gPick = 0

uPick = 0

Wins = 0

Loses = 0

gCount = 0

**mCount = 10**

gRound = 0

**mRound = 10**

**gLetters[]**

resetGame()

gRound = 0

gCount = 10

gLetters[]

resetRound()

gCount = 10

gLetters[]

getGamePick()

gPick = \_\_\_\_\_\_\_\_\_

getUserPick()

uPick = \_\_\_\_\_\_\_\_\_

startGame()

*If* gRound <= **mRound**

gPick()

*If* gCount <= **mCount**

uPick()

gLetters.push[uPick]

*If*

uPick != gPick

alert bad guess

gCount ++

*else*

uPick === gPick

alert good guess

Wins ++

gCount ++

resetRound()

*else If* gCount > **mCount**

*alert round over please try again*

resetRound()

gRound ++

Loses ++

*else if* gRound > **mRound**

Alert game over

resetGame()

|  |
| --- |
|  |
|  | **//Types of scope: Global and Local**  // Variables defined inside a function are in local scope, |
|  | // while variables defined outside of a function are in the global scope. |
|  | // Each function when invoked creates a new scope. |
|  | // Scope of variables inside for/if blocks are global scopes |
|  | // variables inside if or for blocks contained inside a function are local to function |
|  |  |
|  | // Global Scope |
|  | var age; // variable declaration |
|  | console.log(age); // will print undefined in console as no value is defined for variable |
|  | var name; // variable declaration |
|  | name = "bob"; // value assignment |
|  | console.log(name); // will print bob |
|  | var score = 23; // example of declaration and assignment in same line used in most cases |
|  |  |
|  | // Here we are in Global Scope hence function printName is global and can be called from any where |
|  | function printName() { // this is function declaration |
|  | // value assignment to score change here |
|  | score = 1; |
|  | var hobby = "Reading"; // is local to function and visible only inside this function |
|  | console.log(score); //this will print 1 not 23 |
|  | } |
|  | // console.log(score); // What will be the output of this line ?????? |
|  | printName(); // Function will be executed at this line, when it is called |
|  |  |
|  | // Global Scope |
|  | function printAge() { |
|  | //Variables defined inside a function are in the local scope |
|  | // Local Scope #1 |
|  | num = 12; // num is hoisted and considered global, value is assigned at this line |
|  | function printSkills() { // function declared inside a function is local to function in which it is declared |
|  | // Local Scope #2 |
|  | var skills = "React Developer"; |
|  | age = 34; |
|  | console.log(age); // age is available here as well |
|  | console.log(num); // variables declared in parent function are available in child function |
|  | } |
|  |  |
|  | printSkills(); // this function can be called only from inside the function its declared in |
|  | // console.log(skills); // skills is not defined error is shown in console |
|  | // will not work as skills is local to printSkills. Variables inside child function are not available in parent |
|  | } |
|  | printAge(); |
|  |  |
|  | // printSkills(); // Error printSkills is not defined as this function is nested inside printAge |
|  | console.log(age); // Error : age is not Defined |
|  |  |
|  | //Block statements : if and switch conditions or for and while loops, |
|  | // unlike functions, block statments don't create a new scope. |
|  | // Variables defined inside of a block statement will remain in the scope they were already in |
|  | // |
|  | if (true) { |
|  | var nametwo = "June"; |
|  | } |
|  | console.log(nametwo); |
|  |  |
|  | // Global scope lives as long as your application lives. |
|  | // Local Scope lives as long as your functions are called and executed. |
|  |  |
|  | // Lexical Scope |
|  | // Lexical Scope means that in a nested group of functions, |
|  | // the inner functions have access to the variables and other resources of their parent scope. |
|  | // the exact mechanism used when a JavaScript interpreter is trying to find a particular variable. |
|  | // It starts at the innermost scope being executed at the time, |
|  | // and continue until the first match is found, |
|  | // no matter whether there are other variables with the same name in the outer levels or not |
|  | function grandfather() { |
|  | var name = 'John'; |
|  | // likes is not accessible here |
|  | function parent() { |
|  | // name is accessible here |
|  | // likes is not accessible here |
|  | function child() { |
|  | // Innermost level of the scope chain |
|  | // name is also accessible here |
|  | var likes = 'Coding'; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | // variables with the same name can be specified at multiple layers of nested scope. |
|  | // In such case local variables gain priority over global variables. |
|  | // This type of behavior is called shadowing. Simply put, the inner variable shadows the outer. |
|  | var hobby = "Reading"; |
