MỤC LỤC

[CHƯƠNG 1. Variabels 3](#_Toc207977895)

[1.1. Khái niệm Biến trong javascripts 3](#_Toc207977896)

[1.2. Quy tắc đặt tên biến 3](#_Toc207977897)

[1.3. Kiểu dữ liệu 4](#_Toc207977898)

[1.4. Hoisting & Temporal Dead Zone (TDZ) 5](#_Toc207977899)

[1.4.1. Hoisting 5](#_Toc207977900)

[1.4.2. Temporal Dead Zone (TDZ) 6](#_Toc207977901)

[1.5. Phạm vi của biến 7](#_Toc207977902)

[1.6. Tham chiếu và tham trị của biến 8](#_Toc207977903)

[CHƯƠNG 2. Cấu trúc rẽ nhánh 9](#_Toc207977904)

[2.1. Phép Toán 11](#_Toc207977905)

[CHƯƠNG 3. Cấu trúc lặp 12](#_Toc207977906)

[3.1. Vòng lặp dành cho iterable – For … of 12](#_Toc207977907)

[3.2. Vòng lặp dành cho object - for .. in 13](#_Toc207977908)

[3.3. phương thức Foreach của đối tượng array 14](#_Toc207977909)

[3.3.1. Khái niệm 14](#_Toc207977910)

[3.3.2. Cú pháp khai báo foreach 14](#_Toc207977911)

[3.4. Tổng hợp vòng loops 14](#_Toc207977912)

[CHƯƠNG 4. Hàm – Funcitions 17](#_Toc207977913)

[4.1. Khái niệm về funcition 17](#_Toc207977914)

[4.2. Cú pháp và Các loại hàm 17](#_Toc207977915)

[4.3. Tham số - parameters và đối số - arguments 21](#_Toc207977916)

[4.4. Closure 22](#_Toc207977917)

[4.5. Hàm bất đồng bộ - asynchronous function 23](#_Toc207977918)

[4.5.1. Khái niệm bất đồng bộ 23](#_Toc207977919)

[4.5.2. Các hàm hay công việc cần cơ chế bất đồng bộ 26](#_Toc207977920)

[4.5.3. Tại sao lại cần bất đồng bộ khi làm việc với API, đọc file, hoặc setTimeout 27](#_Toc207977921)

[4.5.4. Các cách xử lý bất đồng bộ 28](#_Toc207977922)

[4.5.4.1 Callback - gọi lại khi xong 28](#_Toc207977923)

[4.5.4.2 Promise - “lời hứa” sẽ có kết quả. 29](#_Toc207977924)

[CHƯƠNG 5. DOM manipulation 33](#_Toc207977925)

# Variabels

## Khái niệm Biến trong javascripts

**Biến** là tên (định danh) của một vùng nhớ dùng để lưu trữ giá trị (dữ liệu) và giá trị đó có thể được thay đổi khi thực hiện chương trình.

*Cú pháp khai báo biến :*

*var <ten\_bien> = <gia\_tri>*

*let <ten\_bien> = <gia\_tri>*

*const <ten\_bien> = <gia\_tri>*

*Trong đó:*

*let là từ khóa giúp ta nhận biết đây là* ***biến có thể thay đổi******được dữ liệu****.*

*const là từ khóa giúp ta nhận biến đây là biến* ***không thể thay đổi được dữ liệu***

*ten\_bien : Là tên (nhãn) của biến và sẽ được đặt theo* ***quy tắc đặt tên biến***

## Quy tắc đặt tên biến

*→ Quy tắc :*

*Chỉ chứa chữ cái, số, \_ hoặc $.*

*Lưu ý:*

*Không bắt đầu bằng số.*

*Phân biệt chữ hoa – chữ thường (age khác Age).*

*Không trùng với từ khóa JavaScript (if, for, class...).*

*→ Nên dùng* ***camelCase****: userName, totalPrice.*

*Ví dụ: \_name,*

## Kiểu dữ liệu

*Kiểu dữ liệu nguyên thủy (Primitive types) :*

|  |  |
| --- | --- |
| ***Kiểu dữ liệu*** | ***Ví dụ*** |
| *string* | let name = "Alice"; // dùng nháy kép  let greeting = 'Hello'; // dùng nháy đơn  let message = `Hi, ${name}!`; // template literal (ES6) |
| *number* | let age = 20; // số nguyên  let pi = 3.14159; // số thực (floating point)  let big = 1e9; // số dạng khoa học:1,000,000,000 |
| *boolean* | let isStudent = true;  let isOnline = false; |
| *null* | let car = null; // không có giá trị |
| *undefined* | let address; // chưa gán -> mặc định undefined |
| *Symbol*  *→ Nó dùng để tạo ra một* ***giá trị duy nhất và bất biến (unique & immutable)****.* | const uniqueId = Symbol("id"); |
| *bigint* | const bigNumber = 1234567456789012345678901234567890n |

