Getting Clear about Context Submission

1. The three types of execution contexts are: Global, Function, and Eval. Global execution is described as the default context that is created when a script starts running. This includes all global objects, and any global variables and functions. Hence the name "Global". Function execution is described as the context that is created when a function is being called. This includes all of the functions arguements, local variables, and the value of "this". Hence the name "Function". Eval execution concerns the special context that is created when the 'eval' function is used to execute a string of code in JavaScript.

2. The concept of "hoisting" in JavaScript is used by allowing declarations (such as different variables and functions) to be pushed to the top of their scope. This enables the code more accesible and making it readily available, before it is executed in the script. For example,

x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element

elem.innerHTML = x; // Display x in the element

let x; // Declare x

3. Differences between 'var', 'let', and 'const' when declaring variables:

'var': are globally or function/locally scoped, meaning any variable that is declared with 'var' outside a function block is available for use in the whole window itself. Also meaning, that it is available and can only be accessed within a specific function. Can be re-declared and updated; however, there can be issues with using 'var' so normally the others are more commonly used.

Example: var greeter = "hey hi";

function newFunction() {

var hello = "hello";

}

'let': block-scoped, meaning it is only available for use within that block of code. can be updated but not re-declared. When using 'let', you dont necessarily need to worry if you have used the same for a variable before as a variable only exists within its scope.

Example: let greeting = "say Hi";

if (true) {

let greeting = "say Hello instead";

console.log(greeting); // "say Hello instead"

}

console.log(greeting); // "say Hi"

'const': declarations that maintain consistant values that can only be accessed within the blocks they are declared in. They cannot be updated or re-declared whatsoever. However, every declaration must be initialized first or else error messages will be returned.

Example: const greeting = {

message: "say Hi",

times: 4

}

greeting.message = "say Hello instead";

4. Scope in JavaScript determines the accessibility of variables and functions during runtime. It's defined by where variables and functions are declared in code. Each function creates a new scope, nested within others. Execution context manages scopes, ensuring variables are accessible where defined. Variable environment holds declared variables within each scope.

5. JavaScript Code Snippet Example:

const numbers = [1, 2, 3, 4, 5];

for (let i = 0; i < numbers.length; i++) {

// Checking if the number is even or odd using an if/else statement

if (numbers[i] % 2 === 0) {

console.log(numbers[i] + " is even");

} else {

console.log(numbers[i] + " is odd");

}

}

6. Differences between Arrow, Named, and Anonymous Function Expression:

Named: defined with a name that can be used to refer to the function itself or to call it recursively.

Example: function multiply(a, b) {

return a \* b;

}

// Calling the named function

let result = multiply(5, 3); // result will be 15

Anonymous Function Expression: a function without a name, often assigned to a variable or used as a callback.

Example: let add = function(x, y) {

return x + y;

};

// Calling the function through the variable

let sum = add(2, 3); // sum will be 5

Arrow: concise syntax for writing anonymous function expressions, with implicit return and lexical scope using 'this'.

Example: let square = (num) => {

return num \* num;

};

let squaredValue = square(4); // squaredValue will be 16

7. The JavaScript engine executes scripts by parsing and interpreting code line by line. It manages asynchronous tasks through callback functions, which are invoked (called) upon completion. Promises provide a structured way to handle asynchronous operations, allowing chaining of actions upon resolution or rejection. Async/await syntax simplifies asynchronous code, making it look synchronous while also utilizing Promise methods, so that as a whole it enhances readability and maintainability of code that deals with asynchronous operations.