جاوا اسکریپت از سایت javascript.info :

Try catch: یک راه خوب برای کنترل برنامه بعد از وقوع خطاها:

سه نوع خطا ممکن است پیش بیاید:

* Error معمولی که در حالتی است که اجرا به دلایل نامعلوم به خطا میخورد.
* SyntaxError: خطایی است که در نتیجه مشکلات syntax به وجود می آید.
* Reference Error: خطایی است که در اثر ارجاع به متغیر یا مقادیر ناموجود پیش می آید.

ساختار try – catch- finally به این صورت عمل میکند که اگر خطا پیش نیاید که بخش های try و finally اجرا می شوند و اگر خطا پیش بیاید هر سه قسمت اجرا می شوند. در واقع این ساختار برای حالتی است که در هر صورت میخواهیم که بخشی از کد که بعد از try قرار داده شده حتما اجرا شود. مثال:

let num = +prompt("Enter a positive integer number?", 35)

let diff, result;

function fib(n) {

if (n < 0 || Math.trunc(n) != n) {

throw new Error("Must not be negative, and also an integer.");

}

return n <= 1 ? n : fib(n - 1) + fib(n - 2);

}

let start = Date.now();

try {

result = fib(num);

} catch (err) {

result = 0;

} finally {

diff = Date.now() - start;

}

alert(result || "error occurred");

alert( `execution took ${diff}ms` );

نکته: در حین کد میتوان نوع خطای به وجود آمده را با استفاده از دستور instance of چک کرده و براساس نوع خطای به وجود آمده تصمیم گرفت. مثال:

let json = '{ "age": 30 }'; // incomplete data

try {

let user = JSON.parse(json);

if (!user.name) {

throw new SyntaxError("Incomplete data: no name");

}

blabla(); // unexpected error

alert( user.name );

} catch (err) {

if (err instanceof SyntaxError) {

alert( "JSON Error: " + err.message );

} else {

throw err; // rethrow (\*)

}

}

هر خطا در حالت استفاده از try – catch سه فیلد اصلی دارد:

* Name: نام خطاست که در واقع نوع خطاست که در بالا توضیح داده شد.
* Message : متن پیغام خطاست.
* Stack: در واقع مکان وقوع خطا را در اجراهای مختلف نشان میدهد.

گاهی میتوان خطاهای جدید را از روی نوع خطاهای قبلی ارث بری کرده و کنترل بهتری روی خطاها داشت. به عنوان مثال :

class ValidationError extends Error {

constructor(message) {

super(message);

this.name = "ValidationError";

}

}

// Usage

function readUser(json) {

let user = JSON.parse(json);

if (!user.age) {

throw new ValidationError("No field: age");

}

if (!user.name) {

throw new ValidationError("No field: name");

}

return user;

}

// Working example with try..catch

try {

let user = readUser('{ "age": 25 }');

} catch (err) {

if (err instanceof ValidationError) {

alert("Invalid data: " + err.message); // Invalid data: No field: name

} else if (err instanceof SyntaxError) { // (\*)

alert("JSON Syntax Error: " + err.message);

} else {

throw err; // unknown error, rethrow it (\*\*)

}

}

Callbacks: گاهی پیش می آید که نیاز است که توابعی به صورت asynchronous اجرا شوند، یعنی نیاز نباشد که کار یک بخش به طور کامل به اتمام برسد و سپس نوبت به کار بعدی برسد. مثلا در مورد توابع setTimeOut که اجرای آنها طول میکشد. در این طور مواقع از Callback ها استفاده میشود.

مثال: در این مثال یک اسکریپت قرار است ایجاد شده و سپس اجرا شود، اما ایجاد آن ممکن است زمان ببرد و نیازی نیست که اجرای بقیه پروژه منتظر بماند، همچنین لازم است بدانیم ایجاد آن چه زمانی به اتمام میرسد تا بتوانیم بقیه پروژه را فراخوانی کنیم:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(script);

document.head.append(script);

}

مثلا در این برنامه از تابع script.onload استفاده شده است که زمانی اجرا می­شود که اسکریپتی به پایان برسد. این تابع پارامتری از نوع تابع callback می گیرد. یک نمونه از اجرای این تکه کد در ادامه دیده میشود.

