1-Shallow Copy and Deep Copy in Javascript

Shallow Copy: When a reference variable is copied into a new reference variable using the assignment operator, a shallow copy of the referenced object is created. In simple words, a reference variable mainly stores the address of the object it refers to. When a new reference variable is assigned the value of the old reference variable, the address stored in the old reference variable is copied into the new one. This means both the old and new reference variable point to the same object in memory.

```
var employee = {
    eid: "E102",
    ename: "Jack",
    eaddress: "New York",
    salary: 50000
}


console.log("Employee=> ", employee);
var newEmployee = employee;    // Shallow copy
console.log("New Employee=> ", newEmployee);

console.log("---------After modification----------");
newEmployee.ename = "Beck";
console.log("Employee=> ", employee);
console.log("New Employee=> ", newEmployee);
// Name of the employee as well as
// newEmployee is changed.
```

deep copy makes a copy of all the members of the old object, allocates separate memory locations for the new object, and then assigns the copied members to the new object. In this way, both the objects are independent of each other and in case of any modification to either one, the other is not affected. Also, if one of the objects is deleted, the other remains in the memory. Now to create a deep copy of an object in JavaScript we use JSON.parse() and JSON.stringify() methods. Let us take an example to understand it better.

```
var employee = {
    eid: "E102",
    ename: "Jack",
    eaddress: "New York",
    salary: 50000
}
console.log("=========Deep Copy========");
var newEmployee = JSON.parse(JSON.stringify(employee));
console.log("Employee=> ", employee);
console.log("New Employee=> ", newEmployee);
console.log("---------After modification---------");
newEmployee.ename = "Beck";
newEmployee.salary = 70000;
console.log("Employee=> ", employee);
console.log("New Employee=> ", newEmployee);
```

2-Hoisting

JavaScript Hoisting refers to the process whereby the interpreter appears to move the *declaration* of functions, variables, or classes to the top of their scope, prior to the execution of the code.

Hoisting allows functions to be safely used in code before they are declared.

Variable and class *declarations* are also hoisted, so they too can be referenced before they are declared. Note that doing so can lead to unexpected errors, and is not generally recommended.

a-Function hoisting:

One of the advantages of hoisting is that it lets you use a function before you declare it in your code.

```
catName("Tiger");

function catName(name) {
  console.log(`My cat's name is ${name}`);
}
/*
The result of the code above is: "My cat's name is Tiger"
*/
```

B-Variable hoisting

1-var hoisting we declare and then initialize the value of a var after using it. The default initialization of the var is undefined.

2-let and const hoisting

Variables declared with let and const are also hoisted but, unlike var, are not initialized with a default value. An exception will be thrown if a variable declared with let or const is read before it is initialized.

3-Higer Order Function

In JavaScript, functions are treated as first-class citizens. We can treat functions as values and assign them to another variable, pass them as arguments to another function, or even return them from another function.

This ability of functions to act as first-class functions is what powers higher-order functions in JavaScript.

Basically, a function that takes another function as an argument or returns a function is known as a higher-order function.