, một dạng ngôn ngữ tiền Javascript. Tức là Typescript có cú pháp, công cụ hỗ trợ OOP tốt hơn, nhưng khi chạy chương trình, nó sẽ phải biên dịch sang Javascript.

PHẦN 8

CÚ PHÁP ES6

ES6 là chữ viết tắt của ECMAScript 6 hay ECMAScript 2015, là một tập các quy tắc lập trình không chỉ dành riêng cho Javascript mà nhiều ngôn ngữ lập trình khác nữa như ActionScript. Nhưng có lẽ Javascript là ngôn ngữ lập trình phổ biến nhất nên nhiều người mặc định Javascript và tiêu chuẩn ECMAScript là một cặp đi liền với nhau.

ES6 là phiên bản tiếp theo của ECMAScript, được xuất bản năm 2015, hiện nay ECMAScript đã ra đến phiên bản ES11, nhưng ES6 là phiên bản mà có rất nhiều cải tiến nổi bật, bổ sung rất nhiều tính năng hay hơn hẳn so với bản trước đó và hiện cũng được sử dụng rất nhiều trong các dự án hiện đại, đó cũng là lý do mình đưa vào trong nội dung cuốn sách này.

Hiện tại, ES6 đã được hỗ trợ trên hầu hết các trình duyệt hiện đại như Chrome, Firefox, Edge... và cả trên các phiên bản Node.JS mới nhất.

Không giống với cú pháp mà chúng ta đã học trong các phần trước của cuốn sách, với ES6 bạn sẽ có phần bỏ ngỡ khi đọc mã nguồn của ai đó mà nhìn toàn ký tự “suy ra”, “dấu 3 chấm”... kiểu như sau:

```
var pairs = evens.map(v => ({even: v, odd: v + 1}));
// Statement bodies
nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
```

```
});
```

```
// Hoặc
```

```
const arr1 = [1,2,3]
```

```
const arr2 = [4,5,6]
```

```
const arr3 = [...arr1, ...arr2]
```

Trong chương này của cuốn sách, mình không thể giới thiệu hết tất cả cú pháp của ES6, nhưng sẽ giới thiệu những cú pháp, tính năng phổ biến nhất của ES6 để bạn có thể sử dụng và đọc hiểu mã nguồn Javascript được viết theo chuẩn ES6.



*Có một điểm quan trọng mà bạn cần phải biết đó là ES6 không phải là phiên bản mới hoàn toàn (kiểu đập bỏ hết cái cũ, xây cái mới), nó đơn giản là một phiên bản tiếp theo của ECMAScript, **có tính tương thích ngược**, giống như HTML5 là phiên bản tiếp theo của HTML vậy. **Tức là bạn vẫn có thể viết mã Javascript theo chuẩn ES5 hay ES6 trong cùng một dự án**, miễn là trình duyệt có hỗ trợ phiên bản ES6 là được. Ví dụ như trình duyệt IE9 trở xuống không có hỗ trợ ES6 nên nếu bạn viết mã theo cú pháp ES6 sẽ bị báo lỗi.*

OK, giờ còn chờ gì nữa, bắt đầu thôi nhĩ!

String

String là kiểu dữ liệu quan trọng bậc nhất trong mọi ngôn ngữ lập trình, String được sử dụng ở mọi nơi. Do đó, mọi cải tiến đều cố gắng giúp cho developer làm việc String được thuận tiện và hiệu quả nhất. Phiên bản ES6 này cũng không phải ngoại lệ.

Template Literals

Từ phiên bản ES6, Javascript được bổ sung rất nhiều tính năng hay ho liên quan tới String. Trong đó, đáng kể nhất chính là tính năng **Template Literals**. Tức là bạn có thể truyền biến vào trong một String.

Template literals là kiểu cú pháp mới, cho phép bạn nhúng biểu thức vào trong một chuỗi String. Để sử dụng cú pháp này, chuỗi String phải được bọc

trong ký tự `` thay vì ký tự ngoặc đơn hay ngoặc kép như thông thường. Biểu thức bắt đầu bằng cú pháp `${...}` Dưới đây là ví dụ để bạn tiện hình dung:

```
// Viết theo kiểu cũ - ES5
const programmingLanguage = 'Javascript'
const website = "VNTALKING"
const sentence = "Chúng ta cùng nhau học lập trình " + programmingLanguage + ' cùng với ' + website
console.log(sentence)
// Kết quả ==>
// "Chúng ta cùng nhau học lập trình Javascript cùng với VNTALKING"

// Viết theo kiểu mới ES6
const programmingLanguage = 'Javascript'
const website = "VNTALKING"
const sentence = `Chúng ta cùng nhau học lập trình ${programmingLanguage} cùng với ${website}`
console.log(sentence)
// Kết quả ==>
// "Chúng ta cùng nhau học lập trình Javascript cùng với VNTALKING"
```

Bạn hoàn toàn có thể sử dụng logic trong biểu thức. Ví dụ:

```
// Viết theo kiểu mới ES6
const programmingLanguage = 'Javascript'
const website = "VNTALKING"
const age = 18;

const sentence = `Chúng ta cùng nhau học lập trình ${age > 18 ? programmingLanguage : 'JAVA'} cùng với ${website}`
console.log(sentence)
// Kết quả==>
// Chúng ta cùng nhau học lập trình JAVA cùng với VNTALKING
```

Tagged Templates

Với phương pháp sử dụng **Template Literals**, bạn có thể tạo chuỗi String nhiều dòng, chèn giá trị vào trong chuỗi mà không cần phải nối, nhưng sức

mạnh của template Literals không chỉ dừng lại ở đó, nó còn có thể làm được nhiều hơn khi kết hợp với tagged templates.

Một **Tagged Templates** cho phép bạn chuyển đổi trên template literal và trả về một giá trị. Tag này được chỉ định ở đầu của template ngay trước ký tự ```. Như ví dụ minh họa dưới đây cho dễ hình dung nhé.

```
let message = tag`Hello world`;
```

Trong ví dụ trên, *tag* là template tag được áp dụng cho mẫu ``Hello world``. Đọc đến đây, có vẻ chúng ta vẫn chưa thấy tác dụng của tag template là để làm gì, tại sao nó lại hữu dụng đúng không?! Cứ bình tĩnh, điều thú vị đang chờ đợi bạn ở phía dưới đây.

