

**Module 3 Labs**

## Getting ready.

1. Copy and paste this file into your local se-nz-pt-a-4jul22 repository > your own folder.
2. Commit your new files with the message “Lab six setup complete”.
3. Feel free to use any JS runtime environment to write and test your answers. (Node js, HTML Script tag, or even your Chrome dev tools)
4. Once you write and test your code copy and paste your answers in this document.
5. Once you finish your work,
   1. Commit your work with message “Lab six completed”.
   2. Push your work to se-nz-pt-a-4jul22. (don’t forget to pull first)
6. If you are having any difficulties, contact your trainers by sending slack message or zoom chat.

## Lab exercise.

1. **Fundamentals**

1. What are the results of these expressions?

"" + 1 + 0

"" - 1 + 0

true + false

6 / "3"

"2" \* "3"

4 + 5 + "px"

"$" + 4 + 5

"4" - 2

"4px" - 2

" -9 " + 5

" -9 " - 5

null + 1

undefined + 1

" \t \n" - 2

1. Here’s a code that asks the user for two numbers and shows their sum.

It works incorrectly. The output in the example below is 12 (for default prompt values).

Why? Fix it. The result should be 3.

let a = prompt("First number?", 1);

let b = prompt("Second number?", 2);

alert(a + b); // 12

1. What will be the result for these expressions?

5 > 4

"apple" > "pineapple"

"2" > "12"

undefined == null

undefined === null

null == "\n0\n"

null === +"\n0\n"

1. Will an alert be shown?

if ("0") {

alert( 'Hello' );

}

1. Rewrite this if using the conditional operator '?':

let result;

if (a + b < 4) {

result = 'Below';

} else {

result = 'Over';

}

1. Write the delay method with arrow function, delay(func, ms)

Should work like:

const hello = ( who )=> console.log(‘Hello ’ + who );

const delalyHello = delay(hello, 300);

delayHello(‘world’);

1. Write the function isEmpty(obj) which returns true if the object has no properties, false otherwise.

Should work like:

let schedule = {};

alert( isEmpty(schedule) ); // true

schedule["8:30"] = "get up";

alert( isEmpty(schedule) ); // false

1. There’s a ladder object that allows to go up and down

let ladder = {

step: 0,

up() {

this.step++;

},

down() {

this.step--;

},

showStep: function() { // shows the current step

console.log( this.step );

}

};

Modify the code of up, down and showStep to make the calls chainable, like this:

ladder.up().up().down().showStep(); // 1

1. Create New Accumulator

Create a constructor function Accumulator(startingValue).

Object that it creates should:

* Store the “current value” in the property value. The starting value is set to the argument of the constructor startingValue.
* The read() method should use prompt to read a new number and add it to value.

In other words, the value property is the sum of all user-entered values with the initial value startingValue.

Here’s the demo of the code:

let accumulator = new Accumulator(1); // initial value 1

accumulator.read(); // adds the user-entered value

accumulator.read(); // adds the user-entered value

console.log(accumulator.value); // shows the sum of these values

**2. Intermediate**

1. Uppercase the first character

Write a function ucFirst(str) that returns the string str with the capitalized first character, for instance:

ucFirst("john") == "John";

1. Truncate the text

Create a function truncate(str, maxlength) that checks the length of the str and, if it exceeds maxlength – replaces the end of str with the ellipsis character "…", to make its length equal to maxlength.

The result of the function should be the truncated (if needed) string.

For instance:

truncate("What I'd like to tell on this topic is:", 20) = "What I'd like to te…"

truncate("Hi everyone!", 20) = "Hi everyone!"

1. Array operations

Let’s try 5 array operations.

1. Create an array styles with items “Jazz” and “Blues”.
2. Append “Rock-n-Roll” to the end.
3. Replace the value in the middle by “Classics”. Your code for finding the middle value should work for any arrays with odd length.
4. Strip off the first value of the array and show it.
5. Prepend Rap and Reggae to the array.

