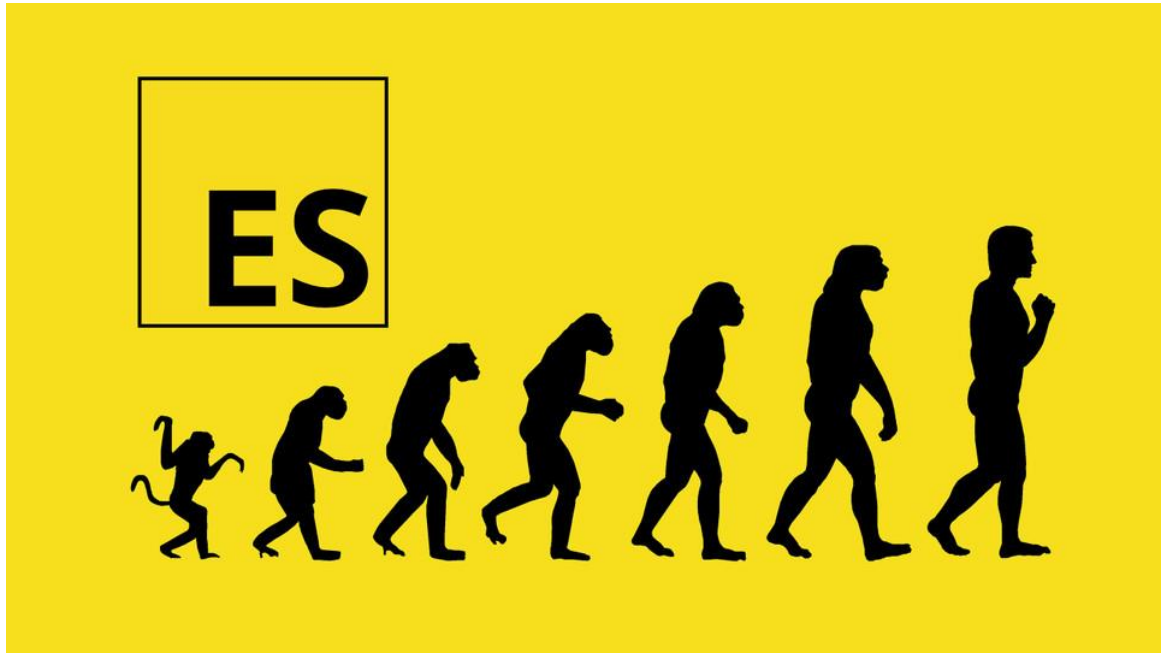
Theory 5.1
ECMAScript Features

WEEK 02

Nội dung chính

Tổng quan về ECMAScript

- ❑ **ECMAScript** (ES) là **tiêu chuẩn kỹ thuật của ngôn ngữ lập trình JavaScript.**
- ❑ Được sử dụng để **thống nhất cách trình duyệt** và **công cụ JavaScript hoạt động**.
- ❑ Giúp code tối ưu, dễ đọc, dễ bảo trì hơn và tương thích trình duyệt tốt hơn.



Tổng quan về ECMAScript

❑ Lịch sử phát triển:

Năm	Phiên bản	Trạng thái	Đặc điểm nổi bật
1997	ES1	Đã phát hành	Phiên bản đầu tiên
1998	ES2	Đã phát hành	Chỉ cập nhật nhỏ để đồng bộ ISO, không thêm tính năng
1999	ES3	Đã phát hành	**Quan trọng**: thêm try/catch, regex, do-while, switch,…
Hủy	ES4	Không phát hành	Nhiều đề xuất lớn, phức tạp nên bị từ chối
2009	ES5	Đã phát hành	strict mode, JSON, forEach, map, filter
2015	ES6 (ES2015)	Đã phát hành	**Đột phá lớn**: let, const, arrow function, class, module
2016 – nay	ES7 -> ES15	Đã phát hành	Cập nhật hàng năm với các tính năng nhỏ nhưng thực tế

