

HOMEWORK

FRESHER TRAINING COURSE

JS – FUNCTION

FRESHER TRAINING COURSE

TABLE OF CONTENT

- ▶ 1. Function parameters
- ▶ 2. Callback
- ▶ 3. Scope
- ▶ 4. Closure
- ▶ 5. Context
- ▶ 6. Change the context
- ▶ 7. Arrow function
- ▶ 8. Render something.

PARAMETERS – DEFAULT PARAMETERS

► In ES5

```
function multiply(a, b) {  
  b = (typeof b !== 'undefined') ? b : 1  
  return a * b  
}
```

```
multiply(5, 2) // 10  
multiply(5)    // 5
```

PARAMETERS – DEFAULT PARAMETERS

► In ES6

```
function multiply(a, b = 1) {  
  return a * b  
}
```

```
multiply(5, 2)           // 10  
multiply(5)              // 5  
multiply(5, undefined)   // 5
```

PARAMETER – CAREFUL WITH OBJECT PARAMETER

- ▶ JS object always access by the reference.

```
1 var myCar = { brand: 'Honda', c
2 console.log(myCar)
3 function myFunc(theObject) {
4     theObject.brand = 'Toyota';
5 }
6 let lastBrand = myCar.brand;
7
```

PARAMETERS – ARGUMENTS

- ▶ An Array-like object accessible inside function
- ▶ Contains the values of the arguments passed to that function.

```
1 function myFunc(a, b, c) {  
2     console.log(arguments);  
3 }  
4  
5 myFunc()  
6 myFunc(1, 2, 3)
```

REST PARAMETERS

- ▶ Allow an indefinite number of arguments as an array

```
1 function myFunc(...list) {  
2     console.log(list);  
3 }  
4  
5 myFunc()  
6 myFunc(1, 2, 3)
```


CALLBACK

- ▶ The function passed into another function as an argument,

- ▶

```
function greeting(name) {  
    alert('Hello ' + name);  
}  
  
function processUserInput(callback) {  
    var name = prompt('Please enter your name. ');  
    callback(name);  
}  
  
processUserInput(greeting);
```

SCOPE – GLOBAL

- ▶ The scope is the portion of code where the variable is visible
- ▶ Global Scope: just have one global scope in javascript.

```
index.html ×
src > index.html > html > body > script
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Document</title>
8  </head>
9  <body>
10     <script src="common.js"></script>
11     <script src="index.js"></script>
12 </body>
13 </html>
```

```
JS common.js ×
src > JS common.js > common
1  var common = {name: "common"}
```

```
JS index.js ×
src > JS index.js
1  console.log(common);
2  // {name: "common"}
```

SCOPE – LOCAL

- ▶ Variables defined inside a function are in the local scope.
- ▶ They have a different scope for every call of that function.
- ▶ Can using variables with the same name in different functions.

SCOPE – LOCAL

- ▶ Each function have own different local scope when it execute.

```
1 // Global Scope
2 function foo() {
3     // Local Scope #1
4     function bar() {
5         // Local Scope #2
6     }
7 }
8 // Global Scope
9 function baz() {
10     // Local Scope #3
11 }
12 // Global Scope
```

LEXICAL SCOPE

- ▶ The inner functions have access to the variables and other resources of their parent scope

```
1 function foo(){
2     // parent scope
3     var _foo = 'foo';
4     function bar(){
5         console.log(_foo);
6         // childen scope included parent scope
7     }
8     bar();
9 }
10
```

BLOCK SCOPE

► From ES6: we have scope inside { } with let, const

►

```
1 if (true) {  
2     var name = 'Kame';  
3     const age = 28;  
4     let weight = 75;  
5     // name is still in the global scope  
6     // age and weight just in the block scope  
7 }  
8  
9 console.log(name); // logs 'Hammad'  
10 console.log(age); // Error: age is not defined  
11 //console.log(weight); // Error: weight is not defined
```

EXAMPLE

- ▶ 1. Write the function count down time.
 - ▶ Input: $n < \text{integer} >$
- ▶ 2. Creating a clock.
 - ▶ The output will come every second.

CLOSURES

- ▶ Closures are all accessible variables when the function been created.
 - ▶ own scope,
 - ▶ the parents' scope,
 - ▶ the global scope.
 - ▶ the arguments of the outer function
- ▶ Can access the variables even after the function has returned.

