## 1. JavaScript là gì?

*JavaScript*là ngôn ngữ lập trình Web và phổ biến nhất thế giới.

*JavaScript* ban đầu được tạo ra để "làm cho các trang Web trở nên sống động".

Các chương trình trong ngôn ngữ này được gọi là *kịch bản (scripts)*. Chúng có thể được viết ngay trong HTML của trang web và chạy khi tải trang.

Các tập lệnh (scripts) được cung cấp và thực thi dưới dạng văn bản thuần túy. Chúng không cần chuẩn bị hay biên dịch để có thể chạy được.

Ở khía cạnh này, JavaScript rất khác với ngôn ngữ lập trình có tên là [Java](https://vi.wikipedia.org/wiki/Java_(ng%C3%B4n_ng%E1%BB%AF_l%E1%BA%ADp_tr%C3%ACnh)). Khi JavaScript được tạo ra, ban đầu nó có một cái tên khác là "LiveScript". Nhưng lúc đó, Java đang rất phổ biến, vì vậy người ta quyết định rằng đặt một ngôn ngữ mới là "em trai" của Java sẽ có ích. Nhưng với sự phát triển của mình, JavaScript đã trở thành một ngôn ngữ hoàn toàn độc lập với những đặc điểm riêng của mình được gọi là [ECMAScript](https://vi.wikipedia.org/wiki/ECMAScript), và bây giờ nó không còn liên quan đến Java.

Ngày nay, JavaScript không những có thể thực thi trên trình duyệt, mà còn trên máy chủ server, hoặc trên bất cứ nơi nào có chương trình tên là [the JavaScript engine](https://en.wikipedia.org/wiki/JavaScript_engine).

Trình duyệt đã có sẵn một JavaScript engine đôi khi được gọi là "JavaScript virtual machine".

Những engine khác nhau thì sẽ có những "tên mã" khác nhau. Chẳng hạn:

* [V8](https://vi.wikipedia.org/wiki/Chrome_V8) -- trong Chorme và Opera
* [SpiderMonkey](https://en.wikipedia.org/wiki/SpiderMonkey) -- trong Firefox
* Có một số tên mã khác nhau như "Chakra" cho IE, "ChakraCore" cho Microsoft Edge, "Nitro" và "SquirrelFish" cho Safari v.v.

Nên ghi nhớ các thuật ngữ trên bởi vì chúng được sử dụng khá nhiều trong các bài viết dành cho nhà phát triển (developer) trên Internet, và cả chúng ta. Ví dụ, nếu "tính năng X được hỗ trợ bởi V8", vậy nó có lẽ sẽ hoạt động trên Chrome và Opera.

Engine khá phức tạp, tuy nhiên có thể hiểu đơn giản như sau:

1. Engine (được nhúng nếu là trình duyệt) đọc ("phân tích cú pháp") tập lệnh.
2. Tiếp theo nó chuyển đổi ("biên dịch") tập lệnh sang mã máy.
3. Và sau đó mã máy chạy, khá nhanh

Engine áp dụng tối ưu hóa ở mỗi bước của quá trình. Nó thậm chí còn theo dõi tập lệnh đã biên dịch khi nó chạy, phân tích dữ liệu chạy qua nó và tối ưu hóa hơn nũa mã máy dựa trên kiến thức đó.

## 2. JavaScript có thể làm gì trong trình duyệt?

Javascript hiện đại là một ngôn ngữ lập trình "an toàn". Nó không cung cấp quyền truy cập cấp thấp vào bộ nhớ hay CPU, bởi vì ban đầu nó được tạo ra cho trình duyệt vốn dĩ không yêu cầu những điều đó.

Sức mạnh của Javascript phụ thuộc rất lớn vào môi trường mà nó đang hoạt động. Chẳng hạn, [Node.js](https://vi.wikipedia.org/wiki/Node.js) hỗ trợ các hàm giúp cho Javascript có thể đọc/ghi các tập tin tùy ý, thực hiện các yêu cầu mạng, etc.

Javascript trong trình duyệt có thể làm mọi thứ có liên quan đến thao tác của trang web, tương tác với người dùng, và máy chủ web (webserver).

Ví dụ, JavaScript trong trình duyệt có khả năng:

* Thêm HTML mới vào trang, thay đổi nội dung có sẵn, sửa đổi phong cách (styles).
* Phản ứng với các hành động của người dùng, chạy khi nhấp chuột, chuyển động của con trỏ, nhấn phím.
* Gửi yêu cầu qua mạng đến các máy chủ từ xa, tải xuống và tải lên file (cái gọi là [AJAX](https://vi.wikipedia.org/wiki/Ajax_(l%E1%BA%ADp_tr%C3%ACnh)) và công nghệ [COMET](https://en.wikipedia.org/wiki/Comet_(programming))).
* Lấy và đặt cookie, đặt câu hỏi cho người truy cập, hiện tin nhắn.
* Ghi nhớ dữ liệu ở phía máy khách ("lưu trữ cục bộ - local storage").

## 3. JavaScript không thể làm gì trong trình duyệt?

Nhiều tính năng của Javascript trong trình duyệt bị giới hạn vì lợi ích an toàn của người dùng. Mục đích là để ngăn chặn những trang web độc hại truy cập thông tin cá nhân hoặc gây hại đến dữ liệu của người dùng.

Một số hạn chế có thể kể là:

* JavaScript trên một trang web không thể đọc/ghi các tệp tùy ý trên đĩa cứng, sao chép chúng hoặc thực thi các chương trình. Nó không có quyền truy cập trực tiếp vào các chức năng của hệ điều hành.

Nhiều trình duyệt hiện đại cho phép làm việc với tập tin, nhưng bị giới hạn và chỉ được truy cập nếu như người dùng thực hiện một hành động nhất định nào đó, ví dụ như "thả" các tập tin vào cửa sổ trình duyệt hoặc chọn chúng qua thẻ <input>.

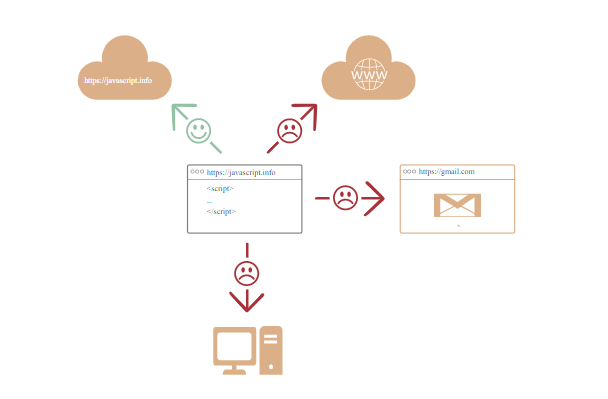
Có nhiều cách để tương tác với camera/microphone và thiết bị khác, nhưng chúng yêu cầu sự cho phép rõ ràng của người dùng. Vì vậy, một trang web hỗ trợ Javascript sẽ không bật lén camera, quan sát và gửi thông tin cho [NSA](https://vi.wikipedia.org/wiki/C%C6%A1_quan_An_ninh_Qu%E1%BB%91c_gia_(Hoa_K%E1%BB%B3)).

* Các tab/cửa sổ nhìn chung không biết gì về nhau. Thỉnh thoảng có, ví dụ như một cửa sổ dùng Javascript để mở cửa sổ khác. Nhưng kể cả như vậy, JavaScript từ trang này vẫn không thể can thiệp vào trang kia nếu như chúng đến từ tên miền, giao thức hoặc port khác.

Cái này được gọi là "Same Origin Policy". Để làm việc với nó, cả 2 trang web cần phải đồng ý cho việc trao đổi dữ liệu và bao gồm cả các đoạn mã Javascript đặc biệt nhằm xử lí nó. Chúng ta sẽ nhắc đến trong phần hướng dẫn sau.

Hạn chế ở đây là, một lần nữa, cho an toàn của người dùng. Một trang web từ http://anysite.com mà người dùng vừa mở không được can thiệp vào trang web có URL là http://gmail.com ở tab khác với mục đích đánh cắp thông tin cá nhân.

* Javascript có thể dễ dàng giao tiếp qua mạng để đến máy chủ - nơi đang chứa trang web hiện tại. Nhưng khả năng nhận dữ liệu từ trang web khác hoàn toàn bị tê liệt. Mặc dù có thể, nó đòi hỏi phải có thỏa thuận rõ ràng (được thể hiện trong HTTP headers) từ phía điều khiển. Lại lần nữa, đây là giới hạn vì an toàn.



Những giới hạn trên sẽ không tồn tại nếu như Javascript được sử dụng bên ngoài trình duyệt, như máy chủ chẳng hạn. Các trình duyệt hiện đại cũng cho phép các plugin/tiện ích có thể hỏi cho các quyền mở rộng.

## 4. Điều gì khiến cho JavaScript khác biệt?

Có ít nhất *ba* điều tuyệt vời ở Javascript:

* Tích hợp hoàn toàn với HTML/CSS.
* Những điều đơn giản được thực hiện một cách đơn giản.
* Được hỗ trợ bởi tất cả các trình duyệt chính và được bật theo mặc định.

Javascript là công nghệ trình duyệt duy nhất kết hợp cả 3 điều trên.

Đó là những thứ khiến cho Javascript trở nên độc đáo. Đó là lí do tại sao nó là công cụ phổ biến nhất để tạo giao diện trình duyệt.

Điều đó nói rằng, JavaScript cũng cho phép tạo các ứng dụng di động, máy chủ v.v.

## 5. Những ngôn ngữ "trên nền tảng" JavaScript

Cú pháp của Javascript không phù hợp cho tất cả mọi người. Những người khác nhau lại muốn các tính năng khác nhau.

Đó là điều được mong đợi, vì các dự án và yêu cầu đều khác nhau đối với mọi người.

Vì vậy gần đây có rất nhiêu ngôn ngữ mới xuất hiện, chúng được *dịch mã* (chuyển đổi) sang Javascript trước khi chúng chạy trên trình duyệt.

