Month-5-Test : FE+BE

ANS-1- (b) <ol type=”a”>

ANS-2- (d) <!DOCTYPE html>

ANS-3- (a) #id

ANS-4- (c) static

ANS-5- (d) e.stopPropagation()

Descriptive Answers

ANS-6- CLOSURE:

Closure means that an inner function always has access to the vars and parameters of its outer function, even after the outer function has returned. Inner function can access variables and parameters of an outer function (however, cannot access arguments object of outer function). A closure is a function bundled with its lexical scope. Closures are created at runtime during function creation.

Example:

```
function OuterFunction() {
   var outerVariable = 100;
   function InnerFunction() {
      alert(outerVariable);
   }
   return InnerFunction;
}
var innerFunc = OuterFunction();

innerFunc(); // 100
```

In the above example, return InnerFunction; returns InnerFunction from OuterFunction when you call OuterFunction(). A variable innerFunc reference the InnerFunction() only, not the OuterFunction(). So now, when you call innerFunc(), it can still access outerVariable which is declared in OuterFunction(). This is called Closure. One important characteristic of closure is that outer variables can keep their states between multiple calls.Closure is useful in hiding implementation detail in JavaScript. In other words, it can be useful to create private variables or functions.

HOISTING:

JavaScript before executing our code parses it, and adds to its own memory every function and variable declarations it finds, and holds them in memory. This is called hoisting. We have some different behaviors for function declarations and function expressions. With function declarations, we can call a function before it's defined, and our code will work. In the other cases, we'll have errors.A general rule of thumb is to always define functions, variables, objects and classes before using them, to avoid surprises.

Example:

```
function bark() {
  alert('wof!')
}
```

Due to hoisting, we can technically invoke bark() before it is declared:

```
bark()
function bark() {
  alert('wof!')
}
```

With functions, this only happens for function declarations. Like in the case above.Not with function expressions.This is a function expression:

```
bark()
var bark = function() {
  alert('wof!')
}
```

In this case, the var declaration is hoisted and initialized with undefined as a value, something like this:

```
var bark = undefined
bark()
bark = function() {
  alert('wof!')
}
```

Running this code will give you a TypeError: bark is not a function error. const and let declarations are hoisted, too, but they are not initialized to undefined like var.

```
const bark = function() {
  alert('wof!')
}
or

let bark = function bark() {
  alert('wof!')
}
```

In this case, if you invoke bark() before declaring it, it will give you a ReferenceError: Cannot access 'bark' before initialization error. The same will happen for any other expression that assigns an object or class to a variableClass declarations work like let and const declarations: they are hoisted, but not initialized, and using a class before its declaration will give a ReferenceError: <Class> is not defined error.

CURRYING(FUNCTION COMPOSITION):

Function composition is a mechanism of combining multiple simple functions to build a more complicated one. The result of each function is passed to the next one. In mathematics, we often write something like: f(g(x)). Taking a quick example, suppose I need to make some arithmetic by doing the following operation: 2 + 3 * 5. As you

may know, the multiplication has the priority over the addition. So you start by calculating 3 * 5 and then when add 2 to the result. Let's write this in JavaScript. The primary and certainly the most simple approach could be:

```
const add = (a, b) => a + b;
const mult = (a, b) => a * b;
add(2, mult(3, 5))
```

This is a form of function composition since this is the result of the multiplication that is passed to the add function. A curried function is a function that takes multiple arguments one at a time. Given a function with 3 parameters, the curried version will take one argument and return a function that takes the next argument, which returns a function that takes the third argument. The last function returns the result of applying the function to all of its arguments.You can do the same thing with more or fewer parameters. For example, given two numbers, a and b in curried form, return the sum of a and b:

```
// add = a => b => Number
const add = a => b => a + b;
```

To use it, we must apply both functions, using the function application syntax. In JavaScript, the parentheses () after the function reference triggers function invocation. When a function returns another function, the returned function can be immediately invoked by adding an extra set of parentheses:

```
const result = add(2)(3); // => 5
```

First, the function takes a, and then returns a new function, which then takes b returns the sum of a and b. Each argument is taken one at a time. If the function had more parameters, it could simply continue to return new functions until all of the arguments are supplied and the application can be completed.The add function takes one argument, and then returns a partial application of itself with a fixed in the closure scope.

