

**CÔNG TY TNHH CÔNG NGHỆ SỐ NAM PHƯƠNG**

**TÀI LIỆU THỰC TẬP**

**MỤC LỤC**

[1) Variables 4](#_Toc175670982)

[1.1) Khái niệm 4](#_Toc175670983)

[1.2) Quy luật khai báo 5](#_Toc175670984)

[2) Datatypes 5](#_Toc175670985)

[2.1) Ý nghĩa 5](#_Toc175670986)

[2.2) Ý tưởng 6](#_Toc175670987)

[3) Type conversions 7](#_Toc175670988)

[4) Operators 8](#_Toc175670989)

[5) Comparisions 9](#_Toc175670990)

[6) Conditional branching 9](#_Toc175670991)

[7) Logical operators 12](#_Toc175670992)

[8) Switch 12](#_Toc175670993)

[9) Nullish coalescing operator 13](#_Toc175670994)

[10) Loops 14](#_Toc175670995)

[10.1) For loops 14](#_Toc175670996)

[10.2) While loops 16](#_Toc175670997)

[11) Functions 17](#_Toc175670998)

[12) Asynchronous, Promise 18](#_Toc175670999)

[13) Handling errors 19](#_Toc175671000)

[14) GIT 21](#_Toc175671001)

[14.1) Khái niệm 21](#_Toc175671002)

[14.2) Lợi ích 21](#_Toc175671003)

[14.3) Thuật ngữ 22](#_Toc175671004)

[14.3.1) Branching 22](#_Toc175671005)

[14.3.2) Git logs 22](#_Toc175671006)

[14.3.3) Git add 22](#_Toc175671007)

[14.3.4) Git commit 22](#_Toc175671008)

[14.3.5) Git push, pull 23](#_Toc175671009)

[14.3.6) Git fetch 23](#_Toc175671010)

[14.4) Thực hành 23](#_Toc175671011)

[15) React 24](#_Toc175671012)

[15.1) Khái niệm 24](#_Toc175671013)

[15.2) JSX 25](#_Toc175671014)

[15.3) Components 25](#_Toc175671015)

[15.4) Props, State 25](#_Toc175671016)

[15.5) Lifecycle 26](#_Toc175671017)

[15.6) Portals 26](#_Toc175671018)

[15.7) Error Boundaries 26](#_Toc175671019)

[15.8) HOCs 27](#_Toc175671020)

[15.9) Hooks trong React 27](#_Toc175671021)

[15.9.1) useState 27](#_Toc175671022)

[15.9.2) useEffect 28](#_Toc175671023)

[15.9.3) useRef 29](#_Toc175671024)

[15.9.4) Memo 29](#_Toc175671025)

[15.9.5) useCallback 31](#_Toc175671026)

[15.9.6) useMemo 32](#_Toc175671027)

[15.9.7) useContext 32](#_Toc175671028)

[15.9.8) useReducer 34](#_Toc175671029)

# Variables

## Khái niệm

Variables hay còn được gọi là các biến, được dùng để lưu trữ và truy xuất dữ liệu. Cần phải khai báo trước khi sử dụng các biến này. Có 4 cách khai báo biến:

* Khai báo tự động:
  + x = 10
* Sử dụng từ khóa var
  + var x = 10
* Sử dụng từ khóa let
  + let x = 10
* Sử dụng từ khóa const
  + const x = 10

Từ khóa var được sử dụng từ năm 1995 đến 2015. Và sau đó, từ khóa let và const được thêm vào trong Javascript. Hiện tại, từ khóa var chỉ nên được sử dụng để viết mã cho các trình duyệt cũ. Đặc biệt, việc sử dụng let và const để khai báo biến một cách linh động hơn, làm cho việc định nghĩa giá trị của các biến một cách chặt chẽ hơn. Đối với biến được khai bằng const thì giá trị của biến đó là cố định và không thể thay đổi, ngược lại thì khi sử dụng let để khai báo biến thì giá trị của biến đó có thể thay đổi.

Khi sử dụng các từ khóa này, cần chú ý:

* Luôn luôn được dùng để khai báo biến
* Luôn sử dụng const nếu giá trị trong biến không thay đổi
* Luôn sử dụng const nếu kiểu dữ liệu trong biến không thay đổi (Array hoặc Object)
* Chỉ sử dụng let khi không thể sử dụng const
* Chỉ sử dụng var nếu cho các trình duyệt cũ

## Quy luật khai báo

Tất cả các biến trong Javascript sẽ được nhận diện với tên duy nhất của chúng, các tên này không được phép trùng nhau. Các tên biến này có thể ngắn chỉ gồm một ký tự như (x, y, z) hoặc có thể khai báo để hiểu rõ ý nghĩa về công dụng của biến như là (totalAge, totalVolume).

Các quy luật chung cho việc đặt tên biến:

* Có thể chứa các chữ cái, kí tự, \_ và $
* Tên phải bắt đầu bằng chữ cái
* Tên cũng có thể bắt đầu bằng \_ và $
* Tên có phân biệt chữ hoa và chữ thường (x và X là hai biến khác nhau)
* Các từ đã được định nghĩa trong Javascript như let, const, var thì không thể được sử dụng để đặt tên biến

# Datatypes

## Ý nghĩa

Trong Javascript có 8 kiểu dữ liệu khác nhau:

* String (Kiểu dữ liệu chuỗi)
* Number (Kiểu dữ liệu số)
* Bigint (Số nguyên lớn)
* Boolean (Chứa giá trị true hoặc false)
* Undefined (Không xác định)
* Null (Chưa có giá trị)
* Symbol
* Object

Ngoài ra còn có kiểu dữ liệu đã được xây dựng sẵn hoặc do người dùng định nghĩa, được gọi là Object Datatype. Gồm objects, arrays, dates, maps, sets, intarrays, floatarrays, promises, ...

**Ví dụ:**

* Kiểu số
  + let length = 16;
  + let weight = 7.5;
* Kiểu chuỗi: let color = “Red”;
* Kiểu boolean:
  + let x = true;
  + let y = false;
* Kiểu object:
  + const person = {firstName: “Long”, lastName: “Le”};
* Kiểu array:
  + const cars = [“Toyota”, “BMW”, “Volvo”]
* Kiểu date:
  + const date = new Date(“2024-07-24”)

## Ý tưởng

Trong lập trình, định nghĩa kiểu dữ liệu là rất quan trọng. Để có thể xử lý tính toán trên biến, thì thật sự cần thiết phải biết kiểu dữ liệu của biến là gì. Nếu không có kiểu dữ liệu thì trong một số trường hợp thì máy tính có thể giải quyết được nhưng có thể sẽ không theo mong muốn của người sử dụng

**Ví dụ:**

let x = 1 + “Hello”

Ta có thể thấy biến x là một phép cộng giữa một kiểu dữ liệu số và một kiểu dữ liệu chuỗi. Về nguyên lý thì chỉ có phép cộng giữa 2 chuỗi hoặc 2 số. Tuy nhiên máy tính sẽ không báo lỗi và thay vào đó, thay đổi kiểu dữ liệu số về dạng chuỗi và thực hiện phép cộng 2 chuỗi bình thường. Máy tính sẽ cho kết quả là “1Hello”.

let x = 1 + 1 + “Hello”

Javascript thực hiện tính toán từ trái sang phải. Vì vậy nó thực hiện phép cộng 2 số và sau đó chuyển số về chuỗi và thực hiện phép cộng hai chuỗi lại với nhau, Kết quả là “2Hello”

# Type conversions

Có 6 cách biến đổi kiểu dữ liệu trong Javascript:

* Strings 🡪 Numbers
* Numbers 🡪 Strings
* Dates 🡪 Numbers
* Numbers 🡪 Dates
* Booleans 🡪 Numbers
* Numbers 🡪 Booleans

Các biến trong Javascript có thể được chuyển thành một biến mới và có kiểu dữ liệu khác so với ban đầu. Để có thể thực hiện được điều này thì có thể sử dụng các hàm của Javascript hoặc để cho máy tính tự động thay đổi kiểu dữ liệu trong quá trình tính toán

* **Strings** 🡪 **Numbers:** sử dụng hàm Number() để chuyển kiểu dữ liệu từ chuỗi sang số.
  + Number(“3”) 🡺 3
  + Number(“1.5”) 🡺 1.5
  + Number(“”) 🡺 0
  + Number(“Car”) 🡺 NaN

Ngoài ra còn có các phương thức như parseFloat() và parseInt() dùng để chuyển kiểu chuỗi về dạng số thực và số nguyên.

