

# Chapter 6

## DOM – AJAX - jQuery

---



Lectured by:  
Nguyễn Hữu Hiếu

**DOM:** Có vẻ như chương này sẽ giải thích về DOM là gì, cách nó đại diện cho cấu trúc của một trang web HTML, và làm sao để JavaScript có thể tương tác và thay đổi cái cấu trúc đó. Chắc là sẽ có mấy cái khái niệm như node, element, attribute các kiểu.

**AJAX:** Phần này chắc sẽ nói về cách trang web có thể gửi và nhận dữ liệu từ server mà không cần phải tải lại toàn bộ trang. AJAX giúp tạo ra những trang web động và tương tác tốt hơn. Có thể sẽ có ví dụ về cách dùng XMLHttpRequest hoặc các cách khác để thực hiện AJAX.

**jQuery:** jQuery là một thư viện JavaScript rất phổ biến, giúp đơn giản hóa việc thao tác với DOM, xử lý sự kiện, tạo hiệu ứng và làm việc với AJAX. Chương này có lẽ sẽ giới thiệu về cú pháp cơ bản của jQuery và cách nó giúp mình viết code JavaScript dễ dàng hơn.

# Anonymous functions

# Anonymous functions

```
function(paramName, ..., paramName) {  
    statements;  
}
```

- anonymous functions can be stored, passed, returned

```
> function foo(x, f) { return f(x) + 1; }  
> foo(5, function(n) { return n * n; })
```

26

JavaScript cho phép mình tạo ra những cái hàm mà không có tên. Mấy cái hàm đó gọi là "hàm ẩn danh" (anonymous functions).

Cú pháp:

Cú pháp của hàm ẩn danh cũng giống như hàm bình thường, chỉ có điều là mình không viết cái tên hàm thôi:

```
function(paramName1, paramName2, ...){  
    // Các câu lệnh của hàm  
}
```

Tại sao lại dùng hàm ẩn danh?

Mặc dù không có tên, nhưng hàm ẩn danh vẫn có thể được sử dụng. Cái hay của nó là mình có thể:

Gán cho biến: Mình có thể gán một hàm ẩn danh cho một biến, và sau đó gọi nó thông qua biến đó:

```
var binhPhuong = function(n) {  
    return n * n;  
};  
console.log(binhPhuong(5)); // Kết quả: 25
```

Truyền như tham số: Mình có thể truyền một hàm ẩn danh vào một hàm khác như một tham số. Cái này hay dùng trong các hàm callback, ví dụ như trong hàm setTimeout() hoặc các hàm xử lý sự kiện.

```
setTimeout(function() {  
    console.log("Đã hết 3 giây!");  
}, 3000);
```

Trả về từ hàm khác: Một hàm có thể trả về một hàm ẩn danh.

```
function foo(x, f) {  
    return f(x) + 1;  
}  
foo(5, function(n) {  
    return n * n;  
}); // Kết quả: 26 [cite: 3]
```

Ở đây, hàm foo nhận vào một số x và một hàm f. Khi gọi hàm foo, mình truyền vào một hàm ẩn danh để tính bình phương. Hàm foo gọi cái hàm ẩn danh đó với x = 5, lấy kết quả (25) cộng thêm 1 và trả về 26.

Tóm lại:

Hàm ẩn danh là một hàm không có tên. Nó rất linh hoạt vì có thể gán cho biến, truyền vào hàm khác hoặc trả về từ hàm. May mắn là nó dùng nhiều trong JavaScript hiện đại.

# Two ways to declare a function

- The following are equivalent:

```
function name(params) {  
  statements;  
}
```

```
var name = function(params) {  
  statements;  
}
```

Cách này sử dụng một biểu thức để gán một hàm ẩn danh cho một biến

```
var squared = function(x) {  
  return x*x;  
};
```

# Array higher-order functions \*

.every( <i>function</i> )	accepts a function that returns a boolean value and calls it on each element until it returns false
.filter( <i>function</i> )	accepts a function that returns a boolean; calls it on each element, returning a new array of the elements for which the function returned true
.forEach( <i>function</i> )	applies a "void" function to each element
.map( <i>function</i> )	applies function to each element; returns new array
.reduce( <i>function</i> ) .reduce( <i>function</i> , <i>initialValue</i> ) .reduceRight( <i>function</i> ) .reduceRight( <i>function</i> , <i>initialValue</i> )	accepts a function that accepts pairs of values and combines them into a single value; calls it on each element starting from the front, using the given <i>initialValue</i> (or element [0] if not passed)  reduceRight starts from the end of the array
.some( <i>function</i> )	accepts a function that returns a boolean value and applies it to each element until it returns true

Mấy hàm này nhận vào một hàm khác làm tham số (callback function) để xử lý các phần tử trong mảng

.**every(function)**: Cái hàm này nó sẽ duyệt qua từng phần tử trong mảng. Với mỗi phần tử, nó sẽ gọi cái hàm mà mà truyền vào. Cái hàm này phải trả về giá trị true hoặc false. Hàm .every() sẽ trả về true nếu tất cả các phần tử trong mảng đều làm cho cái hàm kia trả về true. Chỉ cần có một phần tử làm cho hàm kia trả về false là .every() trả về false luôn. Nó giống như kiểu "tất cả phải đạt yêu cầu".

.**filter(function)**: Hàm này cũng duyệt qua từng phần tử trong mảng và gọi cái hàm mà mà truyền vào. Cái hàm này cũng phải trả về true hoặc false. Hàm .filter() sẽ tạo ra một mảng mới chỉ chứa những phần tử mà khi truyền vào cái hàm kia thì hàm đó trả về true. Nó giống như một cái bộ lọc, chỉ giữ lại những cái "đạt chuẩn".

.**forEach(function)**: Hàm này thì đơn giản hơn. Nó cũng duyệt qua từng phần tử trong mảng và gọi cái hàm mà mà truyền vào cho mỗi phần tử. Nhưng mà cái hàm này thường dùng để thực hiện một hành động gì đó với mỗi phần tử (ví dụ như in ra màn hình), chứ nó không trả về giá trị gì quan trọng (thường là undefined). Nó giống như kiểu "làm cái này cho từng cái".

Tóm lại, mấy cái hàm này rất là mạnh mẽ và thường được dùng để xử lý mảng một cách ngắn gọn và hiệu quả trong JavaScript. Thay vì phải viết vòng lặp for rườm rà, mình có thể dùng mấy hàm này để thực hiện các thao tác phức tạp trên mảng chỉ với vài dòng code.

.**map(function)**: Hàm này cũng duyệt qua từng phần tử và gọi cái hàm mà mà truyền vào. Nhưng khác với .forEach(), cái hàm này phải trả về một giá trị nào đó. Hàm .map() sẽ tạo ra một mảng mới chứa các giá trị mà cái hàm kia trả về sau khi được gọi cho từng phần tử. Nó giống như kiểu "biến đổi từng cái thành một cái mới".

.**reduce(function)** và .**reduce(function, initialValue)**: Hàm này hơi "khoai" một chút. Nó dùng để "gộp" các phần tử trong mảng lại thành một giá trị duy nhất. Nó sẽ lấy hai giá trị (giá trị tích lũy và giá trị hiện tại) và dùng cái hàm mà mà truyền vào để kết hợp chúng lại.

Nếu mà không truyền initialValue (giá trị ban đầu), nó sẽ lấy phần tử đầu tiên của mảng làm giá trị ban đầu.

Nếu mà truyền initialValue, nó sẽ dùng cái giá trị đó làm giá trị ban đầu. Nó sẽ duyệt qua mảng từ trái sang phải.

.**reduceRight(function)** và .**reduceRight(function, initialValue)**: Hàm này hoạt động giống như .reduce() nhưng nó duyệt qua mảng từ phải sang trái.

.**some(function)**: Hàm này cũng duyệt qua từng phần tử và gọi cái hàm mà mà truyền vào (hàm này phải trả về true hoặc false). Hàm .some() sẽ trả về true nếu ít nhất một phần tử trong mảng làm cho cái hàm kia trả về true. Chỉ cần có một phần tử "đạt yêu cầu" là .some() trả về true luôn. Nếu duyệt hết mảng mà không có phần tử nào làm cho hàm kia trả về true thì .some() trả về false. Nó giống như kiểu "có cái nào đạt yêu cầu không?".

# Examples

```
> var a = [1, 2, 3, 4, 5];
> a.map(function(x) { return x*x; })
1,4,9,16,25
```

```
> a.filter(function(x) { return x % 2 == 0; })
2,4
```

# Underscore.js

Underscore is a JavaScript library that provides a whole mess of useful functional programming helpers without extending any built-in objects.

<https://underscorejs.org/>

Underscore.js là một thư viện JavaScript cung cấp một loạt các hàm hỗ trợ lập trình hàm (functional programming helpers) mà không mở rộng bất kỳ đối tượng tích hợp sẵn nào của JavaScript.

Nói một cách đơn giản, Underscore.js là một công cụ giúp mình viết code JavaScript theo kiểu "hàm" hơn, ngắn gọn hơn và dễ đọc hơn. Nó cung cấp nhiều hàm tiện ích để làm việc với mảng, đối tượng, và các kiểu dữ liệu khác.

Ví dụ, mày nhớ mấy cái hàm higher-order function của mảng mà mình vừa nói không? Underscore.js nó cũng có mấy cái hàm tương tự, thậm chí còn nhiều hơn và mạnh mẽ hơn. Thay vì phải viết vòng lặp for thủ công, mình có thể dùng các hàm của Underscore.js để xử lý dữ liệu một cách dễ dàng.

Ngoài ra, Underscore.js còn cung cấp các hàm để làm việc với đối tượng (ví dụ: lấy ra các key, các value, gộp các đối tượng), các hàm để xử lý chuỗi, và nhiều hàm tiện ích khác nữa.

Cái hay của Underscore.js là nó không "sửa đổi" gì mấy cái đối tượng JavaScript sẵn có (như Array, Object). Nó tạo ra các hàm mới để mình dùng thôi. Điều này giúp tránh được các vấn đề không mong muốn khi làm việc với các thư viện khác.

# DOM

## Document Object Model

DOM là gì: DOM (Document Object Model) nó giúp JavaScript có thể "nhìn thấy" và tương tác với các thành phần của một trang web HTML. Nó coi mọi thứ trong trang web như là các đối tượng (object).

DOM và DHTML: Trang web động (Dynamic web pages) sử dụng JavaScript và DOM để thay đổi nội dung và giao diện của trang web mà không cần tải lại trang. DHTML (Dynamic HTML) là một thuật ngữ cũ hơn để chỉ các công nghệ này.

DOM nodes và DOM tree: Các thành phần HTML (như thẻ, thuộc tính, văn bản) được biểu diễn trong DOM dưới dạng các "node" (nút). Các node này được sắp xếp theo một cấu trúc hình cây (DOM tree).

Duyệt, chỉnh sửa và thay đổi DOM nodes: JavaScript có thể dùng DOM để di chuyển qua các node trong DOM tree, cũng như chỉnh sửa nội dung, thuộc tính và cấu trúc của chúng.

Chỉnh sửa text nodes: Text node là các node chứa nội dung văn bản trong các phần tử HTML. JavaScript có thể truy cập và thay đổi nội dung của các node này.

Truy cập, chỉnh sửa và thay đổi thuộc tính của phần tử: Các phần tử HTML có các thuộc tính (attributes) để quy định các thông tin bổ sung (ví dụ: `src` của thẻ `<img>`, `href` của thẻ `<a>`). JavaScript có thể truy cập, chỉnh sửa hoặc thêm/xóa các thuộc tính này.

# DOM & DHTML

---

- Dynamic web pages with JavaScript and DOM
  - DHTML (Dynamic HTML)
- DOM nodes and DOM tree
- Traversing, editing and modifying DOM nodes
- Editing text nodes
- Accessing, editing and modifying elements' attributes

# DOM Concept

- DOM makes all components of a web page accessible
  - HTML elements
  - their attributes
  - text
- They can be created, modified and removed with JavaScript

DOM giúp JavaScript có thể truy cập vào mọi thành phần của một trang web.

Các thành phần đó bao gồm:

Các phần tử HTML (ví dụ: thẻ `<div>`, `<p>`, `<a>`).

Các thuộc tính của chúng (ví dụ: `id`, `class`, `src`, `href`).

Nội dung văn bản bên trong các phần tử.

Với JavaScript, mình có thể tạo mới, chỉnh sửa và xóa các thành phần này.

# DOM Objects

- DOM components are accessible as objects or collections of objects
- DOM components form a tree of nodes
  - relationship parent node – children nodes
  - **document** is the root node
- Attributes of elements are accessible as text
- Browsers can show DOM visually as an expandable tree
  - Firebug for Firefox
  - in IE -> Tools -> Developer Tools

## DOM Components là Objects

Cái quan trọng nhất là, DOM nó coi mọi thứ trong trang web HTML như là các đối tượng (objects) hoặc các tập hợp các đối tượng (collections of objects).

Ví dụ, mỗi thẻ HTML (như `<div>`, `<p>`, `<img>`, `<a>`) nó sẽ được biểu diễn trong DOM như là một đối tượng.

Các thuộc tính của thẻ (như id, class, src, href) cũng có thể được truy cập và thao tác như là các thuộc tính của đối tượng.

Nội dung văn bản bên trong một thẻ cũng được coi là một đối tượng.

## Cấu trúc cây của DOM

Các đối tượng DOM này nó không phải là một đống lộn xộn, mà nó được sắp xếp theo một cấu trúc hình cây (tree structure), gọi là DOM tree.

Trong cái cây này, mỗi đối tượng DOM được gọi là một node (nút).

Các node có quan hệ cha-con (parent-children relationship) với nhau. Ví dụ, trong một đoạn HTML như vầy:

```
<div>  
  <p>Đây là một đoạn văn bản.</p>  
</div>
```

Thì cái thẻ `<div>` là node cha, còn thẻ `<p>` là node con của nó.

Cái node gốc của cả cái cây DOM này là đối tượng document, nó đại diện cho toàn bộ trang web HTML.

## Truy cập thuộc tính

Các thuộc tính của các phần tử HTML có thể được truy cập và thay đổi như là text.

## Trình duyệt hỗ trợ xem DOM

Các trình duyệt web hiện đại đều có các công cụ cho phép mình xem cái DOM tree này một cách trực quan, dưới dạng một cái cây có thể mở ra thu vào được.

Ví dụ, trong Firefox thì có Firebug (giờ là tích hợp sẵn trong Developer Tools).

Trong Internet Explorer thì có Developer Tools (mở bằng cách nhấn F12).

Mấy cái công cụ này rất là hữu ích để mình debug và xem xét cấu trúc của trang web.

Tóm lại: DOM nó coi mọi thứ trong trang web như là các đối tượng, sắp xếp chúng theo cấu trúc cây, và cho phép JavaScript truy cập và thao tác với chúng. Các trình duyệt cũng có công cụ để mình xem cái cây DOM này.