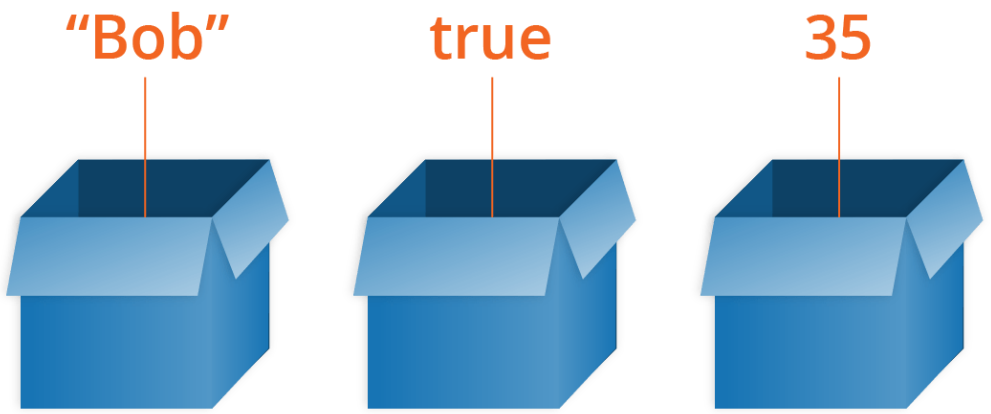
Tính năng Let, Const

❑ **Từ khóa let**:

❍ Từ khóa **let** được sử dụng để khai báo biến thay thế từ khóa **var**

❍ Biến **let** có phạm vi hoạt động trong khối lệnh (**block scope**) **{}**, biến **var** có phạm vi hoạt động trong hàm (**function scope**) và biến **var** có cơ chế **hosting**.

❍ Biến **let** không thể truy cập trước khi khai báo, biến **let** còn có thể gán lại giá trị sau khi khai báo.



Tính năng Let, Const

❑ **Ví dụ so sánh var và let**:

```
// check var scope
function checkVarScope() {
  var a = 1;

  if (a > 0) {
    var a = 2;
    console.log(a);
  }

  console.log(a);
}

checkVarScope(); // ?
```

```
// check let scope
function checkLetScope() {
  let b = 1;

  if (b > 0) {
    let b = 2;
    console.log(b);
  }

  console.log(b);
}

checkLetScope(); // ?
```

Tính năng Let, Const

- **Từ khóa const**:
  - Từ khóa **const** được sử dụng để khai báo hằng số, thông tin cấu hình,…
  - Hằng **const** có phạm vi hoạt động trong khối lệnh (**block scope**) **{}**, giống **let**.
  - Biến **const** không thể thay đổi giá trị sau khi khai báo.
  - Ví dụ: **const PI = 3.14;**

```
function checkConst(radius) {
  const PI = 3.14;
  //   PI = 3.1415; // error
  let perimeter = PI * radius * 2;
  console.log(perimeter);
}

checkConst(2);
```

```
const student = { name: 'Ty' };
console.log(student);

student.name = 'Teo';
console.log(student);

student = {}; // error
console.log(student);
```

Tính năng Arrow Functions

- Là **cú pháp viết ngắn gọn hơn cho function** trong JavaScript.
- Được giới thiệu trong phiên bản **ECMAScript 2015** (ES6)
- Cú pháp:

```
const add = (a, b) => a + b;
```

- Tương đương với:

```
function add (a, b) {
    return a + b;
}
```

- Đặc điểm:
  - Dùng khi cần function ngắn gọn, không cần **this** riêng.
  - Rất hữu ích cho **callback, map(), filter(),...**

Tính năng Arrow Functions

- **Cách sử dụng**:
  - Không tham số:
    () => console.log("Hello");
  - Một tham số: **x => x * 2;**
  - Nhiều dòng:

```
(x, y) => {
    const sum = x + y;
    return sum;
}
```

arrow-function.js > ...

```
// ES5 (JS)
function add(a, b) {
  return a + b;
}

// 1. ES6 (JS): Without any parameters
let greeting = () => console.log("Hello");
greeting();

// 2. ES6 (JS): With one parameter
let doubleX = (x) => x * 2;
console.log(doubleX(2));

// 3. ES6 (JS): With multiple code lines
let sumXY = (x, y) => {
  const sum = x + y;
  return sum;
};
console.log(sumXY(3, 2));
```

