JavaScript ES 6 review

CS568 – Web Application Development I

Computer Science Department

Maharishi International University

Maharishi International University - Fairfield, Iowa

Content

- let/const/var
- Arrow Functions
- The “this” keyword
- Call/apply/bind
- import/export
- Spread / Destructuring
- Class
- promise/async/await
- Array Functions

Let & Const

- **Best practice** is always start with const. If you need to change its value, then make it let.
- Use const for JSON objects.
- When using Let and Const, the variable is only available in the **block** it’s defined in.
- **Const** is a signal that the identifier won’t be reassigned. Most cases, const is objects. You can reassign a new value for its attributes!
- **Let** is a signal that the variable may be reassigned, such as a counter in a loop.

var

```
The task is to print 1 to 5 every second. What will it print?

(function timer() {
  for (var i=1; i<=5; i++) {
    setTimeout(function clog() {console.log(i)}, i*1000);
  }
})();
```

What is happening under the hood

- Var becomes a function-scoped variable.
- setTimeout is provided by the browser or NodeJS environment. Executed asynchronously after the for loop. The loop set the variable “i” to 6. Remember, at the end of the for loop it is “i++”.
- Then the setTimeout gets popped out and prints “6” five times.

Solution I

```
(function timer() {
  for (let i=1; i<=5; i++) {
    setTimeout(function clog() {console.log(i)}, i*1000);
  }
})();
```

Solution II

Alternatively, developers used to fix this bug before ES6 using immediately called functions. Immediately called functions also give data privacy.

```
(function timer() {
  for (var j = 1; j <= 5; j++) {
    (function () {
      var i = j;
      setTimeout(function clog() { console.log(i); }, i * 1000);
    }());
  }
})();
```

Global variable

- It can be accessed anywhere
- It is not recommended

```
y = 2;
console.log(window.y);
```

Arrow Functions

- Were introduced in ES6.
- Allow us to write shorter function syntax.
- An arrow function is not just a syntactic sugar. There are differences between a regular function and arrow function. For example, the **“this”** keyword.
- Always use arrow functions in React class-based components.

Arrow Functions Syntax

```
(param1, param2) => { statements } ;
(param1, param2) => expression; // An expression is any valid
unit of code that resolves to a value.
// Parentheses are optional when there's only one parameter
name:
(singleParam) => { statements }
singleParam => { statements }
// The parameter list for a function with no parameters
// should be written with a pair of parentheses.
() => { statements }
```

Arrow Functions Examples

If the function has only one statement, and the statement **returns** a value, you can remove the brackets and the return keyword.

```
const sayHello = (name) => {
  return 'Hello ' + name;
}
```

```
const sayHello = name =>{
  return 'Hello ' + name;
}
```

```
const sayHello = name => 'Hello ' + name;
```

The “this” keyword in regular function

“This” is the object that owns the function in JavaScript.

```
const test = {
  prop: 42,
  func: function() {
    return this.prop;
  },
};
console.log(test.func());
```

// this will be determined by the object which called the function in the regular function

The “this” in Arrow Functions

- In regular functions, the **this** keyword represents the object that called the function, which could be the window, the document, a button or whatever.
- An arrow function does not have **this** keyword, it will borrow **this** from surrounding code (lexical scope).

What is the output?

```
const obj = {
x: 1,
f: function(){
function innerFunction(){
console.log(this.x);
}
innerFunction();
}
}
obj.f();
```

```
const obj = {
x: 1,
f: function(){
const innerFunction  = () => {
console.log(this.x);
}
innerFunction();
}
}
obj.f();
```

call/bind/apply methods

- These methods allow you specify the “this”.
- One of the use case is to borrow a function from another object and carry a function. The first parameter is the “this” object and the rest is the other parameters to the function.
- **Call** – Takes parameters one by one

call(targetObj, param1, param2...)

- **Apply** – Takes parameters as an array

apply(targetObj, [param1, param2...])