* **Numbers 🡪 Strings:** sử dụng hàm String() hoặc toString()để chuyển kiểu dữ liệu từ số sang chuỗi.
  + String(123) 🡺 “123”
  + (123).toString() 🡺 “123”

Ngoài ra, còn có các phương thức như: toExponential(), toFixed(), toPrecision()

* **Dates 🡪 Numbers:** sử dụng phương thức getTime() hoặc Number() cũng có thể chuyển kiểu ngày/tháng/năm về dạng số
  + d = new Date;

Number(d)

d.getTime()

* **Dates 🡪 Strings:** sử dụng phương thứcString() hoặc toString() để chuyển kiểu ngày/tháng/năm về dạng chuỗi
  + String(Date())
  + Date().toString()
* **Booleans 🡪 Numbers:** sử dụng phương thức Number() để chuyển kiểu Boolean về dạng số
  + Number(false) 🡺 0
  + Number(true) 🡺 1
* **Booleans 🡪 Strings:** sử dụng phương thức String() để chuyển kiểu Boolean về dạng chuỗi
  + false.toStrign() 🡺 false
  + true.toString() 🡺 true
* **Automatic Type Conversion** (Tự động chuyển kiểu dữ liệu): Khi thực hiện tính toán với các biến không cùng kiểu dữ liệu với nhau. Thì Javascript sẽ cố gắng chuyển về đúng kiểu dữ liệu để có thể thực hiện việc tính toán
  + 5 + null 🡺 5
  + “5” + null 🡺 “5null”
  + “5” + 2 🡺 “52”
  + “5” – 2 🡺 3
  + “5” \* “2” 🡺 10

# Operators

Gồm các kiểu:

* Addition (+)
* Subtraction (-)
* Multiplication (\*)
* Exponential (\*\*)
* Division (/)
* Division Remainder (%)
* Increment (++)
* Decrement (--)
* Logical and (&&)
* Logical or (||)
* Logical not (!)

Ngoài ra, khi thực hiện tính toán giữa các biến. Có thể viết các toán tử một cách ngắn gọn hơn

**Ví dụ:**

x += y ⬄ x = x + y

x -= y ⬄ x = x - y

x \*= y ⬄ x = x \* y

x /= y ⬄ x = x / y

x %= y ⬄ x = x % y

x \*\*= y ⬄ x = x \*\* y

# Comparisions

* Bằng (==)
* Bằng giá trị và kiểu dữ liệu (===)
* Không bằng (!=)
* Không bằng giá trị hoặc kiểu dữ liệu (!==)
* Lớn hơn (>)
* Bé hơn (<)
* Lớn hơn hoặc bằng (>=)
* Bé hơn hoặc bằng (<=)
* Toán tử ba ngôi (?)

# Conditional branching

Gồm các từ khóa như:

* ( if ): dùng để chỉ một khối lệnh bên trong được thực thi nếu điều kiện được chỉ định là đúng
* ( else ): dùng để chỉ một khối lệnh được thực thi nếu điều kiện bị sai
* ( else if ): dùng để chỉ một khối lệnh được thực thi dựa trên một điều kiện mới nếu điều kiện trước đó bị sai
* ( switch ): dùng để chỉ nhiều khối lệnh được thi dựa trên điều kiện được xác định của mỗi khối
  + Nó sẽ xét điều kiện từ trên xuống dưới
  + Giá trị của biến sẽ được so sánh với từng giá trị của điều kiện
  + Nếu khớp điều kiện thì khối lệnh bên trong sẽ được thực thi

**Ví dụ:**

if (điều kiện 1){

Khối lệnh được thực thi nếu điều kiện đúng

}else if (điều kiện 2){

Khối lệnh được thực thi nếu điều kiện 1 sai và điều kiện 2 đúng

}else{

Khối lệnh được thi nếu cả điều kiện 1 và điều kiện 2 sai

}

if (time < 10){

console.log(“Good morning”)

}else if (time < 20){

console.log(“Good day”)

}else{

console.log(“Good evening”)

}

switch (expression){

case x:

Khối lệnh được thực thi

break;

case y:

Khối lệnh được thực thi

break;

default:

Khối lệnh được thực thi

}

switch (new Date().getDay()) {  
   case 0:  
     day = "Sunday";  
     break;  
   case 1:  
    day = "Monday";  
     break;  
   case 2:  
     day = "Tuesday";  
     break;  
   case 3:  
    day = "Wednesday";  
     break;  
   case 4:  
    day = "Thursday";  
     break;  
   case 5:  
    day = "Friday";  
     break;  
   default:  
     day = "Saturday";  
}

# Logical operators

Được sử dụng để xác định logic giữa các giá trị hoặc các biến. Có 3 loại

* Toán tử và (&&)
* Toán tử hoặc (||)
* Toán tử phủ định (!)

Giả sử ta có hai biến cùng với giá trị là : x = 1 và y = 2

( x < 5 && y < 1) 🡺 true

(x >= 3 || y < 1) 🡺 false

!(x ==y) 🡺 true

# Switch

Câu lệnh switch được sử dụng để chọn một hoặc nhiều khối lệnh được thực thi

Cú pháp:

Switch(biểu thức){

case x:

// code

break;

case y:

// code

break;

default:

// code

}

Biểu thức trong câu lệnh switch sẽ được xem xét. Giá trị trong biểu thức được so sánh với giá trị trong mỗi trường hợp. Nếu có sự trùng nhau về giá trị thì khối lệnh bên trong sẽ được thực thi. Ngược lại, thì khối lệnh bên trong default sẽ được thực hiện.

Ngoại trừ trường hợp của default, thì từ khóa break được sử dụng ở cuối mỗi khối lệnh. Điều này có ý nghĩa là dừng thực thi câu lệnh switch. Nếu không có câu lệnh break thì giá trị trong biểu thức sẽ được so sánh với các giá trị tiếp đó và thực thi khối lệnh nếu hợp và cho đến khi gặp từ khóa break hoặc là thực thi khối lệnh cuối cùng ( bên trong default) thì câu lệnh switch sẽ được dừng lại.

Khối lệnh trong default sẽ được thực thi khi mà các giá trị so sánh ở trên đều không khớp với nhau. Default không phải là trường hợp cuối cùng được so sánh trong câu lệnh switch. Có thể để ở trên cùng hoặc ở giữa câu lệnh, tuy nhiên phải kết thúc khối lệnh bên trong bằng từ khóa break

# Nullish coalescing operator

Nullish (đại diện cho giá trị null hoặc undefined).

* Null có nghĩa là giá trị rỗng hoặc giá trị không tồn tại, nó có thể được sử dụng để gán cho một biến như là một đại diện không có giá trị
* Undefined có nghĩa là không xác định. Khi khai báo một biến mà chưa gán giá trị thì giá trị của biến đó sẽ là undefined

Nullish coalesing operator được ký hiệu dưới dạng ( ?? ). Trong một biểu thức có sử dụng toán tử này thì nó sẽ trả về tham số đầu tiên nếu giá trị của biến đang xét không phải là kiểu nullish, ngược lại thì sẽ trả về tham số thứ 2.

