Javascript ModuleExercises

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); // output:   1
                        8
                        8
                       9
document.write(b); // output:10
document.write(x); //output:  1
```

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

Global scope in JavaScript is every variable defined directly in the JS file, outside any function. These variables are placed in the name space and should be unique.
Local Scope is the variables defined inside a function
.

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, because B and C   variables are defined only within the function XFunc().`
(b) Do statements in Scope B have access to variables defined in Scope A?
`Yes, Scope A variables are defined throughout the page.`
(c) Do statements in Scope B have access to variables defined in Scope C?
`No, because the variables defined in  scope c have a live time only within the function YFunc ().`

(d) Do statements in Scope C have access to variables defined in Scope A?
```
Yes, Scope A variables are defined throughout the page.
```
(e) Do statements in Scope C have access to variables defined in Scope B?
```
Yes, Scope A variables are defined throughout the page.
```

4. What will be printed by the following (answer without running it)?
```
var x = 9;
function myFunction()
                    { return x * x; }
document.write(myFunction()); //output:81
x = 5;
document.write(myFunction()); //output:25
```

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

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

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 (){
                 let  counter=0;
                 let  function adding(){
                                         return counter+=1; }
                 let function resetting (){ counter=0
                                             return counter;}
                 return  {add: adding,
                          reset: resetting
                                               }
                                         })();
```

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?

```
 The free variable is “counter”. Free  variables are variables that come with
the  function and form closure.
```

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 count= ( function (){
                             let  counter=0;
                             let  function adding( incr){
                                                       return counter+= incr; }
                             let function resetting (){ counter=0
                                                           return counter;}
                             return  {add: adding,

}
                                                        })();
```

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?

```
 We can put parentheses around the functions and  execute them directly, while
not keeping track of the variables defined inside the function. This is the
use of module pattern.
```

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(){
                             let  name="";
                             let age= 0;
                             let  salary= 0.0;
            let setName = function (newName){this.name= newName;}
            let setAge= function ( newAge){ this.age= newAge;}
            let setSalary= function ( newSalary){ this.salary= newSalary;}
            let getName = function (){return name;}
            let getAge= function (){return age;}
            let getSalary= function (){ return salary;}
            let increase= function(percentage)
                                     {this.salary+=(this.salary*percentage);}
            Let incrementAge=function(){this.age+=1;}
                    return { setName: setName,
                             setAge: setAge,
                             setSalary: setSalary,
                             getName: getName,
                             getAge: getAge,
                             getSalary: getSalary,
                             increaseSalary:increase,
                             incrementAge: incrementAge;
                           };

} ());
```

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

```
Var  employee= ( function(){
                             let  name="";
                             let age= 0;
         return{                    let  salary= 0.0;
             setName = function (newName){this.name= newName;}
             setAge= function ( newAge){ this.age= newAge;}
             setSalary= function ( newSalary){ this.salary= newSalary;}
             getName = function (){return name;}
             getAge= function (){return age;}
             getSalary= function (){ return salary;}
             increaseSalary= function(percentage)
                                     {this.salary+=(this.salary*percentage);}
             incrementAge=function(){this.age+=1;}
            }
} ());
```

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

```
Var  employee= ( function(){
                             let  name="";
                             let age= 0;
                             let  salary= 0.0;
                             let resultObject={};

resultObject.setName = function (newName){this.name= newName;}
resultObject.setAge= function ( newAge){ this.age= newAge;}
resultObject.setSalary = function ( newSalary){ this.salary= newSalary;}
resultObject.getName = function (){return name;}
resultObject.getAge= function (){return age;}
resultObject.getSalary= function (){ return salary;}
resultObject.increaseSalary= function(percentage)
                                     {this.salary+=(this.salary*percentage);}
resultObject.incrementAge=function(){this.age+=1;}

return resultObject;

} ());
```

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*( ).

```
let address
employee.setAddress = function(newAddress){
                       this.address= newAddress;
                     };

employee.getAddress = function(){
                       return 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));

  Output: 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)); 4

Output: Success: Hattori
             Error Yoshi
```

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';
});
```

```
Output: success
        error
        Error Caught
```