Những công cụ hiện đại làm cho việc dịch trở nên nhanh chóng và minh bạch, thực sự cho phép các nhà phát triển viết mã bằng ngôn ngữ khác và tự động chuyển đổi nó trơn tru và hiệu quả.

Có thể kể đến một số ngôn ngữ:

* + [CoffeeScript](https://coffeescript.org/) là một "cú pháp đặc biệt" cho JavaScript. Nó giới thiệu cú pháp ngắn hơn, cho phép chúng ta viết mã rõ ràng và chính xác hơn. Thông thường thì các nhà phát triển Ruby thích nó.
  + [TypeScript](https://www.typescriptlang.org/) được tập trung cho việc "gõ dự liệu một cách nghiêm ngặt" nhằm đơn giản hóa việc phát triển và hỗ trợ các hệ thống phức tạp. Nó được phát triển bởi Microsoft.
  + [Flow](http://flow.org/) cũng thêm một vài cách gõ dữ liệu, nhưng theo một cách khác. Phát triển bởi Facebook.
  + [Dart](https://dart.dev/) là một ngôn ngữ độc lập có engine riêng và có thể chạy trong môi trường khác ngoài trình duyệt, nó cũng có thể được dịch sang Javascript. Phát triển bởi Google.
  + [Brython](https://brython.info/) là một trình chuyển tiếp Python sang JavaScript cho phép viết các ứng dụng bằng Python thuần túy mà không cần JavaScript.
  + [Kotlin](https://kotlinlang.org/docs/js-overview.html) là một ngôn ngữ lập trình hiện đại, ngắn gọn và an toàn, mà có thể nhắm đến trình duyệt hoặc Node.

Sẽ có nhiều hơn số kể trên. Dĩ nhiên, ngay cả khi chúng ta sử dụng một trong những ngôn ngữ đó, chúng ta cũng nên biết Javascript để thực sự hiểu chúng ta đang làm gì.

=> Tóm Tắt:

* Javascript được tạo ra với mục đích ban đầu chỉ cho trình duyệt, nhưng bây giờ đã được sử dụng rộng rãi trên nhiều môi trường khác.
* Ngày nay, Javascript có một vị trí khác biệt như là ngôn ngữ trình duyệt được sử dụng rộng rãi nhất với sự tích hợp đầy đủ với HTML/CSS.
* Có nhiều ngôn ngữ được "dịch" sang JavaScript và cung cấp một số tính năng nhất định. Nên xem qua chúng, ít nhất là một thời gian ngắn sau khi thành thạo JavaScript.

## I. First Step

## 1. Thẻ “Script”

Các chương trình JavaScript có thể được chèn vào hầu như bất kỳ đâu trong một tài liệu HTML bằng cách sử dụng thẻ <script>.

Ví dụ:

<!DOCTYPE HTML>

<html>

<body>

  <p>Đoạn trước script...</p>

  <script>

    alert( 'Hello World!' );

  </script>

  <p>...Đoạn sau script.</p>

</body>

</html>

Đoạn mã JavaScript bên trong thẻ <script> sẽ tự động chạy khi trình duyệt xử lý thẻ này.

## 2. Cách đánh dấu hiện đại

Thẻ <script> có vài thuộc tính ngày nay hiếm khi được sử dụng nhưng vẫn có thể gặp đâu đó trong các chương trình cũ:

Thuộc tính type: <script type=...> : Chuẩn HTML4 cũ yêu cầu mỗi script phải có một type. Thường thì đó là type="text/javascript". Hiện nay nó không còn cần thiết nữa. Ngoài ra, chuẩn HTML hiện đại đã thay đổi hoàn toàn ý nghĩa của thuộc tính này. Bây giờ, nó có thể được sử dụng cho các mô-đun JavaScript. Nhưng đó là một chủ đề nâng cao, chúng ta sẽ nói về các mô-đun trong một phần khác của hướng dẫn.

Thuộc tính language: <script language=...> : Thuộc tính này cho biết script được viết bằng ngôn ngữ nào. Ngày nay JavaScript trở thành ngôn ngữ script mặc định, thuộc tính này không còn cần thiết nữa.

Các chú thích trước và sau script. : Trong các sách và hướng dẫn rất cũ, bạn có thể gặp chú thích HTML bên trong thẻ <script> như thế này:

`html no-beautify

<script type="text/javascript"><!--

    ...

//--></script>

`

Thủ thuật này không được sử dụng trong JavaScript hiện đại. Những chú thích này ẩn mã JavaScript khỏi các trình duyệt cũ không biết cách xử lý thẻ <script>. Vì các trình duyệt được phát hành trong 15 năm qua không gặp vấn đề này, nên loại chú thích này có thể giúp bạn xác định mã thực sự cũ.

## 3. Các script ngoài

Nếu có nhiều mã JavaScript, ta có thể đặt chúng trong một tệp riêng.

Script đặt trong tệp này gọi là script ngoài và có thể đưa vào tài liệu HTML bằng thuộc tính src:

<script src="/path/to/script.js"></script>

Ở đây, /path/to/script.js là đường dẫn tuyệt đối tới tệp chứa script tính từ thư mục gốc của site. Bạn cũng có thể cung cấp đường dẫn tương đối so với trang web hiện tại. Ví dụ, src="script.js" chỉ đến tệp "script.js" trong thư mục hiện tại.

Chúng ta có thể cho một URL đầy đủ. Ví dụ:

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js"></script>

Để thêm nhiều script, sử dụng nhiều thẻ:

<script src="/js/script1.js"></script>

<script src="/js/script2.js"></script>

…

Như một quy tắc, chỉ những script đơn giản nhất mới nên đặt trực tiếp trong tài liệu HTML. Những script phức tạp nên đặt trong các tệp riêng.

Ưu điểm khi đặt script trong tệp riêng là trình duyệt sẽ tải chúng về và lưu lại trong [cache (bộ nhớ đệm)](<https://vi.wikipedia.org/wiki/Cache_(tin_h%E1%BB%8Dc)>) của nó.

Sau đó nếu có một trang khác cũng dùng script này, trình duyệt lấy nó từ bộ nhớ đệm mà không cần tải lại.

Điều này giúp giảm thiểu băng thông và tăng tốc độ tải trang.

warn header="Nếu src được dùng, nội dung của thẻ bị bỏ qua." Một thẻ `<script>` script không thể vừa có thuộc tính `src` vừa có script bên trong.

Sẽ không làm việc:

<script \*!\*src\*/!\*="file.js">

  alert(1); // script bị bỏ qua vì thuộc tính src được dùng

</script>

 Bạn buộc phải chọn hoặc sử dụng script ngoài <script src="…"> hoặc thẻ <script> với script đặt trong.

Ví dụ trên có thể chia thành hai thẻ <script> để làm việc:

<script src="file.js"></script>

<script>

  alert(1);

</script>

Tóm tắt:

- Chúng ta có thể sử dụng thẻ `<script>` để thêm mã JavaScript vào trang web.

- Thuộc tính `type` và `language` không cần sử dụng.

- Một script đặt trong tệp ngoài được chèn vào trang web bằng `<script src="path/to/script.js"></script>`.

Còn nhiều điều để học về các script trong trình duyệt và cách nó tương tác với trang web. Nhưng hãy nhớ rằng trong phần này ta chỉ học về JavaScript, bạn không nên mất thời gian tìm hiểu cụ thể cách nó chạy trên trình duyệt. Chúng ta chỉ sử dụng trình duyệt như một cách để chạy JavaScript bởi nó rất thuận tiện cho việc học trực tuyến.

## II. Struction

## 1. Các câu lệnh

Các câu lệnh (statement) là các cấu trúc cú pháp và các lệnh nhằm thực hiện các hành động cụ thể.

Chúng ta đã thấy một câu lệnh là alert('Hello World!') nhằm hiển thị thông báo có nội dung "Hello World!".

Thường thì mã gồm nhiều câu lệnh. Các câu lệnh được ngăn cách với nhau bởi một dấu chấm phảy.

Ví dụ, ta chia "Hello World!" thành hai thông báo:

alert('Chào'); alert('thế giới!');

Mỗi câu lệnh thường viết trên một dòng để dễ đọc hơn:

alert('Chào');

alert('thế giới!');

## 2. Các dấu chấm phảy

Hầu như có thể bỏ qua dấu chấm phảy nếu mỗi lệnh được viết trên một dòng.

Cách viết sau vẫn hoạt động:

alert('Hello')

alert('World!')

Ở đây, JavaScript hiểu "ngầm" mỗi dấu xuống dòng là một dấu chấm phảy. Tính năng này gọi là [automatic semicolon insertion](https://tc39.es/ecma262/#sec-automatic-semicolon-insertion) (tự động chèn dấu chấm phảy).

Trong hầu hết trường hợp, một dấu xuống dòng ngụ ý một dấu chấm phảy. Nhưng cũng có trường hợp ngoại lệ!

Trong các trường hợp này dấu xuống dòng không được JavaScript xem là dấu chấm phảy. Ví dụ:

alert(3 +

1

+ 2);

Đoạn mã trên xuất ra giá trị 6 vì JavaScript không tự động chèn dấu chấm phảy vào vị trí dấu xuống dòng. Nó cho rằng nếu một dòng kết thúc bằng dấu cộng "+", thì nó là một biểu thức chưa hoàn chỉnh, nên không cần dấu chấm phảy đặt vào đó.Trong tình huống này JavaScript đã làm đúng!

Nhưng có những tình huống JavaScript làm sai, không đặt dấu chấm phảy vào nơi cần có.

Các lỗi xảy ra trong trường hợp này khá khó thấy và sửa.

Chúng tôi khuyên bạn nên đặt dấu chấm phảy giữa các câu lệnh ngay cả khi mỗi lệnh viết trên một dòng. Quy tắc này được cộng đồng áp dụng rộng rãi. Cùng nhắc lại một lần nữa -- có thể bỏ qua dấu chấm phảy trong hầu hết trường hợp. Nhưng để an toàn -- đặc biệt cho những người mới -- hãy luôn sử dụng chúng.

