GDSC - FullStack Program - 2

Variable

Austin

Variable Declarations (var let const)

1. const - It is very similar the const in c++. Once a variable is declared using const, it cannot be reassigned.

2. let - It is very similar a regular declaration in c++.

3. var - The var keyword was used in all JavaScript code from 1995 to 2015. The var keyword should only be used in code written for older browsers.[1]

[1] The note of the variable

Variable Declarations (var let const)

```
1   // let
2   let a = 1 // we can modify this variable
3   console.log("before modifying a:", a)
4   a = 2
5   console.log("after modifying a:", a)
6
7   // const
8   const b = 1 // we can't modify this variable
9   console.log("before modifying b:", b)
10  // b = 2 // this will throw an error
11  // console.log("after modifying b:", b)
12
13  // var
14  var c = 1 // we can modify this variable
15  console.log("before modifying c:", c)
16  c = 2
17  console.log("after modifying c:", c)
```

```
zhangjunshideMacBook-Air:
before modifying a: 1
after modifying a: 2
before modifying b: 1
before modifying c: 1
after modifying c: 2
```

When to Use var, let, or const?

1. Always declare variables
2. Always use `const` if the value should not be changed
3. Always use `const` if the type should not be changed (Arrays and Objects)
4. Only use `let` if you can't use `const`
5. Only use `var` if you MUST support old browsers.

[1] The note of the variable

Hoisting - var

1. Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function).

[1] Hoisting

Hoisting - var

```
1   // Hoisting example
2   x1 = 5;
3
4   console.log("x1 = ", x1);
5   var x1;
6
7   // It is equivalent to:
8   var x2;
9   x2 = 5;
10  console.log("x2 = ", x2);
```

```
zhangjunshideMacBook-Air:js_folder abao$ node example.js
x1 =  5
x2 =  5
```

[1] Hoisting

Hoisting - let and const

1. Variables defined with let and const are hoisted to the top of the block, but not initialized. The block of code is aware of the variable, but it cannot be used until it has been declared.

2. Using a let variable before it is declared will result in a ReferenceError. The variable is in a "temporal dead zone" from the start of the block until it is declared

[1] Hoisting

Hoisting - let and const

```
1    // Hoisting for let and const
2    a = 10;
3    console.log(a); // undefined
4    let a;
```

```
zhangjunshideMacBook-Air:js_folder abao$ node example.js
/Users/abao/vscode/GDSC-FullStack-Training/js_folder/example.js:2
a = 10;
  ^

ReferenceError: Cannot access 'a' before initialization
```

[1] Hoisting

Hoisting - let and const

```
1    console.log(a); // undefined
```

```
zhangjunshideMacBook-Air:js_folder abao$ node example.js
/Users/abao/vscode/GDSC-FullStack-Training/js_folder/example.js:2
console.log(a); // undefined
            ^

ReferenceError: a is not defined
```

[1] Hoisting

Scope, Scope Chain[1]

1. JavaScript variables have 3 types of scope:
    - Block scope
    - Function scope
    - Global scope
2. let and const support 3 types of scope, but var only support function scope and global scope.

[1] Scope

The example of three types of scopes

```
1   var globalVar = "Global";  // 全域作用域
2
3   function testScopes() {
4       var functionVar = "Function";  // 函數作用域
5
6       if (true) {
7           let blockVar = "Block";  // 區塊作用域
8           console.log(blockVar);   // 輸出 "Block"
9       }
10      // console.log(blockVar);  // 會拋出錯誤，因為 blockVar 是區塊作用域變數
11      console.log(functionVar);   // 輸出 "Function"
12      console.log(globalVar);     // 輸出 "Global"
13  }
14
15  testScopes();
16  console.log(globalVar);  // 輸出 "Global"
```

```
Block
Function
Global
Global
```

[1] Scope

Use var in the block scope

```
{
    var globalVar = "I am a global variable";
}
✨

console.log(globalVar);    // 會輸出 "I am a global variable"
```

```
zhangjunshideMacBook-Air:js_folder abao$ node example.js
I am a global variable
```

[1] Scope

Scope, Scope Chain[1]

1. The scope chain is a list of all of the scopes that are available to the current scope. The current scope is always at the beginning of the scope chain.