The array in the process:

Jazz, Blues

Jazz, Blues, Rock-n-Roll

Jazz, Classics, Rock-n-Roll

Classics, Rock-n-Roll

Rap, Reggae, Classics, Rock-n-Roll

1. Translate border-left-width to borderLeftWidth

Write the function camelize(str) that changes dash-separated words like “my-short-string” into camel-cased “myShortString”.

That is: removes all dashes, each word after dash becomes uppercased.

Examples:

camelize("background-color") == 'backgroundColor';

camelize("list-style-image") == 'listStyleImage';

camelize("-webkit-transition") == 'WebkitTransition';

1. Create an extendable calculator

Create a constructor function Calculator that creates “extendable” calculator objects.

The task consists of two parts.

First, implement the method calculate(str) that takes a string like "1 + 2" in the format “NUMBER operator NUMBER” (space-delimited) and returns the result. Should understand plus + and minus -.

Usage example:

**let calc = new Calculator();**

**alert( calc.calculate("3 + 7") ); // 10**

Then add the method addMethod(name, func) that teaches the calculator a new operation. It takes the operator name and the two-argument function func(a,b) that implements it.

For instance, let’s add the multiplication \*, division / and power \*\*:

let powerCalc = new Calculator();

powerCalc.addMethod("\*", (a, b) => a \* b);

powerCalc.addMethod("/", (a, b) => a / b);

powerCalc.addMethod("\*\*", (a, b) => a \*\* b);

let result = powerCalc.calculate("2 \*\* 3");

alert( result ); // 8

* No parentheses or complex expressions in this task.
* The numbers and the operator are delimited with exactly one space.
* There may be error handling if you’d like to add it.
* Create an extendable calculator

1. Filter unique array members

Let arr be an array.

Create a function unique(arr) that should return an array with unique items of arr.

For instance:

function unique(arr) {

/\* your code \*/

}

let values = ["Hare", "Krishna", "Hare", "Krishna",

"Krishna", "Krishna", "Hare", "Hare", ":-O"

];

alert( unique(values) ); // Hare, Krishna, :-O

P.S. Here strings are used, but can be values of any type.

1. Iterable keys

We’d like to get an array of map.keys() in a variable and then apply array-specific methods to it, e.g. .push.

But that doesn’t work:

let map = new Map();

map.set("name", "John");

let keys = map.keys();

// Error: keys.push is not a function

keys.push("more");

Why? How can we fix the code to make keys.push work?

1. Store "unread" flags

There’s an array of messages:

let messages = [

{text: "Hello", from: "John"},

{text: "How goes?", from: "John"},

{text: "See you soon", from: "Alice"}

];

Your code can access it, but the messages are managed by someone else’s code. New messages are added, old ones are removed regularly by that code, and you don’t know the exact moments when it happens.

Now, which data structure could you use to store information about whether the message “has been read”? The structure must be well-suited to give the answer “was it read?” for the given message object.

P.S. When a message is removed from messages, it should disappear from your structure as well.

P.P.S. We shouldn’t modify message objects, add our properties to them. As they are managed by someone else’s code, that may lead to bad consequences.

1. Sum the properties

There is a salaries object with arbitrary number of salaries.

Write the function sumSalaries(salaries) that returns the sum of all salaries using Object.values and the for..of loop.

If salaries is empty, then the result must be 0.

For instance:

let salaries = {

"John": 100,

"Pete": 300,

"Mary": 250

};

alert( sumSalaries(salaries) ); // 650

1. The maximal salary

There is a salaries object:

const salaries = {

"John": 100,

"Pete": 300,

"Mary": 250

};

Create the function topSalary(salaries) that returns the name of the top-paid person.

* If salaries is empty, it should return null.
* If there are multiple top-paid persons, return any of them.

P.S. Use Object.entries and destructuring to iterate over key/value pairs.