CLOSURES

- Closures can access the variables even after the function has returned.

```
1 function greet(vname) {  
2     vname = 'Alexx';  
3     return function () {  
4         console.log('Hi ' + vname);  
5     }  
6 }  
7 let greeting = greet("Alexx");  
8 greeting(); // logs 'Hi Alexx'
```

CONTEXT – WHAT IS THIS?

- ▶ *Context* refer to the value of `this`.
- ▶ *Scope* refer to the visibility of variables.

```
1 var player = {  
2   name: "Alexx",  
3   log: function() {  
4     console.log(this.name);  
5   }  
6 }  
7 var log = player.log;  
8 player.log(); // Alexx  
9 log()
```

CONTEXT – WHEN IS THE “THIS” BE SET

- ▶ *The context will then be set when called function, not when defined function*

```
1 var player = {  
2     name: "kame",  
3     showInfo: function () {  
4         console.log("name", this.name);  
5     }  
6 }  
7 player.showInfo();  
8  
9 const button = document.getElementById("BtnLog");  
10 button.addEventListener("mousedown", player.showInfo);
```

CONTEXT – HOW “THIS” WORK?

- ▶ *The value of this depend on how the function call.*

In a method, `this` refers to the **owner object**.

Alone, `this` refers to the **global object**.

In a function, `this` refers to the **global object**.

In a function, in strict mode, `this` is `undefined`.

In an event, `this` refers to the **element** that received the event.

Methods like `call()`, and `apply()` can refer `this` to **any object**.

CONTEXT – CHANGE CONTEXT WITH .CALL(), .APPLY()

▶ *The context can be change while calling function.*

▶ *call(), apply(): borrowing methods.*

▶

```
1 const user = {
2   name: "user01",
3   showInfo: function(){
4     console.log(this.name) // user01
5   }
6 }
7 const player = {
8   name: "player01",
9 }
10 user.showInfo.call(player); // player01
```

CONTEXT – CHANGE CONTEXT WITH .BIND()

- ▶ *.bind()* create a new function with the context we bind.

```
1 var player = {  
2     name: "kame",  
3     showInfo: function () {  
4         console.log("name", this.name);  
5     }  
6 }  
7 player.showInfo();  
8  
9 const button = document.getElementById("BtnLog");  
10 button.addEventListener("mousedown", player.showInfo.bind(player));
```

ARROW FUNCTION

- ▶ *Does not have its own this*
- ▶ *Do not have arguments*
- ▶ *Can not using with call, bind, apply.*
- ▶ *Can not used as contractors*
- ▶ *And should not be used as object methods*

```
1 let min = function(a, b){  
2     return a > b ? a : b;  
3 }  
4  
5 let max = (a, b) => a > b ? a : b;  
6
```


ARROW FUNCTION

```
1 function Renderer() {  
2     this.index = 0;  
3     var self = this;  
4     setInterval(function render() {  
5         self.index++;  
6         console.log(self.index);  
7     }, 17);  
8 }  
9 var renderer = new Renderer();
```

```
1 function Renderer() {  
2     this.index = 0;  
3     setInterval(() => {  
4         this.index++;  
5         console.log(this.index);  
6     }, 17);  
7 }  
8 var renderer = new Renderer();
```


ASSIGNMENT

- ▶ *The logic of pair game*
 - ▶ *Concept: Have 20 cards with 10 different image.*
 - ▶ *open 2 cards each time,*
 - ▶ *if they are matched, hide them, get 1000 coin.*
 - ▶ *if they are not, close them, loss 500 coin.*
 - ▶ *first coin is 10.000.*
 - ▶ *if coin $< 0 \Rightarrow$ game over.*



ASSIGNMENT

▶ *Aim*

- ▶ *Try to render game objects with dom*
- ▶ *familiar with array and function*
- ▶ *try to using event click.*
- ▶ *Try to write the game logic*



INSTRUCTION – PLAYING WITH DOM

- ▶ *Image*

- ▶ *`document.createElement("img");`*

- ▶ *Text*

- ▶ *Just using div*

- ▶ *Position*

- ▶ *Using basic style css*

- ▶ *Event:*

- ▶ *`addEventListener`*

SUMMARY

SUMMARY

- ▶ 1. Be careful with object parameter
- ▶ 2. Context - what is this? Depend on how function call.
- ▶ 3. Closure - all accessible variables when the function created.
- ▶ 4. The scope of inner function included the scope of outer function.
- ▶ 5. Arrow function do not have own this. And can not using in many cases.
- ▶ 6. Remember the use of .call, bind, .apply. It is really basic but important.

Q&A