   The JavaScript interpreter searches for variables in the following order:

- The current scope
- The outer scope
- The global scope

[1] Scope chain

The example of scope chain

```
1   var myVar = "Global";  // 全域作用域
2
3   function outerFunction() {
4       var myVar = "Outer";  // 函數作用域 (outerFunction)
5
6       function innerFunction() {
7           var myVar = "Inner";  // 函數作用域 (innerFunction)
8
9           console.log(myVar);   // 1. 輸出 "Inner" (在當前範圍找到)
10      }
11
12      innerFunction();
13      console.log(myVar);  // 2. 輸出 "Outer" (在 outerFunction 的範圍找到)
14  }
15
16  outerFunction();
17  console.log(myVar);  // 3. 輸出 "Global" (在全域範圍找到)
```

```
Inner
Outer
Global
```

Summary[1]

	Scope	Redeclare	Reassign	Hoisted
var	No	Yes	Yes	Yes
let	Yes	No	Yes	No
const	Yes	No	No	No

[1] The difference between var, let and const

Data Type

Max

Basic types

Number

```
console.log(typeof 10); // number
console.log(typeof 3.14); // number
```

String

```
console.log(typeof 'hihi'); // string
```

Boolean

```
console.log(typeof true); // boolean
console.log(typeof false); // boolean
```

Number

```
console.log(1 + 2 + 3); // 6
console.log(10 * 10); // 100
console.log(2 ** 4); // 16

console.log(1 / 0); // Infinity
console.log(-1 / 0); // -Infinity
console.log(0 / 0); // NaN

console.log(typeof Infinity); // number
console.log(typeof -Infinity); // number
console.log(typeof NaN);      // number
```

String

```
console.log("I'm a string");
console.log('I\'m also a string');
console.log(`I'm a string too`);

console.log(`hello
world
!`)
```

```
hello
world
!
```

String

```
const username = 'Justin';
console.log('Hello ' + username); // Hello Justin
console.log(`Hello ${username}`); // Hello Justin
```

Boolean

```
console.log(true && false); // false
console.log(true || false); // true
```

Special types

```
let a;
console.log(a); // undefined

let b = null;
console.log(b); // null
```

Equality operator

Strict

```
console.log(10 === 10); // true
console.log(123 === '123'); // false
console.log(null === undefined); // false
```

Loose (convert to same type if they are different)

```
console.log(10 == 10); // true
console.log(123 == '123'); // true
console.log(null == undefined); // true
```

Object

```
const player1 = {
  name: 'Justin',
  attack: 100,
  defense: 50
};
console.log(player1.name); // Justin
```

Array

```
const arr = [1, 2, 3, 4];
console.log(arr[0]); // 1
```

```
const todoList = [
  {
    title: 'Learn JS',
    done: true,
  },
  {
    title: 'Learn React.js',
    done: false,
  },
  {
    title: 'Learn Next.js',
    done: false,
  },
];
console.log(todoList[1].title); // Learn React.js
```

Javascript Closure

Name: SGarry

JavaScript Clousre

- JavaScript Clousre is the combination of a function and the Lexical Environment within which that function was declared [1].
- In short, Closure is a function which can store and access the available variables in current environment.

[1] Closures - JavaScript | MDN (mozilla.org)

Lexical Environment

- Lexical Environment consists of
  - Environment Record
    - a specification type used to define the association of Identifiers to specific variables and functions [2]
  - Outer reference of Environment Record
    - Environment Record from parent or global environment
- A function will try to find variables or function in own Environment Record, if it can’t find, it will try to find them from outer.

[2] ECMAScript® 2024 Language Specification (ecma-international.org)

Rule of Accessing Variables

- In JavaScript, inner scope can access the variables from outer scope, but outer scope can’t access inner scope.

```
1  function init(){
2      let name = "John"
3      function displayName(){
4          console.log(name)
5      }
6      displayName()
7  }
8  init()
```

▲ Inner function can access outer variable

```
1  function init(){
2      console.log(name)
3      function displayName(){
4          let name = "John"
5      }
6      displayName()
7  }
8  init()
```

▲ outer function can’t access inner variable

Retention of Environments

- Closure have a feature that it can keep the lexical environment.
- As long as the reference of inner environment exists, the outer one won't disappear.

```
1   function printName() {
2       let name = "John"
3       function displayName() {
4           console.log(name);
5       }
6       return displayName;
7   }
8
9   let func = printName();
10  func();
```

▲Retention of Environments of printName() [3]

[3] [教學] JavaScript Closure (閉包) - Shubo 的程式開發筆記

Independent Environments

- Each function will keep their own environment, which means each of them will keep their reference.

```
1   function makeAdder(x) {
2       return function (y) {
3           return x + y;
4       };
5   }
6
7   const add5 = makeAdder(5);
8   const add10 = makeAdder(10);
9
10  console.log(add5(2)); // 7
11  console.log(add10(2)); // 12
```

▲ The example of producing 2 lexical enviroments [1]

Practical Example of Closure

- Following is an example of Emulating private methods with closures.
- Counter will keep variable as private, Outer environment can’t access.