Bảng 1.1 kiểu dữ liệu nguyên thủy

*Kiểu dữ liệu tham chiếu (Reference Types)*

* *Biến lưu* ***địa chỉ tham chiếu*** *đến vùng nhớ chứa dữ liệu.*

|  |  |
| --- | --- |
| ***Kiểu dữ liệu*** | *Ví dụ* |
| *Object*  *(****Hai kiểu object đặc biệt***  ***RegExp (biểu thức chính quy), Date****)* | const person = {  name: "Tran",  age: 25,  isStudent: true  };  const date = new Date()  const pattern = /[A-Z]/g; |
| *Array* | let colors = ["red", "green", "blue"]; |
| *Function* | const greet = function(name) {  return `Hello, ${name}!`;  }; |

Bảng 1.2 Kiểu dữ liệu phức tạp

## Hoisting & Temporal Dead Zone (TDZ)

### Hoisting

Đây là quá trình mà JAVASCRIPTS sẽ “kéo” phần **khai báo** biến và hàm lên đầu phạm vi (scope) trước khi thực thi code.

Ví dụ:

|  |  |
| --- | --- |
| *Kết quả 1* | *Kết quả 2* |
| console.log(a); // undefined  var a = 10; | var a; // hoisting  console.log(a); // undefined  a = 10; // gán giá trị |

→ Hoisting sẽ giúp khai báo var a kéo lên đầu (kết quả 1) → kết quả 2 là minh họa cho việc đó.

→ *Chú ý : Hosting ko áp dụng với let và const.*

### Temporal Dead Zone (TDZ)

**TDZ** = khoảng thời gian trong phạm vi (scope) từ khi biến được hoist **đến khi dòng khai báo thực sự chạy**.  
  
*→ Nếu biến chứa khai báo mà được gọi thì biến đó đang nằm trong TDZ. (chỉ áp dụng với let, const)*

*A white background with black text

AI-generated content may be incorrect.*

Hình 1.1 Minh họa TDZ

## Phạm vi của biến

**Phạm vi của biến** (variable scope) là “khu vực” trong mã mà biến đó tồn tại và có thể được truy cập

|  |  |  |
| --- | --- | --- |
| *Tên phạm vi* | *Mô tả* | *Ví dụ* |
| ***Global Scope*** *– Phạm vi toàn cục* | ***→ Khai báo****: Bên ngoài mọi hàm hoặc khối {}*  *→* ***Truy cập****: Ở bất kỳ đâu trong chương trình.* | var a = 5;  function demo() {  console.log(a);  }  demo(); |
| ***Function Scope*** *– Phạm vi hàm* | *→* ***Áp dụng****: var*  *→* ***Truy cập****: Biến chỉ tồn tại trong toàn bộ hàm chứa nó, kể cả khi khai báo trong một khối if, for…* | function test() {  if (true) {  var x = 10;  }  console.log(x); // 10 (vẫn truy cập được)  }  test(); |
| ***Block Scope*** *– Phạm vi khối* | ***→ Áp dụng****:* *let và const.*  → ***Truy cập****:* Biến chỉ tồn tại bên trong cặp {} nơi nó được khai báo. | function test() {  if (true) {  let y = 20;  const z = 30;  }  console.log(y); // ❌ ReferenceError  }  test(); |

Bảng 1.3 Bảng phạm vi của biến

## Tham chiếu và tham trị của biến

Tham chiếu và tham trị là hai cơ chế khi biến được truyền đến hàm.

→ Cơ chế tham chiếu và tham trị chỉ là cách áp dụng phụ thuộc vào **kiểu dữ liệu**.

→ *Kiểu dữ liệu nguyên thủy → tham trị*

*→ Kiểu dữ liệu tham chiếu → tham chiếu*

|  |  |  |
| --- | --- | --- |
| *Kiểu* | *Nội dung* | *Ví dụ* |
| *Tham trị* | *Khi gán hoặc truyền vào hàm,* ***một bản sao giá trị*** *được tạo ra. Thay đổi biến sao chép* ***không ảnh hưởng*** *đến biến gốc.* | function changeValue(num) {  num = 100;  }  let a = 50;  changeValue(a);  → console.log(a); // 50 |
| *Thạm chiếu* | Khi gán hoặc truyền vào hàm, bản sao **địa chỉ** được tạo ra, nên cả hai biến cùng trỏ đến **cùng một dữ liệu** trong bộ nhớ. | function changeObj(obj) {  obj.value = 100;  }  let myObj = { value: 50 };  changeObj(myObj);  → console.log(myObj.value); |

Bảng 1.4 Mô tả tham chiếu và tham trị

# Cấu trúc rẽ nhánh

*→ Cấu trúc rẽ nhánh dùng để điều khiển luồng thực thi, tùy thuộc vào một điều kiện nhất định là đúng hay sai.   
→ Các đặc điểm khác các ngôn ngữ C/C++ hoặc Python.*

* *Bảng các giá trị ngầm được chuyển sang Boolean.*

|  |  |
| --- | --- |
| Tên Giá trị | Các giá trị |
| Kết quả là **false** → gọi là **falsy**. | * *false* * *0 (số 0)* * *-0* * *0n (BigInt 0)* * *"" (chuỗi rỗng)* * *null* * *undefined* * *NaN* |
| Kết quả là true → gọi là **truthy**. | *Còn lại là truthy.* |

Bảng 2.1 Truthy và Falsy

*Ví dụ:*

A computer code with text

AI-generated content may be incorrect.