## 3. Các chú thích

Theo thời gian, chương trình ngày càng phức tạp. Trong chương trình ta cần bổ sung thêm các "chú thích" để diễn giải hoạt động của mã trong chương trình.

Các chú thích (comment) có thể đặt ở bất cứ đâu trong script. Nó không ảnh hưởng đến việc chạy script bởi JavaScript bỏ qua nó khi chạy.

Chú thích một dòng bắt đầu bằng hai dấu gạch chéo //.

Toàn bộ phần còn lại của dòng là chú thích. Chú thích một dòng có thể chiếm cả dòng, hoặc theo sau một câu lệnh.

Ví dụ:

// Chú thích này chiếm cả dòng

alert('Hello');

alert('World!'); // Chú thích này theo sau một câu lệnh

Chú thích nhiều dòng bắt đầu bằng một dấu gạch chéo và một dấu sao /\* và kết thúc bằng một dấu sao và một dấu gạch chéo \*/.

Ví dụ:

/\* Một ví dụ về hai thông báo.

Đây là một chú thích nhiều dòng

\*/

alert('Chào');

alert('thế giới!');

Nội dung của chú thích bị bỏ qua, nên nếu đặt mã JavaScript trong /\* ... \*/ nó sẽ không chạy.

Có thể lợi dụng điều này để tạm thời vô hiệu hóa một phần mã:

/\* "Chú thích hóa" mã để vô hiệu hóa nó

alert('');

\*/

alert('World');

Không thể đặt /⋅...⋅ trong một /⋅...⋅ khác.

Mã sau không chạy và dẫn tới một lỗi:

/\*

  /\* chú thích trong chú thích ?!? \*/

\*/

alert( 'thế giới!' );

Đừng do dự hãy chú thích ngay cho mã của bạn.

Chú thích tuy làm tăng kích thước mã, nhưng điều đó không thành vấn đê. Có nhiều công cụ thu nhỏ mã trước khi đưa lên máy chủ. Chúng xóa các chú thích và chú thích không xuất hiện trong mã sản phẩm chính thức. Bởi vậy chú thích không gây ra các ảnh hưởng tiêu cực tới chương trình.

Sau này trong chương info:code-quality chúng ta sẽ học cách viết chú thích sao cho hiệu quả.

## III. Strict-Mode

## 1. "use strict"

Một thời gian dài, JavaScript phát triển mà không gặp vấn đề tương thích. Các tính năng mới được thêm vào mà không cần thay đổi các tính năng cũ.

Điều đó có lợi ích là không bao giờ phá hỏng mã hiện có. Nhưng nhược điểm là bất kỳ sai lầm hoặc một quyết định không hoàn hảo nào tạo ra bởi các tác giả của JavaScript bị mắc kẹt trong ngôn ngữ này mãi mãi.

Việc này tiếp diễn cho đến năm 2009 khi ECMAScript 5 (ES5) xuất hiện. Nó đã thêm nhiều tính năng mới cho JavaScript và sửa lại vài tính năng cũ. Để đảm bảo các mã cũ vẫn chạy, các thay đổi này mặc định bị tắt đi. Nếu bạn không gặp phải vấn đề với những mã cũ bạn có thể kích hoạt những thay đổi này bằng cách sử dụng chỉ dẫn: "use strict".

Chỉ dẫn này trông như một chuỗi: "use strict" hoặc 'use strict'. Khi nó đặt ở đầu script, toàn bộ script làm việc theo cách "hiện đại".

Ví dụ:

"use strict";

// mã này làm việc theo cách "hiện đại"

...

Rất nhanh chúng ta sẽ tìm hiểu các hàm (một cách để nhóm các lệnh), vì vậy hãy lưu ý trước rằng "use strict" có thể được đặt ở đầu một hàm. Làm điều đó chỉ bật chế độ nghiêm ngặt trong hàm đó. Nhưng thông thường mọi người sử dụng nó cho toàn bộ script.

Bạn phải chắc chắn rằng đặt `"use strict"` ở đầu script, nếu không chế độ "strict" không được bật.

Chế độ "strict" không được bật ở đây:

```js no-strict

alert("some code");

// "use strict" bị bỏ qua--nó phải đặt ở đầu

"use strict";

// chế độ strict không được bật

```

Chỉ các chú thích được phép xuất hiện trước `"use strict"`.

`warn header="Không có cách hủy use strict" Không có chỉ dẫn nào kiểu như no use strict để tắt chế độ strict sau khi đã bật.

Một khi đã vào chế độ strict, không còn đường quay lại.

## Console của trình duyệt

Khi bạn sử dụng [console của trình duyệt](info:devtools) để chạy mã, xin lưu ý rằng nó mặc định không sử dụng `use strict`.

Thỉnh thoảng, khi `use strict` tạo ra sự khác biệt, bạn sẽ nhận được các kết quả không đúng.

Vì vậy, làm thế nào để thực sự sử dụng `use strict` trong console?

