Advanced JavaScript

Junghoo Cho

cho@cs.ucla.edu

Variable Scope (Global vs Local)

- A variable declared outside of any block has *global scope*.
- A variable declared with `let` inside a block is valid only withi block: *block-scope local variable*
- A variable used without an explicit `let` declaration has *globa*
  - ***This is strongly discouraged***

Scope Example

```
let a = "a"; // global vs local?
b = "b"; // global vs local?

function f()
{
    c = "c"; // global vs local?
    let d = "d"; // global vs local?
}
```

`let` vs `var`

- `let` was introduced in ES6
- Before `let`, `var` was used with the following difference
  - `var` has function scope (as opposed to block scope)
  - `var` is hoisted (vs no hoisting)
    - declaration is “moved” to the top of its scope
  - read this article to learn more on hoisting
- Use of `let` produces cleaner code. Use it!

var Example

```
var a = 10;  // global vs local?
function f() {
    b = 10;  // global vs local?
    console.log(b);
    var b;
}
f();
console.log(b);
```

Function Object

- In Javascript, functions are objects!
    - Functions can be assigned to a variable
    - Functions can be passed as a parameter
    - Functions can have properties
- But
    - A function is an object type (according to the standard), but `typeof function` not `object`

Function Object Example

```
function square(x) { return x*x; };

function myfunc(x, func) {
    return func(x);
}

myfunc(10, square);
myfunc(10, function (x) { return x * 2; }); // anonymous functio

myfunc.a = 20;
```

Nested Function

- Functions can be defined inside a function!

```
function outer_function() {
    console.log("a")
    function inner_function() {
        console.log(1);
    }
    console.log("b");
    inner_function();
}
outer_function(); // what will be printed?
```

Variable Scope in Nested Function

- Variables in a nested function follow *lexical scope* (not dynan

```
function f() {
    let a = 1;
    let b = 2;
    function g() {
        console.log(b);  // b = ?
        b = 3;
    }
    if (a > 0) {
        let b = 4;
        g();
        console.log(b); // b = ?
    }
    console.log(b);     // b = ?
}
f(); // what will be printed?
```

Nested Function and Closure (1)

- Nested functions can be returned and be called later!
- Example: What will happen?

```
function getFunc() {
    function printUCLA() { console.log("UCLA"); }
    return printUCLA;
}

let func = getFunc();
func();
```

Nested Function and Closure (2)

- *Closure*: When a nested function references non-local variabl returned, it is “bundled together” with the referenced variab

```
function getFunc() {
    let age = 10;
    function printAge() { console.log(++age); }
    return printAge;
}

let func = getFunc();
func(); func(); // what will be printed?
```

- `printAge()` does not have its own local variable, but the ret `printAge` carries the `age` variable from its surrounding conte

Nested Function and Closure (3)

- Q: What will be printed?

```
function getFunc() {
    let age = 10;
    function printAge() { console.log(++age); }
    return printAge;
}

let myFunc1 = getFunc();
myFunc1(); myFunc1();

let myFunc2 = getFunc();
myFunc2(); myFunc2();
```

Nested Function and Closure (4)

- Closures can be used to simulate local variables and functior
  - Avoids polluting global namespace
  - Used *extensively* especially before ES6
- Example

```
(function() {
    var count = 0;
    function helper() {
        console.log(`Help called ${++count} times!`);
    }
    helper();
    // ...
    helper();
})()  // create an anonymous function and call it immediately
```

  - The above code “simulates” block-local scope for `count` and `helper(`

Arrow Function (1)

- In JavaScript, we often have to pass a function as a paramete

```
function ChangeColor(event) {
    document.body.style.color = "red";
}
document.body.addEventListener("click", ChangeColor);
```

- Polluting namespace can be avoided using anonymous func

```
document.body.addEventListener("click", function(event) {
    document.body.style.color = "red";
});
```

Arrow Function (2)

- Arrow function makes this even more concise

```
document.body.addEventListener("click", (event) => {
    document.body.style.color = "red";
});
```

- *Arrow function expression*

```
(param1, ...) => expression
(param1, ...) => { statements; }
```

  - `() => expression` returns the value of `expression`
  - `() => { statements; }` should return a value explicitly

Object-Oriented Programming (OOP)

- Object = data + method
- A “method” can be added to an object in JavaScript
  - Example

    ```
    let o = { x: 10 };
    o.multiply = function (v) { this.x *= v; }

    o.multiply(5);
    console.log(o.x);
    ```

    - Inside an object’s method, `this` points to the object itself
- **Important**: Arrow functions should not be used as an object
  - Arrow function is primarily to be passed as a parameter
  - More on this later

Class in ECMAScript 2015

- ES6 introduced cleaner syntax to define a class
- Example

```
class Shape {
    constructor(color) {  // constructor() is class constructo
        this.color = color;
    }
    info() { return "color: " + this.color; }
    whoami() {
        console.log("I am a Shape with " + this.info());
    }
};

s = new Shape('blue');
s.whoami();
```

