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);
}
```

Undefine,8,8,9,10 and 1

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

Global Scope: a scope which defines the global environment for function and variables Local Scope: the inner scope defined by a function to determine the visibility and accessibility of variables.

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?
(b) Do statements in Scope B have access to variables defined in Scope A?
(c) Do statements in Scope B have access to variables defined in Scope C?
(d) Do statements in Scope C have access to variables defined in Scope A?
(e) Do statements in Scope C have access to variables defined in Scope B?

Ans -> (b),(d) and (e)

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());
```

Ans -> 81 and 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();
```

Ans -> 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.

```
87   var count = (function() {
88       var counter = 2;
89       var add = function() {
90       return counter += 1;
91       }
92       var reset = function(){
93       return counter;
94       }
95       return{
96       add:add
97       }
98       return{
99       reset:reset
100      }
101      })();
102
```

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?

Ans. COUNTER is the free variable Free variable are those variables which are not locally defined inside the closure function or not passed as a parameter to the closure but can be used by the 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
```

```
73      function make_adder(inc) {
74          var counter = 0;
75          var add = function () {
76              return counter += inc;
77          }
78          return add;
79
80      }
81
82      add5 = make_adder(5);
83      console.log(add5()); console.log(add5()); console.log(add5(
84      )); // final counter value is 15
85      add7 = make_adder(7);
86      console.log(add7()); console.log(add7()); console.log(add7());
87
```

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?

Ans. The simplest modification to do would be to use module pattern, wrapping the the entire code construction inside an anonymous function.

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 () {

    var name; //Private
    var age; // Private
    var salary; // Private


    function setAge(newAge) {
        age = newAge;
    }

    function setSalary(newSalary) {
        salary = newSalary;
    }

    function setName(newName) {
        name - newName;
    }

    function getAge() {
        return age;
    }

    function getSalary() {
        return salary;
    }

    function getName() {
        return name;
    }

    function increaseSalary(percentage) {
        setSalary(getSalary + (getSalary * percentage));
    }


    function incrementAge() {
        setAge(getAge() += 1);
    }
    return {
        setAge: setAge,
        setName: setName,
        setSalary: setSalary,
        increaseSalary: increaseSalary,
        incrementAge: incrementAge
    }
}();
```

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

```
var Employee = function () {
    var name;//private field
    var age;//private field
    var salary;//private field

    function getAge() {
        return age;
    }

    function getSalary() {
        return salary;
    }

    function getName() {
        return salary;
    }

    return {
        setAge: function (newAge) {
            age = newAge;
        },
        setName: function (newName) {
            name = newName;
        },
        setSalary: function (newSalary) {
            salary = newSalary;
        },
        increaseSalary: function (percentage) {
            setSalary(getSalary + (getSalary * percentage));
        },

        incrementAge: function () {
            setAge(getAge() += 1);
        }
    }
}();
```

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

```
var Employee = function () {

    var name;//private field
    var age;//private field
    var salary;//private field

    let empObject = {};

    function getAge() {
        return age;
    }

    function getSalary() {
        return salary;
    }

    function getName() {
        return name;
    }

    empObject.setAge = function (newAge) {
        age = newAge;
    }

    empObject.setName = function (newName) {
        age = newName;
    }
    empObject.setSalary = function (newSalary) {
        age = newSalary;
    }

    empObject.increaseSalary = function (percentage) {
        setSalary(getSalary + (getSalary * percentage / 100));
    }
    empObject.incrementAge = function () {

        setAge(getAge() += 1);

    }
  return empObject;
}();

```

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( ).

```
43    Employee.address = " ";
44    Employee.setAddress = function (newAddres){
45    var address;
46    address = newAddress;
47    }
48    Employee.getAddress =function(){
49    return address;
50    }
```

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));
```

Answer : 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));
```

Answer : Success: Hattori