Tính năng Template Literals

- **Template Literals** (chuỗi mẫu) thường **sử dụng back-ticks** (``) **thay vì quotes** (“”) **để định nghĩa** một **chuỗi**.

```
> let text2 = `Hello world!`;
  console.log(text2);

  Hello world!
```

- Với Template Literals, chúng ta còn **có thể dùng cả nháy đơn, nháy đôi** bên **trong string**.

```
> let text3 = `He's often called "Nguyễn Văn Tèo"`;
  console.log(text3);

  He's often called "Nguyễn Văn Tèo"
```

Tính năng Template Literals

- **Template Literals** cho phép **sử dụng biến** trong **chuỗi**.

```
let firstName2 = "Tèo";
let lastName2 = "Nguyễn";

let fullName2 = `Welcome ${firstName2}, ${lastName2}!`;

console.log(fullName2);
```

```
Welcome Tèo, Nguyễn!
```

- **Template Literals** cho phép **sử dụng biểu thức** trong **chuỗi**.

```
let price = 10;
let VAT = 0.25;
let total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;

console.log(total);
```

```
Total: 12.50
```

Tính năng Template Literals

❑ **Template Literals** cho phép **sử dụng HTML** trong **chuỗi**.

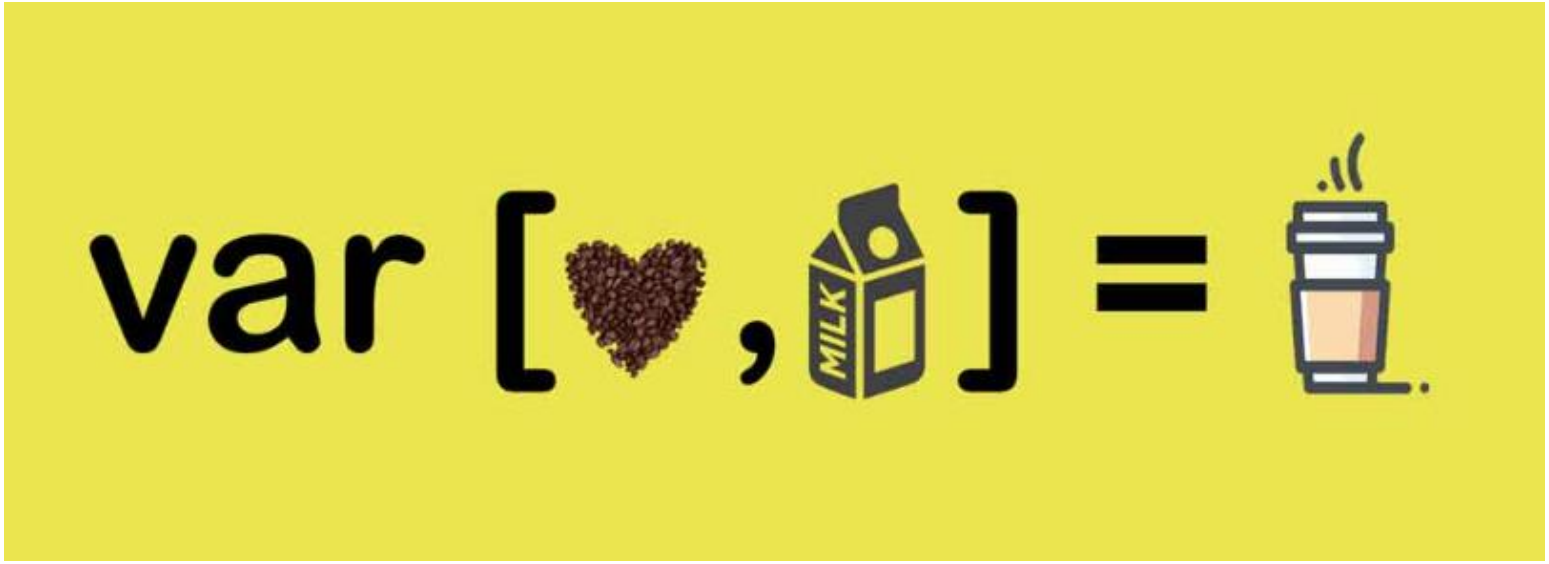
```
> let header = "Templates Literals";
  let tags = ["template literals", "javascript", "es6"];

  let html = `<h2>${header}</h2><ul>`;

  for (const x of tags) {
    html += `<li>${x}</li>`;
  }

  html += `</ul>`;
  document.getElementById("demo").innerHTML = html;
<· '<h2>Templates Literals</h2><ul><li>template literals</li><li>javascript</li><li>es6</li></ul>'
```

