Tổng hợp kiến thức Javascript/Typescript dành cho QA/Tester

Về tài liệu này

Phiên bản

- 2024-09-15: phiên bản đầu tiên.
- 2025-02-19: Sửa chính tả.

Tác giả

Tài liệu này được tổng hợp, biên soạn và hiệu chỉnh bởi team Playwright Việt Nam. Bản quyền thuộc về team Playwright Việt Nam.
Mọi thắc mắc, xin liên hệ: playwrightvietnam@gmail.com/ dominhphong306@gmail.com

Độc giả

Đây là tài liệu dành riêng cho các lớp học của Playwright Việt Nam, các học viên trên Udemy và thành viên của cộng đồng Playwright Việt Nam.
Nội dung của tài liệu này phù hợp với các bạn làm Tester/QA/QE hoặc các bạn bắt đầu học về Javascript/Typescript.

Phân phối

Cộng đồng và hỗ trợ

- Cộng đồng Playwright Việt Nam - học automation test từ chưa biết gì: https://www.facebook.com/groups/playwright.automation.test
- Group hỗ trợ review code, support dành riêng cho học viên/ udemy student: https://www.facebook.com/groups/playwrightvietnam.support

Về ngôn ngữ Javascript

Là một ngôn ngữ lập trình, ra đời năm 1995 bởi Brendan Eich. Có người nói vui rằng: cái tên "Javascript" là ăn theo cái theo hot hit thời bấy giờ - Java.
Nếu có người hỏi: "Javascript" và "Java" có liên quan gì tới nhau không, thì câu trả lời là không bạn nhé.
Javascript là một ngôn ngữ đứng top 1 nhiều năm (> 5 năm), với kho thư viện phong phú và đồ sộ.

[Stackoverflow report]

Javascript

Học automation test từ chưa biết gì

- Javascript?
  - Fun fact: "Ăn theo" cái tên hot hit: Java

- Là một **ngôn ngữ lập trình**.
- Ra đời năm 1995 bởi **Brendan Eich**.
- Giúp cho browser hoạt động được.
- Top language: Stackoverflow report

Cài đặt môi trường, công cụ

Cài đặt NodeJS

Để code Javascript chạy được trên máy cá nhân, vui lòng cài đặt NodeJS trên máy.
Xem hướng dẫn tại trang chủ NodeJS: https://nodejs.org/en/download/package-manager

Cài đặt IDE

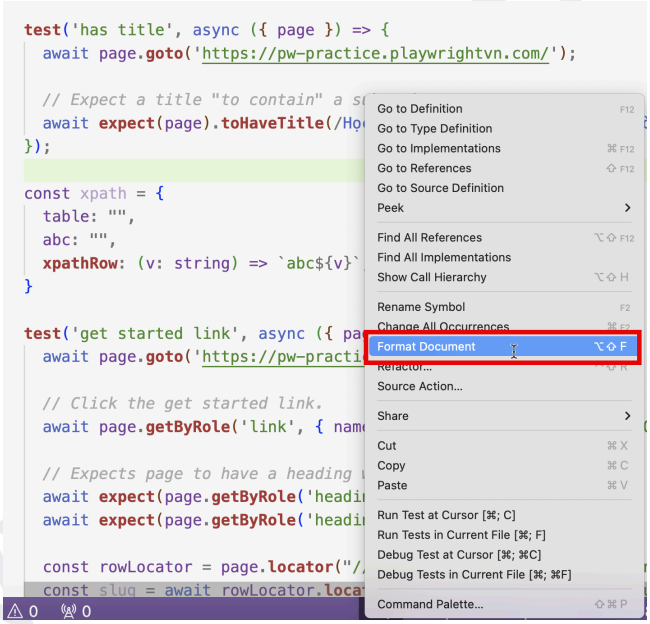
Để code Javascript thuận tiện hơn, ta sử dụng IDE (Integrated Development Environment) **Visual Studio Code**.
Để tải và cài đặt Visual Studio Code, vui lòng xem tại trang chủ: https://code.visualstudio.com/

Format code trong VS Code

Một thói quen rất cần khi code đó là format code. Format code giúp code sạch đẹp, được căn lề, dễ nhìn.
Để format code với Visual Studio Code, ta sử dụng hai cách:

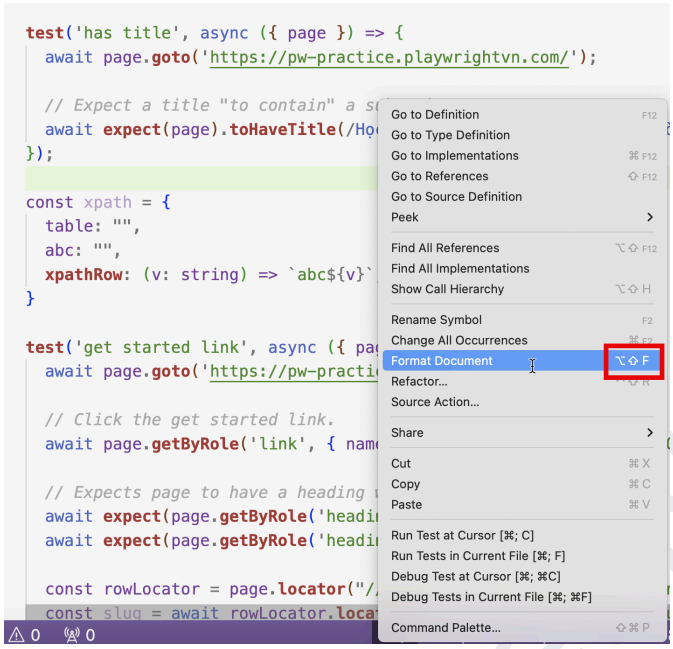
Format trực tiếp với Visual Studio Code

Tại màn hình code bất kì, bạn click chuột phải, chọn “Format document” để format lại file.



Sử dụng phím tắt

Bạn cũng có thể sử dụng phím tắt, được gợi ý ngay phía bên phải của menu:



Với Window, đó là: “Alt + Shift + F”.
Với Mac OS, sẽ là: “Option + Shift + F”.

Cấu hình tự động format

Bạn cũng có thể cấu hình để VSCode tự động format khi bạn lưu file code. Bạn có thể xem thêm tại đây: https://www.alphr.com/auto-format-vs-code/

Convention

Convention là các quy tắc định dạng viết hoa, viết thường, các kí tự cách nhau thế nào trong code. Convention giúp code dễ nhìn, đồng bộ giữa cả team với nhau.
Dưới đây là một số case thường gặp.

Loại case	Định nghĩa	Ví dụ
snake_case	`snake_case` là kiểu viết: tất cả các kí tự viết thường. Giữa các từ cách nhau bởi dấu gạch dưới (_). Cái tên `snake_case` bắt nguồn từ việc các dấu “_” giống như các khúc của con rắn, do đó được liên tưởng	Từ “`This is a variable`” sẽ được viết thành: `this_is_a_variable` Từ “`A simple constant`” sẽ được viết thành `a_simple_constant`

