



---

# **Lesson 05**

# **Commonly Used ES6 Features, Callback, Async/Wait**

Module 02: WEB INTERACTION

# Mục tiêu

---



- Sử dụng được tính năng Modules trong ES6
- Sử dụng được Default Parameter, Rest Parameter
- Sử dụng được Spread Operator, Template Literals
- Sử dụng được Enhanced Object Properties
- Sử dụng được Destructuring Assignment
- Sử dụng được Callback, Asynchronous
- Sử dụng được Promise, Async/Await

# Tính năng Modules trong ES6

---



## ❑ Modules trong ES6:

➤ **Export/Import** giá trị: trước đây khi code JavaScript, tất cả các file mà có khởi tạo biến, hàm,...thì các file load sau nó đều có thể sử dụng được.

### ➤ Cú pháp:

- **export data**;
- **import name from path**;
- Trong đó:
  - **data**: những gì muốn xuất ra cho các module khác có thể sử dụng.
  - **name**: là biến muốn gán để nhận dữ liệu trả về từ module đó.
  - **path**: đường dẫn chứa module bạn cần import.

# Tính năng Modules trong ES6



## ❑ Modules trong ES6:

```
Card.js
src > components > common > Card.js > ...
1 import React from "react";
2
3 function Card() {
4   return (
5     <div className="col-12 col-lg-6 col-xl">
6       <div className="card">
7         <div className="card-body">
8           <div className="row align-items-center">
9             <div className="col">
10              <h6 className="text-uppercase text-muted mb-2">Value</h6>
11              <span className="h2 mb-0">$24,500</span>
12            </div>
13            <div className="col-auto">
14              <span className="h2 fe fe-dollar-sign text-muted mb-0">Xem</span>
15            </div>
16          </div>
17        </div>
18      </div>
19    </div>
20  );
21 }
22
23 export default Card;
24
```

```
Admin.js
src > components > main > Admin.js > ...
1 import React from "react";
2 import Card from "../common/Card";
3 import Navbar from "../common/Navbar";
4 import Table from "../common/Table";
5
6 function Admin() {
7   return (
8     <div className="main-content">
9       { " " }
10      <Navbar />
11      <div className="container-fluid">
12        <div className="row justify-content-center">
13          <div className="col-12">
14            <div className="row">
15              <Card />
16              <Card />
17              <Card />
18            </div>
19
20            <div className="card">
21              <div className="card-header">
22                <h4 className="card-header-title">Tài khoản</h4>
23                <div>
24                  <button className="btn btn-success">Tạo tài khoản</button>
25                </div>
26              </div>
27              <Table />
28            </div>
29          </div>
30        </div>
31      </div>
32    </div>
33  );
34 }
35
36 export default Admin;
```

# Default Parameter, Rest Parameter



- ❑ ES6 **cho phép** các tham số trong hàm có giá trị mặc định:
- ❑ ES6 **cung cấp tham số rest (...)** **cho phép** một hàm coi một số lượng vô hạn đối số là một mảng.

```
function myFunction(x, y = 10) {  
  // y is 10 if not passed or undefined  
  return x + y;  
}
```

```
myFunction(5);  
15
```

```
> function sum(...args) {  
  let sum = 0;  
  for (let arg of args) sum += arg;  
  return sum;  
}
```

```
let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

```
console.log(x);
```

```
326
```



# Spread Operator, Template Literals

- ❑ ES6 cho phép chúng ta xử lý các tham số mở rộng trong lập trình.
- ❑ **Spread Operator** (toán tử **Spread**) hỗ trợ **phân tán các phần tử** của **một danh sách/tập hợp có thể lặp lại** (như **một mảng** hoặc thậm chí **một chuỗi**) thành các phần tử bằng chữ và các tham số hàm riêng lẻ.

```
var params = [ "hello", true, 7 ]
var other = [ 1, 2, ...params ] // [ 1, 2, "hello", true, 7 ]

function f (x, y, ...a) {
    return (x + y) * a.length
}
f(1, 2, ...params) === 9

var str = "foo"
var chars = [ ...str ] // [ "f", "o", "o" ]
```

# Spread Operator, Template Literals



## ❑ Cú pháp Back-Ticks:

➤ **Template Literals** (chuỗi mẫu) thường sử dụng **back-ticks** (```) thay vì **quotes** (`"`) để định nghĩa một chuỗi.