❑ **Template Literals** không được hỗ trợ trong **Internet Explorer**.

Tính năng Destructuring

❑ **ES6** cho phép chúng ta **giải nén giá trị từ object** hoặc **array** và **gán giá trị đấy vào một biến** một cách ngắn gọn.

❑ **Destructuring Assignment** có thể sử dụng với: mảng, đối tượng, tham số hàm, giá trị mặc định.



Tính năng Destructuring

❑ Cách sử dụng:

❍ Destructuring với Array

```
const arr = [1, 2, 3];
const [a, b, c] = arr;
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3


// Maybe ignore element
const [x, , z] = [10, 20, 30];
console.log(x); // x=10
console.log(z); // z=30
```

❍ Destructuring với Object:

```
const person = { name: "Dao", age: 30 };
const { name, age } = person;
console.log(name); // Dao

// Maybe rename property
const { name: fullName } = person;
console.log(fullName); // Dao
```

Tính năng Destructuring

❑ Cách sử dụng:

❍ Destructuring với tham số hàm

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

greet({ name: "Dao", age: 30 });
```

❍ Destructuring với giá trị mặc định:

```
const [a = 1, b = 2] = [];
console.log(a, b); // 1 2

const { city = "HCM" } = {};
console.log(city); // HCM
```

Tính năng Default Parameter

❑ Là tính năng cho phép **đặt giá trị mặc định cho tham số** nếu không được truyền khi gọi hàm.
❑ Giúp code dễ bảo trì, tránh **undefined** khi thiếu tham số.
❑ **Ví dụ minh họa:**

```
function sum(a = 1, b = 2) {
  return a + b;
}

console.log(sum()); // 3
console.log(sum(5)); // 7
console.log(sum(5, 10)); // 15
```

Spread Operator, Rest Operator

❑ **Tính năng Spread Operator:**

❍ Là cú pháp dùng để **dàn trải** (giải nén) **các phần tử của** **array, object** hoặc **function paramters** vào vị trí cần phần tử riêng lẻ.

❍ Thường dùng trong:

➢ **Mảng** (Array): nhân bản, hợp nhất, thêm phần tử

➢ **Đối tượng** (Object): nhân bản, ghi đè, hợp nhất phần tử

➢ **Hàm** (Function): truyền nhiều tham số từ mảng

❍ **Spread Operator** giúp viết code ngắn gọn, rõ ràng và bất biến (immutable), rất hữu ích trong **ReactJS**.

Spread Operator, Rest Operator

- **Cú pháp:** (...)
  - Với **Array**
  - Với **Object**
  - Với **Function**

```
// with Array
const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4]; // [1, 2, 3, 4]
console.log(arr2);

// with Object
const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 }; // { a: 1, b: 2, c: 3 }
console.log(obj2);

