**CS ACTIVITY FORM**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Student Name** | Pasamonte, Claire A. | | **Year & Section** | | CS2D | **Activity No.** | Perf1 | |
| **Class Instructor** | | **CO Pascua** | **Class Subject** | DSA | | **Class Schedule** | | 8:00 - 9-30 |

**Activity Proper:**

**1. Differentiation of let, const, and var in JavaScript**

In JavaScript, let, const, and var are used to declare variables, but they behave differenty in terms of scope, reassignability, and best practices.

**var** - In JavaScript, var is function-scoped, meaning that a variable declared with var is accessible within the entire function, even if it is defined inside smaller blocks like if or for. You can reassign and redeclare a var variable within the same scope. One important aspect of var is hoisting, which means the declaration is moved to the top of the scope before code execution, but the value remains undefined until assigned. This behavior can sometimes lead to unexpected results and bugs, which is why var is generally avoided in modern JavaScript.

**let -** The let keyword is block-scoped, meaning that it is limited to the block in which it is defined (for example, inside an if or for block). Unlike var, let cannot be redeclared in the same scope, but it can be reassigned. Although let is hoisted, it remains in a "temporal dead zone" until its actual declaration, preventing it from being accessed before initialization. This makes it a more predictable choice for variables that need to change value throughout the program.

**const -** Similar to let, const is also block-scoped. However, once a variable is declared with const, it cannot be reassigned. Although you cannot change the value of a const variable, if it holds an object, the properties of the object can still be modified. This makes const ideal for values that should remain constant throughout the program, helping prevent accidental changes. Using const is a common best practice when the value of the variable is not expected to change, as it ensures immutability at the reference level.

**2. Concept of Falsy Values in JavaScript**

In JavaScript, a falsy value is a value that evaluates to `false` when encountered in a Boolean context, such as in conditional statements. These values, when used in expressions or conditions, are treated as `false`, even though they are not explicitly 'false' themselves. JavaScript has six falsy values: `false`, `0`, `""` (empty string), `null`, `undefined`, and `NaN`. Falsy values are useful in conditional statements like `if` or loops, where JavaScript automatically converts these values into a Boolean context and treats them as `false`.

For example, the number `0` is considered falsy because it represents the absence of a numerical value. The empty string `""` is falsy because it signifies no content. `null` is also falsy because it represents a deliberate non-value or empty reference in JavaScript. These falsy values make it easy to write conditional checks without needing to explicitly compare them to `false`.