Private variable using closure [1] ▶

```
const counter = (function () {
    let privateCounter = 0;
    function changeBy(val) {
        privateCounter += val;
    }

    return {
        increment() {
            changeBy(1);
        },
        decrement() {
            changeBy(-1);
        },
        value() {
            return privateCounter;
        },
    };
})();

console.log(counter.value()); // 0.

counter.increment();
counter.increment();
console.log(counter.value()); // 2.

counter.decrement();
console.log(counter.value()); // 1.
```

Performance of Closure

- Javascript closure lets function manages own scope and closure, which leads to the worse performance if closures are not needed for a particular function.
- For instance, when creating a new object/class, methods should normally be associated to the object's prototype rather than defined into the object constructor. [1]

```
function MyObject(name, message) {
    this.name = name.toString();
    this.message = message.toString();
    this.getName = function () {
        return this.name;
    };

    this.getMessage = function () {
        return this.message;
    };
}
```

▲ function within a object [1]

```
function MyObject(name, message) {
    this.name = name.toString();
    this.message = message.toString();
}
MyObject.prototype = {
    getName() {
      return this.name;
    },
    getMessage() {
      return this.message;
    },
};
```

▲ function prototype within a object [1]

“function”

Joanne

Arrow function

Definition:

A more concise syntax for writing functions.

Benefit:

No need for the `function` keyword or braces.

“this” binding to outer scope.

Arrow function

Syntax:

No parameters:

```
const greet = () => console.log('Hello!');
greet();
```

Single parameter:

```
const square = x => x * x;
console.log(square(4));
```

Arrow function

Syntax:

Multiple parameters:

```
const multiply = (a, b) => a * b;
console.log(multiply(3, 5));
```

Multiple lines:

```
const add = (a, b) => {
    const result = a + b;
    return result;
};
```

Arrow function

Use Cases:

Short callback functions:

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(num => num * 2);
console.log(doubled);
```

filter:

```
const numbers = [1, 2, 3, 4, 5, 6];
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers);  // output [2, 4, 6]
```

Arrow function

Use Cases:

Reduce:

```
const numbers = [1, 2, 3, 4, 5];
const total = numbers.reduce((sum, num) => sum + num, 0);
console.log(total);
```

Combine with setInterval:

Arrow function

Use Cases:

Combine with setInterval / setTimeout:

```
function Timer() {
    this.seconds = 0;
    setInterval(() => {
      this.seconds++;
      console.log(this.seconds);
    }, 1000);
  }

  const timer = new Timer();
```

Higher-Order function

Definition:

Can take another function as an argument or return a function.

Benefit:

Allowing for more modular and flexible code.

Higher-Order function

Original:

```
const numbers = [1, 2, 3, 4];
const doubled = [];

for (let i = 0; i < numbers.length; i++) {
  doubled.push(numbers[i] * 2);
}

console.log(doubled);  // Output: [2, 4, 6, 8]
```

Higher-order function:

```
const numbers = [1, 2, 3, 4];

// taking a function as an argument
const doubled = numbers.map(num => num * 2);

console.log(doubled);  // Output: [2, 4, 6, 8]
```

Higher-Order function

Returning a Function:

```
function createMultiplier(multiplier) {
    return function (num) {
      return num * multiplier;
    };
  }

  const double = createMultiplier(2);
  const triple = createMultiplier(3);

  console.log(double(5));  // 10
  console.log(triple(5));  // 15
```

Higher-Order function

Combining Function as Argument and Return Value:

```
function performOperation(operation) {
    return function(a, b) {
      return operation(a, b);
    };
  }

  const add = (a, b) => a + b;
  const subtract = (a, b) => a - b;

  const addOperation = performOperation(add);
  const subtractOperation = performOperation(subtract);

  console.log(addOperation(5, 3));      // 8
  console.log(subtractOperation(5, 3)); // 2
```

“this”

Kai

“this”

Definition (General)

"this" refers to the owner object of the currently executing function.

How is it more complex than cpp?

"this" behaves differently depending on where and how it is invoked (e.g., in a function, method, or globally).

“this”

1. Under an object

   “this” refers to the object itself

