Front-end Assignment

Day 2 Assignment 1

- Snippet 1

```
function printName() {
  if (true) {
    var name = "Akshay";
  }
  console.log(name);
}

printName();
```

Observation: The code runs successfully and the output is ***Akshay***.

- Snippet 2

```
let age = "18";

if (age === 18) {
  console.log("Adult");
} else {
  console.log("Minor");
}
```

Observation: The code runs successfully and the output is ***Minor***, but the logic is not correct. Strict equality (===) check for type too and that is why 18===”18” will result in false. Moreover, the relational operator that should be used should be >=, since adult will be considered when the person is more than or equal to 18 years of age.

- Snippet 3

```
const arr = [10, 20, 30];


for (let i = 0; i <= arr.length; i++) {
  console.log(arr[i]);
}
```

Observations: The code is executed succesfully and the output will be:

10

20

30

undefined

In simple words, the loop will iterate through the elements in the array one by one, and the last undefined is printed because the condition says i<=*arr.length*, at the last iteration the value of i will get out of bounds and hence the undefined is printed. The correct condition should have been : *i<arr.length.*

- Snippet 4

```
let data;


setTimeout(() => {
  data = "Loaded";
}, 1000);


console.log(data);
```

Observations: The code executes successful and the output is ***undefined***. It may seem like the output should have been “Loaded”, but in the ***setTimeout*** function it tells the browser to wait for 1000ms but the JS doesn’t wait and moves to the next line which is ***console.log(data)*** which prints that default value i.e. ***undefined*** since data wasn't initialized at the start.

- Snippet 5

```
function add(a, b) {
  a + b;
}
const result = add(2, 3);
console.log(result);
```

Observations: The code runs successfully and the output is ***undefined***. The logic is correct for addition here, but in JS we have to explicitly tell what does a function return, and if not specified, it returns undefined by default.

- Snippet 6

```
const user = {
  name: "John",
  age: 25,
};

function updateAge(u) {
  u.age = 30;
}

updateAge(user);
console.log(user.age);
```

Observations: The code executes successfully and the output is ***30***. Here the value of user is passed as reference so the changes are reflected at the end.

- Snippet 7

html file

```
<button id="btn">Click</button>
<script>
  const btn = document.getElementById("btn");
  btn.addEventListener("click", handleClick());

  function handleClick() {
    alert("Clicked");
  }
</script>
```

Observations: The script executed succesfully and the alert “Clicked” is triggered when the page is loading, but nothing happens when the button is clicked. It is because in the event listener we are calling the function there and then itself. And then returns ***undefined*** since no return value is assigned to the function. So when the button is clicked and function is called again it is expecting a function in its second argument but instead it get undefined. The function call becomes : ***btn.addEventListener("click", undefined);*** So instead of calling the function there and then itself we can remove the parenthesis in the function call, this tell the browser to store this function and run it only when the click event occurs.

- Snippet 8

```
fetch("https://api.example.com/data")
  .then((res) => {
    res.json();
  })
  .then((data) => {
    console.log(data);
  });
```

Observations: The script executes successfully and the output will be ***undefined***. Even though the fetch request initiates correctly, the data is lost between the two ***.then()*** blocks. In the first ***.then()***, the code calls ***res.json()***, which is a method that returns a Promise containing the parsed data. However, because the arrow function uses curly braces ***{ }*** without a return statement, the function body executes but sends nothing back to the chain.

In JS, a function that doesn't explicitly return a value returns ***undefined*** by default. Therefore, the second ***.then()*** receives undefined as its data argument instead of the parsed JSON. To fix this, you must either add the ***return*** keyword before ***res.json()*** or remove the curly braces to use an implicit return: ***(res) => res.json()***. This ensures the "baton" is passed to the next stage of the Promise chain.

- Snippet 9

```
const nums = [1, 2, 3, 4];


const result = nums.map(👍 => {
  if (n % 2 === 0) {
    return n * 2;
  }
});


console.log(result);
```

Observations: Then code runs successfully and the output is ***[undefined, 4, undefined, 8]***. This is because the map function created the array of same length as the original, and hence expects value of each iteration. In case of even numbers, there is not issue; it simply executes the ***if*** block, but when value of ***n*** is odd, the function goes in the else block and there is no else block defined. So it returns undefined for odd number iterations.

- Snippet 10

```
const person = {

  name: "Amar",

  greet: () => {

    console.log("Hello " + this.name);

  },

};

person.greet();
```

Observations: The code executes successfully and the output will be ***Hello undefined***. Although it appears that name and greet are in the same location, the curly braces of an object literal do not create a scope for the arrow function to "bind" to.

In JS, arrow functions do not have their own this context; they inherit it from the surrounding lexical environment. Since the person object is defined in the global scope, the arrow function treats this as the global object (like window). Even though the code is physically inside the object, the object itself is not a scope. Therefore, ***this.name*** looks for a variable named name on the global object rather than the person object. To fix this and allow the function to "see" its parent object, you must use a regular function or method shorthand, which establishes a this binding to the object when called via ***person.greet()***.