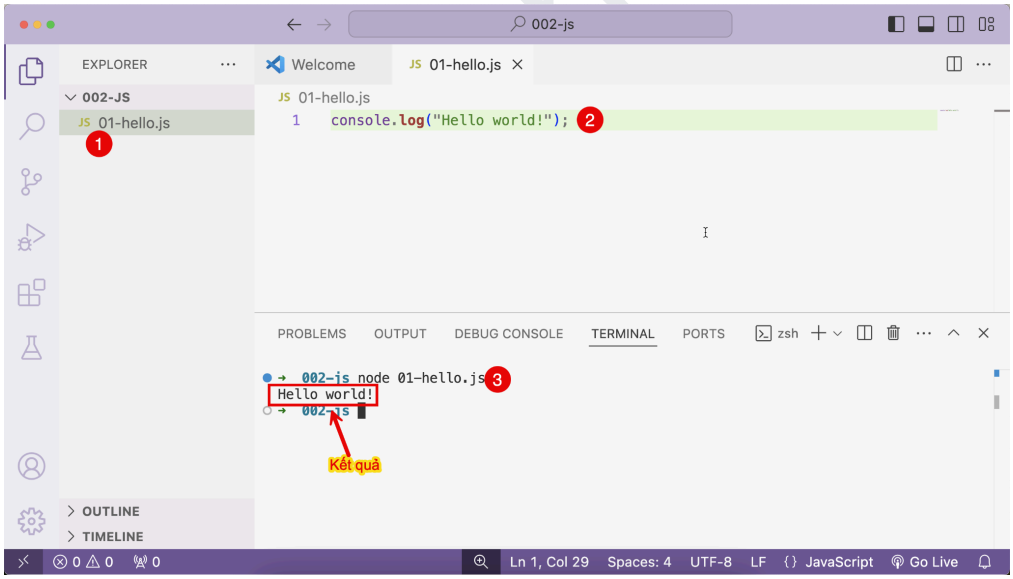
	đến hình dạng của một con rắn đang trườn. `snake_case` trong Typescript thường dùng để đặt tên các hằng số dùng chung. Ví dụ: `MAX_USER`, `NUMBER_OF_PRODUCT`	
kebab-case	`kebab-case` là kiểu viết: tất cả các kí tự viết thường. Giữa các từ cách nhau bởi dấu gạch ngang hay dấu trừ (-). Cái tên `kebab-case` bắt nguồn từ hình ảnh các thành phần của món kebab được xâu chuỗi lại với nhau bởi một que, tương tự như các từ trong kiểu viết này được nối với nhau bởi dấu gạch ngang. `kebab-case` thường dùng để đặt tên file, folder.	Từ `“This is a variable”` sẽ được viết thành: `this-is-a-variable` Từ `“A simple constant”` sẽ được viết thành `a-simple-constant`
camelCase	`camelCase` là kiểu viết mà từ đầu tiên viết thường, các từ tiếp theo bắt đầu bằng chữ cái viết hoa và không có dấu cách giữa các từ. Cái tên `camelCase` bắt nguồn từ hình ảnh của một con lạc đà (camel) với bướu lưng; các chữ cái viết hoa trong từ này giống như các "bướu" nổi lên giữa các từ. `camelCase` thường dùng để đặt tên biến, function, method.	Từ `“This is a variable”` sẽ được viết thành: `thisIsAVariable` Từ `“A simple function”` sẽ được viết thành: `aSimpleFunction`
PascalCase	`PascalCase` là kiểu viết tương tự như `camelCase`, nhưng điểm khác	Từ `“This is a class”` sẽ được viết thành: `ThisIsAClass`

	biệt là tất cả các từ đều bắt đầu bằng chữ cái viết hoa. Cái tên `PascalCase` bắt nguồn từ ngôn ngữ lập trình Pascal, nơi phong cách này được sử dụng phổ biến để đặt tên cho các biến và hàm. `PascalCase` thường dùng để đặt tên class, type, interface.	Từ `"A simple constant"` sẽ được viết thành: `ASimpleConstant`

Hello world

Có một fun fact là với tất cả các ngôn ngữ lập trình, khi học thì bài học kinh điển đầu tiên luôn là: in ra màn hình dòng chữ "Hello world!".
Để thực hiện in ra dòng chữ Hello world đầu tiên, ta làm như sau:

1. Tạo file: `"01-hello.js"`
2. Thêm vào file này nội dung: `console.log("Hello world!");`
3. Mở terminal lên, gõ dòng lệnh: `node 01-hello.js`
4. Quan sát kết quả



Giải thích:

- Bước 1: cần tạo file có đuôi `.js` để VS Code hiểu: đây là file Javascript, hãy hiển thị theo cú pháp của Javascript.
- Bước 2:

    - Dòng lệnh `console.log();` dùng để in ra màn hình dòng chữ phía trong
    - Lưu ý: luôn có dấu `;` ở cuối các câu lệnh. Dù điều này là không bắt buộc, nhưng đó là một thói quen tốt.
- Bước 3: Dòng lệnh `node <đường dẫn tới file>` dùng để chạy code nằm trong file tương ứng.
    - Lưu ý:
- Bước 4: Tại terminal, sau khi gõ dòng lệnh `node 01-hello.js`, ta thấy in ra màn hình dòng chữ `Hello world!`là kết quả của câu lệnh `console.log("Hello world!")`. Từ nay, nếu ta cần in chuỗi kí tự nào đó ra màn hình, hãy thay Hello world bằng chuỗi kí tự đó nhé.

Comment code

Comment code thường sử dụng để giải thích cho một đoạn code làm việc gì.
Comment code nói với Javascript rằng không nên chạy đoạn code đã được comment.

Comment cho một dòng

Comment cho một dòng được bắt đầu bởi `//`
Xét đoạn code sau:

```
1 | // Dòng dưới đây sẽ in ra lời chào Playwright Việt Nam
2 | console.log("Hello Playwright Việt Nam");
3 | // Dòng này cũng không chạy.
```

Trong đoạn code trên, dòng số 1 và dòng số 3 sẽ không được Javascript chạy, vì nó bắt đầu bởi `//`. Chỉ dòng số 2 được chạy.

Comment một dòng thường dùng để giải thích cho đoạn code ngắn phía dưới.
Trong ví dụ dưới đây là comment một dòng từ một thư viện nổi tiếng: dotenvx.

```
// for use with run
const envs = []
function collectEnvs (type) {
  return function (value, previous) {
    envs.push({ type, value })
    return previous.concat([value])
  }
}

// global log levels
program
  .option('-l, --log-level <level>', 'set log level', 'info')
  .option('-q, --quiet', 'sets log level to error')
  .option('-v, --verbose', 'sets log level to verbose')
  .option('-d, --debug', 'sets log level to debug')
  .hook('preAction', (thisCommand, actionCommand) => {
    const options = thisCommand.opts()

    setLogLevel(options)
  })
```

Comment cho nhiều dòng

Comment cho nhiều dòng được bắt đầu bởi `/*` và kết thúc bởi `*/`
Ví dụ:

```
/* Đây là
comment nằm trên
nhiều dòng
khác nhau */
```

Comment nhiều dòng thường nằm ở đầu các function, class, method, giúp giải thích một cách chi tiết hơn cách mà method, class hoạt động; một số ví dụ, các trường hợp nên sử dụng và reference.

```
/**
 * The method returns an element locator that can be used to perform actions on this
 page / frame. Locator is resolved
 * to the element immediately before performing an action, so a series of actions on the
 same locator can in fact be
 * performed on different DOM elements. That would happen if the DOM structure between
 those actions has changed.
 *
 * [Learn more about locators](https://playwright.dev/docs/locators).
 * @param selector A selector to use when resolving DOM element.
 * @param options
 */
locator(selector: string, options?: {
```

TODO: hướng dẫn viết comment code, viết comment cho hàm hiệu quả.

Khai báo

Biến

Biến = biến thiên = thay đổi.

Định nghĩa

Biến dùng để lưu trữ giá trị các dữ liệu, hay các đối tượng. Giá trị của biến **có thể được thay đổi, cập nhật** trong quá trình ứng dụng hoạt động. Mục này trình bày các vấn đề về biến như tên biến, khai báo và khởi tạo biến …

Quy tắc đặt tên biến

Tên biến do bạn đặt và khai báo, tên biến cần đảm bảo quy tắc sau:

- Tên biến được tạo ra bởi các ký tự chữ, số, `_` và `$`.
- Tên **không được phép bắt đầu bằng số** (chỉ bắt đầu bằng ký tự chữ hoặc $ hoặc _).
- **Không được** chứa các ký hiệu đặc biệt như ký hiệu toán học, logic (như +, -, *, >, < ...).
- **Không được** chứa khoảng trắng.
- **Không được** đặt tên biến trùng với các từ khóa dành riêng cho ngôn ngữ Javascript liệt kê ở dưới:

```
abstract   else       instanceof super
boolean    enum       int        switch
break      export     interface  synchronized
byte       extends    let        this
case       false      long       throw
catch      final      native     throws
char       finally    new        transient
class      float      null       true
const      for        package    try
continue   function   private    typeof
debugger   goto       protected  var
default    if         public     void
delete     implements return     volatile
do         import     short      while
double     in         static     with
```

Khai báo và khởi tạo biến

Để khai báo biến, ta có 2 từ khoá: var và let.
Ta có thể khai báo và gán giá trị ngay. Ví dụ:

```
var name = “Playwright Viet Nam”;
let major = “Học automation test từ chưa biết gì”;
```

Hoặc khai báo và gán giá trị sau:

```
var name;
let major;
name = “Playwright Việt Nam”;
major = “Học automation test từ chưa biết gì”;
```

Lưu ý: tên biến không được trùng nhau.