Đầu tiên, bạn có thể thử nhấn phím `key:Shift+Enter` để nhập mã trên nhiều dòng và đặt `use strict` ở đầu, như sau:

```js

'use strict'; <Shift+Enter để xuống dòng>

// ...mã của bạn

<Enter để chạy>

Nó làm việc trong hầu hết trình duyệt, như Firefox và Chrome.

Nếu không, ví dụ trong một trình duyệt cũ, có một cách xấu xí, nhưng đáng tin cậy để đảm bảo use strict. Đặt nó bên trong loại bao bọc này:

(function() {

'use strict';

// ...mã của bạn đây ...

})()

## 2. Chúng ta có nên sử dung “use strict”

Câu hỏi nghe có vẻ hiển nhiên, nhưng không phải vậy.

Người ta có thể khuyên bạn nên bắt đầu các tập lệnh với "use strict"... Nhưng bạn biết điều gì thú vị không?

JavaScript hiện đại hỗ trợ "classes" và "modules" - các cấu trúc ngôn ngữ nâng cao (chắc chắn chúng ta sẽ tìm hiểu chúng), mà tự động sử dụng use strict. Vì vậy, chúng ta không cần phải thêm chỉ thị "use strict", nếu chúng ta sử dụng chúng.

Vì vậy, bây giờ "use strict"; là một vị khách được chào đón ở đầu các tập lệnh của bạn. Sau đó, khi mã của bạn nằm trong các "class" và "module", bạn có thể bỏ qua nó.

Hiện tại, chúng ta đã biết về use strict nói chung.

Trong các chương tiếp theo, khi chúng ta tìm hiểu các tính năng của ngôn ngữ, chúng ta sẽ thấy sự khác biệt giữa các chế độ nghiêm ngặt và cũ. May mắn thay, không có nhiều khác biệt và chúng thực sự làm cho cuộc sống của chúng ta tốt hơn.

Tất cả các ví dụ trong hướng dẫn này giả định chế độ nghiêm ngặt trừ khi (hiếm hoi) được chỉ định khác.

## IV. Variables

## 1. Biến

Một [biến (variable)](https://vi.wikipedia.org/wiki/Bi%E1%BA%BFn_(khoa_h%E1%BB%8Dc_m%C3%A1y_t%C3%ADnh)) là một "vùng nhớ được đặt tên" chứa dữ liệu. Chúng ta có thể sử dụng các biến để lưu trữ các hàng hóa, người mua, và các dữ liệu khác...

Để tạo biến trong JavaScript, sử dụng từ khóa let.

Câu lệnh dưới đây tạo (còn gọi là khai báo) một biến có tên "message":

let message;

Sau khi tạo, chúng ta có thể lưu thông tin vào biến bằng toán tử gán =:

let message;

\*!\*

message = 'Hello'; // lưu chuỗi

\*/!\*

Chuỗi trên giờ được lưu vào vùng nhớ tướng ứng với biến. Chúng ta có thể truy cập vùng nhớ này bằng cách sử dụng tên biến.

let message;

message = 'Hello!';

\*!\*

alert(message); // hiển thị nội dung của biến

\*/!\*

Để ngắn gọn, chúng ta có thể kết hợp khai báo đồng thời gán giá trị cho biến trong một dòng:

let message = 'Hello!'; // định nghĩa biến và gán giá trị

alert(message); // Hello!

Chúng ta có thể khai báo nhiều biến trên một dòng:

let user = 'John', age = 25, message = 'Hello';

Trông ngắn hơn, nhưng bạn không nên làm như vậy. Để chương trình dễ đọc, vui lòng khai báo và gán giá trị cho mỗi biến trên một dòng.

Đây là phiên bản viết trên nhiều dòng. Tuy dài, nhưng dễ đọc hơn:

let user = 'John';

let age = 25;

let message = 'Hello';

Một vài người định nghĩa nhiều biến theo cách đặc biệt:

let user = 'John',

  age = 25,

  message = 'Hello';

...Hoặc đặt dấu phảy đằng trước:

let user = 'John'

  , age = 25

  , message = 'Hello';

Tất cả các cách trên đều hợp lệ, chọn cách nào là do sở thích của bạn.

smart header="var thay vì let" Trong các mã chương trình cũ, bạn sẽ bắt gặp từ khóa var thay vì let:

\*!\*var\*/!\* message = 'Hello';

Từ khóa var hầu như giống hệt let. Nó cũng khai báo biến, nhưng có chút khác biệt, bởi nó làm việc theo cách cũ.

Các khác biệt này được đề cập đến ở bài info:var, giờ ta chưa cần quan tâm đến chúng.

**Sự tương tự với đời thực**

Chúng ta có thể dễ dàng nắm bắt khái niệm "biến" nếu tưởng tượng nó như một "hộp" chứa dữ liệu được gắn nhãn.

Ví dụ, biến message có thể xem như một hộp gắn nhãn message lưu giá trị "Hello!" bên trong:

![](variable.svg)

Ta có thể đặt bất cứ giá trị nào trong hộp:

Ta cũng có thể thay thế giá trị trong hộp nếu muốn:

`js run

let message;

message = 'Hello!';

message = 'World!'; // thay đổi giá trị

alert(message);

`

Khi thay đổi giá trị, giá trị cũ bị loại bỏ, giá trị mới thay thế cho nó:

![](variable-change.svg)

Chúng ta có thể khai báo hai biến và sao chép dữ liệu từ biến này sang biến kia.

`js run

let hello = 'Hello world!';

let message;

\*!\*

// sao chép 'Hello world' từ hello sang message

message = hello;

\*/!\*

// giờ hai biến chứa dữ liệu giống nhau

alert(hello); // Hello world!

alert(message); // Hello world!

`

warn header="Khai báo hai lần sẽ gây ra lỗi"

Một biến chỉ nên được khai báo một lần.

Sự khai báo lặp lại cùng một biến là một lỗi:

`js run

let message = "This";

// lặp lại 'let' dẫn đến một lỗi

let message = "That"; // SyntaxError: 'message' đã được khai báo rồi

`

Vì thế, chúng ta nên khai báo mỗi biến một lần và tham chiếu đến nó mà không có let.

## 2. Đặt tên biến

Có hai ràng buộc khi đặt tên biến trong JavaScript:

1. Tên chỉ được có chữ cái, chữ số hoặc kí hiệu $ và \_.
2. Kí tự đầu tiên không được là chữ số.

Ví dụ về các tên hợp lệ:

let userName;

let test123;

Khi tên biến chứa nhiều từ, cách viết [camelCase](https://en.wikipedia.org/wiki/Camel_case) thường được sử dụng. Đó là: các từ viết liền nhau, ngoài từ đầu tiên các chữ cái bắt đầu mỗi từ đều được viết hoa: myVeryLongName.

Điều thú vị là kí tự '$' và '\_' được phép dùng. Chúng cũng chỉ là những kí tự thông thường, giống như các chữ cái, không có ý nghĩa gì đặc biệt.

Các tên sau hợp lệ:

let $ = 1; // khai báo biến có tên "$"

let \_ = 2; // và biến có tên "\_"

alert($ + \_); // 3

Ví dụ về các biến không hợp lệ:

let 1a; // không được bắt đầu là chữ số

let my-name; // dấu gạch ngang '-' không được phép sử dụng

Các biến có tên "apple" và "AppLE" khác nhau.

Có thể sử dụng chữ cái trong bất cứ ngôn ngữ nào để đặt tên biến, ví dụ:

`js let имя = '...';

let 我 = '...';

Có một [danh sách các từ khóa](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical\_grammar#Keywords), được sử dụng bởi chính JavaScript và không cho phép dùng làm tên biến.

Ví dụ: let, class, return, và function là các từ khóa.

Mã dưới dưới đây tạo ra một lỗi cú pháp:

`js run no-beautify

let let = 5; // lỗi, không thể đặt tên biến là từ khóa "let"!

let return = 5; // cũng không thể đặt tên biến là từ khóa "return"!

`

warn header="Lệnh gán khi không dùng use strict"

Thông thường, chúng ta cần khai báo biến trước khi sử dụng. Nhưng ngày trước, có thể tạo ra một biến chỉ bằng cách gán giá trị cho nó mà không cần sử dụng let. Hiện nay cách này vẫn làm việc nếu ta không sử dụng use strict.

// chú ý: không dùng "use strict" trong ví dụ này

num = 5; // biến "num" được tạo ra nếu chưa có

alert(num); // 5

Đây là cách viết không tốt và sẽ gây lỗi trong chế độ strict:

"use strict";

\*!\*

num = 5; // lỗi: num chưa định nghĩa

\*/!\*

**Các hằng**

Để khai báo hằng (biến có giá trị không đổi) sử dụng từ khóa const thay vì let:

`js

const myBirthday = '18.04.1982';

`

Biến khai báo bằng từ khóa const được gọi hà "hằng" (constant). Chúng không thể bị gán lại. Nếu cố tình làm thế sẽ dẫn tới một lỗi:

`js run

const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // lỗi, không thể gán lại một hằng!

`

Khi một lập trình viên muốn một biến sẽ không bao giờ thay đổi, họ có thể khai báo biến bằng const để chắc chắn điều này.

**Các hằng được viết hoa**

Có một quy ước được sử dụng rộng rãi là dùng hẳng để đặt tên cho những giá trị "khó nhớ" không thay đổi trong suốt chương trình.

Các hằng này được đặt tên bằng chữ viết hoa và ngăn cách các từ bằng "\_".

Ví dụ, tạo hẳng đặt tên các mã màu:

`js run

const COLOR\_RED = "#F00";

const COLOR\_GREEN = "#0F0";

const COLOR\_BLUE = "#00F";

const COLOR\_ORANGE = "#FF7F00";

// ...khi cần lấy một mã màu

let color = COLOR\_ORANGE;

alert(color); // #FF7F00

`

Ưu điểm:

- "COLOR\_ORANGE" dễ nhớ hơn "#FF7F00".

- Dễ gõ nhầm "#FF7F00" hơn COLOR\_ORANGE.

- Khi đọc, COLOR\_ORANGE có nhiều ý nghĩa hơn #FF7F00.

Khi nào nên sử dụng chữ hoa để đặt hằng và khi nào thì sử dụng chữ thường?

Một "hằng" có nghĩa là giá trị không bao giờ thay đổi. Nhưng có những hằng mà giá trị đã biết trước khi chạy (như các hằng mã màu ở trên) và những hằng được \*tính\* khi chạy chương trình, và không thay đổi sau đó.

Ví dụ:

`js

const pageLoadTime = /\* time taken by a webpage to load \*/;

`

Giá trị của "pageLoadTime" không được biết trước khi chương trình chạy, nên nó đặt tên theo cách bình thường. Nhưng nó vẫn là hẳng vì giá trị của nó không đổi sau đó.

Nói cách khác, hằng chỉ được viết hoa nếu giá trị của nó đã biết trước khi chạy chương trình.

**Đặt tên sao cho đúng?**

Nói về biến, có một thứ cực kỳ quan trọng.

Tên biến cần rõ ràng, mang nhiều ý nghĩa, mô tả dữ liệu nó chứa.

Đặt tên biến sao cho có nghĩa là một trong những kỹ năng quan trọng và phức tạp nhất trong lập trình. Nhìn thoáng qua tên biến ta có thể phân biệt được đây là đoạn mã viết bởi một lập trình viên dày dạn kinh nghiệm hay lập trình viên nghiệp dư.

Trong dự án thực tế, hầu hết thời gian được sử dụng để chỉnh sửa và mở rộng các đoạn mã hiện có hơn là viết lại từ đầu. Khi ta cần đọc lại các mã đã được chỉnh sửa trước đó, sẽ dễ hơn nếu các thông tin được mô tả tốt. Hay nói cách khác các biến được đặt tên tốt.

Hãy dành thời gian suy nghĩ về việc đặt tên biến trước khi khai báo nó. Bạn sẽ được đền đáp xứng đáng sau này.

Đây là vài quy tắc đặt tên bạn nên tuân theo:

- Đặt tên để mọi người hiểu được như userName hoặc shoppingCart.

- Đừng đặt tên viết tắt hoặc tên quá ngắn như a, b, c, trừ khi bạn biết mình đang làm gì.

- Tên cần ngắn gọn nhất nhưng mang tính mô tả nhiều nhất. Các ví dụ về cái tên tệ là data và value. Bởi chúng không nói lên điều gì đặc biệt. Chúng chỉ sử dụng được nếu ngữ cảnh của mã khiến dữ liệu hoặc giá trị mà chúng tham chiếu khác biệt rõ ràng với giá trị và dữ liệu ở nơi khác trong chương trình.

- Tuân thủ các quy tắc đặt tên biến trong nhóm của bạn và các quy tắc cá nhân. Nếu người thăm trang web được gọi là một "user" thì ta nên đặt tên các biến liên quan là currentUser hoặc newUser thay vì currentVisitor hay newManInTown.

Nghe có vẻ đơn giản? Đúng là như vậy thật! Nhưng tạo ra các tên biến vừa ngắn gọn vừa mô tả tốt thì không đơn giản chút nào.

`smart header="Tái sử dụng hay tạo mới?"

Và lưu ý cuối cùng. Có vài lập trình viên lười, thay vì tạo biến mới thường cố dùng lại các biến cũ.

