Day-17 | JavaScript Output Based Question | github → rajeshjha2000

(81)
```
const person = {
  name: 'Rajesh',
};
Object.seal(person);
person.name = "FEDeveloper";
person.skill = "JS";
delete person.name;
console.log(person);
```

Ans:> 1. Object Initialization →
- The 'person' object is created with a single property 'name' set to 'Rajesh'.

2. Sealing the Object →
- 'Object.seal(person)' makes the 'person' object sealed. A sealed object prevents new properties from being added, and existing properties can't be deleted. However, the values of existing properties can still be modified.

3. Modifying Properties →
- 'person.name = "FEDeveloper"' This changes the 'name' property to "FEDeveloper" because modifying existing properties is allowed in a sealed object.
- 'person.skill = "JS"' This attempt to add a new property 'skill' will fail because adding new properties to a sealed object

is not allowed. In non-strict mode, this operation will fail silently without an error.

4. Attempting to Delete a Property →

- 'delete person.name', this operation will fail because properties can't be deleted from a sealed object. Again, in non-strict mode, this operation will fail silently.

Therefore, the output will be →

— {
  name: 'FEDeveloper'
}

(82) const handler = {
  set: () => console.log ('Added a new property!'),
  get: () => console.log ('Accessed a property!');
};

const person = new Proxy ({}, handler);

person.name = 'Rajesh';

person.name;

Ans:- In the code, we are using JS 'proxy' object to intercept and customize operations performed on the 'person' object. Specially, we have provided handlers for the 'set' and 'get' operations.

1. Handler Def →

- The 'set' handler is triggered whenever a property is added or modified on the 'Proxy' object.
- The 'get' handler is triggered whenever a property is accessed on the 'Proxy' object.

2. Proxy Creation →

- A new 'Proxy' object is created that wraps around an empty object '{}'.
- The 'handler' object is passed to the 'Proxy', meaning any 'set' or 'get' operation on the 'person' object will trigger the respective handler functions.

3. Setting a Property →

- When 'person.name = Rajesh' is executed, the 'set' handler is triggered, which results in the following output:
  - Added a new Property!
- Even though the handler doesn't return anything specific the property 'name' is still set to 'Rajesh' on the 'Proxy' object.

4. Getting a Property →

- When 'person.name;' is executed, the 'get' handler is triggered, which results in the following output:
- Accessed a property!

Therefore, the output will be →

- Added a new Property!
- Accessed a property!

(83) `const MESSAGE = 108;`

```
function getInfo() {
    console.log(MESSAGE);
    const MESSAGE = 'Hello world';
}
getInfo();
```

Ans:→ 1. Variable Declaration →

- The constant 'MESSAGE' is declared and initialized with the value '108'.

2. Function Def →

- Inside the 'getInfo' function, there's a local 'MESSAGE' variable that is declared using 'const'.
- However, the `console.log(MESSAGE);` statement comes before the local 'MESSAGE' is initialized.

Explanation of the Error:

- Hoisting → In JS, variable declaration (but not their initializations) are "hoisted" to the top of their scope.
- This means that within the 'getInfo' function, the declaration of `const MESSAGE` is hoisted to the top, but

it remains in the "Temporal Dead Zone" until the execution reaches the initialization line 'const MESSAGE = 'Hello world'.

- Accessing Before Initialization →
- When the 'console.log(MESSAGE)' statement is executed, JS attempts to access the local 'MESSAGE' variable.
- However, since the local 'MESSAGE' hasn't been initialized yet, attempting to access it results in a 'ReferenceError'.

Important Note →

- The 'MESSAGE' variable at the top level (with value '108') is not accessible within the function scope because the local 'MESSAGE' is declared, which shadows the outer 'MESSAGE' variable.

Therefore, the output will be →

- 'ReferenceError: Can't access 'MESSAGE' before initialization.

(84)
```
const myFunc = ({x, y, z}) => {
    console.log(x, y, z);
};

myFunc(1, 2, 3);
```

A:) • Function Def ⟶

'myFunc' is a function that expects a single argument, which should be an object containing properties 'x', 'y', 'z'.

• Function call ⟶

• Here, we are passing three individual arguments '1', '2' & '3' to the function.

• However, the function is expecting a single object argument. Since '1' is not an object, JS will try to destructure it as if it were an object, leading to an error.

Therefore, the output will be ⟶

• Type Error: Can't destructure property 'x' of '1' as it is not an object.

(85) `const add = x => y => z => {`

```
    console.log(x, y, z);
    return x + y + z;
};
add(10)(20)(30);
```

Ans:) 1. Function Def ⟶ 'add' is an arrow function that takes 'x' as its argument and returns another function that takes 'y' as its argument.

- The second function then returns another function that takes 'z' as its argument.
- The innermost function logs 'x', 'y' and 'z' to the console and then returns the sum of 'x', 'y' and 'z'.

2. Function Call →

- The function is invoked in a curried fashion:
- 'add(10)' returns a function that takes 'y'.
- '(20)' is passed to this returned function, which then returns another function that takes 'z'.
- '(30)' is passed to this final function, which executes the 'console.log' statement and returns the sum.

Execution steps →

- Step 1: 'add(10)' returns 'y' ⇒ z ⇒ {...}', with 'x' bound to '10'.
- Step 2: '(20)' is passed to the returned function, which returns 'z ⇒ {...}', with 'x' still '10' and 'y' bound to '20'.
- Step 3: '(30)' is passed to the final function, which logs the values and returns the sum:
  - 'console.log(x,y,z)' logs '10 20 30'
  - 'return x+y+z' returns '10+20+30' which is 60.

Therefore the output will be →

- 10 20 30

For more questions, visit

github → rajeshjha2000