1. Biến

- Có 3 kiểu khai báo dữ liệu, bao gồm

```
- Using var
- Using let
- Using const
```

- Khai bsao kiểu **var**:
  - Cách khai báo:

```
1   var a = 3
2   console.log(a)

CONSOLE ×

3
```

    - Khai báo kiểu **var** có thể được phép khai báo lại:

```
1   var a = 3
2   var a = 6
3   console.log(a)

CONSOLE ×

6
```

- Khai báo kiểu **Let**:
  - Cách khai báo:

```
1   let a = 3
2   console.log(a)

CONSOLE ×

3
```

  - Kiểu **let** không cho phép khai báo 2 lần:

```
let a = 3
let a =5
console.log(a)
```

```
CONSOLE ×
The symbol "a" has already been declared
```

- Các biến khai báo kiểu **let** phải được khai báo trước khi sử dụng.
-

- Khai báo kiểu **const** :
  - Cách khai báo:

```
const a = 3
console.log(a)
```

```
CONSOLE ×
3
```

  - Biến được khai báo kiểu **const** không thể thay đổi giá trị hoặc khai báo lại:

```
const a = 3
a = 5
console.log(a)
```

```
CONSOLE ×
Cannot assign to "a" because it is a constant
```

★ Lưu ý: Cả kiểu **var, let, const**

- Khi khai báo biến toàn cục:
  - Với trường hợp dưới, nếu biến a trong hàm change() không được khai báo, mà biến a được khai báo kiểu **var** hay **let** bên ngoài hàm, thì biến a được hiểu là biến toàn cục, tức là biến a ở trong và ngoài hàm là 1.

```
var a = 3
function change(){
  a = 5
}
change()
console.log(a)
```

```
CONSOLE ×
5
```

  - Nhưng nếu biến a được khai báo lại trong hàm, thì lúc này biến a trong hàm và biến a ở ngoài hàm là khác nhau.

```
let a = 3
function change(){
  let a = 4
}
change()
console.log(a)
```

```
CONSOLE ×

3
```

- Đối với kiểu **const**
  - Nếu trong hàm biến của const được khai báo lại, thì biến trong và ngoài hàm là khác nhau

```
const a = 3
function test(){
  var a = 4
  console.log(a)
}
test()
```

```
CONSOLE ×

4
```

  - Nếu trong hàm không được khai báo, thì biến trong và ngoài hàm là 1, chỉ có thể sử dụng, không thể thay đổi giá trị

```
const a = 3
function test(){
  console.log(a)
}
test()
```

```
CONSOLE ×

3
```

2. Toán Tử

```
var x = 5
var y = 10
let sum = x + y // toán tử cộng '+'
let subtrac = x - y // toán tử trừ '-'
let multi = x * y // toán tử nhân '*'
let div = x / y // toán tử chia
let Exponent = x ** y // toán tử lũy thừa '**'
let Increment = x++ // toán tử tăng '++'
let Decrement = x-- // toán tử giảm

```

Operator	Example	Same As
=	x = y	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
**=	x **= y	x = x ** y

- Cộng chuỗi với 1 số => chuỗi và ngược lại.

```
var a = "hello " + 5
console.log("a is",typeof(a))
```

```
CONSOLE ×

a is  string
```

```
var x = 10
var y = "20"
var z = x + y
console.log(z)
```

```
CONSOLE ×

1020
```

- Trừ 1 số với một chuỗi số => số và ngược lại

```
var x = 10
var y = "20"
var z = x - y
console.log(z)
```

```
CONSOLE ×

-10
```

- Nhân 1 số với 1 chuỗi số => số và ngược lại

```
1   var x = 10
2   var y = "20"
3   var z = x * y
4   console.log(z)
CONSOLE ×

200
```

- Chia 1 số với 1 chuỗi số => số và ngược lại

```
1   var x = 10
2   var y = "20"
3   var z = x / y
4   console.log(z)
CONSOLE ×

0.5
```

- Chia dư 1 số với 1 chuỗi số => số và ngược lại

```
1   var x = 10
2   var y = "20"
3   var z = x % y
4   console.log(z)
CONSOLE ×

10
```

- Lũy thừa 1 số với 1 chuỗi số => số và ngược lại

```
1   var x = 10
2   var y = "20"
3   var z = x ** y
4   console.log(z)
CONSOLE ×

100000000000000000000
```