Tag thực chất đơn giản là một function được gọi với template đã xử lý dữ liệu. Tagged Template cho phép bạn parse template string với một function. Điều này khiến mình liên tưởng tới khái niệm tag trong quản lý mã nguồn GIT. Tag là một cách đánh dấu đại diện cho một cái gì đó mà bạn muốn.

Cú pháp định nghĩa tag:

```
function tag(literals, ...substitutions) {  
  // trả về một chuỗi String  
}
```

Còn đây là cú pháp lúc sử dụng:

```
function`template string with ${expression}`;
```

Để hiểu rõ hơn, chúng ta sẽ xem xét ví dụ sau:

```
let count = 10,  
    price = 0.25,  
    message = passthru`${count} vật phẩm có giá $${(count * price).toFixed(2)}.`;
```

Bạn thấy đấy, chúng ta gọi một hàm để xử lý động một chuỗi string nhưng nhìn như là một string, không giống với cách gọi hàm thông thường. Với hàm tag - ở đây chúng ta đặt tên là *passthru()*, bạn có thể làm bất cứ điều gì với chuỗi String đó để tạo ra một chuỗi đúng yêu cầu của bài toán.

Như đây là một cách:

```
function passthru(literals, ...substitutions) {
  let result = '';
  // chạy một vòng lặp để đếm số lượng
  for (let i = 0; i < substitutions.length; i++) {
    result += literals[i];
    result += substitutions[i];
  }
  // thêm chữ cuối cùng
  result += literals[literals.length - 1];
  return result;
}
```

Với đoạn code trên, kết quả cuối bạn nhận được sẽ là chuỗi sau:

```
console.log(message); // "10 vật phẩm có giá $2.50."
```

Dưới đây là toàn bộ code của ví dụ minh họa trên

```
function passthru(literals, ...substitutions) {
  let result = '';
  // chạy một vòng lặp để đếm số lượng
  for (let i = 0; i < substitutions.length; i++) {
    result += literals[i];
    result += substitutions[i];
  }
  // thêm chữ cuối cùng
  result += literals[literals.length - 1];
  return result;
}

let count = 10,
    price = 0.25,
    message = passthru`${count} vật phẩm có giá ${((count * price).toFixed(2))}
  `;

console.log(message); // "10 vật phẩm có giá $2.50."
```

Function

Phần này mình sẽ trình bày những điểm mới mà ES6 có hoặc khác biệt so với bản ES5.

Function với tham số có giá trị mặc định

Nếu như ở phiên bản ES5, để có thể gán giá trị mặc định cho tham số trong trường hợp người truyền vào *undefined* hoặc *null*, bạn chỉ có thể thực hiện trong thân của function mà thôi.

Ví dụ:

```
// Cách viết ES5
function makeRequest(url, timeout, callback) {
  timeout = timeout || 2000;
  callback = callback || function () {};
  // the rest of the function
}
```

Trong ví dụ trên, hai tham số *timeout* và *callback* là những tham số tùy chọn, tức là người dùng có thể truyền vào hoặc không truyền giá trị vào cũng được. Trong trường hợp người dùng không truyền gì vào, tức là chúng có giá trị *undefined*, lúc đó nhờ toán tử OR (||) sẽ tự động gán giá trị mặc cho biến. Toán tử OR sẽ luôn *return* giá trị toán hạng thứ 2 nếu toán hạng đầu tiên bị sai.

Với ES6, bạn có cách viết ngắn gọn hơn rất nhiều, bạn có thể khởi tạo giá trị mặc định ngay tại lúc định nghĩa tham số.

```
// Cách viết ES6
function makeRequest(url, timeout = 2000, callback = function() {}) {
  // thực hiện logic của hàm
}
```

Nhìn code ngắn gọn hơn hẳn đúng không!?

Arrow function

Arrow function là một trong những thay đổi nổi bật của phiên bản ES6. Đúng như tên gọi, Arrow function là hàm được định nghĩa bằng dấu mũi tên (=>),

thay vì từ khóa *function*. Về cơ bản, Arrow function cũng tương tự như một hàm được định nghĩa thông thường bằng từ khóa *function*, nó chỉ khác một chút thôi, đó là:

- Không sử dụng được *this*, *super*, *arguments*: Khác với function thông thường, arrow function không có bind. Vì vậy, không định nghĩa lại *this*. Do đó, *this* sẽ tương ứng với ngữ cảnh gần nhất của nó.
- Không thể gọi hàm với từ khóa *new*: Các Arrow function không có *constructor*, do đó không thể dùng arrow function để làm hàm constructor cho các class.
- Không có prototype: Arrow function không có thuộc tính prototype.

Cú pháp:

Arrow function có cú pháp rất đơn giản, bạn chỉ cần bỏ từ khóa *function* và thay thế bằng dấu mũi tên(\Rightarrow). Trong trường hợp, hàm có 1 tham số, bạn có thể bỏ luôn dấu ngoặc chứa tham số cũng được. Sau đây là một số ví dụ cách viết arrow function.

```
let sum = (num1, num2) => num1 + num2;
// Tương đương với cách viết function truyền thống như sau
let sum = function(num1, num2) {
    return num1 + num2;
};
```

Với trường hợp hàm có 1 tham số, bạn có thể bỏ dấu ngoặc ().

```
let reflect = value => value;
// Tương đương với cách viết function truyền thống như sau
let reflect = function (value) {
    return value;
};
```

Với trường hợp hàm không có tham số thì sao?

```
let getName = () => 'Dương Anh Sơn';
// Tương đương với cách viết function truyền thống như sau
let getName = function () {
```

```
    return 'Dương Anh Sơn';  
};
```

Cuối cùng, trong trường hợp bạn muốn định nghĩa hàm mà không trả về giá trị nào cả, chỉ đơn giản thực hiện một tác vụ nào đó mà thôi. Lúc này, thân hàm bắt buộc phải để trong dấu ngoặc nhọn.

```
let doNothing = () => {  
    console.log('làm một cái gì đó');  
};  
// Tương đương với cách viết function truyền thống như sau  
let doNothing = function () {  
    console.log('làm một cái gì đó');  
};
```

Class

Như ở phần trên cuốn sách, chúng ta đã biết trong Javascript không hỗ trợ Class giống như cách các ngôn ngữ thuần OOP như Java, C#... đã làm. Điều này hoàn toàn đúng với JS từ phiên bản ES5 trở về trước. Do nhu cầu lớn từ nhiều developer mà rất nhiều thư viện ra đời để giúp javascript có thể hỗ trợ Class. Chính vì điều này mà từ phiên bản ES6 đã chính thức đưa Class vào mục hỗ trợ sẵn.