```
const obj = {
  value: 1,
  hello: function() {
    console.log(this.value)
  }
}

obj.hello() // 1
```

```
class Car {
  setName(name) {
    this.name = name
  }

  getName() {
    return this.name
  }
}

const myCar = new Car()
myCar.setName('hello')
console.log(myCar.getName()) // hello
```

“this”

2. In Global Context

- In strict mode: undefined
- In node.js: global
- In the browser: the window object

Edit the value of “this”

1. Call & Apply

Overwrite the value for “this”.

```
'use strict';
function hello(a, b){
  console.log(this, a, b)
}

hello.call('yo', 1, 2) // yo 1 2
hello.apply('hihihi', [1, 2]) // hihihi 1 2
```

2. Bind

Define “this” permanently

```
'use strict';
function hello() {
  console.log(this)
}

const myHello = hello.bind('my')
myHello.call('call') // my
```

“this”

3. When function is also an object

The function itself cannot also be the owner object, so its “this” refers to global.

4. Arrow function

Its “this" follows the object/function where it’s declared.

```
1   var x = 1;
2   var obj = {
3       x: 20,
4       fn: function(){
5           console.log(this.x)
6           var test = function() {
7               console.log(this.x)
8           }
9           test()
10      }
11  }
12  obj.fn()
```

“this”

3. When function is also an object

The function itself cannot also be the owner object, so its “this” refers to global.

4. Arrow function

Its “this" follows the object/function where it’s declared.

```
1   const obj = {
2       x: 1,
3       hello: function(){
4           const test = () => {
5               console.log(this.x)
6           }
7           test()
8       }
9   }
10
11  obj.hello()
12  const hello = obj.hello
13  hello()
```

How to figure out what “this” refers to?

Remember: "this" refers to the owner object of the currently executing function.

```
const obj = {
  value: 1,
  hello: function() {
    console.log(this.value)
  }
}

obj.hello() // 1
const hey = obj.hello
hey() // undefined
```

```
const obj = {
  value: 1,
  hello: function() {
    console.log(this.value)
  },
  inner: {
    value: 2,
    hello: function() {
      console.log(this.value)
    }
  }
}

const obj2 = obj.inner
const hello = obj.inner.hello
obj.inner.hello()
obj2.hello()
hello()
```

```
const obj = {
  value: 1,
  hello: function() {
    console.log(this.value)
  },
  inner: {
    value: 2,
    hello: function() {
      console.log(this.value)
    }
  }
}

const obj2 = obj.inner
const hello = obj.inner.hello
```

```
obj.inner.hello() // obj.inner.hello.call(obj.inner) => 2
obj2.hello() // obj2.hello.call(obj2) => 2
hello() // hello.call() => undefined
```

Application:

html

```html
<button class="toggle-btn">Button 1</button>
<button class="toggle-btn">Button 2</button>
<button class="toggle-btn">Button 3</button>
```

javascript

```javascript
document.querySelectorAll('.toggle-btn').forEach(function(button) {
  button.addEventListener('click', function() {
    this.classList.toggle('active');
    if (this.classList.contains('active')) {
      this.textContent = 'Clicked!';
    } else {
      this.textContent = 'Click Me';
    }
  });
});
```

Reference:

[1]:https://blog.huli.tw/2019/02/23/javascript-what-is-this/

[2]:https://kuro.tw/posts/2017/10/12/What-is-THIS-in-JavaScript-%E4%B8%8A/

JS - Review 10/27

1 Loops and Iterations & Array(Austin)

- for … of loop
- for … in loop
- What are the different array traversal methods in JavaScript?

JS - Review 10/27

2. Control flow (Joanne)

- if … else
- Conditional Operators
- Ternary Operators
- throw statement
- try/catch/finally
- error objects

JS - Review 10/27

3. Keyed Collections (Kai)

- Map v.s. Weak Map
- Set v.s. Weak Set
- What is difference between Map and Object?

JS - Review 10/27

4. Prototypal Inheritance(Max)

- Prototype chain

JS - Review 10/27

5. Shallow Copy& Deep Copy(Garry)

JS - Review 10/27

6. Asynchronous JavaScript