★ **Toán tử logic**

- Toán tử so sánh

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

- Toán tử logic

Operator	Description
&&	logical and
||	logical or
!	logical not

3. **Hàm** (chưa cần arrow function)
   - Là một khối mã được tạo ra để thực hiện 1 tác vụ nào đó
   - Cấu trúc 1 hàm cơ bản:

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

   - Hàm có kiểu trả về 1 giá trị

```
let x = myFunction(4, 3);   // Function is called, return value will end up in x

function myFunction(a, b) {
  return a * b;             // Function returns the product of a and b
}
```

- Một hàm có thể là 1 đối tượng để chứa nhiều hàm khác
  -
```
game = function(){
  function init(){}
  function load(){}
  function control(){}
  function update(){}
}
var player = new game()
console.log(typeof(player))
```
```
CONSOLE ×

object
```
-

4. **DataType** (tìm hiểu và làm ví dụ kĩ về array và object)

★ **Object**

- **Là 1 đối tượng, trong đó có nhiều đối tượng khác**
- Ví dụ về cách khai báo:

```
var student = {
    name    : 'hacker',
    age     : 1,
    address : 'earth'
}
console.log(typeof(student))
```
```
CONSOLE ×

object
```

  - đối tượng được bao bọc bởi dấu {}
  - Trong khối {}, là các cặp key và value, mỗi cặp như vậy được ngăn cách nhau bởi dấu ‘,’. Có dạng: **key:value,**

- **Cách gọi thuộc tính trong đối tượng, có 2 cách :**
  - Cách 1:

```
var student = {
    name    : 'hacker',
    age     : 1,
    address : 'earth'
}
console.log(student.name)
```

```
CONSOLE ×

hacker
```

- Cách 2:

```
var student = {
    name    : 'hacker',
    age     : 1,
    address : 'earth'
}
console.log(student['name'])
```

```
CONSOLE ×

hacker
```

- **Phương thức của đối tượng**
  - **Method this**: this không phải là 1 biến, nó là từ khóa để chỉ đối tượng đang bao bọc nó.

```
var student = {
    firstname : 'hacker',
    lastname  : 'lord',
    age       : 1,
    address   : 'earth',
    name      : function(){
        return this.firstname + this.lastname
    }
}
console.log(student.name())
```

```
CONSOLE ×

hackerlord
```

- **Một biến được khai báo đi kèm với từ khóa new là 1 đối tượng**

```
1   game = function(){
2     function init(){}
3     function load(){}
4     function control(){}
5     function update(){}
6   }
7   var player = new game()
8   console.log(typeof(player))
```

CONSOLE ×

```
object
```

★ **Array**

- Mảng là một biến đặc biệt, có thể lưu nhiều hơn 1 giá trị
- Khởi tạo mảng
  -
```
1   var a = []
2   // hoặc
3   var b = new Array()
4
```
- **Thêm các phần tử cho mảng**

Example

```
const cars = [];
cars[0]= "Saab";
cars[1]= "Volvo";
cars[2]= "BMW";
```

  - Hoặc

```
var a = []
a.push(1)
a.push(2)
a.push(3)
a.push('hacker')
a.push('lord')
console.log(a)
```

```
CONSOLE ×

▸ (5) [1, 2, 3, "hacker", "lord"]
```

- Sử dụng **splice()** để thêm phần tử

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");
console.log(fruits)
```

```
CONSOLE ×

▸ (6) ["Banana", "Orange", "Lemon", "Kiwi"...]
```

  - Đối số đầu tiên là ‘2’: là thêm 2 phần tử “lemon” và “Kiwi” vào vị trí số 2 của mảng
  - Đối số thứ hai là ‘0’: là số phần tử của mảng sẽ bị xóa kể từ vị trí số 2 ở trên.
  - Đối số thứ ba là các phần tử cần thêm
  - Thay đổi linh hoạt các đối số để có thể thực hiện đúng mục đích.

- **Lấy chiều dài mảng**

```
var a = [1,2,3,4,5]
console.log(a.length)
```

```
CONSOLE ×

5
```

- **Biến mảng thành 1 chuỗi**
  - Sử dụng method **Join**, sẽ nối các phần tử của mảng lại cùng với ký tự truyền vào hàm join.