Hình 2.1 Mô phỏng truthy.

* *switch … case dùng* ***so sánh nghiêm ngặt*** *(===), nên* ***kiểu dữ liệu phải trùng khớp***.

Ví dụ:

switch (1) {

case "1": // ❌ không khớp vì khác kiểu

console.log("String 1");

break;

case 1: // ✅ khớp

console.log("Number 1");

break;

}.

*→ Lưu ý : Biểu thức* ***không so sánh theo toán học “chuỗi”*** *như ta tưởng (kiểu toán học: 3 < a < 5).*

*Ví dụ:*

a = 4.

if (3 < a < 5) {

console.log("Hello World");

}

*→ Kết quả : in ra Hello World   
Nó thực hiện* ***theo thứ tự trái → phải****:*

1. *3 < a → 3 < 4 →* ***true****.*
2. *Lúc này ta có: true < 5.*
   * *Trong so sánh số học, true được ép kiểu thành 1.*
   * *1 < 5 →* ***true****. Kết quả cuối cùng =* ***true*** *→ nên in "Hello World".*

## Phép Toán

→ Có thêm *toán tử so sánh === và !==.*

*Toán tử so sánh:*

*→ && : toán tử AND*

*→ || : toán tử hoặc*

|  |  |  |
| --- | --- | --- |
| *Cấu trúc rẽ nhánh* | *Ý nghĩa* | *Cú pháp* |
| *if* | *Dùng khi bạn chỉ muốn thực thi một đoạn mã* ***nếu điều kiện là đúng.*** | *if (dieu\_kien) {*  *}* |
| *if...else* | *Dùng khi bạn* ***muốn thực thi một đoạn mã nếu điều kiện đúng****, và một* ***đoạn mã khác nếu điều kiện sai.*** | *if (dieu\_kien) {*  *} else {*  *}* |
| *if...else if...else* | *Dùng khi bạn có* ***nhiều điều kiện cần kiểm tra.*** | *if (dieu\_kien\_1) {*  *} else if (dieu\_kien\_2) {*  *} else {*  *}* |
| *switch...case* | *Là một* ***giải pháp thay thế*** *cho chuỗi if...else if...else khi bạn cần so sánh một biến với nhiều giá trị khác nhau.* | *switch (bieu\_thuc) { case gia\_tri\_1:*  *….*  *break; ... có thể có nhiều case khác default: // Khối mã thực thi nếu không có case nào khớp }* |

Bảng 2.2 Tổng cấu trúc rẽ nhánh

# Cấu trúc lặp

→ *Cấu trúc lặp là cách để lặp lại một đoạn mã nhiều lần cho đến khi một điều kiện không còn đúng nữa.*

*→ Xử lý dữ liệu, duyệt mảng, hoặc thực hiện các tác vụ lặp đi lặp lại mà không cần viết lại code.*

*→ Các vòng lặp đặc biệt chỉ có trong loop áp dụng trong các trường hợp đặc biệt*

## Vòng lặp dành cho iterable – For … of

*→ Iterable là một* ***đối tượng*** *có thể được “duyệt qua từng phần tử” bằng vòng lặp.*

* *Array → [1,2,3]*
* *String → "Hello"*
* *Map → new Map([["a",1],["b",2]])*
* *Set → new Set([1,2,3])*

→ *Duyệt* ***giá trị*** *các phần tử của iterable.*

*→ Cú pháp :*

***for (let*** *value* ***of*** *arr****) {***

*// Khối mã để thực thi*

*}*

## Vòng lặp dành cho object - for .. in

**→ *Duyệt qua các thuộc tính (key)*** *của một đối tượng.*

*→ Lưu ý:   
 (Array cũng là một đối tượng với key là chỉ số)*

*→ Cú pháp :*

*for (let key in object) {*

*// code xử lý với object[key]*

*}*

*Trong đó :*

*key: là tên thuộc tính (kiểu chuỗi) của object.*

*object: là đối tượng bạn muốn duyệt.*

*Ví dụ :*

*const person = { name: "Nam", age: 25, city: "Hà Nội" };*

*for (let key in person) {*

*console.log(key, person[key]);*

*}*

*Ví dụ:*

*`let arr = ['A', 'B'];`*

*`for (let item in arr) { console.log(item); }`*

*→ in ra 0 và 1.*

*Mảng arr có:*

* *Key: "0", "1" (chỉ số của phần tử)*
* *Value: "A", "B"*

## phương thức Foreach của đối tượng array

### Khái niệm

*→ Là* ***một phương thức của mảng*** *(Array method) trong JavaScript.*

*→* ***Duyệt qua từng phần tử của mảng*** *và* ***gọi một hàm callback*** *cho mỗi phần tử.*

*→ Thường dùng để* ***thực hiện hành động*** *(in ra, tính toán, cập nhật dữ liệu…)*

*Tại sao lại có foreach ? → vì forEach* ***ngắn gọn, dễ đọc*** *hơn for hoặc for...of.*

***Lưu ý:*** *Không phù hợp khi cần* ***thoát vòng lặp sớm****. (không thể dùng break)*

### Cú pháp khai báo foreach

*Cú pháp:*

*array.forEach(function(currentValue, index) {*

*// code xử lý*

*}, thisValue);*

**Tham số của callback:**

1. currentValue *(bắt buộc)* – giá trị phần tử hiện tại.
2. index *(tùy chọn)* – chỉ số của phần tử hiện tại.
3. thisValue *(tùy chọn)* – giá trị để dùng làm this bên trong callback.