# Example

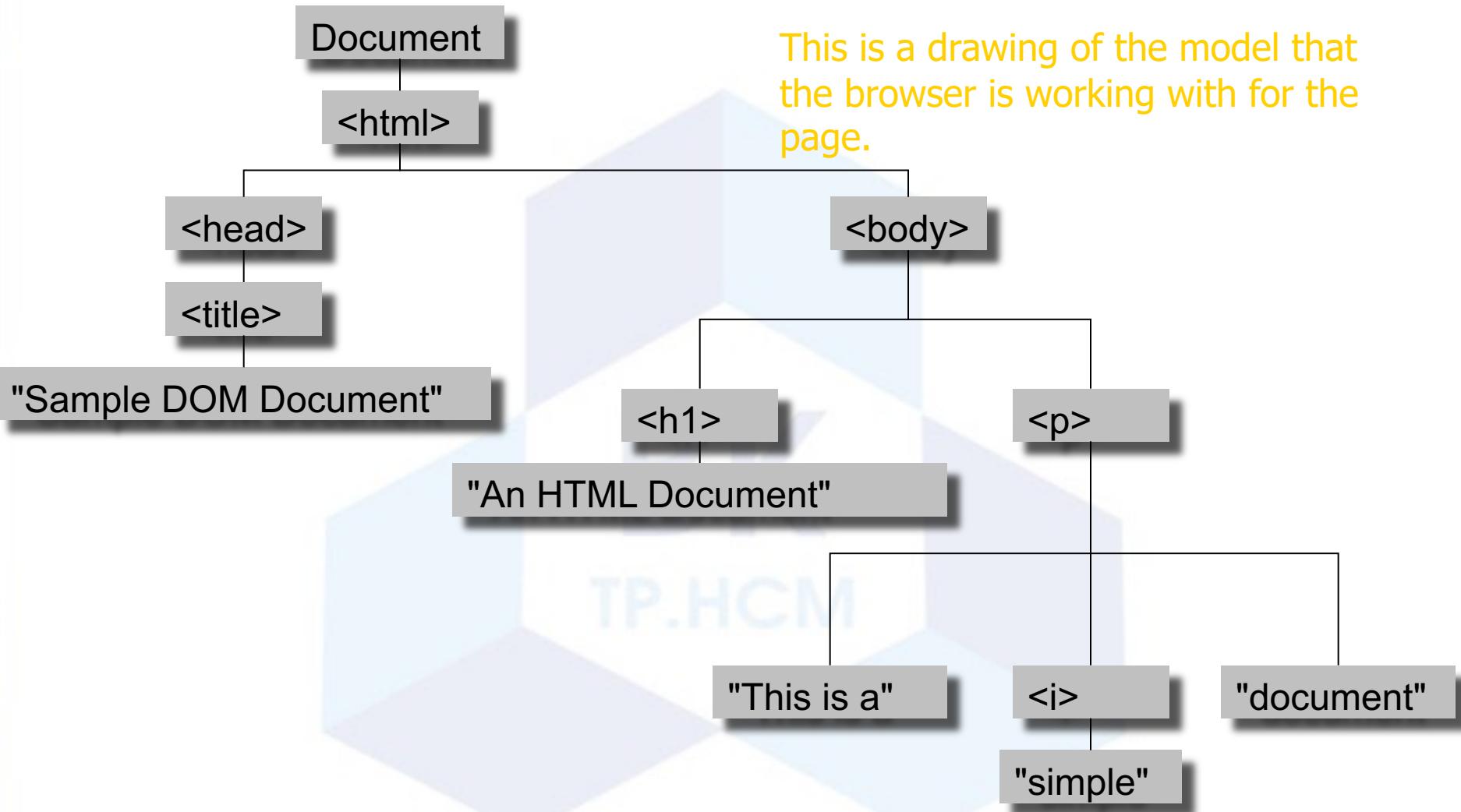
This is what the browser reads

```
<html>
  <head>
    <title>Sample DOM Document</title>
  </head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
  </body>
</html>
```

This is what the browser displays on screen.



# Example



Document: Đây là cái node gốc (root node) của cả cây DOM. Nó đại diện cho toàn bộ trang HTML.

<html>: Đây là phần tử gốc của trang HTML, nó là con trực tiếp của node Document. Tất cả các nội dung khác của trang đều nằm bên trong thẻ <html>.

<head>: Đây là một nhánh con của <html>. Nó chứa các thông tin meta về trang web, ví dụ như tiêu đề trang (<title>).

<title>: Đây là con của <head>. Nội dung bên trong thẻ <title> là "Sample DOM Document", đây là tiêu đề sẽ hiển thị trên tab trình duyệt.

<body>: Đây là nhánh con thứ hai của <html>. Nó chứa tất cả nội dung hiển thị của trang web.

<h1>: Đây là con của <body>. Nội dung bên trong là "An HTML Document", đây là tiêu đề chính của trang.

<p>: Đây là một con khác của <body>. Nó chứa một đoạn văn bản: "This is a".

<i>: Bên trong thẻ <p> có một thẻ <i> (in nghiêng), nó là con của <p>. Nội dung của nó là "simple".

<p>: Đây là một thẻ <p> khác, cũng là con của <body>. Nội dung của nó là "TP.HCM". Như mày thấy đó, cái hình này nó vẽ ra đúng cái cấu trúc lồng nhau của các thẻ HTML. Mỗi thẻ, mỗi đoạn văn bản đều được coi là một node trong cái cây này. Mỗi quan hệ cha-con giữa các node cũng được thể hiện rõ ràng. Trình duyệt sẽ dựa vào cái DOM tree này để hiển thị trang web và JavaScript sẽ dựa vào nó để tương tác và thay đổi nội dung cũng như cấu trúc của trang.

# DOM Standards

- W3C [www.w3.org](http://www.w3.org) defines the standards
- DOM Level 3 recommendation
  - [www.w3.org/TR/DOM-Level-3-Core/](http://www.w3.org/TR/DOM-Level-3-Core/)
- DOM Level 2 HTML Specification
  - [www.w3.org/TR/DOM-Level-2-HTML/](http://www.w3.org/TR/DOM-Level-2-HTML/)
  - additional DOM functionality specific to HTML, in particular objects for XHTML elements
- **But**, the developers of web browsers
  - **don't** implement all standards
  - implement some standards **differently**
  - implement some **additional features**

W3C định nghĩa các tiêu chuẩn: Tổ chức W3C (World Wide Web Consortium) là tổ chức quốc tế chuyên phát triển các tiêu chuẩn cho World Wide Web. Họ là người "quyết định" xem DOM phải hoạt động như thế nào.

Các phiên bản tiêu chuẩn DOM:

**DOM Level 3 Core:** Đây là một trong những phiên bản chính của DOM. Mày có thể xem chi tiết tại trang web này: [www.w3.org/TR/DOM-Level-3-Core/](http://www.w3.org/TR/DOM-Level-3-Core/)

**DOM Level 2 HTML:** Phiên bản này tập trung vào các tính năng DOM liên quan đến HTML. Chi tiết ở đây: [www.w3.org/TR/DOM-Level-2-HTML/](http://www.w3.org/TR/DOM-Level-2-HTML/)

Ngoài ra, còn có các phiên bản và các mô-đun khác của DOM, mỗi cái nó sẽ định nghĩa thêm các chức năng cụ thể.

**Đặc thù của HTML:** DOM Level 2 HTML bổ sung các đối tượng (objects) cho các phần tử XHTML (một phiên bản "khắt khe" hơn của HTML).

Vấn đề với trình duyệt:

Mặc dù có các tiêu chuẩn rõ ràng, nhưng các nhà phát triển trình duyệt web lại không phải lúc nào cũng tuân thủ 100%.

Có trình duyệt thì không triển khai hết tất cả các tiêu chuẩn.

Có trình duyệt thì triển khai nhưng theo một cách hơi khác so với tiêu chuẩn.

Có trình duyệt thì lại thêm vào một số tính năng riêng mà tiêu chuẩn không có.

Tóm lại:

DOM có các tiêu chuẩn được quy định bởi W3C, nhưng việc các trình duyệt triển khai các tiêu chuẩn này không phải lúc nào cũng thống nhất. Điều này gây ra một số khó khăn cho các nhà phát triển web, vì họ phải viết code sao cho nó chạy được trên nhiều trình duyệt khác nhau.

# Accessing Nodes by `id`

- Access to elements by their `id`
  - `document.getElementById(<id>)`
    - returns the element with `id <id>`
  - `id` attribute can be defined in each start tag
    - `div` element with `id` attribute can be used as an root node for a dynamic DOM subtree
    - `span` element with `id` attribute can be used as a dynamic inline element
- The preferred way to access elements

## Truy cập bằng ID:

`document.getElementById(<id>)`: Cái này là "ngon" nhất nè. Nó cho phép mình lấy về chính xác cái phần tử có cái ID mà mình chỉ định.

Hàm này trả về phần tử có ID là `<id>`.

Mỗi phần tử HTML có thể có một cái thuộc tính id.

Thẻ `<div>` với thuộc tính id thường được dùng làm "gốc" cho một cái cây DOM con (DOM subtree) động.

Thẻ `<span>` với thuộc tính id thì hay được dùng cho các phần tử inline động.

Nói chung, đây là cách "ưu tiên" để truy cập các phần tử, vì nó nhanh và chính xác.

Thẻ `<div>` làm "gốc" cho DOM subtree động:

`<div>` là block-level element: Điều này có nghĩa là thẻ `<div>` nó sẽ chiếm toàn bộ chiều ngang có sẵn trên trang web (hoặc trong phần tử cha của nó). Nó sẽ tạo ra một "khối" riêng biệt. Các phần tử block-level khác sẽ tự động xuống dòng khi gặp nó.

Dễ dàng quản lý một nhóm các phần tử: Vì nó tạo ra một khối riêng, nên mình có thể dễ dàng coi một thẻ `<div>` (với một cái id cụ thể) như là một "vùng chứa" cho nhiều phần tử HTML khác bên trong nó.

Tạo ra các section hoặc component động: Khi mình muốn thay đổi một phần lớn nội dung trên trang web một cách động (ví dụ: hiển thị một form mới, một danh sách sản phẩm, một thông báo...), thì việc có một thẻ `<div>` với id làm "gốc" sẽ rất tiện. Mình chỉ cần tìm tới cái div đó bằng document.

`getElementsByClassName()` rồi sau đó dùng JavaScript để thêm, sửa, hoặc xóa các phần tử bên trong nó. Mình có thể coi nó như là một "component" động của trang web.

Ví dụ: Mày tưởng tượng một trang web có một khu vực để hiển thị thông tin chi tiết của sản phẩm khi người dùng click vào một sản phẩm nào đó. Mình có thể có một thẻ div với id="product-details". Ban đầu, nó có thể rỗng hoặc chứa một thông báo "Chọn một sản phẩm để xem chi tiết". Khi người dùng click vào sản phẩm, JavaScript sẽ lấy thông tin chi tiết của sản phẩm từ server (chắc là dùng AJAX, mình sẽ nói tới sau) rồi tạo ra các phần tử HTML mới (ví dụ: tên sản phẩm, hình ảnh, giá...) và "nhét" chúng vào bên trong cái div có id="product-details" này.

Thẻ `<span>` dùng cho phần tử inline động:

`<span>` là inline element: Khác với `<div>`, thẻ `<span>` là một phần tử inline. Nó chỉ chiếm vừa đủ không gian cần thiết cho nội dung của nó thôi và không làm cho các phần tử khác bị xuống dòng. Nó thường được dùng để nhóm các phần tử nhỏ hoặc một đoạn text lại với nhau để áp dụng một số kiểu dáng hoặc hành động nào đó.

Thao tác động với một phần nhỏ của văn bản hoặc các phần tử inline: Khi mình muốn thay đổi một phần nhỏ của văn bản hoặc một vài phần tử inline một cách động (ví dụ: thay đổi màu chữ của một từ, thêm một icon nhỏ bên cạnh một chữ, hiển thị một tooltip nhỏ...), thì thẻ `<span>` với id là lựa chọn phù hợp.

Không phá vỡ luồng bố cục của trang: Vì nó là inline, việc thay đổi nội dung bên trong thẻ `<span>` thường sẽ không làm ảnh hưởng đến bố cục tổng thể của trang web nhiều như khi thay đổi nội dung của một thẻ div.

Ví dụ: Mày tưởng tượng một đoạn văn bản, và mình muốn làm nổi bật một từ nào đó khi người dùng hover chuột vào. Mình có thể bọc cái từ đó trong một thẻ span với một cái id (ví dụ: id="highlighted-word"). Sau đó, dùng JavaScript để bắt sự kiện mouseover trên cái span này và thay đổi màu chữ của nó.

# Other Access Methods

- Access by elements' tag
  - there are typically several elements with the same tag
  - `document.getElementsByTagName(<tag>)`
    - returns the collection of all elements whose tag is `<tag>`
    - the collection has a `length` attribute
    - an item in the collection can be reached by its index
  - e.g.
    - `var html = document.getElementsByTagName("html")[0];`
- Access by elements' `name` attribute
  - several elements can have the same name
  - `document.getElementsByName(<name>)`
    - returns the collection of elements with `name <name>`

## Truy cập bằng tên thẻ:

`document.getElementsByTagName(<tag>)`: Cái này thì nó trả về một "tập hợp" (collection) các phần tử có cùng cái tên thẻ mà mình chỉ định.

Ví dụ, nếu mà dùng `document.getElementsByTagName("p")` thì nó sẽ trả về tất cả các thẻ `<p>` trong trang.

Cái "tập hợp" này có một cái thuộc tính `length` để cho mình biết là có bao nhiêu phần tử trong đó.

Mình có thể truy cập từng phần tử trong "tập hợp" này bằng số thứ tự (`index`) của nó. Ví dụ, `document.`

`getElementsByName("html")[0]` sẽ trả về thẻ `<html>` đầu tiên (thường là thẻ `<html>` duy nhất) của trang.

## Truy cập bằng thuộc tính name:

`document.getElementsByName(<name>)`: Cái này trả về một "tập hợp" các phần tử có cùng cái thuộc tính `name` mà mình chỉ định.

Lưu ý là, nhiều phần tử có thể có cùng cái thuộc tính `name`.

```
<p>Đoạn văn bản thứ nhất.</p>
```

```
<p name="para">Đoạn văn bản thứ hai.</p>
```

```
<p>Đoạn văn bản thứ ba.</p>
```

```
<p name="para">Đoạn văn bản thứ tư.</p>
```

```
<script>
```

```
var paras = document.getElementsByName("para");
```

```
console.log("Số lượng phần tử có name='para': " + paras.length); // In ra: 2
```

```
console.log("Nội dung của phần tử thứ nhất có name='para': " + paras[0].textContent); // In ra: Đoạn văn bản thứ hai.
```

```
console.log("Nội dung của phần tử thứ hai có name='para': " + paras[1].textContent); // In ra: Đoạn văn bản thứ tư.
```

```
</script>
```

Tóm lại:

`getElementById()` là "trùm cuối", nhanh và chính xác.

`getElementsByTagName()` và `getElementsByName()` thì trả về một đống, nên phải dùng `index` để lấy từng cái.

# Traversing DOM tree

- Traversal through node properties
  - **childNodes** property
    - the value is a collection of nodes
      - has a **length** attribute
      - an item can be reached by its index
    - e.g. `var body = html.childNodes[1];`
  - **firstChild**, **lastChild** properties
  - **nextSibling**, **previousSibling** properties
  - **parentNode** property

`childNodes`: Cái này trả về một "tập hợp" (collection) các node con của một node. Cái "tập hợp" này có thuộc tính `length` để biết có bao nhiêu node con. Mình có thể truy cập từng node con bằng số thứ tự (`index`). Ví dụ: `html.childNodes[1]` (ví dụ trong tài liệu) có thể trả về thẻ `<body>` nếu nó là phần tử con thứ hai của thẻ `<html>`.

`firstChild, lastChild`: Cái này trả về node con đầu tiên và node con cuối cùng của một node.

`nextSibling, previousSibling`: Cái này trả về node anh/chị em kế tiếp và node anh/chị em trước đó của một node.

`parentNode`: Cái này trả về node cha của một node.

# Other Node Properties

- **nodeType** property
  - **ELEMENT\_NODE**: HTML element
  - **TEXT\_NODE**: text within a parent element
  - **ATTRIBUTE\_NODE**: an attribute of a parent element
    - attributes can be accessed another way
  - **CDATA\_SECTION\_NODE**
    - CDATA sections are good for unformatted text
- **nodeName** property
- **nodeValue** property
- **attributes** property
- **innerHTML** property
  - not standard, but implemented in major browsers
  - very useful
- **style** property
  - object whose properties are all style attributes, e.g., those defined in CSS

`nodeType`: Cái này nó cho mình biết cái "kiểu" (type) của một node là gì. Nó trả về một số nguyên (integer) đại diện cho kiểu node. Có mấy cái kiểu node quan trọng này:

`ELEMENT_NODE`: Cái này cho biết node đó là một phần tử HTML (ví dụ: thẻ `<div>`, `<p>`, `<a>`).

`TEXT_NODE`: Cái này cho biết node đó là một đoạn văn bản nằm bên trong một phần tử cha.

`ATTRIBUTE_NODE`: Cái này cho biết node đó là một thuộc tính của một phần tử cha (ví dụ: thuộc tính `id`, `class`). Tuy nhiên, thường thì người ta sẽ có cách khác để truy cập các thuộc tính, chứ ít khi dùng `nodeType` để lấy thuộc tính.

`CDATA_SECTION_NODE`: Cái này ít gặp hơn, nó đại diện cho một phần CDATA. CDATA là một khối văn bản mà trình phân tích cú pháp HTML sẽ không cố gắng diễn giải nó như là HTML. Nó thường được dùng cho mấy cái văn bản mà mình không muốn nó bị định dạng (unformatted text).