```
> let text2 = `Hello world!`;
   console.log(text2);
```

```
Hello world!
```

➤ Với **template literals**, chúng ta có thể dùng cả nháy đơn, nháy đôi bên trong string.

```
> let text3 = `He's often called "Nguyễn Văn Tèo"`;
   console.log(text3);
```

```
He's often called "Nguyễn Văn Tèo"
```

# Spread Operator, Template Literals



## ❑ Biến/biểu thức thay thế (Variable/Expression Substitution):

- **Template Literals** cho phép sử dụng **biến** trong chuỗi.

```
let firstName2 = "Tèo";  
let lastName2 = "Nguyễn";
```

```
let fullName2 = `Welcome ${firstName2}, ${lastName2}!`;
```

```
console.log(fullName2);
```

```
Welcome Tèo, Nguyễn!
```

- **Template Literals** cho phép sử dụng **biểu thức** trong chuỗi.

```
> let price = 10;  
let VAT = 0.25;  
let total = `Total: ${((price * (1 + VAT)).toFixed(2))}`;
```

```
console.log(total);
```

```
Total: 12.50
```



# Spread Operator, Template Literals



## ❑ Mẫu HTML (HTML Templates):

➤ **Template Literals** cho phép sử dụng **HTML** trong chuỗi.

```
> let header = "Templates Literals";  
   let tags = ["template literals", "javascript", "es6"];  
  
   let html = `

## ${header}</h2><ul>`; for (const x of tags) { html += `- ${x}</li>`; } html += `</ul>`; document.getElementById("demo").innerHTML = html; < ' <h2>Templates Literals</h2><ul><li>template literals</li><li>javascript</li><li>es6</li></ul>'


```

➤ **Templates Literals không** được **hỗ trợ trong Internet Explorer**.



# Enhanced Object Properties

## ❑ Enhanced Object Properties trong ES6:

➤ Hỗ trợ các kiểu khai báo thuộc tính, phương thức ngắn gọn, đơn giản để làm việc với đối tượng dễ dàng hơn.

➤ Hỗ trợ các kiểu khai báo và sử dụng sau:

▪ **Property Shorthand**: binding biến vào trong object và sẽ nhận luôn tên của biến đó là thuộc tính của object. Ví dụ:

```
> var name = "Nguyễn Văn Tèo";  
   var age = 18;  
   var student = {name: name, age: age};  
   console.log(student);
```

ES5

```
▶ {name: 'Nguyễn Văn Tèo', age: 18}
```

```
var name = "Nguyễn Văn Tèo";  
var age = 18;  
var student = {name, age};  
console.log(student);
```

ES6

```
▶ {name: 'Nguyễn Văn Tèo', age: 18}
```

# Enhanced Object Properties



## ❑ Enhanced Object Properties trong ES6:

- Hỗ trợ các kiểu khai báo thuộc tính, phương thức ngắn gọn, đơn giản để làm việc với đối tượng dễ dàng hơn.
- Hỗ trợ các kiểu khai báo và sử dụng sau:
  - **Computed Property Names:** tính toán, tạo biểu thức các giá trị ngay trên tên của thuộc tính. Ví dụ:

```
> var n = "Name";  
   var student = {  
     ["student" + n] : "Nguyễn Văn Tèo",  
     age: 18  
   };  
   console.log(student);  
  
▶ {studentName: 'Nguyễn Văn Tèo', age: 18}
```

# Enhanced Object Properties



## ❑ Enhanced Object Properties trong ES6:

- Hỗ trợ các kiểu khai báo thuộc tính, phương thức ngắn gọn, đơn giản để làm việc với đối tượng dễ dàng hơn.
- Hỗ trợ các kiểu khai báo và sử dụng sau:
  - **Method Properties:** Trước đây, để khai báo một phương thức trong đối tượng thì chúng ta thường phải lách luật bằng cách đặt nó dưới dạng một thuộc tính và xử lý bằng hàm ẩn danh (closure). Ví dụ:

```
> var student = {  
  name: "Nguyễn Văn Tèo",  
  age: 18,  
  getName: function () {  
    return this.name;  
  },  
  getAge: function () {  
    return this.age;  
  }  
};  
console.log(student.getName()); // Nguyễn Văn Tèo  
console.log(student.getAge()); // 18
```

# Enhanced Object Properties



## ❑ Enhanced Object Properties trong ES6:

- Hỗ trợ các kiểu khai báo thuộc tính, phương thức ngắn gọn, đơn giản để làm việc với đối tượng dễ dàng hơn.
- Hỗ trợ các kiểu khai báo và sử dụng sau:
  - **Method Properties:** Với ES6, ta có thể khai báo phương thức như một hàm bình thường, mà **không phải** dựa vào hàm ẩn danh nữa. Ví dụ:

```
> var student = {  
  name: "Nguyễn Văn Tèo",  
  age: 18,  
  getName() {  
    return this.name;  
  },  
  getAge() {  
    return this.age;  
  }  
};  
console.log(student.getName()); // Nguyễn Văn Tèo  
console.log(student.getAge()); // 18
```

```
> var student = {  
  name: "Nguyễn Văn Tèo",  
  setName(name) {  
    this.name = name;  
  },  
  getName() {  
    return this.name;  
  }  
};  
student.setName("Trần Văn Tồ")  
console.log(student.getName()); // Trần Văn Tồ
```

# Destructuring Assignment



## ❑ Destructuring Assignment trong ES6:

➤ Hỗ trợ so khớp, chuyển đổi mảng, object trong ES6.

### ➤ Array Matching:

- Trong ES6, ta có thể pass data từ mảng ra biến nhanh chóng, tiện lợi.

```
> var list = [1,2,3];  
var [a, b] = list;  
console.log("a = " + a, "b = " + b);  
  
a = 1 b = 2
```

- Ngoài ra, ta còn có thể đổi vị trí các phần tử rất ngắn gọn như sau:

```
> var list = [1,2,3];  
var [a, b] = list;  
console.log("a = " + a, "b = " + b);  
  
[b, a] = [a, b]; //đổi vị trí  
console.log("a = " + a, "b = " + b);  
  
a = 1 b = 2  
  
a = 2 b = 1
```

# Destructuring Assignment



## ❑ Destructuring Assignment trong ES6:

- Hỗ trợ so khớp, chuyển đổi mảng, object trong ES6.
- **Object Matching, Shorthand Notation:**
  - Trong ES6, với Object, ta cũng có thể làm tương tự như với mảng.

```
> var getObject = () => {  
    return {  
        name : "Nguyễn Văn Tèo",  
        age : 18,  
        hometown: "Việt Nam"  
    };  
}  
var {name, age, hometown} = getObject();  
console.log("name: " + name,  
    "age: " + age,  
    "hometown: " + hometown);
```

```
name: Nguyễn Văn Tèo age: 18 hometown: Việt Nam
```

# Destructuring Assignment



## ❑ Destructuring Assignment trong ES6:

➤ Hỗ trợ so khớp, chuyển đổi mảng, object trong ES6.

➤ **Object Matching, Deep Matching:**

▪ Nếu như trong trường hợp object của bạn chứa 1 object con nữa thì bạn cũng có thể matching data như sau:

```
> var getObject = () => {  
  return {  
    name : "Nguyễn Văn Tèo",  
    option: {  
      age : 18,  
      hometown: "Việt Nam",  
    }  
  };  
}  
var {name, option: { age, hometown } } = getObject();  
console.log("name: " + name,  
  "age: " + age,  
  "hometown: " + hometown);
```

```
name: Nguyễn Văn Tèo age: 18 hometown: Việt Nam
```



# Destructuring Assignment



## ❑ Destructuring Assignment trong ES6:

➤ Hỗ trợ so khớp, chuyển đổi mảng, object trong ES6.

### ➤ Object And Array Matching, Default Values:

▪ Trường hợp không biết chính xác số lượng phần tử có trong mảng hay object để matching thì ES6 hỗ trợ thiết lập giá trị mặc định như sau:

- Đối với mảng:

```
> var num = [1];  
   var [a = "Default", b = "Default"] = num;  
   console.log(a, b);  
   // a = 1  
   // b = Default  
  
1 'Default'
```



# Destructuring Assignment

---

## ❑ Destructuring Assignment trong ES6:

➤ Hỗ trợ so khớp, chuyển đổi mảng, object trong ES6.

### ➤ Object And Array Matching, Default Values:

▪ Trường hợp không biết chính xác số lượng phần tử có trong mảng hay object để matching thì ES6 hỗ trợ thiết lập giá trị mặc định như sau:

- Đối với object:

```
> var student = { name: "Nguyễn Văn Tèo"};
  var {name = "Not Set", age = "Not Set"} = student;
  console.log(name, age);
  // name = Vũ Thanh Tài
  // age = Not Set
```

---

Nguyễn Văn Tèo Not Set

---

# Destructuring Assignment



## ❑ Destructuring Assignment trong ES6:

➤ Hỗ trợ so khớp, chuyển đổi mảng, object trong ES6.

### ➤ **Parameter Context Matching:**

▪ Tương tự, bạn cũng có thể áp dụng vào việc truyền tham số như sau:

• Đối với mảng:

```
> function logArray ([a, b]) {  
    console.log(a, b);  
}  
logArray(["Tham Số A -", "- Tham số B"]);  
//Tham Số A - - Tham số B
```

---

Tham Số A - - Tham số B

# Destructuring Assignment



## ❑ Destructuring Assignment trong ES6:

➤ Hỗ trợ so khớp, chuyển đổi mảng, object trong ES6.

### ➤ **Parameter Context Matching:**

▪ Tương tự, bạn cũng có thể áp dụng vào việc truyền tham số như sau:

• Đối với object:

```
> function logObject ({a, b}) {  
    console.log(a, b);  
}  
logObject({a: "Tham Số A -", b: "- Tham số B"});  
//Tham Số A - - Tham số B
```

---

Tham Số A - - Tham số B

---

# Callback, Asynchronous

---



## ❑ Callback trong ES6:

- Hàm gọi lại (**callback**) là một hàm truyền một đối số vào một hàm khác.
- Kỹ thuật này cho phép một hàm gọi tới một hàm khác.
- Hàm callback có thể chạy sau khi một hàm khác kết thúc.



# Callback, Asynchronous

---



## ❑ **Callback** trong ES6:

➤ Một cách dễ hiểu, callback tức là ta truyền một đoạn code (**Hàm A**) này vào một đoạn code khác (**Hàm B**). Tới một thời điểm nào đó, Hàm A sẽ được hàm B gọi lại (**callback**).



# Callback, Asynchronous



## ❑ Hàm tuần tự - Function Sequence:

➤ Các hàm JavaScript được thực thi theo trình tự mà chúng được gọi. Không theo trình tự mà chúng được xác định.

<pre>&gt; function myDisplayer(some) {   console.log(some); }  function myFirst() {   myDisplayer("Hello"); }  function mySecond() {   myDisplayer("Goodbye"); }  myFirst(); mySecond();</pre>	<pre>&gt; function myDisplayer(some) {   console.log(some); }  function myFirst() {   myDisplayer("Hello"); }  function mySecond() {   myDisplayer("Goodbye"); }  mySecond(); myFirst();</pre>
Hello	Goodbye
Goodbye	Hello

# Callback, Asynchronous



## ❑ Kiểm soát tuần tự - [Sequence Control](#):

- Đôi khi bạn muốn kiểm soát tốt hơn thời điểm thực thi một hàm.
- Giả sử bạn muốn thực hiện một phép tính và sau đó hiển thị kết quả.
- Bạn có thể gọi một hàm máy tính (myCalculator), lưu kết quả, sau đó gọi một hàm khác (myDisplayer) để hiển thị kết quả:

```
> function myDisplayer(some) {  
  console.log(some);  
}  
  
function myCalculator(num1, num2) {  
  let sum = num1 + num2;  
  return sum;  
}  
  
let result = myCalculator(5, 5);  
myDisplayer(result);  
  
10
```

```
> function myDisplayer(some) {  
  console.log(some);  
}  
  
function myCalculator(num1, num2) {  
  let sum = num1 + num2;  
  myDisplayer(sum);  
}  
  
myCalculator(5, 5);  
  
10
```



# Callback, Asynchronous



## ❑ Kiểm soát tuần tự - Sequence Control:

➤ Bạn có thể gọi một hàm máy tính (myCalculator), lưu kết quả, sau đó gọi một hàm khác (myDisplayer) để hiển thị kết quả:

```
> function myDisplayer(some) {  
    console.log(some);  
}  
  
function myCalculator(num1, num2) {  
    let sum = num1 + num2;  
    return sum;  
}  
  
let result = myCalculator(5, 5);  
myDisplayer(result);  
  
10
```

```
> function myDisplayer(some) {  
    console.log(some);  
}  
  
function myCalculator(num1, num2) {  
    let sum = num1 + num2;  
    myDisplayer(sum);  
}  
  
myCalculator(5, 5);  
  
10
```

➔ Ví dụ 1: phải gọi 2 hàm để có kết quả. Ví dụ 2, không thể ngăn chặn hàm tính toán từ việc hiển thị kết quả. -> Có thể sử dụng callback để xử lý.

# Callback, Asynchronous



## ❑ Hàm gọi lại – Callback Function:

➤ Sử dụng callback có thể gọi hàm tính toán (myCalculator) như là một callback, và hàm tính toán sẽ được chạy lại sau khi việc tính toán hoàn thành:

```
> function myDisplayer(some) {  
  console.log(some);  
}  
  
function myCalculator(num1, num2, myCallback) {  
  let sum = num1 + num2;  
  myCallback(sum);  
}  
  
myCalculator(5, 5, myDisplayer);  
10
```

➤ Trong ví dụ trên, **myDisplayer** là **tên** của **một hàm** và nó **sẽ được truyền vào myCalculator()** như là **một đối số**. **Lưu ý**: không dùng dấu ngoặc đơn

Right: myCalculator(5, 5, myDisplayer);

Wrong: ~~myCalculator(5, 5, myDisplayer());~~

# Callback, Asynchronous

---



## ❑ Bất đồng bộ – *Asynchronous*:

➤ Các hàm chạy song song với các hàm khác được gọi là bất đồng bộ (*Asynchronous*).

➤ Một ví dụ điển hình cho bất đồng bộ là hàm *setTimeout()*

```
> function myDisplayer(some) {  
    console.log(some);  
}
```

```
function myCalculator(num1, num2, myCallback) {  
    let sum = num1 + num2;  
    myCallback(sum);  
}
```

```
myCalculator(5, 5, myDisplayer);
```

---

```
10
```

# Callback, Asynchronous



## ❑ Bất đồng bộ – Asynchronous:

➤ Khi sử dụng hàm `setTimeout()`, bạn có thể chỉ định một hàm gọi lại (callback) được thực thi khi hết thời gian chờ.

```
> setTimeout(myFunction, 3000);  
  
function myFunction() {  
    console.log("I love You !!");  
}
```

```
// I Love You !! (after 3s)
```

➤ Trong đó:

- `myFunction` được dùng như là một hàm callback
- `myFunction` được truyền vào hàm `setTimeout()` như là một đối số
- `3000` là số mili giây trước khi time-out, và `myFunction()` sẽ được gọi lại sau 3 giây.

Lưu ý: không dùng ngoặc đơn khi truyền hàm như một đối số

Right: `setTimeout(myFunction, 3000);`

Wrong: `setTimeout(myFunction(), 3000);`

# Callback, Asynchronous



## ❑ Bất đồng bộ – **Asynchronous**:

➤ Thay vì chuyển tên của một hàm làm đối số cho một hàm khác, bạn luôn có thể chuyển toàn bộ một hàm để thay thế:

```
> setTimeout(function() { myFunction("I love You !!!"); }, 3000);
```

```
function myFunction(value) {  
  console.log(value);  
}
```

```
< 58
```

```
I love You !!!
```

➤ Trong đó:

- `function(){ myFunction("I love You !!!"); }` được dùng như là một callback
- `3000` là số mili giây trước khi time-out, và `myFunction()` sẽ được gọi sau 3s

# Callback, Asynchronous



## ❑ Bất đồng bộ – Asynchronous:

➤ Khi sử dụng hàm `setInterval()`, bạn có thể chỉ định một hàm gọi lại (`callback`) được thực thi cho mỗi khoảng thời gian (`interval`):