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {

alert(`Cool, the script ${script.src} is loaded`);

});

در واقع بعد از اینکه اسکریپت به طور کامل لود شد، پیغام مبنی بر آن ظاهر می گردد.

Callback in callback : بعضی مواقع میتوان پشت سر هم این callbackها را اجرا نمود. مثال:

loadScript('/my/script.js', function(script) {

loadScript('/my/script2.js', function(script) {

loadScript('/my/script3.js', function(script) {

// ...continue after all scripts are loaded

});

});

});

اما نکته ای که نباید از آن غافل شد، handle کردن خطاهای پیش آمده است، مانند مثال زیر:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(null, script);

script.onerror = () => callback(new Error(`Script load error for ${src}`));

document.head.append(script);

}

loadScript('/my/script.js', function(error, script) {

if (error) {

// handle error

} else {

// script loaded successfully

}

});

نکته: اصطلاح callback hell یا Pyramid of doom زمانی پیش می آید که callback های تو در توی زیادی وجود داشته که هم خوانایی کد را با مشکل مواجه میکند و هم کنترل برنامه را سخت میکند. که راه حل های مختلفی برای جلوگیری از آن وجود دارد که یک نمونه آن call کردن توابع به صورت standalone به جای بازگشتی میباشد. در ادامه یک مثال به همراه راه حل آن آمده است:

|  |  |
| --- | --- |
| loadScript('1.js', step1);  function step1(error, script) {  if (error) {  handleError(error);  } else {  // ...  loadScript('2.js', step2);  }  }  function step2(error, script) {  if (error) {  handleError(error);  } else {  // ...  loadScript('3.js', step3);  }  }  function step3(error, script) {  if (error) {  handleError(error);  } else {  // ...continue after all scripts are loaded (\*)  }  } | loadScript('1.js', function(error, script) {  if (error) {  handleError(error);  } else {  // ...  loadScript('2.js', function(error, script) {  if (error) {  handleError(error);  } else {  // ...  loadScript('3.js', function(error, script) {  if (error) {  handleError(error);  } else {  // ...continue after all scripts are loaded (\*)  }  });  }  });  }  }); |

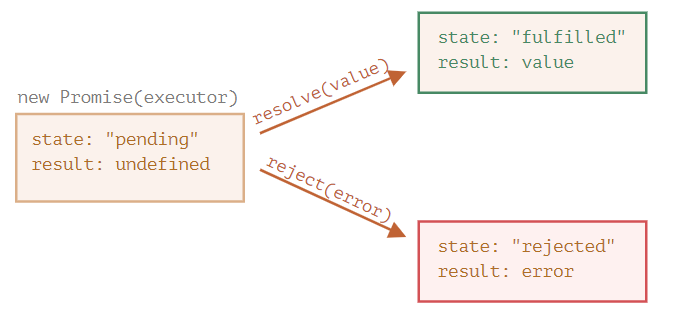
Promiseها: راه حل اصلی کنترل اجرای Asynchronous برنامه که حاوی توابعی است که اجرای آنها زمان میبرد، استفاده از promise هاست. در واقع Promise ها قولی است که تابع در حال اجرا میدهد، اینکه اگر به درستی اجرا شد چه اتفاقی بیفتد و اگر دچار خطا شد چه اتفاقی بیفتد. به حالت اول resolve و به حالت دوم reject گفته می شود.

let promise = new Promise(function(resolve, reject) {

// executor (the producing code, "singer")

});

تابعی که promise میشود در ابتدا حالت pending داشته و اگر به درستی اجرا شود به حالت fulfilled و اگر خطا پیش بیاید به حالت rejected میرود، همچنین خروجی تابع ابتدا undefined بوده و در حالت اجرای موفق خروجی value و در حالت اجرای همراه با خطا خروجی error میدهد.