`nodeName`: Cái này trả về tên của một node.

Đối với element node, nó trả về tên thẻ (ví dụ: "DIV", "P", "A").

Đối với text node, nó trả về "#text".

Đối với attribute node, nó trả về tên của thuộc tính.

`nodeValue`: Cái này trả về giá trị của một node.

Đối với text node, nó trả về nội dung văn bản.

Đối với attribute node, nó trả về giá trị của thuộc tính.

Đối với element node, nó thường trả về null.

`attributes`: Cái này trả về một "tập hợp" (collection) các thuộc tính của một element node. Mình có thể dùng nó để truy cập và thay đổi các thuộc tính của phần tử.

`innerHTML`: Cái này không phải là một thuộc tính chuẩn, nhưng nó được hầu hết các trình duyệt phổ biến hỗ trợ. Nó cho phép mình lấy hoặc thiết lập nội dung HTML bên trong một phần tử. Cái này rất là hữu ích khi mình muốn thay đổi nội dung của một phần tử một cách nhanh chóng.

`style`: Cái này trả về một đối tượng (object) mà các thuộc tính của nó là các thuộc tính CSS. Ví dụ, mình có thể dùng `element.style.color = "red"` để thay đổi màu chữ của một phần tử.

```
<!DOCTYPE html>
<html>
<body>

<div id="myDiv">Đây là mt đon vǎn bn.</div>

<script>
  var divElement = document.getElementById("myDiv");
  var textNode = divElement.firstChild;
  var attributeNode = divElement.attributes[0]; // Ly thuc tinh đu tiên (id)

  console.log("Tên ca element node (div): " + divElement.nodeName);    // In ra: DIV
  console.log("Tên ca text node: " + textNode.nodeName);                // In ra: #text
  console.log("Tên ca attribute node (id): " + attributeNode.nodeName); // In ra: id
</script>

</body>
</html>

<!DOCTYPE html>
<html>
<body>

<div id="myDiv">Đây là một đoạn vǎn bản.</div>

<script>
  var divElement = document.getElementById("myDiv");
  var textNode = divElement.firstChild;
  var attributeNode = divElement.attributes[0]; // Lấy thuộc tính đầu tiên (id)

  console.log("Giá trị của element node (div): " + divElement.nodeValue); // In ra: null
  console.log("Giá trị của text node: " + textNode.nodeValue);           // In ra: Đây là một đoạn
vǎn bản.
  console.log("Giá trị của attribute node (id): " + attributeNode.nodeValue); // In ra: myDiv
</script>

</body>
</html>
```

# Accessing JS Object's Properties

- There are two different syntax forms to access object's properties in JS (
  - **<object>.<property>**
    - dot notation, e.g., `document.nodeType`
  - **<object>[<property-name> ]**
    - brackets notation, e.g., `document["nodeType"]`
    - this is used in `for-in` loops
- this works for properties of DOM objects, too

JavaScript có cho mình hai cách cú pháp khác nhau để "chạm" tới các thuộc tính của một object:

## 1. Dùng dấu chấm (.) - Dot notation:

Cú pháp: <object>.<property>

Ví dụ: document.nodeType

Cái này là cách phổ biến nhất và dễ đọc nhất.

Nó giống như kiểu mình nói "lấy cái thuộc tính property của cái đối tượng object".

## 2. Dùng dấu ngoặc vuông ([]) - Brackets notation:

Cú pháp: <object>[<property-name>]

Ví dụ: document["nodeType"]

Cái này thì linh hoạt hơn, đặc biệt là khi tên của thuộc tính là một biến hoặc có các ký tự đặc biệt.

Nó thường được dùng trong các vòng lặp for...in để duyệt qua các thuộc tính của một đối tượng.

```
var myObject = {  
    name: "John",  
    age: 30,  
    city: "New York"  
};
```

```
for (var key in myObject){  
    console.log("Thuộc tính: " + key + ", Giá trị: " + myObject[key]);  
}
```

Hai cách này đều dùng được cho các thuộc tính của các đối tượng DOM (DOM objects) luôn.

# Attributes of Elements

- Access through **attributes** property
  - **attributes** is an array
  - has a **length** attribute
  - an item can be reached by its index
  - an item has the properties **name** and **value**
  - e.g.
    - `var src = document.images[0].attributes[0].value;`
- Access through function **getAttribute(<name>)**
  - returns the value of attribute **<name>**
  - e.g.
    - `var src = document.images[0].getAttribute("src");`

## Thuộc tính của các phần tử (Attributes of Elements):

### Truy cập thông qua thuộc tính attributes:

attributes là một mảng (array) chứa tất cả các thuộc tính của một phần tử.

Nó có thuộc tính length để biết số lượng thuộc tính.

Mình có thể truy cập từng thuộc tính bằng số thứ tự (index).

Mỗi thuộc tính trong mảng attributes có hai thuộc tính quan trọng: name (tên của thuộc tính) và value (giá trị của thuộc tính).

Ví dụ: `var src = document.images[0].attributes[0].value;` (lấy giá trị của thuộc tính đầu tiên của thẻ `<img>` đầu tiên).

### Truy cập thông qua hàm `getAttribute(<name>)`:

Hàm này trả về giá trị của thuộc tính có tên là `<name>`.

Ví dụ: `var src = document.images[0].getAttribute("src");` (lấy giá trị của thuộc tính src của thẻ `<img>` đầu tiên).

# Text Nodes

Text node là gì?

Trong một trang HTML, những cái chữ mà mình viết bên trong các thẻ HTML đó, nó sẽ được DOM coi là các "text node".

Ví dụ:

<p>Đây là một đoạn văn bản.</p>

## ■ Text node

- can only be as a leaf in DOM tree
- it's `nodeValue` property holds the text
- `innerHTML` can be used to access the text

Thì cái cụm từ "Đây là một  
đoạn văn bản." nó là một text  
node.

## ■ Watch out:

- **There are many more text nodes than you would expect!**

Vị trí của text node trong DOM tree:

Text node chỉ có thể là "lá" (leaf) trong cây DOM. Tức là nó không thể có con. Nó luôn nằm ở cuối các nhánh của cây.

`nodeValue` của text node:

Như mình đã nói ở trên, cái thuộc tính `nodeValue` của một text node nó sẽ chứa chính cái nội dung văn bản đó.

`innerHTML` để lấy text:

Ngoài ra, mình cũng có thể dùng thuộc tính `innerHTML` của phần tử cha để lấy được cái text bên trong nó.

```
var pElement = document.querySelector("p"); // Lấy thẻ <p> đầu tiên  
var text = pElement.innerHTML; // Lấy nội dung HTML bên trong thẻ <p>  
console.log(text); // In ra: "Đây là một đoạn văn bản."
```

Lưu ý quan trọng:

Cái này hay bị "lừa" nè: Trong DOM nó có nhiều text node hơn mình tưởng tượng đó!

Lý do là vì mấy cái khoảng trắng, dòng mới trong code HTML nó cũng có thể được coi là text node.

Ví dụ:

```
<div>
```

Đây là một đoạn văn bản.

```
</div>
```

Trong ví dụ này, ngoài cái text node "Đây là một đoạn văn bản." thì còn có thể có các text node khác đại diện cho khoảng trắng và dòng mới trước và sau cái đoạn văn bản đó.

# Modifying DOM Structure

- **document.createElement(<tag>)**
  - creates a new DOM element node, with <tag> tag.
  - the node still needs to be inserted into the DOM tree
- **document.createTextNode(<text>)**
  - creates a new DOM text with <text>
  - the node still needs to be inserted into the DOM tree
- **<parent>.appendChild(<child>)**
  - inserts <child> node behind all existing children of <parent> node
- **<parent>.insertBefore(<child>, <before>)**
  - inserts <child> node before <before> child within <parent> node
- **<parent>.replaceChild(<child>, <instead>)**
  - replaces <instead> child by <child> node within <parent> node
- **<parent>.removeChild(<child>)**
  - removes <child> node from within <parent> node

### `document.createElement(<tag>):`

Hàm này dùng để tạo ra một cái phần tử DOM mới, với cái thẻ là `<tag>`.

Ví dụ: `document.createElement("div")` sẽ tạo ra một thẻ `<div>` mới.

Nhưng mà, cái node này nó mới chỉ được tạo ra thôi, nó chưa có nằm trong cái cây DOM của trang web đâu nha. Mình phải dùng mấy cái hàm khác để "nhét" nó vào nữa.

### `document.createTextNode(<text>):`

Hàm này dùng để tạo ra một cái node text mới, với nội dung là `<text>`.

Ví dụ: `document.createTextNode("Đây là một đoạn văn bản.")` sẽ tạo ra một node text chứa cái đoạn văn bản đó.

Giống như `createElement()`, cái node này cũng mới chỉ được tạo ra thôi, chưa nằm trong DOM tree.

### `<parent>.appendChild(<child>):`

Hàm này dùng để "nhét" một cái node con (`<child>`) vào cuối danh sách các node con của một node cha (`<parent>`).

```
var divElement = document.createElement("div");
```

```
var textNode = document.createTextNode("Đây là nội dung của div.");
```

```
divElement.appendChild(textNode); // Nhét cái textNode vào trong divElement
```

```
document.body.appendChild(divElement); // Nhét cái divElement vào trong thẻ <body>
```

### <parent>.insertBefore(<child>, <before>):

Hàm này cũng dùng để "nhét" node con, nhưng nó cho phép mình chỉ định vị trí cụ thể hơn. Nó sẽ nhét cái node con (<child>) vào trước một cái node con khác (<before>) đã có trong node cha (<parent>).

```
var newParagraph = document.createElement("p");
var textNode = document.createTextNode("Đây là đoạn văn bản mới.");
newParagraph.appendChild(textNode);
var firstParagraph = document.querySelector("p"); // Lấy thẻ <p> đầu tiên
document.body.insertBefore(newParagraph, firstParagraph); // Nhét cái newParagraph vào trước cái firstParagraph
```

### <parent>.replaceChild(<child>, <instead>):

Hàm này dùng để thay thế một cái node con đã có (<instead>) bằng một cái node con mới (<child>) trong node cha (<parent>).

Ví dụ:

```
var newHeading = document.createElement("h1");
var textNode = document.createTextNode("Tiêu đề mới");
newHeading.appendChild(textNode);
var oldHeading = document.querySelector("h2"); // Lấy thẻ <h2> đầu tiên
document.body.replaceChild(newHeading, oldHeading); // Thay thế thẻ <h2> bằng thẻ <h1>
```

### <parent>.removeChild(<child>):

Hàm này dùng để xóa một cái node con (<child>) khỏi node cha (<parent>).

Ví dụ:

```
var paragraphToRemove = document.querySelector("p"); // Lấy thẻ <p> đầu tiên
document.body.removeChild(paragraphToRemove); // Xóa thẻ <p> đó đi
```

# Modifying Node Attributes

- **<node>.setAttribute (<name>, <value>)**
  - sets the value of attribute **<name>** to **<value>**
  - e.g.
    - `document.images[0].setAttribute ("src", "keiki.jpg");`
- That's the standard
  - but it doesn't work in IE, there you have to use
    - **setAttribute (<name=value>)**
  - e.g.
    - `document.images[0].setAttribute ("src=\"keiki.jpg\"");`

**<node>.setAttribute(<name>, <value>);**

Hàm này dùng để đặt giá trị cho một thuộc tính của một node.

<node> là cái node mà mà muốn thay đổi thuộc tính.

<name> là tên của cái thuộc tính mà mà muốn thay đổi.

<value> là cái giá trị mới mà mà muốn đặt cho cái thuộc tính đó.

Ví dụ:

```
document.images[0].setAttribute("src", "keiki.jpg");
```

Cái này nó sẽ đặt giá trị của thuộc tính src (đường dẫn ảnh) của cái thẻ <img> đầu tiên thành "keiki.jpg".

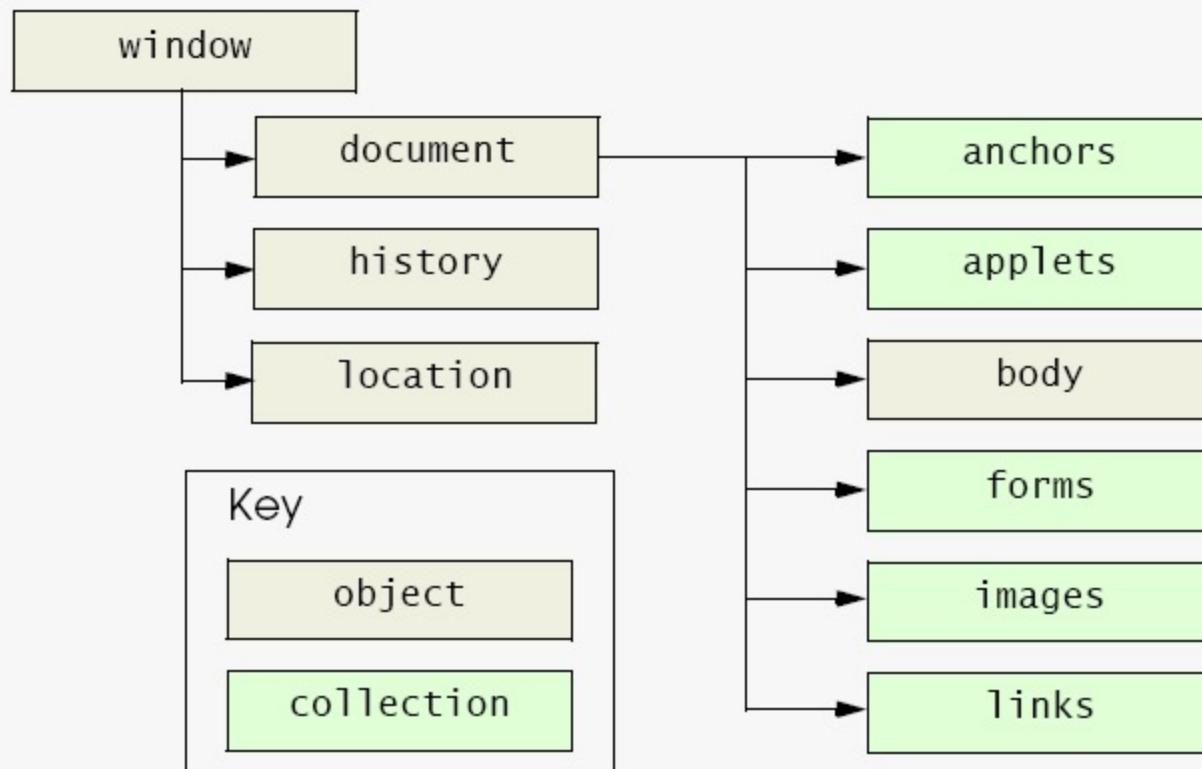
Vấn đề với Internet Explorer:

Cú pháp <node>.setAttribute(<name>, <value>) là cái chuẩn, nhưng mà nó không chạy được trên Internet Explorer (IE).

Để chạy được trên IE, mà phải dùng cú pháp này:

```
document.images[0].setAttribute("src=\\"keiki.jpg\\\"");
```

Cái giá trị của thuộc tính phải được đặt trong dấu nháy kép ("") và cái dấu bằng (=) nằm trong cái chuỗi luôn.



# Special DOM Objects

## ■ **window**

- the browser window
- new popup **window**s can be opened

## ■ **document**

- the current web page inside the **window**

## ■ **body**

- **<body>** element of the **document**

## ■ **history**

- sites that the user visited
- makes it possible to go back and forth using scripts

## ■ **location**

- URL of the **document**
- setting it goes to another page

window:

Cái này đại diện cho cái cửa sổ trình duyệt.  
Mình có thể dùng nó để mở ra các cửa sổ popup mới.

document:

Cái này đại diện cho cái trang web hiện tại đang hiển thị trong cửa sổ trình duyệt.

body:

Cái này đại diện cho thẻ **<body>** của trang web.

history:

Cái này chứa thông tin về các trang web mà người dùng đã ghé thăm.  
Mình có thể dùng nó để cho phép người dùng đi tới hoặc lui các trang web trước đó bằng script.

location:

Cái này chứa thông tin về cái URL của trang web hiện tại.  
Mình có thể dùng nó để chuyển người dùng đến một trang web khác.

# AJAX

# Asynchronous JavaScript And XML

---



# AJAX

- A lot of hype
  - It has been around for a while
  - Not complex
- Powerful approach to building websites
  - Think differently
- Allows for more interactive web applications
  - Gmail, docs.google.com, Flickr, ajax13, etc.