## Tổng hợp vòng loops

|  |  |  |
| --- | --- | --- |
| *Tên phương thức* | *Ý nghĩa* | *Ví dụ* |
| *for* | biết số lần lặp. | for (let i = 0; i < 5; i++) {  console.log(i);  } |
| *for … in* | duyệt key. | let obj = { a: 1, b: 2, c: 3 };  for (let key in obj) {  console.log(key, obj[key]);  } |
| *for … of* | duyệt value | let arr = [10, 20, 30];  for (let value of arr) {  console.log(value);  } |
| *while* | lặp khi điều kiện đúng. | let i = 0;  while (i < 5) {  console.log(i);  i++;  } |
| *do … while* | Được thử 1 lần rồi lặp khi điều kiện đúng. | let i = 0;  do {  console.log(i);  i++;  } while (i < 5) |
| *forEach* | duyệt mảng bằng hàm callback. | let arr = [1, 2, 3];  arr.forEach((value, index) ⇒ {  console.log(index, value);  }); |

Bảng 3.1 cấu trúc lặp

# Hàm – Funcitions

## Khái niệm về funcition

→ **Function** là một **đối tượng đặc biệt** (object) có thể được gọi (callable).

→ Là một kiểu dữ liệu trong JS.

* + Tại vì nó gán cho biến hoặc thuộc tính.
  + Truyền cho các hàm khác như đối số (callback)
  + Trả về từ một hàm

## Cú pháp và Các loại hàm

|  |  |  |
| --- | --- | --- |
| *Tên gọi hàm* | *Cú pháp* | *Ý nghĩa* |
| *Function Declaration* ***(Hàm khai báo)*** | ***Cú pháp: function*** *ten\_ham(tham số1, tham\_so\_2 …) {  thân hàm;*  *}*  ***Gọi hàm:***  *console.log(ten\_ham(tham\_so\_1…));* | *→ Dùng khi cần định nghĩa “hàm mức module” rõ ràng.*  *→ Được hoisting (có thể gọi hàm trước khi khai báo)* |
| ***Biểu thức hàm*** *(Function Expression)* | *const ten\_ham = function(thamSo1, thamSo2){  thân hàm; }*  ***Gọi hàm:*** *console.log(ten\_ham(tham\_so…));* | *→ Tạo* ***hàm ẩn danh****/gán vào biến/hoặc làm đối số.*  *→ Hàm ẩn danh* ***không thể tự gọi lại chính nó*** *trừ khi gán vào biến hoặc dùng named function expression.*  *Ví dụ :*  *// Gán một hàm ẩn danh vào biến*  *const chao = function() {*  *console.log("Xin chào!");*  *};*  *chao(); // Output: Xin chào!*  *→ function() { ... } không có tên, nhưng được gán cho biến chao.* |
| *Arrow Function (****Hàm mũi tên****)* | *const tenHam = (thamSo1, thamSo2) ⇒ { return thamSo1 + thamSo2; };*  *→ Gọi hàm:*  *//* ***Nếu thân hàm chỉ có một dòng*** *và trả về một giá trị, có thể bỏ qua {} và return:*  *const tinhTong = (a, b) => a + b;*  ***// Nếu chỉ có một tham số, có thể bỏ qua ():***  *const binhPhuong = x => x \* x;* ***// Nếu không có tham số***  *const loiChao = () ⇒ console.log("Hello World!");*  *Gọi hàm:  console.log(tinhTong(5, 3)); // Output: 8 console.log(binhPhuong(4)); // Output: 16 loiChao(); // Output: Hello World!* | *→ Không có this / arguments riêng, không làm constructor.*  *→ Dùng làm callback, đối số cho map, filter, reduce.*  *Ví dụ:*  *const newArr = arr.map(item => item \* 2); // Arrow function trong callback với arrow fuc có một tham số và một dòng* |
| *Method Shorthand (viết tắt phương thức)* | *const obj = {*  ***say(name) {***  ***console.log("Hi", name);***  ***}***  *};*  *Kiểu ngày xưa để khai báo một phương thức có kiểu dữ liệu function const obj = {*  ***say: function sayHello(name)*** *{*  *console.log("Hi", name);*  *}*  *};* | *→ Ngắn gọn, this trỏ về object.*  *→ Sử dụng khi định nghĩa method trong object (Khi một hàm là thuộc tính của một đối tượng, chúng ta gọi nó là một* ***phương thức (method)****.)* |
| *Immediately Invoked Function Expression (IIFE)* | *Khai báo một biểu thức hàm nhưng không gán vào biến nào cả mà được bọc bởi ().*  *→ Cú pháp: (function(ten) {*  *console.log("Xin chào, " + ten + "!");*  *})("Thế giới"); // Output: Xin chào, Thế giới!* | ***IIFE*** *(phát âm là "iffy") là một hàm được định nghĩa và thực thi ngay lập tức.*  *→ Và sau đó được gọi ngay bằng một cặp dấu ngoặc đơn () thứ 2 .*  *Tạo scope riêng.*  *→ Dùng khi ta muốn đóng gói biến lại.* |
| *Generator Function (Hàm Generator)* | *Cú pháp:*  *function\* tenGenerator(thamSo1, thamSo2) {*  *// code*  *yield giaTri1;*  *// code*  *yield giaTri2;*  *return giaTriCuoi;*  *}*  *Cách gọi: Gọi như một hàm bình thường.*  *const generator = demSo(); console.log(generator.next())* | *→ Tạm dừng việc thực thi tại bất kỳ điểm nào (với toán tử yield).*  *→ Tiếp tục thực thi từ điểm đã dừng. Hay nói cách khác không phải bắt đầu lại từ đầu.*  ***→ Sử dụng phương thức Next() trả về iterator****.*  *.next() trả về object { value, done }:*   * *value: giá trị từ yield hoặc return.* * *done: false nếu còn giá trị, true nếu đã kết thúc.* |

