Javascript Module Exercises

1. Determine what this java script 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);
}
```

Answer: x-->10 , a-->8, b-->8, b-->9, b-->10, x-->1

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

Answer: *Global scope refer to declaration of variable at the top level where every other class can common share or update it. There is typically one global scope, and each function defined has its own (nested) local scope. Any function defined within another function has a local scope which in turn is linked to the outer function. If we define a function and create variables inside it, those variables become locally scoped.*

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());
```

Answer: x→81 followed by x→25

5. What will the *alert* print out? (Remember 'hoisting'.)?

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

Answer: 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;
            }
})();
```

```
Var add= (function(){
      Var counter= 0;
          var  objectCounter={
              var add:  function(){ counter++;},
               var reset: function (counter=0;)
          };
      return objectCounter;
  };)();
```

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.

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?

Answer: It is a variable which is defined outside a given the scope o but still can be part of that block using function closure concept.

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 add= (function(){
      var counter= 0;
    return function( inc){
make_adder: return counter+=inc;
          };
      },)();
```

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?

Answer: organizing the code using 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 module10= (function(){
	var name;
	var age;
	var salary;
	var setAge = function(newAge){ this.age= newage;);
	var setSalary= function(newSalary){salary= newSalary;};
	var setName= function(newname){name= newname;};
	var _getAge= function(){ return this.age;};
	var _getSalary= function(){ return this.salary;};
	var _getName= function(){ return this.name;};
	var increaseSalary= function(percentage){ setSalary
	=getSalary*(1+ percentage/100)
	var increaseAge= function(){getAge =_getAge()+1;};
	
	return{
	setAge: setAge
	setSalary: setSalary,
	setName: setName,
	increaseSalary: increaseSalary,
	increaseAge: increaseAge
	};
	};)();

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

```
var Module = (function () {
   default:{ var name:null,
             var age: null,
             var salary:0};
     var _getAge=  function(){ return this.age;};
     var _getSalary=  function(){ return this.salary;};
     var _getName=  function(){ return this.name;};

return {
           setAge:  function(newAge){ this.age= newage;),
            setSalary :function(newSalary){salary= newSalary;},
            setName= function(newname){name= newname;},
            increaseSalary: function(percentage){ setSalary =getSalary*(1+
                percentage/100)},
            increaseAge: function(){setAge =_getAge()+1;}
  };

})();
```

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

Answer: Local scope means a variable/function declared inside a scope.

```
var Module = (function () {
  // locally scoped Object
  var myObject = {};
     var _getAge=  function(){ return this.age;};
     var _getSalary=  function(){ return this.salary;};
     var _getName=  function(){ return this.name;};
```

```
    myObject. setAge = function () {
    };
myObject. setAge = function(newAge){ this.age= newage;);
myObject.setSalary= function(newSalary){salary= newSalary;};
 myObject.setName= function(newname){name= newname;};
myObject.increaseSalary= function(percentage){ setSalary
                          =getSalary*(1+  percentage/100)
myObject. increaseAge= function(){getAge =_getAge()+1;};

  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*( ).

```
Var Module10.extension= (function (){
      var address;
  var  setAdress= function(newAddress){ address= newAddress;);
  var getAddress= function(){ return this.address;};
 return {
     setAddress: getAddress,
     getAddress:  getAddress
};
});();
```

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));
```

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));
```

Success: Hattori
Error: Yoshi

16. What if 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 displays the information stored inside the variable data.