// with Function
function sum(a, b, c) {
  return a + b + c;
}
const nums = [1, 2, 3];
const total = sum(...nums);
console.log(total); // 6
```

Spread Operator, Rest Operator

❑ **Tính năng Rest Operator:**

❍ Là cú pháp dùng để **gom lại** (nén) **nhiều phần tử** thành một **array** hoặc **object**

❍ Là cú pháp ngược lại với **Spread Operator** và cũng dùng cú pháp (...)

❍ **Rest Operator** giúp viết hàm kể cả khi không biết trước số lượng đối số.

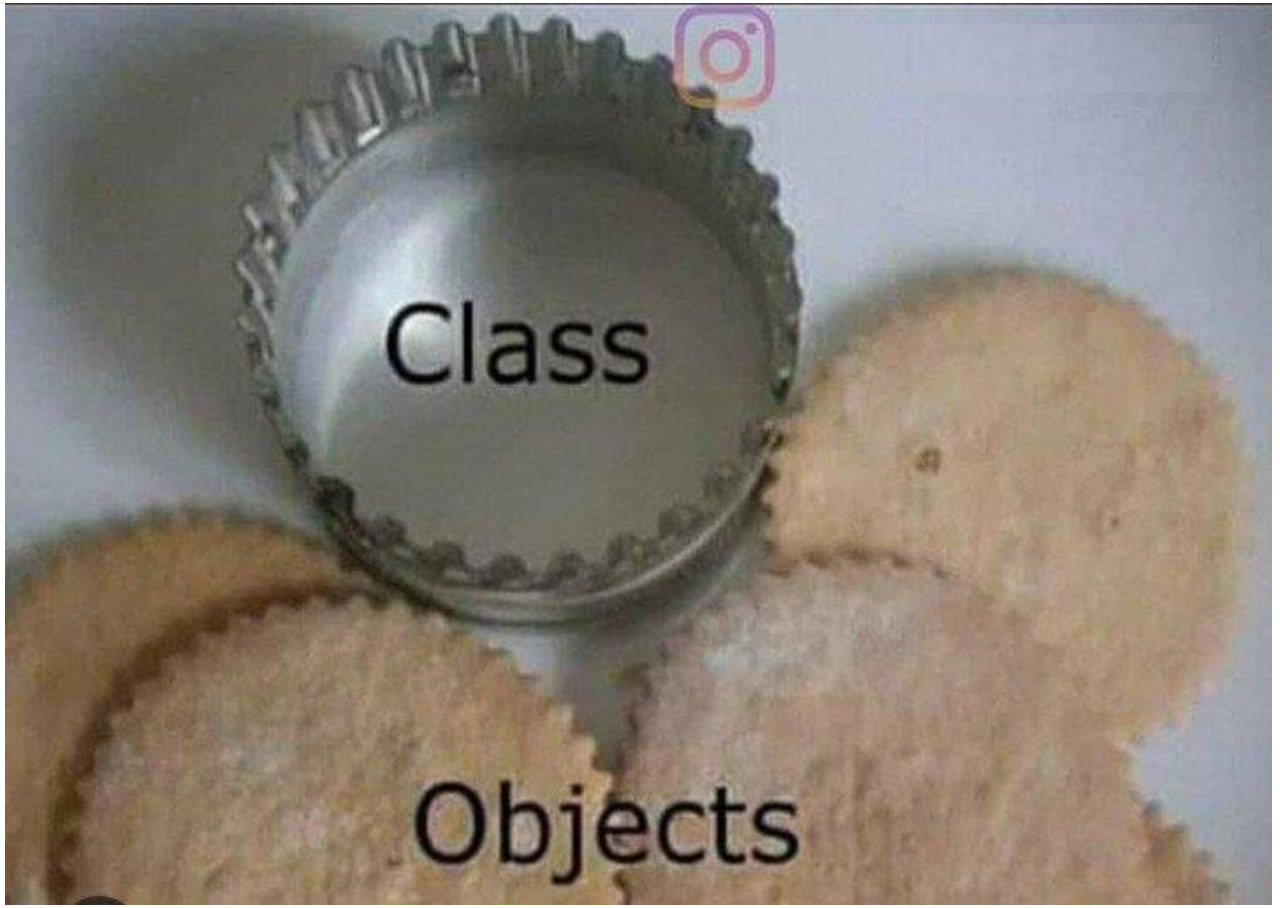
```
// with Array: include 2, 3, 4 into array rest
const [first, ...rest] = [1, 2, 3, 4];
console.log(first); // 1
console.log(rest);  // [2, 3, 4]