- **Bind** – Returns another function with the giving object as the “this”.

bind(targetObj, param1, param2...)

call/bind/apply methods

```
const car1 = {
color: "car1",
setColor(color){
this.color = color;
}
}
const car2 = {
color: ""
}
```

call/bind/apply methods

```
let f = car1.setColor.bind(car2);
f('car2');
console.log(car2.color, car1.color);

car1.setColor.call(car2, 'car3');
console.log(car2.color, car1.color);

car1.setColor.apply(car2, ['car4']);
console.log(car2.color, car1.color);
```

Exports

The export statement is used when creating JavaScript modules to export live bindings to functions, objects, or primitive values from the module so they can be used by other programs with the import statement

There are two types of exports:

1. Named Exports (Zero or more exports per module)
2. Default Exports (One per module)

Named Exports

- Named exports are useful to export several values.
- During the import, it is mandatory to use the same name of the corresponding object.

Named Exports

```
export let fname, lname;
export let fname = 'umur', lname = 'inan';
export function functionName(){...}
export class ClassName {...}
// Export list
export { fname, lname };
// Renaming exports
export { fname as firstname, lname as lastname };
```

Default Exports

```
export default expression;
export default function (…) { … }
export default function name1(…) { … }
export { name1 as default };
```

Import

import statement is used to import read only live bindings which are exported by another module.

Import

```
import defaultExport from "module-name";

import { export1 } from "module-name";

import { export1 as alias1 } from "module-name";

import { export1 , export2 } from "module-name";
```

Exports and Imports

```
//student.js
const student = {
  name : 'bob'
}
export default student;
```

```
//helper.js
export const minutesInHour = 60;
export const sayHi = () =>
'Hello';
```

```
//app.js
import student from './student.js';
import stu from './student.js'
import {minutesInHour} from './helper.js';
import {sayHi as tellHi} from './helper.js';
```

Require vs import

- Require for importing modules in NodeJS is built on top of the "Common JS". The Common JS is for organizing code in JS in a modular fashion. Later, ECMAScript standardized JS modularity using import/export.
- Require was born before the import. Now it standardized. Starting node 18, you can use import.
- Require can be used inside if/else conditions just like an expression or a variable where import is only used on top of the JS file.
- "Require" imports module synchronously whereas the "import" can import asynchronously.

Class

- Classes are a template for creating objects. They encapsulate data with code to work on that data.
- Classes in JS are built on prototypes. Prototypes are the mechanism by which JavaScript objects inherit features from one another.
- Classes are in fact "special functions", and just as you can define a function.
- An important difference between function declarations and class declarations is that function declarations are hoisted and class declarations are not.

Constructor

- The constructor method is a special method for creating and initializing an object created with a class.
- There can only be one special method with the name "constructor" in a class.
- A constructor can use the super keyword to call the constructor of the super class.

Class Example

```
class Student extends Person {
 constructor(name){
   super();
   this.name = name;
 }
 sayHi = () => 'Hi ' + this.name;
}
const student1 = new Student('Bob');
console.log(student1.sayHi());
```

Extends

- The extends keyword is used in class declarations or class expressions to create a class as a child of another class.
- If there is a constructor present in the subclass, it needs to first call super() before using "this".