## AJAX:

Có vẻ như AJAX được nhắc đến khá nhiều.

Thực ra nó đã xuất hiện từ lâu rồi chứ không phải là cái gì mới mẻ.

Nó cũng không phức tạp lắm đâu.

Nhưng nó là một cách tiếp cận rất mạnh mẽ để xây dựng website.

Nó giúp mình "suy nghĩ khác" về cách làm web.

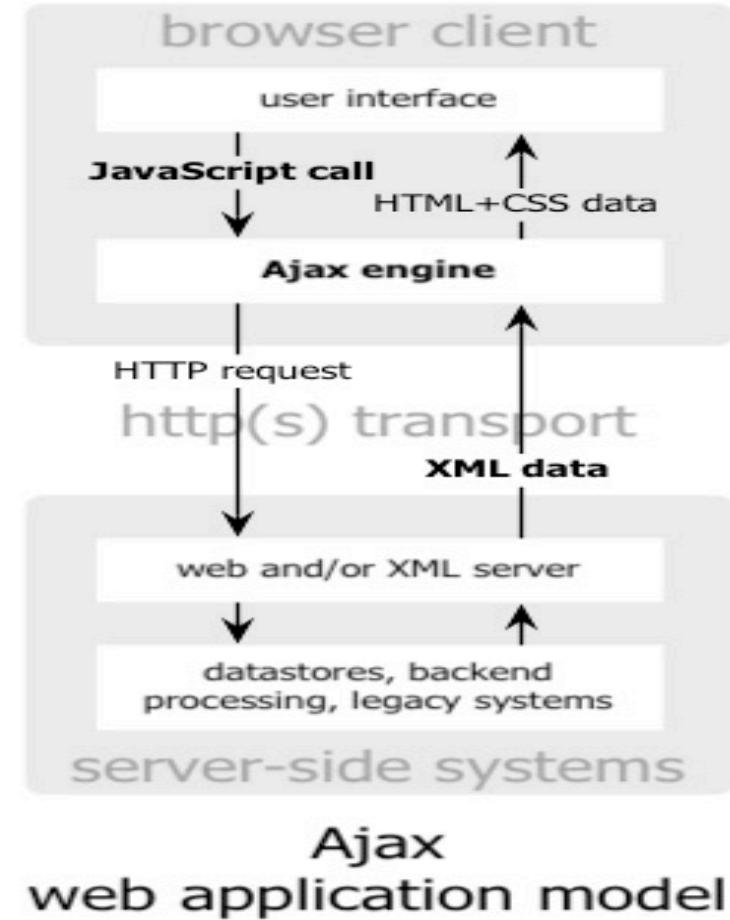
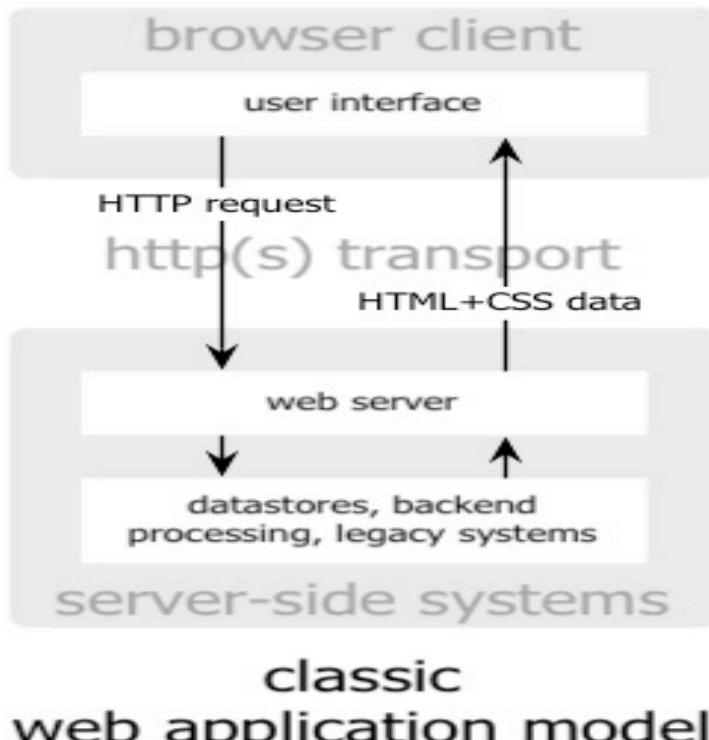
Nó cho phép tạo ra các ứng dụng web tương tác tốt hơn.

Mấy cái trang web như Gmail, Google Docs, Flickr,... đều dùng AJAX đó.

Nói một cách dễ hiểu

Nói một cách dễ hiểu, AJAX cho phép trang web của mình "nói chuyện" với server ở phía sau mà không cần phải tải lại toàn bộ trang. Điều này giúp cho trang web nhanh hơn, mượt mà hơn và người dùng có trải nghiệm tốt hơn.

# AJAX



# AJAX Technologies

---

- HTML
  - Used to build web forms and identify fields
- Javascript
  - Facilitates asynchronous communication and modification of HTML in-place
- DHTML - Dynamic HTML
  - Additional markup for modifying and updating HTML
- DOM - Document Object Model
  - Used via Javascript to work with both the structure of your HTML and also XML from the server

## HTML:

Dùng để xây dựng các form và xác định các trường dữ liệu trên trang web.

Ví dụ: Các thẻ `<form>`, `<input>`, `<button>`,...

## JavaScript:

Đóng vai trò quan trọng trong việc thực hiện giao tiếp bất đồng bộ (asynchronous communication) và thay đổi nội dung HTML trực tiếp trên trang.

Nó là cái "chất keo" kết nối mọi thứ lại với nhau trong AJAX.

## DHTML (Dynamic HTML):

Là một thuật ngữ cũ hơn để chỉ các kỹ thuật bổ sung cho việc sửa đổi và cập nhật HTML.

Thực ra, AJAX là một phần của DHTML.

## DOM (Document Object Model):

Được JavaScript sử dụng để làm việc với cả cấu trúc của HTML lẫn dữ liệu XML từ server trả về.

DOM giúp JavaScript "hiểu" và thao tác với trang web.

# The XMLHttpRequest Object

- Base object for AJAX
  - Used to make connections, send data, receive data, etc.
- Allows your javascript code to talk back and forth with the server all it wants to, without the user really knowing what is going on.
- Available in most browsers
  - But called different things

## XMLHttpRequest là gì?

Nó là đối tượng cơ bản để làm AJAX.

Mình dùng nó để tạo các kết nối, gửi dữ liệu, nhận dữ liệu,... Nói chung là mọi thứ liên quan đến việc "nói chuyện" với server.

Nó cho phép code JavaScript của mình giao tiếp qua lại với server mà người dùng không hề hay biết gì. Trang web vẫn mượt mà, không bị giật lag.

XMLHttpRequest có ở đâu?

Hầu hết các trình duyệt đều hỗ trợ nó.

Nhưng mà, mỗi trình duyệt có thể gọi nó bằng một cái tên khác.

# The XMLHttpRequest Object

```
[?]<script language="javascript" type="text/javascript">
```

```
var request;
```

```
function createRequest() {
```

```
    try {
```

```
        request = new XMLHttpRequest();
```

```
        if (request.overrideMimeType) {
```

```
            request.overrideMimeType('text/xml');
```

```
        }
```

```
    } catch (trymicrosoft) {
```

```
        try {
```

```
            request = new ActiveXObject("Msxml2.XMLHTTP");
```

```
    } catch (othermicrosoft) {
```

```
        try {
```

```
            request = new ActiveXObject("Microsoft.XMLHTTP");
```

```
    } catch (failed) {
```

```
        request = false;
```

```
    }
```

```
}
```

```
if (!request)
```

```
    alert("Error initializing XMLHttpRequest!");
```

```
}
```

```
</script>
```

Đoạn code này nó định nghĩa một hàm `createRequest()` để tạo ra một đối tượng `XMLHttpRequest`.

Tại sao lại phức tạp như vậy?

Là vì như ta đã nói ở trên, các trình duyệt khác nhau có thể hỗ trợ `XMLHttpRequest` theo những cách khác nhau.

Mấy cái trình duyệt cũ của Microsoft (Internet Explorer) nó dùng `ActiveXObject` thay vì `XMLHttpRequest`.

Đoạn code này nó cố gắng "bắt" các trường hợp khác nhau để tạo ra đối tượng `XMLHttpRequest` một cách chính xác, bất kể trình duyệt nào đang dùng.

`try...catch:`

Mấy cái `try...catch` là để "bắt" lỗi. Nếu một cái đoạn code trong `try` bị lỗi, thì code trong `catch` sẽ được chạy.

Ví dụ, nếu `new XMLHttpRequest()` bị lỗi (trên IE cũ), thì nó sẽ thử `new ActiveXObject("Msxml2.XMLHTTP")`, rồi `new ActiveXObject("Microsoft.XMLHTTP")` nếu cái trước đó cũng lỗi.

`overrideMimeType():`

Hàm này dùng để "ép" cái kiểu dữ liệu trả về từ server là `text/xml`. Cái này quan trọng khi mình muốn nhận dữ liệu XML từ server.

`if (!request):`

Nếu sau tất cả các cố gắng mà vẫn không tạo được đối tượng `XMLHttpRequest` (tức là `request` vẫn là `false`), thì nó sẽ hiện ra thông báo lỗi.

Nói chung là, cái đoạn code này nó là một cách "kinh điển" để tạo ra đối tượng `XMLHttpRequest` tương thích với nhiều trình duyệt khác nhau. Tuy nhiên, với các trình duyệt hiện đại ngày nay, thì mình thường chỉ cần dùng `new XMLHttpRequest()` là đủ rồi.

# Communicating

## ■ Steps

- Gather information (possibly from HTML form)
- Set up the URL
- Open the connection
- Set a callback method
- Send the request

```
function getCustomerInfo()
{
    var phone = document.getElementById("phone").value;
    var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);
    request.open("GET", url, true);
    request.onreadystatechange = updatePage;
    request.send(null);
}
```

## Các bước giao tiếp AJAX:

### Thu thập thông tin:

Bước đầu tiên là mình cần thu thập thông tin mà mình muốn gửi đi.

Thông tin này có thể đến từ các trường trong một cái form HTML (ví dụ: người dùng nhập tên, số điện thoại,...).

### Thiết lập URL:

Bước tiếp theo là mình cần thiết lập cái URL mà mình muốn gửi yêu cầu đến.

Cái URL này thường là đường dẫn đến một cái script (ví dụ: PHP, Python,...) trên server, nó sẽ xử lý cái yêu cầu của mình.

### Mở kết nối:

Mình dùng đối tượng XMLHttpRequest để mở một kết nối đến cái URL đã thiết lập.

Mình có thể chọn phương thức gửi dữ liệu (ví dụ: GET, POST).

### Thiết lập hàm callback:

Mình cần thiết lập một cái hàm callback để xử lý cái dữ liệu trả về từ server.

Hàm callback này sẽ được gọi khi server trả về dữ liệu.

### Gửi yêu cầu:

Cuối cùng, mình gửi cái yêu cầu đi.

```
var phone = document.getElementById("phone").value;: Dòng này lấy giá trị từ cái trường nhập liệu có id="phone" trên trang web.
```

```
var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);: Dòng này thiết lập cái URL để gửi yêu cầu. Nó gửi một yêu cầu GET đến cái script /cgi-local/lookupCustomer.php trên server, và nó truyền cái số điện thoại (phone) như là một tham số trong URL. Hàm escape() dùng để mã hóa cái số điện thoại cho an toàn khi truyền qua URL.
```

```
request.open("GET", url, true);: Dòng này mở một kết nối GET đến cái URL đã thiết lập. Cái true ở đây có nghĩa là mình muốn thực hiện giao tiếp bất đồng bộ (asynchronous).
```

```
request.onreadystatechange = updatePage;: Dòng này thiết lập cái hàm callback. Hàm updatePage sẽ được gọi khi có dữ liệu trả về từ server.
```

```
request.send(null);: Dòng này gửi cái yêu cầu đi. null ở đây có nghĩa là mình không gửi dữ liệu nào kèm theo trong phần thân của yêu cầu (vì mình đang dùng phương thức GET).
```

# Handling Server Responses

- When the server responds, your callback method will be invoked.
  - It is called at various stages of the process
  - Test readyState

```
function updatePage()
{
    if (request.readyState == 4) {
        if (request.status == 200) {
            // Handle the response
        } else
            alert("status is " + request.status);
    }
}
```

Khi server trả lời:

Khi server trả dũ liệu về sau khi mình gửi yêu cầu AJAX, thì cái hàm callback mà mình đã thiết lập trước đó sẽ được gọi.

Cái hàm callback này nó sẽ được gọi ở nhiều giai đoạn khác nhau trong quá trình xử lý.

Kiểm tra readyState:

Để biết chắc chắn là mình đã nhận được toàn bộ dữ liệu từ server và quá trình giao tiếp đã hoàn thành, mình cần kiểm tra cái thuộc tính readyState của đối tượng XMLHttpRequest.

readyState nó có thể có nhiều giá trị khác nhau, nhưng mình chỉ quan tâm đến giá trị 4. Giá trị 4 có nghĩa là quá trình giao tiếp đã hoàn thành và mình đã nhận được dữ liệu đầy đủ từ server.

function updatePage() { ... }: Đây là cái hàm callback.

if (request.readyState == 4) { ... }: Dòng này kiểm tra xem readyState có bằng 4 hay không. Nếu bằng 4, thì mình biết là đã nhận được dữ liệu xong xuôi.

if (request.status == 200) { ... }: Dòng này kiểm tra cái thuộc tính status của đối tượng XMLHttpRequest. status là mã trạng thái HTTP, nó cho mình biết là cái yêu cầu của mình có thành công hay không. Mã 200 có nghĩa là "OK", tức là yêu cầu thành công.

// Xử lý phản hồi: Nếu readyState là 4 và status là 200, thì mình sẽ viết code để xử lý cái dữ liệu trả về từ server.

else alert("status is " + request.status);: Nếu readyState là 4 nhưng status không phải là 200, thì có nghĩa là có lỗi xảy ra.

Đoạn code này sẽ hiện ra thông báo lỗi.

Nói chung là, khi server trả lời, hàm callback sẽ được gọi. Mình cần kiểm tra readyState để biết khi nào nhận được dữ liệu xong, và kiểm tra status để biết yêu cầu có thành công hay không. Nếu mọi thứ đều ổn, thì mình sẽ xử lý cái dữ liệu nhận được.

# HTTP Ready States

- 0: The request is uninitialized
  - Before calling open()
- 1: The request is set up, but hasn't been sent
  - Before calling send()
- 2: The request is sent and is being processed
  - Sometimes you can get content headers now
- 3: The request is being processed
  - The server hasn't finished with its response
- 4: The response is complete

readyState cho mình biết cái "trạng thái" hiện tại của quá trình giao tiếp AJAX.

## 0: The request is uninitialized

Trạng thái này có nghĩa là cái yêu cầu chưa được khởi tạo.

Nó là trạng thái ban đầu, trước khi mình gọi hàm open().

## 1: The request is set up, but hasn't been sent

Trạng thái này có nghĩa là mình đã thiết lập xong cái yêu cầu (gọi hàm open() rồi), nhưng chưa gửi nó đi (chưa gọi hàm send()).

## 2: The request is sent and is being processed

Trạng thái này có nghĩa là mình đã gửi yêu cầu đi rồi, và nó đang được xử lý.

Đôi khi, ở trạng thái này, mình có thể lấy được các header của phản hồi (content headers).

## 3: The request is being processed

Trạng thái này có nghĩa là server vẫn đang xử lý yêu cầu, chưa xong.

Mình có thể nhận được một phần dữ liệu trả về từ server ở trạng thái này.

## 4: The response is complete

Trạng thái này có nghĩa là quá trình giao tiếp đã hoàn thành.

Mình đã nhận được toàn bộ dữ liệu trả về từ server.

Nói chung là, readyState nó đi qua các trạng thái từ 0 đến 4, cho mình biết cái tiến trình của quá trình giao tiếp AJAX. Mình thường chỉ quan tâm đến trạng thái 4 thôi, vì đó là lúc mình biết chắc chắn là đã nhận được dữ liệu đầy đủ.