```
var a = [5,2,1,4,3]
var b = a.join('*')
console.log(b)
```

```
CONSOLE ×

5*2*1*4*3
```

  - Sử dụng hàm **toString()**, sẽ mặc định đưa về chuỗi ngăn cách các ptu bởi dấu phẩy

```
1   var a = [5,2,1,4,3]
2   var b = a.toString()
3   console.log(b)

CONSOLE ×

5,2,1,4,3
```

- **Xóa phần tử**
  - Xóa phần tử tại vị trí đầu tiên

```
1   var a = [1,2,3,4,5]
2   a.shift()
3   console.log(a)

CONSOLE ×

▸ (4) [2, 3, 4, 5]
```

  - Xóa phần tử ở vị trí cuối cùng

```
1   var a = [1,2,3,4,5]
2   a.pop()
3   console.log(a)

CONSOLE ×

▸ (4) [1, 2, 3, 4]
```

- Sử dụng **concat** để nối các mảng hiện có bằng cách tạo một mảng mới và nối chúng lại với nhau.

```
1   var a = [1,2,3,4,5]
2   var b = [6,8,9,10]
3   var c = ['hacker','lord']
4   var d = a.concat(b,c)
5   console.log(d)

CONSOLE ×

▸ (11) [1, 2, 3, 4, 5, 6, 8, 9, 10, "hacker...]
```

- **Method slice()**: Cắt một phần của mảng tạo thành mảng mới mà không làm thay đổi mảng gốc
  - Mảng được được cắt sẽ từ đối số thứ nhất đến đối số thứ 2

```
const fruits = ["Banana", "Orange", "Apple", "Mango","asasd"];
var newfruits = fruits.slice(2,4);
console.log(newfruits)
```

```
CONSOLE ×
▸ (2) ["Apple", "Mango"]
```

- **Săp xếp mảng**

```
var a = [5,2,1,4,3]
console.log(a.sort())
```

```
CONSOLE ×
▸ (5) [1, 2, 3, 4, 5]
```

  - Hàm **sort()** chỉ cùng để sắp xếp chuỗi, phương thức **sort()** tạo ra kết quả không chính xác khi sắp xếp các số. Giải thích:

```
const a = [5, 1, 10, 25, 100, 40];
a.sort();
console.log(a);
// 25 đứng sau 100 vì chữ số 2 > chữ số 1
// 5 được xếp đứng cuối vì chữ số 5 là chữ số lớn nhất
// điều này xảy ra là vì hàm sort() chỉ dành cho chuỗi
```

```
CONSOLE ×
▸ (6) [1, 10, 100, 25, 40, 5]
```

  - Để khắc phục tình trạng trên, xử lý như sau:

```
const a = [5, 1, 10, 25, 100, 40];
a.sort(function (a,b){return a - b});
console.log(a);
// truyền hàm function (a,b){return a - b} là đối số của hàm sort()
```

```
CONSOLE ×
▸ (6) [1, 5, 10, 25, 40, 100]
```

    → Khi hàm sort () so sánh hai giá trị, nó sẽ gửi các giá trị đến hàm so sánh và sắp xếp các giá trị theo giá trị trả về (âm, không, dương).
    → Nếu kết quả là âm, a được sắp xếp trước b.
    → Nếu kết quả là dương b được sắp xếp trước a.
    → Nếu kết quả là 0 thì không có thay đổi nào được thực hiện với thứ tự sắp xếp của hai giá trị.

- **Tìm Giá trị lớn nhất, nhỏ nhất**
  - Giá trị lớn nhất

```
const a = [5, 1, 10, 25, 100, 40];
var max = Math.max.apply(null,a)
console.log(max)
```

```
CONSOLE ×

100
```

- Giá trị nhỏ nhất

```
const a = [5, 1, 10, 25, 100, 40];
var min = Math.min.apply(null,a)
console.log(min)
```

```
CONSOLE ×

1
```

- **Mảng với vòng lặp**
  - Lặp **foreach()**
    - Cách hoạt động: gọi lại 1 hàm callback, mỗi lần tương ứng với mỗi phần tử

```
const a = [1, 2, 3, 4, 5,6,7,8,9,10];
var sum = 0
function getSum(value,index,arr){
  sum += value
}
a.forEach(getSum)
console.log(sum)
```

```
CONSOLE ×

55
```

  - Lặp **for(; ;)**

```
const a = [1, 2, 3, 4, 5,6,7,8,9,10];
var sum=0
for(let i=0;i<a.length;i++)
  sum+=a[i]
console.log(sum)
```

```
CONSOLE ×

55
```

  - Lặp **map()**:
    - Trả về các thay đổi ở mảng cũ cho 1 mảng mới, mà không làm thay đổi mảng cũ

```
const a = [1, 2, 3, 4, 5,6,7,8,9,10];
var sum = 0
function chang2(value){
  return value*2
}
var b = a.map(chang2)
console.log('a: ',a)
console.log('b: ',b)
```

