**Part-1 answers:**

1.

The difference in the behaviour can be explained as:

1. When we use var, the variable declaration is hoisted to the top and automatically initialized with undefined.

2. When we use let and const, the variable declaration is hoisted but it is not initialized.

3. In the case of var, we can access the variable before its assignment and it will return undefined.

4. In the case of let and const, if we try to access the variable before it is declared this results in a ReferenceError.

2.

Hoisting is the feature in JavaScript where variable declarations are moved to the top of their scope before the code runs.

->When we use var, the variable is hoisted and automatically set to undefined so when we try to access it will return undefined instead of causing an error.

->with let and const. The declaration is hoisted but not initialized. When we try to access the variable before its declaration it gets a ReferenceError.

->This period between when a block starts and when a let or const variable is declared is called the temporal dead zone.

**Part-2 Answers**

1. In the first code, setTimeout is passed a normal function, and in JS normal functions do not automatically bind this to the object that contains them.

It all depends on how the function is called and inside setTimeout the callback is called by the timer, so this refers to the global object (undefined), not to the user object.

1. we save the value of this (the user object) in another variable like self. Then, even inside the setTimeout callback, self still points to the correct object. This is because JavaScript functions can remember variables from the outer scope this is a technique known as closure.
2. Arrow functions don't create their own this. They simply use the this of where they were defined. This is called lexical this binding. So when we use an arrow function within setTimeout, it doesn't change what this refers to, it keeps the same this as the function it's inside. That means this continues to refer to the object (such as user), so it works correctly.

**Part-3 Answers**

1. A closure occurs when a function remembers the variables from where it was created even after that place is removed. In the given example, the increment and decrement functions still "remember" and can use the count variable from setupCounter even though setupCounter already ran. This enables each counter to have its own individual private version of count, and that state is stored independently using the closure.

**Part-4 Answers**

1. If very few arguments are provided, the missing parameters become undefined.

If too many arguments are passed, the extra ones are ignored unless processed with the arguments object or the rest parameter.

1. The rest parameter is defined as it allows a function to accept an

indefinite number of arguments as an array. It must be the last parameter in

function definition.

Syntax: function myFunction(a, b, ...rest) {

}