هر تابع promise تنها میتواند یک خروجی درست و یک خروجی خطا داشته باشد و خروجی های بعدی نادیده گرفته میشوند.

let promise = new Promise(function(resolve, reject) {

resolve("done");

reject(new Error("…")); // ignored

setTimeout(() => resolve("…")); // ignored

});

اگر یک تابع بخواهد هر دو خروجی را بر اساس وضعیتهای مختلف داشته باشد از .then استفاده میشود. Then دو پارامتر میگیرد، یک تابع برای حالت خروجی درست و یک تابع برای حالت خروجی همراه با خطا:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => resolve("done!"), 1000);

});

// resolve runs the first function in .then

promise.then(

result => alert(result), // shows "done!" after 1 second

error => alert(error) // doesn't run

);

تابع .catch زمانی به کار میرود که تنها حالت وقوع خطا برای ما اهمیت داشته باشد و در واقع برای حالت error handling به کار می رود.

let promise = new Promise((resolve, reject) => {

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

// .catch(f) is the same as promise.then(null, f)

promise.catch(alert); // shows "Error: Whoops!" after 1 second

تابع .finally زمانی به کار میرود که در هر دو صورت خروجی درست و خطا، بخشی از برنامه باید حتما اجرا شود، همان طور که در try-catch-finally دیدیم.

new Promise((resolve, reject) => {

/\* do something that takes time, and then call resolve or maybe reject \*/

})

// runs when the promise is settled, doesn't matter successfully or not

.finally(() => stop loading indicator)

// so the loading indicator is always stopped before we go on

.then(result => show result, err => show error)

برای درک بهتر موضوع و شرح تفاوت promise ها با callback مثال قبلی loadscript را با روش promise میبینیم:

function loadScript(src) {

return new Promise(function(resolve, reject) {

let script = document.createElement('script');

script.src = src;

script.onload = () => resolve(script);

script.onerror = () => reject(new Error(`Script load error for ${src}`));

document.head.append(script);

});

}

let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(

script => alert(`${script.src} is loaded!`),

error => alert(`Error: ${error.message}`)

);

Promise chaining: گاهی اوقات پیش می آید که زنجیره ای از Promise ها پشت سر هم اجرا می شوند. در این حالت خروجی هر promise به promise بعدی خواهد رفت. مثال:

new Promise(function(resolve, reject) {

setTimeout(() => resolve(1), 1000); // (\*)

}).then(function(result) { // (\*\*)

alert(result); // 1

return result \* 2;

}).then(function(result) { // (\*\*\*)

alert(result); // 2

return result \* 2;

}).then(function(result) {

alert(result); // 4

return result \* 2;

});

Error Handling با Promiseها: با استفاده از .catch میتوان اجرای برنامه را کنترل کرده و هرجا که در then های موجود خطایی پیش بیاید، بلافاصله اجرا متوقف شده و به catch منتقل می شود. مثال:

fetch('/article/promise-chaining/user.json')

.then(response => response.json())

.then(user => fetch(`https://api.github.com/users/${user.name}`))

.then(response => response.json())

.then(githubUser => new Promise((resolve, reject) => {

let img = document.createElement('img');

img.src = githubUser.avatar\_url;

img.className = "promise-avatar-example";

document.body.append(img);

setTimeout(() => {

img.remove();

resolve(githubUser);

}, 3000);

}))

.catch(error => alert(error.message));

گاهی اوقات میتوان یک خطا را که به catch فرستاده شده را handle کرده و دوباره به then بعدی اجرای برنامه را بازگرداند. مثال:

new Promise((resolve, reject) => {

throw new Error("Whoops!");

}).catch(function(error) {

alert("The error is handled, continue normally");

}).then(() => alert("Next successful handler runs"));

Promise APIs: شش تابع استاتیک در کلاس promise وجود دارد که قابلیتهای مختلفی به این مفهوم میدهد.