Sử dụng biến

Ta có thể sử dụng biến ngay trong hàm console.log:

```
console.log(name);
```

Thay đổi giá trị

Để thay đổi giá trị của biến, ta thực hiện gán lại giá trị của biến sang giá trị khác mà không cần từ khoá var/let.

```
var name = “Playwright”;
name = “Playwright Viet Nam”;
```

Hằng

Hằng = hằng số = không thay đổi.

Định nghĩa

Hằng số là các giá trị không thay đổi được sau khi khai báo.

Quy tắc đặt tên hằng

Giống với quy tắc đặt tên biến. Lưu ý tránh từ khoá.

Khai báo và khởi tạo hằng

Hằng số cần gán giá trị ngay tại thời điểm khai báo:

```
const framework = “Playwright”;
```

Thay đổi giá trị

Hằng số không thể thay đổi được giá trị. Nếu cố tình thay đổi, sẽ gây ra lỗi:

```
const course = "Youtube";
course = "Udemy";
//     ^

// TypeError: Assignment to constant variable.
```

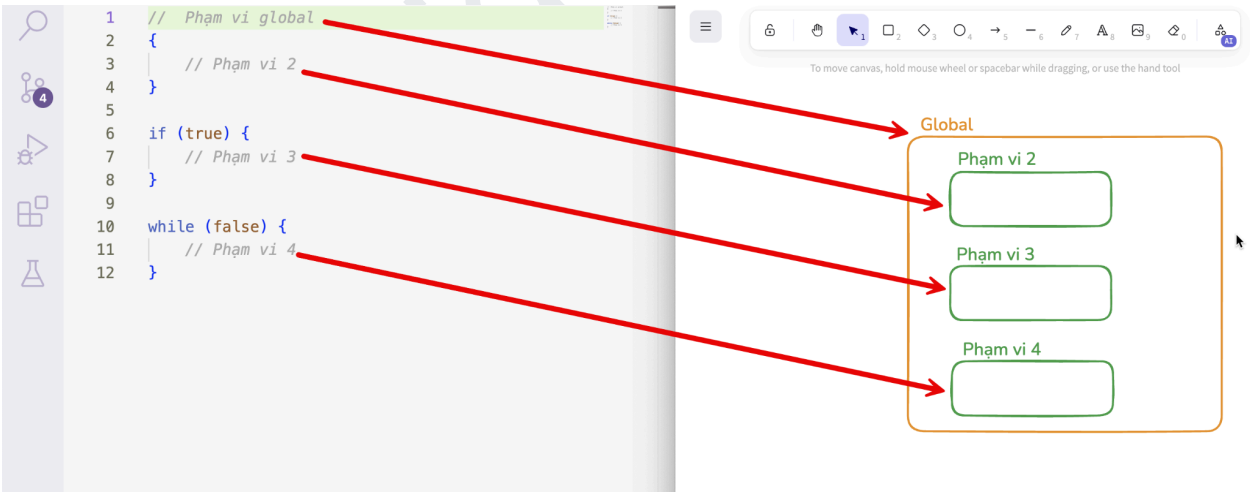
Lời khuyên khai báo

- Đối với các giá trị không thay đổi, dùng `const`.
- Khai báo biến sử dụng `let` để dễ kiểm soát phạm vi truy cập. Không dùng `var`.

Phạm vi của biến

Phạm vi (scope) là khu vực hoạt động của biến.
Trong Javascript, phạm vi được thể hiện thông qua các cặp ngoặc nhọn ({}).

Đoạn code không nằm trong cặp ngoặc nhọn nào gọi là phạm vi global (toàn cục).



Kiểu dữ liệu

Các nhóm

Javascript chia kiểu dữ liệu ra thành 2 nhóm:

- **Nhóm kiểu dữ liệu nguyên thuỷ**: đây là các kiểu dữ liệu được xây dựng sẵn trong Javascript. Có 8 kiểu dữ liệu trong nhóm này. Đó là:
  - `boolean`
  - `null`
  - `undefined`
  - `number`
  - `string`
  - `symbol`
- **Nhóm kiểu dữ liệu đối tượng**: đây là các kiểu dữ liệu cho người dùng tự tạo ra.

Nhóm kiểu dữ liệu nguyên thuỷ (primitive)

Kiểu logic - boolean

Biểu diễn giá trị đúng/sai:

- Giá trị đúng: `true`
- Giá trị sai: `false`

Thường dùng trong các biểu thức của câu điều kiện.
Ví dụ:

```
var isPlaywright = true;
let isCypress = false;
const isSelenium = false;
```

Kiểu null - rỗng

Kiểu null đại diện cho giá trị rỗng - không chứa gì cả. Ví dụ:

```
var lazy = null;
```

Kiểu undefined

Kiểu undefined đại diện cho giá trị chưa được định nghĩa. Ví dụ:

```
var name = undefined;
console.log(name);
```

```
let age;
console.log(age);
```

Kiểu number

Kiểu number đại diện cho các giá trị kiểu số. Trong Javascript, không phân biệt kiểu dữ liệu số nguyên, số thực mà gọi chung tất cả là “kiểu number”

```
var num1 = 12;
let num2 = 15.893;
const num3 = 0;
```

Kiểu string

Kiểu string đại diện cho các giá trị kiểu chuỗi, dạng văn bản. Có thể lưu giá trị kiểu chuỗi nằm trong cặp ngoặc kép hoặc ngoặc đơn:

```
var str1 = ‘simple’;
let str2 = “double”;
const str3 = ‘triple’;
```

Các kí tự đặc biệt

Đối với các kí tự đặc biệt, bạn cần dùng kí tự escape (`\`) đứng trước

- Kí tự \: `\\`
- Kí tự nháy đơn: `\’`
- Kí tự nháy kép: `\”`
- Kí tự xuống dòng: `\n`
- Kí tự đầu dòng: `\r`
- Kí tự tab: `\t`

Nhóm kiểu dữ liệu đối tượng

OOP và Object

Có thể bạn có từng nghe về khái niệm OOP (Object-Oriented Programming), hay lập trình hướng đối tượng. Object chính là trái tim của OOP.

Kiểu dữ liệu đối tượng, hay Object, là kiểu dữ liệu lưu trữ một **tập hợp giá trị** dưới dạng **key-value**. Object sẽ lưu dữ liệu theo thành từng “đối tượng”, khiến dữ liệu trở nên gọn gàng và dễ quản lý hơn.

Cú pháp

Để sử dụng Object, ta dùng cú pháp khai báo với let hoặc const:

```
let/const <object_name> = {
 'key1': value1,
 'key2': value2,
 …
 'keyN': valueN
}
```

Trong đó 'key' là các 'khoá' giúp chúng ta lấy ra các dữ liệu tương ứng sau này, value là các giá trị tương ứng với một 'khoá'. Các key cũng có thể gọi là các property (thuộc tính) của object.

Kiểu dữ liệu của key luôn là string.
Kiểu dữ liệu của value có thể là bất cứ kiểu dữ liệu nào đã học: string, number, boolean, null, object, array,...

Dùng object và không dùng object

Ví dụ với việc quản lý thông tin sinh viên, giả sử bạn cần quản lý 3 sinh viên với các thông tin: mã sinh viên, tên sinh viên, tuổi.

Không dùng object

Khi không dùng object, ta cần khai báo các biến riêng biệt để lưu trữ dữ liệu cho các thông tin của sinh viên

```
const sv1ID = 1;
const sv1Name = "Alex";
const sv1Age = 20;

const sv2ID = 2;
const sv2Name = "Nagi";
const sv2Age = 18;

const sv3ID = 3;
const sv3Name = "Lua";
const sv3Age = 19;
```

Khi cần in ra thông tin sinh viên, ta cũng dùng các biến riêng lẻ để thực hiện lấy giá trị ra

```
console.log(`- Thông tin SV1: ${sv1ID}, ${sv1Name}, ${sv1Age}`);
console.log(`- Thông tin SV1: ${sv2ID}, ${sv2Name}, ${sv2Age}`);
console.log(`- Thông tin SV1: ${sv3ID}, ${sv3Name}, ${sv3Age}`);
```

Việc quản lý các biến rất dễ gây nhầm lẫn. Ví dụ, bạn có thể gõ nhầm biến `sv3Name` cho sinh viên 2.