Kết quả, các biến của họ giống như những chiếc hộp mà mọi người ném đủ thứ vào mà không thay đổi nhãn. Vậy nên không ai biết trong hộp hiện tại chứa cái gì. Muốn biết lại cần đến gần và kiểm tra.

Những lập trình viên này tiết kiệm được chút ít thời gian tạo biến, nhưng mất thời gian gấp 10 lần như vậy để gỡ lỗi.

Thêm một biến là tốt chứ không xấu.

Bởi các minifier và trình duyệt hiện đại sẽ tối ưu được việc có nhiều biến trong chương trình, và hiệu năng chương trình không bị ảnh hưởng nhiều. Sử dụng các biến khác nhau cho những giá trị khác nhau thậm chí còn giúp JavaScript engine tối ưu mã của bạn.

**Tóm tắt**

Chúng ta có thể tạo ra các biến để lưu dữ liệu bằng cách sử dụng các từ khóa var, let, hoặc const.

- let -- là cách khai báo hiện đại.

- var -- là cách khai báo cũ. Thường chúng ta không còn dùng nó nữa, nhưng ta cũng sẽ chỉ ra chút khác biệt của nó so với let ở bài <info:var>, nếu bạn cần biết.

- const -- giống như let, nhưng giá trị của biến không thay đổi được.

Các biến nên được đặt tên sao cho dễ hiểu được dữ liệu bên trong nó

## V. Types

## 1. Các kiểu dữ liệu

Một giá trị trong JavaScript luôn thuộc một loại nhất định. Ví dụ, một chuỗi hoặc một số.

Có tám kiểu dữ liệu cơ bản trong JavaScript. Ở đây, chúng ta sẽ trình bày tổng quát về chúng và trong các chương tiếp theo chúng ta sẽ nói chi tiết về chúng.

Chúng ta có thể đặt bất kỳ kiểu nào trong một biến. Ví dụ, một biến tại một thời điểm có thể là một chuỗi và sau đó lưu trữ một số:

*// no error*

*let* *message* *=* *"hello";*

*message* *=* *123456;*

Các ngôn ngữ lập trình cho phép điều đó, chẳng hạn như JavaScript, được gọi là "ngôn ngữ có kiểu động" hay "dynamically typed", có nghĩa là có tồn tại các kiểu dữ liệu, nhưng các biến không bị ràng buộc với bất kỳ kiểu nào trong số chúng.

## 2. Kiểu số

*let* *n* *=* *123;*

*n* *=* *12.345;*

Kiểu số (number) biểu diễn được cả số nguyên lẫn số thực.

Có nhiều toán tử làm việc với các số như: nhân \*, chia /, cộng +, trừ -, v.v.

Ngoài các số thông thường, còn có các giá trị số đặc biệt khác là: *Infinity*, *-Infinity* và NaN.

* *Infinity* biểu diễn giá trị [vô cùng](https://vi.wikipedia.org/wiki/V%C3%B4_t%E1%BA%ADn) ∞ trong toán học. Nó là một giá trị đặc biệt lớn hơn bất kỳ số nào.

Chúng ta có thể thu được giá trị này bằng cách chia một số dương cho không:

*alert(* *1* */* *0* *);* *// Infinity*

Or just reference it directly:

*alert(* *Infinity* *);* *// Infinity*

* NaN biểu diễn một lỗi tính toán. Nó là kết quả của một phép tính sai hoặc không xác định, ví dụ:

*alert(* *"not a number"* */* *2* *);* // NaN, phép chia như vậy là sai lầm

NaN rất khó chịu. Bất kỳ thao tác nào khác trên NaN sẽ trả về NaN:

*alert(* *"not a number"* */* *2* *+* *5* *);* *// NaN*

Cho nên, nếu NaN xuất hiện trong một biểu thức toán học, nó lan truyền tới kết quả của cả biểu thức.

*Làm toán trong JavaScript rất "an toàn". Ta có thể làm bất cứ thứ gì: chia cho không, coi một chuỗi như một số, ...*

*Tập lệnh sẽ không bao giờ dừng lại. Tệ nhất, chúng ta cũng nhận được giá trị `NaN`.*

Các giá trị số đặc biệt chính thức thuộc về kiểu "number". Tất nhiên chúng không phải là những con số theo nghĩa thông thường của từ này.

Ta sẽ học được nhiều hơn về các số ở chương info:number.

## 3. BigInt

Trong JavaScript, loại "số" không thể biểu diễn cho các giá trị số nguyên lớn hơn (253-1) (đó là 9007199254740991) hoặc nhỏ hơn -(253-1) đối với số âm. Đó là một hạn chế kỹ thuật do sự biểu diễn nội bộ của chúng gây ra.

Đối với hầu hết các mục đích đó là đủ, nhưng đôi khi chúng ta cần những con số thực sự lớn, ví dụ cho mật mã hoặc thời điểm chính xác đến micro giây.

Kiểu BigInt gần đây đã được thêm vào ngôn ngữ để biểu diễn các số nguyên có độ dài tùy ý.

Một giá trị BigInt được tạo bằng cách thêm n vào cuối một số nguyên:

*// "n" ở cuối nghĩa là nó là một BigInt*

*const* *bigInt* *=* *1234567890123456789012345678901234567890n;*

Vì các số BigInt hiếm khi cần thiết, chúng tôi không trình bày chúng ở đây mà dành cho chúng một chương riêng <info: bigint>. Hãy đọc nó khi bạn cần những con số lớn như vậy.

*Hiện tại, `BigInt` được hỗ trợ trong Firefox / Chrome / Edge / Safari, nhưng không hỗ trợ trong IE.*

Bạn có thể kiểm tra [MDN BigInt compatibility table](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt#Browser_compatibility) để biết phiên bản trình duyệt nào có hỗ trợ.

## 4. Kiểu chuỗi

Một chuỗi trong JavaScript bắt buộc phải nằm giữa các dấu nháy.

*let str = "Hello";*

*let str2 = 'Single quotes are ok too';*

*let phrase = `can embed ${str}`;*

Trong JavaScript, có 3 kiểu dấu nháy:

1. Dấu nháy kép: *"Hello".*
2. Dấu nháy đơn: *'Hello'.*
3. Backticks: *`Hello`*.

Dấu nháy kép và dấu nháy đơn là những dấu nháy "đơn giản". Thực tế không có sự khác biệt giữa chúng trong JavaScript.

Backtick là các dấu nháy với tính năng mở rộng. Nó cho phép ta nhúng biến và biểu thức vào chuỗi bằng cách đặt chúng trong ${…}, ví dụ:

*let name = "John";*

*// nhúng một biến*

*alert( `Hello, \*!\*${name}\*/!\*!` ); // Hello, John!*

*// nhúng một biểu thức*

*alert( `the result is \*!\*${1 + 2}\*/!\*` ); // the result is 3*

Biểu thức trong ${…} được chạy và kết quả của nó trở thành một phần của chuỗi. Chúng ta có thể đặt bất cứ gì vào đó: một biến như name hay một biểu thức số học như 1 + 2 hoặc biểu thức phức tạp hơn.

Chú ý rằng chỉ backtick cho phép tính năng này. Dấu nháy đơn và nháy kép không hỗ trợ!

*alert( "the result is ${1 + 2}" ); // the result is ${1 + 2} (nháy kép không làm gì cả)*

Chúng ta sẽ tìm hiểu về chuỗi kỹ hơn ở chương info:string.

Trong một số ngôn ngữ, có một kiểu "ký tự" đặc biệt cho một ký tự đơn lẻ. Ví dụ, trong ngôn ngữ C và trong Java, nó được gọi là "char".

Trong JavaScript, không có kiểu như vậy. Chỉ có một kiểu: `string` (chuỗi).

## 5. Boolean( kiểu logic)

Kiểu lôgic chỉ có hai giá trị: *true* và *false*.

Kiểu này thường dùng để lưu các giá có/không: true nghĩa là "có, đúng", và false nghĩa là "không, sai".

Ví dụ:

*let* *nameFieldChecked* *=* *true;* *// có, trường name đã được kiểm tra*

*let* *ageFieldChecked* *=* *false;* *// không, trường age chưa được kiểm tra*

Giá trị lôgic cũng là kết quả của các phép so sánh:

*let* *isGreater* *=* *4* *>* *1;*

*alert(* *isGreater* *);* *// true (kết quả so sánh là "đúng")*

Chúng ta sẽ tìm hiểu sâu hơn về kiểu này trong chương info:logical-operators.

## 6. Giá trị “Undefined”

Giá trị đặc biệt undefined cũng đứng một mình. Nó tạo ra một kiểu riêng, giống như null.

Ý nghĩa của undefined là "chưa được gán giá trị".

Nếu một biến đã được khai báo, nhưng chưa được gán, giá trị của nó là undefined:

*let* *age;*

*alert(age);* *// hiện "undefined"*

Về mặt kỹ thuật, có thể gán undefined cho một biến một cách tường minh:

*let* *age* *=* *100;*

*// đổi giá trị thành undefined*

*age* *=* *undefined;*

*alert(age);* *// "undefined"*

Nhưng chúng ta không nên làm điều đó. Thông thường, người ta sử dụng null để gán giá trị "trống" hoặc "không xác định" cho một biến, trong khi undefined được dành riêng làm giá trị khởi tạo mặc định cho những thứ chưa được gán.

## 7. Objects and Symbols

Kiểu object hay đối tượng là một kiểu đặc biệt.

Tất cả các kiểu khác được gọi là "primitive" (nguyên thủy) vì giá trị của chúng chỉ có thể chứa một thứ duy nhất (có thể là một chuỗi hoặc một số hoặc bất cứ thứ gì). Ngược lại, các đối tượng được sử dụng để lưu trữ các tập hợp dữ liệu và các thực thể phức tạp hơn.

Vì quan trọng như vậy, các đối tượng xứng đáng được đối xử đặc biệt. Chúng ta sẽ bàn về chúng sau trong chương info:object, sau khi chúng ta tìm hiểu thêm về các giá trị nguyên thủy.

Kiểu symbol (biểu tượng) được sử dụng để tạo các định danh duy nhất cho các đối tượng. Chúng ta buộc phải đề cập đến nó ở đây chỉ để khỏi thiếu sót, nhưng cũng trì hoãn các chi tiết cho đến khi chúng ta biết về các đối tượng.

## 8. Toán tử Typeof

Toán tử typeof trả về kiểu của đối số. Nó hữu dụng khi chúng ta muốn kiểm tra kiểu dữ liệu của một giá trị để thực hiện các công việc khác nhau dựa trên kết quả.

Nó hỗ trợ hai cú pháp:

1. Như một toán tử: *typeof x.*
2. Như một hàm: *typeof(x).*

Nói cách khác, nó làm việc với cả dạng có dấu ngoặc đơn hoặc không có dấu ngoặc đơn. Kết quả hoàn toàn giống nhau.

Gọi typeof x trả về một chuỗi mô tả tên của kiểu dữ liệu:

*typeof undefined // "undefined"*

*typeof 0 // "number"*

*typeof 10n // "bigint"*

*typeof true // "boolean"*

*typeof "foo" // "string"*

*typeof Symbol("id") // "symbol"*

*\*!\**

*typeof Math // "object" (1)*

*\*/!\**

*\*!\**

*typeof null // "object" (2)*

*\*/!\**

*\*!\**

*typeof alert // "function" (3)*

*\*/!\**

Ba dòng cuối cần phải giải thích thêm:

1. Math là một đối tượng có sẵn cung cấp các phép tính toán học. Chúng ta sẽ tìm hiểu nó trong chương info:number. Ở đây, nó chỉ đóng vai trò như một ví dụ về một đối tượng.
2. Kết quả của typeof null là "object". Đó là một lỗi được chính thức công nhận trong hành vi của typeof, xuất hiện từ những ngày đầu của JavaScript và được giữ lại để tương thích. Chắc chắn, null không phải là một đối tượng. Nó là một giá trị đặc biệt với một kiểu riêng biệt của nó.
3. Kết quả của typeof alert là "function", vì alert là một hàm. Chúng ta sẽ nghiên cứu các hàm trong các chương tiếp theo, chúng ta cũng sẽ thấy rằng không có kiểu "function" đặc biệt nào trong JavaScript. Các hàm thuộc về kiểu đối tượng. Nhưng typeof xử lý chúng theo cách khác, trả về "function". Điều đó cũng đến từ những ngày đầu của JavaScript. Về mặt kỹ thuật, hành vi như vậy không đúng, nhưng có thể thuận tiện trong thực tế.

## Tóm tắt

Có 8 kiểu dữ liệu cơ bản trong JavaScript.

* number dành cho các số thuộc bất kỳ loại nào: số nguyên hoặc dấu phẩy động, số nguyên được giới hạn bởi ±(253-1).
* bigint dành cho các số nguyên có độ dài tùy ý.
* string dành cho các chuỗi. Một chuỗi có thể có không hoặc nhiều ký tự, không có kiểu ký tự đơn riêng biệt.
* boolean dành cho các giá trị true/false.
* null dành cho các giá trị không xác định -- một kiểu độc lập có một giá trị duy nhất null.
* undefined dành cho các giá trị chưa được gán -- một kiểu độc lập có một giá trị duy nhất undefined.
* object dành cho các cấu trúc dữ liệu phức tạp hơn.
* symbol dành cho các định danh duy nhất.