Class Inheritance

```
class Rectangle extends Shape {
    constructor(color, x, y) {
        super(color);  // super refers to the parent class
        this.x = x;
        this.y = y;
    }
    info() {
        return `${super.info()}, x: ${this.x}, y: ${this.y}`;
    }
};
let r = new Rectangle("red", 2, 3);
r.whoami();
```

- Internally, class inheritance is implemented via `prototype` ol
    - To learn the detail, read MDN document on inheritance and prototy

Optional Chaining (ECMAScript 2020)

- If a vaiable is `undefined` or `null`, we get an error

```
let obj;
console.log(obj.name); // Error: obj is undefined!
```

- Checking for the error is ugly

```
let obj;
console.log(obj ? obj.name : undefined);
```

- Instead of throwing an error, *optional chaining operator* retur `undefined`:

```
let obj;
console.log(obj?.name);  // returns undefined
```

Keyword `this`

- Unfortunately, the meaning of `this` is a source of great confu bug in JavaScript
- Three bindings of `this`
  1. In a function called via object/class method
     - `this` = called object/class
  2. In a function called via event triggering
     - `this` = DOM element to which event handler was set
  3. Everywhere else (in top-level block or in other function calls)
     - `this` = the global object

`this` in Event Handling Call

- When called via event triggering, `this` binds to the DOM ele where the handler is set
- Example

```
<body id="body_id">
    ...
</body>
<script>
    document.body.addEventListener("click", function (event) {
        console.log(this.id); // what does this bind to?
    });
</script>
```

`this` in Other Places

- If `this` is used in other than class method or event handler, `t` to the *global object*
- *Global object* (`globalThis` in ES2020)
  - In browser, `window` object
  - In Node.js, `global` object
  - Any variable assigned without declaration becomes a property of th object

Arrow Function and this Binding

- Arrow function (`() => {}`) ***does not*** provide its own `this` bir
  - It retains the `this` binding of the enclosing lexical context
- Example

```
<body id="body_id">
    ...
</body>
<script>
    document.body.addEventListener("click", (event) => {
        console.log(this.id); // what does this bind to?
    });
</script>
```

10
10
10
20
10

Tricky Example of this

```
x = 10;

function_printx = function() { console.log(this.x); };
arrow_printx = () => { console.log(this.x); };

o = { x: 20 };
o.printx_f = function_printx;
o.printx_a = arrow_printx;

// What will be printed?
console.log(this.x);
function_printx();
arrow_printx();
o.printx_f();
o.printx_a();
```

Notes on `this`

- The binding of `this` changes *dynamically* depending on hov function is called
    - This "dynamic scoping" makes `this` confusing and hard to understa to many bugs
- Use `this` only in object/class method
- ***Never*** use arrow functions to define an object/class method

Array Manipulation

- Mutator vs Accessor
  - *Mutator*: modifies input array ***in-place***
    - `reverse`, `sort`, `push`, `pop`, `shift`, `unshift`, `splice`
  - *Accessor*: input array stays in tact
    - `concat`, `slice`, `filter`, `map`
    - ***A new output array is created and returned***

Array Manipulation Example (1)

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

Array Manipulation Example (2)

```
let a = [1, 2, 3];
let b = a.reverse();       // reverse is a mutator
console.log(b);
a[1] = 6;
console.log(b);

a = [1, 2, 3];
b = a.concat([4, 5]);   // concat is an accessor
console.log(b);
a[1] = 6;
console.log(b);
```

Destructuring Assignment

```
let o = { userid: 10, password: "secret" };
const { userid, password, email = "default_email" } = o;
// userid = 10, password = "secret", email = "default_email"

let a = [1, 2, 3, 4];
let [a1, a2, ...rest] = a;
// a1 = 1, a2 = 2, rest = [3, 4]
```

ES Module

- ECMAScript 2015 added support for modules
  - Similar to Java “packages”
  - One JavaScript file ↔ one module
  - Everything in a module stays local unless declared `export`
  - `export`ed entities can be `import`ed and used by other JavaScript co

(Multiple) Named Export

```
//------ lib.js ------
export function square(x) {
    return x * x;
}
export function dist(x, y) {
    return Math.sqrt(square(x) + square(y));
}

//------ main.js ------
import { square } from './lib.js';
square(11);

//----- main2.js ------
import * as mylib from './lib.js';
mylib.dist(4, 3);
```

(Single) Default Export

```
//------ lib.js ------
export default function () { ... }

//------ main1.js ------
import myFunc from './lib.js';
myFunc();
```

- Remark:
  - No `{ }` to import `default` export
  - `{ }` to import named export (even if we import just one)

References

- Javascript: The Definitive Guide by David Flanagan
  - Strongly recommended if you plan to code in JavaScript extensively
- ECMAScript standard: ECMA 262 https://www.ecma-international.org/ecma-262/
  - The ultimate reference on what is really correct
  - But very boring to read and learn from
  - Browser support is a few generations behind