Kayone Lee
4/19/2023

1. **What are the three types of execution contexts in JavaScript? Briefly explain each type.**

   a. **Function-** This execution context is created whenever a function is executed and includes global objects and variables.
   b. **Global-**The global execution context is the default execution context that is created by the JavaScript engine. Codes that are not inside any function are in the global execution context. It will include any arguments, local variables, and "this" values.
   c. **Eval-**The eval function is created for turning a string into executable JavaScript code. The function is not recommended by modern JavaScript interpreters because it cannot control the privileges of it.

2. **Explain the concept of variable hoisting in JavaScript. Provide an example to illustrate your explanation.**
   Hoisting is a JavaScript mechanism where declarations are moved to the top of their scope. Variables or functions can be used before it has been declared.

   ```
   var a;
   a= 5;
   console.log (a); //this will log 5
   ```

3. **What is the difference between 'var', 'let', and 'const' when declaring variables in JavaScript? Provide examples for each.**
   "Var" is the older way of declaring variables meanwhile the more modern variables are "let" and "const". Variables that are declared with "var" have no block scope. The visibility is scoped to the current function or globally. Although "let" and "const" are considered the newer ways of declaring variables, they have their differences. Variables with "let" can be updated but not re-declared. You can also identify the "let" variables with curly braces. Variables with "const" can neither be updated nor re-declared.

   <u>EXAMPLES:</u>

"Var" (*The scope is global when a "Var" variable is declared outside a function.)*

```
var greet = " hi";
function newFunction() {
var hello = "hello!!!";
}
```

"**Let"** *(Anything within the curly brace is considered a block scope)*

```
let greeting = "Hi there";
let times = 6;

if (times > 5) {
let hello = "How Are You!! ";
console.log(hello);    // printing log will show " How Are You!! "
```

"**Const"** (*Variable values are constant and maintained. Cannot be re-declared)*

Kayone Lee
4/19/2023

```
const greeting = {
message: "say Hi",
times: 4
}
```

4. **Explain the concept of scope in JavaScript. How does it relate to execution context and variable environment**?
   The scope is where a variable is available in your code; it determines the accessibility of variables. Scope is related to execution and the variable environment because it determines which variables/functions are accessible at a given time. For example, if a variable is declared inside a function, then it can only be visible inside of that function.

5. **Write a JavaScript code snippet to demonstrate the use of an if/else statement and a for loop.**

```
let name = "Sally";

if (name === "Sally") {
  console.log("Welcome Home, Sally!");
} else {
  console.log("Hello, Again!");
}

for (let i = 0; i < 3; i++) {     // Loop through 0 to 2
  console.log(i);
}
```

6. **What is the difference between a named function, an anonymous function expression, and an arrow function? Provide examples for** each.

A **named** function is defined with a name during declaration and can be used when you need to call the same function more than once.	`function addSomething (a, b) {`
	` return a + b;`
	`}`
	
	`let sum = addSomething(3,3);`
	`console.log(sum);     // log prints out 6`

An **anonymous** function is defined with no name and typically only used when you want to define a function one time. It will instead be assigned to a variable or passed as an argument to another function.	`let addSomething = function (a, b) {`
	` return a + b;`
	`};`
	
	`let sum = addSomething (3,3);`
	`console.log(sum);     // log prints out 6`

An **arrow** function uses a syntax of "=>" to take a single argument and have a single expression in the body.	`let addSomething = (a, b) => a+b;`
	
	`let sum = addSomething(3,3);`
	`console.log(sum);     // log prints out 6`

Kayone Lee
4/19/2023

7. **Briefly explain the role of the JavaScript engine in processing a script, and how it handles asynchronous tasks using callback functions, Promises, and async/await syntax.**
   The JavaScript engine processes codes in two main phases (*parsing and execution*). The code that we write is executed line by line which means JavaScript can only execute one task at a time.

   Asynchronous execution doesn't wait for the task to be complete but instead adds the task to the event queue and continues executing the remaining codes. Once the task is completed, the engine adds a callback function to the event queue. By using callbacks, you will be able to write reusable code. Callbacks run when their parent function finishes executing. Promises allow you to chain multiple asynchronous tasks and when the task is completed, the promise can either fulfill or reject. The result will then be handled by the **.then** and **.catch**. Async syntax is used to define a function that returns a promise. Await syntax is used to put a pause on the code executing until the promise is fulfilled or rejected.