



VIETNAM
AUSTRALIA
Vocational College

Các cấu trúc lệnh, phạm vi biến, var, let, const

Mentor: Nguyễn Bá Minh Đạo



Nội dung:

1. Các cấu trúc điều khiển
2. Chế độ nghiêm ngặt strict
3. Các cấu trúc bước nhảy
4. Các cấu trúc vòng lặp
5. Phạm vi biến, var, let, const



Các cấu trúc điều khiển

- ❑ Ví dụ khi **mua điện thoại ở cửa hàng di động**, nhân viên bán hàng hỏi bạn: “**Bạn có muốn thêm bảo vệ màn hình chỉ với \$9.99 không?**”
- ❑ Nhân viên đã **tạo ra một quyết định** cho bạn. Đây là câu hỏi “**yes/no**”.
- ❑ Có **nhiều cách** để bạn **diễn đạt điều kiện** trong **chương trình** -> Loại thường thấy là lệnh **if**. Hiểu đơn giản “**Nếu điều kiện này đúng, hãy làm...**”.





Các cấu trúc điều khiển

- ❑ Lệnh **if** cần **diễn đạt trong dấu ngoặc đơn ()** và sẽ được **thể hiện qua 2 giá trị true hoặc false**.
- ❑ Trong ví dụ, biểu thức **amount < bank_balance**, nó sẽ xác định **true** hoặc **false** tùy thuộc vào giá trị của biến **bank_balance**.

JS if_else.js X

JS if_else.js > ...

```
1  var bank_balance = 302.13; //số dư ngân hàng
2
3  var amount = 99.99;
4  if (amount < bank_balance) {
5      console.log("I want to buy this phone!");
6  }
```



Các cấu trúc điều khiển

- ❑ Bạn cũng **có thể cung cấp một sự kiện thay thế nếu điều kiện không thỏa mãn**, gọi là mệnh đề **else**. Ví dụ:
- ❑ Nếu **amount < bank_balance** là **true**, chúng ta sẽ **lấy thêm phụ kiện**.
- ❑ Ngược lại là **false**, thì mệnh đề **else** trả lời **không cần**.

JS if_else_02.js X

JS if_else_02.js > ...

```
1  const ACCESSORY_PRICE = 9.99; //giá phụ kiện
2  var bank_balance = 302.13; //số dư ngân hàng
3  var amount = 99.99;
4  amount = amount * 2; //giá điện thoại tăng gấp đôi
5
6  //can we afford the extra purchase?
7  if (amount < bank_balance) {
8      console.log("I'll take the accessory!");
9      amount = amount + ACCESSORY_PRICE;
10 } else {
11     console.log("No, thanks.");
12 }
```



Các cấu trúc điều khiển

- ❑ Đôi khi bạn cũng sẽ **phải sử dụng một loại điều kiện if...else..if** như sau:
- ❑ Cấu trúc này hoạt động hơi rườm rà vì bạn cần kiểm tra a cho mỗi trường hợp.
- ❑ Một phương thức khác hỗ trợ trường **if** trên hiệu quả hơn là **biểu thức switch**.
- ❑ **break** quan trọng nếu bạn muốn mỗi biểu thức trong một **case** hoạt động.

```
JS if_else_if.js X
JS if_else_if.js > ...
1  var a = parseInt("Enter a: ");
2
3  if (a == 2) {
4      // do something
5  } else if (a == 10) {
6      // do something
7  } else if (a == 42) {
8      // do something
9  } else {
10     // not do something will here
11 }
```



Các cấu trúc điều khiển

- ❑ Đôi khi bạn cũng sẽ **phải sử dụng một loại điều kiện if...else..if** như sau:
- ❑ Cấu trúc này hoạt động hơi rườm rà vì bạn cần kiểm tra a cho mỗi trường hợp.
- ❑ Một phương thức khác hỗ trợ trường **if** trên hiệu quả hơn là **biểu thức switch**.
- ❑ **break** quan trọng nếu bạn muốn mỗi biểu thức trong một **case** hoạt động.

```
> // Kiểm tra số chẵn, số lẻ
var number = parseInt(prompt("Nhập số cần kiểm tra"));

var mod = (number % 2);

switch (mod) {
  case 0 : {
    console.log(number + " là số chẵn");
    break;
  }
  case 1: {
    console.log(number + " là số lẻ");
    break;
  }
  default : {
    console.log("Ký tự bạn nhập không phải số");
  }
}
```