Toán tử typeof cho phép chúng ta xem kiểu nào được lưu trữ trong một biến.

* Hai dạng: typeof x hoặc typeof(x).
* Trả về một chuỗi với tên của kiểu dữ liêu, ví dụ "string".
* Với null nó trả về "object" -- đây là một lỗi còn tồn tại trong ngôn ngữ, nó không thực sự là một đối tượng.

Trong các chương tiếp theo, chúng ta sẽ tập trung vào các giá trị nguyên thủy và khi chúng ta đã quen thuộc với chúng, chúng ta sẽ chuyển sang các đối tượng.

## V. Alert-Prompt-Confirm

## 1. Alert

Phương thức alert() dùng để hiển thị một hộp thoại thông báo đến người dùng.

Phương thức alert() được sử dụng theo cú pháp như sau:

alert("Hello");

Tương tự, để hiển thị các ký tự như dấu ngoặc kép (") hoặc dấu ngoặc đơn (') thì ta sẽ sử dụng \" và \'

## 2. Prompt

Phương thức prompt() dùng để hiển thị một hộp thoại thông báo và yêu cầu người dùng nhập dữ liệu vào.

Phương thức prompt() được sử dụng theo cú pháp như sau:

result = prompt(title, [default]);

default : Tham số thứ hai tùy chọn, giá trị ban đầu cho trường đầu vào, không bắt buộc.

Chạy mã này trong Internet Explorer để xem:

let test = prompt("Test");

Vì vậy, để lời nhắc hiển thị tốt trong IE, bạn nên luôn cung cấp đối số thứ hai:

let test = prompt("Test", ''); // <-- for IE

## 3. Confirm

Phương thức confirm() dùng để hiển thị một hộp thoại thông báo và yêu cầu người dùng phải xác nhận là đồng ý hoặc quay lại (OK hoặc Cancel).

Phương thức confirm() được sử dụng theo cú pháp như sau:

result = confirm(question);

## VI. Type Conversions

## 1. Chuyển đổi loại

Hầu hết thời gian, các toán tử và hàm tự động chuyển đổi các giá trị được cung cấp cho chúng thành đúng loại.

Ví dụ: **alert** tự động chuyển đổi bất kỳ giá trị nào thành chuỗi để hiển thị giá trị đó. Các phép toán chuyển đổi giá trị thành số.

Cũng có những trường hợp khi chúng ta cần chuyển đổi rõ ràng một giá trị thành loại dự kiến.

In this chapter, we won't cover objects. For now we'll just be talking about primitives.

Later, after we learn about objects, in the chapter <info:object-toprimitive> we'll see

## 2. Chuyển đổi chuỗi

Chuyển đổi chuỗi xảy ra khi chúng ta cần dạng chuỗi của một giá trị.

Ví dụ, alert(value)hiện nó để hiển thị giá trị.

Chúng ta cũng có thể gọi String(value)hàm để chuyển đổi một giá trị thành một chuỗi:

*let* *value* *=* *true;*

*alert(typeof* *value);* *// boolean*

*\*!\**

*value* *=* *String(value);* *// now value is a string "true"*

*alert(typeof* *value);* *// string*

*\*/*!*\**

Chuyển đổi chuỗi chủ yếu là rõ ràng. A false trở thành "false", null trở thành "null", v.v.

## 3. Chuyển đổi số

Chuyển đổi số xảy ra tự động trong các hàm và biểu thức toán học.

Ví dụ: khi phép chia /được áp dụng cho các số không phải là số:

*alert( "6" / "2" ); // 3, strings are converted to numbers*

Chúng ta có thể sử dụng hàm Number(value) để chuyển đổi một cách rõ ràng value thành một số:

*let str = "123";*

*alert(typeof str); // string*

*let num = Number(str); // becomes a number 123*

*alert(typeof num); // number*

Chuyển đổi rõ ràng thường được yêu cầu khi chúng ta đọc một giá trị từ nguồn dựa trên chuỗi như biểu mẫu văn bản nhưng mong muốn nhập một số.

Nếu chuỗi không phải là số hợp lệ, kết quả của việc chuyển đổi như vậy là NaN. Ví dụ:

*let age = Number("an arbitrary string instead of a number");*

*alert(age); // NaN, conversion failed*

Quy tắc chuyển đổi số:

| **Giá trị** | **Trở thành...** |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true and false | 1Và0 |
| string | Khoảng trắng từ đầu và cuối được loại bỏ. Nếu chuỗi còn lại trống, kết quả là 0. Mặt khác, số được "đọc" từ chuỗi. Một lỗi cho NaN. |

Ví dụ:

*alert( Number(" 123 ") ); // 123*

*alert( Number("123z") ); // NaN (error reading a number at "z")*

*alert( Number(true) ); // 1*

*alert( Number(false) ); // 0*

Xin lưu ý điều đó null và undefined hành xử khác ở đây: null trở thành 0 trong khi undefined trở thành NaN.

Hầu hết các toán tử toán học cũng thực hiện chuyển đổi như vậy, chúng ta sẽ thấy điều đó trong chương tiếp theo.

## 4. Chuyển đổi Boolean

Chuyển đổi Boolean là chuyển đổi đơn giản nhất.

Nó xảy ra trong các phép toán logic (sau này chúng ta sẽ gặp các bài kiểm tra điều kiện và những thứ tương tự khác) nhưng cũng có thể được thực hiện một cách rõ ràng bằng lệnh gọi tới Boolean(value).

Quy tắc chuyển đổi:

* Các giá trị trực quan là "trống", như 0, một chuỗi rỗng, null, undefined và NaN, trở thành false.
* Các giá trị khác trở thành true.

Ví dụ:

*alert( Boolean(1) ); // true*

*alert( Boolean(0) ); // false*

*alert( Boolean("hello") ); // true*

*alert( Boolean("") ); // false*

warn header="Xin lưu ý: chuỗi có số 0 \"0\"là true" Một số ngôn ngữ (cụ thể là PHP) coi 0 là false. Nhưng trong JavaScript, một chuỗi không trống luôn là true.

*alert( Boolean("0") ); // true*

*alert( Boolean(" ") ); // spaces, also true (any non-empty string is true)*

*## Summary*

*The three most widely used type conversions are to string, to number, and to boolean.*

*\*\*`String Conversion`\*\* -- Occurs when we output something. Can be performed with `String(value)`. The conversion to string is usually obvious for primitive values.*

*\*\*`Numeric Conversion`\*\* -- Occurs in math operations. Can be performed with `Number(value)`.*

*The conversion follows the rules:*

*| Value | Becomes... |*

*|-------|-------------|*

*|`undefined`|`NaN`|*

*|`null`|`0`|*

*|<code>true&nbsp;/&nbsp;false</code> | `1 / 0` |*

*| `string` | The string is read "as is", whitespaces from both sides are ignored. An empty string becomes `0`. An error gives `NaN`. |*

*\*\*`Boolean Conversion`\*\* -- Occurs in logical operations. Can be performed with `Boolean(value)`.*

Follows the rules:

*| Value | Becomes... |*

*|-------|-------------|*

*|`0`, `null`, `undefined`, `NaN`, `""` |`false`|*

*|any other value| `true` |*

*Most of these rules are easy to understand and memorize. The notable exceptions where people usually make mistakes are:*

*- `undefined` is `NaN` as a number, not `0`.*

*- `"0"` and space-only strings like `" "` are true as a boolean.*

Objects aren't covered here. We'll return to them later in the chapter <info:object-toprimitive>

## VII. Operators

## 1. Số hạng: "nguyên đơn", "nhị phân","toán hạng"

Trước khi tiếp tục, hãy nắm bắt một số thuật ngữ phổ biến.

* Toán hạng -- áp dụng cho các toán tử. Chẳng hạn, trọng phép nhân của 5 \* 2 có hai toán hạng: toán hạng bên trái là 5 và toán hạng bên phải là 2. Đôi khi, mọi người gọi là những "đối số" thay vì "toán hạng".
* Một toán hạng là đơn nguyên nếu nó có một toán hạng đơn. Ví dụ: phủ định đơn nguyên - đảo ngược dấu của một số:

let x = 1;

\*!\*

x = -x;

\*/!\*

alert( x ); // -1, phủ định đơn nguyên được áp dụng

Một toán tử là binary nếu nó có hai toán hạng. Trừ cũng tồn tại tương tự ở dạng đơn nguyên:

let x = 1, y = 3;

alert( y - x ); // 2, binary minus subtracts values

Về hình thức, trong các ví dụ trên, chúng ta có hai toán tử khác nhau có chung ký hiệu: toán tử phủ định, toán tử một ngôi đảo ngược dấu và toán tử trừ, toán tử nhị phân trừ một số khỏi một số khác.

## 2. Toán học

Các hoạt động toán học sau đây được hỗ trợ:

* Cộng +,
* Trừ -,
* Nhân \*,
* Chia /,
* Chia lấy dư %,
* Lũy thừa \*\*.

Bốn cái đầu tiên rất đơn giản, trong khi % và \*\* cần một vài giải thích về chúng.

## 3. Chia lấy dư %

Toán tử chia lấy dư %, mặc dù xuất hiện nhưng không liên quan đến phần trăm.

Kết quả của a % b là [phần dư](https://en.wikipedia.org/wiki/Remainder) của phép chia số nguyên của a cho b.

Ví dụ:

alert( 5 % 2 ); // 1, phần dư của 5 chia 2

alert( 8 % 3 ); // 2, phần dư của 8 chia 3

## 4. Lũy thừa

Toán tử lũy thừa a \*\* b nâng a lên lũy thừa của b.

Ví dụ:

alert( 2 \*\* 2 ); // 2² = 4

alert( 2 \*\* 3 ); // 2³ = 8

alert( 2 \*\* 4 ); // 2⁴ = 16

Cũng giống như trong toán học, toán tử lũy thừa cũng được xác định cho các số không nguyên.

Ví dụ, một căn bậc hai là một lũy thừa của ½:

alert( 4 \*\* (1/2) ); // 2 (power of 1/2 is the same as a square root)

alert( 8 \*\* (1/3) ); // 2 (power of 1/3 is the same as a cubic root)

## 5. Chuyển đổi số, đơn nguyên

Dấu cộng + tồn tại ở hai dạng: dạng nhị phân chúng ta đã dùng ở trên và dạng đơn nguyên.

Cộng đơn nguyên, hay nói cách khác, toán tử cộng + áp dụng cho giá trị đơn, không đụng gì đến các số. Nhưng nếu toán hạng không phải là một số, thì dấu cộng đơn nguyên sẽ chuyển nó thành một số.

Ví dụ:

// Không đụng đến số

let x = 1;

alert( +x ); // 1

let y = -2;

alert( +y ); // -2

\*!\*

// Chuyển đổi những thứ không phải số

alert( +true ); // 1

alert( +"" );   // 0

\*/!\*

Nó làm điều tương tự như là Number(...), nhưng ngắn hơn.

Nhu cầu chuyển đổi chuỗi thành số phát sinh rất thường xuyên. Ví dụ: nếu chúng ta đang nhận các giá trị từ các trường nhập liệu HTML, thì chúng thường là các chuỗi. Nếu chúng ta muốn cộng chúng thì sao?

Cộng nhị phân sẽ cộng chúng như là các chuỗi:

let apples = "2";

let oranges = "3";

alert( apples + oranges ); // "23", cộng nhị nhân nối chuỗi

Nếu ta muốn xem chúng như là các số, ta cần biến đổi chúng và sau đó cộng:

let apples = "2";

let oranges = "3";

\*!\*

// cả hai giá trị được chuyển đổi sang số trước khi cộng nhị phân

alert( +apples + +oranges ); // 5

\*/!\*

// biến thể dài hơn

// alert( Number(apples) + Number(oranges) ); // 5

Từ quan điểm của một nhà toán học, sự phong phú của các phép cộng có vẻ lạ. Nhưng theo quan điểm của một lập trình viên, không có gì đặc biệt: các phép cộng đơn được áp dụng trước tiên, họ chuyển đổi chuỗi thành số và sau đó phép cộng nhị phân cộng chúng lại.

Tại sao các phép cộng đơn nguyên được áp dụng cho các giá trị trước các giá trị nhị phân? Như chúng ta thấy, đó là vì sự ưu tiên cao hơn của nó.

## 6. Toán tử ưu tiên

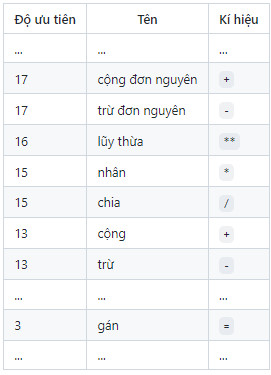
Nếu một biểu thức có nhiều hơn một toán tử, thì thứ tự thực thi được xác định theo độ ưu tiên của chúng, hay nói cách khác, thứ tự ưu tiên mặc định của các toán tử.

Từ trường học, tất cả chúng ta đều biết rằng phép nhân trong biểu thức 1 + 2 \* 2 nên được tính trước khi cộng. Đó chính xác là độ ưu tiên. Phép nhân được cho là có độ ưu tiên cao hơn phép cộng.

Dấu ngoặc đơn ghi đè bất kỳ quyền ưu tiên nào, vì vậy nếu chúng ta không hài lòng với thứ tự mặc định, chúng ta có thể sử dụng chúng để thay đổi. Ví dụ, viết (1 + 2) \* 2.