Ví dụ:

let name = null;

let text = “missing”

let result = name ?? text 🡺 result = “missing”

# Loops

Trong Javascript, loops có nghĩa là các vòng lặp. Có 2 loại vòng lặp chính: For và While. Mỗi loại sẽ có công dụng và cách thức sử dụng khác nhau

Đối với vòng lặp For (gồm có For, For in For of), While (gồm có While, Do while)

## For loops

* Vòng lặp For (Tạo ra một vòng lặp với 3 biểu thức tùy chỉnh)

for (biểu thức 1; biểu thức 2; biểu thức 3){

Khối lệnh bên trong

}

Biểu thức 1 được thực thi trước khi khối lệnh bên trong được thực thi

Biểu thức 2 xác định điều kiện thực thi của khối lệnh bên trong

Biểu thức 3 được thực thi sau khi khối lệnh bên trong đã được thực thi

Ví dụ:

for (let i = 0; i < 5; i++){

text += “The number is ” + i + “<br/>”

}

* Vòng lặp For in (là vòng lặp sẽ đi qua các thuộc tính có trong một biến thuộc kiểu dữ liệu Object

Ví dụ:

for (key in object) {

// code

}

const numbers = [ 45, 4, 9, 16, 25];

let txt = “”;

for (let x in numbers) {

txt += numbers[x]

}

Ngoài ra, vòng lặp For in được sử dụng để đi qua các giá trị của một biến thuộc kiểu dữ liệu Array

Ví dụ:

For (variable in array){

//code

}

Đối với một mảng mà chúng ta cần có thông tin của các index thì không nên sử dụng vòng lặp for in. Trong trường hợp này, sử dụng vòng lặp for, for of hoặc Array.forEach() sẽ tốt hơn

Array.forEach() là phương thức được gọi để duyệt qua tất các phần tử có trong mảng

Ví dụ:

const numbers = [45, 5, 6]

let txt = “”

numbers.forEach((value, index) => {

txt += value ;

})

* Vòng lặp For of

Ví dụ:

For (variable of iterable){

//code

}

const cars = [“BMW” , “Volvo”, “Mini”]

let txt = “”;

for (let x of cars){

txt += x

}

## While loops

* while

Vòng lặp while sẽ đi qua khối lệnh khi mà điều kiện được chỉ định là true

Ví dụ:

while (điều kiện chỉ định){

//code

}

while (i < 10){

text += “The number is ” + i;

i++;

}

* do while

Là một biến thể của vòng lặp while. Vòng lặp này sẽ thực thi khối lệnh bên trong một lần trước khi kiểm tra điều kiện chỉ đinh. Và nó sẽ tiếp tục thực thi vòng lặp khi điều kiện chỉ định là true

Ví dụ:

do{

//code

}

while (điều kiện chỉ định)

do{

text += “The number is ” + i;

i++;

} while (i < 10)

# Functions

* Là một khối lệnh được thiết kế để giải quyết một tác vụ cụ thể nào đó. Nó sẽ được thực thi khi có lệnh gọi. Function được định nghĩa với một từ khóa là function, theo sau nó sẽ là tên gọi cùng với dấu () cùng với tham số đầu vào bên trong, tên function có thể chứa các kí tự, số, dấu gạch dưới, dấu $ ( tương tự như cách đặt tên biến).
* Một function có thể chứa nhiều tham số đầu vào và sẽ được ngắn cách bởi dấu ‘,’. Khối lệnh của function sẽ được nằm bên trong dấu {}. Các tham số truyền vào chính là các giá trị mà function sẽ lấy để thực hiện các tác vụ tính toán và trả về kết quả.
* Khối lệnh bên trong function sẽ được thực thi khi có lệnh gọi. Kết quả thực thi của function sẽ được trả về thông qua từ khóa return ở cuối khối lệnh.
* Khi viết định nghĩa một function để giải quyết một tác vụ cụ thể. Chúng ta có thể sử dụng function đã được định nghĩa này để giải quyết các tác vụ tương tự. Điều đó cho thấy, một function có thể được gọi và tái sử dụng nhiều lần, giúp việc viết lệnh trở nên ngắn gọn và hiệu quả hơn.

Ví dụ:

function myFunction (x, y) {

return x + y

}

let a = myFunction(1,1) 🡺 a =2

# Asynchronous, Promise

Asynchronous có nghĩa là bất đồng bộ. Khi một hàm được thực thi song song với những hàm khác thì nó được gọi là bất đồng bộ (hàm setTimeout() trong Javascript là một hàm bất đồng bộ được xây dựng sẵn trong Javascript)

Từ khóa Async, Await dùng để trả về một Promise và chờ đời một Promise. Và await chỉ có thể được sử dụng bên trong một hàm có định nghĩa async trước đó. Await dùng để làm cho hàm tạm dừng thực thi và chờ đợi kết quả từ Promise trước khi tiếp tục.

Ví dụ:

async function myDisplay() {  
  let myPromise = new Promise(function(resolve, reject) {  
    resolve("I love You !!");  
  });  
  document.getElementById("demo").innerHTML = await myPromise;  
}  
  
myDisplay();

Promise là một Object chứa Producing code và Consuming code. Nó đại diện cho các tác vụ bất đồng bộ và được sử dụng kết hợp với async và await.

Producing code là lệnh thỉnh thoảng có thể thực thi

Consuming code là lệnh phải chờ đợi để thực thi khi có kết quả

Khi Producing code nhận kết quả, nó sẽ gọi 1 trong 2 callbacks:

* Khi thành công nó sẽ gọi hàm Resolve
* Khi thất bại nó sẽ gọi hàm Reject

Ví dụ:  
 let myPromise = new Promise(function(myResolve, myReject) {

myResolve(); // Khi thành công

myReject(); // Khi thất bại

})

myPromise.then(

function(value){

// Thực thi lệnh nếu thành công

},

function(error){

// Thực thi lệnh nếu thất bại

}

)

Object promise gồm 3 loại:

* Pending
* Fulfiled
* Rejected

Gồm 2 thuộc tính: state và result

Pending có state là “pending” 🡪 Giá trị của result là undefined

Fulfilled có state là “fulfilled” 🡪 Result có giá trị

Rejected có state là “rejected” 🡪 Result là một error object

Chú ý: Không thể truy cập thuộc tính state và result của Promise mà sử dụng một phương thức của Promise để xử lý

# Handling errors

Khi thực thi mã lệnh, các lỗi có thể xảy ra. Vì vậy, cần phải có một điều gì đó để có thể bắt các lỗi này để chương trình có thể chạy hiệu quả hơn. Trong Javascript, sử dụng lệnh try catch để có thể bắt các lỗi có thể xảy ra

Ví dụ:

try{

// Lệnh thực thi

}catch(err){

// Lỗi sẽ được bắt tại đây

}

Khi xảy ra lỗi, Javascript sẽ dừng thực thi lệnh và thông báo lỗi. Về mặt kỹ thuật, hành động này được gọi là throw an errror. Nó sẽ tạo ra một Error object với 2 thuộc tính: name và message

* Throw Statement

Câu lệnh throw cho phép tạo một thông báo lỗi dưới dạng tùy chỉnh. Và khi sử dụng throw cùng với câu lệnh try catch, có thể điều khiển được luồng hoạt động của chương trình và thông báo lỗi nếu có ( nội dung thông báo đã được tùy chỉnh thông qua throw)

* Finally Statement

Cho phép thực thi lệnh bất kể kết quả từ lệnh try catch trước đó

Ví dụ:

try {  
  // Khối lệnh thực thi thử}  
catch(err) {  
  // Khối lệnh để bắt lỗi}  
finally {

 Khối lệnh được thực thi bất kể lệnh try catch trước đó}

* Error Object

Javascript có các built in error object dùng để cung cấp các thông tin lỗi khi xảy ra. Gồm 2 thuộc tính: name và message

* RangeError: lỗi được thông báo khi một số có giá trị nằm ngoài dãy giá trị cho phép
* ReferenceError: lỗi được thông báo khi có biến không được khai báo
* SyntaxError: lỗi được thông báo khi thực thi lệnh có cú pháp sai
* TypeError: lỗi được thông báo khi có kiểu dữ liệu không khớp với kiểu đã được định nghĩa trước đó
* URIError: lỗi được thông báo khi sử dụng các kí tự không hợp lệ trong Uniform Resource Identifier Function

# GIT

## Khái niệm

Git là một hệ thống quản lý phiên bản phân tán **(Distributed Version Control System*– DVCS*)**, nó là một trong những hệ thống quản lý phiên bản phân tán phổ biến nhất hiện nay. **Git**cung cấp cho mỗi lập trình viên kho lưu trữ (**repository**) riêng chứa toàn bộ lịch sử thay đổi.

VCS nghĩa là hệ thống giúp lập trình viên có thể lưu trữ nhiều phiên bản khác nhau của một mã nguồn được nhân bản (**clone**) từ một kho chứa mã nguồn (**repository**), mỗi thay đổi vào mã nguồn trên local sẽ có thể ủy thác (**commit**) rồi đưa lên server nơi đặt kho chứa chính.

