**REPORT TUẦN 1 – LIZAI INTERNSHIP PROGRAM**

1. Kiến thức cơ bản của NodeJS
2. Operator

Trong JavaScript, các toán tử (operators) được sử dụng để thực hiện các phép tính và thao tác trên dữ liệu.

1. Toán tử số học (Arithmetic Operators):

+: Cộng

-: Trừ

\*: Nhân

/: Chia

%: Lấy phần dư

\*\*: Lũy thừa

1. Toán tử gán (Assignment Operators):

=: Gán giá trị

+=: Cộng và gán

-=: Trừ và gán

\*=: Nhân và gán

/=: Chia và gán

%=: Lấy phần dư và gán

1. Toán tử so sánh (Comparison Operators):

==: Bằng

===: Bằng về cả giá trị và kiểu dữ liệu

!=: Khác

!==: Khác về cả giá trị và kiểu dữ liệu

>: Lớn hơn

<: Nhỏ hơn

>=: Lớn hơn hoặc bằng

<=: Nhỏ hơn hoặc bằng

1. Toán tử logic (Logical Operators):

&&: Phép AND logic

||: Phép OR logic

!: Phép NOT logic

1. Toán tử chuỗi (String Operators):

+: Nối chuỗi

1. Toán tử ba ngôi (Ternary Operator):

condition ? expression1 : expression2: Kiểm tra một điều kiện và trả về một giá trị dựa trên kết quả của điều kiện.

1. Toán tử increment và decrement:

++: Tăng giá trị lên một đơn vị

--: Giảm giá trị đi một đơn vị

1. Function

Trong JavaScript, function (hàm) là một khối mã thực thi mà có thể được gọi và tái sử dụng trong chương trình.

1. Function Declaration (Khai báo hàm)

function functionName(parameters) {

// code

}

1. Function Expression (Biểu thức hàm)

let sayHello = function() {

console.log("Hello!");

};

sayHello();

1. Arrow Function

let functionName = (parameters) => {

// code

};

1. Data Type
2. Kiểu dữ liệu Number: Đại diện cho các số, bao gồm cả số nguyên và số thập phân.
3. Kiểu dữ liệu String: Đại diện cho các chuỗi ký tự, được đặt trong dấu nháy đơn hoặc nháy kép.
4. Kiểu dữ liệu Boolean: Đại diện cho giá trị true (đúng) hoặc false (sai).
5. Kiểu dữ liệu Null: Đại diện cho giá trị null, tức là không có giá trị hoặc không tồn tại.
6. Kiểu dữ liệu Undefined: Đại diện cho biến chưa được gán giá trị.
7. Kiểu dữ liệu Object: Đại diện cho một đối tượng, gồm các thuộc tính và phương thức.

(Note: Object value bao gồm các kiểu dữ liệu: Object, Date, Array)

1. Callback
2. Callback là một hàm được truyền vào một hàm khác như một tham số và được gọi lại (execute) sau khi một tác vụ hoặc một sự kiện hoàn thành.

Ví dụ:

**const** calculateSquare = (array, callback) => {  
 **const** result = array.map((value) => value \* 2);  
 callback(result);  
};  
**const** array = [1, 2, 4, 5, 7];  
  
calculateSquare(array, (result) => {  
 console.log(result);  
});

1. Callback có những đặc điểm sau:

Được truyền như một tham số: Callback là một hàm được truyền vào một hàm khác như một tham số.

Gọi lại (execute) sau khi hoàn thành: Callback được gọi lại (execute) sau khi một tác vụ hoàn thành hoặc một sự kiện xảy ra.

Đồng bộ hoặc không đồng bộ: Callback có thể được sử dụng để xử lý cả các tác vụ đồng bộ và không đồng bộ.

Xử lý lỗi: Callback có thể được sử dụng để xử lý và báo cáo lỗi trong quá trình thực thi mã. Bằng cách truyền một callback xử lý lỗi.

1. Những lưu ý:

Khi sử dụng callback chúng ta rất dễ phải gặp Callback hell

asyncFunction1(function(result1) {

asyncFunction2(result1, function(result2) {

asyncFunction3(result2, function(result3) {

asyncFunction4(result3, function(result4) {

});

});

});

});

Để tránh callback hell, có một số phương pháp như:

Sử dụng Promises: Promises là một phương pháp xử lý bất đồng bộ trong JavaScript giúp giảm bớt việc lồng callback. Promises cho phép chúng ta viết mã theo kiểu tuần tự và xử lý lỗi một cách dễ dàng.

Sử dụng Async/Await: Async/Await là một cú pháp mới trong JavaScript giúp xử lý bất đồng bộ một cách đồng bộ hơn. Nó cung cấp một cách viết mã tương tự như đồng bộ trong khi vẫn sử dụng Promises để xử lý các tác vụ bất đồng bộ.

1. Promise

Promise là một cơ chế xử lý bất đồng bộ được sử dụng để xử lý các tác vụ hoặc truy vấn mà có thể mất thời gian hoặc trả về kết quả sau một khoảng thời gian.

