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

A function which is bundled around its lexical scope forms a Closure. (lexical scope is the ability for a function to access variables from the parent scope)

Closures are useful because:

1. It helps in data privacy as it allows variables to be private within a function by enabling Data Encapsulation.
2. It helps in managing the state between function call
3. It is useful in functional programming as it allows functions to be passed around with their own environment.

Example:

function x(){

let val = 5

// function y is bundled with the contents of x.

function y(){

console.log(val)

}

return y

}

z = x(); // z has the function y

z(); // this will print "val". Although the execution of function x is completed but z has the contents of function x as function y was bundled with the content's of function x and returned when we executed z = x();

**When should you choose to use “let” or “const”**

let: We should use “let” for variables whose values need to be re-assigned

const: We should use “const” for variables whose values should not be re-assigned

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

A common mistake related to hoisting is using a variable (“let” or “const”) before it is declared would result into “ReferenceError” or “SyntaxError”.

Example:

car = "volvo"

const car

console.log(car)

The above would give the following error:

SyntaxError: Missing initializer in const declaration

To fix the error, we need to first declare the variables then use it.

Example:

const car = "volvo"

console.log(car)

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

const arr = [1, 2];

function foo1(arg) {

  arg.push(3);

}

foo1(arr);

console.log(arr);

Output: [1, 2, 3]

Explanation:

Due to the “const” keyword the variable “arr” can’t be reassigned to a different value, but it allows to modify the contents of the Array. When we pass “arr” to the function “foo1”, the parameter “arg” becomes a reference to to “arr” (a copy of the array “arr”). Inside the function “push” adds 3 to “arg”. Since “arg” and “arr” refer to the same array this modification is reflected in “arr” too.

const arr = [1, 2];

function foo2(arg) {

  arg = [1, 2, 3, 4];

}

foo2(arr);

console.log(arr);

Output: [1,2]

Explanation:

Since we are doing a reassignment, the changes are only reflected in the local reference “arg” within the function scope, but it doesn’t affect the original reference outside the function.

Using methods like push, pop, shift, unshift, splice, etc., modifies the contents of the array without changing the reference. These modifications are reflected in all references to that array. Whereas reassignment only affects the local reference.

const arr = [1, 2];

function foo3(arg) {

  let b = arg;

  b.push(3);

}

foo3(arr);

console.log(arr);

Output: [1, 2, 3]

Explanation:

The value 3 is pushed to “b” and is reflected in “arr” because, we know “arg” is reference to “arr” so if we make any changes to the contents of “arg” it will be reflected in “arr”. Here we are assigning “arg” to “b” meaning “b” now also references to “arr”. Thus making changes to the contents of “b” would also be reflected in “arr”

const arr = [1, 2];

function foo4(arg) {

  let b = arg;

  b = [1, 2, 3, 4];

}

foo4(arr);

console.log(arr);

Output: [1, 2]

Explanation:

Since we are do reassignment and reassignment is function scoped, so the changes are not reflected in arr. Thus the output is [1,2]