9 là số lẻ



Các cấu trúc điều khiển

- ❑ Một **dạng khác** của **điều kiện trong JavaScript** là “**điều hành điều kiện**”, thường **được gọi** là “**toán tử bậc 3**”
- ❑ Nó như là **một dạng rút gọn của biểu thức if..else**. Ví dụ:

```
JS if_else_03.js X
JS if_else_03.js > ...
1  var a = 42;
2  var b = (a > 41) ? "hello" : "world";
3
4  // the same with:
5  // if (a > 41) {
6  //     b = "hello";
7  // } else {
8  //     b = "world";
9  // }
```

- ❑ Nếu **biểu thức** ($a > 41$ ở đây) thỏa **true**, kết quả là **mệnh đề đầu tiên("hello")**
- ❑ Ngược lại, kết quả là ("world"), và cho **dù kết quả là gì** thì **đều gán vào b**

```
JS if_else_03.js X
JS if_else_03.js > ...
1  var a = 42;
2  var b = (a > 41) ? "hello" : "world";
3
4  // the same with:
5  // if (a > 41) {
6  //     b = "hello";
7  // } else {
8  //     b = "world";
9  // }
```




Chế độ nghiêm ngặt strict

- ❑ Trong ES5, JavaScript được bổ sung chế độ strict “**strict mode**”.
- ❑ Giúp các hành vi nhất định có nguyên tắc chặt chẽ hơn, giữ cho code an toàn và phù hợp hơn.
- ❑ Bạn có thể sử dụng chế độ strict cho một hàm riêng biệt hay toàn bộ file, tùy thuộc vào việc bạn đặt strict đó ở đâu.

```
JS strict_01.js X JS strict_02.js
JS strict_01.js > hello
1 function hello() {
2
3     "use strict";
4     // The code below use strict mode
5     function world() {
6         // The code here also use strict mode
7     }
8 }
9
10 // The code below do not use strict mode anymore
```

```
JS strict_01.js X JS strict_02.js X
JS strict_02.js > ...
1 "use strict";
2
3 function hello() {
4
5     // The code below use strict mode
6     function world() {
7         // The code here also use strict mode
8     }
9 }
10
11 // The code here also use strict mode
```



Chế độ nghiêm ngặt strict

- ❑ Điểm khác biệt mấu chốt (cải tiến) ở chế độ strict ở 2 ví dụ trên là **không cho phép tự động tiềm ẩn khai báo toàn cục khi bỏ qua var.**

JS strict_03.js X

JS strict_03.js > ...

```
1  function hello() {  
2  
3      "use strict"; // use strict mode  
4      a = 1; // `var` missing, ReferenceError  
5  }  
6  
7  hello();
```



Cấu trúc vòng lặp

- ❑ Lặp lại một tập hợp hành động cho đến khi có điều kiện thất bại – **chỉ lặp lại khi điều kiện thỏa mãn** – là công việc của **vòng lặp** chương trình.
- ❑ Một vòng lặp **bao gồm điều kiện kiểm tra** cũng như một block (thường là **{..}**). **Mỗi lần block vòng lặp** được **thực hiện**, nó được gọi là **sự lặp lại**.





Cấu trúc vòng lặp

❑ Cú pháp **for**:

```
for (begin; condition; step) {  
    // code to be executed  
}
```

❑ Trong đó:

- **for**: khởi tạo vòng lặp.
- **begin** (khởi tạo): khởi tạo giá trị cho biến đếm
- **condition** (điều kiện): điều kiện đặc biệt nếu đúng sẽ thực hiện bước tiếp theo.
- **step** (lặp lại): tăng hoặc giảm biến đếm



Cấu trúc vòng lặp

❑ **Ví dụ 1:** Tính tổng các số từ 1 đến 10 sử dụng **vòng lặp for**.