# The XMLHttpRequest Object

## ■ Methods

- **abort()**
  - cancel current request
- **getAllResponseHeaders()**
  - Returns the complete set of http headers as a string
- **getResponseHeader("headername")**
  - Return the value of the specified header
- **open("method", "URL", async, "uname", "passwd")**
  - Sets up the call
- **setRequestHeader("label", "value")**
- **send(content)**
  - Actually sends the request

# Method và Object của XMLHttpRequest

## Các phương thức (Methods):

### abort():

Dùng để hủy bỏ cái yêu cầu hiện tại.

Ví dụ, nếu mình thấy cái yêu cầu đang chạy lâu quá, hoặc mình không cần kết quả nữa, thì mình có thể dùng abort() để dừng nó lại.

### getAllResponseHeaders():

Trả về toàn bộ các HTTP header của phản hồi từ server dưới dạng một chuỗi (string).

HTTP header là các thông tin bổ sung được gửi kèm với dữ liệu trả về từ server, ví dụ như kiểu dữ liệu, mã hóa,...

### getResponseHeader("headername"):

Trả về giá trị của một header cụ thể.

headername là tên của cái header mà mình muốn lấy giá trị.

Ví dụ, nếu mình muốn lấy kiểu dữ liệu của dữ liệu trả về, mình có thể dùng getResponseHeader("Content-Type").

### open("method", "URL", "async", "uname", "passwd"):

Thiết lập cái yêu cầu AJAX.

method: Phương thức HTTP (ví dụ: "GET", "POST").

URL: Địa chỉ URL mà mình muốn gửi yêu cầu đến.

async: Có thực hiện bất đồng bộ hay không (true/false). Thường thì mình sẽ đặt là true để không làm "đứng" trang web.

uname, passwd: Tên người dùng và mật khẩu (nếu cần xác thực).

### setRequestHeader("label", "value"):

Thiết lập một HTTP request header.

label: Tên của header.

value: Giá trị của header.

Ví dụ, mình có thể dùng setRequestHeader("Content-Type", "application/json") để chỉ định là mình muốn gửi dữ liệu JSON.

### send(content):

Gửi cái yêu cầu đi.

content: Dữ liệu mình muốn gửi đi (có thể là null nếu không có dữ liệu).

# The XMLHttpRequest Object

## ■ Properties

- onreadystatechange
  - Event handler for an event that fires at every state change
- readyState
  - Returns the state of the object
- responseText
  - Returns the response as a string
- responseXML
  - Returns the response as XML - use W3C DOM methods
- status
  - Returns the status as a number - ie. 404 for “Not Found”
- statusText
  - Returns the status as a string - ie. “Not Found”

## Các thuộc tính (Properties):

### **onreadystatechange:**

Là một trình xử lý sự kiện (event handler) cho cái sự kiện readystatechange.

Sự kiện này xảy ra mỗi khi cái trạng thái của yêu cầu thay đổi.

Mình thường gán một hàm callback cho cái thuộc tính này để xử lý các phản hồi từ server.

### **readyState:**

Trả về trạng thái của đối tượng XMLHttpRequest.

Như mình đã nói ở trên, nó có các giá trị từ 0 đến 4.

### **responseText:**

Trả về phản hồi từ server dưới dạng một chuỗi (string).

Nếu dữ liệu trả về là text, HTML, hoặc JSON, thì mình sẽ lấy nó từ cái thuộc tính này.

### **responseXML:**

Trả về phản hồi từ server dưới dạng XML.

Nếu dữ liệu trả về là XML, thì mình sẽ dùng cái thuộc tính này để lấy nó dưới dạng một đối tượng DOM (Document Object Model).

### **status:**

Trả về mã trạng thái HTTP dưới dạng một số.

Ví dụ: 200 (OK), 404 (Not Found), 500 (Internal Server Error),...

### **statusText:**

Trả về mã trạng thái HTTP dưới dạng một chuỗi (string).

Ví dụ: "OK", "Not Found", "Internal Server Error",...

# Typical AJAX Flow

- Make the call
  - Gather information (possibly from HTML form)
  - Set up the URL
  - Open the connection
  - Set a callback method
  - Send the request
- Handle the response (in callback method)
  - When `request.readyState == 4` and `request.status == 200`
  - Get the response in either text or xml
    - `request.responseText` or `request.responseXML`
  - Process the response appropriately for viewing
  - Get the objects on the page that will change
    - `document.getElementById` or `document.getElementsByName`, etc.
  - Make the changes

một luồng AJAX điển hình sẽ diễn ra như sau:

### Thực hiện cuộc gọi (Make the call):

Thu thập thông tin: Đầu tiên, mình thu thập dữ liệu cần thiết (có thể từ một cái form HTML).

Thiết lập URL: Mình xác định cái URL mà mình muốn gửi yêu cầu đến.

Mở kết nối: Mình dùng XMLHttpRequest để mở một kết nối đến cái URL đó.

Thiết lập hàm callback: Mình định nghĩa một cái hàm callback để xử lý phản hồi từ server.

Gửi yêu cầu: Mình dùng hàm send() để gửi cái yêu cầu đi.

### Xử lý phản hồi (Handle the response): (trong hàm callback)

Kiểm tra trạng thái: Khi có phản hồi từ server, mình kiểm tra xem request.readyState có bằng 4 và request.status có bằng 200 hay không. Nếu cả hai điều kiện này đều đúng, có nghĩa là yêu cầu đã thành công.

Lấy dữ liệu trả về: Mình lấy dữ liệu trả về từ server, có thể là dưới dạng text (request.responseText) hoặc XML (request.responseXML).

Xử lý dữ liệu: Mình xử lý cái dữ liệu đó cho phù hợp với mục đích hiển thị.

Lấy các đối tượng trên trang: Mình dùng các hàm như document.getElementById() hoặc document.getElementsByName() để lấy ra các đối tượng HTML mà mình muốn thay đổi.

Thực hiện các thay đổi: Mình cập nhật nội dung hoặc thuộc tính của các đối tượng HTML đó để hiển thị cái dữ liệu mới nhận được từ server.

Nói chung là, AJAX nó hoạt động theo kiểu "gọi đi - trả về". Mình gửi một yêu cầu đến server, server xử lý rồi trả về dữ liệu, và mình dùng JavaScript để cập nhật trang web với cái dữ liệu đó.

# AJAX Response Handler

```
function updatePage()
{
    if (request.readyState == 4) {
        if (request.status == 200) {
            var response = request.responseText.split("|"); // "order|address"
            document.getElementById("order").value = response[0];
            document.getElementById("address").innerHTML = response[1];
        } else
            alert("status is " + request.status);
    }
}
```

# AJAX vs Fetch vs Axios

- Ajax uses XMLHttpRequest object to request data, and fetch is a method of window
- Ajax is based on native XHR development. The architecture of XHR itself is not clear, and there is an alternative to fetch.
- Fetch has a better and more convenient way of writing

```
1 fetch('http://example.com/movies.json')
2   .then((response) => {
3     return response.json();
4   })
5   .then((myJson) => {
6     console.log(myJson);
7   });

```

```
axios.get("https://example.com/movies.json")
  .then(response => console.log("response", response.data))
```

AJAX: Dùng cái đối tượng XMLHttpRequest để gửi yêu cầu dữ liệu. Cái này là cách làm "cổ điển" rồi.

Fetch: Là một cái phương thức có sẵn của window (trong trình duyệt). Nó được coi là một sự thay thế cho AJAX vì nó có cách viết "xịn sò" và tiện lợi hơn. AJAX thì dựa trên cái XMLHttpRequest gốc, mà kiến trúc của nó không được rõ ràng lắm.

Axios: Tao không thấy nó nhắc tới trong cái đoạn mà gửi, nhưng nó là một thư viện bên ngoài, cũng dùng để gửi yêu cầu HTTP và được nhiều người thích vì nó dễ dùng và có nhiều tính năng hay.

# await/async vs fetch/then

## For function that return a promise: fetch/axios

```
// Code 1
function fetchData() {
    fetch(url)
        .then(response => response.json())
        .then(json => console.log(json))
}

// Code 2
async function fetchData() {
    const response = await fetch(url);
    const json = await response.json();
    console.log(json);
}
```

so sánh hai cách viết code để xử lý mấy cái hàm mà trả về Promise (như Workspace hoặc axios).

### Cách 1: Dùng .then() (Code 1)

```
function fetchData() {
    fetch(url)
        .then(response => response.json())
        .then(json => console.log(json));
}
```

Ở đây, sau khi gọi Workspace(url), mình dùng .then() để xử lý cái response (phản hồi từ server). Cái response.json() nó cũng trả về một Promise, nên mình lại dùng .then() tiếp để lấy cái dữ liệu JSON thật sự (json) và in nó ra. Cách này thì hơi nhiều .then() nếu mình có nhiều bước xử lý bất đồng bộ liên tiếp.

### Cách 2: Dùng async/await (Code 2)

```
async function fetchData() {
    const response = await fetch(url);
    const json = await response.json();
    console.log(json);
}
```

Ở đây, mình khai báo cái hàm WorkspaceData là async. Bên trong hàm, mình dùng từ khóa await trước những cái hàm mà trả về Promise (như Workspace(url) và response.json()).

Khi gặp await, JavaScript nó sẽ "dừng" lại ở đó cho đến khi cái Promise đó được giải quyết (thành công hoặc thất bại). Sau đó, nó mới đi tiếp.

Cách này nhìn code nó giống như code đồng bộ (chạy tuần tự từng dòng), dễ đọc và dễ hiểu hơn nhiều so với việc dùng nhiều .then().

Tóm lại:

Cả hai cách đều làm cùng một việc là xử lý các tác vụ bất đồng bộ. Tuy nhiên, async/await nó giúp code mình trông gọn gàng và dễ đọc hơn, đặc biệt là khi mình có nhiều bước xử lý bất đồng bộ liên tiếp. Mặc dù vậy, về cơ bản thì async/await nó cũng chỉ là một cách "viết đẹp" hơn cho Promise thôi chứ không phải là một cái gì đó hoàn toàn mới.

# jQuery Javascript Library

---



# What is jQuery

- jQuery is a lightweight, open-source JavaScript library that simplifies interaction between HTML and JavaScript

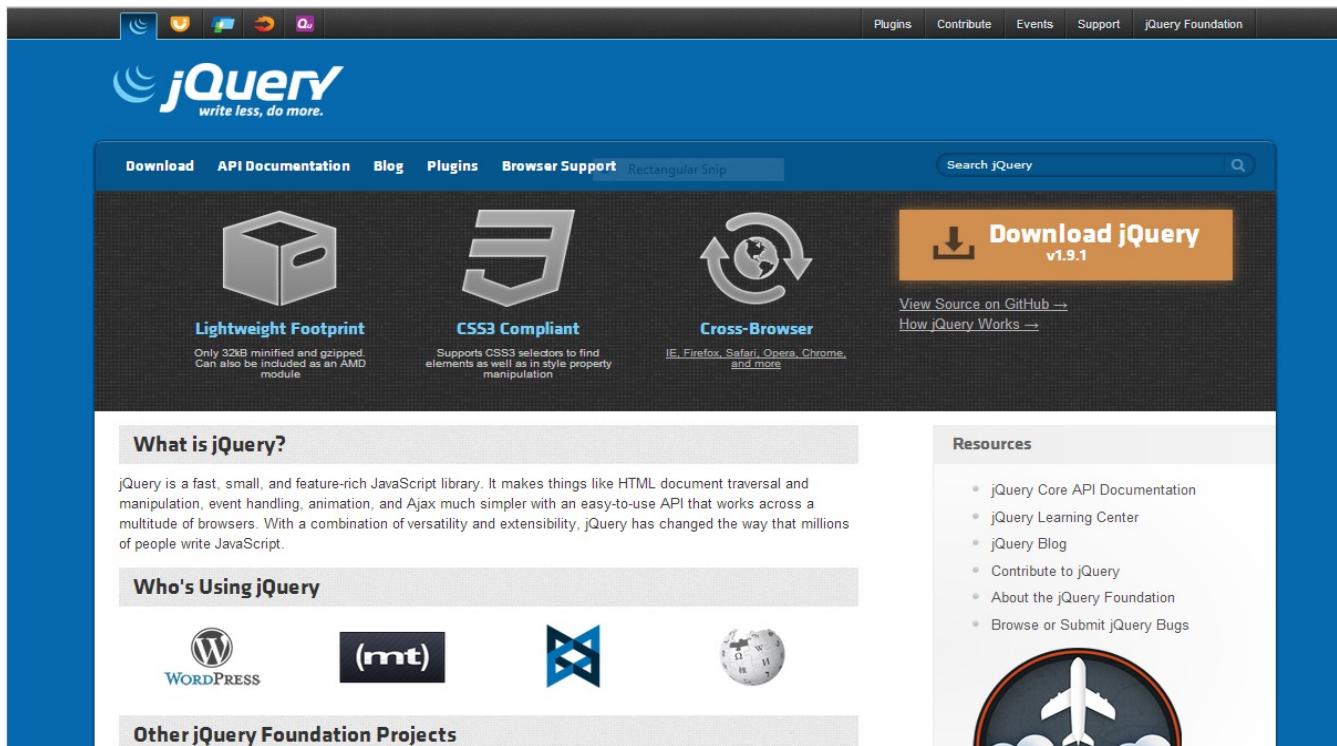
Một thư viện JavaScript gọn nhẹ, mã nguồn mở, giúp đơn giản hóa việc tương tác giữa HTML và JavaScript.
- It has a great community, great documentation, tons of plugins, and also adopted by Microsoft

Nó có một cộng đồng lớn mạnh, tài liệu tốt, rất nhiều plugin, và còn được Microsoft chấp nhận nữa.

# Download

Download the latest version from

<http://jquery.com>



Nhúng trực tiếp:

Mình dùng thẻ <script> để nhúng một file JavaScript vào trang HTML.

Cái file jquery.js này phải chứa một bản sao của cái code jQuery (đã được nén để cho nhẹ).

# Reference it in your markup

```
<script src="jquery.js"/>
```

jquery.js should contain a copy of the compressed production code

Nhúng từ Google:

Mình có thể lấy file jQuery từ các máy chủ của Google. Cách này có thể giúp trang web tải nhanh hơn vì có thể người dùng đã tải cái file này rồi khi vào một trang web khác.

## You can also reference it from Google

```
<script src="//ajax.googleapis.com/ajax/libs/  
    jquery/1.9.1/jquery.min.js">  
</script>
```

Or by another CDN (Content Delivery Network)

`var el = $("<div/>");` Cái này dùng để tạo ra các phần tử HTML "on the fly" (ngay tức thì). Ví dụ, cái dòng code này sẽ tạo ra một thẻ `<div>` mới.

## The Magic `$()` function

```
var el = $("“<div/>”)
```

Create HTML elements on the fly

# The Magic **\$()** function

```
$(window).width()
```

`$`(window).width(): Cái này dùng để thao tác với các phần tử DOM đã có. Ví dụ, cái dòng code này sẽ lấy chiều rộng của cửa sổ trình duyệt.

## Manipulate existing DOM elements

# The Magic **\$()** function

```
$(“div”).hide();
```

`$(“div”).hide();`: Cái này dùng để chọn các phần tử trong trang web. Ví dụ, cái dòng code này sẽ chọn tất cả các thẻ `<div>` và ẩn chúng đi..

Selects document elements  
(more in a moment...)

`$(function(){...});` hoặc `$(document).ready(function(){...});`: Cái này dùng để chạy code khi trang web đã sẵn sàng để lập trình. Cái cú pháp đầy đủ là `$(document).ready(function(){...});`; thì nên dùng hơn.

## The Magic `$()` function

```
$(function(){...});
```

Fired when the document is **ready** for programming.

Better use the **full** syntax:

```
$(document).ready(function(){...});
```

## Tạo phần tử HTML "on the fly":

Khi mà bạn viết `$(<div>) hoặc $(<span>)` hoặc nói chung là `$(tên thẻ HTML/)`, jQuery sẽ tạo ra một cái phần tử HTML mới trong bộ nhớ, nhưng nó chưa được thêm vào trang web thật đâu nha. Nó giống như bạn có một món đồ mới, nhưng chưa đặt nó vào đúng chỗ vậy đó.

