

arrow functions



Arrow functions trong JavaScript là một cách viết gọn gàng hơn để khai báo hàm, được giới thiệu từ ES6 (ECMAScript 2015). Dưới đây là các đặc điểm chính và ví dụ về arrow functions:

Cú pháp cơ bản

javascript

Copy code

```
const functionName = (parameters) => {
  // Code block
  return value;
};
```



Nếu hàm chỉ có một biểu thức trả về, bạn có thể bỏ `return` và `{}`:

javascript

Copy code

```
const add = (a, b) => a + b;
console.log(add(2, 3)); // Output: 5
```

Nếu hàm không có tham số, bạn dùng `()`:

javascript

Copy code

```
const sayHello = () => 'Hello!';
console.log(sayHello()); // Output: Hello!
```



Nếu hàm có một tham số, bạn có thể bỏ luôn `()`:

javascript

Copy code

```
const square = x => x * x;
console.log(square(4)); // Output: 16
```

Các đặc điểm nổi bật

- Không có `this` riêng: Arrow function không tạo binding mới cho `this`. Thay vào đó, nó kế thừa `this` từ ngữ cảnh bao quanh (lexical scope).

javascript

Copy code

```
function Person() {
  this.name = "John";
  setTimeout(() => {
    console.log(this.name); // 'John', kế thừa this từ Person
  }, 1000);
}
new Person();
```



Nếu dùng hàm thông thường:

javascript

Copy code

```
function Person() {
  this.name = "John";
  setTimeout(function() {
    console.log(this.name); // undefined, vì this trong hàm thông thường là global (hoàn toàn)
  }, 1000);
}
new Person();
```

2. Không thể dùng làm constructor: Arrow functions không thể được sử dụng với `new`. Cố gắng làm vậy sẽ gây lỗi:

javascript

Copy code

```
const MyFunction = () => {};
const obj = new MyFunction(); // Error: MyFunction is not a constructor
```

3. Không có `arguments`: Arrow functions không có đối tượng `arguments` như hàm thông thường. Nếu cần, bạn có thể dùng toán tử rest (`...`):

javascript

Copy code

```
const logArgs = (...args) => console.log(args);
logArgs(1, 2, 3); // Output: [1, 2, 3]
```

4. Gọn gàng hơn trong các trường hợp đơn giản: Arrow functions thường được dùng trong các callback hoặc các thao tác như `map`, `filter`, `reduce`:

javascript

Copy code

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // Output: [2, 4, 6, 8]
```

Khi nào không nên dùng arrow functions?

- Khi cần hàm có `this` riêng.
- Khi cần dùng hàm làm constructor.
- Khi cần sử dụng đối tượng `arguments`.

Nếu bạn có thêm câu hỏi hoặc muốn giải thích kỹ hơn về một khía cạnh nào đó, cứ cho mình biết nhé! 😊

Array methods: map, filter, reduce, forEach



Dưới đây là giải thích và ví dụ chi tiết về các phương thức phổ biến của mảng trong JavaScript: `map`, `filter`, `reduce`, và `forEach`.

1. `map()`

Phương thức `map` tạo một mảng mới bằng cách áp dụng hàm callback lên từng phần tử của mảng ban đầu.

Cú pháp:

javascript

Copy code

```
array.map(callback(element, index, array), thisArg);
```

- `callback`: Hàm được gọi cho mỗi phần tử.
 - `element`: Giá trị hiện tại.
 - `index`: Chỉ số của phần tử (tùy chọn).
 - `array`: Mảng gốc (tùy chọn).

- `thisArg`: Giá trị để làm `this` khi thực thi `callback` (tùy chọn).

Ví dụ:

javascript

 Copy code

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // Output: [2, 4, 6]
```

2. `filter()`

Phương thức `filter` tạo một mảng mới với những phần tử thỏa mãn điều kiện trong hàm callback.