1- Promise.all : این تابع چند promise را به صورت یکجا پردازش کرده و نتیجه آن را به صورت یکجا برمیگرداند. این مکانیزم برای کاربردهایی همچون پردازش url های مختلف کاربرد دارد. مثال:

let urls = [

'https://api.github.com/users/iliakan',

'https://api.github.com/users/remy',

'https://api.github.com/users/jeresig'

];

// map every url to the promise of the fetch

let requests = urls.map(url => fetch(url));

// Promise.all waits until all jobs are resolved

Promise.all(requests)

.then(responses => responses.forEach(

response => alert(`${response.url}: ${response.status}`)

));

تابع promise.all به این صورت عمل میکند که اگر یکی از promise ها reject شود، همگی reject خواهند شد و برای کاربردهای یا همه یا هیچ کدام مناسب است، مثال:

Promise.all([

new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),

new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),

new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))

]).catch(alert); // Error: Whoops!

2- Promise.allSettled : این تابع چند promise را به صورت یکجا فراخوانی میکند، اما بر خلاف promise.all اگر یکی از promise ها اجرا نشود، بقیه در صورت عدم مشکل اجرا می شوند و یک آرایه به عنوان خروجی برمیگردد که دو عضوی است، status و value . برای هر کدام از promise ها مقدار و وضعیت برگردانده می شود. مثال:

let urls = [

'https://api.github.com/users/iliakan',

'https://api.github.com/users/remy',

'https://no-such-url'

];

Promise.allSettled(urls.map(url => fetch(url)))

.then(results => { // (\*)

results.forEach((result, num) => {

if (result.status == "fulfilled") {

alert(`${urls[num]}: ${result.value.status}`);

}

if (result.status == "rejected") {

alert(`${urls[num]}: ${result.reason}`);

}

});

});

در این مثال دو promise اول اجرا شده و promise سوم اجرا نمیشود. بنابراین خروجی به شکل زیر خواهد بود:

[

{status: 'fulfilled', value: ...response...},

{status: 'fulfilled', value: ...response...},

{status: 'rejected', reason: ...error object...}

]

3- Promise.race : این تابع مانند promise.all است، یعنی چند promise را به صورت همزمان اجرا میکند، اما تنها اولین promise برگردانده شده و بقیه promise ها ignore میشوند. در مثال زیر تنها promise اول اجرا شده و نتیجه آن برگردانده میشود.

Promise.race([

new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),

new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),

new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))

]).then(alert); // 1

4- Promise.any : این تابع مانند promise.race است، اما تنها اولین promise ی برگردانده میشود که نتیجه آن موفقیت آمیز باشد و بقیه promise ها ignore میشوند. اگر هیچ کدام از promise ها موفقیت آمیز نباشند، Aggregate Error برگردانده میشود. مثال:

Promise.any([

new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 1000)),

new Promise((resolve, reject) => setTimeout(() => resolve(1), 2000)),

new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))

]).then(alert); // 1

به عنوان مثال در حالت زیر تمامی promise ها reject شده و خطا برگردانده می شود:

Promise.any([

new Promise((resolve, reject) => setTimeout(() => reject(new Error("Ouch!")), 1000)),

new Promise((resolve, reject) => setTimeout(() => reject(new Error("Error!")), 2000))

]).catch(error => {

console.log(error.constructor.name); // AggregateError

console.log(error.errors[0]); // Error: Ouch!

console.log(error.errors[1]); // Error: Error!

});

5- Promise.resolve : این تابع یک promise موفقیت آمیز(resolved) با نتیجه value برمیگرداند. مثال:

let cache = new Map();

function loadCached(url) {

if (cache.has(url)) {

return Promise.resolve(cache.get(url)); // (\*)

}

return fetch(url)

.then(response => response.text())

.then(text => {

cache.set(url,text);

return text;

});

}

6- Promise.reject : این تابع یک promise شکست خورده(rejected) با نتیجه خطا برمیگرداند. این تابع در عمل هیچ گاه استفاده نمیشود.