ANS-7- CSS Selectors:

- #id selectors are worth 100
- .class selectors are worth 10
- tag selectors are worth 1

The selector with the highest “score” will prevail, no matter the order in which the CSS rules appear. When the browser needs to resolve what styles to apply to a given HTML element, it uses a set of CSS precedence rules. Given these rules, the browser can determine what styles to apply. The rules are:

- !important after CSS properties.
- Specificity of CSS rule selectors.
- Sequence of declaration.

Note, that CSS precedence happens at CSS property level. Thus, if two CSS rules target the same HTML element, and the first CSS rule takes precedence over the second, then all CSS properties specified in the first CSS rule takes precedence over the CSS properties declared in the second rule. However, if the second CSS rule contains CSS properties that are not specified in the first CSS rule, then these are still applied. The CSS rules are combined - not overriding each other.

>> !Important

If you need a certain CSS property to take precedence over all other CSS rules setting the same CSS property for the same HTML elements, you can add the instruction !important after the CSS property when you declare it. The !important instruction has the highest precedence of all precedence factors. Here is an !important example:

Example:

```
<style>
 div {
    font-family: Arial;
    font-size: 16px !important;
 }
 .specialText {
    font-size: 18px;
 }
</style>

<div class="specialText">
  This is special text.
</div>
```

This example contains two CSS rules which both target the div element.Normally, a CSS rule with CSS class selector has higher specificity than a CSS rule with an element selector, so normally the second CSS rule (.specialText {...}) would take precedence over the first CSS rule (div {...}). That means, that the .specialText rule would set the font-size of the div element to 18px.However, since the div {...} CSS rule contains the instruction !important after the font-size CSS property, then that CSS property declaration takes precedence over all other declarations without the !important instruction targeting the same HTML element.The specificity of a CSS rule depends on its selector. The more specific the CSS selector is, the higher is the precedence of the CSS property declarations inside the CSS rule owning the selector.The different CSS selector types has different specificity. By specificity is meant how specifically the CSS selector targets the element is selects. Here is a list of CSS selector specificity:

Example:

```
<body>
  <style>
    body     { font-size: 10px; }
    div      { font-size: 11px; }
    [myattr] { font-size: 12px; }
    .aText   { font-size: 13px; }
    #myId    { font-size: 14px; }
  </style>

  <div                          > Text 1 </div>
  <div myattr                       > Text 2 </div>
  <div myattr class="aText"         > Text 3 </div>
  <div myattr class="aText" id="myId" > Text 4 </div>
</body>
```

This example contains 5 different CSS rules which all target one or more of the div elements in the example.The first CSS rule targets the body element. The styles set for the body element in this CSS rule are inherited by the div elements. The CSS properties set in this CSS rule will have the lowest precedence for the div elements, as they are not set directly on the div elements, but rather on their parent element, the body element.The second CSS rule targets the div elements. This CSS rule is more specific to div elements that the styles inherited from the body element.The third CSS rule targets all HTML elements with an attribute named myattr. This is more specific than all div elements. Thus, the div element with the myattr attribute has this CSS rule applied. If you had set an attribute value on the attribute selector, it would have been even more specific.The fourth CSS rule targets all HTML elements with the CSS class named aText. The CSS class selector is more specific than a the div element selector and [myattr] attribute selector, so the div element with the CSS class aText will have this CSS rule applied. The fifth CSS rule targets the HTML element with the ID myId. The ID selector is more specific than the element selector, attribute selector and class selector, so the div element with the ID myId has this CSS rule applied.

ANS-8- OUTPUT:

```
Index:4,element:undefined
Index:4,element:undefined
Index:4,element:undefined
Index:4,element:undefined
```

>> Explanation:

we get element as undefined due to hoisting and index as 4 because have used setTimeout Function to Increase the output time of every console output with 3000ms time delay. Hence it loops over the length of arr which is 4 times and gives the above result.

ANS-9- OUTPUT:

```
1
4
3
```

>> Explanation:

At stage one once new promise is on the way it will console 1 and will wait for the success and after success it will wait for the response to send back and mean time it will console 4 and once response received at server end it will console 3.