Cú pháp:

javascript

 Copy code

```
array.filter(callback(element, index, array), thisArg);
```

Ví dụ:

javascript

 Copy code

```
const numbers = [1, 2, 3, 4, 5, 6];
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // Output: [2, 4, 6]
```

3. `reduce()`

Phương thức `reduce` áp dụng hàm callback lần lượt cho từng phần tử trong mảng để tính toán và trả về một giá trị duy nhất.

Cú pháp:

javascript

 Copy code

```
array.reduce(callback(accumulator, currentValue, index, array), initialValue);
```

• `callback`:

- `accumulator`: Giá trị tích lũy (kết quả tạm thời).
- `currentValue`: Giá trị hiện tại trong mảng.
- `index`: Chỉ số của phần tử hiện tại (tùy chọn).
- `array`: Mảng gốc (tùy chọn).

• `initialValue`: Giá trị khởi tạo cho `accumulator` (tùy chọn).

Ví dụ:

javascript

 Copy code

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, num) => acc + num, 0);
console.log(sum); // Output: 10
```

4. `forEach()`

Phương thức `forEach` thực thi một hàm callback cho mỗi phần tử của mảng, nhưng không trả về mảng mới.

Cú pháp:

```
javascript
```

 Copy code

```
array.forEach(callback(element, index, array), thisArg);
```

Ví dụ:

```
javascript
```

 Copy code

```
const numbers = [1, 2, 3, 4];
numbers.forEach(num => console.log(num * 2));
// Output:
// 2
// 4
// 6
// 8
```

So sánh nhanh

Phương thức	Trả về mảng mới	Chỉnh sửa mảng gốc	Công dụng chính
map	Có	Không 	Biến đổi từng phần tử trong mảng.
filter	Có	Không	Lọc phần tử dựa trên điều kiện.
reduce	Không	Không	Tính toán một giá trị duy nhất từ mảng.
forEach	Không	Không	Thực hiện thao tác cho từng phần tử.

Nếu bạn cần ví dụ cụ thể hơn hoặc muốn giải thích kỹ hơn, cứ nói mình nhé! 😊

Destructuring và Rest/Spread operators.



1. Destructuring (Phân rã)

Destructuring trong JavaScript cho phép bạn trích xuất giá trị từ mảng hoặc đối tượng và gán chúng vào các biến một cách ngắn gọn.

1.1. Destructuring Array

Gán các giá trị từ mảng vào các biến.

Ví dụ:

```
javascript
```

 Copy code

```
const numbers = [1, 2, 3];
const [a, b, c] = numbers;

console.log(a); // Output: 1
console.log(b); // Output: 2
console.log(c); // Output: 3
```

Bỏ qua giá trị:

```
javascript
```

 Copy code

```
const numbers = [1, 2, 3, 4];
const [first, , third] = numbers;
```

```
console.log(first); // Output: 1
console.log(third); // Output: 3
```

Gán giá trị mặc định:

javascript

 Copy code

```
const numbers = [1];
const [a, b = 2] = numbers;

console.log(a); // Output: 1
console.log(b); // Output: 2
```



1.2. Destructuring Object

Gán giá trị từ thuộc tính của đối tượng vào các biến.

Ví dụ:

javascript

 Copy code

```
const person = { name: "Alice", age: 25 };
const { name, age } = person;

console.log(name); // Output: Alice
console.log(age); // Output: 25
```



Đổi tên biến:

javascript

 Copy code

```
const person = { name: "Alice", age: 25 };
const { name: userName, age: userAge } = person;

console.log(userName); // Output: Alice
console.log(userAge); // Output: 25
```

Gán giá trị mặc định:

javascript

 Copy code

```
const person = { name: "Alice" };
const { name, age = 30 } = person;

console.log(name); // Output: Alice
console.log(age); // Output: 30
```



1.3. Destructuring trong Hàm

javascript

 Copy code