Async-await: یک راه خیلی ساده تر برای استفاده از مکانیزمهای promise-based می باشد. syntax آن به صورت زیر می باشد:

async function f() {

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("done!"), 1000)

});

let result = await promise; // wait until the promise resolves (\*)

alert(result); // "done!"

}

f();

نکته: await را نمیتوان در توابع معمولی به کار برد و حتما باید در توابع دارای Async بکار گرفته شود.

برای مقایسه سادگی استفاده از async-await ها، یک مثال از قبل را که با promise معمولی نوشته شده بود را با استفاده از async-await بازنویسی میکنیم:

async function showAvatar() {

// read our JSON

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

// read github user

let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);

let githubUser = await githubResponse.json();

// show the avatar

let img = document.createElement('img');

img.src = githubUser.avatar\_url;

img.className = "promise-avatar-example";

document.body.append(img);

// wait 3 seconds

await new Promise((resolve, reject) => setTimeout(resolve, 3000));

img.remove();

return githubUser;

}

showAvatar();

نکته: وقتی از async-await ها استفاده میکنیم، از همان try-catch معمولی میتوان استفاده کرد.

async function f() {

try {

let response = await fetch('http://no-such-url');

} catch(err) {

alert(err); // TypeError: failed to fetch

}

}

اگر try-catch به هر دلیلی فراموش شد، میتوان آن را به صورت دستی استفاده کرد. مثال:

async function f() {

let response = await fetch('http://no-such-url');

}

// f() becomes a rejected promise

f().catch(alert);

توابع async-awaitکاملا با promise.all سازگار بوده و با کدی بسیار خواناتر با آنها کار میکنند. مثال:

let results = await Promise.all([

fetch(url1),

fetch(url2),

]);

نکته: وقتی قصد داریم از توابع async و با حفظ promise-based بودن آن در درون توابع معمولی استفاده کنیم، باید به صورت دستی .then را استفاده کنیم. مثال:

async function wait() {

await new Promise(resolve => setTimeout(resolve, 1000));

return 10;

}

function f() {

// shows 10 after 1 second

wait().then(result => alert(result));

}

Generator Functions: توابع از این نوع برخلاف توابع Regular هستند و اجرای آنها را میتوان معطل کرده و بیش از یک خروجی (به شکل object) در چند مرحله داشت. خروجی این توابع به جای return توسط yield می­باشد و خروجی هر yield یک object است که دارای دو فیلد value و done می باشد. value که مقداری است که توسط yield بازگردانده میشود و done ادامه دار بودن اجرا را نشان میدهد، تا زمانی که اجرا ادامه دارد done برابر false بوده و اگر اجرا به خاتمه رسیده باشد، مقدار true خواهد داشت. یک مثال ساده ببینیم:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

// "generator function" creates "generator object"

let generator = generateSequence();

alert(generator); // [object Generator]

در مثال بالا اجرای تابع به سه بخش تقسیم شده است و هر بار یک object برگردانده می­شود. مکانیزم توابع generation به linklist ها نیز شباهت دارد و با تابع next به جلو میرود. مثال:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

let generator = generateSequence();

let one = generator.next();

alert(JSON.stringify(one)); // {value: 1, done: false}

مکانیزم next به این صورت است که اجرای برنامه را تا اولین yield بعدی پیش برده و سپس خروجی آن را برمیگرداند و نکته اینجاست که خطوط برنامه را تا ابتدای آن yield اجرا میکند.