Có nhiều toán tử trong JavaScript. Mỗi toán tử có một số ưu tiên tương ứng. Cái có số lượng lớn hơn được thực hiện đầu tiên. Nếu quyền ưu tiên là như nhau, thứ tự thực hiện là từ trái sang phải.

Đây là đoạn trích từ [bảng độ ưu tiên](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence) (bạn không cần phải nhớ nó, nhưng nhớ rằng các toán tử đơn nguyên có độ ưu tiên cao hơn các toán tử nhị phân):



Như chúng ta thấy, phép "cộng đơn nguyên" có độ ưu tiên là 17 cao hơn 13 của phép "cộng" (cộng nhị phân). Đó là lý do, trong biểu thức "+apples + +oranges", cộng đơn nguyên chạy trước cộng nhị phân.

## 7. Phép gán

Hãy lưu ý rằng một phép gán = cũng là một toán tử. Nó được liệt kê trong bảng ưu tiên với mức ưu tiên rất thấp là 3.

Đó là lý do tại sao, khi chúng ta gán một biến, như x = 2 \* 2 + 1, các phép tính được thực hiện trước và sau đó dấu =được chạy, lưu trữ kết quả trong x.

let x = 2 \* 2 + 1;

alert( x ); // 5

## 8. Phép gán = trả về một giá trị

Việc “=” là một toán tử, không phải là một cấu trúc ngôn ngữ "ma thuật" có một hàm ý thú vị.

Một toán tử trong JavaScript trả về một giá trị. Điều đó rõ ràng đúng với phép cộng + hoặc phép trừ -, nhưng cũng đúng với =.

Gọi x = value nghĩa là gán value vào x và sau đó trả về nó.

Đây là demo sử dụng một phép gán như một phần của biểu thức phức tạp hơn:

let a = 1;

let b = 2;

\*!\* let c = 3 - (a = b + 1);

\*/!\*

alert( a ); // 3

alert( c ); // 0

Trong ví dụ trên, kết quả của biểu thức (a = b + 1) được gán cho a (đó là 3). Sau đó nó được sử dụng tiếp để tính toán.

Lập trình thật hài hước, phải không? Chúng ta nên hiểu cách thức hoạt động của nó, bởi vì đôi khi chúng ta thấy nó trong các thư viện JavaScript.

Tuy nhiên, đừng nên viết code như thế. Viết như vậy chắc chắn không làm cho code rõ ràng hay dễ đọc hơn.

## 9. Phép gán chuỗi

Một tính năng thú vị khác là khả năng gán chuỗi:

let a, b, c;

\*!\*

a = b = c = 2 + 2;

\*/!\*

alert( a ); // 4

alert( b ); // 4

alert( c ); // 4

Gán chuỗi đánh giá từ phải sang trái. Đầu tiên, biểu thức ngoài cùng bên phải 2 + 2 được tính rồi gán cho các biến ở bên trái: c, b và a. Cuối cùng, tất cả các biến chia sẻ một giá trị duy nhất.

Một lần nữa, với mục tiêu dễ đọc dễ hiểu, ta nên chia code thành nhiều dòng:

c = 2 + 2;

b = c;

a = c;

Nó khiến code dễ đọc, đặc biệt khi quan sát nhanh code.

## 10. Sửa trực tiếp

Chúng ta thường thực hiện phép tính trên một biến và lưu kết quả và chính nó.

Ví dụ:

let n = 2;

n = n + 5;

n = n \* 2;

Điều này có thể làm ngắn hơn bằng cách dùng += và \*=:

let n = 2;

n += 5; // now n = 7 (same as n = n + 5)

n \*= 2; // now n = 14 (same as n = n \* 2)

alert( n ); // 14

Phép tính ngắn "sửa-và-gán" tồn tại cho tất cả các toán tử số học và bitwise: /=, -=, vv.

Các toán tử như vậy có cùng mức độ ưu tiên như một phép gán thông thường, vì vậy chúng chạy sau hầu hết các phép tính khác:

let n = 2;

n \*= 3 + 5;

alert( n ); // 16  (phần bên phải thực hiện trước, giống như n \*= 8)

## 11. Tăng/giảm

Tăng hoặc giảm một số là một trong những hoạt động số phổ biến nhất.

Vì vậy, có các toán tử đặc biệt cho nó:

* Phép tăng ++ làm tăng giá trị của biến lên 1:

let counter = 2;

counter++; // hoạt động giống như counter = counter + 1, nhưng ngắn hơn

alert( counter ); // 3

* Phép giảm -- giảm giá trị của biến đi 1:

let counter = 2;

counter--;        // hoạt động giống như counter = counter - 1, nhưng ngắn hơn

alert( counter ); // 1

Toán tử ++ và -- có thể được đặt trước hoặc sau một biến.

* Khi toán tử đi sau biến, nó ở "dạng hậu tố": counter++.
* "Dạng tiền tố" là khi toán tử đi trước biến: ++counter.

Cả hai câu lệnh này đều làm cùng một điều: tăng counter lên 1.

Có gì khác biệt không? Có, nhưng chúng ta chỉ thấy chúng nếu ta sử dụng giá trị trả về của ++/--.

Hãy làm rõ. Như chúng ta biết, tất cả các toán tử đều trả về một giá trị. Phép tăng/phép giảm cũng không ngoại lệ. Dạng tiền tố trả về giá trị mới trong khi dạng hậu tố trả về giá trị cũ (trước khi tăng/giảm).

Để thấy sự khác biệt, xem ví dụ sau

let counter = 1;

let a = ++counter; // (\*)

alert(a); // \*!\*2\*/!\*

Trong dòng (\*), dạng *tiền tố* ++counter tăng counter và trả về giá trị mới là 2. Vì vậy, hàm alert sẽ hiển thị 2.

Giờ, hãy sử dụng dạng hậu tố:

let counter = 1;

let a = counter++; // (\*) thay đổi ++counter sang counter++

alert(a); // \*!\*1\*/!\*

Trong dòng (\*), dạng *hậu tố* counter++ cũng tăng counter nhưng trả về giá trị *cũ* (trước khi tăng). Do đó, hàm alert sẽ hiển thị 1.

**Tóm tắt:**

* Nếu kết quả tăng/giảm không được sử dụng, sẽ không có sự khác biệt trong các dạng được sử dụng:

let counter = 0;

counter++;

++counter;

alert( counter ); // 2, những dòng phía trên đều giống nhau

* Nếu chúng ta muốn tăng giá trị *và* ngay lập tức sử dụng kết quả của toán tử, chúng ta cần dạng tiền tố:

let counter = 0;

alert( ++counter ); // 1

* Nếu chúng tâ muốn tăng một giá trị nhưng sử dụng giá trị trước đó của nó, thì chúng ta cần dạng hậu tố:

let counter = 0;

alert( counter++ ); // 0

## 12. Toán tử Bitwise

Toán tử bitwise coi các đối số là số nguyên 32 bit và hoạt động ở mức độ biểu diễn nhị phân của chúng.

Các toán tử này không dành riêng cho JavaScript. Chúng được hỗ trợ trong hầu hết các ngôn ngữ lập trình.

Danh sách các toán tử:

* AND ( & )
* OR ( | )
* XOR ( ^ )
* NOT ( ~ )
* LEFT SHIFT ( << )
* RIGHT SHIFT ( >> )
* ZERO-FILL RIGHT SHIFT ( >>> )

Các toán tử này rất hiếm khi được sử dụng, khi chúng ta cần xử lý các số ở mức rất thấp (bitwise). Chúng ta sẽ không cần đến những toán tử này trong thời gian tới, vì việc phát triển web ít sử dụng chúng, nhưng trong một số lĩnh vực đặc biệt, chẳng hạn như mật mã, chúng rất hữu ích. Bạn có thể đọc chương [Toán tử](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Bitwise)[Bitwise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Bitwise) trên MDN khi có nhu cầu.

## 13. Dấu phảy

Toán tử dấu phẩy , là một trong những toán tử hiếm nhất và bất thường nhất. Đôi khi, nó được sử dụng để viết mã ngắn hơn, vì vậy chúng ta cần biết nó để hiểu chuyện gì đang xảy ra.

Toán tử dấu phẩy cho phép chúng ta đánh giá một số biểu thức, chia chúng bằng dấu phẩy. Mỗi trong số chúng được đánh giá nhưng chỉ kết quả của cái cuối cùng được trả về.

Ví dụ:

\*!\*

let a = (1 + 2, 3 + 4);

\*/!\*

alert( a ); // 7 (kết quả của 3 + 4)

Đây, biểu thức đầu tiên 1 + 2 được chạy và kết quả của nó bị ném đi. Sau đó, 3 + 4 được chạy và trả về một kết quả.

Xin lưu ý rằng toán tử dấu phẩy có độ ưu tiên rất thấp, thấp hơn =, vì vậy dấu ngoặc đơn rất quan trọng trong ví dụ trên.

Nếu không có chúng: a = 1 + 2, 3 + 4 chạy + trước, tổng hợp các số thành a = 3, 7 sau đó phép gán = gán a = 3, và phần còn lại bị bỏ qua. Nó giống như (a = 1 + 2), 3 + 4

Tại sao chúng ta cần một toán tử loại bỏ mọi thứ trừ biểu thức cuối cùng?

Đôi khi, mọi người sử dụng nó trong các cấu trúc phức tạp hơn để đặt một số hành động trên một dòng.

Ví dụ:

// ba toán tử trên một dòng

for (\*!\*a = 1, b = 3, c = a \* b\*/!\*; a < 10; a++) {

 ...

}

Các thủ thuật như vậy được sử dụng trong nhiều JavaScript frameworks. Đó là lý do tại sao chúng ta đề cập đến chúng. Nhưng thường thì nó không cải thiện khả năng dễ đọc của code vì vậy chúng ta nên suy nghĩ kỹ trước khi sử dụng chúng.

## VIII. Comparison

## 1. So sánh

Chúng tôi biết nhiều toán tử so sánh từ toán học.

Trong JavaScript, chúng được viết như thế này:

* Lớn hơn/nhỏ hơn: *a > b, a < b.*
* Lớn hơn/nhỏ hơn hoặc bằng: *a >= b, a <= b.*
* Equals: *a == b,* xin lưu ý rằng dấu đẳng thức kép ==có nghĩa là phép thử đẳng thức, trong khi dấu đẳng thức đơn *a = b* có nghĩa là phép gán.
* Không bằng. Trong toán học ký hiệu là ≠, nhưng trong JavaScript nó được viết là *a != b.*

Trong bài viết này, chúng ta sẽ tìm hiểu thêm về các loại so sánh khác nhau, cách JavaScript tạo ra chúng, bao gồm cả những đặc thù quan trọng.

Cuối cùng, bạn sẽ tìm thấy một công thức hay để tránh các vấn đề liên quan đến "những điều kỳ quặc về JavaScript".

## 2. Boolean là kết quả

Tất cả các toán tử so sánh trả về một giá trị boolean:

* true-- có nghĩa là "có", "chính xác" hoặc "sự thật".
* false-- có nghĩa là "không", "sai" hoặc "không phải sự thật".

Ví dụ:

*alert( 2 > 1 ); // true (correct)*

*alert( 2 == 1 ); // false (wrong)*

*alert( 2 != 1 ); // true (correct)*

Một kết quả so sánh có thể được gán cho một biến, giống như bất kỳ giá trị nào:

*let result = 5 > 4; // assign the result of the comparison*

*alert( result ); // true*

## 3. So sánh chuỗi

Để xem liệu một chuỗi có lớn hơn chuỗi khác hay không, JavaScript sử dụng cái gọi là thứ tự "từ điển" hoặc "từ điển".

Nói cách khác, các chuỗi được so sánh từng chữ một.

Ví dụ:

*alert( 'Z' > 'A' ); // true*

*alert( 'Glow' > 'Glee' ); // true*

*alert( 'Bee' > 'Be' ); // true*

Thuật toán so sánh hai chuỗi rất đơn giản:

1. So sánh ký tự đầu tiên của cả hai chuỗi.
2. Nếu ký tự đầu tiên từ chuỗi đầu tiên lớn hơn (hoặc nhỏ hơn) so với chuỗi khác, thì chuỗi đầu tiên lớn hơn (hoặc nhỏ hơn) chuỗi thứ hai. Đã được thực hiện.
3. Mặt khác, nếu các ký tự đầu tiên của cả hai chuỗi giống nhau, hãy so sánh các ký tự thứ hai theo cùng một cách.
4. Lặp lại cho đến khi kết thúc một trong hai chuỗi.
5. Nếu cả hai chuỗi kết thúc ở cùng độ dài, thì chúng bằng nhau. Mặt khác, chuỗi dài hơn sẽ lớn hơn.

Trong ví dụ đầu tiên ở trên, phép so sánh 'Z' > 'A'cho kết quả ở bước đầu tiên.

So sánh thứ hai 'Glow'và 'Glee'cần nhiều bước hơn vì các chuỗi được so sánh theo từng ký tự:

1. Gcũng giống như G.
2. lcũng giống như l.
3. o lớn hơn e. Dừng ở đây. Chuỗi đầu tiên lớn hơn.

## 4. So sánh các loại khác nhau

Khi so sánh các giá trị của các loại khác nhau, JavaScript sẽ chuyển đổi các giá trị thành số.

Ví dụ:

*alert(* *'2'* *>* *1* *);* *// true, string '2' becomes a number 2*

*alert(* *'01'* *==* *1* *);* *// true, string '01' becomes a number 1*

Đối với các giá trị boolean, true trở thành 1và false trở thành 0.

Ví dụ:

*alert(* *true* *==* *1* *);* *// true*

*alert(* *false* *==* *0* *);* *// true*

It is possible that at the same time:

- Two values are equal.

- One of them is `true` as a boolean and the other one is `false` as a boolean.

For example:

```js run

let a = 0;

alert( Boolean(a) ); // false

let b = "0";

alert( Boolean(b) ); // true

alert(a == b); // true!

```

From JavaScript's standpoint, this result is quite normal. An equality check converts

## 5. Bình đẳng nghiêm ngặt

Một kiểm tra bình đẳng thường xuyên ==có một vấn đề. Nó không thể phân biệt 0 với false:

*alert(* *0* *==* *false* *);* *// true*

Điều tương tự cũng xảy ra với một chuỗi rỗng:

*alert(* *''* *==* *false* *);* *// true*

Điều này xảy ra vì các toán hạng thuộc các loại khác nhau được chuyển đổi thành số bởi toán tử đẳng thức ==. Một chuỗi rỗng, giống như false, trở thành số không.

Phải làm gì nếu chúng ta muốn phân biệt 0với false?

Toán tử đẳng thức nghiêm ngặt ===kiểm tra đẳng thức mà không cần chuyển đổi kiểu.

Nói cách khác, nếu avà bthuộc các loại khác nhau, thì a === bngay lập tức trả về falsemà không cần chuyển đổi chúng.

Hãy thử nó:

*alert(* *0* *===* *false* *);* *// false, because the types are different*

Ngoài ra còn có một toán tử "không bình đẳng nghiêm ngặt" !==tương tự như !=.

Toán tử đẳng thức nghiêm ngặt dài hơn một chút để viết, nhưng làm cho nó rõ ràng những gì đang xảy ra và để lại ít lỗi hơn.