Dùng object

Dùng object, ta khai báo đối tượng sinh viên như sau:

```
const sv1 = {
    "ID": 1,
    "name": "Alex",
    "age": 20
};
```

Bây giờ, chúng ta chỉ có 1 hằng số sv1 duy nhất lưu trữ thông tin của sinh viên. Khi truy cập các thuộc tính của sinh viên 1, ta có thể truy cập thông qua việc dùng dấu `.`:

```
console.log(sv1.ID);
```

hoặc dùng ngoặc vuông `["<key>"]`:

```
console.log(sv1["name"]);
```

Object lồng nhau

Một object có thể chứa một object khác. Ví dụ dưới đây mô tả object student chứa một object address:

```
const student = {
    id: 1,
    name: "Alex",
    address: {
        province: "Ha Noi",
        isCapital: true,
        country: "Viet Nam"
    }
}
```

Thêm thuộc tính vào object

Để thêm thuộc tính mới vào object, chúng ta chỉ cần dùng dấu . hoặc ngoặc vuông [] để định nghĩa thuộc tính mới. Ví dụ:

```
let bike = {
  make: 'Yamaha',
  model: 'YZF-R3'
};

bike.color = “Blue”;
bike[“price new”] = 100;

console.log(bike);
```

Kết quả:

```
{make: 'Yamaha', model: 'YZF-R3', color: 'Blue', 'price new' :
100}
```

Xóa thuộc tính của object

Để xóa thuộc tính của object, chúng ta dùng hàm `delete`:

```
let employee = {
  name: 'Le Van C',
  age: 30,
  department: 'HR'
};
delete employee.age;
console.log(employee);

// Kết quả:
{name: 'Le Van C', department: 'HR'}
```

Tips

1. Các thuộc tính có thể khai báo nằm trên cùng một dòng, có thể dùng nháy đơn hoặc nháy kép để khai báo thuộc tính.

   ```
   const student = { 'id': 10 };
   const student2 = { "id": 20 };
   ```

2. Đối với các thuộc tính không có dấu cách, có thể bỏ dấu nháy đơn và nháy kép.
```
const student = { id: 10 };
const student2 = { id: 20 };
```

3. Thông thường, chúng ta hạn chế đặt tên thuộc tính có dấu cách, để việc khai báo và truy cập thuộc tính dễ dàng hơn.
Không nên: đặt tên thuộc tính chứa dấu cách
```
const student = { 'my name': "Alex" };
```

Nên: đặt tên thuộc tính sử dụng camelCase, không chứa dấu cách.
```
const student = { myName: "Alex" };
```

4. Ta không thể thay đổi giá trị của hằng số. Tuy nhiên, nếu hằng số này là một Object, ta có thể thay đổi giá trị các **thuộc tính** của chúng.
Không thể làm thế này:
```
const student = { id: 10, name: "Alex" };
student = { id: 11, name: "Nagi" };
//      ^

// TypeError: Assignment to constant variable.
```

Có thể làm thế này:
```
const student = { id: 10, name: "Alex" };
student.id = 11;
student.name = "Nagi";
```

Mảng

Định nghĩa

Mảng (hay array) là một đối tượng trong Javascript, giúp lưu trữ một tập hợp các giá trị, lưu trữ dưới một tên biến và có thể sử dụng các thao tác trên mảng.

Một mảng có thể chứa **nhiều loại dữ liệu khác nhau**.

Khai báo mảng

Để khai báo mảng, ta sử dụng cú pháp:

```
let/const/var <tên_mảng> = [<danh sách các giá trị, cách nhau bởi dấu phẩy ","]
```

Ví dụ:

```
let numberArr = [1, 20.5, -300, 4];
const strArr = ["Playwright", "Việt", "Nam"];
var mixedArr = ["Playwright", 10, true, null, {id: 1, name: "Alex"}];

console.log(numberArr);
console.log(strArr);
console.log(mixedArr);

// Kết quả:
// [ 1, 20.5, -300, 4 ]
// [ 'Playwright', 'Việt', 'Nam' ]
// [ 'Playwright', 10, true, null, { id: 1, name: 'Alex' } ]
```

Giải thích:

- Mảng `numberArr` chứa các dữ liệu kiểu số.
- Mảng `strArr` chứa các dữ liệu kiểu chuỗi.
- Mảng `mixedArr` chứa các dữ liệu kiểu hỗn hợp: chuỗi, số, boolean, object.

Thao tác trên mảng

Lấy độ dài của mảng

Để lấy độ dài của mảng, ta dùng thuộc tính length của array:

```
const arr = [1, 2];
console.log(arr.length);
// Kết quả
// 2
```

Thường ứng dụng trong việc lặp qua các phần tử trong mảng

```
const arr = [20, 50, 30, 40];
```

```
for (let i = 0;i < arr.length; i++) {
    console.log(arr[i]);
}
// Kết quả
// 20
// 50
// 30
// 40
```

Truy xuất phần tử của mảng

Để truy xuất phần tử trong mảng, ta dùng cú pháp arr[<index>], trong đó index là số thứ tự, tính từ 0.

```
const udemy = [20, 50, 30, 40];


console.log(udemy[0]);
console.log(udemy[1]);
console.log(udemy[2]);
console.log(udemy[3]);
// Kết quả
// 20
// 50
// 30
// 40
```

Giải thích:

- Mảng có 4 phần tử: 20, 50, 30, 40
    - Phần tử 20: index = 0
    - Phần tử 50: index = 1
    - Phần tử 30: index = 2
    - Phần tử 40: index = 3
- Khi ta sử dụng: a[3] tức là đang lấy phần tử có vị trí số 3.

Thêm phần tử vào mảng

Thêm vào đầu mảng

Sử dụng hàm `unshift()`:

```
const arr = [20, 50];

arr.unshift(10);


console.log(arr);

// Kết quả

// [ 10, 20, 50 ]
```

Thêm vào cuối mảng

Sử dụng hàm `push()`:

```
const arr = [20, 50];
arr.push(10);

console.log(arr);
// Kết quả
// [ 20, 50, 10 ]
```

Xóa phần tử khỏi mảng

Xóa phần tử đầu của mảng

Sử dụng hàm `shift()`

```
const arr = [20, 50, 10];
const deleted = arr.shift();

console.log(arr);
console.log(deleted);
// Kết quả
// [ 50, 10 ]
// 20
```

Giải thích:

- Hàm `shift()` xóa phần tử đầu tiên của mảng và trả về.

Xóa phần tử cuối của mảng

Sử dụng hàm `pop()` để xóa phần tử cuối cùng của mảng

```
const arr = [20, 50, 10];
const deleted = arr.pop();

console.log(arr);
console.log(deleted);
// Kết quả
// [ 20, 50 ]
// 10
```

Giải thích:

- Hàm `pop()` lấy giá trị cuối cùng của phần tử ra khỏi mảng và trả về.

Tips

- Mặc dù mảng có thể chứa nhiều kiểu dữ liệu khác nhau, nhưng trong thực tế, chúng ta thường chỉ sử dụng một loại dữ liệu duy nhất cho một mảng
  Không nên: khai báo mix các kiểu dữ liệu trong cùng một mảng

  ```
  var mixedArr = ["Playwright", 10, true, null, {id: 1, name: "Alex"}];
  ```

  Nên: tách kiểu dữ liệu tương ứng thành từng mảng.

  ```
  let numberArr = [1, 20.5, -300, 4];
  const strArr = ["Playwright", "Việt", "Nam"];
  ```

-

Các toán tử

Toán tử gán

Toán tử gán dùng để gán giá trị ở bên phải vào biến/hằng ở bên trái.