|  |  |
|  | function myHobby() { |
|  | var hobby = "Music"; |
|  | //hobby = "music" |
|  | console.log(hobby); // Music |
|  | } |
|  | var myhobby = function () { |
|  |  |
|  | } |
|  | console.log(hobby); //Reading |
|  | myHobby(); |
|  |  |
|  | console.log(hobby); //Music |
|  |  |
|  | //If a local variable is assigned without first being declared with the var keyword, |
|  | //it becomes a global variable. |
|  | //To avoid such unwanted behavior, you should always declare your local variables before you use them. |
|  | //Any variable declared with the var keyword inside of a function is a local variable. |
|  | // It’s considered best practice to declare your variables. |
|  |  |
|  | // Variables created without the keyword var, are always global, even if they are created inside a function. |
|  | // Hoisting |
|  | // JavaScript variables consist of two parts: a declaration and an assignment |
|  | var state; // variable declaration |
|  | state = "ready"; // variable definition (assignment) |
|  |  |
|  | var state = "ready"; // declaration plus definition |
|  |  |
|  | // We already know that any variable declared within a scope belongs to that scope. |
|  | // But what we don’t know yet, is that no matter where variables are declared within |
|  | // a particular scope, |
|  | // all variable declarations are moved to the top of their scope (global or local). |
|  | // This is called hoisting, as the variable declarations are hoisted to the top of the scope. |
|  | // Note that hoisting only moves the declaration. Any assignments are left in place |
|  |  |
|  | console.log(city); // output: undefined because we reference it before the actual assignment |
|  | var city = "New York"; // variable definition (assignment) |
|  |  |
|  | //Here is how the code is interpreted by a JavaScript engine |
|  |  |
|  | var city; // moved to the top |
|  | console.log(city); |
|  | city = "New York"; // left in place |
|  |  |
|  | //function declaration and function expression |
|  |  |
|  | // function showState() {} // function declaration |
|  | // var showState = function() {}; // function expression |
|  |  |
|  | // Function declarations are hoisted completely. |
|  | // This means that the entire function’s body is moved to the top. |
|  | // This allows you to call a function before it has been declared: |
|  |  |
|  | // showState(); // output: Ready |
|  |  |
|  | // function showState() { |
|  | // console.log("Ready"); |
|  | // } |
|  |  |
|  | var showState = function () { |
|  | console.log("Idle"); |
|  | }; |
|  |  |
|  | // JavaScript engine moves the declaration of showState() function, |
|  | //and all its content, to the beginning of the scope. The code is interpreted like this: |
|  | // |
|  | // function showState() { // moved to the top (function declaration) |
|  | // console.log("Ready"); |
|  | // } |
|  |  |
|  | // var showState; // moved to the top (variable declaration) |
|  |  |
|  | // showState(); |
|  |  |
|  | // showState = function() { // left in place (variable assignment) |
|  | // console.log("Idle"); |
|  | // }; |
|  |  |
|  | //only the function declaration is hoisted, |
|  | //but the function expression is not. |
|  | //When a function is assigned to a variable, |
|  | //the rules are the same as for variable hoisting |
|  | //(only the declaration is moved, while the assignment is left in place). |
|  | //the function declaration takes precedence over the variable declaration |
|  |  |
|  |  |
|  | // function declaration versus variable assignment, variable assignment takes priority |
|  |  |
|  | // var showState = function() { |
|  | // console.log("Idle"); |
|  | // }; |
|  |  |
|  | // function showState() { |
|  | // console.log("Ready"); |
|  | // } |
|  |  |
|  | // showState();  // output: Idle |
|  |  |
|  |  |
|  | // Interpreted as |
|  | // function showState(){ // moved to the top (function declaration) |
|  | // console.log("Ready"); |
|  | // } |
|  |  |
|  | // var showState; // moved to the top (variable declaration) |
|  |  |
|  | // showState = function(){ // left in place (variable assignment) |
|  | // console.log("Idle"); |
|  | // }; |
|  |  |
|  | // showState(); |
|  | // |
|  | // All declarations, both functions and variables, are hoisted to the top of the containing scope, before any part of your code is executed. |
|  | // Functions are hoisted first, and then variables. |
|  | // Function declarations have priority over variable declarations, but not over variable assignments |
|  |  |
|  |  |
|  | var names = ["Manali", "Kelly", "John"]; |
|  |  |
|  | console.log(name[0]); |
|  | console.log(name[1]); |
|  | console.log(name[2]); |
|  |  |
|  | for (var i = 0; i < names.length; i++) { |
|  | console.log(name[i]); |
|  | if (i === 1) { |
|  | break; |
|  | } else { |
|  | continue; |
|  | } |
|  | } |