Javascript Module Exercises

1. Determine what this Javascript code will print out (without running it):

```
 x = 1;
 var a = 5;
 var b = 10;
 var c = function(a, b, c) {
                  document.write(x);
                  document.write(a);
                  var f = function(a, b, c) {
                                    b = a;
                                    document.write(b);
                                    b = c;
                                    var x = 5;
                                }
                  f(a,b,c);
                  document.write(b);
                  var x = 10;
              }
c(8,9,10);
document.write(b);
document.write(x);
}
```

➔ undefined 8 8 9 10 1

2. Define *Global Scope* and *Local Scope* in Javascript.

➔ Any variable declared outside the function belongs to global Scope and accessible from anywhere in code and any variable declared within the function and is only accessible from that function and nested function belongs to local scope

3. Consider the following structure of Javascript code:

```
// Scope A
function XFunc () {
      // Scope B
      function YFunc () {
            // Scope C
      };
};
```

(a) Do statements in Scope A have access to variables defined in Scope B and C?

➔ No

(b) Do statements in Scope B have access to variables defined in Scope A?

➔ Yes

(c) Do statements in Scope B have access to variables defined in Scope C?

➔ No

(d) Do statements in Scope C have access to variables defined in Scope A?

➔ Yes

(e) Do statements in Scope C have access to variables defined in Scope B?

➔ Yes

4. What will be printed by the following (answer without running it)?

```
var x = 9;
function myFunction() {
    return x * x;
}
document.write(myFunction());
x = 5;
document.write(myFunction());
```

➔ 81 25

5.

```
var foo = 1;
function bar() {
        if (!foo) {
                var foo = 10;
        }
        alert(foo);
}
bar();
```

What will the *alert* print out? (Answer without running the code. Remember 'hoisting'.)?

➔ 10

6. Consider the following definition of an *add*( ) function to increment a *counter* variable:

```
var add = (function () {
    var counter = 0;
    return function () {
            return counter += 1;
            }
    })();
```

Modify the above module to define a *count* object with two methods: *add*( ) and *reset*( ). The *count.add*( ) method adds one to the *counter* (as above). The *count.reset*( ) method sets the *counter* to 0.

➔

```
var count = (function () {
    var counter = 0;
    function addCounter() {
        return counter += 1;
    }
    function reset() {
        counter = 0;
    }

    return {
        add: addCounter,
        reset: reset
    };
})();


count.add(); //1
```

```
count.add(); //2
count.reset(); //0
count.add(); //1
```

7. In the definition of *add*( ) shown in question 6, identify the "free" variable. In the context of a function closure, what is a "free" variable?

➔ **counter** variable is free variable in add function. Free variables are simply the variables that are neither locally declared nor passed as parameter.

8. The *add*( ) function defined in question 6 always adds 1 to the *counter* each time it is called. Write a definition of a function *make_adder*(*inc*), whose return value is an *add* function with increment value *inc* (instead of 1). Here is an example of using this function:

```
add5 = make_adder(5);
add5( );   add5( );   add5( );  // final counter value is 15

add7 = make_adder(7);
add7( );   add7( );   add7( );  // final counter value is 21
```

➔

```
var make_adder = (function (value) {
    var counter = 0;
    return function () {
        return counter = counter + value;

    }
});

add5 = make_adder(5);
add5( );  add5( );  add5( );  // final counter value is 15

add7 = make_adder(7);
add7( );  add7( );  add7( );  // final counter value is 21
```

9. Suppose you are given a file of Javascript code containing a list of many function and variable declarations. All of these function and variable names will be added to the Global Javascript namespace. What simple modification to the Javascript file can remove all the names from the Global namespace?

➔ Using module pattern, we can remove all the names from the Global namespace

10. Using the *Revealing Module Pattern*, write a Javascript definition of a Module that creates an *Employee* Object with the following fields and methods:

Private Field: name
Private Field: age
Private Field: salary

Public Method: setAge(newAge)
Public Method: setSalary(newSalary)
Public Method: setName(newName)
Private Method: getAge( )
Private Method: getSalary( )
Private Method: getName( )
Public Method: increaseSalary(percentage) // uses private getSalary( )
Public Method: incrementAge( ) // uses private getAge( )