Toán tử	Ví dụ	Ý nghĩa
=	name = “Playwright”;	Gán giá trị “Playwright” cho biến name
+=	age += 10;	Là viết tắt của age = age + 10;

		Cộng thêm vào age 10 đơn vị.
-=	age -= 10;	Là viết tắt của age = age - 10; Trừ đi cho biến age 10 đơn vị.
*=	money *= 10;	Là viết tắt của money = money*10; Nhân giá trị của biến money thêm 10 lần.
/=	money /= 10	Là viết tắt của money = money / 10; Chia giá trị của biến money đi 10 lần.
%=	money %= 10	Là viết tắt của money = money % 10; Lấy giá trị phần dư của money sau khi chia cho 10

Toán tử số học

Sử dụng cho các kiểu dữ liệu có kiểu number.

Toán tử	Ví dụ	Ý nghĩa
+	2 + 5	Phép cộng
-	2 - 5	Phép trừ
*	2 * 5	Phép nhân
/	2 / 5	Phép chia
%	2 % 5	Phép chia dư
++	2++	Phép tăng lên 1 đơn vị
--	2--	Phép giảm đi 1 đơn vị

Toán tử so sánh

Toán tử so sánh dùng để so sánh giá trị của 2 vế. Toán tử so sánh sẽ trả về true nếu đúng, trả về false nếu sai.

Toán tử	Ví dụ	Kết quả	Ý nghĩa

>	2 > 5	false	Phép so sánh lớn hơn
<	2 < 5	true	Phép so sánh nhỏ hơn
>=	2 >= 5	false	Phép so sánh lớn hơn hoặc bằng
<=	2 <= 5	true	Phép so sánh nhỏ hơn hoặc bằng
==	5 == '5' 2 == 2	true true	Phép so sánh bằng, chỉ so sánh giá trị, không so sánh về kiểu dữ liệu
===	5 === '5' '5' === '5'	false true	Phép so sánh bằng, so sánh cả về giá trị và về kiểu dữ liệu
!=	5 != '5' 2 != 3	false true	Phép so sánh khác, chỉ so sánh về giá trị, không so sánh về kiểu dữ liệu
!==	5 !== '5' '5' !== '5'	true false	Phép so sánh khác, so sánh cả về giá trị và về kiểu dữ liệu

Toán tử logic

Là các toán tử dùng để kết hợp các giá trị với nhau.

Toán tử	Ví dụ	Kết quả	Ý nghĩa
&&	true && true true && false false && false	true false false	Phép và, trả về true nếu cả 2 mệnh đề đều đúng
||	false || true true || true false || false	true true false	Phép hoặc, trả về true nếu một trong hai mệnh đề đúng
!	!true !false	false true	Phép nghịch đảo, trả về giá trị ngược lại của biểu thức

Toán tử với chuỗi

Toán tử	Ví dụ	Kết quả	Ý nghĩa
+	`"Hello" + "Playwright"`	`HelloPlaywright`	Trả về chuỗi là

			nối của 2 chuỗi

``	`const center = “PW Việt Nam”;`
	`const str = `Hello ${center}`;`
	`console.log(str);`
		`Hello PW Việt Nam`	Gọi là template literal. Giúp đưa giá trị của biến hoặc biểu thức vào vị trí `${}`

Toán tử typeof

Toán tử typeof trả về kiểu dữ liệu của biến cần kiểm tra:

```
const pw1 = 100;
const pw2 = "Playwright Viet Nam";
const pw3 = true;
const pw4 = null;
const pw5;

console.log(typeof(pw1)); // trả về number
console.log(typeof(pw2)); // trả về string
console.log(typeof(pw3)); // trả về boolean
console.log(typeof(pw4)); // trả về object
console.log(typeof(pw5)); // trả về undefined
console.log(typeof(pw6)); // trả về undefined
```

Khối (block)

Code có thể tổ chức thành từng khối (block), bằng cách đặt vào trong cặp ngoặc: ({}).

```
{
    // Block code 1
}

{
    // Block code 2
}
```

Việc chia code thành các block giúp kiểm soát phạm vi truy cập của biến tốt hơn.
Ví dụ:

Câu điều kiện

Khái niệm

Câu điều kiện kiểm tra nếu điều kiện đúng thì sẽ thực thi code trong khối lệnh, giúp thực hiện các logic rẽ nhánh phức tạp.

Có một số loại câu điều kiện thường dùng:

- if
- if…else
- if…else…if
- switch…case

Điều kiện `if`

Cú pháp điều kiện if:

```
if (condition) {
    // code block
}
```

Trong đó, condition = true thì sẽ chạy đoạn `code block`.

Điều kiện if…else…

Cú pháp điều kiện if…else…:

```
if (condition) {
    // code block 1
} else {
    // code block 2
}
```

Trong đó:

- Nếu `condition = true`, sẽ chạy logic ở code block 1.
- Nếu `condition = false`, sẽ chạy logic ở code block 2.

Có thể hiểu đơn giản, điều kiện if…else là điều kiện dạng nếu…thì… Nếu đúng thì chạy logic ở block của nếu. Nếu không đúng thì chạy logic ở block của thì…

Điều kiện if…else if…

Cú pháp điều kiện if…else if…:

```
if (condition) {
    // Code block 1
} else if (condition2) {
    // Code block 2
} else if (condition3) {
    // Code block 3
} else {
    // Code block 4
}
```

Có thể hiểu if…else if là câu điều kiện nâng cao hơn câu điều kiện if, thêm một lần kiểm tra điều kiện, chia thành nhiều trường hợp:

- Nếu `condition` đúng, chạy logic trong code block 1
- Nếu `condition` sai, kiểm tra `condition2`
    - Nếu `condition2` đúng, chạy logic trong code block 2
    - Nếu `condition2` sai, kiểm tra `condition3`
        - Nếu `condition3` đúng, chạy logic trong code block 3
        - Nếu `condition3` sai, chạy logic trong code block 4

Có thể hiểu, câu lệnh else là “các trường hợp còn lại”.

Điều kiện switch…case

`switch…case` thường dùng để rẽ nhánh trong trường hợp có nhiều điều kiện khác nhau. `switch…case` giúp code trở nên gọn gàng, dễ nhìn hơn.

Cú pháp switch…case:

```
switch (condition) {
    case "<case_value_1>":
        // code block 1
        break;
    case "<case_value_2>":
        // code block 2
        break;
    case "<case_value_3>":
```

```
        // code block 3
        break;
    default:
        // code block 4
}
```

Trong đó:

- `condition` gọi là biểu thức điều kiện
- Các `case_value1, case_value2, case_value3…` được gọi là các giá trị của các trường hợp.
- Nếu sau khi tính toán, giá trị của `condition` rơi vào `case_value` nào thì sẽ chạy đoạn code tương ứng với case đó.
- Lưu ý, mỗi case cần có `break` để thoát khỏi đoạn logic của case. Nếu không có break, đoạn logic của case phía dưới sẽ tiếp tục được chạy.
- `default` là case mặc định. Trong trường hợp nếu không có case nào match, logic default sẽ được chạy.

Xét ví dụ: kiểm tra biến `day`, nếu giá trị 2 thì in ra "thứ 2", giá trị 3 thì in ra thứ 3, …, giá trị 7 in ra "thứ 7".

Nếu với lệnh if, ta sẽ làm như sau:

```
const day = 3;

if (day === 2) {
    console.log("Thứ 2");
} else if (day === 3) {
    console.log("Thứ 3");
} else if (day === 4) {
    console.log("Thứ 4");
} else if (day === 5) {
    console.log("Thứ 5");
} else if (day === 6) {
    console.log("Thứ 6");
} else if (day === 7) {
    console.log("Thứ 7");
} else {
    console.log("Chủ nhật");
```

```
}
```

Có thể thấy, với if..else if… trông khá rối.
Viết lại đoạn code trên sử dụng switch…case:

```
const day = 3;

switch (day) {
    case 2:
        console.log("Thứ 2");
        break;
    case 3:
        console.log("Thứ 3");
        break;
    case 4:
        console.log("Thứ 4");
        break;
    case 5:
        console.log("Thứ 5");
        break;
    case 6:
        console.log("Thứ 6");
        break;
    case 7:
        console.log("Thứ 7");
        break;
    default:
        console.log("Chủ nhật");
}
```

Logic lúc này trông gọn gàng và sáng hơn nhiều.
Hãy sử dụng `switch…case…` trong trường hợp có nhiều trường hợp bạn nhé. ^^

Vòng lặp

Khái niệm

Vòng lặp là cách chạy 1 đoạn logic 1 số lần nhất định.