JS for.js X

JS for.js > ...

```
1  var i, sum = 0;
```

```
2
```

```
3  for (i = 0; i < 10; i++) {
```

```
4      sum += i;
```

```
5  }
```

```
6  console.log(`Sum is: ${sum}`);
```



Cấu trúc vòng lặp

❑ Cú pháp **while**:

begin

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

❑ Trong đó:

- **begin** (khởi tạo): khởi tạo giá trị cho biến đếm
- **while**: trong khi điều kiện còn đúng, còn thực thi khối lệnh.
- **condition** (điều kiện): điều kiện đặc biệt nếu đúng sẽ thực hiện bước tiếp theo.
- **step** (lặp lại): tăng hoặc giảm biến đếm



Cấu trúc vòng lặp

❑ **Ví dụ 2:** Tính tổng các số từ 1 đến 10 sử dụng **vòng lặp while**.

```
JS while.js X
JS while.js > ...
1   var i = sum = 0;
2
3   while (i < 10) {
4       sum += i;
5       i++;
6   }
7   console.log(`Sum is: ${sum}`);
```



Cấu trúc vòng lặp

❑ Cú pháp **do-while**:

begin

do {

// code to be executed

step

} **while** (**condition**);

❑ Trong đó:

- **begin** (khởi tạo): khởi tạo giá trị cho biến đếm
- **do..while**: sẽ thực thi khối lệnh ít nhất 1 lần .
- **condition** (điều kiện): điều kiện đặc biệt nếu đúng sẽ thực hiện bước tiếp theo.
- **step** (lặp lại): tăng hoặc giảm biến đếm



Cấu trúc vòng lặp

❑ **Ví dụ 3:** Tính tổng các số từ 1 đến 10 sử dụng **vòng lặp do-while**.

```
JS do_while.js X
```

```
JS do_while.js > ...
```

```
1   var i = sum = 0;
```

```
2
```

```
3   do {
```

```
4       sum += i;
```

```
5       i++;
```

```
6   } while (i < 10);
```

```
7   console.log(`Sum is: ${sum}`);
```



Lệnh break và continue

❑ Lệnh **break**:

- Lệnh **break** được sử dụng để thoát ra khỏi cấu trúc lặp (for, while, do..while) và cấu trúc rẽ nhánh (switch..case)
- Lệnh **break** không cần dùng điều kiện kết thúc vòng lặp.
- Khi có nhiều vòng lặp lồng nhau, **break** thoát ra khỏi vòng lặp bên trong khối lệnh lặp chứa nó.

```
JS break.js  X
JS break.js > ...
1  for (let i = 0; i <= 10; i++) {
2      if ((i == 3) || (i == 5) || (i == 7)) {
3          break;
4          console.log(i);
5      }
6      console.log(i);
7  }
```



Lệnh break và continue

❑ Lệnh **continue**:

- Lệnh **continue** được sử dụng để bắt đầu một vòng mới của cấu trúc lặp chứa nó.
- Lệnh **continue** thường được sử dụng bên trong thân của vòng lặp **for**.

```
JS continue.js X
JS continue.js > ...
1  for (let i = 0; i <= 10; i++) {
2      if ((i == 3) || (i == 5) || (i == 7)) {
3          continue;
4          console.log(i);
5      }
6      console.log(i);
7  }
```



Phạm vi biến, var, let, const

- ❑ Khi viết chương trình, ta sẽ gặp **các đoạn code thực thi các công việc giống nhau**, làm sao để **tránh viết đi viết lại** nhiều lần 1 đoạn code?
- ❑ Ta cần xác định một **hàm** (function).
- ❑ Một hàm thường là **một phần code được đặt tên** và **có thể gọi bằng tên**, code bên trong nó **sẽ chạy** cho **mỗi lần gọi**.



```
JS function_01.js X
JS function_01.js > ...
1  function printAmount() {
2      console.log(amount.toFixed(2));
3  }
4
5  var amount = 99.99;
6  printAmount(); // "99.99"
7  amount = amount * 2;
8  printAmount(); // "199.98"
```