Bảng 4.1 Hàm thông dụng và cú pháp khai báo

* *Truyền đối số cho Hàm Generator*

*Ví dụ:*

function\* demo() {

    const name = yield "Nhập tên:";

    console.log(`Xin chào, ${name}`);

}

const g = demo();

console.log(g.next());       // { value: "Nhập tên:", done: false }

console.log(g.next("Nam"));  // In: "Xin chào, Nam"

→ **Điểm mấu chốt**: Dòng const name = ... chưa hoàn thành. Nó đang chờ một giá trị được gửi vào từ lần gọi next() tiếp theo.

→ Đây là bước quan trọng nhất. Chúng ta gọi .next() một lần nữa, nhưng lần này ta truyền vào một đối số là "Nam" nếu không có tham số thì đó sẽ là name = undefined.

→ Giá trị "Nam" này được **gửi vào trong** generator, và trở thành kết quả của biểu thức yield đang tạm dừng.

## Tham số - parameters và đối số - arguments

→ JS **không kiểm tra số tham số**: Gọi thiếu thì biến dư sẽ là undefined.

→ Gọi thừa thì bỏ qua (trừ khi dùng arguments/rest).

→ Có các loại parameters đặc biệt:

* ***Default parameters:***

***Ví dụ:*** *→ function greet(name="friend"){...}*

* ***Rest parameters: chấp nhận một số lượng đối số không xác định*** *và gộp chúng lại thành* ***một mảng duy nhất.***

→ *Cú pháp : Sử dụng dấu ba chấm (...) đặt trước tên của* ***tham số cuối cùng*** *trong định nghĩa hàm.*

*→ function tenHam(thamSo1, thamSo2,* ***...cacThamSoConLai****) { ... }*

*Ví dụ:*

function tinhTong(...cacSo) {

console.log(cacSo); // cacSo là một mảng

let tong = 0;

for (let so of cacSo) {

tong += so;

}

return tong;

}

console.log(tinhTong(1, 2, 3)); // Output: [1, 2, 3] -> 6

console.log(tinhTong(10, 20, 30, 40)); // Output: [10, 20, 30, 40] -> 100

* ***arguments (giả mảng, legacy) → phương pháp cũ bỏ. (****Nó sử dụng một đối tưởng mảng giả gọi là argument****)***
* ***destructuring params:*** Tham số là một đối tượng hoặc một mảng từ đó trích xuất và gán nó vào các biến.

*Ví dụ:*

// Sử dụng destructuring params

function displayUserOptimized({ name, age }) {

console.log(`Tên: ${name}, Tuổi: ${age}`);

}

displayUserOptimized(user); // Output: Tên: Alice, Tuổi: 30

## Closure

→ *Là một hàm được định nghĩa bên trong một hàm khác.*

*Ví dụ:*

function outer() {

let count = 0; // biến trong phạm vi outer

return function inner() {

count++;

console.log(count);

};

}

const counter = outer(); // outer() chạy, trả về inner()

counter(); // 1

counter(); // 2

→ Tại sao không in ra 1 1 mà lại là 1 2.

