1. Why are closures useful in JavaScript? Give an example use case.

Ans: Closures in JavaScript are useful because they allow functions to access variables from an outer scope even after that scope has finished executing. This enables powerful programming patterns such as data encapsulation, where variables can be kept private and only accessible through specific functions, and maintaining state in asynchronous operations, ensuring variables persist between function calls or events.

Example:

```
function outerFunction() {
    let outerVariable = 'I am from the outer function';

    function innerFunction() {
        console.log(outerVariable);
    }

    // Returning the inner function
    return innerFunction;
}

// Calling outerFunction returns innerFunction, which holds a closure over outerVariable
const innerFunc = outerFunction();

// Executing innerFunc will still have access to outerVariable
innerFunc(); // Output: I am from the outer function
```

When outerFunction is called, it defines outerVariable and declares innerFunction, which gains access to outerVariable through a closure formed during outerFunction's execution. Even after outerFunction completes, innerFunction retains the ability to access and modify outerVariable when invoked separately as innerFunc. This illustrates how closures in JavaScript preserve the scope chain, allowing functions to retain access to variables from their parent scopes beyond the lifetime of those scopes.

2. When should you choose to use “let” or “const”.

Ans: Use **const** when you want to declare a variable that will not be reassigned. It ensures the variable remains constant throughout its scope, preventing accidental reassignment.

Use **let** when you anticipate the need to reassign the variable later. This allows you to declare a variable that can have its value changed as needed within its scope.

3. Give an example of a common mistake related to hoisting and explain how to fix it.

Ans: In JavaScript, when using arrow functions with var, the variable declaration (var myArrowFunc) is hoisted to the top of its scope, but not its initialization (myArrowFunc = () => {...}). This means that if you try to call myArrowFunc() before the arrow function is assigned to myArrowFunc, it will throw a TypeError because myArrowFunc is initially undefined. To fix this issue, ensure that the arrow function assignment (var myArrowFunc = () => {...}) precedes any calls to myArrowFunc(). This ensures that myArrowFunc is properly initialized with the arrow function before it's invoked, preventing the TypeError and allowing the function to execute as intended

4. What will the outcome of each console.log() be after the function calls? Why?

Ans:

```
const arr = [1, 2];
function foo1(arg) {
    arg.push(3);
}
foo1(arr);
console.log(arr); // Output: [1, 2, 3]
```

When you pass an array like arr to a function in JavaScript, changes made to the array inside the function, such as pushing elements, directly modify the original array arr. This is why console.log(arr) after foo1(arr) displays [1, 2, 3].

```
const arr = [1, 2];

function foo2(arg) {
  arg = [1, 2, 3, 4];
}

foo2(arr);
console.log(arr);  // Output: [1, 2, 3]
```

In the function foo2, arg initially refers to the same array as arr when foo2(arr) is called. However, the assignment arg = [1, 2, 3, 4]; inside foo2 creates a new array [1, 2, 3, 4] and assigns it to arg. This assignment does not affect the original array arr that was passed into foo2

```
function foo3(arg) {
  let b = arg;
  b.push(3);
}
foo3(arr);
console.log(arr);  // Output: [1, 2, 3, 3]
```

foo3 creates a new reference b to the same array as arr. Modifying b inside foo3 (b.push(3)) also modifies arr, because b and arr reference the same array object. Therefore, arr becomes [1, 2, 3, 3].

```
function foo4(arg) {
  let b = arg;
  b = [1, 2, 3, 4];
}
foo4(arr);
console.log(arr);  // Output: [1, 2, 3, 3]
```

foo4 assigns a new array [1, 2, 3, 4] to the variable b, which is local to foo4. This assignment does not affect arr because b is a new reference pointing to a different array. The modification made to b does not impact arr, so arr remains [1, 2, 3, 3].