Phạm vi biến, var, let, const

- ❑ Các **hàm** có thể **tùy ý lấy đối số** (giá trị mà các bạn truyền vào) và chúng có thể **tùy ý trả lại giá trị**.
- ❑ Các **hàm** thường được **sử dụng với mục đích gọi ra nhiều lần**, nhưng nó cũng có thể hữu ích trong việc tổ chức code, kể cả **khi chỉ gọi 1 lần**.

```
JS function_02.js X
JS function_02.js > ...
1  function printAmount(amt) {
2    console.log(amt.toFixed(2));
3  }
4
5  function formatAmount() {
6    return "$" + amount.toFixed(2);
7  }
8
9  var amount = 99.99;
10 printAmount(amount * 2); // "199.98"
11 amount = formatAmount();
12 console.log(amount); // "$99.99"
```

```
JS function_03.js X
JS function_03.js > ...
1  const TAX_RATE = 0.08;
2
3  function calculateFinalPurchaseAmount(amt) {
4    //calculate the new amount with the tax
5    amt = amt + (amt * TAX_RATE);
6    // return the new amount
7    return amt;
8  }
9
10 var amount = 99.99;
11 amount = calculateFinalPurchaseAmount(amount);
12 console.log(amount.toFixed(2)); // "107.99"
```



Phạm vi biến, var, let, const

- ❑ Nếu bạn hỏi nhân viên cửa hàng về một **mẫu điện thoại** mà **cửa hàng không có**, cô ta sẽ không thể bán chiếc điện thoại bạn muốn.
- ❑ Cô ta **chỉ có thể bán những chiếc điện thoại trong kho**. Và bạn có thể sẽ phải thử tìm ở cửa hiệu khác để tìm chiếc điện thoại bạn muốn.

Phạm vi của biến
JavaScript



Phạm vi biến, var, let, const

- ❑ Tương tự tình huống trên, lập trình có một thuật ngữ cho khái niệm này là: **phạm vi - scope** (kỹ thuật gọi là **lexical scope**)
- ❑ Trong JavaScript, mỗi hàm đều có scope của nó.
- ❑ **Scope** cơ bản là **một bộ tập hợp** của **các biến** cũng như **quy tắc cho các biến đó được gọi theo tên**.

Phạm vi của biến
JavaScript



Phạm vi biến, var, let, const

- ❑ Chỉ có **code trong hàm mới có thể tiếp cận được** với **các biến trong scope của hàm đó**.
- ❑ **Tên biến trong cùng scope phải là duy nhất** – không thể có hai biến a khác nhau tồn tại kế bên.
- ❑ Nhưng **biến a trùng nhau có thể tồn tại trong các scope khác nhau**.

Phạm vi của biến
JavaScript



Phạm vi biến, var, let, const

❑ **Ví dụ:** Hai biến a ở hai hàm scope khác nhau nên chúng có thể cùng tồn tại.

JS scope_01.js X

JS scope_01.js > ...

```
1  function one() {
2      // this `a` only belongs to the `one()` function
3      var a = 1;
4      console.log(a);
5  }
6
7  function two() {
8      // this `a` only belongs to the `two()` function
9      var a = 2;
10     console.log(a);
11 }
12
13 one(); // 1
14 two(); // 2
```



Phạm vi biến, var, let, const

- ❑ Một scope **có thể lồng bên trong scope khác**. Khi đó, code bên trong scope sâu nhất **có thể tiếp cận** với mọi biến ở các scope bên ngoài.
- ❑ Các nguyên tắc của lexical scope **cho phép code có thể truy cập các biến của phạm vi bên trong** hay **bên ngoài scope**

```
JS scope_02.js X
JS scope_02.js > ...
1  function outer() {
2      var a = 1;
3
4      function inner() {
5          var b = 2;
6          // we can access both `a` and `b` here
7          console.log(a + b); // 3
8      }
9      inner();
10
11     // we can only access `a` here
12     console.log(a); // 1
13 }
14
15 outer();
```



Phạm vi biến, var, let, const

❑ const:

➤ Dùng để **khai báo một hằng số** - là một giá trị không thay đổi được trong suốt quá trình chạy.