Promise có ba trạng thái chính:

* Pending: Trạng thái ban đầu của một Promise khi tác vụ bất đồng bộ đang được thực thi.
* Fulfilled: Đại diện cho trạng thái khi tác vụ bất đồng bộ hoàn thành thành công.
* Rejected: Đại diện cho trạng thái khi tác vụ bất đồng bộ gặp lỗi hoặc thất bại

**function** fetchData(url) {  
 **const** promise = **new** Promise((resolve, reject) => {  
 fetch(url)  
 .then((res) => {  
 **if** (res.ok) {  
 resolve(res.json());  
 } **else** {  
 reject(**new** Error(`request fail`));  
 }  
 })  
 .catch((err) => reject(err));  
 });  
 **return** promise;  
}  
fetchData("https://pokeapi.co/api/v2/pokemon/ditto")  
 .then((res) => console.log(res))  
 .catch((err) => console.log(err));

Các phương thức quan trọng của Promise:

Promise.then(): Được sử dụng để xử lý kết quả thành công (Fulfilled) của Promise.

Promise.catch(): Được sử dụng để xử lý lỗi (Rejected) của Promise.

Promise.all(): Được sử dụng để xử lý một mảng các Promise cùng một lúc. Phương thức này nhận vào một mảng các Promise và trả về một Promise mới. Khi tất cả các Promise trong mảng đều hoàn thành thành công

**const** arrUrl = [  
 "https://pokeapi.co/api/v2/pokemon/ditto",  
 "https://pokeapi.co/api/v2/pokemon-species/aegislash",  
];  
// sử dụng promise.all để xử lý đồng thời api  
**const** fetchPoke = (arrUrl) => {  
 **const** promise = arrUrl.map((url) => fetch(url));  
  
 Promise.all(promise).then((responses) => {  
 **const** jsonPromise = responses.map((res) => res.json());  
 Promise.all(jsonPromise).then((data) => {  
 data.forEach((data) => console.log(data));  
 });  
 });  
};  
  
fetchPoke(arrUrl);

Promise.race(): Được sử dụng để xử lý chỉ một Promise đầu tiên hoàn thành trong một mảng các Promise. Phương thức này nhận vào một mảng các Promise và trả về một Promise mới.

Promise.race([fetchData1(), fetchData2()])  
 .then((res) => {  
 console.log(res);  
 })  
 .catch((err) => console.log(err));

Promise.any(): Được sử dụng để xử lý chỉ một Promise đầu tiên hoàn thành và thành công trong một mảng các Promise. Phương thức này nhận vào một mảng các Promise và trả về một Promise mới.

Promise.any([fetchData1(), fetchData2()])  
 .then((res) => {  
 console.log(res);  
 })  
 .catch((err) => console.log(err));

1. Async/Await

Async/await là một cú pháp trong JavaScript được sử dụng để xử lý bất đồng bộ một cách đồng bộ hơn. Nó cung cấp một cách viết mã tương tự như đồng bộ trong khi vẫn sử dụng Promises để xử lý các tác vụ bất đồng bộ.

**const** fetchApi = **async** (url) => {  
 **const** res = **await** fetch(url);  
 **if** (res.ok) {  
 **const** json = **await** res.json();  
 console.log(json);  
 } **else** {  
 **return new** Error("api not found");  
 }  
};  
  
fetchApi("https://pokeapi.co/api/v2/pokemon/ditto");

1. Những đặc điểm của Async/Await:

* Cú pháp đơn giản: Async/await giúp viết code bất đồng bộ theo kiểu tương tự như code đồng bộ, làm cho nó dễ đọc và hiểu hơn so với việc sử dụng các callback hoặc Promises.
* Sử dụng từ khóa "async": Để khai báo một hàm bất đồng bộ, chúng ta sử dụng từ khóa "async" trước khai báo hàm. Điều này cho phép hàm trả về một Promise.
* Sử dụng từ khóa "await": Để đợi cho một Promise được giải quyết (resolve) hoặc từ chối (reject), chúng ta sử dụng từ khóa "await" trong hàm bất đồng bộ.
* Giải quyết callback hell: Async/await là một trong những công nghệ giúp giải quyết callback hell. Thay vì lồng callback một cách sâu trong mã

**async function** foo() {  
 **return** `bar`;  
}  
**function** foo2() {  
 **return** Promise.resolve(`bar`);  
}

Hai function này về mặt bản chất chính là promise mà thay vào đó chúng ta sử dụng async.

1. Debug

Trong javascript chúng ta có thể debug bằng Console.log hoặc debugger, và 1 vài phương thức khác

1. console.log()

Để thực hiện gỡ lỗi và sử dụng console.log() cho mục đích hiển thị thông tin trong quá trình phát triển.

Khi sử dụng console.log() để gỡ lỗi, bạn có thể chèn các thông báo và giá trị của biến vào mã của bạn để kiểm tra giá trị và xem luồng thực thi.

console.log("Giá trị của x là: " + x);

1. debugger

debugger là một từ khóa trong JavaScript, khi gặp nó, trình duyệt sẽ dừng việc thực thi mã tại đó và mở một bước gỡ lỗi (debugger) để bạn có thể xem và kiểm tra các giá trị của biến.

Bạn có thể đặt từ khóa debugger ở bất kỳ đâu trong mã của bạn để dừng thực thi tại điểm đó.

Khi trình duyệt gặp từ khóa debugger, nó sẽ mở một bước gỡ lỗi (debugger) trong công cụ phát triển của trình duyệt.

let x = 5;

debugger;

console.log("Giá trị của x là: " + x);