Front-end Advanced

Training Assignment

Document Code	**25e-BM/HR/HDCV/FSOFT**
Version	**1.1**
Effective Date	**7/1/2019**

Hanoi, mm/yyyy

RECORD OF CHANGES

No	Effective Date	Change Description	Reason	Reviewer	Approver
1	30/May/2019	Create a new assignment	Create new	DieuNT1	VinhNV
2	07/Jun/2019	Update Fsoft Template	Update	DieuNT1	VinhNV

Contents

FRESHER ACADEMY	CODE:	Async-JS.M.A101 (Async 01)
	TYPE:	Medium
	LOC:	300
	DURATION:	180

Day 1. Unit 1-2: Async Programming

Objectives:

- Understand how to abstract asynchronous task in JavaScript (Async Programming)
- Understand the core concept of Async in JavaScript: the Event Loop
- Understand the concept of Parallel Programming and how JavaScript implement that concept
- Able to working with Async API with ease (Callback, Promise, async/await)

Problem Descriptions:

Exercise 01

The setTimeout() function is commonly used if you wish to run your function a specified number of milliseconds from when the setTimeout() method was called. The general syntax of the method is:

```
setTimeout(expression, timeout);
```

where expression is the JavaScript code to run after timeout milliseconds have elapsed.

Use **setTimeOut** to log **Hello Fresher Academy** after 1000ms.

Exercise 02

The setInterval() function, as the name suggests is commonly used to set a delay for functions that are executed again and again like animations. The setInterval() function is very closely related to setTimeout() - they even have same syntax:

```
setInterval(expression, interval);
```

The only difference is:

setTimeout() *triggers the* expression *only once while* setInterval() *keeps triggering* expression*regularly after the given interval of time. (unless you tell it to stop).*

Use **setInterval** to log **Hello Fresher Academy repeatedly after every 3 seconds.**

Exercise 03

Given the following code, what will be the output ? And Explain why ?

```
(function() {
    console.log(1);
    setTimeout(function(){console.log(2)}, 1000);
    setTimeout(function(){console.log(3)}, 0);
    console.log(4);
})();
```

Exercise 04

What order log messages take in the following cases? And Explain why ?

Case 1

```
console.log("A");
setTimeout(function() { console.log("B"); }, 0);
setTimeout(function() { console.log("C"); }, 0)
console.log("D");
```

Case 2

```
setTimeout(function() {
  setTimeout(function() {
    console.log('A');
  }, 0);
}, 0);

setTimeout(function() {
  console.log('B');
}, 0);

setTimeout(function() {
  setTimeout(function() {
    console.log('C');
  }, 0);
}, 10);

setTimeout(function() {
  console.log('D');
}, 0);
```

Case 3

```
var x = 'A';

setTimeout(function() {
  console.log(x);
  x = 'B';
}, 3);

setTimeout(function() {
  console.log(x);
  x = 'C';
}, 2);

setTimeout(function() {
  console.log(x);
  x = 'D';
}, 1);

setTimeout(function() {
  console.log(x);
}, 4);
```

Case 4

```
var t1 = setTimeout(function() {
  console.log('A');
  setTimeout(function() {
    console.log('B');
  }, 0);
}, 100);

var t2 = setTimeout(function() {
  console.log('C');
  setTimeout(function() {
    console.log('D');
  }, 0);
}, 200);

clearTimeout(t1);

setTimeout(function() {
  clearTimeout(t2);
}, 250);
```

Exercise 05

Async calls, just as sync, can be structured and composed differently. You should be able to predict the sequence precisely without even running the code (running it in your mind). Always think about frames (event loop iterations) in which a particular code has to be executed.

What order log messages take in the following cases?

Case 1

```
function logA() {
  console.log('A');
}

function logB() {
  console.log('B');
}

function logC() {
  console.log('C');
}

function logD() {
  console.log('D');
}

logD(logA(logB(logC())));
```

Case 2

```
function alogA() {
  setTimeout(function() {
    console.log('A');
  }, 0);
}

function alogB() {
  setTimeout(function() {
    console.log('B');
  }, 0);
}

function alogC() {
  setTimeout(function() {
    console.log('C');
  }, 0);
}

function alogD() {
  setTimeout(function() {
    console.log('D');
  }, 0);
}

alogD(alogA(alogB(alogC())));
```

Case 3

```
setTimeout(function() {
  console.log('A');
  setTimeout(function() {
    console.log('B');
    setTimeout(function() {
      console.log('C');
      setTimeout(function() {
        console.log('D');
      }, 0);
    }, 100);
  }, 200);
}, 300);
```

Case 4

```
setTimeout(function() {
  console.log('A');
  setTimeout(function() {
    console.log('B');
  }, 100);
}, 200);

setTimeout(function() {
  console.log('C');
  setTimeout(function() {
    console.log('D');
  }, 200);
}, 100);
```

Exercise 06

Given the following code, what will be the output ?

```
for (var i = 0; i < 5; i++) {
    setTimeout(function() { console.log(i); }, i * 1000 );
}
```

Exercise 07

Given an array of promises, Promise.all returns a promise that waits for all of the promises in the array to finish. It then succeeds, yielding an array of result values. If a promise in the array fails, the promise returned by all fails too, with the failure reason from the failing promise.

Implement something like this yourself as a regular function called Promise_all.

Remember that after a promise has succeeded or failed, it can’t succeed or fail again, and further calls to the functions that resolve it are ignored. This can simplify the way you handle failure of your promise.

```
function Promise_all(promises) {
  return new Promise((resolve, reject) => {
    // Your code here.
  });
}

// Test code.
Promise_all([]).then(array => {
  console.log("This should be []:", array);
});
function soon(val) {
  return new Promise(resolve => {
    setTimeout(() => resolve(val), Math.random() * 500);
  });
}
Promise_all([soon(1), soon(2), soon(3)]).then(array => {
  console.log("This should be [1, 2, 3]:", array);
});
Promise_all([soon(1), Promise.reject("X"), soon(3)])
  .then(array => {
    console.log("We should not get here");
  })
  .catch(error => {
    if (error != "X") {
      console.log("Unexpected failure:", error);
    }
  });
```