1. How many seconds have passed today?

Write a function getSecondsToday() that returns the number of seconds from the beginning of today.

For instance, if now were 10:00 am, and there was no daylight savings shift, then:

getSecondsToday() == 36000 // (3600 \* 10)

The function should work in any day. That is, it should not have a hard-coded value of “today”.

1. Exclude backreferences

In simple cases of circular references, we can exclude an offending property from serialization by its name.

But sometimes we can’t just use the name, as it may be used both in circular references and normal properties. So we can check the property by its value.

Write replacer function to stringify everything, but remove properties that reference meetup:

let room = {

number: 23

};

let meetup = {

title: "Conference",

occupiedBy: [{name: "John"}, {name: "Alice"}],

place: room

};

// circular references

room.occupiedBy = meetup;

meetup.self = meetup;

alert( JSON.stringify(meetup, function replacer(key, value) {

/\* your code \*/

}));

/\* result should be:

{

"title":"Conference",

"occupiedBy":[{"name":"John"},{"name":"Alice"}],

"place":{"number":23}

}

\*/

**3. Advanced**

1. Are counters independent?

Here we make two counters: counter and counter2 using the same makeCounter function.

Are they independent? What is the second counter going to show? 0,1 or 2,3 or something else?

function makeCounter() {

let count = 0;

return function() {

return count++;

};

}

let counter = makeCounter();

let counter2 = makeCounter();

alert( counter() ); // 0

alert( counter() ); // 1

alert( counter2() ); // ?

alert( counter2() ); // ?

1. Set and decrease counter?

Modify the code of makeCounter() so that the counter can also decrease and set the number:

* counter() should return the next number (as before).
* counter.set(value) should set the counter to value.
* counter.decrease() should decrease the counter by 1.

See the sandbox code for the complete usage example.

P.S. You can use either a closure or the function property to keep the current count. Or write both variants.

1. Output every second

Write a function printNumbers(from, to) that outputs a number every second, starting from from and ending with to.

Make two variants of the solution.

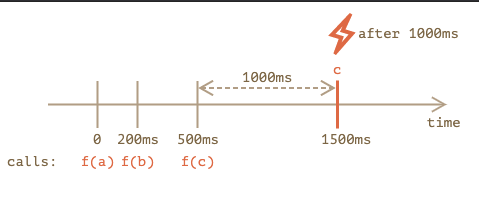
1. Using setInterval.
2. Using nested setTimeout.
3. Debounce decorator

The result of debounce(f, ms) decorator is a wrapper that suspends calls to f until there’s ms milliseconds of inactivity (no calls, “cooldown period”), then invokes f once with the latest arguments.

In other words, debounce is like a secretary that accepts “phone calls”, and waits until there’s ms milliseconds of being quiet. And only then it transfers the latest call information to “the boss” (calls the actual f).

For instance, we had a function f and replaced it with f = debounce(f, 1000).

Then if the wrapped function is called at 0ms, 200ms and 500ms, and then there are no calls, then the actual f will be only called once, at 1500ms. That is: after the cooldown period of 1000ms from the last call.



…And it will get the arguments of the very last call, other calls are ignored.

Here’s the code for it (uses the debounce decorator from the [Lodash library](https://lodash.com/docs/4.17.15#debounce)):

let f = \_.debounce(alert, 1000);

f("a");

setTimeout( () => f("b"), 200);

setTimeout( () => f("c"), 500);

// debounced function waits 1000ms after the last call and then runs: alert("c")

1. Partial application for login

What should we pass askPassword in the code below, so that it calls user.login(true) as ok and user.login(false) as fail?

function askPassword(ok, fail) {

let password = prompt("Password?", '');

if (password == "rockstar") ok();

else fail();

}

let user = {

name: 'John',

login(result) {

alert( this.name + (result ? ' logged in' : ' failed to log in') );

}

};

askPassword(?, ?); // ?

Your changes should only modify the highlighted fragment.