```
> setInterval(myFunction, 1000);  
  
function myFunction() {  
  let d = new Date();  
  console.log(d.getHours() + ":" + d.getMinutes() + ":" + d.getSeconds());  
}
```

⏪ 73

11:27:29

11:27:30

11:27:31

➤ Trong đó:

- `myFunction` được dùng như là một `callback`
- `myFunction` được truyền vào hàm `setInterval()` như một đối số
- `1000` là số mili giây trước khi giữa mỗi lần `interval`, và `myFunction()` sẽ được gọi sau mỗi giây

# Promise, Async/Await

---



❑ Đối tượng lời hứa – **Promise Object**:

➤ Đối tượng lời hứa (**Promise Object**) chứa cả **producing code** và gọi tới **consuming code**.

➤ Trong đó:

- **producing code**: là code dùng có thể mất 1 chút thời gian
- **consuming code**: là code dùng phải chờ một kết quả nào đó



# Promise, Async/Await



## ❑ Đối tượng lời hứa – Promise Object:

➤ Đối tượng lời hứa (**Promise Object**) chứa cả **producing code** và gọi tới **consuming code**.

➤ Cú pháp:

```
> let myPromise = new Promise(function(myResolve, myReject) {  
  // "Producing Code" (May take some time)
```

```
  myResolve(); // when successful  
  myReject();  // when error  
});
```

```
// "Consuming Code" (Must wait for a fulfilled Promise)  
myPromise.then(  
  function(value) { /* code if successful */ },  
  function(error) { /* code if some error */ }  
);
```

```
< ▶ Promise {<fulfilled>: undefined}
```

➤ Khi **producing code** chứa kết quả, nó sẽ gọi một trong 2 **callback** sau:

Result	Call
Success	myResolve(result value)
Error	myReject(error object)



# Promise, Async/Await



## ❑ Các thuộc tính của đối tượng lời hứa – Promise Object Properties:

- Đối tượng lời hứa (**Promise Object**) hỗ trợ 2 thuộc tính: **state** và **result**
- Đối tượng lời hứa (**Promise Object**) có thể có các trạng thái (**state**):
  - Pending -> khi **state** là “**pending**” (working), **result** là **undefined**.
  - Fulfilled -> khi **state** là “**fulfilled**”, **result** là một **value**.
  - Rejected -> khi **state** là “**rejected**”, **result** là một đối tượng **error**.

myPromise.state	myPromise.result
“ <b>pending</b> ”	Chưa được định nghĩa <b>undefined</b>
“ <b>fulfilled</b> ”	Một giá trị <b>value</b>
“ <b>rejected</b> ”	Một đối tượng <b>error</b>

# Promise, Async/Await

---



## ❑ Cách dùng Promise:

```
myPromise.then(  
  function(value) { /* code if successful */ },  
  function(error) { /* code if some error */ }  
);
```

- **Promise.then()** có 2 đối số, một hàm **callback** cho trường hợp thành công, và một hàm **callback** khác cho trường hợp thất bại.
- Cả 2 đều tùy chọn, bạn có thể thêm chỉ một hàm callback cho trường hợp thành công hoặc cho trường hợp thất bại.

# Promise, Async/Await



## ❑ Cách dùng Promise - Ví dụ:

```
> function myDisplayer(some) {  
  console.log(some);  
}
```

```
let myPromise = new Promise(function(myResolve, myReject) {  
  let x = 0;  
  
  // some code (try to change x to 5)  
  if (x == 0) {  
    myResolve("OK");  
  } else {  
    myReject("Error");  
  }  
});
```

```
myPromise.then(  
  function(value) {myDisplayer(value);},  
  function(error) {myDisplayer(error);}  
);
```

OK

```
< ▼ Promise {<fulfilled>: undefined} ⓘ  
  ► [[Prototype]]: Promise  
    [[PromiseState]]: "fulfilled"  
    [[PromiseResult]]: undefined
```

>

```
> function myDisplayer(some) {  
  console.log(some);  
}
```

```
let myPromise = new Promise(function(myResolve, myReject) {  
  let x = 5;  
  
  // some code (try to change x to 5)  
  if (x == 0) {  
    myResolve("OK");  
  } else {  
    myReject("Error");  
  }  
});
```

```
myPromise.then(  
  function(value) {myDisplayer(value);},  
  function(error) {myDisplayer(error);}  
);
```

Error

```
< ▼ Promise {<fulfilled>: undefined} ⓘ  
  ► [[Prototype]]: Promise  
    [[PromiseState]]: "fulfilled"  
    [[PromiseResult]]: undefined
```

>

# Promise, Async/Await



## ❑ Cách dùng Promise – với Timeout:

### ➤ Khi chỉ dùng Callback với Timeout:

```
> setTimeout(function() { myFunction("I love You !!!"); }, 3000);
```

```
function myFunction(value) {  
  console.log(value);  
}
```

```
< 72
```

```
I love You !!!
```

### ➤ Khi dùng Promise với Timeout:

```
> const myPromise2 = new Promise(function(myResolve, myReject) {  
  setTimeout(function(){ myResolve("I love You !!"); }, 3000);  
});
```

```
myPromise2.then(function(value) {  
  console.log(value);  
});
```

```
< ▼ Promise {<pending>} 1
```

```
  ► [[Prototype]]: Promise  
    [[PromiseState]]: "pending"  
    [[PromiseResult]]: undefined
```

```
I love You !!
```

# Promise, Async/Await

---



❑ Từ khóa `async/await`:

➤ Từ khóa `async` và `await` làm cho `promises` dễ viết hơn.

- `async`: làm một hàm trả về một Promise
- `await`: làm một hàm đợi một Promise



# Promise, Async/Await



## ❑ Từ khóa `async/await`:

- Từ khóa `async`: trước một hàm tạo hàm trả về một `promise`.

<pre>&gt; async function myFunction() {   return "Hello"; }</pre>	giống →	<pre>&gt; function myFunction() {   return Promise.resolve("Hello"); }</pre>
---	---------	--

- Ví dụ sử dụng `async`:

<pre>&gt; function myDisplayer(some) {   console.log(some); }</pre> <pre>async function myFunction() {return "Hello";}</pre> <pre>myFunction().then(   function(value) {myDisplayer(value);},   function(error) {myDisplayer(error);} );</pre> <hr/> <pre>Hello</pre> <hr/> <pre>&lt; ▶ Promise {&lt;fulfilled&gt;: undefined}</pre>	đơn giản hơn →	<pre>&gt; function myDisplayer(some) {   console.log(some); }</pre> <pre>async function myFunction() {return "Hello";}</pre> <pre>myFunction().then(   function(value) {myDisplayer(value);} );</pre> <hr/> <pre>Hello</pre> <hr/> <pre>&lt; ▶ Promise {&lt;fulfilled&gt;: undefined}</pre>
--	----------------	---

# Promise, Async/Await



## ❑ Từ khóa `async/await`:

➤ Từ khóa `await`: trước một hàm tạo hàm chờ một `promise`.

```
> let value = await promise;
```

\*Lưu ý: Từ khóa `await` chỉ có thể dùng bên trong hàm `async`

➤ Ví dụ sử dụng `await`:

```
> async function myDisplay() {  
  let myPromise = new Promise(function(resolve, reject) {  
    resolve("I love You !!");  
  });  
  console.log(await myPromise);  
}
```

```
myDisplay();
```

```
I love You !!
```

```
< ▼ Promise {<fulfilled>: undefined} ⓘ  
  ► [[Prototype]]: Promise  
    [[PromiseState]]: "fulfilled"  
    [[PromiseResult]]: undefined
```

không cần reject

```
> async function myDisplay() {  
  let myPromise = new Promise(function(resolve) {  
    resolve("I love You !!");  
  });  
  console.log(await myPromise);  
}
```

```
myDisplay();
```

```
I love You !!
```

```
< ▼ Promise {<fulfilled>: undefined} ⓘ  
  ► [[Prototype]]: Promise  
    [[PromiseState]]: "fulfilled"  
    [[PromiseResult]]: undefined
```

# Promise, Async/Await



❑ Từ khóa `async/await` – với Timeout:

```
> async function myDisplay() {  
  let myPromise = new Promise(function(resolve) {  
    setTimeout(function() {resolve("I love You !!");}, 3000);  
  });  
  console.log(await myPromise);  
}
```

```
myDisplay();
```

```
< ▼ Promise {<pending>} ⓘ  
  ► [[Prototype]]: Promise  
    [[PromiseState]]: "fulfilled"  
    [[PromiseResult]]: undefined
```

```
I love You !!
```

```
>
```





# Tóm tắt bài học

---

- Tính năng Modules trong ES6
- Default Parameter, Rest Parameter
- Spread Operator, Template Literals
- Enhanced Object Properties
- Destructuring Assignment
- Callback, Asynchronous
- Promise, Async/Await