Ví dụ với việc in ra 100 lần câu “Hello Playwright Việt Nam”, ta có 2 lựa chọn:
Cách 1: gõ 100 dòng `console.log(“Hello Playwright Việt Nam”);`

```
console.log(“Hello Playwright Việt Nam”);
console.log(“Hello Playwright Việt Nam”);
console.log(“Hello Playwright Việt Nam”);
console.log(“Hello Playwright Việt Nam”);
...
console.log(“Hello Playwright Việt Nam”);
console.log(“Hello Playwright Việt Nam”);
```

Cách 2: Sử dụng vòng lặp

```
for (let i = 0;i < 100; i++) {
    console.log("Hello Playwright Việt nam");
}
```

Một số vòng lặp thường gặp:

- for
- forEach
- for…in
- for…of
- while
- do…while
- break & continue.

Vòng lặp for

Cú pháp

```
for (<khởi tạo>; <điều kiện lặp>; <điều kiện thay đổi>) {
    // Code ở đây
}
```

Trong đó:

- `<khởi tạo>`: là điều kiện khởi tạo ban đầu, trước khi vòng lặp ban đầu
- `<điều kiện lặp>`: là điều kiện kiểm tra trước mỗi lần vòng lặp chạy; nếu điều kiện này đúng, vòng lặp sẽ được thực hiện, nếu điều kiện này sai, vòng lặp sẽ dừng lại.

- `<thay đổi>`: là đoạn code chạy sau mỗi lần vòng lặp chạy xong, nhằm thay đổi giá trị khởi tạo, giúp vòng lặp dừng lại được.

Ví dụ

Ví dụ 1: in ra số từ 1 đến 100

```
for (let i = 0;i < 100; i++) {
    console.log(i);
}
```

Trong đó:

- Mệnh đề khởi tạo:`let i = 0`: khởi tạo biến i có giá trị là 0.
- Mệnh đề điều kiện lặp: kiểm tra biến `i < 100` thì mới chạy đoạn code nằm trong vòng for
- Mệnh đề thay đổi: `i++`: sau mỗi vòng lặp, mệnh đề này sẽ được chạy để tăng giá trị của biến i lên 1 đơn vị.
- Đoạn code trong vòng lặp: console.log(i): in ra giá trị của biến i.

Ví dụ 2: in ra các số lẻ từ 1 đến 100

```
for (let i = 0;i < 100; i++) {
    if (i % 2 === 1) {
        console.log(i);
    }
}
```

Tips sử dụng

- Vòng lặp for thường dùng khi bạn cần lấy thứ tự chạy của vòng lặp (biến i).
- Mệnh đề thay đổi có thể rất đa dạng theo nhu cầu. Ví dụ:
    - Tăng lên 2 đơn vị: i+=2;
    - Giảm đi 1 đơn vị : i--;
    - Nhân đôi giá trị: i*=2;
    - ...

Vòng lặp forEach

Cú pháp

```
<biến_là_tên_mảng>.forEach(callbackFn)
```

Trong đó:

- `<biến_là_tên_mảng>`: tên của biến
- `callbackFn`: viết tắt của callback function, là hàm xử lý vòng for. Thường hàm này sẽ là: `item => { // code ở đây }.`

Ví dụ

```
let numberArr = [1, 20.5, -300, 4];
numberArr.forEach(number => {
    console.log(number)
})
```

Giải thích:

- `numberArr`: tên biến
- `number => { console.log(number)}`: hàm callback chứa thông tin mảng.
    - Chú ý tới tên biến `number`, biến này đại diện cho các phần tử trong mảng. Ở mỗi vòng lặp, biến `number` sẽ được gán giá trị của từng phần tử trong mảng.

Tips sử dụng

- Vòng lặp `forEach` dùng để lặp các phần tử trong mảng, khi bạn không cần lấy thứ tự của các phần tử.
- Trong callback function, nếu chỉ có 1 dòng duy nhất, có thể bỏ cặp ngoặc nhọn bao ngoài:
  Đầy đủ:

```
numberArr.forEach(number => {
    console.log(number)
})
```

  Rút gọn:

```
numberArr.forEach(number => console.log(number));
```

-

Vòng lặp for…in

Cú pháp

```
for (const property in object) {
    // Code ở đây
}
```

Trong đó:

- `const property`: khai báo một biến tên là property. Trước mỗi vòng lặp, biến property này sẽ có giá trị là các thuộc tính của object
- `object`: là tên biến có kiểu dữ liệu object

Ví dụ

```
const student = {
    id: 1,
    name: "Alex",
    isGraduated: true
};

for (const property in student) {
    console.log(property);
}
```

```
// Kết quả:
// id
// name
// isGraduated
```

Giải thích:

- object student có 3 thuộc tính: `id, name, isGraduated`.
- Vòng lặp `for…in` chạy, mỗi vòng lặp sẽ gán lần lượt giá trị của các thuộc tính `id, name, isGraduated` vào hằng số property

Tips sử dụng

- Vòng lặp for..in thường được sử dụng khi bạn muốn lặp trong các thuộc tính của object.
- Có thể kết hợp với cú pháp truy xuất thuộc tính của object để lấy ra giá trị tương ứng

```
for (const property in student) {
    console.log(`Property: ${property}, value: ${student[property]}`);
}
```

- Thứ tự lấy ra các thuộc tính phụ thuộc vào loại trình duyệt mà bạn đang chạy. Vì vậy, bạn chỉ nên dùng for…in trong trường hợp không quan tâm tới thứ tự chạy.

Vòng lặp for…of

Cú pháp

```
for (const item of <tên_biến_của_mảng>) {
    // Code ở đây
}
```

Trong đó:

- `item` là tên của hằng số, giá trị hằng số sẽ được khởi tạo trong mỗi lần lặp
- `<tên_biến_của_mảng>` là tên biến chứa mảng.

Ví dụ

```
const arr = [1, 3, 5, 2];
for (const item of arr) {
    console.log (item);
}
```

```
// Kết quả:
// 1
// 3
// 5
// 2
```

Giải thích:

- arr là mảng chứa các phần tử 1, 3, 5, 2
- Trong vòng lặp for…of, ta khởi tạo hằng số item.
    - Mỗi lần vòng lặp chạy, item sẽ lấy giá trị lần lượt của từng phần tử: 1, 3, 5, 2.
    - Trong vòng lặp, ta sử dụng giá trị của các biến này.

Tips sử dụng

- Lặp qua các giá trị của mảng: for...of là cách đơn giản và rõ ràng để lặp qua tất cả các giá trị trong một mảng, đặc biệt khi bạn không cần quan tâm đến chỉ số của từng phần tử.
- Lặp qua các chuỗi: Chuỗi trong JavaScript cũng có thể được coi là một dãy các ký tự, vì vậy bạn có thể sử dụng for...of để lặp qua từng ký tự trong chuỗi.
  Ví dụ:

```
const str = "Hello";
for (const char of str) {
    console.log(char);
}

// Kết quả:
// H
// e
// l
// l
// o
```

Vòng lặp while

Cú pháp

```
while (<điều kiện>) {
  // Code được thực thi khi điều kiện đúng
}
```

Ví dụ

```
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}

// Kết quả:
// 0
// 1
// 2
// 3
// 4
```

Giải thích:

- i = 0: Khởi tạo biến i bằng 0.
- while (i < 5): Vòng lặp tiếp tục chạy khi i nhỏ hơn 5.
- console.log(i): In giá trị của i ra console.
- i++: Tăng giá trị của i lên 1 sau mỗi lần lặp.

Tips sử dụng

- Vòng lặp while phù hợp khi bạn muốn lặp lại một logic code khi một điều kiện nào đó còn đúng.
- Hãy cẩn thận để đảm bảo rằng điều kiện cuối cùng sẽ trở thành false, nếu không vòng lặp sẽ chạy vô hạn.