Và một máy tính khác nếu họ có quyền truy cập cũng có thể clone lại mã nguồn từ kho chứa hoặc clone lại một tập hợp các thay đổi mới nhất trên máy tính kia.

Lập trình viên có thể xem lại danh sách các sự thay đổi của file như xem một dòng thời gian của các phiên bản. Mỗi phiên bản bao gồm: nội dung file bị thay đổi, ngày giờ sửa đổi, người thay đổi là ai, lý do thay đổi hay tên phiên bản…

## Lợi ích

Các dự án thực tế thường có nhiều lập trình viên làm việc song song. Vì vậy, một hệ thống kiểm soát phiên bản như Git là cần thiết để đảm bảo không có xung đột code giữa các lập trình viên.

Ngoài ra, các yêu cầu trong các dự án như vậy thay đổi thường xuyên. Vì vậy, một hệ thống kiểm soát phiên bản cho phép các nhà phát triển revert và quay lại phiên bản cũ hơn của code.

Cuối cùng, đôi khi một số dự án đang được chạy song song liên quan đến cùng một cơ sở code. Trong trường hợp như vậy, khái niệm phân nhánh trong Git là rất quan trọng.

## Thuật ngữ

### Branching

Các Branch (nhánh) đại diện cho các phiên bản cụ thể của một kho lưu trữ tách ra từ project chính của bạn. Branch cho phép bạn theo dõi các thay đổi thử nghiệm bạn thực hiện đối với kho lưu trữ và có thể hoàn nguyên về các phiên bản cũ hơn.

Sử dụng lệnh git checkout để chuyển giữa các branch. Chỉ cần nhập git checkout theo sau là tên của branch bạn muốn chuyển đến hoặc nhập git checkout master để trở về branch chính (master branch).

### Git logs

Dùng để hiển thị lịch sử của các lần commit để có thể theo dõi những sự thay đổi đối với kho lưu trữ chính.

### Git add

  Dùng để thêm thay đổi đến stage/index trong thư mục làm việc.

### Git commit

Commit nghĩa là một action để Git lưu lại một snapshot của các sự thay đổi trong thư mục làm việc. Và các tập tin, thư mục được thay đổi đã phải nằm trong Staging Area. Mỗi lần commit nó sẽ được lưu lại lịch sử chỉnh sửa của code kèm theo tên và địa chỉ email của người commit. Ngoài ra trong Git bạn cũng có thể khôi phục lại tập tin trong lịch sử commit của nó để chia cho một branch khác, vì vậy bạn sẽ dễ dàng khôi phục lại các thay đổi trước đó.

***git commit -m ”Đây là message, bạn dùng để note những thay đổi để sau này dễ dò lại”***

### Git push, pull

Push hoặc Pull các thay đổi đến remote. Nếu bạn đã added và committed các thay đổi và bạn muốn đẩy nó lên hoặc remote của bạn đã update và bạn apply tất cả thay đổi đó trên code của mình.

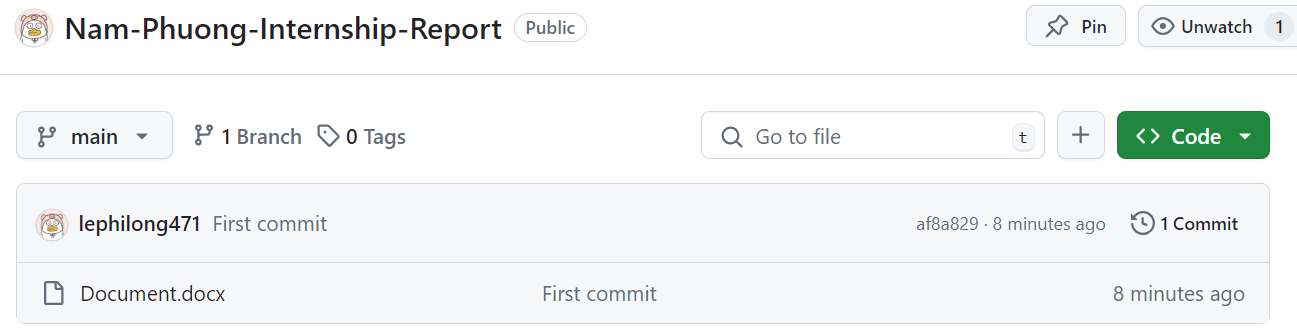
Cách dùng: git pull <:remote:> <:branch:> and git push <:remote:> <:branch:>

### Git fetch

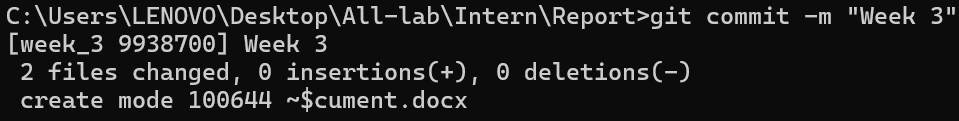
Lệnh git fetch tìm nạp các bản sao và tải xuống tất cả các tệp branch vào máy tính của bạn. Sử dụng nó để lưu các thay đổi mới nhất vào kho lưu trữ của bạn. Nó có thể tìm nạp nhiều branch cùng một lúc.

## Thực hành

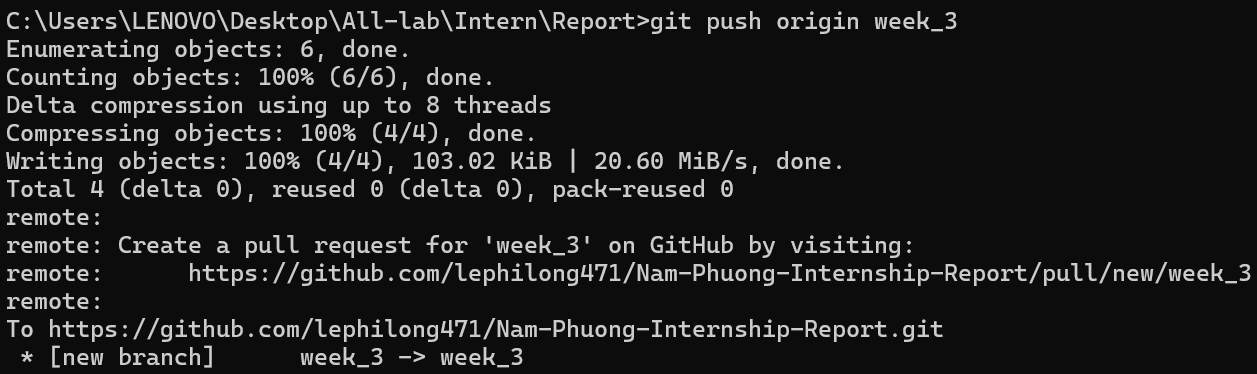
Tạo remote repository, commit và push lên remote repository



Checkout qua một nhánh khác và thực hiện commit các thay đổi lên remote repository



Sử dụng git push để lưu các thay đổi trên remote repository



# React

## Khái niệm

**React** là một thư viện JavaScript được phát triển bởi Facebook (hiện giờ là Meta), dùng để xây dựng giao diện người dùng cho các ứng dụng web và di động. Nó cho phép các nhà phát triển xây dựng các giao diện động và tương tác bằng cách chia nhỏ giao diện thành các thành phần (components) tái sử dụng được. Đây là một số điểm chính về React:

* Component-Based Architecture: React xây dựng ứng dụng bằng cách chia giao diện thành các thành phần nhỏ gọi là "components". Mỗi component có thể chứa cấu trúc HTML, CSS, và logic JavaScript riêng của nó.
* Declarative: React sử dụng phương pháp lập trình khai báo (declarative programming), điều này có nghĩa là bạn mô tả giao diện người dùng theo cách bạn muốn, và React tự động cập nhật giao diện khi dữ liệu thay đổi.
* Virtual DOM: React sử dụng Virtual DOM, một bản sao nhẹ của DOM thực tế, để cải thiện hiệu suất. Khi có sự thay đổi trong dữ liệu, React cập nhật Virtual DOM trước, sau đó so sánh với DOM thực tế và chỉ cập nhật những phần cần thiết.
* Unidirectional Data Flow: Dữ liệu trong React chảy theo một chiều từ cha (parent) xuống con (child) thông qua props (properties). Điều này giúp quản lý trạng thái và luồng dữ liệu dễ dàng hơn.
* Hooks: Từ phiên bản React 16.8, Hooks được giới thiệu để quản lý trạng thái và các side effects mà không cần viết các lớp component.