Ví dụ:

```
var newDiv = $("<div>"); // Tạo một thẻ div mới  
newDiv.text("Đây là nội dung mới."); // Thêm nội dung  
vào cái div đó  
$("body").append(newDiv); // Thêm cái div đó vào cuối  
thẻ body của trang web
```

## Thao tác với các phần tử DOM đã có:

Khi mà bạn viết `$(window).width()` hoặc `$(document).height()` hoặc `$(selector).method()`, jQuery sẽ tìm kiếm các phần tử HTML đã có trên trang web dựa vào cái selector (ví dụ: tên thẻ, class, id...) và sau đó thực hiện một cái hành động gì đó lên chúng (ví dụ: lấy chiều rộng, lấy chiều cao, ẩn đi, hiện ra...).

Ví dụ:

```
var width = $(window).width(); // Lấy chiều rộng của  
cửa sổ trình duyệt  
console.log("Chiều rộng cửa sổ là: " + width);  
$("button").click(function() { // Khi người dùng click  
vào bất kỳ thẻ <button> nào  
    $(this).text("Đã click!"); // Thay đổi nội dung của cái  
button vừa click  
});
```

## Chọn các phần tử trong trang web:

Cái này là một trong những công dụng mạnh nhất của jQuery. Khi mà bạn viết `$(div)`, `$(".my-class")`, `$("#my-id")`, jQuery sẽ sử dụng một cú pháp giống như CSS để chọn ra các phần tử HTML mà bạn muốn thao tác.

Ví dụ:

```
 $("p").css("color", "blue"); // Chọn tất cả các thẻ <p> và đổi  
màu chữ thành xanh  
 $(".important").addClass("highlight"); // Chọn tất cả các  
phần tử có class là "important" và thêm class "highlight" vào  
 $("#main-title").hide(); // Chọn phần tử có id là "main-title"  
và ẩn nó đi
```

## Chạy code khi trang web đã sẵn sàng:

Khi mà bạn viết `$(function(){ ... })` hoặc `$(document).ready(function(){ ... })`, cái đoạn code bên trong cái hàm đó sẽ chỉ được chạy khi toàn bộ trang web HTML đã được tải xong và DOM đã được xây dựng hoàn chỉnh. Cái này rất quan trọng, vì nếu bạn cố gắng thao tác với một phần tử HTML mà nó chưa được tải lên trình duyệt thì sẽ bị lỗi.

Ví dụ:

```
$(document).ready(function(){  
    // Code ở đây sẽ chỉ chạy sau khi trang web đã tải xong  
    $("#loading-message").hide(); // Ẩn cái thông báo "Đang tải  
..." đi  
    $("#content").fadeIn(); // Hiển thị nội dung chính của trang  
});
```

# jQuery's programming philosophy is:

**GET >> ACT**

```
$(“div”).hide()
```

```
$(“<span/>”).appendTo(“body”)
```

```
$(“button”).click()
```

**GET (Chọn):** Đầu tiên, mình dùng jQuery để chọn ra những cái phần tử HTML mà mình muốn tác động tới. Cái này chính là cái hàm \$ mà mình đã nói ở trên đó. Mình có thể chọn bằng tên thẻ ("div"), bằng class ("my-class"), bằng id ("#my-id"), hoặc bằng nhiều cách khác nữa.

**ACT (Hành động):** Sau khi đã chọn được các phần tử rồi, mình sẽ thực hiện một cái hành động gì đó lên chúng. Ví dụ:

`$("div").hide();`: Chọn tất cả các thẻ <div> và ẩn chúng đi.

`$("<span/>").appendTo("body");`: Tạo một thẻ <span> mới và thêm nó vào cuối thẻ <body>.

`$("button").click();`: Chọn tất cả các thẻ <button> và thiết lập một cái hàm xử lý sự kiện khi người dùng click vào chúng.

Cái hay của jQuery là hầu như tất cả các hàm của nó đều trả về một đối tượng jQuery. Điều này cho phép mình viết code theo kiểu "fluent programming" (lập trình trôi chảy) và "chainability" (khả năng kết chuỗi).

## Almost every function returns jQuery, which provides a **fluent** programming interface and **chainability**:

Cái đoạn code này nó sẽ thực hiện 3 hành động liên tiếp lên tất cả các thẻ <div>:

Hiển thị chúng (show()).

Thêm class "main" vào chúng (addClass("main")).

Thay đổi nội dung HTML bên trong chúng thành "Hello jQuery" (html("Hello jQuery")).

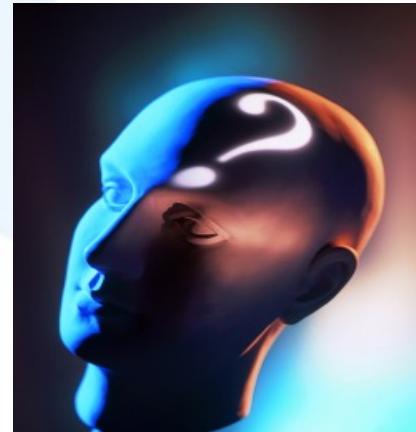
Vì mỗi hàm đều trả về đối tượng jQuery, nên mình có thể "chấm chấm chấm" liên tục như vậy, nhìn code rất là gọn gàng và dễ hiểu.

```
$("div").show()  
    .addClass("main")  
    .html("Hello jQuery");
```

# Three Major Concepts of jQuery



The `$()` function



Get > Act



Chainability

# All Selector

## All Selector

```
$(“*”) // find everything
```

Selectors return a pseudo-array of jQuery elements

Bộ chọn "All" \$(“\*”): Cái này nó chọn hết tất cả mọi thứ trên trang web, không chừa một ai.

# Basic Selectors

Bộ chọn cơ bản:

Theo tên thẻ (\$("div")): Cái này chọn tất cả các thẻ `<div>`. Ví dụ, nếu có đoạn HTML `<div>Hello jQuery</div>` thì nó sẽ chọn cái thẻ `<div>` đó.

Theo ID (\$("#usr")): Cái này chọn cái phần tử có ID là "usr". Ví dụ, nếu có thẻ `<span id="usr">John</span>` thì nó sẽ chọn cái thẻ `<span>` đó.

Theo class (\$.menu)): Cái này chọn tất cả các phần tử có class là "menu". Ví dụ, nếu có thẻ `<ul class="menu">Home</ul>` thì nó sẽ chọn cái thẻ `<ul>` đó.

By Tag:

```
$("div")
```

```
// <div>Hello jQuery</div>
```

By ID:

```
$("#usr")
```

```
// <span id="usr">John</span>
```

By Class:

```
$(".menu")
```

```
// <ul class="menu">Home</ul>
```

Yes, jQuery implements CSS Selectors!

# More Precise Selectors

`$("div.main")`: Cái này chọn các thẻ `<div>` mà đồng thời có class là "main".

`$("table#data")`: Cái này chọn thẻ `<table>` mà có ID là "data".

```
$("div.main") // tag and class
```

```
$("table#data") // tag and id
```

AND

# Combination of Selectors

`$("#content, .menu")`: Cái này chọn các phần tử có ID là "content" hoặc có class là "menu".

`("h1, h2, h3, div.content")`: Cái này chọn các thẻ `<h1>`, `<h2>`, `<h3>` hoặc các thẻ `<div>` có class là "content".

*// find by id + by class*

`$("#content, .menu")`

*// multiple combination*

`("h1, h2, h3, div.content")`

OR

a > b là chọn con b trực tiếp của cha a, a b là chọn con không quan trọng cấp độ, a + b là chọn b có a đứng ngay trước nó, #a ~ b là chọn b có cùng cấp độ với id = a.

Lập Trình Web

56

# Hierarchy Selectors

`$("table td")`: Cái này chọn tất cả các thẻ `<td>` là "hậu duệ" (descendants) của thẻ `<table>`. Tức là, nó chọn bất kỳ thẻ `<td>` nào nằm trong bất kỳ thẻ `<table>` nào, không quan trọng nó nằm ở cấp độ nào.

`$("tr > td")`: Cái này chọn tất cả các thẻ `<td>` là "con trực tiếp" (children) của thẻ `<tr>`. Tức là, nó chỉ chọn các thẻ `<td>` nằm ngay bên trong thẻ `<tr>`, không chọn các thẻ `<td>` nằm sâu hơn (ví dụ: trong một thẻ `<div>` nào đó nằm trong thẻ `<tr>`).

`$("table td")` // *descendants*

`$("tr > td")` // *children*

`$("label + input")` // *next*

`$("#content ~ div")` // *siblings*

`$("label + input")`: Cái này chọn thẻ `<input>` ngay sau thẻ `<label>`. Tức là, nó chọn thẻ `<input>` nào có thẻ `<label>` đứng ngay trước nó (là "anh/chị em liền kề").

`$("#content ~ div")`: Cái này chọn tất cả các thẻ `<div>` là "anh/chị em" (siblings) của thẻ có ID là "content". Tức là, nó chọn các thẻ `<div>` nào có cùng cấp độ với thẻ có ID là "content" trong cây DOM.

`$("table td")`: Chọn tất cả các thẻ `<td>` nằm bên trong bất kỳ thẻ `<table>` nào (gọi là "hậu duệ").

`$("tr > td")`: Chọn tất cả các thẻ `<td>` là con trực tiếp của thẻ `<tr>`. Dấu `>` có nghĩa là "con trực tiếp".

`$("label + input")`: Chọn thẻ `<input>` ngay sau thẻ `<label>`. Dấu `+` có nghĩa là "phần tử kế tiếp".

`$("#content ~ div")`: Chọn tất cả các thẻ `<div>` là anh chị em (cùng cấp độ) với thẻ có `id="content"`. Dấu `~` có nghĩa là "các phần tử anh chị em theo sau".

# Selection Index Filters

```
$("tr:first")      // first element  
$("tr:last")       // last element  
$("tr:lt(2)")      // index less than  
$("tr:gt(2)")      // index gr. than  
$("tr:eq(2)")      // index equals
```

`$("tr:first")`: Chọn thẻ `<tr>` đầu tiên.

`$("tr:last")`: Chọn thẻ `<tr>` cuối cùng.

`$("tr:lt(2)")`: Chọn tất cả các thẻ `<tr>` có chỉ mục nhỏ hơn 2 (tức là thẻ thứ nhất và thẻ thứ hai, vì chỉ mục bắt đầu từ 0). `lt` là "less than".

`$("tr:gt(2)")`: Chọn tất cả các thẻ `<tr>` có chỉ mục lớn hơn 2 (tức là thẻ thứ tư trở đi). `gt` là "greater than".

`$("tr:eq(2)")`: Chọn thẻ `<tr>` có chỉ mục bằng 2 (tức là thẻ thứ ba). `eq` là "equal".

# Visibility Filters

```
$(“div:visible”) // if visible  
$(“div:hidden”) // if not
```

`$(“div:visible”)`: Chọn tất cả các thẻ `<div>` đang được hiển thị trên trang.

`$(“div:hidden”)`: Chọn tất cả các thẻ `<div>` đang bị ẩn đi (ví dụ: bằng CSS `display: none` hoặc `visibility: hidden`)

# Attribute Filters

```
$("div[id]")          // has attribute  
$("div[dir='rtl']")    // equals to  
$("div[id^='main ']")   // starts with  
$("div[id$='name ']")   // ends with  
$("a[href*='msdn ']")   // contains
```

## Attribute Filters (Bộ lọc theo thuộc tính):

`$("div[id]")`: Chọn tất cả các thẻ `<div>` có thuộc tính id.

`$("div[dir='rtl']")`: Chọn tất cả các thẻ `<div>` có thuộc tính dir bằng với 'rtl' (right-to-left).

`$("div[id^='main']")`: Chọn tất cả các thẻ `<div>` có thuộc tính id bắt đầu bằng 'main'. Dấu ^= có nghĩa là "bắt đầu bằng".

`$("div[id$='name']")`: Chọn tất cả các thẻ `<div>` có thuộc tính id kết thúc bằng 'name'. Dấu \$= có nghĩa là "kết thúc bằng".

`$("a[href*='msdn']")`: Chọn tất cả các thẻ `<a>` có thuộc tính href chứa chuỗi 'msdn'. Dấu \*= có nghĩa là "chứa".

# Forms Selectors

`$("input:checkbox")`: Chọn tất cả các thẻ `<input>` có `type="checkbox"`.

`$("input:radio")`: Chọn tất cả các thẻ `<input>` có `type="radio"`.

`$("input:checkbox")` // checkboxes

`$("input:radio")` // radio buttons

`$(":button")` // buttons

`$(":text")` // text inputs

`$(":button")`: Chọn tất cả các thẻ `<button>`.

`$(":text")`: Chọn tất cả các thẻ `<input>` có `type="text"`.

# Forms Filters

`$("input:checked")`: Chọn tất cả các thẻ `<input>` (checkbox hoặc radio) đang được chọn.

`$("input:selected")`: Chọn tất cả các thẻ `<input>` (thường là trong thẻ `<select>`) đang được chọn.

```
$("input:checked") // checked
```

```
$("input:selected") // selected
```

```
$("input:enabled") // enabled
```

```
$("input:disabled") // disabled
```

`$("input:enabled")`: Chọn tất cả các thẻ `<input>` đang được kích hoạt (không bị disabled).

`$("input:disabled")`: Chọn tất cả các thẻ `<input>` đang bị vô hiệu hóa (disabled).

# Find Dropdown Selected Item

Tìm mục được chọn trong dropdown

```
<select id="cities">  
    <option value="1">Tel-Aviv</option>  
    <option value="2" selected="selected">Yavne</option>  
    <option value="3">Raanana</option>  
</select>
```

```
$("#cities option:selected").val()
```

```
$("#cities option:selected").text()
```

`$("#cities option:selected").val(): Lấy giá trị của  
option được chọn. -> 2`

`$("#cities option:selected").text(): Lấy nội dung  
text của option được chọn. -> "Yavne"`

# SELECTORS DEMO

`$("div").length`: Cái này sẽ trả về số lượng các thẻ `<div>` mà mà đã chọn được trên trang web. Nó là một cách rất tốt để kiểm tra xem cái bộ chọn (selector) của mà có hoạt động đúng như ý không.

## A Selector returns a pseudo array of jQuery objects

```
$("div").length
```

Returns **number** of selected elements.  
It is the best way to check selector.

# Getting a specific DOM element

```
$("div").get(2) or $("div")[2]
```

Returns a 2<sup>nd</sup> DOM element of the selection

`$("div").get(2)` hoặc `$("div")[2]`: Hai cái này đều dùng để lấy một phần tử DOM cụ thể từ cái "pseudo-array" (giả mảng) mà jQuery trả về khi mà chọn nhiều phần tử. Cái số 2 ở đây có nghĩa là mà muốn lấy cái phần tử thứ ba trong danh sách (vì index bắt đầu từ 0). Cái này nó trả về một đối tượng DOM thuần túy, không phải là đối tượng jQuery nữa nha.

# Getting a specific jQuery element

```
$("div").eq(2)
```

Returns a 2<sup>nd</sup> **jQuery** element of the selection

`$("div").eq(2)`: Cái này cũng dùng để lấy một phần tử cụ thể, nhưng nó trả về một đối tượng jQuery. Cái số 2 ở đây cũng là index, nên nó sẽ trả về phần tử thứ ba trong danh sách các thẻ `<div>` đã chọn. Thường thì người ta hay dùng `.eq()` hơn vì nó vẫn là đối tượng jQuery, nên mình có thể tiếp tục "chain" các method khác của jQuery lên nó.

`each(fn)`: Cái này dùng để duyệt qua từng phần tử trong cái danh sách mà mà đã chọn. Cái `fn()` là một cái hàm (function) mà mà sẽ định nghĩa để thực hiện một hành động gì đó trên mỗi phần tử. Bên trong cái hàm đó, cái từ khóa `this` nó sẽ đại diện cho cái phần tử DOM hiện tại đang được duyệt.

## each(`fn`) traverses every selected element calling `fn()`

```
var sum = 0;  
$("div.number").each(  
    function(){  
        sum += $(this).innerHTML;  
    } );  
                                // Lấy nội dung HTML, chuyển thành số và cộng vào  
                                biến sum
```

*this* – is a current DOM element

each(fn) cũng truyền vào index: Cái hàm mà mà truyền vào .each() nó còn có thể nhận thêm một tham số nữa là index (vị trí) của cái phần tử hiện tại trong danh sách.