```
const person = { name: "Alice", age: 25 };

function greet({ name, age }) {
  console.log(`Hello ${name}, you are ${age} years old.`);
}

greet(person); // Output: Hello Alice, you are 25 years old.
```

2. Rest và Spread Operators

Cả hai đều sử dụng cú pháp ..., nhưng chức năng thì khác nhau.



2.1. Rest Operator

Dùng để gộp các giá trị còn lại vào một mảng hoặc đối tượng.

Rest trong Mảng:

javascript

Copy code

```
const [a, ...rest] = [1, 2, 3, 4];
console.log(a); // Output: 1
console.log(rest); // Output: [2, 3, 4]
```



Rest trong Object:

javascript

Copy code

```
const { name, ...other } = { name: "Alice", age: 25, city: "Paris" };
console.log(name); // Output: Alice
console.log(other); // Output: { age: 25, city: "Paris" }
```



Rest trong Hàm:

javascript

Copy code

```
function sum(...numbers) {
  return numbers.reduce((acc, num) => acc + num, 0);
}

console.log(sum(1, 2, 3)); // Output: 6
```



2.2. Spread Operator

Dùng để trai các phần tử của mảng hoặc thuộc tính của đối tượng.

Spread trong Mảng:

javascript

Copy code

```
const numbers = [1, 2, 3];
const newNumbers = [...numbers, 4, 5];

console.log(newNumbers); // Output: [1, 2, 3, 4, 5]
```



Spread trong Object:

javascript

Copy code

```
const person = { name: "Alice", age: 25 };
const updatedPerson = { ...person, city: "Paris" };

console.log(updatedPerson);
// Output: { name: "Alice", age: 25, city: "Paris" }
```



Spread trong Hàm:

javascript

Copy code

```
const numbers = [1, 2, 3];
console.log(Math.max(...numbers)); // Output: 3
```



So sánh Rest và Spread

Operator	Công dụng chính	Ví dụ
Rest	Gộp các giá trị còn lại thành mảng hoặc đối tượng.	const [a, ...rest] = [1, 2, 3];

Nếu cần ví dụ cụ thể hơn hoặc giải thích chi tiết thêm, cứ cho mình biết nhé! 😊

Template literals.



Template Literals (Chuỗi mẫu)

Template literals là cách viết chuỗi linh hoạt hơn trong JavaScript, được giới thiệu từ ES6. Chúng được bao bọc bởi dấu backticks (`), thay vì dấu ngoặc kép (" hoặc ') , và cung cấp nhiều tính năng mạnh mẽ.



1. Cú pháp

javascript

Copy code

```
const message = `Đây là một chuỗi template literal.`;
```

2. Tính năng chính

2.1. Nội suy biến (Interpolation)



Template literals cho phép bạn nhúng giá trị của biến hoặc biểu thức vào chuỗi bằng cú pháp \${} .

Ví dụ:

javascript

Copy code

```
const name = "Alice";
const age = 25;
const message = `Tên: ${name}, Tuổi: ${age}`;

console.log(message);
// Output: Tên: Alice, Tuổi: 25
```

Bạn cũng có thể sử dụng biểu thức bên trong \${} :

javascript



Copy code

```
const a = 5;
const b = 10;
console.log(`Tổng của ${a} và ${b} là ${a + b}.`);
// Output: Tổng của 5 và 10 là 15.
```

2.2. Chuỗi nhiều dòng (Multiline Strings)

Template literals cho phép bạn viết chuỗi nhiều dòng mà không cần sử dụng ký tự thoát (\n).

Ví dụ:

javascript

Copy code

```
const message = `Dòng 1
Dòng 2
Dòng 3`;
```

```
console.log(message);
// Output:
// Dòng 1
// Dòng 2
// Dòng 3
```



2.3. Nhúng hàm (Tagged Templates)

Tagged templates cho phép bạn xử lý nội dung chuỗi bằng một hàm.

Cú pháp:

```
javascript
function tag(strings, ...values) {
    console.log(strings); // Mảng các phần tử của chuỗi
    console.log(values); // Mảng các giá trị được nội suy
}

const name = "Alice";
const age = 25;
tag`Tên: ${name}, Tuổi: ${age}`;
// Output:
// ["Tên: ", ", Tuổi: ", ""]
// ["Alice", 25]
```

Copy code



Bạn có thể sử dụng tagged templates để xử lý hoặc tùy chỉnh chuỗi đầu ra (VD: mã hóa, định dạng, dịch ngôn ngữ).

2.4. Nhúng biểu thức phức tạp

Template literals hỗ trợ nhúng các biểu thức tệp, như gọi hàm, toán tử logic, hoặc biểu thức điều kiện.

Ví dụ:

```
javascript
const loggedIn = true;
const message = `Trạng thái: ${loggedIn ? "Đang nhập" : "Chưa đăng nhập"}`;

console.log(message);
// Output: Trạng thái: Đang nhập
```

Copy code



Lợi ích của Template Literals

1. **Dễ đọc và dễ viết:** Đặc biệt với chuỗi nhiều dòng hoặc chuỗi có nội suy biến.
2. **Tối ưu cú pháp:** Tránh việc nối chuỗi thủ công bằng dấu `+`.
3. **Linh hoạt:** Cho phép nhúng biểu thức, gọi hàm, và xử lý chuỗi dễ dàng hơn.

Nếu bạn muốn thêm ví dụ hoặc giải thích chi tiết về phần nào, cứ nói nhé! 😊



Modules (import/export).



Modules trong JavaScript

modules trong JavaScript

Modules trong JavaScript cho phép bạn chia code thành các file riêng biệt để quản lý tốt hơn, tái sử dụng, và duy trì dễ dàng. Từ ES6 (ECMAScript 2015), JavaScript hỗ trợ `import` và `export` để làm việc với modules.

1. Export

Dùng `export` để xuất ra các biến, hàm, lớp, h  đối tượng từ một module.

1.1. Export Cụ thể (Named Export)

Cho phép xuất nhiều giá trị từ một file.

Ví dụ:

```
javascript   
// math.js  
export const add = (a, b) => a + b;  
export const subtract = (a, b) => a - b;  
export const PI = 3.14;
```

Import:

```
javascript   
// main.js  
import { add, subtract, PI } from './math.js';  
  
console.log(add(2, 3)); // Output: 5  
console.log(subtract(5, 2)); // Output: 3  
console.log(PI); // Output: 3.14
```

Bạn có thể đổi tên khi import bằng từ khóa `as`:

```
javascript   
import { add as sum, PI as piValue } from './math.js';  
  
console.log(sum(2, 3)); // Output: 5  
console.log(piValue); // Output: 3.14
```

1.2. Export Mặc Định (Default Export)

Một module chỉ có thể có **một** giá trị `default export`, thường được dùng khi module chỉ xuất ra một thứ chính.

Ví dụ:

```
javascript   
// Logger.js  
export default function log(message) {  
  console.log(message);  
}
```

Import:

```
javascript   
// main.js  
import log from './logger.js';  
  
log('Hello, World!'); // Output: Hello, World!
```

Bạn có thể đặt bất kỳ tên nào khi import default export:

```
javascript Copy code  
  
import customLogger from './logger.js';  
  
customLogger('This is a custom log!'); // Output: This is a custom log!
```

1.3. Kết hợp Export

Bạn có thể kết hợp **named export** và **default export** trong một module.

Ví dụ:

```
javascript Copy code  
  
// math.js  
export const add = (a, b) => a + b;  
export const subtract = (a, b) => a - b;  
export default function multiply(a, b) {  
    return a * b;  
}
```

Import:

```
javascript Copy code  
  
// main.js  
import multiply, { add, subtract } from './math.js';  
  
console.log(multiply(2, 3)); // Output: 6  
console.log(add(2, 3)); // Output: 5  
console.log(subtract(5, 2)); // Output: 3
```

2. Import

Dùng `import` để lấy các giá trị từ module khác vào file hiện tại.

2.1. Import Tất Cá

Bạn có thể import tất cả các export từ module bằng cú pháp `*` và đặt chúng dưới một namespace.

Ví dụ:

```
javascript Copy code  
  
// math.js  
export const add = (a, b) => a + b;  
export const subtract = (a, b) => a - b;  
export const PI = 3.14;
```

```
javascript Copy code  
  
// main.js  
import * as math from './math.js';  
  
console.log(math.add(2, 3)); // Output: 5  
console.log(math.subtract(5, 2)); // Output: 3  
console.log(math.PI); // Output: 3.14
```

3. Tính năng quan trọng

- Modules là chế độ strict mode mặc định: Bạn không cần khai báo `"use strict"`.
- Modules được thực thi một lần duy nhất: Nếu một module được import nhiều lần, nó chỉ được

thực thi một lần và các import khác sẽ dùng lại kết quả đã được tạo.

- Dùng đường dẫn tương đối hoặc tuyệt đối

- `import { add } from './math.js';` (đường dẫn tương đối)
- `import { add } from '/modules/math.js';` (đường dẫn tuyệt đối)

4. Ví dụ tổng hợp

File `utils.js`:

```
javascript
export const greet = (name) => `Hello, ${name}!`;
export const farewell = (name) => `Goodbye, ${name}!`;
export default function saySomething(message) {
  console.log(message);
}
```

Copy code

File `main.js`:

```
javascript
import saySomething, { greet, farewell } from './utils.js';

console.log(greet('Alice')); // Output: Hello, Alice!
console.log(farewell('Alice')); // Output: Goodbye, Alice!
saySomething('Have a great day!'); // Output: Have a great day!
```

Copy code

Nếu bạn cần giải thích sâu hơn hoặc có tình huống cụ thể, cứ hỏi mình nhé! 😊

< 2/2 >

Promises và async/await.



Promises và Async/Await trong JavaScript

JavaScript sử dụng Promises và async/await để lý các tác vụ bất đồng bộ (asynchronous), giúp code dễ đọc và tránh các vấn đề như callback hell.

1. Promises

Promise là một đối tượng đại diện cho một tác vụ bất đồng bộ, có thể:

- **Hoàn thành thành công:** Trả về giá trị (resolved).
- **Thất bại:** Trả về lỗi (rejected).

Cú pháp:

```
javascript
const promise = new Promise((resolve, reject) => {
  // Xử lý bất đồng bộ
  if (/^thành công */) {
    resolve("Kết quả thành công");
  } else {
    reject("Có lỗi xảy ra");
  }
});
```

Copy code

Trạng thái của Promise:

1. Pending: Đang chờ xử lý.
2. Fulfilled: Thành công (kết quả được trả về qua `resolve`). 
3. Rejected: Thất bại (lỗi được trả về qua `reject`).

Sử dụng `.then()` và `.catch()`:

- `then()`: Được gọi khi Promise hoàn thành thành công.
- `catch()`: Được gọi khi Promise thất bại.

Ví dụ:

```
javascript  Copy code   
  
const fetchData = () => {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      const success = true;  
      if (success) {  
        resolve("Dữ liệu đã tải xong");  
      } else {  
        reject("Lỗi khi tải dữ liệu");  
      }  
    }, 1000);  
  });  
  
fetchData()  
  .then(data => {  
    console.log(data); // Output: Dữ liệu đã tải xong  
  })  
  .catch(error => {  
    console.error(error); // Output: Lỗi khi tải dữ liệu (nếu có lỗi)  
  });  
  

```

Chaining Promises:

Bạn có thể nối tiếp nhiều Promises để thực hiện các tác vụ liên tiếp.

Ví dụ:

```
javascript  Copy code   
  
const step1 = () => Promise.resolve("Bước 1 hoàn thành");  
const step2 = () => Promise.resolve("Bước 2 hoàn thành");  
  
step1()  
  .then(result => {  
    console.log(result); // Output: Bước 1 hoàn thành  
    return step2();  
  })  
  .then(result => {  
    console.log(result); // Output: Bước 2 hoàn thành  
  })  
  .catch(error => {  
    console.error(error);  
  });  
  

```

2. Async/Await

Async/Await là cú pháp giúp xử lý Promises một cách trực quan và dễ đọc hơn. Nó hoạt động như một cách viết đồng bộ nhưng thực hiện bất đồng bộ.

Cú pháp:

- Hàm được khai báo với từ khóa `async` luôn trả về một Promise.
- `await` dừng việc thực thi của hàm cho đến khi Promise được hoàn thành.

Ví dụ:

```
javascript ↓ Copy code

const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = true;
      if (success) {
        resolve("Dữ liệu đã tải xong");
      } else {
        reject("Lỗi khi tải dữ liệu");
      }
    }, 1000);
  });
}

const getData = async () => {
  try {
    const data = await fetchData();
    console.log(data); // Output: Dữ liệu đã tải xong
  } catch (error) {
    console.error(error); // Output: Lỗi khi tải dữ liệu (nếu có lỗi)
  }
};

getData();
```

3. So sánh Promises và Async/Await

Đặc điểm	Promises	Async/Await
Cú pháp	Dựa trên <code>.then()</code> và <code>.catch()</code> .	Dùng <code>async</code> và <code>await</code> .
Độ dễ đọc	Khó đọc khi có nhiều Promise chaining.	Dễ đọc và gần giống cách viết code đồng bộ.
Xử lý lỗi	Sử dụng <code>.catch()</code> .	Sử dụng <code>try...catch</code> .
Hiệu quả	Hiệu quả như nhau (đều sử dụng Promises bên trong).	Hiệu quả như Promises.

4. Ví dụ kết hợp

Kịch bản:

- Tải danh sách người dùng từ một API giả lập.
- Tải thông tin chi tiết của từng người dùng.

Promise:

```
javascript ↓ Copy code

const fetchUsers = () => {
  return Promise.resolve(["Alice", "Bob", "Charlie"]);
};

const fetchUserDetails = user => {
  return Promise.resolve(`Thông tin chi tiết của ${user}`);
};
```

```

fetchUsers()
  .then(users => {
    console.log(users); // Output: ["Alice", "Bob", "Charlie"]
    return fetchUserDetails(users[0]);
  })
  .then(details => {
    console.log(details); // Output: Thông tin chi tiết của Alice
  })
  .catch(error => {
    console.error(error);
  });

```

Async/Await:

```

javascript
Copy code

const fetchUsers = () => {
  return Promise.resolve(["Alice", "Bob", "Charlie"]);
};

const fetchUserDetails = user => {
  return Promise.resolve(`Thông tin chi tiết của ${user}`);
};

const getUserInfo = async () => {
  try {
    const users = await fetchUsers();
    console.log(users); // Output: ["Alice", "Bob", "Charlie"]

    const details = await fetchUserDetails(users[0]);
    console.log(details); // Output: Thông tin chi tiết của Alice
  } catch (error) {
    console.error(error);
  }
};

getUserInfo();

```

5. Một số lưu ý

- `await` chỉ sử dụng được trong hàm có `async`.
- Chạy nhiều tác vụ song song: Sử dụng `Promise.all` để tăng hiệu suất.

Ví dụ:

```

javascript
Copy code

const task1 = () => Promise.resolve("Tác vụ 1 hoàn thành");
const task2 = () => Promise.resolve("Tác vụ 2 hoàn thành");

const runTasks = async () => {
  const [result1, result2] = await Promise.all([task1(), task2()]);
  console.log(result1); // Output: Tác vụ 1 hoàn thành
  console.log(result2); // Output: Tác vụ 2 hoàn thành
};

runTasks();

```

Nếu bạn cần thêm ví dụ hoặc giải thích, cứ hỏi nhé! 😊



DOM Manipulation trong JavaScript

DOM (Document Object Model) là một giao diện lập trình cho phép bạn tương tác và thay đổi cấu trúc của tài liệu HTML hoặc XML. Trong JavaScript, bạn có thể thao tác với DOM để thay đổi nội dung, cấu trúc, hoặc kiểu dáng của trang web. JavaScript cung cấp các phương thức để truy cập và sửa đổi các phần tử trong DOM, giúp bạn tạo ra các trang web động và tương tác.



1. DOM là gì?

- DOM là một đối tượng đại diện cho toàn bộ trang web dưới dạng cây đối tượng.
- Mỗi phần tử HTML (như `<div>`, `<p>`, ``) trong trang web sẽ là một "node" trong cây DOM.
- JavaScript có thể truy cập và thay đổi các nodes này thông qua DOM API.

2. Truy cập các phần tử trong DOM

Bạn có thể sử dụng các phương thức của DOM API để truy cập các phần tử trong HTML.



2.1. getElementById()

Truy cập phần tử qua id.

javascript

Copy code

```
const element = document.getElementById('myId');
console.log(element);
```

2.2. getElementsByClassName()

Truy cập các phần tử qua class. (Trả về một `HTMLCollection`).

javascript

Copy code

```
const elements = document.getElementsByClassName('myClass');
console.log(elements); // Trả về một HTMLCollection
```

2.3. getElementsByTagName()

Truy cập các phần tử qua tag name. (Trả về một `HTMLCollection`).

javascript

Copy code

```
const elements = document.getElementsByTagName('p');
console.log(elements); // Trả về một HTMLCollection
```

2.4. querySelector()

Truy cập phần tử đầu tiên khớp với selector CSS (ID, class, hoặc tag name).

javascript

Copy code

```
const element = document.querySelector('.myClass');
console.log(element);
```

2.5. querySelectorAll()

Truy cập tất cả các phần tử khớp với selector CSS (Trả về một `NodeList`).

javascript

Copy code

```
const elements = document.querySelectorAll('.myClass');
console.log(elements); // Trả về một NodeList
```

3. Thao tác với DOM

3.1. Thay đổi nội dung

- `innerHTML` : Thay đổi nội dung HTML của một phần tử.
- `textContent` : Thay đổi nội dung văn bản của một phần tử.

javascript

 Copy code

```
const element = document.getElementById('myId');
element.innerHTML = '<strong>Đây là nội dung mới</strong>'; // Cập nhật HTML
element.textContent = 'Đây là nội dung v' ↓ 'bản mới'; // Cập nhật văn bản
```

3.2. Thêm hoặc loại bỏ phần tử

- `createElement()` : Tạo một phần tử mới.
- `appendChild()` : Thêm một phần tử con vào phần tử hiện tại.
- `removeChild()` : Loại bỏ một phần tử con khỏi phần tử hiện tại.

javascript

 Copy code

```
// Tạo một phần tử mới
const newElement = document.createElement('div');
newElement.textContent = 'Phần tử mới';

// Thêm phần tử mới vào DOM
document.body.appendChild(newElement);

// Loại bỏ phần tử
document.body.removeChild(newElement);
```

3.3. Thay đổi thuộc tính của phần tử

- `setAttribute()` : Thay đổi thuộc tính của phần tử.
- `getAttribute()` : Lấy giá trị thuộc tính của phần tử.
- `removeAttribute()` : Loại bỏ thuộc tính của phần tử.

javascript

 Copy code

```
const image = document.querySelector('img');

// Thay đổi thuộc tính src
image.setAttribute('src', 'new-image.jpg');

// Lấy giá trị thuộc tính alt
const altText = image.getAttribute('alt');
console.log(altText);