Khai báo một class

Từ ES5 trở về trước, để sử dụng class, cách làm khả dĩ nhất là sử dụng prototype. Ví dụ như dưới đây:

```
function Person(name) {  
    this.name = name;  
}  
Person.prototype.sayName = function () {  
    console.log(this.name);  
};  
var person = new Person('Dương Anh Sơn');  
person.sayName(); // Kết quả: "Dương Anh Sơn"  
console.log(person instanceof Person); // true  
console.log(person instanceof Object); // true
```


Trong đoạn code trên, hàm `Person` là một hàm khởi tạo với mục đích là khởi tạo một thuộc tính *name*. Còn hàm sẽ được gán cho prototype. Khi sử dụng, chúng ta sẽ dùng từ khóa *new* để tạo một instance.

Từ ES6, bạn có thể sử dụng từ khóa *class* để định nghĩa một class. Lúc này nhìn mã nguồn của bạn tương đối giống với các ngôn ngữ thuần OOP khác.

```
class Person {
  // Tương đương với hàm function Person(name) {...} ở trên
  constructor(name) {
    this.name = name;
  }
  // Tương đương với Person.prototype.sayName
  sayName() {
    console.log(this.name);
  }
}
let person = new Person('Dương Anh Sơn');
person.sayName(); // outputs "Dương Anh Sơn"
console.log(person instanceof Person); // true
console.log(person instanceof Object); // true
```

Về cơ bản trong dự án, bạn có thể viết theo cách ES5 hay ES6 đều được, miễn là nó giải quyết đúng yêu cầu của bài toán. Tuy nhiên, giữa hai cách viết có sự khác nhau nho nhỏ.

Vậy tại sao bạn sử dụng từ khóa **class**?

- Sử dụng từ khóa **class** sẽ không hỗ trợ hoisted hoàn toàn. Tức là Class hoạt động giống như bạn khai báo biến bằng từ khóa **let** vậy. Cụ thể như nào thì mời bạn đọc lại phần [khái niệm và cơ chế hoisting](#) ở phần trên cuốn sách nhé.
- Tất cả code được khai báo trong *class* đều mặc định chạy dưới chế độ “strict mode”. Bạn không có lựa chọn nào khác cả.
- Gọi hàm khởi tạo để tạo mới một instance mà không dùng từ khóa *new* sẽ bị lỗi ngay lập tức.
- Không thể đặt tên hàm bên trong class giống với tên của class được.

Một số cách viết định nghĩa class khác mà bạn có thể sử dụng ngoài cách viết kiểu phổ biến như trên.

```
let PersonClass = class {
  constructor(name) {
    this.name = name;
  }

  // Tương đương với Person.prototype.sayName
  sayName() {
    console.log(this.name);
  }
};

let person = new PersonClass('VNTALKING');
person.sayName(); // Kết quả "VNTALKING"
```

Hoặc cũng có thể viết như này:

```
let PersonClass = class PersonClass2 {
  constructor(name) {
    this.name = name;
  }

  // tương đương với Person.prototype.sayName
  sayName() {
    console.log(this.name);
  }
};

let person = new PersonClass("VNTALKING");
person.sayName(); // Kết quả "VNTALKING"
```

Kế thừa

Trong bản ES6 giới thiệu một từ khóa mới để thực hiện kế thừa, đó là *extends*, bản chất từ khóa này là wrapper một prototype tới một function. Bạn cũng có thể sử dụng từ khóa *super()* để copy hàm khởi tạo của class cha.

```
// ES5
function Rectangle(length, width) {
  this.length = length;
```

```

    this.width = width;
}
Rectangle.prototype.getArea = function () {
    return this.length * this.width;
};
function Square(length) {
    Rectangle.call(this, length, length);
}
Square.prototype = Object.create(Rectangle.prototype, {
    constructor: {
        value: Square,
        enumerable: true,
        writable: true,
        configurable: true,
    },
});
var square = new Square(3);
console.log(square.getArea()); // Kết quả: 9

```

Cách viết trên sẽ tương đương với cách trong ES6 như sau:

```

class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }
    getArea() {
        return this.length * this.width;
    }
}
class Square extends Rectangle {
    constructor(length) {
        // Tương đương với Rectangle.call(this, length, length)
        super(length, length);
    }
}
var square = new Square(3);
console.log(square.getArea()); // Kết quả: 9

```

Nhìn cách viết mới này khá giống với cách viết của JAVA phải không!?

Destructuring

Object và Array là hai thành phần được sử dụng thường xuyên nhất trong Javascript, bên cạnh định dạng dữ liệu có cấu trúc JSON. Khi làm việc với Javascript, bạn sẽ phải thường xuyên truy xuất vào JSON để bóc tách ra Object hay Array. Từ phiên bản ES6, nhà phát triển giới thiệu một khái niệm mới, đó là destructuring, với mục đích để đơn giản hóa công việc truy xuất lấy giá trị Array hoặc Object và đưa chúng vào biến.

Destructuring chia làm 2 nhóm:

- Object Destructuring
- Array Destructuring

Chúng ta sẽ cùng nhau tìm hiểu từng nhóm một nhé.

Object Destructuring

Từ ES5 trở về trước, bạn bắt buộc phải truy xuất lần lượt từng thuộc tính của Object, sau đó mới gán cho biến được. Với ES6, bạn tạo các biến từ các thuộc tính của Object một cách nhanh chóng và hiệu quả hơn nhiều. Bạn có thể xem và tự cảm nhận cách viết của ES6 nó có hay hơn không nhé.

```
//ES5
const route = {
  title: 'Trang chủ',
  path: '/home'
};

// truy xuất lấy thông tin từ Object và chỉ truy xuất từng thuộc tính một.
let title = route.title;
let path = route.path;

console.log(title); // "Trang chủ"
console.log(path); // '/home'
```

Còn với ES6 thì có thể viết như này:

```
//ES6
const route = {
  title: 'Trang chủ',
  path: '/home'
};

// Khai báo biến hàng loạt.
const { title, path } = route;

console.log(title); // "Trang chủ"
console.log(path); // '/home'
console.log(component); // undefined
```

Array Destructuring

Tương tự với Object destructuring, bạn có thể nhanh chóng lấy các giá trị trong mảng và gán cho biến riêng lẻ. Chỉ khác là cú pháp dùng ký tự mảng thay vì object và destructuring thực hiện trên các vị trí trong mảng thay vì các thuộc tính được đặt tên trong object.