// with Object: include { age: 30, job: "dev" } into object info
const { name, ...info } = { name: "Dao", age: 30, job: "dev" };
console.log(name); // "Dao"
console.log(info); // { age: 30, job: "dev" }

// with Function: include 1, 2, 3 into array numbers
function sum(...numbers) {
  return numbers.reduce((a, b) => a + b, 0);
}
sum(1, 2, 3); // 6
```

Tính năng Classes, Modules

❑ **Tính năng Classes:**

❍ Là cú pháp mới để định nghĩa **constructor function** theo cách hướng đối tượng, giúp code gọn gàng, rõ ràng hơn.

❍ **Class** trong **ES6** giúp **JavaScript** gần hơn với lập trình hướng đối tượng (OOP), dễ học, dễ mở rộng, và rất phổ biến trong **TypeScript, React, Node.js**



Tính năng Classes, Modules

❑ **Tính năng Classes:**

❍ **constructor**: hàm khởi tạo, chạy khi dùng từ khóa **new**

❍ **Phương thức**: định nghĩa trực tiếp trong **class**

❍ **this**: từ khóa để gọi **property**, **method** của **class hiện tại**

❍ **super():** gọi **constructor** hoặc **method** từ **class cha**

❍ **extends**: tạo **class con kế thừa** từ **class cha**

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

class Duck extends Animal {
  speak() {
    console.log(`${this.name} barks Quack Quack.`);
  }
}

const duck = new Duck("Donan");
duck.speak(); // Donan barks.
```

Tính năng Classes, Modules

- **Tính năng Modules:**
  - **ES6 Modules** cho phép **chia nhỏ code thành các file độc lập**, có thể **export/import lẫn nhau** – giúp tổ chức, tái sử dụng và bảo trì code dễ dàng hơn.
  - **Cú pháp**:

package.json > ...

```
{
  "type": "module"
}
```

9-math.js > ...

```
// export const, function:
export const PI = 3.14;
export function add(a, b) {
  return a + b;
}
```

9-modules.js

```
// import const, function:
import { PI, add } from "./9-math.js";

console.log(PI); // 3.14
console.log(add(2, 3)); // 5
```

Tính năng Classes, Modules

❑ Tính năng Modules:

❍ Các cách Export:

Kiểu export	Cú pháp	Ghi chú	Lúc nào dùng
Named Export	export const x = 1	Import dùng { x }	Nên dùng khi cần export 1 hoặc nhiều biến, hàm
Default Export	Export default function()	Import không cần {}	Nên dùng khi cần export 1 class hoặc 1 component
Re-export	Export { func } from ‘./other.js’	Export lại từ 1 module khác mà không cần trung gian	Nên dùng khi cần tổng chức lại module theo nhóm hoặc export những gì cần dùng từ thư viện

Xử lý bất đồng bộ trong JS, ES

❑ **Bất đồng bộ (asynchronous) trong JavaScript:**

❍ **JavaScript** là ngôn ngữ đơn luồng (single-threaded), xử lý các tác vụ bất đồng bộ (asynchronous) nhờ **Event Loop**.

❍ **Event Loop** xử lý các tác vụ bất đồng bộ, giúp chương trình không bị đứng khi chờ các tác vụ như gọi API, đọc file, đợi thời gian,...

```
// Asynchronous with setTimeout() method
// => "Stop" will be printed before "Say hello after 2 senconds"
console.log("Start");

setTimeout(() => {
  console.log("Say hello after 2 senconds");
}, 2000);

console.log("Stop");
```

