Day 2 | JavaScript Output Based Question

DELTA
Date / /
Pg No.

(6)
```
let data= "size";
const bird = {

    size : "small",

 };
console.warn (bird [data]);
console.warn (bird ["size"]);
console.warn (bird.size);
console.warn (bird.data);
```

A.> 1. console.warn (bird [data]);

- The Variable 'data' holds the string '"size"'.
- 'bird [data]' is equivalent to 'bird ["size"]'.
- Therefore, this will output the value associated with the key "size", which is "small".

2. console.warn (bird ["size"]);

- This accesses the value of the key "size" in the 'bird' object.
- The value associated with '"size"' is '"small"'.

DELTA
Date / /
Pg No.

3. console.warn(bird.size);

- This accesses the value of the 'size' property directly in the 'bird' object.

- The value of 'size' is '"small"'.

4. console.warn(bird.data);

- There is no property named 'data' in the 'bird' object.

- Accessing a non-existent property return 'undefined'.

Note → • JavaScript does not throw a 'Reference Error' if a property is missing from an object

• A 'ReferenceError' would occur if we try to access a variable that has not been declared, but accessing an undefined property on an object is handled gracefully with 'undefined'.

Therefore, the output will be –

- small
- small
- small
- undefined

Note → In JS, when we use a variable to access a property on an object, that variable's value determine which property is accessed.

- When we use `bird[data]`, we are using bracket notation to access a property of 'bird' object.

- In bracket notation, the expression inside the brackets is evaluated to get the property name.

- Since 'data' contains the string "size", 'bird[data]' is equivalent to 'bird["size"]'.

→ bird.size –
- This uses dot notation to access the 'size' property on the 'bird' object.

→ bird.data –
- Since 'data' is not a property on 'bird', 'bird.data' returns undefined.

Key Points →

- Bracket Notation – (object [propertyName]);

- Dot Notation – (object.propertyName);

(7) let c = {name: "Rajesh"};

let d;

d = c;

c.name = "Rajesh";
console.log(d.name);

A) 1. Object Creation → let c = {name: "Rajesh"};

• This creates an object 'c' with a property 'name' set to "Rajesh".

2. Variable Assignment → let d; d = c;

• The variable 'd' is assigned the value of 'c'. Both 'c' and 'd' now refer to the same object in memory.

3. Modifying the Object → c.name = "Rajesh";

• This changes the 'name' property of the object that both 'c' and 'd' refer to. The property 'name' is updated to "Rajesh".

4. Logging the property → since 'd' refers to the same object as 'c', the 'name' property of the object 'd' refers to is also "Rajesh".

DELTA
Date / /
Pg No.

Therefore, the output will be -
• Rajesh

Note → The reason both 'c' and 'd' reflect the updated value is because they both point to the same Object. When the Object's property is changed through one reference, it is reflected in all references to that object.

ex → c.name = "Jha";

```
console.log (d.name);
→ Jha  ← output
```

(8)
```
Var x;
Var x = 10;
console.log (x);
```

A) • The JS Engine hoists the declaration of the variable 'x' to the top of it's scope.

The code is interpreted as

```
Var x;   // declaration (hoisted)
x = 10;  // Initialization
```

DELTA
Date / /
Pg No.

- Var x = 10; is a declaration and initialization. The 'x' variable is first declared with var x and then it is assigned the value '10'.

Therefore, the output will be –
- 10

(Q)
```
Var x ;
let x = 10;

console.log(x);
```

A) • JavaScript does not allow redeclaring a variable with 'let' if it has already been declared with 'var' in the same scope.

- This causes a syntax error during the parsing phase before the code is executed.

Therefore, the output will be –

- SyntaxError : Identifier 'x' has already been declared.

Note: → Variable shadowing occurs when a variable declared within a certain scope has the same name as a variable declared in outer scope.

→ In same scope variable shadowing doesn't work.

ex →
```
var x = 5;    // Global scope
{
  let x = 10;   // Block scope, shadows the
                   global x
  console.log(x);
}
console.log(x);
```

(10)
```
let a = 3;
let b = new Number(3);

console.log(a == b);
console.log(a === b);
```

A7 • let a=3; ⟶
• 'a' is a primitive number with the value '3'.
• typeOf 'a' → number

• let b= new Number(3); ⟶
• 'b' is an instance of the 'Number' object, which is an object wrapper around the primitive number '3'.

• typeOf 'b' → object

→ • console.log (a==b); ⟶

• The '==' (loose equality) operator compares 'a' and 'b' for equality, converting them to common type if necessary.

• When comparing a primitive to an object, JS converts the object to it's primitive value before comparison.

• 'b' (a 'Number' object) is converted to it's primitive value ('3').

• Thus, '3 == 3' evaluates to 'true'.

→ console.log(a===b); →

- The '===' (strict equality) operator compares 'a' and 'b' for equality without type conversion.

- 'a' is a primitive number and 'b' is a 'Number' object.

- Since they are of different types (number VS object), the comparison evaluate to false.

Therefore, the output will be,

- true
- false