➤ Ví dụ:

```
const a = 5;
```

```
a = 10; // Lỗi Uncaught TypeError: Assignment to constant variable
```

```
JS const.js  X
JS const.js > ...
1  const a = 5;
2  a = 10; // Lỗi Uncaught TypeError: Assignment to constant variable
```



Phạm vi biến, var, let, const

❑ var & let:

➤ Tạo ra một biến **chỉ có thể truy cập được trong block bao quanh nó**, khác với **var** - tạo ra một biến **có phạm vi truy cập xuyên suốt function chứa nó**. Ví dụ var & let:

```
JS var.js X
JS var.js > ...
1  function foo() {
2      var x = 10;
3      if (true) {
4          var x = 20; // x ở đây cũng là x ở trên
5          console.log(x); // in ra 20
6      }
7      console.log(x); // vẫn là 20
8  }
9
10 foo();
```

```
JS let.js X
JS let.js > ...
1  function foo() {
2      let x = 10;
3      if (true) {
4          let x = 20; // x này là x khác rồi đây
5          console.log(x); // in ra 20
6      }
7      console.log(x); // in ra 10
8  }
9
10 foo();
```



Phạm vi biến, var, let, const

❑ var & let:

➤ Ngoài ra, khi ở **global scope** (tức là không nằm trong một function nào cả), **từ khóa var** tạo ra thuộc tính mới cho **global object** (**this**), còn **let** thì không.

```
> var x = 'global';  
   let y = 'global';  
   console.log(this.x); // "global"  
   console.log(this.y); // undefined
```

global

undefined



Phạm vi biến, var, let, const

❑ var & let:

➤ Có một trường hợp dùng **let** rất hiệu quả đó là sử dụng **callback** trong một vòng lặp. **Giá trị** của **biến i** bên trong hàm **callback** luôn là giá trị cuối cùng của i trong vòng lặp.

```
JS let_callback.js X
JS let_callback.js > ...
1   for (var i = 0; i < 5; i++) {
2       |   setTimeout(function() {
3           |       console.log('Yo! ', i);
4           |   }, 1000);
5       |   }
```

```
Yo! 5
Yo! 5
Yo! 5
Yo! 5
Yo! 5
```



Phạm vi biến, var, let, const

❑ var & let:

➤ Để giải quyết vấn đề này, chúng ta thay **var** bằng **let**:

```
JS let_callback_02.js X
```

```
JS let_callback_02.js > ...
```

```
1  ✓ for (let i = 0; i < 5; i++) {  
2  ✓   |   setTimeout(function() {  
3      |       console.log('Yo! ', i);  
4      |   }, 1000);  
5  }  |
```

```
Yo! 0  
Yo! 1  
Yo! 2  
Yo! 3  
Yo! 4
```



Phạm vi biến, var, let, const

❑ Khi nào dùng **var**, khi nào dùng **let**:

➤ **Lưu ý**: sự phân biệt **var**, **let** này chỉ áp dụng khi các bạn **làm việc với ES6** nhé!

➤ **Nếu dùng ES6**, ta lưu ý:

- **Không dùng var** trong bất kỳ **mọi trường hợp**.
- **Thay vào đó** thì **dùng let**
- Dùng **const** khi cần **định nghĩa một hằng số**.



Tổng kết:

- ☐ Các cấu trúc điều khiển
- ☐ Chế độ nghiêm ngặt strict
- ☐ Các cấu trúc bước nhảy
- ☐ Các cấu trúc vòng lặp
- ☐ Phạm vi biến, var, let, const

Let's
Recap