// Loại bỏ thuộc tính src
image.removeAttribute('src');
```

3.4. Thêm hoặc loại bỏ lớp CSS

- `classList.add()` : Thêm lớp CSS.
- `classList.remove()` : Loại bỏ lớp CSS.
- `classList.toggle()` : Đổi trạng thái của lớp CSS (nếu có thì loại bỏ, nếu không có thì thêm vào).

javascript

 Copy code

```
const element = document.getElementById('myElement');

// Thêm Lớp mới
```

```
element.classList.add('new-class');

// Loại bỏ Lớp
element.classList.remove('old-class');

// Đổi trạng thái Lớp
element.classList.toggle('active');
```

4. Xử lý sự kiện

JavaScript cho phép bạn thêm các sự kiện (event) như `click`, `keypress`, `mouseover`, `submit`, v.v.

4.1. Thêm sự kiện:

- `addEventListener()`: Cách hiện đại và linh hoạt để thêm sự kiện vào phần tử.

```
javascript ↓ Copy code

const button = document.getElementById('myButton');

// Thêm sự kiện click
button.addEventListener('click', function() {
  alert('Button clicked!');
});
```

4.2. Loại bỏ sự kiện:

- `removeEventListener()`: Loại bỏ một sự kiện đã được đăng ký.

```
javascript ↓ Copy code

const handleClick = function() {
  alert('Button clicked!');
};

button.addEventListener('click', handleClick);

// Sau này Loại bỏ sự kiện
button.removeEventListener('click', handleClick);
```

4.3. Sự kiện khác:

Bạn có thể thêm nhiều sự kiện khác nhau như `input`, `change`, `keydown`, `focus`, v.v.

```
javascript ↓ Copy code

const inputElement = document.getElementById('myInput');

// Sự kiện 'input'
inputElement.addEventListener('input', function(event) {
  console.log('Đã nhập:', event.target.value);
});
```

5. DOM Traversing (Duyệt qua DOM)

Bạn có thể di chuyển giữa các phần tử trong DOM bằng các thuộc tính như:

- `parentNode`: Truy cập phần tử cha.
- `childNodes`: Truy cập các phần tử con.
- `nextSibling`: Truy cập phần tử kế tiếp.
- `previousSibling`: Truy cập phần tử trước đó.

```
javascript ↓ Copy code
```

```
const element = document.getElementById('myElement');

// Truy cập phần tử cha
const parent = element.parentNode;

// Truy cập phần tử con
const firstChild = element.firstChild; ↓

// Truy cập phần tử kế tiếp
const nextElement = element.nextSibling;
```

6. Ví dụ tổng hợp

Thêm, sửa và loại bỏ phần tử trong DOM:

```
html
<!DOCTYPE html>
<html lang="vi">
<head>
    <meta charset="UTF-8">
    <title>DOM Manipulation Example</title>
</head>
<body>
    <div id="container">
        <button id="addBtn">Thêm phần tử</button>
    </div>

    <script>
        const addBtn = document.getElementById('addBtn');
        const container = document.getElementById('container');

        addBtn.addEventListener('click', function() {
            // Tạo phần tử mới
            const newDiv = document.createElement('div');
            newDiv.textContent = 'Đây là phần tử mới!';

            // Thêm phần tử vào container
            container.appendChild(newDiv);
        });
    </script>
</body>
</html>
```

Kết luận

DOM manipulation là công cụ mạnh mẽ giúp JavaScript tương tác với các phần tử HTML, từ việc thay đổi nội dung, thêm phần tử mới, đến xử lý các sự kiện của người dùng. Việc sử dụng DOM API hiệu quả có thể giúp bạn xây dựng các trang web động và tương tác.

Nếu bạn cần làm rõ thêm phần nào hoặc có ví dụ cụ thể, hãy cho mình biết nhé! 😊



Message ChatGPT



?