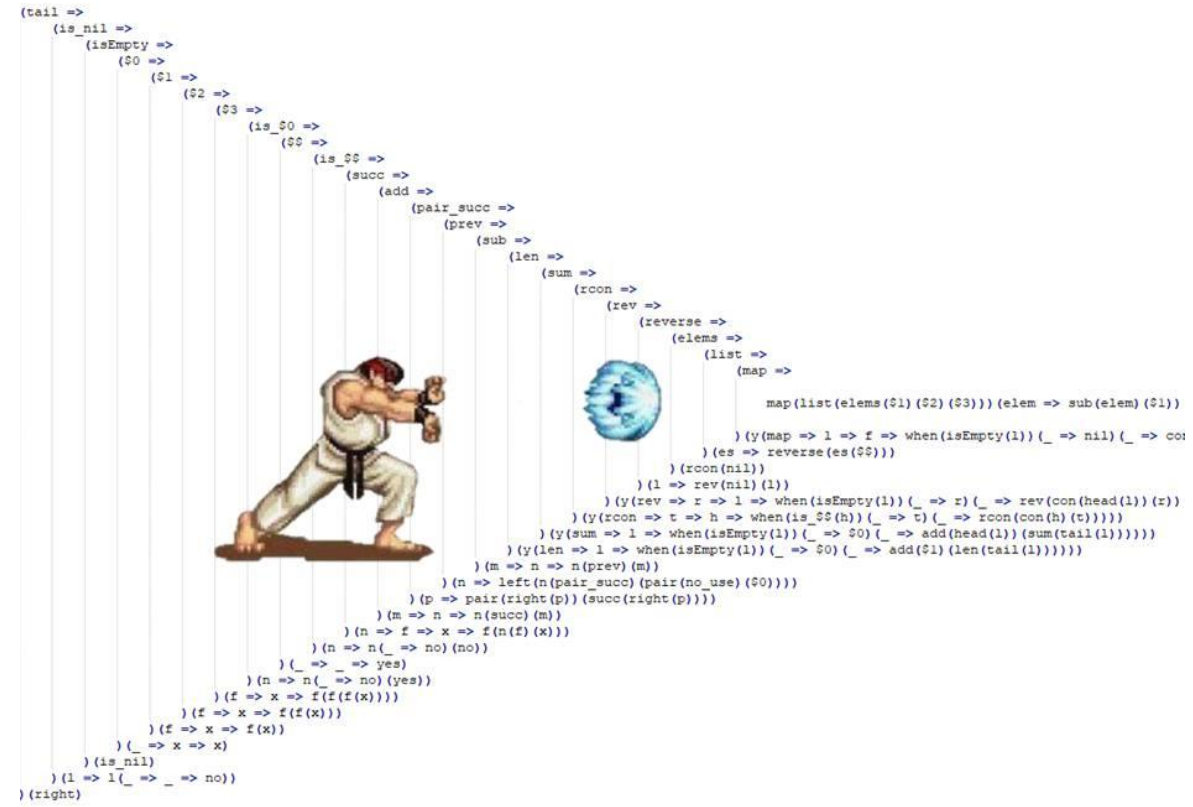
Xử lý bất đồng bộ trong JS, ES

- **Hàm gọi lại (callback) trong JavaScript:**
  - **Callback** là **hàm được truyền vào một hàm khác như một đối số**, và sẽ được gọi lại (call back) **sau khi hàm kia hoàn thành**
  - **Callback** hữu ích trong các tác vụ xử lý bất đồng bộ.

```
// Asynchronous with setInterval() method
// => Print console.log each one sencond
function printTime() {
  console.log("Each one second: " + new Date().toLocaleTimeString());
}

const intervalId = setInterval(printTime, 1000); // pass callback function

// After 10 seconds, interval time will be clear
setTimeout(() => {
  clearInterval(intervalId);
  console.log("Stopped setInterval() method after 10 seconds!");
}, 10000);
```

Xử lý bất đồng bộ trong JS, ES

- **Callback hell trong JavaScript:**
  - **Callback hell** là trường hợp **các hàm callback lồng nhau quá nhiều trong các tác vụ bất đồng bộ liên tiếp**.
  - **Callback hell** khiến code khó đọc, khó bảo trì và dễ bị lỗi logic.
  - **Giải pháp**:
    - Sử dụng **Promise (ES6), async/await (ES8)** giúp viết code tuyến tính, dễ hiểu hơn,



