**PART 1**

Q2) Hoisting is a JavaScript behavior where function and variable declarations get hoisted at the top of their scope during the compiling stage. That means, we can make use of variables and functions even before we can declare them in the code. However, whether variables are hoisted or not depends on whether they are declared or not. Variables declared using var are hoisted and initialized with undefined and hence can be called even before their declaration without runtime error. Variables declared using let and const are hoisted but not initialized. They are in a temporal dead zone from the beginning of the block till the point of declaration. So, calling them before the declaration will throw a ReferenceError.

**PART 2**

Q1) The argument passed to setTimeout is a normal function, not an arrow function. When its executed, **this** is determined by how the function is called, not where it was defined. Since the function is called by the browser’s setTimeout and not by the user, **this** refers to:

* The global object, or
* undefined if we’re in strict mode.

Neither of these has a name property like "Alice", so this.name is undefined.

Q2) We store **this** in a variable in traditional functions in JavaScript to preserve the correct context of **this** inside nested functions.

By storing **this** from the outer scope into a variable like self, we can use that inside the inner function.

Here self is a regular variable, so its value is preserved. When called inside the setTimeout function, self still refers to the original object, even though **this** does not.

Q3) Using an **arrow function** within setTimeout correctly refers to the user object because **arrow functions do not have their own this**. Instead, they use **lexical this binding**, meaning they inherit **this** from the scope in which they are defined.

**PART 3**

Q1) A closure is a feature in JavaScript, where an inner function remembers and has access to the variables from its outer function’s scope, even after the outer function has finished executing.

A closure enables the increment and decrement counter to capture and retain the count variable from where they were created, so that they can continue to modify and access it independently.

**PART 4**

Q1) In JavaScript, **functions are flexible** with the number of arguments passed--they don't require the number of arguments to match the number of parameters defined.

JavaScript handles in the following ways;

* Too Few Arguments:

Parameters that don’t get a corresponding argument are set to undefined.

* Too Many Arguments:

Extra arguments are simply ignored, unless the function access them using arguments object or rest parameters.

Q2) The rest parameter in JavaScript allows to capture an indefinite number of arguments passed to a function into a single array. This is especially useful when you don’t know how many arguments will be passed in.

Purpose:

* To handle variable number of arguments.
* To replace the older argument object with cleaner, array-based syntax.
* To group remaining parameters into single array.

Syntax:

Function functionName(…restParameter){

//restParameter is an array of remaining elements

}

* The …(three dots) is the rest parameter.
* The rest parameter must be the last in the parameter list.