Ví dụ:

```
//ES6
let colors = ['red', 'green', 'blue'];
let [firstColor, secondColor] = colors;

console.log(firstColor); // "red"
console.log(secondColor); // "green"
```

Ngoài ra, bạn cũng thể bỏ qua một số vị trí trong mảng, lúc này, bạn không cần phải đặt tên biến. Như ví dụ trên, mình chỉ muốn gán biến cho vị trí thứ 3 của mảng thì làm như sau:

```
//ES6
let colors = ['red', 'green', 'blue'];
```

```
let [, , thirdColor] = colors;
```

```
console.log(thirdColor); // "blue"
```

Rất thú vị phải không? Phần tiếp theo còn thú vị hơn nữa cơ. Nhưng cứ bình tĩnh, nhâm nhi cốc cafe rồi học tiếp nhé.

Spread operator (...)

Ngày mình mới tiếp cận với Javascript gặp ngay quả đọc mã nguồn của đồng nghiệp thấy toàn dấu (...) mà thật sự thấy choáng ~_~. Lúc đó, mình cứ nghĩ đây là một cú pháp thần thánh với quyền năng vô hạn, chỗ nào cũng thấy dùng, công dụng thì thật “vi diệu”.

Spread (hay chính xác là dấu ba chấm ...) là một cú pháp mới, lần đầu ra mắt trong phiên bản ES6. Spread operator là một cách hữu dụng và ngắn gọn để bạn thao tác với mảng (như thêm phần tử vào mảng, ghép nối/kết hợp các mảng...) hoặc truyền tham số cho hàm.v.v...

Dưới đây là một số cách dùng phổ biến của spread operator.

Ghép nối/kết hợp nhiều mảng

```
//ES5
const users = ['Nguyễn Văn A', 'Nguyễn Văn B', 'Nguyễn Văn C']
const extendusers = ['Nguyễn Văn D', 'Nguyễn Văn E']
console.log([users, extendusers])
//Kết quả: [['Nguyễn Văn A', 'Nguyễn Văn B', 'Nguyễn Văn C'], ['Nguyễn Văn D', 'Nguyễn Văn E']]
```

```
//ES6
const users = ['Nguyễn Văn A', 'Nguyễn Văn B', 'Nguyễn Văn C']
const moreUsers = [...users, 'Nguyễn Văn D', 'Nguyễn Văn E']

console.log(moreUsers)
//Kết quả: ["Nguyễn Văn A" ,"Nguyễn Văn B" ,"Nguyễn Văn C" ,"Nguyễn Văn D" ,"Nguyễn Văn E"]
```

Bằng cách sử dụng spread operator, chúng ta đã kết hợp hai mảng lại với nhau được một mảng “phẳng”

Kết hợp 2 hay nhiều Object với nhau

```
//ES6
let objectOne = {
  name: "VNTALKING",
  age: 4
}
let objectTwo = {
  createDate: "16/12/2017",
  founder: "Dương Anh Sơn"
}

//Kết hợp hai objectOne và objectTwo, đồng thời thêm cả thuộc tính mới
let merge = {...objectOne, ...objectTwo, status: "Đang hoạt động"}
console.log(merge)
// Kết quả: merge = {name: "VNTALKING" ,age: 4 ,createDate: "16/12/2017" ,
founder: "Dương Anh Sơn", status: "Đang hoạt động"}
```

Làm đối số cho function

Bạn có thể sử dụng Spread để lấy các phần tử của mảng làm đối số cho một function. Từ phiên bản ES5 trở về trước, để làm được điều này, bạn cần phải sử dụng *apply()* để unpack các giá trị trong mảng.

```
// ES5
function multiply(a, b) {
  return a * b
}
const numbers = [5, 6]
let result = multiply.apply(null, numbers)
console.log(result)
//Kết quả: 30
```

```
// ES6
function multiply(a, b) {
```

```

    return a * b
}
const numbers = [5, 6]

let result = multiply(...numbers)
console.log(result)
//Kết quả: 30

```

Rest Parameters

Trước đây, bạn có thể truyền bao nhiêu đối số tùy ý vào một function cũng được, nhưng chỉ những giá trị đối số tương ứng mới được sử dụng. Bây giờ, bạn có thể sử dụng spread để đặt một lượng giá trị đối số không xác định đó vào một mảng.

Để dễ hình dung, chúng ta sẽ cùng nhau xem xét ví dụ sau: Chúng ta định nghĩa một hàm *sum()* chỉ có 2 tham số a và b. Nhưng khi gọi hàm này, chúng ta lại truyền rất nhiều giá trị vào. Mà những giá trị truyền vào này thì lúc định nghĩa function, bạn lại không biết được, không thể biết là lúc gọi hàm, người ta sẽ truyền vào bao nhiêu tham số.

```

function display(a, b) {
    console.log(a) // 4
    console.log(b) // 8
}
display(4, 8, 15, 16, 17, 18)
// Kết quả: 4, 8

```

Bằng cách sử dụng spread, bạn có thể giải quyết được khó khăn khi không biết người dùng truyền vào bao nhiêu tham số.

```

function display(a, b, ...theRest) {
    console.log(a) // 4
    console.log(b) // 8
    console.log(theRest)
}
display(4, 8, 15, 16, 17, 18)
// Kết quả: 4, 8, 15, 16, 17, 18

```



```
display(4, 8, 15, 16)
// Kết quả: 4, 8, 15, 16
```

Trên đây là 4 cách sử dụng spread phổ biến nhất mà bạn rất hay gặp và dùng trong các dự án thực tế sau này.

Modules

Module được hiểu đơn giản là cách chúng ta đóng gói mã nguồn thành một bộ phận chạy độc lập trong dự án. Các biến, hàm được khai báo trong module sẽ chỉ có tác dụng trong phạm vi module đó thôi và không thể gọi ra bên ngoài. Ngược lại, các thành phần bên ngoài module không thể truy cập vào module, trừ những phần chỉ định được phép truy cập.

Một cách hình tượng hóa dễ hình nhất đó là module tương tự với một chương trong cuốn sách vậy. Mỗi chương trong sách sẽ đề cập đến một nội dung cụ thể, có phần độc lập với chương khác, khi mà bạn chỉ đọc một chương đó thôi cũng có thể hiểu được chương đó mà không nhất thiết phải đọc các chương còn lại.

Từ phiên bản ES6, bạn có thể sử dụng module để *import*, *export* các biến, hàm, và class giữa các file javascript.