```
CONSOLE ×
a:  ▸ (10) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b:  ▸ (10) [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

- Array **filter()**:
  - Lọc các phần tử thỏa mãn 1 điều kiện nào đó và trả về cho mảng mới, không làm thay đổi mảng cũ:

```
const a = [1, 2, 3, 4, 5,6,7,8,9,10];
function filter(value,index,arr){
  if(value%2==0)
    return value
}
var b = a.filter(filter)
console.log('b: ',b)
```

```
CONSOLE ×
b:  ▸ (5) [2, 4, 6, 8, 10]
```

- **Tìm kiếm phần tử trong mảng:**
  - Phương thức **indexOf()**: Tìm 1 phần tử và trả về vị trí của phần tử đó nếu không có trả vè -1

```
const numbers = [1, 2, 3, 4, 5];
var findX = numbers.indexOf(4)
console.log('vị trí của 4 trong mảng là',findX)
```

```
CONSOLE ×
vị trí của 4 trong mảng là 3
```

  - Phương thức **find ()** trả về giá trị của phần tử mảng đầu tiên vượt qua một hàm kiểm tra.

```
const numbers = [1, 2, 3, 4, 5];
function myFunc (value,index,arr){
  if(value>3)
    return value
}
var x = numbers.find(myFunc)
console.log('Giá trị đầu tiên thỏa mãn test:',x)
```

```
CONSOLE ×

Giá trị đầu tiên thỏa mãn test:  4
```

5. Condition

- Câu lệnh điều kiện

  Syntax

  ```
  if (condition1) {
    //  block of code to be executed if condition1 is true
  } else if (condition2) {
    //  block of code to be executed if the condition1 is false and condition2 is true
  } else {
    //  block of code to be executed if the condition1 is false and condition2 is false
  }
  ```

- Switch case:

  Syntax

  ```
  switch(expression) {
    case x:
      // code block
      break;
    case y:
      // code block
      break;
    default:
      // code block
  }
  ```

6. Loop

★ Vòng Lặp for()

- **For(; ;)**

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

- **for in:**
  - Sử dụng để lặp qua thuộc tính của một đối tượng

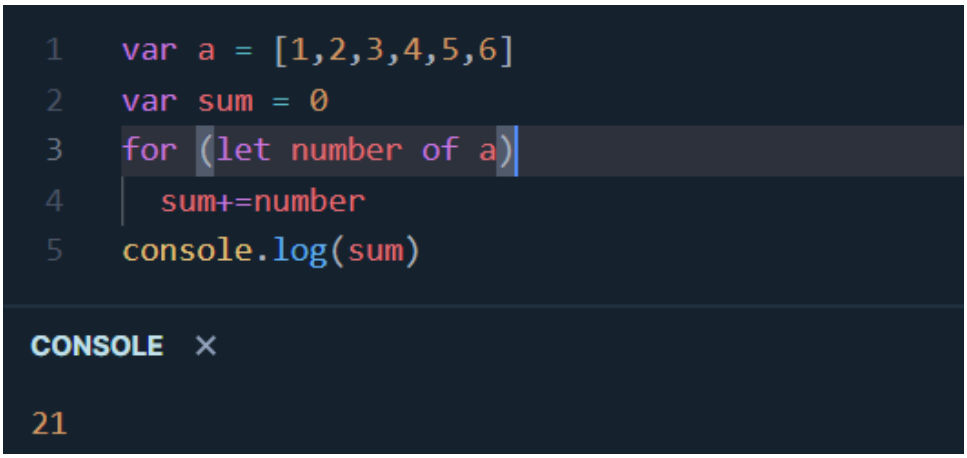
```
for (key in object) {
  // code block to be executed
}
```

  - Nó cũng được dùng để lặp qua các phần tử của một mảng

```
for (variable in array) {
  code
}
```

- **foreach()**
  - Được giới thiệu kĩ ở mục mảng,phần mảng với vòng lặp **foreach()**
- **For at()**
  - Nó cho phép lặp qua các cấu trúc dữ liệu có thể lặp lại như Mảng, Chuỗi, map, Danh sách Node, v.v.
  - Ví dụ, lặp các phần tử của 1 chuỗi

  - Ví dụ lặp các phần tủ của một mảng



    - lặp với object tương tự

★ Vòng lặp while()

- Cấu trúc cơ bản
    - Syntax

```
while (condition) {
  // code block to be executed
}
```

- do while
    - Syntax

```
do {
  // code block to be executed
}
while (condition);
```