## JSX

**JSX** (JavaScript XML) là một cú pháp mở rộng cho JavaScript, được React sử dụng để mô tả cấu trúc của giao diện người dùng. JSX cho phép viết mã HTML giống như cú pháp trong JavaScript. Đây là một số điểm quan trọng về JSX:

* Cú Pháp Gần Gũi Với HTML: JSX cho phép viết các cấu trúc giao diện người dùng giống như HTML, nhưng thực tế chúng sẽ được biên dịch thành các đối tượng JavaScript.
* Ví dụ: const element = <h1>Hello, world!</h1>;
* Tích Hợp JavaScript: JSX cho phép bạn tích hợp các biểu thức JavaScript vào trong cấu trúc HTML bằng cách sử dụng dấu ngoặc nhọn {}.
* Ví dụ: const name = 'John';

const element = <h1>Hello, {name}!</h1>;

* Tạo Component: JSX thường được sử dụng để tạo ra các component trong React. Component có thể được viết dưới dạng hàm hoặc lớp và có thể trả về JSX để mô tả giao diện.
* Tính Tương Thích: JSX không phải là một phần của JavaScript tiêu chuẩn, vì vậy cần có một công cụ biên dịch để chuyển đổi nó thành JavaScript có thể thực thi trên trình duyệt.

## Components

Làcác khối xây dựng cơ bản của giao diện người dùng. Chúng giúp chia nhỏ giao diện thành các phần nhỏ hơn, có thể tái sử dụng, dễ quản lý và bảo trì. Dưới đây là cái nhìn tổng quan về các loại components và cách sử dụng chúng:

## Props, State

Props là cách để truyền dữ liệu từ component cha xuống component con. Chúng là tham số đầu vào của component và là immutable (không thể thay đổi bởi component nhận).

State là cách để quản lý dữ liệu thay đổi trong một component. Nó có thể được sử thông qua hook useState().

## Lifecycle

Trong React, lifecycle methods (phương thức vòng đời) là các hàm đặc biệt được gọi tự động tại các thời điểm khác nhau trong vòng đời của một component, giúp thực hiện các thao tác nhất định khi component được gắn vào DOM, cập nhật, hoặc bị gỡ bỏ. Nó được quản lý bằng cách sử dụng Hooks, đặc biệt là useEffect

## Portals

Trong React, portal là một tính năng cho phép render một component vào một DOM node nằm ngoài cấu trúc DOM hiện tại của component cha. Điều này rất hữu ích khi cần hiển thị một phần tử như modal, tooltip, hoặc menu dropdown mà không muốn bị ảnh hưởng bởi các quy tắc CSS của phần tử cha.

**Lợi ích**

* Tách biệt CSS: Modal hoặc tooltip có thể được định dạng độc lập mà không bị ảnh hưởng bởi các thuộc tính CSS của phần tử cha.
* Quản lý z-index: Bằng cách render vào body hoặc một container cụ thể, có thể dễ dàng kiểm soát các vấn đề về z-index.
* Tránh lỗi về overflow: Khi hiển thị các phần tử như modal, có thể tránh các vấn đề về overflow của phần tử cha.

## Error Boundaries

Là một tính năng của React giúp xử lý lỗi trong các component của ứng dụng. Chúng cung cấp một cách để "bắt" lỗi JavaScript trong cây component, giúp ứng dụng không bị sập hoàn toàn khi gặp lỗi, và cho phép hiển thị một giao diện lỗi thay vì để ứng dụng bị hỏng.

**Khi nào nên sử dụng**

* Khi bạn có các phần của ứng dụng mà bạn không muốn lỗi làm ảnh hưởng đến toàn bộ ứng dụng: Ví dụ, bạn có thể sử dụng Error Boundaries để bao quanh các component như form, bảng điều khiển hoặc các phần tử tương tác người dùng.
* Khi bạn cần hiển thị giao diện lỗi tùy chỉnh: Giúp cải thiện trải nghiệm người dùng khi có sự cố xảy ra.
* Khi bạn muốn ghi log lỗi hoặc gửi báo cáo lỗi: Để giúp việc gỡ lỗi và duy trì ứng dụng dễ dàng hơn.

## HOCs

**Higher-Order Components (HOCs)** là một pattern trong React để tái sử dụng logic giữa các component. HOCs là các hàm nhận vào một component và trả về một component mới với các chức năng hoặc props bổ sung. Điều này cho phép bạn chia sẻ logic giữa các component mà không cần phải lặp lại mã nguồn.

**Lợi ích**

* Tái sử dụng Logic: HOCs giúp bạn chia sẻ và tái sử dụng logic giữa các component mà không cần phải viết lại mã nguồn.
* Kéo dài Component: Bạn có thể mở rộng chức năng của một component mà không cần phải thay đổi mã nguồn của component gốc.

**Các trường hợp nên sử dụng**

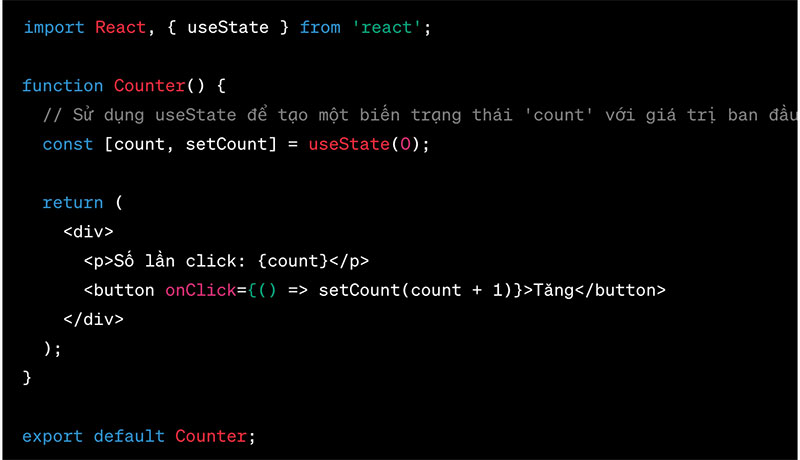
* Khi bạn cần chia sẻ logic giữa nhiều component
* Khi bạn muốn tách biệt logic ra khỏi giao diện: Để giữ cho các component đơn giản và dễ bảo trì.

## Hooks trong React

Hooks là **một tập hợp các hàm có sẵn trong thư viện React** giúp bạn sử dụng trạng thái (state) và các tính năng khác của React trong các thành phần không phải là lớp (functional components). Những hàm này giúp bạn viết code dễ đọc hơn, tái sử dụng dễ dàng hơn và giảm thiểu sự phức tạp trong việc quản lý trạng thái của ứng dụng.

### useState

* Cho phép bạn thêm React state vào function components.
* useState() khai báo một “state variable”
* Tham số duy nhất được truyền vào hook useState() là state ban đầu. Không giống như class, state không nhất thiết phải là object mà có thể là số hoặc chuỗi.
* useState() trả về một cặp giá trị dưới dạng mảng: state hiện tại và một hàm để update nó.



### useEffect

Là 1 hook mạnh mẽ cho phép xử lý các tác vụ như là: gọi API, quản lý side effects và theo dõi vòng đời của functional components.