Các module được thiết kế tốt là module có tính tự đóng gói cao với những tính năng riêng biệt, cho phép chúng có thể bị xáo trộn, xóa bỏ, hay thêm vào nếu cần thiết mà không làm hỏng hệ thống.

Export

Bạn có thể sử dụng từ khóa *export* để xuất bản một đoạn mã (có thể là hàm, biến...) ra bên ngoài có thể sử dụng.

Trong trường hợp đơn giản, bạn *export* các hàm, biến để các module khác có thể sử dụng và gọi chúng.

Ví dụ:

```
// Lưu đoạn mã này thành example.js
// export data
export var color = "red";
export let name = "Nicholas";
export const magicNumber = 7;

// export function
export function sum(num1, num2) {
  return num1 + num2;
}
// export class
export class Rectangle {
  constructor(length, width) {
    this.length = length;
    this.width = width;
  }
}
// function này là private, các module khác bên ngoài không truy cập được
function subtract(num1, num2) {
  return num1 - num2;
}

// Định nghĩa một function...
function multiply(num1, num2) {
  return num1 * num2;
}
// ... sau đó export cũng được.

export multiply;
```

Import

Ngược với *export* là *import*. Tức là để sử dụng đoạn mã được export từ một module, bạn cần phải import module vào.

Cú pháp chung:

```
import { identifier1, identifier2 } from "./example.js";
```

Giả sử, đoạn mã trong ví dụ ở phần export, bạn lưu thành file *example.js*. Và để sử dụng module này, chúng ta tiến hành import chúng vào mã nguồn:

Ví dụ: chúng ta chỉ import một hàm duy nhất *sum()* để sử dụng:

```
// import
import { sum } from "./example.js";
console.log(sum(1, 2)); // Kết quả: 3
```

Trong phần này, chúng ta đã cùng nhau tìm hiểu một loạt những tính năng, cú pháp được nâng cấp mới từ ES5 lên ES6 mà được sử dụng nhiều nhất, từ String, function, đến module.v.v... Mục đích của phần này đơn giản là để bạn đọc hiểu và viết được cú pháp của ES6 một cách nhanh nhất.

Trong phiên bản ES6 còn rất nhiều cải tiến nữa, thật sự là rất nhiều. Tuy nhiên, bạn không cần phải dành toàn bộ tâm trí để tìm hiểu hết, cứ làm đến đâu tìm hiểu đến đó, vừa thực hành vừa học mới là cách khiến bạn nhớ lâu nhất.

PHẦN 9

JAVASCRIPT FRAMEWORK

Như bạn biết, Javascript chỉ là ngôn ngữ lập trình, việc hiểu và vận dụng tốt các kiến thức về Javascript là nền tảng để bạn bước vào các dự án thực tế.

Thông thường, các dự án thực tế ít ai xây dựng một ứng dụng mà chỉ sử dụng thuần Javascript, bởi vì nó sẽ tốn rất nhiều công sức và thời gian. Thay vì đó, họ sẽ sử dụng các Javascript framework, điều này giúp họ tăng tốc độ phát triển dự án, nhanh chóng đưa sản phẩm ra thị trường.

Tất nhiên, để bạn có thể học và làm chủ một Javascript framework bất kỳ, trước hết bạn phải nắm vững kiến thức nền tảng ngôn ngữ lập trình Javascript, đó cũng là lý do tại sao phần này mình để ở cuối cuốn sách này.

Phần này, mình sẽ giới thiệu một số Javascript framework nổi bật nhất mà bạn sẽ thường xuyên được nghe nói hoặc có nhiều cơ hội làm việc trong các dự án thực tế trong tương lai.

1. jQuery

Mặc dù, jQuery theo định nghĩa thì không phải là một javascript framework đúng nghĩa. Nhưng vì mức độ phổ biến của nó mà mình vẫn xếp jQuery vào phần này của cuốn sách.

Các thư viện jQuery được sử dụng bằng cách bạn import các thư viện vào phần mã nguồn nào của ứng dụng cần sử dụng tới.

Ví dụ: Nếu bạn muốn sử dụng một jQuery template cho tính năng tự động điền vào form. Bạn chỉ cần import thư viện jQuery và gọi mã thích hợp, mã jQuery sẽ truy xuất tính năng trong jQuery và hiển thị kết quả trên trình duyệt của người dùng.

Có một đặc trưng mà bạn hay gặp khi làm việc với thư viện jQuery đó là ký tự \$. Để gọi bất kỳ tính năng nào trong thư viện jQuery, bạn đều phải sử dụng tới ký tự \$.

Ví dụ:

```
$(".pWithClass").css("color", "blue"); // colors the text blue
```

Link trang chủ: <https://jquery.com/>

2. ReactJS

Tương tự jQuery, mặc dù về mặt kỹ thuật React là một thư viện Javascript, nhưng React lại luôn đứng đầu trong bảng xếp hạng các javascript framework phổ biến nhất trong những năm gần đây.

Được sự hẫu thuẫn bởi gã khổng lồ Facebook, cùng với cộng đồng người dùng đông đảo, React luôn là giải pháp được ưu tiên lựa chọn cho các dự án quy mô từ nhỏ tới lớn.

Link trang chủ: <https://reactjs.org>

3. VueJS

Vue (đọc là “vếu” ☺, đùa đấy! thực ra từ này mượn từ tiếng Pháp, phát âm và có nghĩa tương đương với từ View trong tiếng Anh) là một Javascript framework được phát hành năm 2014 bởi kỹ sư kì cựu Evan You (anh từng làm việc cho Google).

Vue đang ngày càng trở nên phổ biến, nó kết hợp thế mạnh giữa Angular và React, đặc biệt là với người mới chuyển sang học và làm việc với Vue, rất dễ học và làm chủ framework này.

Cá nhân mình thấy Vue hay ở chỗ là cú pháp dựa trên HTML, cho phép developer viết các trang ở định dạng HTML thay vì phải học một ngôn ngữ mới (trường hợp React bạn phải làm quen với JSX).

Nếu cho mình được chọn JS framework cho dự án, mình sẽ ưu tiên chọn Vue.