➔

```
var Employee = function () {

    //private fields
    var name, age, salary;

    let getName = function () {
        return this.name;
    }

    let setName = function (name) {
        this.name = name;
    }

    let getAge = function () {
        return this.age;
    }

    let setAge = function (age) {
        this.age = age;
    }
    let getSalary = function () {
        return this.salary;
    }

    let setSalary = function (salary) {
        this.salary = salary;
    }

    let increaseSalary = function (percentage) {
        setSalary(parseFloat(getSalary()) + parseFloat(getSalary() * percentage / 100));
    }

    let incrementAge = function () {
        setAge(parseInt(getAge()) + 1);
    }

    return {
        getName: getName,
        setName: setName,
        getAge: getAge,
        setAge: setAge,
        getSalary: getSalary,
        setSalary: setSalary,
        increaseSalary: increaseSalary,
```

```
        incrementAge: incrementAge
    };
}();
```

11. Rewrite your answer to Question 10 using the *Anonymous Object Literal Return Pattern*.

➔

```
var Employee = function () {

    //private fields
    var name, age, salary;

    let getName = function getName() {
        return this.name;
    }

    let setName = function (name) {
        this.name = name;
    }

    let getAge = function () {
        return this.age;
    }

    let setAge = function (age) {
        this.age = age;
    }
    let getSalary = function () {
        return this.salary;
    }

    let setSalary = function (salary) {
        this.salary = salary;
    }

    let increaseSalary = function (percentage) {
        setSalary(parseFloat(getSalary()) + parseFloat(getSalary() * percentage / 100));
    }

    let incrementAge = function () {
        setAge(parseInt(getAge()) + 1);
    }

    return {
        getName: function () { return getName(); },
        setName: function (name) { setName(name); },
        getAge: function () { return getAge(); },
        setAge: function (age) { setAge(age); },
        getSalary: function () { return getSalary(); },
        setSalary: function (salary) { return setSalary(salary); },
        increaseSalary: function (percentage) { increaseSalary(percentage); },
        incrementAge: function () { incrementAge(); }
    };
}();
```

12. Rewrite your answer to Question 10 using the *Locally Scoped Object Literal Pattern*.

➔

```
var Employee = function () {

    // locally scoped Object
    let myObject = {};
    //private fields
    var name, age, salary;


    let getName = function getName() {
        return this.name;
    }

    let setName = function (name) {
        this.name = name;
    }

    let getAge = function () {
        return this.age;
    }

    let setAge = function (age) {
        this.age = age;
    }
    let getSalary = function () {
        return this.salary;
    }

    let setSalary = function (salary) {
        this.salary = salary;
    }

    let increaseSalary = function (percentage) {
        setSalary(parseFloat(getSalary()) + parseFloat(getSalary() * percentage / 100));
    }

    let incrementAge = function () {
        setAge(parseInt(getAge()) + 1);
    }

    myObject.getName = function () { return getName(); },
        myObject.setName = function (name) { setName(name); },
        myObject.getAge = function () { return getAge(); },
        myObject.setAge = function (age) { setAge(age); },
        myObject.getSalary = function () { return getSalary(); },
        myObject.setSalary = function (salary) { return setSalary(salary); },
        myObject.increaseSalary = function (percentage) { increaseSalary(percentage); },
        myObject.incrementAge = function () { incrementAge(); }

    return myObject;
}();
```

13. Write a few Javascript instructions to extend the Module of Question 10 to have a public *address* field and public methods *setAddress*(*newAddress*) and *getAddress*( ).

➔

```
Employee.address = "";
Employee.setAddress = function (newAddress) {
    this.address = newAddress;
}

Employee.getAddress = function () {
    return this.address;
}
```

14. What is the output of the following code?

```
const promise = new Promise((resolve, reject) => {
      reject("Hattori");
});

promise.then(val => alert("Success: " + val))
       .catch(e => alert("Error: " + e));
```

➔ Shows alert message "Error: Hattori".

15. What is the output of the following code?

```
const promise = new Promise((resolve, reject) => {
      resolve("Hattori");
      setTimeout(()=> reject("Yoshi"), 500);
});

promise.then(val => alert("Success: " + val))
       .catch(e => alert("Error: " + e));
```

➔ Shows alert "Success: Hattori". Promise is already resolved so calling reject has no effect.

16. What is the output of the following code?

```
function job(state) {
    return new Promise(function(resolve, reject) {
        if (state) {
            resolve('success');
        } else {
            reject('error');
        }
    });
}

let promise = job(true);

promise.then(function(data) {
               console.log(data);
               return job(false);})
       .catch(function(error) {
               console.log(error);
               return 'Error caught';
});
```

➔ It prints “success” in console and after that it prints “error” in console.