Day-8

JavaScript Output Based Question

gitHub → rajeshjha2000

(36)
```
function Car() {
  this.make = 'tata';
  return {make: 'Kia'};
  const myCar = new Car();
  console.log(myCar.make);
```

A> 1. Function Def (`Car`): →

- The `Car` function is a constructor function. In JS, constructor function are used to create new objects.

- Inside the `Car` function:
- `this.make = 'tata';` is meant to set a property `make` on the new object being created.

- The `return {make: 'Kia'};` statement explicitly return an object with 'make' set to `'Kia'`.

2. Object Creation (`new Car()`): →

- When we call `new Car()`, it creates a new object.
- Normally, `this.make = 'tata';` would set the `make` property of this new object to `'tata'`.
- However, since the constructor function explicitly returns an object `{make: 'Kia'}`, this object is returned instead of the one that was implicitly created by 'new';

3. Logging the Object ⟶

- The 'myCar' variable now holds the object '{make: 'Kia'}' because that's what was returned from the constructor.
- 'console.log(myCar.make);' will log "Kia" because the 'make' property of the returned object is "Kia".

Therefore, the output will be ⟶

- Kia

Key Points ⟶

- Default Object Creation ⟶ Normally, a constructor function creates a new object, and 'this' refers to that new object.
- Explicit Return ⟶ If a constructor function explicitly returns an object, that object will be returned instead of the one created by 'new'. If nothing or a non-object (e.g., a string or a number) is returned, the default object is used.

(37)
```
(() => {
    let x = (y = 10);
    })();

    console.log(typeof x);
```

Ans 1. Immediately Invoked Function Expression (IIFE) ⟶

- '(() => {----})();' is an immediately Invoked Function Expression (IIFE). It's a function that is defined and then immdiately executed.

2. Inside the IIFE →

- Inside this IIFE

```
let x= (y=10);
```

- \`y= 10' assigns the value \`10' to \`y'. Since \`y' is not declared with \`let', \`const', or \`var', it becomes a global Variable.
- \`x' is declared with \`let' and assigned the value of \`y =10', so \`x' becomes \`10' within the IIFE's scope.

3. Outside the IIFE →

- After the IIFE has run, the variable \`x' defined inside it is not accessible outside because it was declared with 'let' which is block-scoped.
- Therefore, 'x' does not exist in global scope.

4. Logging \`typeof x': →

- When we try to log \`typeof x' outside the IIFE, \`x' is not defined in the global scope.
- In JS, when we check the type of an undeclared variable using 'typeof', it does not throw an error but returns "undefined".
- \`typeof' operator is designed in a way that it does not throw a \`Reference Error' even if the variable being checked is not declared in any scope (global or local) Instead, it returns \`"undefined"'.
- This is a safety feature of JS to allow us to check if variable is defined or not without causing our script to crash.

Scanned with CamScanner

Therefore, the output will be ⟶

- Undefined

Note ⟶ • console.log(x) ⟶ Reference Error
• console.log(typeof x) ⟶ Undefined

Note ⟶ • console.log(typeof y) ⟶ number

(38)
```
(() => {
    let x = 10;
  })();

(() => {
   let x = 10;
 })();

 console.log(typeof x);
```

A⟩ • The 'x' variable in both IIFEs is block-scoped, meaning it only exists within the function and is removed afterward.

- Outside of the IIFEs, there is no 'x' variable in the global scope.

Therefore, the output will be ⟶

- Undefined

(39)
```
let x = 100;
(() => {
  var x = 20;
})();
console.log(x);
```
A> • 'let' creates a block-scoped variable (or memory inside 'script').

• 'var' creates a function-scope inside function.

• The 'x' declared with 'var' inside the IIFE does not affect the 'x' declared with 'let' outside the IIFE because they are in different scopes.

Therefore, the output will be ⟶

• 100

(40) console.log(!true - true);

Ans> • The '!' operator is a logical NOT operator, which inverts the boolean value.

• '!true' becomes 'false'.

• In JS, 'false' is coerced to '0', and 'true' is coerced to '1' when performing arithematic operations.

Therefore, the output will be ⟶

• −1

(41) console.log (true + + "10");

Ans:-> • The '+' before the String "10" is a unary plus operator which attempts to convert the String into a number.

- "10" is converted to the number '10'.
- In JS, the boolean 'true' is coerced to the number '1' when used in arithematic operations.
- So, 'true + 10' becomes '1 to 10'.

Therefore, the output will be ->

- 11

For, more Questions, visit ->

gitHub -> rajeshjha2000