* *Do cơ chế của closure ghi “nhớ. Closure* ***không lưu bản sao giá trị****, mà lưu* ***tham chiếu*** *tới biến trong môi trường (****Lexical Environment****) đã tạo ra nó.*
* *Miễn là còn ít nhất một hàm con giữ tham chiếu, biến đó sẽ* ***không bị garbage collector xóa****.*

→ ***Môi trường từ vựng (Lexical Environment):*** *phạm vi biến được xác định* ***ngay tại thời điểm viết code****, dựa trên vị trí hàm được khai báo,* ***không*** *phụ thuộc vào nơi hàm được gọi. Hay có thể hiểu function tạo ra các môi trường riêng biệt khi được gọi.*

***→ Scope Chain*** *là quá trình mà biên dịch đi tìm chuỗi scope cho biến. Chuỗi này được tạo ra khi khai báo biến.*

## Hàm bất đồng bộ - asynchronous function

### Khái niệm bất đồng bộ

→ *Trong JS có 2 cơ chế chạy code.*

|  |  |
| --- | --- |
| *Tên cơ chế* | *Ví dụ* |
| *Đồng bộ (Synchronous)*  *Là* ***dòng code hoàn thành thì mới chạy dòng code tiếp theo****, nếu dòng code đang thực hiện quá lâu thì chương trình đang ở trạng thái chờ.* | console.log('Bắt đầu');  console.log('Kết thúc');  // Bắt đầu  // Kết thúc |
| *Bất đồng bộ (Asynchronous)*  *Có thể* ***bỏ qua trạng thái chờ*** *nếu không cần thiết và thực hiện các dòng code tiếp theo.* | console.log('Bắt đầu');  setTimeout(() => console.log('Đã đợi 2 giây'), 2000);  console.log('Kết thúc');  // Bắt đầu  // Kết thúc  // Đã đợi 2 giây |

Bảng 4.2 Cơ chế trong JS.

*→ Tại sao JS lại có thể xử lý bất đồng bộ: → Do nhờ sự phân chia nhiệm vụ của các phần như* ***Event Loop và các API của môi trường thực thi*** *(như trình duyệt hoặc Node.js).*

A screenshot of a computer

AI-generated content may be incorrect.

Hình 4.1 Mô phỏng cơ chế hoạt động của JS.

*→ Trong đó:*

* ***Call Stack****: Nơi lưu trữ các hàm đang được thực thi. Mỗi khi một hàm được gọi, nó được đẩy vào stack. Khi hàm hoàn thành, nó được lấy ra khỏi stack.*
* ***Web APIs (hoặc Node.js APIs)****: Các* ***tác vụ bất đồng bộ*** *như setTimeout, fetch, hoặc đọc file không được xử lý trực tiếp trong JavaScript mà được chuyển cho các API của môi trường thực thi (trình duyệt hoặc Node.js). Các API này chạy ngoài luồng chính.*
* ***Task Queue (Hàng đợi tác vụ)****: Khi một tác vụ bất đồng bộ hoàn thành (ví dụ: setTimeout hết thời gian hoặc fetch nhận được phản hồi), nó được đẩy vào hàng đợi tác vụ.*
* ***Event Loop****: Liên tục kiểm tra Call Stack và Task Queue. Nếu Call Stack trống,* ***Event Loop sẽ lấy tác vụ từ Task Queue và đẩy vào Call Stack để thực thi****.*

*→ Quy trình:*

* *Một tác vụ bất đồng bộ (như setTimeout hoặc fetch) được gửi đến Web APIs.*
* *Call Stack tiếp tục xử lý các tác vụ đồng bộ khác.*
* *Khi tác vụ bất đồng bộ hoàn thành, callback hoặc Promise được đẩy vào Task Queue.*
* *Event Loop chuyển các tác vụ từ Task Queue vào Call Stack khi stack trống.*

*→ Ví dụ:*

setTimeout(() => console.log(1), 2000);

console.log(2);

setTimeout(() => console.log(3), 0);

console.log(4);

→ Kết quả như hình 4.1.

*→ Giải thích:*

setTimeout(() ⇒ console.log(1), 2000) là tác vụ bất đồng bộ → Chuyển vào Web Apis.

→ Chuyển callback: () ⇒ console.log(1) vào callback queue → EveenLoop thấy có code đang xử lý ở Callback stack nên không lấy code CallBack queue thực hiện.

Console.log(2) → tác vụ đồng bộ → chuyển vào Callstack → Xử lý in ra 2. → thực hiện xóa Console.log(2).

setTimeout(() ⇒ console.log(3), 2000); là tác vụ bất đồng bộ → Chuyển vào Web Apis. → Chuyển callback: () ⇒ console.log(3) vào callback queue → EveenLoop thấy có code đang xử lý ở Callback stack nên không lấy code CallBack queue thực hiện.

Console.log(4) → tác vụ đồng bộ → chuyển vào Callstack → Xử lý in ra 4. → thực hiện xóa console.log(4).

→ Bây giờ khi CallStack trồng thì EnvenLoop sẽ thực hiện lấy các callback ở trong Queue ( Theo cơ chế FiFo ( vào đầu ra đầu )).

Ta thấy 1 vào trước và 3 vào sau suy ra.