Vòng lặp do…while

Cú pháp

```
do {
  // Code được thực thi ít nhất một lần
} while (<điều kiện>);
```

Ví dụ

```
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);

// Kết quả:
// 0
// 1
// 2
// 3
// 4
```

Giải thích:

- Khác với while, vòng lặp do...while sẽ thực thi logic code bên trong ít nhất một lần, sau đó mới kiểm tra điều kiện.
- Vòng lặp tiếp tục chạy khi i nhỏ hơn 5.

Tips sử dụng

- Vòng lặp do...while phù hợp khi bạn muốn đảm bảo rằng một logic code được thực thi ít nhất một lần, bất kể điều kiện ban đầu là gì.

Break

Cú pháp

Break dùng để thoát khỏi vòng lặp hoặc mệnh đề switch. Thường kết hợp với một điều kiện để dừng vòng lặp sớm hơn khi đã thoả mãn điều kiện, hoặc

```
for(let i = 0; i < 100; i++) {
    // some code ...
    break;
    //....
}
```

Giải thích:
trong ví dụ trên, break sẽ thoát khỏi vòng lặp ngay khi gặp break;

Ví dụ

Xét bài toán sau: lặp các số từ 1 đến 100. Mỗi lần lặp, cộng dồn giá trị vào biến sum sao cho sum <= 20

Không có break

```
let sum = 0;
for(let i = 0; i < 100; i++) {
    if (sum <= 20) {
        sum += i;
    }
}
console.log("Sum without break:", sum);
```

Giải thích:

- Khi không dùng break, ta bắt buộc phải chạy vòng lặp hết 100 lần.
- Ở mỗi vòng lặp, kiểm tra nếu tổng hiện tại nhỏ hơn hoặc bằng 20 thì cộng i vào sum. Tuy nhiên, vòng lặp vẫn sẽ chạy hết 100 lần, dù sum đã vượt quá 20 từ lâu.

Có break

```
let sum = 0;
 for (let i = 0; i < 100; i++) {
     if (sum + i > 20) {
         break; // Thoát khỏi vòng lặp nếu cộng i vào sum sẽ vượt
quá 20
     }
     sum += i;
 }
 console.log("Sum with break:", sum);
```

Giải thích:

- Với break, vòng lặp sẽ dừng lại ngay khi tổng sum + i lớn hơn 20.
- Điều này giúp tiết kiệm tài nguyên và thời gian thực thi, đặc biệt khi vòng lặp có số lượng lớn các lần lặp.
- Trong trường hợp này, vòng lặp sẽ không chạy hết 100 lần.

Tips

- `break` thường được dùng khi bạn muốn tìm kiếm một giá trị cụ thể trong một mảng hoặc một tập hợp dữ liệu, và bạn muốn dừng lại ngay khi tìm thấy giá trị đó.
- Sử dụng `break` giúp tránh việc thực hiện các phép tính không cần thiết sau khi đã đạt được mục tiêu.
- Khi sử dụng `break` trong các vòng lặp lồng nhau, nó chỉ thoát khỏi vòng lặp gần nhất chứa nó.

Continue

Cú pháp

```
continue;
```

Ví dụ

Có continue

```
for (let i = 0; i < 10; i++) {
    if (i % 2 === 0) { // Nếu i là số chẵn
        continue; // Bỏ qua các lệnh còn lại trong vòng lặp hiện
tại và chuyển sang lần lặp tiếp theo
    }
    console.log(i); // Chỉ in ra các số lẻ
}

// Kết quả:
// 1
// 3
// 5
// 7
// 9
```

Không có continue

```
for (let i = 0; i < 10; i++) {
    if (i % 2 === 0) {
        // Không có continue ở đây
    } else {
        console.log(i);
    }
}
// Kết quả:
// 1
// 3
// 5
// 7
// 9
```

Giải thích:

- `continue` dùng để bỏ qua phần còn lại của vòng lặp hiện tại và chuyển sang lần lặp tiếp theo.
- Trong ví dụ trên, khi i là số chẵn, continue được gọi, khiến cho console.log(i) bị bỏ qua, do đó chỉ các số lẻ được in ra.

Tips

- `continue` giúp bạn bỏ qua một số trường hợp cụ thể trong vòng lặp mà không cần thoát khỏi vòng lặp hoàn toàn.

Hàm

Khái niệm hàm

Hàm là một khối code. Hàm thường được dùng để viết các đoạn logic lặp đi lặp lại.

Khai báo hàm

Cú pháp

```
function <tên_hàm(<danh_sách_tham_số>) {
    // Code của hàm
}
```

Trong đó:

- `function`: từ khoá khai báo hàm
- `<tên_hàm>`: tên của hàm.
- `<danh sách tham số>`: các tham số của hàm, cách nhau bởi dấu phẩy “,”
- `Code của hàm`: đoạn logic thực thi trong hàm.

Ví dụ với hàm đơn giản

```
function hello() {
    console.log(“Hello Playwright Viet Nam”);
}
```

Gọi hàm

Hàm không thể tự chạy được. Muốn chạy, bạn cần “gọi hàm”.

Hàm không có tham số

Đối với hàm không có tham số, ta chỉ cần “gọi hàm” bằng cách viết tên hàm và thêm cặp ngoặc tròn phía sau:

```
hello();
```

Hàm có tham số

Tham số giúp hàm trở nên linh động hơn. Tham số nằm trong cặp ngoặc tròn `()`.

Ví dụ

```
function describePerson(name, age, city) {
    console.log(name + " is " + age + " years old and lives in " +
city + ".");
}

describePerson("Bob", 30, "New York");
// Kết quả: Bob is 30 years old and lives in New York.
```

Hàm có giá trị trả về

Hàm có giá trị trả về là hàm trả về một giá trị sau khi hoàn thành việc thực thi. Giá trị trả về được chỉ định bằng từ khóa `return`.

Ví dụ

```
function add(a, b) { // a và b là tham số
    return a + b; // Trả về tổng của a và b
}

let sum = add(5, 3);
```

Một số kiểu khai báo hàm khác

Hàm anonymous

Là hàm không có tên. Thường được sử dụng làm callback hoặc gán cho một biến.

Cú pháp

```
function (parameters) {
  // Code here
}
```

Ví dụ

```
let myFunc = function(x) {
    return x * 2;
};

console.log(myFunc(4)); // Kết quả: 8
```

Hàm lambda (arrow function)

Lambda Function (hay Arrow Function) là một cách ngắn gọn để viết các hàm trong JavaScript. Arrow function sử dụng dấu `=>` thay vì từ khóa `function`.

Cú pháp

```
(parameters) => {
  // Code here
}
```

Ví dụ

```
const square = (x) => x * x
console.log(square(5)); // Kết quả: 25
```

Một số hàm utils thường dùng

Khái niệm hàm util

Hàm util là các hàm nhỏ, có mục đích cụ thể, thường được sử dụng lại nhiều lần trong một dự án. Chúng giúp làm cho code sạch hơn, dễ đọc và dễ bảo trì hơn.

String util functions

1. `trim()`

- **Khái niệm**: Phương thức `trim()` dùng để loại bỏ khoảng trắng ở đầu và cuối chuỗi. Khoảng trắng bao gồm cả dấu cách, dấu tab, và các ký tự không in khác.

Ví dụ:

```
let str = "   JavaScript is awesome! ";
str = str.trim();
console.log(str); // Kết quả: "JavaScript is awesome!"
```

2. `toLowerCase()` và `toUpperCase()`

- **Khái niệm**:
    - `toLowerCase()` chuyển đổi tất cả các ký tự trong chuỗi thành chữ thường.
    - `toUpperCase()` chuyển đổi tất cả các ký tự trong chuỗi thành chữ hoa.

Ví dụ:

```
let str = "JavaScript is awesome!";
console.log(str.toLowerCase());
// Kết quả: "javascript is awesome!"
console.log(str.toUpperCase());
```

3. `includes()`

- **Khái niệm**:
    - Phương thức `includes()` kiểm tra xem một chuỗi có chứa một chuỗi con (substring) hay không.
    - Nó trả về `true` nếu tìm thấy và `false` nếu không.

Ví dụ:

```
let str = "JavaScript is awesome!";
console.log(str.includes("awesome")); // Kết quả: true
console.log(str.includes("Awesome")); // Kết quả: false
```

4. `replace()`

- **Khái niệm**:
    - Phương thức `replace()` dùng để thay thế một chuỗi con trong chuỗi bằng một chuỗi khác.
    - Bạn có thể thay thế chỉ chuỗi đầu tiên hoặc tất cả chuỗi con bằng cách sử dụng biểu thức chính quy.