## each(fn) also passes an indexer

```
$("table tr").each(  
    function(i){  
        if (i % 2) // Kiểm tra nếu index là số lẻ  
            $(this).addClass("odd");  
    })); // Thêm class "odd" vào cái hàng đó
```

`$(this)` – convert DOM to jQuery  
`i` - index of the current element

Traversing HTML (Duyệt HTML): jQuery cung cấp mấy cái method này để mình có thể "đi lại" giữa các phần tử trong cây DOM:

.next(expr): Lấy phần tử anh/chị em kế tiếp (ngay sau) của phần tử hiện tại. Cái expr là một bộ chọn tùy chọn để lọc.

.prev(expr): Lấy phần tử anh/chị em trước đó (ngay trước) của phần tử hiện tại. Cái expr cũng là bộ chọn tùy chọn.

.siblings(expr): Lấy tất cả các phần tử anh/chị em (cung cấp độ) của phần tử hiện tại. Cái expr là bộ chọn tùy chọn để lọc.

.children(expr): Lấy tất cả các phần tử con trực tiếp của phần tử hiện tại. Cái expr là bộ chọn tùy chọn để lọc.

.parent(expr): Lấy phần tử cha trực tiếp của phần tử hiện tại. Cái expr là bộ chọn tùy chọn để lọc.

## Traversing HTML

- **next(expr)** // next sibling
- **prev(expr)** // previous sibling
- **siblings(expr)** // siblings
- **children(expr)** // children
- **parent(expr)** // parent

Check for expression (.is()): Cái method .is() dùng để kiểm tra xem cái phần tử mà mình đã chọn có khớp với một cái bộ chọn nào đó hay không. Nó trả về true hoặc false.

## Check for expression

```
$("table td").each(function() {  
    if ($(this).is(":first-child")) {  
        // Kiểm tra xem cái thẻ <td> này có phải là con đầu tiên của thẻ cha không  
        $(this).addClass("firstCol");  
        // Nếu đúng thì thêm class "firstCol"  
    }  
});
```

Find in selected (.find()): Cái method .find() dùng để tìm kiếm các phần tử con (ở bất kỳ cấp độ nào) bên trong các phần tử mà mình đã chọn trước đó. Nó giống như mình "lọc" thêm một lần nữa trong phạm vi các phần tử đã chọn.

## Find in selected

```
// select paragraph and then find  
// elements with class 'header' inside  
$("p").find(".header").show();
```

// Chọn tất cả các thẻ <p>, sau đó tìm bên trong chúng các phần tử có class là "header" và hiển thị chúng

// equivalent to:

```
$(".header", $("p")).show();
```

# Advanced Chaining

Tạo một thẻ `<li>` chứa một thẻ `<span>`.

```
$("<li><span></span></li>") // li
  .find("span") // span
    .html("About Us") // span
    .andSelf() // span, li
      .addClass("menu") // span, li
      .end() // span
    .end() // li ?
  .appendTo("ul.main-menu");
```

.find("span"): Chọn cái thẻ `<span>` bên trong.

Đặt nội dung HTML của thẻ `<span>` thành "About Us".

Thêm cái thẻ `<li>` ban đầu vào cái tập hợp đang chọn (bây giờ là cả `<span>` và `<li>`).

.addClass("menu"): Thêm class "menu" vào cả thẻ `<span>` và thẻ `<li>`.

.end(): Trả về cái tập hợp các phần tử trước đó (trong trường hợp này là chỉ còn thẻ `<span>`).

.end(): Trả về cái tập hợp các phần tử trước đó nữa (trong trường hợp này là thẻ `<li>` ban đầu).

.appendTo("ul.main-menu"): Thêm cái thẻ `<li>` vào cuối cái thẻ `<ul>` có class là "main-menu".

# Get Part of Selected Result

```
$("div")
    .slice(2, 5)
    .not(".green")
    .addClass("middle");
```

Get Part of Selected Result (.slice(), .not()): Mấy cái method này giúp mình "cắt" hoặc "loại bỏ" bớt các phần tử trong cái danh sách đã chọn.

\$(“div”).slice(2, 5): Lấy một phần của danh sách các thẻ `<div>`, bắt đầu từ index 2 và kết thúc ở index 4 (không bao gồm index 5).

.not(“.green”): Loại bỏ khỏi danh sách các phần tử có class là “green”.

.addClass(“middle”): Thêm class “middle” vào các phần tử còn lại trong danh sách.

# HTML Manipulation

"HTML Manipulation" (thao tác với HTML) của jQuery

# Getting and Setting Inner Content

`$(“p”).html(“<div>Hello $!</div>”);` Cái này sẽ chọn tất cả các thẻ `<p>` trên trang và thay thế toàn bộ nội dung bên trong chúng bằng một thẻ `<div>` có nội dung là “Hello \$!”. Lưu ý là `.html()` có thể nhận cả mã HTML.

```
$(“p”).html(“<div>Hello $!</div>”);
```

*// escape the content of div.b*

```
$(“div.a”).text($(“div.b”).html()));
```

`$(“div.a”).text($(“div.b”).html());` Cái này sẽ chọn thẻ `<div>` có class là a, sau đó lấy nội dung HTML bên trong thẻ `<div>` có class là b (dùng `.html()`) và đặt nó làm nội dung text (dùng `.text()`) bên trong thẻ `<div>` có class là a. `.text()` sẽ tự động “escape” (mã hóa) các ký tự HTML đặc biệt để tránh bị lỗi hoặc tấn công XSS.

`$("input:checkbox:checked").val();`: Cái này sẽ chọn tất cả các thẻ `<input>` có `type="checkbox"` đang được checked (đánh dấu) và lấy giá trị (`value attribute`) của một trong số chúng (thường là cái đầu tiên tìm thấy). Nếu có nhiều checkbox được chọn, nó chỉ trả về giá trị của cái đầu tiên.

## Getting and Setting Values

```
// get the value of the checked checkbox  
$("input:checkbox:checked").val();
```

`$(":text[name='txt']").val("Hello");`: Cái này sẽ chọn thẻ `<input>` có `type="text"` và thuộc tính `name` là "txt", sau đó đặt giá trị của nó thành "Hello".

```
// set the value of the textbox  
$(":text[name='txt']").val("Hello");
```

```
// select or check lists or checkboxes  
$("#lst").val(["NY","IL","NS"]);
```

`$("#lst").val(["NY","IL","NS"]);`: Cái này thường dùng cho các thẻ `<select>` (dropdown list) hoặc các thẻ `<input>` có `type="checkbox"` hoặc `type="radio"` cho phép chọn nhiều giá trị. Nó sẽ chọn cái phần tử có id="lst" và thiết lập giá trị được chọn của nó thành một mảng các giá trị ["NY", "IL", "NS"]. Ví dụ, nếu #lst là một thẻ `<select>` có `multiple attribute`, nó sẽ chọn các option có `value` là "NY", "IL", và "NS".

`$(“p”).removeClass(“blue”).addClass(“red”);`: Cái này sẽ chọn tất cả các thẻ `<p>`, xóa class "blue" (nếu có) và thêm class "red" vào chúng.

`$(“div”).toggleClass(“main”);`: Cái này sẽ chọn tất cả các thẻ `<div>`. Nếu chúng đang có class "main" thì nó sẽ xóa đi, còn nếu chưa có thì nó sẽ thêm vào. Nó giống như một công tắc bật/tắt class vậy đó.

## Handling CSS Classes

*// add and remove class*

```
$(“p”).removeClass(“blue”).addClass(“red”);
```

*// add if absent, remove otherwise*

```
$(“div”).toggleClass(“main”);
```

`if ($(“div”).hasClass(“main”)) { //... };`: Cái này dùng để kiểm tra xem một hoặc nhiều thẻ `<div>` có class "main" hay không. Nó trả về true nếu có ít nhất một thẻ `<div>` có class "main", và false nếu không có thẻ nào.

*// test for class existance*

```
if ($(“div”).hasClass(“main”)) { //... }
```

`$(“h1”).append(“<li>Hello $!</li>”);` Cái này sẽ chọn tất cả các thẻ `<h1>` và thêm một thẻ `<li>` mới có nội dung "Hello \$!" vào cuối bên trong mỗi thẻ `<h1>`.

`$(“ul”).prepend(“<li>Hello $!</li>”);` Cái này sẽ chọn tất cả các thẻ `<ul>` và thêm một thẻ `<li>` mới có nội dung "Hello \$" vào đầu bên trong mỗi thẻ `<ul>`.

## Inserting new Elements

```
// select > append to the end  
$(“h1”).append(“<li>Hello $!</li>”);  
  
// select > append to the beginning  
$(“ul”).prepend(“<li>Hello $!</li>”);
```

`$(“<li/>”).html(“9”).appendTo(“ul”);`: Cái này tạo ra một thẻ `<li>` mới (dùng `<li/>` là cách viết ngắn gọn), đặt nội dung HTML của nó là "9", và sau đó thêm nó vào cuối tất cả các thẻ `<ul>` trên trang.

`$(“<li/>”).html(“1”).prependTo(“ul”);`: Tương tự như trên, nhưng nó thêm thẻ `<li>` vào đầu tất cả các thẻ `<ul>`.

```
// create > append/prepend to selector  
$(“<li/>”).html(“9”).appendTo(“ul”);  
$(“<li/>”).html(“1”).prependTo(“ul”);
```

# Replacing Elements

`$(“h1”).replaceWith(“<div>Hello</div>”);` Cái này sẽ chọn tất cả các thẻ `<h1>` và thay thế chính chúng bằng một thẻ `<div>` mới có nội dung "Hello".

```
// select > replace
$(“h1”).replaceWith(“<div>Hello</div>”);
```

`$(“<div>Hello</div>”).replaceAll(“h1”);` Cái này tạo ra một thẻ `<div>` mới có nội dung "Hello" và sau đó thay thế tất cả các thẻ `<h1>` bằng cái thẻ `<div>` này. Hai cái này về cơ bản là làm cùng một việc nhưng cú pháp hơi khác nhau.

```
// create > replace selection
$(“<div>Hello</div>”).replaceAll(“h1”);
```

# Replacing Elements while keeping the content

```
$(“h3”).each(function(){  
    $(this).replaceWith(“<div>”  
        + $(this).html()  
        + ”</div>”);  
});
```

Cái đoạn code này sẽ chọn tất cả các thẻ <h3>, sau đó duyệt qua từng thẻ (dùng .each()), và thay thế mỗi thẻ <h3> bằng một thẻ <div> mới. Nội dung bên trong thẻ <h3> (lấy bằng \$(this).html()) sẽ được giữ lại và đặt vào bên trong thẻ <div> mới.

# Deleting Elements

```
// remove all children  
$("#mainContent").empty();
```

`$("#mainContent").empty();`: Cái này sẽ chọn phần tử có id="mainContent" và xóa tất cả các phần tử con của nó, nhưng bản thân cái phần tử #mainContent thì vẫn còn.

```
// remove selection  
$("span.names").remove();
```

`$("span.names").remove();`: Cái này sẽ chọn tất cả các thẻ <span> có class là names và xóa chúng hoàn toàn khỏi DOM.

```
// change position  
$("p").remove(":not(.red)")  
    .appendTo("#main");
```

`$("p").remove(":not(.red)").appendTo("#main");`: Cái này hơi phức tạp một chút. Nó sẽ:  
Chọn tất cả các thẻ <p>.  
Dùng `.remove(":not(.red)")` để xóa tất cả các thẻ <p> mà không có class là red.  
Sau đó, nó sẽ lấy những thẻ <p> còn lại (tức là những thẻ có class red) và dùng `appendTo("#main")` để di chuyển chúng vào cuối phần tử có id="main".

# Handling attributes

```
$(“a”).attr(“href”, “home.htm”);  
// <a href=“home.htm”>...</a>
```

\$(“a”).attr(“href”, “home.htm”);  
Cái này chọn tất cả các thẻ `<a>` và đặt thuộc tính href của chúng thành “home.htm”.  
Ví dụ: `<a href=“home.htm”>...</a>`

```
// set the same as the first one  
$(“button:gt(0)”).attr(“disabled”,  
$(“button:eq(0)”).attr(“disabled”));
```

\$(“button:gt(0)”).attr(“disabled”, \$(“button:gt(0)”).attr(“disabled”));  
Cái này chọn tất cả các thẻ `<button>` có index lớn hơn 0 (tức là từ nút thứ hai trở đi) và đặt thuộc tính disabled của chúng giống như thuộc tính disabled của cái nút đầu tiên.

```
// remove attribute - enable  
$(“button”).removeAttr(“disabled”)
```

\$(“button”).removeAttr(“disabled”);  
Cái này chọn tất cả các thẻ `<button>` và xóa thuộc tính disabled của chúng (tức là kích hoạt chúng).

# Setting multiple attributes

```
$(“img”).attr({  
    “src” : “/images/smile.jpg”,  
    “alt” : “Smile”,  
    “width” : 10,  
    “height” : 10  
});
```

`$(“img”).attr({ “src” : “/images/smile.jpg”, “alt” : “Smile”, “width” : 10, “height” : 10 });`

Cái này chọn tất cả các thẻ `<img>` và thiết lập các thuộc tính `src`, `alt`, `width`, `height` cho chúng cùng một lúc.

# CSS Manipulations

```
// get style  
$("div").css("background-color");
```

Cái này chọn tất cả các thẻ <div> và lấy giá trị của thuộc tính CSS background-color của cái đầu tiên trong số chúng.

```
// set style  
$("div").css("float", "left");
```

\$("div").css("float", "left");  
Cái này chọn tất cả các thẻ <div> và đặt thuộc tính CSS float của chúng thành "left".

```
// set multiple style properties  
$("div").css({ "color": "blue",  
               "padding": "1em",  
               "margin-right": "0",  
               marginLeft: "10px" }));
```

Cái này chọn tất cả các thẻ <div> và thiết lập nhiều thuộc tính CSS cùng một lúc: color, padding, margin-right, và marginLeft.

# Events



# When the DOM is ready...

```
$(document).ready(function(){  
    //...  
});
```

- Fires when the document is ready for programming.
- Uses advanced listeners for detecting.
- `window.onload()` is a fallback.

```
$(document).ready(function(){ //... });
```

Cái này là một cái "khung" mà mình hay dùng để đặt code jQuery của mình vào bên trong. Cái hàm bên trong ready() sẽ chỉ được chạy khi toàn bộ DOM (Document Object Model) của trang web đã được tải xong và sẵn sàng để mình thao tác.

jQuery nó dùng các "advanced listeners" (cơ chế lắng nghe nâng cao) để phát hiện khi nào DOM đã sẵn sàng.

Cái window.onload() cũng tương tự, nhưng nó đợi đến khi tất cả các tài nguyên của trang (bao gồm cả hình ảnh, video,...) được tải xong thì mới chạy. Thường thì mình dùng \$(document).ready() là đủ và nó chạy nhanh hơn.

# Attach Event

```
// execute always  
$("div").bind("click", fn);  
  
// execute only once  
$("div").one("click", fn);
```

## Possible event values:

blur, focus, load, resize, scroll, unload,  
beforeunload, click, dblclick, mousedown, mouseup,  
mousemove, mouseover, mouseout, mouseenter,  
mouseleave, change, select, submit, keydown,  
keypress, keyup, error

(or any custom event)

`$("div").bind("click", fn);`: Cái này sẽ gắn một cái hàm fn (function) vào sự kiện "click" của tất cả các thẻ `<div>` trên trang. Cái hàm fn sẽ được chạy mỗi khi người dùng click vào một thẻ `<div>`.  
`$("div").one("click", fn);`: Cái này cũng gắn một cái hàm fn vào sự kiện "click" của tất cả các thẻ `<div>`, nhưng cái hàm fn này chỉ chạy một lần duy nhất khi người dùng click vào lần đầu tiên. Sau đó, nó sẽ tự động bị gỡ bỏ.

Possible event values (Các giá trị sự kiện có thể có): Cái danh sách này liệt kê một vài sự kiện phổ biến mà mình có thể gắn vào các phần tử HTML, ví dụ như:

`blur, focus`: Khi một phần tử mất hoặc nhận focus.

`load`: Khi một tài nguyên (ví dụ: hình ảnh) đã tải xong.

`resize`: Khi kích thước cửa sổ trình duyệt thay đổi.

`scroll`: Khi người dùng cuộn trang.

`click`: Khi người dùng click chuột.

`dblclick`: Khi người dùng double-click chuột.

`mousedown, mouseup`: Khi chuột được nhấn xuống hoặc thả ra.

`mousemove`: Khi chuột di chuyển.