**useEffect() là sự kết hợp của 3 phương thức lifecycle:**

* componentDidMount()
* componentDidUpdate()
* componentWillUnmount()

Các trường hợp sử dụng

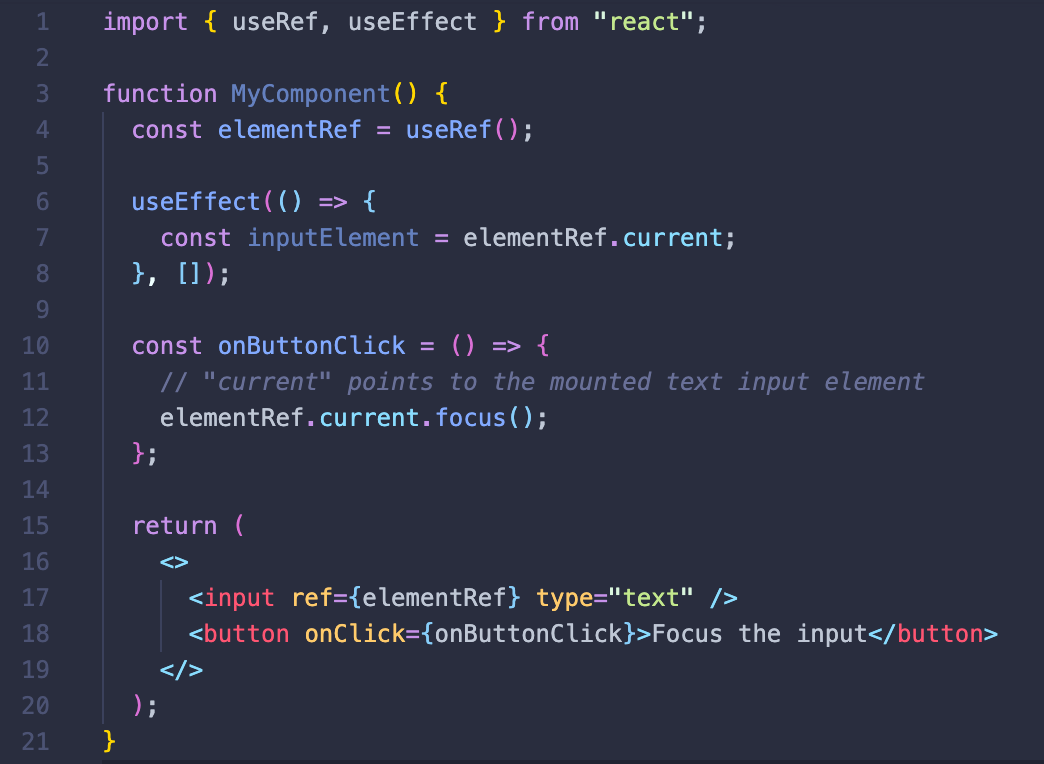
* useEffect(callback): hàm callback được gọi mỗi khi thành phần được tạo ra và cập nhật
* useEffect(callback, []): hàm callback được gọi khi thành phần được tạo ra
* useEffect(callback, [dependencies]): hàm callback được gọi sau khi thành phần được tạo ra và giá trị trong dependencies thay đổi

### useRef

React Ref (React reference) hiểu đơn giản là một đối tượng tham chiếu đến một biến, một component giữ cho giá trị của nó không thay đổi giữa các lần render và truy xuất các giá trị đó thông qua key current.

Có 2 lý do chính mà chúng ta sẽ sử dụng useRef: Truy cập DOM nodes hoặc React elements và lưu giữ một biến có thể mutate mà không làm re-render component

Ví dụ:



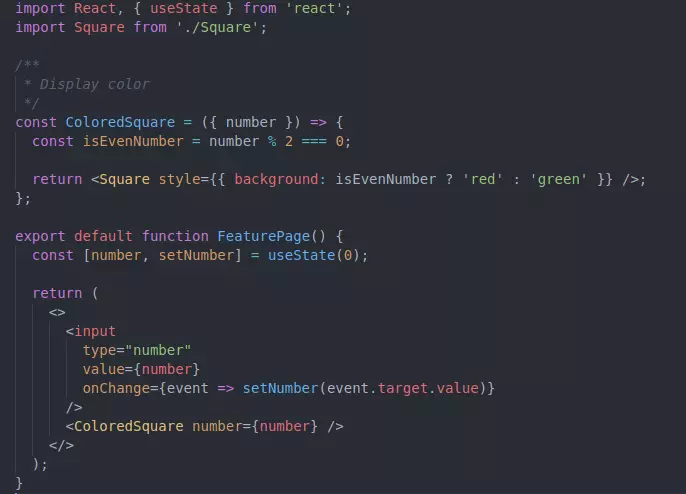
### Memo

 Khi React.memo() bao quanh một component, React sẽ ghi nhớ kết quả render và bỏ qua các quá trình render không cần thiết, nhằm tối ưu hóa việc hiệu năng của quá trình render các functional component.

Các trường hợp nên dùng Memo:

* Trước hết thì component phải là functional component.
* Component thường xuyên bị re-render.
* Nếu component luôn luôn bị re-render mặc dù prop không thay đổi.
* Component của chứa một lượng lớn tính toán logic và UI như Chart, Canvas, 3D library….

Ví dụ:



Component ColoredSquare sẽ luôn được render lại mỗi khi người dùng đổi giá trị ở input, bất kể nó là giá trị gì. Để nâng cao hiệu năng của ứng dụng thì ta cần xử lý chỉ cho component render lại nếu giá trị đổi từ số chẵn sang lẻ hoặc lẻ sang chẵn. Ta có thể sử dụng React.memo như sau:



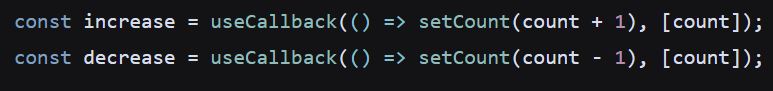
Đối số thứ hai truyền vào React.memo sẽ là hàm dùng để React xác định có render lại hay không dựa trên sự thay đổi của props. Nếu giá trị trả về của hàm là true thì có nghĩa là React sẽ không render lại component này mà dùng lại kết quả đã render trước đó và ngược lại nếu trả về giá trị false thì React sẽ render lại component này.

### useCallback

Khái niệm: Là một react hooks giúp tạo ra một memoized callback và chỉ tạo ra callback mới khi dependencies thay đổi.

* Nhận vào 2 tham số: 1 là function, 2 là dependencies.
* Trả về memoized callback.
* Chỉ tạo ra function mới khi dependencies thay đổi.
* Nếu dùng empty dependencies thì không bao giờ tạo ra function mới.

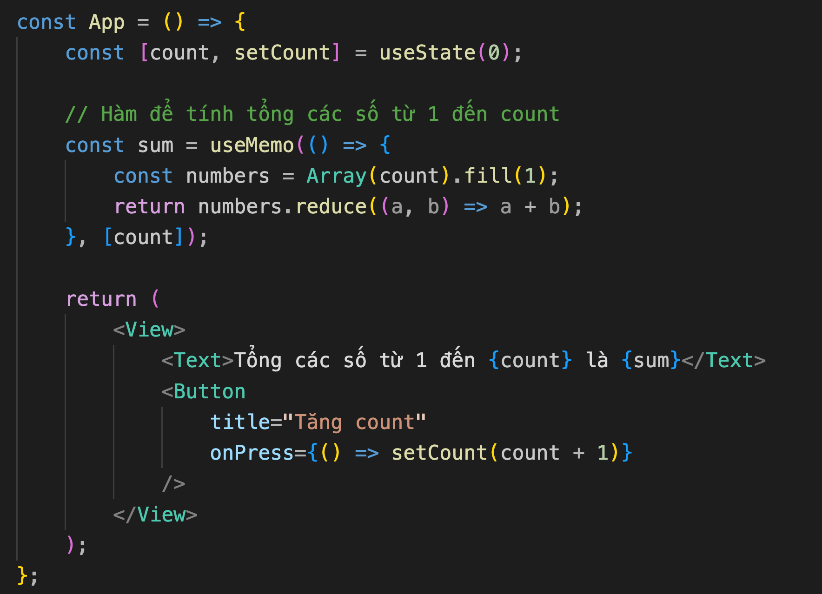
Ví dụ:



### useMemo

Khái niệm: Là một react hooks giúp tạo ra một memoized value và chỉ tính toán ra value mới khi dependencies thay đổi.

* Nhận vào 2 tham số: 1 là function, 2 là dependencies.
* Trả về memoized value
* Chỉ tính toán value mới khi dependencies thay đổi.
* Nếu dùng empty dependencies thì không bao giờ tính toán lại value mới.



### useContext

**useContext** **là** một hook trong React được sử dụng để truy cập các giá trị của một **Context** API. **Context** API trong React **là** một cách để truyền dữ liệu từ một thành phần cha đến các thành phần con mà không cần truyền props qua nhiều lớp con trung gian.