Link trang chủ: <https://vuejs.org>

4. Angular

Cuối cùng, Angular là phiên bản viết lại của AngularJS, được Google chống lưng. Trong số các JS framework thì Angular có vẻ khó học nhất vì cú pháp và cấu trúc của nó hơi rắc rối. Angular sử dụng Typescript thay vì Javascript thuần, do đó bạn cũng sẽ phải học làm quen với Typescript trước khi bắt tay vào tìm hiểu Angular. Tuy nhiên, các dự án và nhu cầu tuyển dụng kỹ sư Angular lại khá lớn, lương cũng rất cao, do đó bạn cũng nên cân nhắc nếu muốn nâng cao con đường sự nghiệp sau này.

Link trang chủ: <https://angular.io>

Trên đây là những framework và thư viện Javascript phổ biến nhất, được sử dụng trong rất nhiều dự án lớn nhỏ. Nếu bạn quyết định lựa chọn con đường làm front-end developer thì đây là những kỹ năng gần như bắt buộc phải có trong tờ CV. Phần này của cuốn sách, mình sẽ không thể trình bày chi tiết từng framework được, chúng ta sẽ gặp nhau trong một cuốn sách chi tiết về từng framework nhé.



Hiện tại, VNTALKING đã xuất bản một cuốn sách về ReactJS, bạn có thể tham khảo trong mục bên dưới, mình sẽ giới thiệu cuốn sách này.

PHẦN 10

BÀI TẬP

Xin chúc mừng (^_^) vì bạn đã đến được tận đây. Chúc mừng vì sự kiên trì và nỗ lực của bạn.

Sau khi xong phần lý thuyết, để có thể nắm chắc và nhớ lâu thì không thể thiếu phần thực hành. Bạn làm càng nhiều, va chạm với nhiều loại yêu cầu, càng giúp bạn nhanh nhạy và nhớ được lý thuyết lâu hơn, giống như chiến binh càng dạn dày trận mạc sẽ càng trở lên tinh nhuệ hơn.

Phần này, chúng ta sẽ cùng nhau làm bài tập để kiểm tra xem bạn đã nắm được kiến thức đến đâu, cũng như rèn luyện kỹ năng viết code của mình.

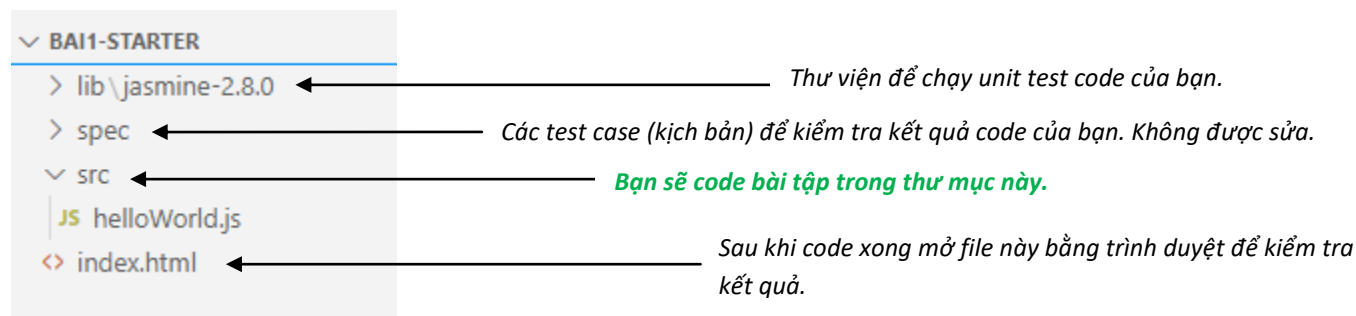
Các bài tập sẽ được sắp xếp từ dễ đến khó. Các bạn cố gắng tự thực hiện sau đó mới tham khảo đáp án. Nên nhớ, đáp án chỉ mang tính chất tham khảo, bạn làm theo cách nào cũng được, miễn là chương trình chạy đúng kết quả là được.

1. Hướng dẫn làm bài tập và kiểm tra kết quả.

Với mục đích để bạn tập trung hoàn toàn vào xử lý logic của bài toán thay vì phải mất thời gian vào cấu hình dự án, nên mỗi bài tập, mình sẽ chia ra làm 2 project:

- Starter: Dự án này có sẵn bộ khung, bạn chỉ việc code phần xử lý logic của bài toán
- Solution: Đáp án gợi ý của tác giả.

Cấu trúc dự án của mỗi bài tập như sau:



Để tiến hành code, bạn mở thư mục `src` và mở các tệp trong đó, mình có để sẵn các hàm, bạn chỉ cần code vào thân của hàm đó, không nên đổi tên các hàm mà mình đã để sẵn, mục đích để sau còn chạy unit test kiểm tra code của bạn đã chạy đúng chưa.

Để minh họa, mình sẽ lấy bài 1 làm hướng dẫn chi tiết các bước nhé.

Bài 1: Hello World

Yêu cầu: Viết hàm `sayHello()` trả về chuỗi "Hello".

Đầu tiên: Bạn tải project starter về: <https://github.com/vntalking/Book-Javascript/tree/master/baitap/Bai1-starter>

Bạn mở `helloWorld.js` trong thư mục `src`.

```
HelloWorld = function() {}

HelloWorld.prototype.sayHello = function(){
    // Bạn viết code vào đây và return ra kết quả.
    return '';
}
```

Bạn tiến hành code vào bên trong hàm `sayHello()`, return kết quả như kết quả mong muốn.

```
HelloWorld = function() {}

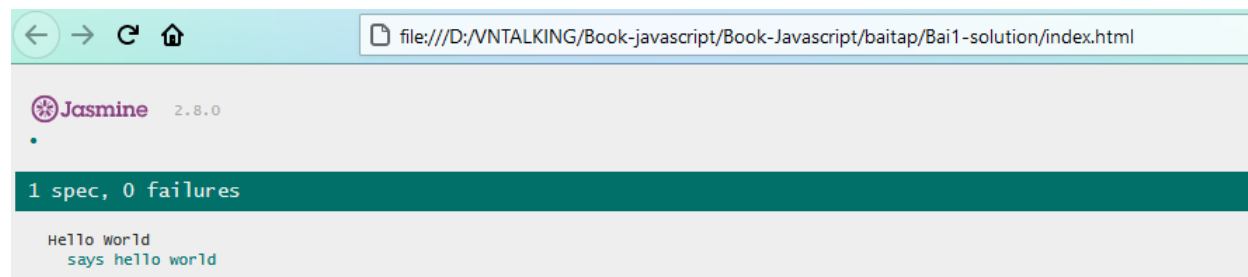
HelloWorld.prototype.sayHello = function(){
    return 'Hello';
}
```


Bạn có thể xem trước kịch bản unit test để biết chương trình sẽ kiểm tra những điều kiện nào, bằng cách mở tệp `***.spec.js` trong thư mục `spec` nhé.

```
describe('Hello World', function() {  
  var talking =new HelloWorld();  
  
  it('says hello world', function() {  
    expect(talking.sayHello()).toEqual('Hello');  
  });  
});
```

Như kịch bản trên, nó yêu cầu hàm `sayHello()` phải trả về một chuỗi “Hello”. Nếu đúng thì cho pass, còn không là failed.

