**Understanding Scope and Lexical Environment**

**Exercise 1.1: Function Scope**

function outerFunction() {

let a = 10;

function innerFunction() {

console.log(a); // What will this log?

}

innerFunction();

}

outerFunction();

**Questions:**

* **What value does innerFunction log? Why?**
  + innerFunction logs 10 because it can access the variable a declared in its outer function outerFunction due to JavaScript's lexical scoping. The a variable is within the scope of innerFunction.
* **What would happen if you call innerFunction outside of outerFunction?**
  + If you try to call innerFunction outside of outerFunction, it would result in a reference error because innerFunction is not accessible outside the scope of outerFunction.

**Exercise 1.2: Block Scope with let and const**

for (let i = 0; i < 5; i++) {

console.log(i);

}

console.log(i); // What will this log?

**Questions:**

* **What happens when you try to log i outside the loop?**
  + Trying to log i outside the loop will result in a reference error because i is scoped to the block of the for loop, and is not accessible outside of it when declared with let.
* **How does block scope differ from function scope?**
  + Block scope confines the variable to the block in which it is declared, such as within a loop or an if statement. Function scope confines the variable to the function in which it is declared. var has function scope, while let and const have block scope.

**2: Closures and Lexical Environment**

**Exercise 2.1: Creating a Closure**

function createCounter() {

let count = 0;

return function() {

count++;

console.log(count);

};

}

const counter1 = createCounter();

const counter2 = createCounter();

counter1(); // What will this log?

counter1();

counter2(); // What will this log?

counter2();

counter1();

**Questions:**

* **Why do counter1 and counter2 maintain separate counts?**
  + counter1 and counter2 maintain separate counts because each call to createCounter creates a new lexical environment with its own count variable. Each returned function has a closure over its own count.
* **What does this tell you about closures and lexical environment?**
  + This demonstrates that closures capture the variables from their lexical environment at the time they are created, allowing each closure to maintain its own independent state.

**Exercise 2.2: Closure in Loop**

let functionArray = [];

for (let i = 0; i < 3; i++) {

functionArray.push(function() {

console.log(i);

});

}

functionArray[0](); // What will this log?

functionArray[1]();

functionArray[2]();

**Questions:**

* **What values are logged and why?**
  + 0, 1, and 2 are logged because using let creates a new lexical environment for each iteration of the loop, capturing the current value of i for each function in functionArray.
* **What would happen if let was changed to var?**
  + If let was changed to var, all the functions in functionArray would log 3 because var is function-scoped (or globally scoped if not in a function) and not block-scoped. Each function would capture the final value of i after the loop ends, which is 3.

**3: Advanced Concepts with Lexical Environment**

**Exercise 3.1: IIFE and Lexical Environment**

const logVariable = (function() {

let secret = "I am a secret";

return function() {

console.log(secret);

};

})();

logVariable(); // What will this log?

**Questions:**

* **How does the IIFE create a new lexical environment?**
  + The IIFE creates a new lexical environment because it immediately executes, capturing the variable secret within its own scope. The returned function retains access to this scope, creating a closure.
* **Why can the returned function still access secret?**
  + The returned function can still access secret because it forms a closure over the lexical environment of the IIFE where secret is defined, allowing it to access secret even after the IIFE has executed.

**Exercise 3.2: Lexical Environment and this**

const obj = {

value: 42,

logThis: function() {

console.log(this);

}

};

obj.logThis(); // What will this log?

**Questions:**

* **What does this refer to in the method?**
  + In the logThis method, this refers to the object obj because the method is called as a property of obj.
* **How does this differ from a lexical environment variable?**
  + this is a special keyword that refers to the object context in which a function is called, whereas lexical environment variables are specific to the scope in which they are defined and follow the static structure of the code. The value of this can change depending on how a function is called, but lexical environment variables do not change based on the calling context.