Nên sử dụng useContext khi:

* Có nhiều state cần quản lý
* Muốn chia sẻ các state với nhau
* Truyền dữ liệu từ thành phần cha đến thành phần con nhưng có nhiều thành phần con trung gian

Ví dụ:



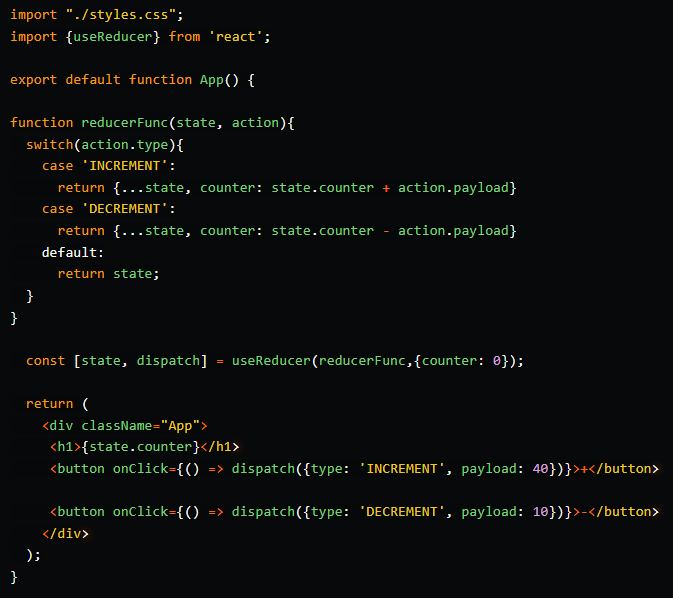
### useReducer

Reducer hiểu một cách đơn giản là một hàm có 2 tham số là state, action và trả về new state sau khi thực hiện một action, hãy tưởng tượng reducer như là một bộ chuyển đổi, nhận input, thực hiện action tác động đến input đó, rồi tạo ra output.

useReducer là một hook dùng để thêm reducer vào trong component. Nó được sử dụng trong trường hợp component có state phức tạp và có nhiều action type làm thay đổi state đó

Nên sử dụng useReducer khi:

* Có sự phức tạp về các state toàn cục
* Có nhiều hàm cập nhật state toàn cục và cần để chung một chỗ để dễ quản lý
* Muốn tách biệt logic từ component



# Typescript

## Khái niệm

TypeScript là một ngôn ngữ lập trình mã nguồn mở được phát triển bởi Microsoft. Nó là một siêu ngôn ngữ của JavaScript, nghĩa là TypeScript mở rộng JavaScript bằng cách thêm các tính năng và kiểu dữ liệu mới, đồng thời vẫn giữ tính tương thích với JavaScript.

***Mục đích sử dụng TypeScript:***

* **Cải thiện khả năng bảo trì mã nguồn:** TypeScript cung cấp hệ thống kiểu tĩnh, giúp phát hiện lỗi ngay trong quá trình biên dịch, điều này giúp tránh các lỗi phổ biến trong mã nguồn.
* **Hỗ trợ phát triển quy mô lớn:** Với tính năng như interface, lớp (class), và module, TypeScript hỗ trợ việc phát triển các ứng dụng quy mô lớn một cách dễ dàng và tổ chức hơn.
* **Hỗ trợ công cụ phát triển:** TypeScript tích hợp tốt với các công cụ phát triển và editor như Visual Studio Code, cung cấp các tính năng như gợi ý mã, kiểm tra lỗi, và refactoring.
* **Tương thích với JavaScript:** TypeScript biên dịch thành JavaScript, giúp mã nguồn TypeScript có thể chạy trên bất kỳ môi trường nào hỗ trợ JavaScript.

***Mối quan hệ giữa TypeScript và JavaScript:***

* **TypeScript là siêu ngôn ngữ của JavaScript:** TypeScript được xây dựng trên nền tảng của JavaScript, nghĩa là mọi mã JavaScript hợp lệ đều là mã TypeScript hợp lệ.
* **Biên dịch thành JavaScript:** Mã nguồn TypeScript cần được biên dịch (transpile) thành JavaScript để có thể chạy trên các trình duyệt hoặc môi trường JavaScript. Quá trình này được thực hiện bởi TypeScript Compiler (tsc).
* **Hỗ trợ tính tương thích:** TypeScript hỗ trợ các tính năng của phiên bản mới nhất của ECMAScript (ES6/ES7) và cung cấp tính năng tương thích với các phiên bản JavaScript cũ hơn. Bạn có thể cấu hình TypeScript để biên dịch mã nguồn thành mã JavaScript tương thích với các phiên bản ECMAScript khác nhau.
* **Mở rộng JavaScript:** TypeScript cung cấp thêm các tính năng như kiểu dữ liệu tĩnh, lớp, và module, giúp mở rộng khả năng của JavaScript và cải thiện khả năng tổ chức mã nguồn.

## Kiểu dữ liệu

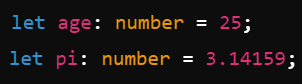
Các kiểu dữ liệu cơ bản gồm:

* Primitive
* Union
* Tuples
* Dynamic
* Literal
* Enums
* Generics

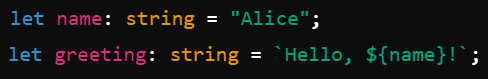
### Primitive

Là những kiểu dữ liệu cơ bản nhất, không thể chia nhỏ hoặc phân tích thêm. Chúng được sử dụng để đại diện cho các giá trị đơn giản và cơ bản. Dưới đây là các kiểu dữ liệu nguyên thủy trong TypeScript:

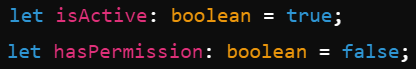
* Number (Đại diện cho các số nguyên, số thực)



* String (Đại diện cho chuỗi văn bản)



* Boolean (Đại diện cho giá trị true hoặc false)



* Symbol (Đại diện cho các giá trị duy nhất và không thay đổi, thường được sử dụng như khóa đối tượng hoặc để tạo ra các giá trị duy nhất)



* Bigint (Đại diện cho các số nguyên lớn, lớn hơn giới hạn của kiểu number)



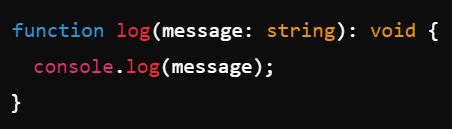
* Null (Được dùng để chỉ ra rằng một biến có thể không có giá trị)



* Undefined (Đại diện cho giá trị không được xác định, thường dùng khi một biến chưa được khởi tạo hoặc không có giá trị)



* Void (Thường dùng để chỉ ra rằng một hàm không trả về giá trị gì)



### Union

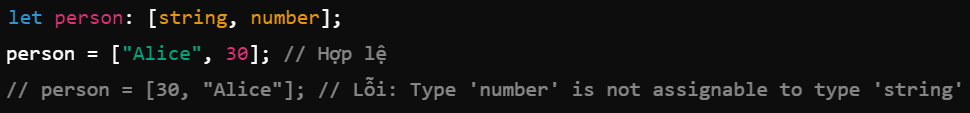
Là kiểu cho phép xác định một biến có thể nhận nhiều loại kiểu dữ liệu khác nhau. Đây là một cách mạnh mẽ để linh hoạt hóa kiểu dữ liệu của biến và giúp quản lý các tình huống mà một giá trị có thể thuộc nhiều kiểu khác nhau.

Để khai báo một kiểu Union, sử dụng dấu “|” để phân tách các dữ liệu khác nhau.



### Tuples

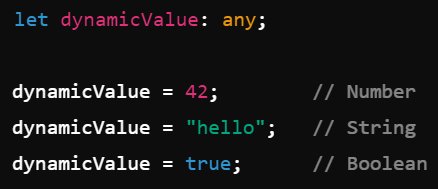
Là một loại cấu trúc dữ liệu đặc biệt cho phéplưu trữ một tập hợp các giá trị với kiểu dữ liệu cụ thể cho từng phần tử, và số lượng phần tử cũng được xác định. Đây là một cách hữu ích để làm việc với các dữ liệu có cấu trúc cố định và hỗn hợp kiểu



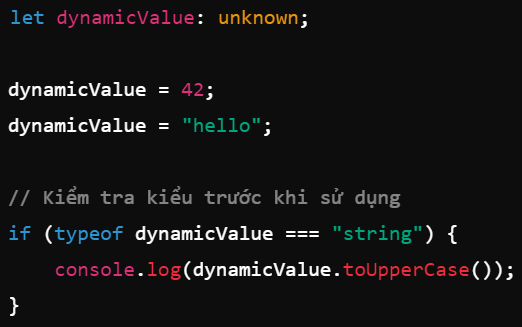
### Dynamics

TypeScript cung cấp một số cách để xử lý và làm việc với dữ liệu động hoặc không xác định kiểu ngay từ đầu.