`mouseover`: Khi chuột di chuyển vào một phần tử.

`mouseout`: Khi chuột di chuyển ra khỏi một phần tử.

`mouseenter`: Tương tự mouseover nhưng không kích hoạt khi di chuyển qua các phần tử con.

`mouseleave`: Tương tự mouseout nhưng không kích hoạt khi di chuyển qua các phần tử con.

`change`: Khi giá trị của một form element (ví dụ: input, select) thay đổi.

`select`: Khi người dùng chọn một đoạn text trong một input hoặc textarea.

`submit`: Khi một form được submit.

`keydown, keypress, keyup`: Khi người dùng nhấn xuống, giữ, hoặc thả một phím.

`error`: Khi có lỗi xảy ra (ví dụ: khi tải hình ảnh bị lỗi).

Và còn nhiều sự kiện tùy chỉnh khác nữa.

# jQuery.Event object

## CONTENTS

[1 jQuery.Event](#)

[2 Attributes](#)

[2.1 event.type](#)

[2.2 event.target](#)

[2.3 event.data](#)

[2.4 event.relatedTarget](#)

[2.5 event.currentTarget](#)

[2.6 event.pageX/Y](#)

[2.7 event.result](#)

[2.8 event.timeStamp](#)

[3 Methods](#)

[3.1 event.preventDefault\(\)](#)

[3.2 event.isDefaultPrevented\(\)](#)

[3.3 event.stopPropagation\(\)](#)

[3.4 event.isPropagationStopped\(\)](#)

[3.5 event.stopImmediatePropagation\(\)](#)

[3.6 event.isImmediatePropagationStopped\(\)](#)

## 1. Thuộc tính (Attributes):

event.type: Cái này cho biết loại sự kiện nào đã xảy ra (ví dụ: "click", "mouseover", "keydown").

event.target: Đây là cái phần tử HTML mà trên đó sự kiện bắt đầu xảy ra. Ví dụ, nếu click vào một thẻ `<span>` nằm bên trong một thẻ `<div>`, thì `event.target` sẽ là cái thẻ `<span>`.

event.data: Cái này chứa dữ liệu tùy chỉnh mà mà có thể truyền vào khi gắn sự kiện bằng hàm `.bind()` hoặc `.on()`.

event.relatedTarget: Đối với các sự kiện liên quan đến chuột (ví dụ: mouseover, mouseout), cái này tham chiếu đến phần tử mà chuột vừa rời khỏi (đối với mouseover) hoặc vừa đi vào (đối với mouseout).

event.currentTarget: Đây là cái phần tử HTML mà cái hàm xử lý sự kiện hiện tại đang được gọi trên đó. Nó có thể khác với `event.target` nếu sự kiện "nổi bọt" (bubbling) lên trên cây DOM.

event.pageX / event.pageY: Cái này cho biết tọa độ X và Y của con chuột tại thời điểm xảy ra sự kiện, tính từ góc trên bên trái của toàn bộ trang web.

event.result: Cái này chứa giá trị cuối cùng được trả về bởi cái hàm xử lý sự kiện mà đã được kích hoạt cuối cùng cho sự kiện này (sau khi sự kiện có thể đã "nổi bọt" qua nhiều phần tử).

event.timeStamp: Đây là một số đại diện cho thời điểm chính xác mà sự kiện xảy ra, tính bằng mili giây kể từ khi trang web được tải.

event.preventDefault(): Cái hàm này dùng để ngăn chặn cái hành động mặc định của trình duyệt đối với sự kiện đó. Ví dụ, nếu click vào một thẻ `<a>`, hành động mặc định là trình duyệt sẽ chuyển đến cái URL được chỉ định trong thuộc tính `href`. Nếu mà gọi `event.preventDefault()` trong hàm xử lý sự kiện click, thì trình duyệt sẽ không thực hiện cái hành động chuyển trang đó.

event.isDefaultPrevented(): Cái hàm này trả về `true` nếu `event.preventDefault()` đã được gọi cho sự kiện này, và `false` nếu chưa.

event.stopPropagation(): Cái hàm này dùng để ngăn chặn sự kiện "nổi bọt" lên các phần tử cha trong cây DOM. Ví dụ, nếu mà có một sự kiện click được gắn vào cả thẻ `<span>` và thẻ `<div>` chứa nó, và mà gọi `event.stopPropagation()` trong hàm xử lý sự kiện của thẻ `<span>`, thì hàm xử lý sự kiện của thẻ `<div>` sẽ không được gọi khi mà click vào thẻ `<span>`.

event.isPropagationStopped(): Cái hàm này trả về `true` nếu `event.stopPropagation()` đã được gọi cho sự kiện này, và `false` nếu chưa.

event.stopImmediatePropagation(): Cái hàm này còn mạnh hơn cả `event.stopPropagation()`. Nó không chỉ ngăn chặn sự kiện "nổi bọt" lên các phần tử cha mà còn ngăn chặn tất cả các hàm xử lý sự kiện khác được gắn vào cùng một phần tử cho cùng một loại sự kiện được thực thi.

event.isImmediatePropagationStopped(): Cái hàm này trả về `true` nếu `event.stopImmediatePropagation()` đã được gọi cho sự kiện này, và `false` nếu chưa.

# Detaching Events

```
$(“div”).unbind(“click”, fn);
```

`$(“div”).unbind(“click”, fn);`: Cái này sẽ gỡ bỏ cái hàm fn đã được gắn vào sự kiện "click" của tất cả các thẻ `<div>`. jQuery nó sẽ thêm một cái ID duy nhất vào mỗi hàm được gắn để có thể gỡ bỏ đúng cái hàm đó.

(Unique ID added to every attached function)

`$("div").trigger("click");`: Cái này sẽ kích hoạt sự kiện "click" trên tất cả các thẻ `<div>`. Nó không chỉ gọi cái hàm đã được gắn vào sự kiện "click" mà còn kích hoạt luôn cả hành động mặc định của trình duyệt cho sự kiện đó (ví dụ, nếu là một nút button thì nó sẽ thực hiện hành động submit form nếu có).

## Events Triggering

Mình cũng có thể dùng `trigger()` để kích hoạt các sự kiện tùy chỉnh mà mình đã tạo ra.

Các sự kiện được kích hoạt bằng `trigger()` sẽ "nổi bọt" (bubble up) trong cây DOM, tức là nếu không có hàm xử lý sự kiện nào được tìm thấy trên phần tử hiện tại, thì nó sẽ "lan" lên phần tử cha và cứ tiếp tục như vậy cho đến khi gặp một hàm xử lý hoặc lên đến gốc của cây DOM.

```
$("div").trigger("click");
```

Triggers browser's event action as well.

Can trigger custom events.

Triggered events bubble up.

# Effects

# Showing or Hiding Element

Showing or Hiding Element (Hiển thị hoặc ẩn phần tử):

// just show

```
$(“div”).show();
```

// reveal slowly, slow=600ms

```
$(“div”).show(“slow”);
```

// hide fast, fast=200ms

```
$(“div”).hide(“fast”);
```

// hide or show in 100ms

```
$(“div”).toggle(100);
```

\$(“div”).show(): Hiển thị ngay lập tức tất cả các thẻ <div> đang bị ẩn.

\$(“div”).show(“slow”): Hiển thị từ từ tất cả các thẻ <div> đang bị ẩn. “slow” tương đương với khoảng 600ms.

\$(“div”).hide(“fast”): Ẩn đi nhanh chóng tất cả các thẻ <div> đang hiển thị. “fast” tương đương với khoảng 200ms.

\$(“div”).toggle(100): Nếu thẻ <div> đang hiển thị thì nó sẽ ẩn đi, còn nếu đang ẩn thì nó sẽ hiển thị ra, với thời gian chuyển đổi là 100ms.

# Sliding Elements

```
$(“div”).slideUp();  
$(“div”).slideDown(“fast”);  
$(“div”).slideToggle(1000);
```

`$("div").slideUp();`: Trượt lên trên để ẩn tất cả các thẻ `<div>`.

`$("div").slideDown("fast")`: Trượt xuống dưới để hiển thị tất cả các thẻ `<div>` đang bị ẩn, với tốc độ "fast".

`$("div").slideToggle(1000)`: Nếu thẻ `<div>` đang hiển thị thì nó sẽ trượt lên để ẩn, còn nếu đang ẩn thì nó sẽ trượt xuống để hiển thị, với thời gian chuyển đổi là 1000ms.

# Fading Elements

```
$("div").fadeIn("fast");  
$("div").fadeOut("normal");  
// fade to a custom opacity  
$("div").fadeTo ("fast", 0.5);
```

`$("div").fadeIn("fast");`: Làm cho tất cả các thẻ `<div>` đang ẩn mờ dần hiện ra với tốc độ "fast".

`$("div").fadeOut("normal");`: Làm cho tất cả các thẻ `<div>` đang hiển thị mờ dần biến mất với tốc độ "normal" (thường là 400ms).

`$("div").fadeTo ("fast", 0.5);`: Làm cho tất cả các thẻ `<div>` mờ dần đến độ trong suốt là 0.5 (50%) với tốc độ "fast".

Lưu ý: Fading thực chất là thay đổi độ trong suốt (opacity) của phần tử.

Fading === changing opacity

# Detecting animation completion

```
$("div").hide("slow", function() {  
    alert("The DIV is hidden");  
});
```

`$("div").hide("slow", function() { alert("The DIV is hidden"); });`: Sau khi hiệu ứng ẩn thẻ `<div>` hoàn thành (với tốc độ "slow"), cái hàm bên trong (callback function) sẽ được gọi và hiển thị một thông báo.

`$("div").show("fast", function() { $(this).html("Hello jQuery"); });`: Sau khi hiệu ứng hiển thị thẻ `<div>` hoàn thành (với tốc độ "fast"), cái hàm bên trong sẽ được gọi và thay đổi nội dung HTML của cái thẻ `<div>` đó thành "Hello jQuery". Lưu ý là `$(this)` ở đây đại diện cho cái phần tử DOM hiện tại đang được xử lý.

Hầu hết các hàm hiệu ứng đều có thể nhận thêm một tham số là một hàm callback, nó sẽ được gọi khi hiệu ứng kết thúc. Cú pháp thường là `(speed, callback)`.

```
$("div").show("fast", function() {  
    $(this).html("Hello jQuery");  
}); // this is a current DOM element
```

Every effect function has a (speed, callback)  
overload

# Custom Animation

```
// .animate(options, duration)
$("div").animate({
    width: "90%",
    opacity: 0.5,
    borderWidth: "5px"
}, 1000); Cái này cho phép mình
tạo ra các hiệu ứng phức tạp hơn bằng cách thay đổi
các thuộc tính CSS theo thời gian. Ở đây, nó sẽ thay
đổi width, opacity, và borderWidth của tất cả các
thẻ <div> trong vòng 1000ms.
```

width: "90%",  
opacity: 0.5,  
borderWidth: "5px"  
, 1000);

# Chaining Animation

Chaining Animation (Kết chuỗi hiệu ứng):

`$("div").animate({width: "90%"},100).animate({opacity: 0.5},200).animate({borderWidth: "5px"});` Mình có thể "xếp hàng" nhiều hiệu ứng lại với nhau bằng cách dùng "chaining". Theo mặc định, các hiệu ứng sẽ được thực hiện lần lượt theo thứ tự mình viết.

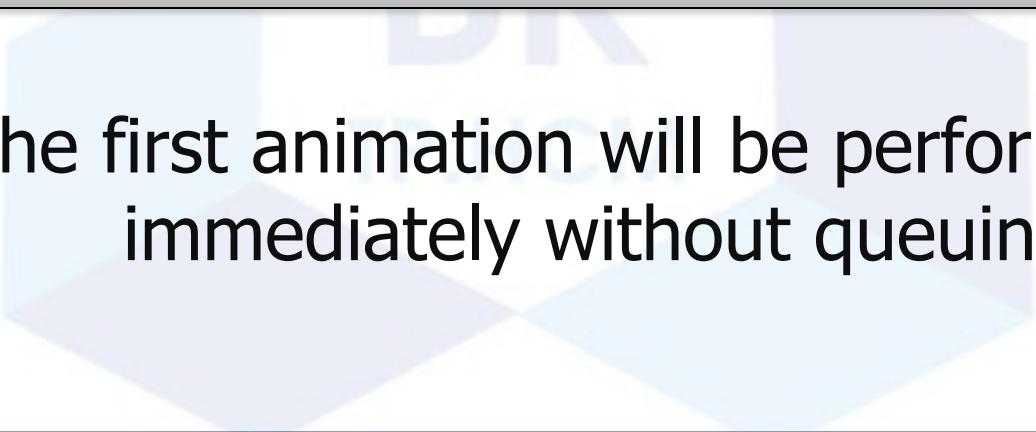
```
$("div").animate({width: "90%"},100)  
    .animate({opacity: 0.5},200)  
    .animate({borderWidth: "5px"});
```

By default animations are queued and than performed one by one

# Controlling Animations Sync

`$("div").animate({width: "90%"}, {queue:false, duration:1000}).animate({opacity : 0.5});` Bình thường thì các hiệu ứng sẽ được đưa vào một hàng đợi và chạy lần lượt. Nhưng nếu mình đặt `queue:false` trong cái object tùy chọn của hàm `animate()`, thì cái hiệu ứng đó sẽ được thực hiện ngay lập tức mà không cần chờ các hiệu ứng khác hoàn thành. Trong ví dụ này, cái hiệu ứng thay đổi `width` sẽ chạy đồng thời với cái hiệu ứng thay đổi `opacity`.

```
$("div")
    .animate({width: "90%",
              {queue:false, duration:1000})
    .animate({opacity : 0.5});
```



The first animation will be performed immediately without queuing

# AJAX with jQuery

BK  
TP.HCM

# Loading content

```
$.ajax({url: "test.php",
success: function(result) {
    $("#div1").html(result);
}
});
```

Cái này là cách tổng quát nhất để làm AJAX trong jQuery.  
\$.ajax() nhận vào một object các tùy chọn.  
url: "test.php": Chỉ định cái URL mà mình muốn gửi yêu cầu đến (trong trường hợp này là file test.php).  
success: function(result) { \$("#div1").html(result); }: Đây là một hàm callback sẽ được gọi khi yêu cầu thành công. Cái biến result sẽ chứa dữ liệu mà server trả về. Trong ví dụ này, nó sẽ lấy cái dữ liệu đó và đặt nó vào bên trong cái phần tử HTML có id="div1".

# Sending GET/POST requests

Cái này là một cách viết ngắn gọn để gửi yêu cầu GET.

"test.php": URL để gửi yêu cầu.

{id:1}: Dữ liệu sẽ được gửi đi dưới dạng query parameters (ví dụ: test.php?id=1).

function(data){alert(data)}: Hàm callback sẽ được gọi khi yêu cầu thành công. Cái biến data chứa dữ liệu server trả về. Trong ví dụ này, nó sẽ hiển thị một hộp thoại alert với dữ liệu đó.

```
$.get("test.php", {id:1},  
      function(data){alert(data)});
```

```
$.post("test.php", {id:1},  
       function(data){alert(data)});
```

Cái này tương tự như \$.get(), nhưng nó dùng phương thức POST để gửi dữ liệu. Dữ liệu {id:1} sẽ được gửi trong phần thân của yêu cầu.

# Retrieving JSON Data

```
$.getJSON("users.php", {id:1},  
    function(users)  
    {  
        alert(users[0].name);  
    } );
```

`$.getJSON("users.php", {id:1}, function(users) { alert(users[0].name); });`

Cái này là một cách viết ngắn gọn để gửi yêu cầu GET và tự động xử lý dữ liệu trả về dưới dạng JSON.

"users.php": URL để gửi yêu cầu.

{id:1}: Dữ liệu gửi đi dưới dạng query parameters.

function(users) { alert(users[0].name); }: Hàm callback sẽ được gọi khi yêu cầu thành công. Cái biến users sẽ là một đối tượng JavaScript (thường là một mảng hoặc một object) được tạo ra từ dữ liệu JSON mà server trả về. Trong ví dụ này, nó sẽ hiển thị tên của người dùng đầu tiên trong mảng users.

# Tài Liệu Tham Khảo

- [1] Stepp,Miller,Kirst. Web Programming Step by Step.( 1st Edition, 2009) Companion Website:  
<http://www.webstepbook.com/>
- [2] W3Schools,  
<http://www.w3schools.com/html/default.asp>