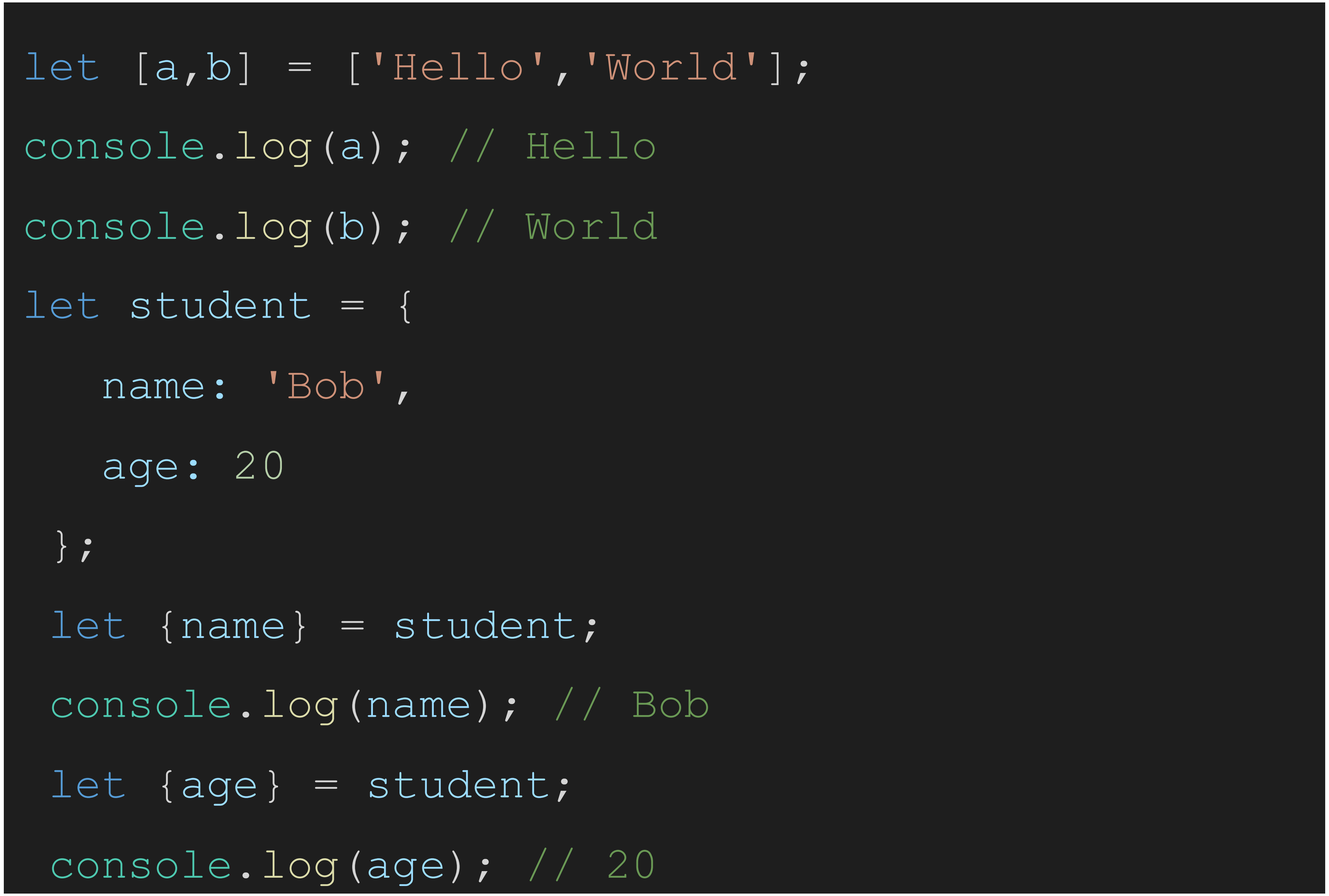
Spread Operator (...)

Used to split up array elements or object properties. So we can expand, copy an array, or clone an object.

```
function sum(x, y, z) { return x + y + z; }
const numbers = [1, 2, 3];
console.log(sum(...numbers));

let obj1 = { foo: 'bar', x: 42 };
let clonedObj = { ...obj1 };
let mergedObj = { ...obj1, ...obj2 };
```

Destructuring

- Extract array elements or object properties into variables.
- Spread operator takes all the elements or all the properties whereas destructuring pulls out single element or single property to variables.



```
let [a,b] = ['Hello','World'];
console.log(a); // Hello
console.log(b); // World
let student = {
   name: 'Bob',
   age: 20
 };
 let {name} = student;
 console.log(name); // Bob
 let {age} = student;
 console.log(age); // 20
```

Shorthand syntax

if you want to define an object who's keys have the same name as the variables passed-in as properties, you can use the shorthand and simply pass the key name.

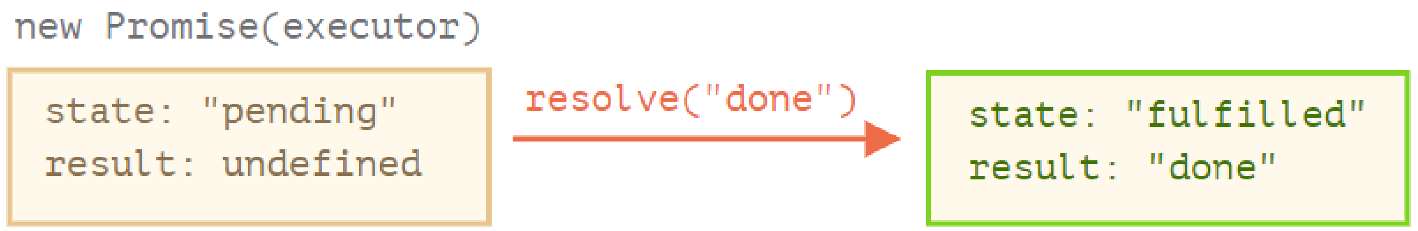
```
let cat = 'Miaow';
let dog = 'Woof';
let bird = 'Peet peet';

let someObject = {
  cat~~: cat~~,
  dog~~: dog~~,
  bird~~: bird~~
}

console.log(someObject);
```