به دلیل ماهیت تکرار پذیری و قابل شمارش بودن اجراهای generation function ها میتوان از ابزارهای iteration همانند spread ها و حلقه های for استفاده کرد. مثال:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

let generator = generateSequence();

for(let value of generator) {

alert(value); // 1, then 2

}

نکته در مورد خروجی کد بالا این است که مقدار 3 بر گردانده نمیشودو فقط مقادیر بعد از yield برگردانده میشوند. و مثال در مورد spread ها در ادامه آمده است:

function\* generateSequence() {

yield 1;

yield 2;

yield 3;

}

let sequence = [0, ...generateSequence()];

alert(sequence); // 0, 1, 2, 3

generation composition: میتوان خروجی توابع generation را با هم ترکیب کرده و خروجی یک تابع generation را به عنوان ورودی تابع دیگر در نظر گرفت. مثال:

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) yield i;

}

function\* generatePasswordCodes() {

// 0..9

yield\* generateSequence(48, 57);

// A..Z

yield\* generateSequence(65, 90);

// a..z

yield\* generateSequence(97, 122);

}

let str = '';

for(let code of generatePasswordCodes()) {

str += String.fromCharCode(code);

}

alert(str); // 0..9A..Za..z

مثال بالا یک تابع تولید پسورد است که ترکیبی از اعداد و حروف بزرگ و حروف کوچک را می­سازد. یک نمونه کد جدید در کد بالا استفاده شده است و آنهم yield\* می­باشد و کار آن این است که به صورت یک حلقه خروجی ها را برمیگرداند.

دو طرفه بودن yield : یکی از نکات مهم در استفاده از generation function ها این است که yield ها همانند یک خیابان دو طرفه هستند و میتوانند خروجی خود را به عنوان ورودی به اجرای yield بعدی بدهند. به مثال زیر دقت کنید:

function\* gen() {

let ask1 = yield "2 + 2 = ?";

alert(ask1); // 4

let ask2 = yield "3 \* 3 = ?"

alert(ask2); // 9

}

let generator = gen();

alert( generator.next().value ); // "2 + 2 = ?"

alert( generator.next(4).value ); // "3 \* 3 = ?"

alert( generator.next(9).done ); // true

در کد بالا زمانی که تکه کد generator.next(4) اجرا می­شود، عدد 4 به عنوان خروجی yield قبلی در نظر گرفته می­شود، یعنی در متغیر ask1 قرار میگیرد، لذا بعد از اجرا شدن این next(4) خط alert(ask1) خروجی 4 را چاپ کرده و سپس yield بعدی اجرا شده و 3\*3 =? را چاپ می­کند. خط بعدی که generator.next(9) هست، دو کار انجام می­دهد، ابتدا 9 را به خروجی قبلی داده و آن را در ask2 میریزد. سپس alert(ask2) را اجرا کرده و مقدار 9 را چاپ میکند و سپس چون yield دیگری وجود ندارد، بنابراین مقدار done یعنی false را چاپ می­کند.

Generator.return: این دستور اجرا را به پایان رسانده و value موجود در خود را برمیگرداند و بعد از آن اگر yield ی وجود داشته باشد با مقدار value برابر با undefined و مقدار done برابر با true خواهد بود. مثال:

function\* gen() {

yield 1;

yield 2;

yield 3;

}

const g = gen();

g.next(); // { value: 1, done: false }

g.return('foo'); // { value: "foo", done: true }

g.next();

Modules : زمانی که پروژه ها بزرگ می­شوند، نیاز است که برنامه به قطعات مختلفی تقسیم شده و هر قسمت از آن در فایل دیگری وجود داشته باشد. برای فراخوانی ماژول ها از دستورات import و export استفاده میشود به این صورت که فایلی که ماژول را صادر میکند باید توسط دستور export این کار را انجام دهد و فایلی که قرار است از آن ماژول استفاده کند از دستور import استفاده می­کند. مثال:

export function sayHi(user) {

alert(`Hello, ${user}!`);

}

import {sayHi} from './sayHi.js';

alert(sayHi); // function...

sayHi('John'); // Hello, John!