Sau khi bạn code xong, mở file `index.html` để kiểm tra kết quả. Nếu code của bạn pass qua hết các unit test là OK. Unit test nào pass sẽ có màu xanh, còn failed sẽ có màu đỏ. Như hình bên dưới là đã pass.

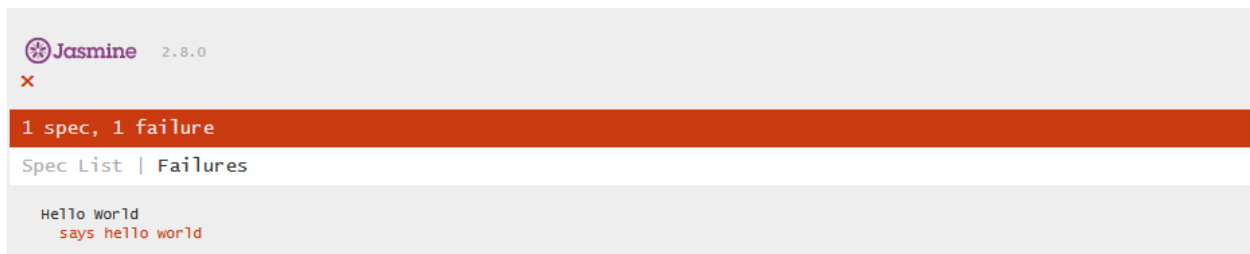


Hình 10.1: Minh họa bài tập đã pass

Còn nếu như thế này thì là failed nhé.



Hình 10.2: Minh họa bài tập chưa pass, có 1 test case bị failed



Hình 10.3: Danh sách các test case bị failed

Cuối cùng là project solution để tham khảo đáp án:

<https://github.com/vntalking/Book-Javascript/tree/master/baitap/Bai1-solution>

Ok! Bạn đã hiểu cách làm bài tập rồi đúng không? Chúng ta tiến hành thực hành nhé.

Bài 2: Calculator

Yêu cầu: Mục tiêu của bài này là bạn tạo ra một máy tính cá nhân, có thể thực hiện được các phép tính: cộng, trừ nhân, tính tổng của nhiều số, tính lũy thừa, tính giai thừa.

Các bạn tải dự án starter về để tiến hành code nhé.

Starter project: <https://github.com/vntalking/Book-Javascript/tree/master/baitap/Bai2-starter>

Bạn tiến hành code vào tệp *Calculator.js*

```
let Calculator = function() {}

/**
 * Hàm thực hiện phép tính cộng.
 * Ví dụ nhập (15, 10). Tính 15 + 10 = 25. (return 25)
 */
Calculator.prototype.add = function (a, b) {

}

/**
 * Hàm thực hiện phép tính tổng của nhiều số.
 * Đầu vào là một mảng các chữ số.
```

```

* Ví dụ: [1, 3, 6, 10]. Kết quả mong muốn là 1 + 3 + 6 + 10 = 20. (return 20)
*/
Calculator.prototype.sum = function (array) {

}

/**
 * Hàm thực hiện phép tính trừ.
 * ví dụ: nhập (15, 10), tính 15 - 10 = 5.
 */
Calculator.prototype.subtract = function (a, b) {

}

/**
 * Hàm thực hiện phép tính nhân của nhiều số. Đầu vào là một mảng
 * Ví dụ: đầu vào là [2, 3, 4]. Tính: 2 * 3 * 4 = 24.
 */
Calculator.prototype.multiply = function (array) {

}

/**
 * Hàm thực hiện phép tính lũy thừa.
 * Ví dụ: nhập vào (2,3), tính 2^3 = 2*2*2 = 8
 */
Calculator.prototype.power = function(a, b) {

}

/**
 * Hàm thực hiện phép tính giai thừa.
 * Ví dụ: nhập vào n = 3, tính 3! = 1 * 2 * 3 = 6
 */
Calculator.prototype.factorial = function(n) {

}

```

Đáp án:

<https://github.com/vntalking/Book-Javascript/tree/master/baitap/Bai2-solution>

Bài 3: Dãy Fibonacci

Yêu cầu: Dãy Fibonacci là dãy vô hạn các số tự nhiên bắt đầu bằng 1 và 1. Sau đó các số tiếp theo sẽ bằng tổng của 2 số liền trước nó. Cụ thể, các số đầu tiên của dãy Fibonacci là 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610... Yêu cầu là viết một hàm trả về giá trị của một số fibonacci cụ thể. Ví dụ, mình truyền vào là thứ tự của số đó, trả về giá trị của nó, fibonacci(4) trả về 3, fibonacci(6) trả về 8.

Các bạn tải dự án starter về để tiến hành code nhé.

Starter project: <https://github.com/vntalking/Book-Javascript/tree/master/baitap/Bai3-starter>

Bạn tiến hành code vào tệp *Fibonacci.js*

```
const Fibonacci = function () {};  
  
/**  
 * Viết một hàm mà truyền vào thứ tự và trả về giá trị của số đó trong dãy số fibonacci.  
 * Ví dụ: Fibonacci.get(4) // Trả về giá trị của số thứ 4 trong dãy fibonacci (1, 1, 2, 3). Trong ví dụ này trả về số 3.  
 */  
Fibonacci.prototype.get = function(count) {  
    // code here  
}
```

Đáp án:

<https://github.com/vntalking/Book-Javascript/tree/master/baitap/Bai3-solution>

Bài 4: Kiểm tra năm nhuận

Yêu cầu: Viết một hàm để kiểm tra một năm bất kỳ có phải là năm nhuận hay không?

Quy tắc để tính năm nhuận: năm nhuận là năm chia hết cho 4 (ví dụ: 1984 hay 2004). Tuy nhiên, các năm chia hết cho 100 thì lại không phải là năm nhuận (ví dụ: 1800 hay 1900) trừ khi chúng cũng chia hết cho 400 (ví dụ: 1600 hay 2000).

