x = 1;

var a = 5;

var b = 10;

var c = function(a, b, c) {

document.write(x); //undefined

document.write(a); //8

var f = function(a, b, c) {

b = a;

document.write(b); //8

b = c;

var x = 5;

}

f(a,b,c);

document.write(b); //9

var x = 10;

}

c(8,9,10);

document.write(b); //10

document.write(x); //1

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

Global Scope is the webpage inside which all the code is being executed, i.e. Variables declared outside a function have global scope. Global variables belong to the window object. On the other hand, Local Scope implies that variables declared within a function become local to that function, i.e. they can’t be accessed outside. It’s hidden from other functions and scripting code

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?

Ans: No

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

Ans: Yes

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

Ans: No

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

Ans: Yes

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

Ans: Yes

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

var x = 9;

function myFunction() {

return x \* x;

}

document.write(myFunction()); // 81

x = 5;

document.write(myFunction()); //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’.)?

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.

var count = (function () {

var counter = 0;

return {

add: function () {

return counter + = 1;

},

reset: function (){

counter = 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?

Ans: free variable = counter,

Free variable is a private variable of the parent and can be accessed by the closure even after the parent function has closed.

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 (val) {

var counter = 0;

return function () {

return counter += val;

}

};

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: We use module pattern to make the function and variable names inaccessible to the Global scope. We enclose the file in a function which calls itself immediately, and this is called Immediately-Invoked-Function-Expressions. Doing so, we creates a new scope and privacy. So, the functions and variables won’t be accessed outside of the module scope.

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

var Employee = (function(){

var name;

var age;

var salary;

var getAge = function(){

return age;

}

var getSalary = function(){

return salary;

}

var getName = function(){

return name;

}

var setAge = function(newAge){

age = newAge;

}

var setSalary = function(newSalary){

salary = newSalary;

}

var setName = function(newName){

name = newName;

}

var increaseSalary = function(percentage){

salary = getSalary() \* (1 + percentage/100);

}

var incrementAge = function (){

age = getAge() + 1;

}

return {

setAge: setAge,

setName: setName,

setSalary: setSalary,

incrementAge: incrementAge,

increaseSalary: increaseSalary

};

})();

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

var Employee = (function (){

var name;

var age;

var salary;

var getAge = function (){

return age;

}

var getSalary = function () {

return salary;

}

var getName = function () {

return name;

}

return {

setAge : function (newAge){

age = newAge;

},

setSalary : function (newSalary){

salary = newSalary;

},

setName : function (newName){

name = newName;

},

increaseSalary : function (percentage){

salary = getSalary() \* (1 + percentage/100);

},

incrementAge : function (){

age = getAge + 1;

}

};

})();

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

var employee = (function (){

var name;

var age;

var salary;

var newObj = {}

var getAge = function (){

return age;

}

var getSalary = function () {

return salary;

}

var getName = function () {

return name;

}

newObj.setAge = function (newAge) {

age = newAge;

}

newObj.setSalary = function (newSalary){

salary = newSalary;

}

newObj.setName = function (newName){

name = newName;

}

newObj.increaseSalary = function (percentage){

salary = getSalary() \* (1 + percentage/100);

}

newObj.incrementAge = function (){

age = getAge() + 1;

}

})();

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 extendEmployee = (function (Employee){

Employee.address = address;

Employee.setAddress = function (newAddress){

address = newAddress;

}

Employee.getAddress = function (){

return address;

}

})(Employee || {})

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));

Ans: 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));

Ans: 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';

});

Ans: success

Error

Error caught