Xử lý bất đồng bộ trong JS, ES

❑ **Minh họa callback hell trong JS và cách giải quyết bằng Promise trong ES6:**

```
// Callback Hell
// => hard to read, maintain and occur logic errors
console.log("Start");

setTimeout(() => {
  console.log("Step 1");
  setTimeout(() => {
    console.log("Step 2");
    setTimeout(() => {
      console.log("Step 3");
      setTimeout(() => {
        console.log("Stop");
      }, 1000);
    }, 1000);
  }, 1000);
}, 1000);
```

```
// Should prevent callback hell
// Solution: Promise, async/await
function delay(ms, message) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(message);
      resolve();
    }, ms);
  });
}

console.log("Start");

delay(1000, "Step 1")
  .then(() => delay(1000, "Step 2"))
  .then(() => delay(1000, "Step 3"))
  .then(() => console.log("Stop"));
```

Xử lý bất đồng bộ trong JS, ES

❑ **Promise trong ECMAScript:**

❍ **Promise** là một tính năng trong ES6 dùng để xử lý **bất đồng bộ** (**asynchronous**) thay cho **callback lồng nhau** (**callback hell**) trong các tác vụ bất đồng bộ liên tiếp.

Xử lý bất đồng bộ trong JS, ES

- **Promise trong ECMAScript:**
  - **Promise Object** chứa cả **producing code** và gọi tới **consuming code**.
  - **Cú pháp**:

```
> let myPromise = new Promise(function(myResolve, myReject) {
// "Producing Code" (May take some time)

  myResolve(); // when successful
  myReject();  // when error
});

// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
<- ▶ Promise {<fulfilled>: undefined}
```

  - Khi **producing code** chứa kết quả, nó sẽ gọi một trong 2 **callback** sau:

Result	Call
Success	myResolve(result value)
Error	myReject(error object)

Xử lý bất đồng bộ trong JS, ES

- **Các thuộc tính của Promise:**
  - **Promise Object** hỗ trợ 2 thuộc tính: **state** và **result**.
  - **Promise Object** có thể có các trạng thái (**state**) sau:

myPromise.state	myPromise.result
"pending"	Chưa được định nghĩa undefined
"fulfilled"	Một giá trị value
"rejected"	Một đối tượng error

Xử lý bất đồng bộ trong JS, ES

❑ **Cách sử dụng Promise:**

```
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

❍ **Promise.then()** có 2 đối số, một hàm **callback** cho trường hợp thành công, và một hàm **callback** khác cho trường hợp thất bại.

❍ Cả 2 đều tùy chọn, bạn có thể thêm chỉ một hàm **callback** cho trường hợp thành công hoặc cho trường hợp thất bại.

Xử lý bất đồng bộ trong JS, ES

❑ **Xem lại ví dụ callback hell trong JS và cách giải quyết bằng Promise trong ES6:**

```
// Callback Hell
// => hard to read, maintain and occur logic errors
console.log("Start");

setTimeout(() => {
  console.log("Step 1");
  setTimeout(() => {
    console.log("Step 2");
    setTimeout(() => {
      console.log("Step 3");
      setTimeout(() => {
        console.log("Stop");
      }, 1000);
    }, 1000);
  }, 1000);
}, 1000);
```

```
// Should prevent callback hell
// Solution: Promise, async/await
function delay(ms, message) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(message);
      resolve();
    }, ms);
  });
}

console.log("Start");

delay(1000, "Step 1")
  .then(() => delay(1000, "Step 2"))
  .then(() => delay(1000, "Step 3"))
  .then(() => console.log("Stop"));
```

Tóm tắt bài học

- ❑ Tổng quan về ECMAScript
- ❑ Tính năng Let, Const
- ❑ Tính năng Arrow Function
- ❑ Tính năng Template Literals
- ❑ Tính năng Destructuring
- ❑ Tính năng Default Parameters
- ❑ Spread Operator, Rest Operator
- ❑ Tính năng Classes, Modules
- ❑ Xử lý bất đồng bộ trong JS, ES