Ví dụ:

```
let str = "JavaScript is awesome!";
str = str.replace("awesome", "fun");
console.log(str); // Kết quả: "javascript is fun!"
```

5. `split()`

- **Khái niệm**: Phương thức `split()` chia một chuỗi thành một mảng các chuỗi con, dựa trên một ký tự (delimiter).

Ví dụ:

```
let str = "JavaScript is awesome!";
let words = str.split(" ");
console.log(words); // Kết quả: ["JavaScript", "is", "awesome!"]
```

6. `substring()`

- **Khái niệm**: Phương thức `substring()` trả về một phần của chuỗi, bắt đầu từ chỉ số (index) được chỉ định đến một chỉ số khác hoặc đến cuối chuỗi.

Ví dụ:

```
let str = "Hello World!";
console.log(str.substring(0, 5)); // Kết quả: "Hello"
console.log(str.substring(6));    // Kết quả: "World!"
```

7. `indexOf()`

- **Khái niệm**: Phương thức `indexOf()` trả về vị trí xuất hiện đầu tiên của một chuỗi con trong chuỗi, hoặc `-1` nếu không tìm thấy.

Ví dụ:

```
let str = "Hello World!";

console.log(str.indexOf("World")); // Kết quả: 6

console.log(str.indexOf("JavaScript")); // Kết quả: -1
```

Array util functions

1. `map()`

- **Khái niệm**: Phương thức `map()` tạo ra **một mảng mới** bằng cách áp dụng một hàm lên từng phần tử của mảng gốc.

Ví dụ:

```
let numbers = [1, 2, 3, 4];
let squared = numbers.map(num => num * 2);
console.log(squared); // Kết quả: [2, 4, 6, 8]
```

2. `filter()`

- **Khái niệm**: Phương thức `filter()` tạo ra **một mảng mới** chỉ bao gồm các phần tử thỏa mãn điều kiện được chỉ định trong hàm callback.

Ví dụ:

```
let numbers = [1, 2, 3, 4];
let evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // Kết quả: [2, 4]
```

3. `find()`

- **Khái niệm**: Phương thức `find()` **trả về giá trị của phần tử đầu tiên trong mảng** thỏa mãn điều kiện được chỉ định trong hàm callback, nếu không có phần tử nào thỏa mãn thì trả về `undefined`.

Ví dụ:

```
let numbers = [1, 2, 3, 4];
let firstEven = numbers.find(num => num % 2 === 0);
console.log(firstEven); // Kết quả: 2
```

4. `reduce()`

- **Khái niệm**: Phương thức `reduce()` áp dụng một hàm lên từng phần tử của mảng (từ trái qua phải) để trả về một giá trị duy nhất.

Ví dụ:

```
let numbers = [1, 2, 3, 4];
let sum = numbers.reduce((total, num) => total + num, 0);
console.log(sum); // Kết quả: 10

//total: biến nhận giá trị duy nhất
```

```
//num: phần tử của mảng
//0 giá trị khởi tạo cho biến total
```

5. some()

- **Khái niệm**: Phương thức `some()` kiểm tra xem có ít nhất một phần tử trong mảng thỏa mãn điều kiện được chỉ định trong hàm callback. Trả về `true` nếu tìm thấy, ngược lại trả về `false`.

Ví dụ:

```
let numbers = [1, 2, 3, 4];
let hasEven = numbers.some(num => num % 2 === 0);
console.log(hasEven); // Kết quả: true
```

6. every()

- **Khái niệm**: Phương thức `every()` kiểm tra xem **tất cả** các phần tử trong mảng có thỏa mãn điều kiện được chỉ định trong hàm callback hay không. Trả về `true` nếu **tất cả** đều thỏa mãn, ngược lại trả về `false`.

Ví dụ:

```
let numbers = [2, 4, 6, 8];
let allEven = numbers.every(num => num % 2 === 0);
console.log(allEven); // Kết quả: true
```

7. push()

- **Khái niệm**: Phương thức `push()` **thêm một hoặc nhiều phần tử vào cuối mảng** và trả về độ dài mới của mảng.

Ví dụ:

```
let numbers = [1, 2, 3];
numbers.push(4, 5);
console.log(numbers); // Kết quả: [1, 2, 3, 4, 5]
```

8. shift()

- **Khái niệm**: Phương thức `shift()` **loại bỏ phần tử đầu tiên của mảng** và **trả về phần tử bị loại bỏ**. Phương thức này thay đổi độ dài của mảng**.**

Ví dụ:

```
let numbers = [1, 2, 3];
let firstElement = numbers.shift();
console.log(firstElement); // Kết quả: 1
console.log(numbers); // Kết quả: [2, 3]
```

9. sort()

- **Khái niệm**: Phương thức `sort()` **sắp xếp các phần tử của mảng** theo thứ tự tăng dần hoặc theo hàm so sánh được cung cấp. Nó thay đổi mảng ban đầu.
- `sort()` mặc định: **sắp xếp các phần tử của mảng như chuỗi theo thứ tự từ điển học Unicode (ASCII hoặc UTF-16)**. Điều này có nghĩa là mỗi phần tử sẽ được **chuyển đổi thành chuỗi** (nếu không phải là chuỗi) trước khi so sánh.

Ví dụ:

```
let numbers = [4, 2, 3, 1];

numbers.sort();

console.log(numbers); // Kết quả: [1, 2, 3, 4]
```

Part 2: Typescript

Class

- **Class** là một khái niệm quan trọng trong lập trình hướng đối tượng (OOP), được dùng để tạo ra các đối tượng (object).
- Class là một khuôn mẫu định nghĩa các **thuộc tính** và **phương thức** mà các đối tượng thuộc class đó sẽ có.

Khai báo

```
class Animal {
    name: string; // Khai báo thuộc tính

    constructor(name: string) {
        this.name = name;
    }

    makeSound() {
        console.log("Generic animal sound");
    }
}
```

Hàm tạo (constructor)

`constructor` là một phương thức đặc biệt được gọi khi một đối tượng mới của class được tạo ra. Nó được sử dụng để khởi tạo các thuộc tính của đối tượng.

Kế thừa

Tính kế thừa: giúp tạo ra các class con từ class cha, giảm thiểu được việc lặp lại code và tăng tính linh hoạt trong code.

```
class Dog extends Animal {
    breed: string;

    constructor(name: string, breed: string) {
        super(name); // Gọi constructor của lớp cha
        this.breed = breed;
    }

    makeSound() {
        console.log("Woof!"); // Ghi đè phương thức của lớp cha
    }

    wagTail() {
        console.log("Wagging tail");
    }
}

const myDog = new Dog("Buddy", "Golden Retriever");
myDog.makeSound(); // Woof!
myDog.wagTail(); // Wagging tail
```

Các concept nâng cao

Var và let khác nhau thế nào?

- `var` có phạm vi hàm (function scope) hoặc phạm vi toàn cục (global scope). Có thể khai báo lại và cập nhật.
- `let` có phạm vi khối (block scope). Không thể khai báo lại trong cùng một phạm vi, nhưng có thể cập nhật.

Ví dụ

- Đối với `var`:

```
for (var i = 0; i < 5; i++) {
   console.log(i);
}

console.log(i); // i = 5
```

- Đối với `let`:

```
for (let i = 0; i < 5; i++) {
   console.log(i);
}

console.log(i); //  Lỗi: i is not defined
```

Null và undefined khác nhau thế nào?

- `null` là một **giá trị gán**, biểu thị rằng **một biến không có giá trị**. Bạn phải gán `null` cho một biến một cách rõ ràng.
- `undefined` biểu thị một biến **đã được khai báo** nhưng **chưa được gán giá trị**. Nó cũng là giá trị trả về mặc định của một hàm không có return.

Ví dụ

```
let myVar; // myVar là undefined
let myNullVar = null; // myNullVar là null

console.log(myVar); // undefined
console.log(myNullVar); // null

console.log(typeof myVar); // undefined
console.log(typeof myNullVar); // object
```