→ Kết quả: 2 4 1 3

### Các hàm hay công việc cần cơ chế bất đồng bộ

|  |  |
| --- | --- |
| ***Công việc cơ chế bất đồng bộ*** | ***Các hàm liên quan*** |
| *Tác vụ liên quan đến thời gian* | * *setTimeout(callback, delay)* * *setInterval(callback, delay)* * *requestAnimationFrame(callback)* |
| *Gọi API / Yêu cầu mạng (Networking)* | * *fetch(url) → trả về Promise* * *XMLHttpRequest (cách cũ)* * *Các thư viện HTTP như axios.get(...)* * *WebSocket events (onmessage, onopen, onclose)* |
| *Tác vụ I/O (Input/Output)* | * ***Trong Node.js****:* * *Đọc/ghi file: fs.readFile, fs.writeFile (bản async)* * *Truy vấn cơ sở dữ liệu (MySQL, MongoDB, PostgreSQL…)* * *Gửi/nhận dữ liệu qua mạng (net, http, https modules)* * ***Trong trình duyệt****:* * *Đọc file từ <input type="file"> qua FileReader* * *Lưu trữ dữ liệu với IndexedDB API* |
| *Sự kiện người dùng (User Events)* | * *Click, scroll, keypress… (addEventListener)* * *Drag & drop* * *Touch events trên mobile* |
| *Tác vụ tính toán nặng nhưng cần tách luồng* | * *Xử lý ảnh, video, âm thanh* * *Mã hóa/giải mã dữ liệu* * *Machine Learning trong JS (TensorFlow.js, ONNX.js)* |

Bảng 4.3 Các công việc có cơ chế bất đồng bộ

### Tại sao lại cần bất đồng bộ khi làm việc với API, đọc file, hoặc setTimeout

Trong JavaScript, chương trình chạy theo **cơ chế đơn luồng (single-thread)**, tức là nó chỉ thực hiện **một việc tại một thời điểm**. Nếu ta viết code đồng bộ, thì:

* Khi bạn gọi **API** để lấy dữ liệu từ server → phải chờ vài giây.
* Khi bạn **đọc file lớn** → phải chờ file được đọc xong.
* Khi bạn dùng **setTimeout** → phải chờ hết thời gian trễ.

→ *Trong lúc chờ, chương trình* ***bị đứng lại, không làm gì khác****. Điều này gây trải nghiệm rất tệ cho người dùng (ứng dụng bị "đơ").*

**Bất đồng bộ (asynchronous)** giúp giải quyết điều này:

* Thay vì đứng chờ, JS sẽ **giao nhiệm vụ cho hệ thống** (ví dụ trình duyệt hoặc Node.js lo việc đọc file, gọi API).
* Trong lúc đó, chương trình tiếp tục chạy các lệnh khác.
* Khi có kết quả, hệ thống sẽ **báo lại** (*callback, promise, async/await*).

### Các cách xử lý bất đồng bộ

#### Callback - gọi lại khi xong

*→* ***Hàm được truyền vào hàm khác******như một đối số****, và sẽ được* ***gọi lại*** *khi tác vụ bất đồng bộ hoàn thành.*

*Ví dụ:*

setTimeout(*() ⇒ {*

*console.log("Đã hoàn thành sau 2 giây."); // Hàm này là callback*

}, 2000);

→ Trong ví dụ trên callback là một arrow function. Điều này giúp code gọn gàn hơn các cú pháp khai báo hàm khác.

→ *Khi hàm bất đồng bộ được gọi xong thì callback sẽ được thực hiện.*

*→ Callback là cách làm nhanh nhưng tìm ẩn rủi ro về callback hell (Việc callback bị lồng nhiều lần trong một bất đồng bộ, khó đọc và khó bảo trì.)*

#### Promise - “lời hứa” sẽ có kết quả.

→ ***Promise*** là ***một đối tượng đại diện cho kết quả của một tác vụ bất đồng bộ***,

→ Trạng thái:

* *pending → đang chờ*
* *fulfilled → thành công (resolve)*
* *rejected → thất bại (reject)*

*→ Promise giúp tránh callback hell và xử lý lỗi tốt hơn bằng cách* ***chaining*** *.then() và .catch().*

*→* ***Cú pháp khai báo Promise:***

***const promise*** *=* ***new Promise((resolve, reject)*** *⇒ {*

*// Thực hiện tác vụ bất đồng bộ ở đây*

*if (/\* thành công \*/) {*

***resolve(ketQua****); // báo hoàn thành và trả về dữ liệu*

*} else {*

***reject(loi);*** *// báo lỗi và trả về lý do*

*}*

*});*

*Trong đó:*

*→ new Promise(...):* ***tạo một Promise mới.***

*→ Nhận vào arrow function với hai tham số.*

* + - ***resolve(value)*** *→ gọi khi công việc thành công, trả về value.*
    - ***reject(error)*** *→ gọi khi công việc thất bại, trả về error.*

*Ví dụ:*

*const myPromise = new Promise((resolve, reject) ⇒ {*

*setTimeout(() ⇒ {*

*const randomNumber = Math.random();*

*if (randomNumber > 0.5) {*

*resolve(randomNumber); // Thành công*

*} else {*

*reject("Số quá nhỏ!"); // Thất bại*

*}*

*}, 2000);*

*});*

*Ở ví dụ trên: khai báo biến myPromise để lưu trữ một Promise. Hàm bất đồng bộ setTimeout() chạy xong trong 2 giây. Nếu không có lỗi logic thì nó trả resolve*