1. Searching algorithm

The task has two parts.

Given the following objects:

let head = {

glasses: 1

};

let table = {

pen: 3

};

let bed = {

sheet: 1,

pillow: 2

};

let pockets = {

money: 2000

};

* Use \_\_proto\_\_ to assign prototypes in a way that any property lookup will follow the path: pockets → bed → table → head. For instance, pockets.pen should be 3 (found in table), and bed.glasses should be 1 (found in head).
* Answer the question: is it faster to get glasses as pockets.glasses or head.glasses? Benchmark if needed.

1. Create an object with the same constructor

Imagine, we have an arbitrary object obj, created by a constructor function – we don’t know which one, but we’d like to create a new object using it.

Can we do it like that?

let obj2 = new obj.constructor();

Give an example of a constructor function for obj which lets such code work right. And an example that makes it work wrong.

1. Add the decorating "defer()" to functions

Add to the prototype of all functions the method defer(ms), that returns a wrapper, delaying the call by ms milliseconds.

Here’s an example of how it should work:

function f(a, b) {

alert( a + b );

}

f.defer(1000)(1, 2); // shows 3 after 1 second

Please note that the arguments should be passed to the original function.

1. Add toString to the dictionary

There’s an object dictionary, created as Object.create(null), to store any key/value pairs.

Add method dictionary.toString() into it, that should return a comma-delimited list of keys. Your toString should not show up in for..in over the object.

Here’s how it should work:

let dictionary = Object.create(null);

// your code to add dictionary.toString method

// add some data

dictionary.apple = "Apple";

dictionary.\_\_proto\_\_ = "test"; // \_\_proto\_\_ is a regular property key here

// only apple and \_\_proto\_\_ are in the loop

for(let key in dictionary) {

alert(key); // "apple", then "\_\_proto\_\_"

}

// your toString in action

alert(dictionary); // "apple,\_\_proto\_\_"

1. Extended clock

We’ve got a Clock class. As of now, it prints the time every second.

class Clock {

constructor({ template }) {

this.template = template;

}

render() {

let date = new Date();

let hours = date.getHours();

if (hours < 10) hours = '0' + hours;

let mins = date.getMinutes();

if (mins < 10) mins = '0' + mins;

let secs = date.getSeconds();

if (secs < 10) secs = '0' + secs;

let output = this.template

.replace('h', hours)

.replace('m', mins)

.replace('s', secs);

console.log(output);

}

stop() {

clearInterval(this.timer);

}

start() {

this.render();

this.timer = setInterval(() => this.render(), 1000);

}

}

Create a new class ExtendedClock that inherits from Clock and adds the parameter precision – the number of ms between “ticks”. Should be 1000 (1 second) by default.

* Your code should be in the file extended-clock.js
* Don’t modify the original clock.js. Extend it.

1. Inherit from SyntaxError

Create a class FormatError that inherits from the built-in SyntaxError class.

It should support message, name and stack properties.

Usage example:

let err = new FormatError("formatting error");

alert( err.message ); // formatting error

alert( err.name ); // FormatError

alert( err.stack ); // stack

alert( err instanceof FormatError ); // true

alert( err instanceof SyntaxError ); // true (because inherits from SyntaxError)

1. Delay with a promise - not mandatory

The built-in function setTimeout uses callbacks. Create a promise-based alternative.

The function delay(ms) should return a promise. That promise should resolve after ms milliseconds, so that we can add .then to it, like this:

function delay(ms) {

// your code

}

delay(3000).then(() => alert('runs after 3 seconds'));

1. Rewrite using async/await - not mandatory

Rewrite this example code from the chapter Promises chaining using async/await instead of .then/catch:

​​function loadJson(url) {

return fetch(url)

.then(response => {

if (response.status == 200) {

return response.json();

} else {

throw new Error(response.status);

}

});

}

loadJson('no-such-user.json')

.catch(alert); // Error: 404