Các bạn tải dự án starter về để tiến hành code nhé.

Starter project: <https://github.com/vntalking/Book-Javascript/tree/master/baitap/Bai4-starter>

Bạn tiến hành code vào tệp *leapYears.js*

```
const leapYear = function () {};  
  
/**  
 * Viết hàm để xác định một năm bất kỳ có phải là năm nhuận hay không? Trả về giá  
 * trị boolean.  
 * Ví dụ: isLeapYears(2000) = true  
 * isLeapYears(2000) = false  
 */  
leapYear.prototype.isLeapYears = function(year) {  
    // code here  
}
```

Đáp án:

<https://github.com/vntalking/Book-Javascript/tree/master/baitap/Bai4-solution>

Bài 5: Xóa một phần tử khỏi mảng

Yêu cầu: Viết một hàm nhận vào 1 mảng và một số. Bạn cần phải tìm và xóa đối số đó khỏi mảng.

Ví dụ: *removeFromArray([1, 2, 3, 4], 3);* // Xóa phần tử 3 và trả về mảng mới [1,2,4]

Các bạn tải dự án starter về để tiến hành code nhé.

Starter project: <https://github.com/vntalking/Book-Javascript/tree/master/baitap/Bai5-starter>

Bạn tiến hành code vào tệp *removeFromArray.js*

```
var ArrayUtils = function () {};  
  
/**  
 * Viết một hàm nhận vào 1 mảng và một số. Bạn cần phải tìm và xóa đối số đó khỏi  
 * mảng.  
 * Ví dụ: ArrayUtils.removeFromArray([1, 2, 3, 4], 3) = [1, 2, 4]  
 */  
ArrayUtils.prototype.removeFromArray = function(...args) {  
    // code here  
}
```

Đáp án:

<https://github.com/vntalking/Book-Javascript/tree/master/baitap/Bai5-solution>

Trên đây là 5 bài tập để bạn thực hành với Javascript. Tất nhiên, luyện tập không bao giờ là đủ cả. Mình hi vọng, sau khi làm xong 5 bài tập này, bạn vẫn tiếp tục luyện tập (có thể là làm bài tập hoặc tham gia dự án) để hiểu và làm chủ ngôn ngữ lập trình Javascript tuyệt vời này.

Xin chúc mừng!

Xin chúc mừng bạn đã hoàn thành nội dung cuốn sách!

Khi đọc đến dòng chữ này, mình đánh giá rất cao sự nỗ lực và kiên trì của bạn. Với những kiến thức nền tảng về Javascript, bạn sẽ có rất nhiều cơ hội cho sự nghiệp lập trình trong tương lai. Hãy luôn giữ vững tinh thần học tập tuyệt vời như thế này nhé.

Thay mặt các bạn trong đội ngũ VNTALKING, xin chúc bạn mọi điều tốt đẹp nhất trong hành trình trở thành một lập trình viên chuyên nghiệp.

Hi vọng bạn sẽ thích cuốn sách này, và muốn tìm hiểu thêm về thế giới lập trình.

Trong quá trình biên soạn cuốn sách sẽ không tránh khỏi những thiếu sót.

Mình rất mong nhận được phản hồi từ bạn, hãy gửi email tới

support@vntalking.com.

Cuốn sách là một phần trong dự án học lập trình của VNTALKING. Mong bạn ủng hộ website hướng dẫn học lập trình tại: <https://vntalking.com>

Hẹn gặp lại ở những cuốn sách sau.

VNTALKING!

PHỤ LỤC

KẾT NỐI VỚI VNTALKING

Một lần nữa, VNTALKING đánh giá rất cao sự nỗ lực của bạn, chứng tỏ bởi việc bạn đã đọc hết cuốn sách và đọc đến tận trang sách này. Cảm ơn bạn rất nhiều ^_^

Đặc biệt, VNTALKING cũng rất vui khi được đồng hành cùng bạn trên con đường học để trở thành một lập trình viên chuyên nghiệp nói chung và Javascript nói riêng.

Vì vậy, bất cứ khi nào bạn cần tư vấn, có thắc mắc hay khó khăn hãy "trút bầu tâm sự" với VNTALKING.

Liên hệ với VNTALKING bằng bất kỳ hình thức nào dưới đây.

- Website: <https://vntalking.com>
- Fanpage: <https://facebook.com/vntalking>
- Group: <https://www.facebook.com/groups/hoidaplaptrinh.vntalking>
- Email: support@vntalking.com

PHỤ LỤC

THÔNG TIN TÁC GIẢ

VNTALKING.COM là một website được thành lập từ 25/12/2016 và đang được vận hành bởi nhóm Dương Anh Sơn (một developer “kì cựu” – chuẩn bị về quê chăn lợn).

Bọn mình luôn hướng tới một trải nghiệm miễn phí mà hiệu quả. VNTALKING gồm những thành viên luôn muốn đem đến cho độc giả những kiến thức, kinh nghiệm thực tiễn, được cập nhật nhanh nhất. Đồng hành cùng VNTALKING để khám phá niềm đam mê lập trình trong bạn.

Thông tin thêm về tác giả.



Tên đầy đủ là Dương Anh Sơn, gọi tắt là Sơn Dương (^_^). Tốt nghiệp ĐH Bách Khoa Hà Nội. Mình bắt đầu nghiệp coder khi ra trường không xin được việc đúng chuyên ngành. Mình tin rằng chỉ có chia sẻ kiến thức mới là cách học tập nhanh nhất.

PHỤ LỤC

CUỐN SÁCH CỦA VNTALKING

Đến thời điểm hiện tại, VNTALKING đã hoàn thành 3 dự án sách học lập trình. Sách học Javascript này là cuốn sách thứ 3 mà VNTALKING thực hiện.

Nếu bạn muốn tìm hiểu nhiều hơn về back-end hoặc front-end, mời bạn tham khảo cuốn sách:



Lập trình Node.JS thật đơn giản

Đọc sách

<https://vntalking.com/sach-hoc-lap-trinh-node-js-that-don-gian-html>



Lập trình React thật đơn giản

Đọc sách

<https://vntalking.com/tai-lieu-hoc-reactjs-tieng-viet>

Hi vọng trong thời gian tới, VNTALKING sẽ tiếp tục nhận được sự ủng hộ của độc giả. Thành công của bạn chính là động lực để VNTALKING cho ra nhiều cuốn sách với chất lượng tốt hơn nữa, đáp ứng nhu cầu học lập trình của mọi người.