نکته 1: اسکریپتهایی که دارای import هستند، زمانی که در html page قرار می­گیرند، باید تگ ماژول بخورند. مثال:

<script type="module">

import {sayHi} from './say.js';

document.body.innerHTML = sayHi('John');

</script>

نکته 2: اگر بیش از یکبار یک ماژول Export شود، تنها بار اول اجرا و ارزیابی میشود، بدین صورت که اگر ماژولی export شده و بعد از آن مقدار یک متغیر تغییر کند، اگر دوباره آن ماژول Export شود، مقدار متغیر همان مقدار تغییر پیدا کرده است و نه مقدار اولیه. مثال:

import {admin} from './admin.js';

admin.name = "Pete";

import {admin} from './admin.js';

alert(admin.name); // Pete

نکته 3: اسکریپتهای از نوع module دیرتر اجرا می­شوند و اجرای آنها به لود شدن کامل صفحه html موکول می­گردد. مگر اینکه از async استفاده کنیم که در این صورت به محض کامل شدن لودینگ اسکریپت، اجرا می­شود. مثال:

<script async type="module">

import {counter} from './analytics.js';

counter.count();

</script>

نکته 4: ماژول های بدون آدرس برخلاف nodejs نادیده گرفته می­شوند. مثال:

import {sayHi} from 'sayHi'; // Error, "bare" module

export قبل از declaration: قبل و در حین اعلان متغیرها و توابع و کلاسها و حتی مقادیر ثابت می­توان export را انجام داد. مثال:

export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// export a constant

export const MODULES\_BECAME\_STANDARD\_YEAR = 2015;

// export a class

export class User {

constructor(name) {

this.name = name;

}

}

ولی export از اعلان می­تواند جدا هم باشد. مثال:

function sayHi(user) {

alert(`Hello, ${user}!`);

}

function sayBye(user) {

alert(`Bye, ${user}!`);

}

export {sayHi, sayBye};

import \*: می­توان تمامی ماژولها و فایلها را به صورت یکجا با این دستور import کرد. مثال:

import \* as say from './say.js';

say.sayHi('John');

say.sayBye('John');

import as: میتوان فایلها و ماژولهایی که import می­شوند را نامگذاری مجدد کرد. مثال:

import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!

bye('John'); // Bye, John!

export as: میتوان فایلها و ماژولهایی که export می­شوند را نامگذاری مجدد کرد. مثال:

export {sayHi as hi, sayBye as bye};

import \* as say from './say.js';

say.hi('John'); // Hello, John!

say.bye('John'); // Bye, John!

export default: می­توان تنها یکی از فایلها را به عنوان default جوری export کرد که نیازی به استفاده از {} هنگام import آن نباشد و باید توجه داشت که تنها یکی از فایلها را می­توان به عنوان default در نظر گرفت. مثال:

export default class User { // just add "default"

constructor(name) {

this.name = name;

}

}

-------------------------------------------------------------

import User from './user.js'; // not {User}, just User

new User('John');

re-export : با این دستور می­توان فایلی را که import کرده ایم، بلافاصله در فایل دیگری export کرد. برای مواردی به درد میخورد که یک فایل را به عنوان فایل مرکزی در نظر گرفت. مثال:

export {sayHi} from './say.js'; // re-export sayHi

export {default as User} from './user.js';

dynamic report: نمیتوان چیزی را که قرار است import یا Export کرد از خروجی یک تابع استخراج کرده و یا خروجی یک شرط را در نظر گرفت. به عنوان مثال export های زیر خطا می­دهند:

import ... from getModuleName();

if(...) {

import ...; // Error, not allowed!

}

{

import ...; // Error, we can't put import in any block

}

تنها جایی که می­توان به نوعی این کار را به صورت dynamic انجام داد، حالتی است که از توابع async و await استفاده کرده و در صورت resolve شدن، آن را به عنوان خروجی برگرداند. مثال:

let modulePath = prompt("Which module to load?");

import(modulePath)

.then(obj => <module object>)

.catch(err => <loading error, e.g. if no such module>)

let {hi, bye} = await import('./say.js');

hi();

bye();