Promise

- A promise is an object that represents something that will be available in the future. In programming, this "something" is values.
- Promises propose that instead of waiting for the value we want (e.g. the image download), we receive something that represents the value in that instant so that we can "get on with our lives" and then at some point go back and use the value generated by this promise.
- Promises are based on time events and have some states that classify these events:
    - Pending: still working, the result is undefined;
    - Fulfilled: when the promise returns the correct result, the result is a value.
    - Rejected: when the promise does not return the correct result, the result is an error object.

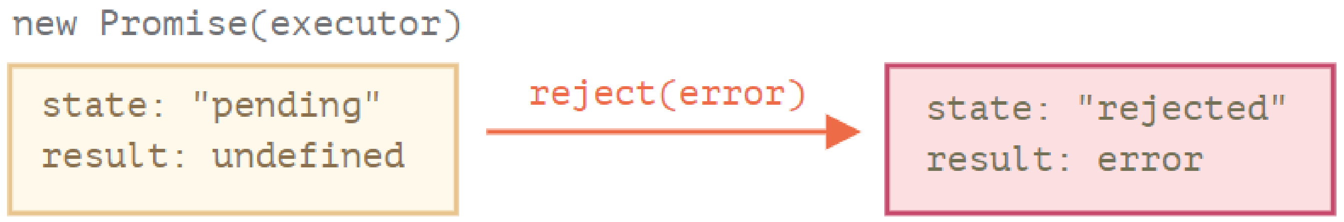
Promise

```
let promise = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed
  // after 1 second signal that the job is done with the result "done"
  setTimeout(() => resolve("done"), 1000);
});
```



Promise

```
let promise = new Promise(function (resolve, reject) {
  // after 1 second signal that the job is finished with an error
  setTimeout(() => reject(new Error("Whoops!")), 1000);
})
```



Async function

- `async` can be placed before a function.
- An `async` function always returns a `promise`:
  - When no return statement defined, or return without a value. It turns a resolving a promise equivalent to `return Promise.Resolve()`
  - When a return statement is defined with a value, it will return a resolving promise with the given return `value`, equivalent to `return Promise.Resolve(value)`
  - When an error is thrown, a rejected promised will be returned with the thrown error, equivalent to `return Promise.Reject(error)`

Async function

```
console.log('start');
async function f() {
    return 1;
}
f().then(console.log);
console.log('end');
```

Await

- The keyword `await` makes JavaScript wait until that `promise` settles and returns its result.
- `await` literally suspends the function execution until the `promise` settles, and then resumes it with the `promise` result. That doesn’t cost any CPU resources, because the JavaScript engine can do other jobs in the meantime: execute other scripts, handle events, etc.
- It’s just a more elegant syntax of getting the `promise` result than `promise.then`.

Await

```
console.log('start');
async function foo() {
    return 'done!';
}
async function bar() {
    console.log('inside bar - start');
    let result = await foo();
    console.log(result); // "done!"
    console.log('inside bar - end');
}
bar();
console.log('end');
```

Array methods: map, filter, reduce

```
//functional programming:  map, filter, reduce can replace many loops

const a = [1,3,5,3,3];

//translate/map all elements in an array to another set of values
const b = a.map(function(elem, i, array) {
  return elem + 3;})// [4,6,8,6,6]

//select elements based on a condition
const c = a.filter(function(elem, i, array){
  return elem !== 3;});//[1,5]

//find first element or index of first element satisfying condition
const d = a.find(function(elem) {return elem > 1;}); //3
const e = a.findIndex(function(elem) {return elem > 1;}); //1

//find a cumulative or concatenated value based on elements across the array
const f = a.reduce(function(accumulator, currentValue, currentIndex, array){
  return prevVal + elem;}); //15
```

Summary

- Variables: var/let/count
- Arrow function
- This keyword: bind/call/apply
- Export/import
- Class: constructor, super, extends
- Speading / destructuring
- Promise/async/await
- Array methods: map, filter, reduce, find, findIndex