* Kiểu any (cho phép biến chứa bất kỳ loại dữ liệu nào mà không bị kiểm tra kiểu. Điều này cho phép xử lý dữ liệu động nhưng cũng làm mất đi tính an toàn của kiểu dữ liệu)

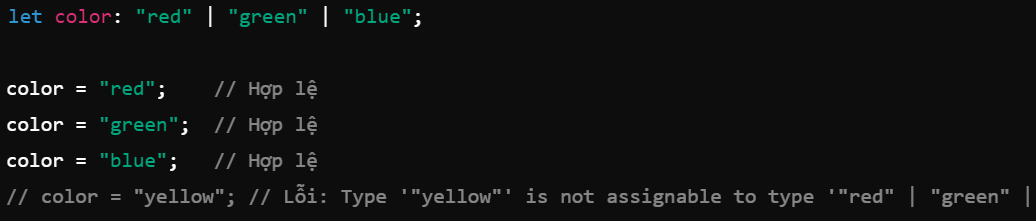


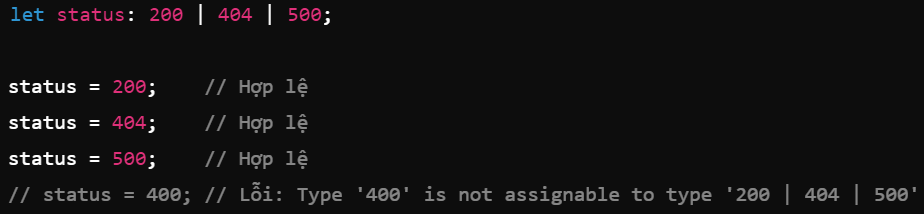
* Kiểu unknown là một kiểu dữ liệu tương tự như any, nhưng an toàn hơn. Với unknown, thì phải cần kiểm tra kiểu dữ liệu trước khi thực hiện các thao tác với giá trị đó.

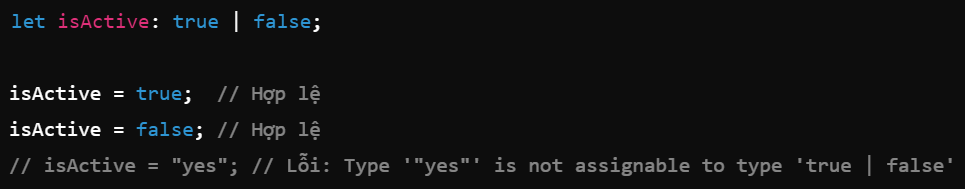


### Literal

Là các kiểu dữ liệu mà giá trị của chúng được xác định chính xác. Đây là cách để chỉ định một tập hợp cụ thể các giá trị cho các biến hoặc các thuộc tính, giúp tăng cường kiểm tra kiểu và bảo mật cho mã nguồn.

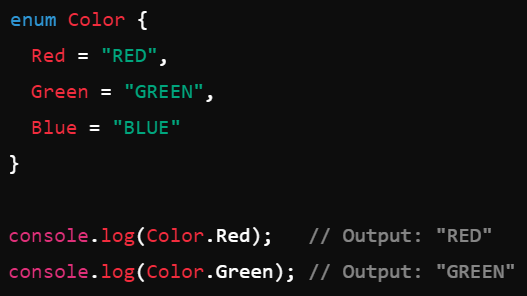






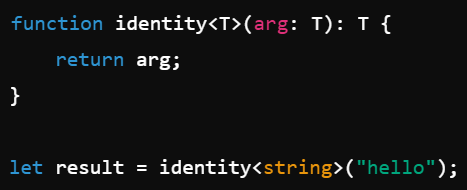
### Enums

Là một cách để định nghĩa một tập hợp các giá trị tên có liên quan. Enum cung cấp cách để tổ chức các giá trị hằng số theo nhóm có thể sử dụng các tên thay vì các giá trị số hoặc chuỗi, giúp mã nguồn trở nên rõ ràng và dễ hiểu hơn. Enum là một tính năng mạnh mẽ của TypeScript cho phép nhóm các hằng số lại với nhau.



### Generics

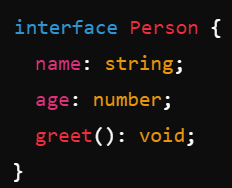
Là một tính năng mạnh mẽ cho phép viết các hàm, lớp, hoặc giao diện có thể hoạt động với nhiều kiểu dữ liệu khác nhau mà không cần phải xác định các kiểu cụ thể trong mã nguồn. Generics giúp tạo ra mã nguồn có thể tái sử dụng và linh hoạt hơn bằng cách cho phép các kiểu dữ liệu được chỉ định khi sử dụng các cấu trúc dữ liệu hoặc hàm.

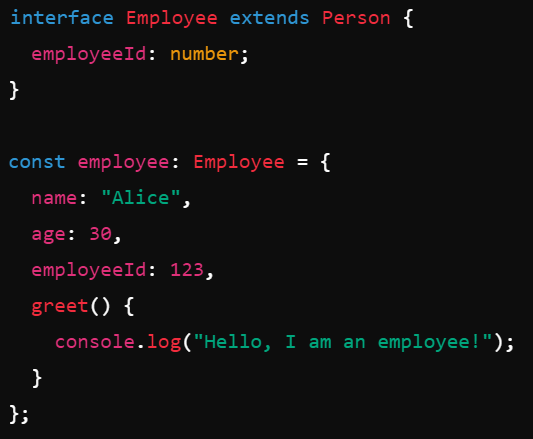


### Type và Interface

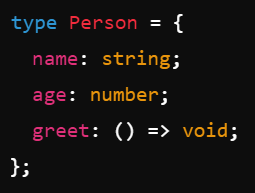
Type và Interface đều được sử dụng để định nghĩa cấu trúc của các đối tượng và kiểu dữ liệu. Tuy nhiên, chúng có một số điểm khác biệt quan trọng về cú pháp và khả năng mở rộng.

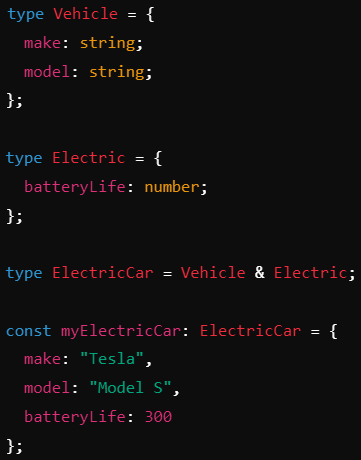
**Interface:** là một cách để định nghĩa cấu trúc của đối tượng, bao gồm các thuộc tính và phương thức mà một đối tượng phải có. Ngoài ra, interface hỗ trợ kế thừa, cho phép tạo ra các interface mới dựa trên các interface trước đó.





**Type:** là một cách khác để định nghĩa các kiểu dữ liệu. Bạn có thể sử dụng type để định nghĩa các kiểu phức tạp hơn, bao gồm các union types, intersection types, và các kiểu hằng số.





**So sánh giữa Interface và Type**

|  |  |  |
| --- | --- | --- |
| **Khả năng** | **Interface** | **Type** |
| Kế thừa và mở rộng | Hỗ trợ kế thừa thông qua extends và cho phép khai báo lại nhiều lần. | Không hỗ trợ kế thừa, nhưng có thể sử dụng intersection types để kết hợp các thuộc tính lại với nhau. |
| Tái sử dụng | Tốt cho việc định nghĩa cấu trúc đối tượng với khả năng mở rộng tốt hơn. | Linh hoạt hơn trong việc kết hợp và định nghĩa các kiểu dữ liệu phức tạp. |
| Khai báo lại | Có thể khai báo lại nhiều lần và TypeScript sẽ gộp tất cả các khai báo. | Không thể khai báo lại, phải định nghĩa một kiểu mới nếu cần thay đổi. |