*→* ***Cú pháp để sử dụng Promise****: Có hai cách chính để xử lý Promise là* ***sử dụng .then() / .catch()*** *hoặc* ***async/await. → đẩy các công việc vào Callback Queue (nếu là .then()).***

* ***.then () / .cath () /. finally () – bắt kết quả mà promise trả về***

→ *Cú pháp:*

***promise // tên hàm bọc Promise hoặc biến được gán hàm chứa Promise***

*.****then((result)*** *⇒ { // Xử lý khi Promise thành công (resolve) })*

***.catch((error)*** *⇒ { // Xử lý khi Promise thất bại (reject) })*

***.finally(()*** *⇒ { // (Tùy chọn) Chạy bất kể thành công hay thất bại });*

*Ví dụ 1:*

myPromise

  .then(result ⇒ console.log(result))

  .catch(error ⇒ console.error(error));

.finally(() ⇒ { console.log("Promise đã hoàn tất!"); });

*Ví dụ 2:*

function getData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve("Dữ liệu đã nhận");

}, 2000);

});

}

getData()

.then((result) => console.log(result))

.catch((err) => console.error(err));

***→ Có thể chain nhiều .then() để xử lý tuần tự.***

* ***async/await () – cách viết gọn hơn (nên dùng)***

→ *Dùng từ khóa* ***async để khai báo hàm****, và* ***await để chờ kết quả Promise.***

*→ Cú pháp khai báo:*

***async*** *function tenHam() {*

***try {***

*const result =* ***await******promise****; // Xử lý kết quả*

*}*

***catch*** *(error) {*

*// Xử lý lỗi*

*}*

*}*

*Trong đó:*   
 *→* ***Từ khóa async*** *trước* ***hàm đảm bảo hàm trả về*** *một* ***Promise****.*

*→ Cho phép sử dụng await bên trong hàm.*

*→* ***await tạm dừng thực thi hàm cho đến khi Promise*** *được resolve hoặc reject.*

*→ Chỉ dùng await bên trong hàm async.*

→ *Sử dụng thêm từ khóa try .. catch để có thể bắt lỗi.*

Ví dụ:

*async function handlePromise() {*

*try {*

*const result = await myPromise;*

*console.log("Thành công:", result);*

*} catch (error) {*

*console.log("Lỗi:", error);*

*} finally {*

*console.log("Promise đã hoàn tất!");*

*}*

*}*

*handlePromise(); // Gọi hàm*

# DOM manipulation – Thao tác trên DOM

→ Là việc dùng JS để **tìm, thêm, sửa, xoá, thay đổi nội dung hoặc style** của các phần tử trong HTML.

→ Các phương thức thao tác trên dom phổ biến:

|  |  |  |
| --- | --- | --- |
| ***Phương thức*** | ***Mô tả*** | ***Ví dụ*** |
| *getElementById(id)* | *Chọn phần tử theo ID. Trả về một element duy nhất.* | document.getElementById('title') |
| *getElementsByClassName(className)* | *Chọn các phần tử theo class. Trả về HTMLCollection (giống mảng).* | document.getElementsByClassName('content') |
| *getElementsByTagName(tagName)* | *Chọn theo thẻ HTML. Trả về HTMLCollection.* | document.getElementsByTagName('p') |
| *querySelector(selector)* | *Chọn phần tử đầu tiên theo CSS selector (rất linh hoạt).* | document.querySelector('#title') hoặc document.querySelector('.content') |
| *querySelectorAll(selector)* | *Chọn tất cả phần tử theo CSS selector. Trả về NodeList (có thể lặp như mảng).* | document.querySelectorAll('p') |

Bảng 5.1 Các phương thức thao tác trên DOM

## QuerySlector

→ *Là một phương thức được sử dụng để chọn phần tử HTML đầu tiên trong tài liệu (document) khớp với một bộ chọn CSS (CSS selector) được chỉ định.*

*→ Cú pháp:*

*element.querySelector(selector)*

*Trong đó:*

*element →* ***Phần tử gốc*** *mà bạn muốn bắt đầu tìm kiếm (****thường là document****, nhưng có thể là bất kỳ phần tử DOM nào).*

*→* ***document*** *là* ***đối tượng đại diện cho toàn bộ tài liệu HTML*** *đang được trình duyệt tải và hiển thị.* *Nó là* ***gốc của cây DOM*** *(Document Object Model) — từ đây bạn có thể truy cập, đọc và thay đổi mọi phần tử, nội dung, hoặc cấu trúc của trang web.*

*Các selector css:*

|  |  |
| --- | --- |
| *id* | *#myid* |
| *class* | *.myclass* |
| *tag* | *\*,p,button* |
| *bộ chọn kết hợp* | *div p* |
| *Pseudo-classes : Lớp giả* | *a:hover* |
| *Phần tử giả* | *p::after* |

## Event listeners