

Q1. Explain the difference between null and undefined in JavaScript.

Answer -

In JavaScript, null and undefined are both used to represent the absence of a value, but they are used in different contexts and have distinct meanings

Undefined:

- Meaning: undefined is a variable that has been declared but has not been assigned a value yet. It also represents the absence of a value when a function does not explicitly return a value.

```
• let x;  
• console.log(x);
```

output

PS D:\code\FSD\Day 2> node js_day2.js

undefined

Null:

- Meaning: null is an assignment value that represents the intentional absence of any object value. It is used when you want to explicitly indicate that a variable should not have any value.

```
• let y = null;  
• console.log(y);
```

output

PS D:\code\FSD\Day 2> node js_day2.js

null

Q2. What will be the output of the following code snippet, and why?

Answer –



```
console.log('10' + 5);  
console.log('10' - 5);  
console.log(true + 2);  
console.log(false + undefined);
```

```
Console.log('10' + 5);
```

The output for the following snippet is 105

In JavaScript the "+" operator is going to concatenate the both the values which results in as 105

```
Console.log('10'-5);
```

The output for the following snippet is 5

In JavaScript the "-" operator is going to function as subtraction and here js is going to convert the string to a number type to get the result.

```
Console.log(true + 2);
```

The output for the following snippet is 2

In JavaScript, when performing arithmetic operations, true is automatically converted to the number 1 which results in as 1 + 2 as 3.

```
Console.log(false + undefined);
```

The output for the following snippet is NaN(not a number)

In JavaScript false is converted to 0 when used in arithmetic operations, undefined cannot be converted to a meaningful number in arithmetic operations, so the result of adding 0 and undefined is NaN.

Q3. What is the difference between == and === in JavaScript? Provide examples.

In JavaScript == (Loose Equality) and === (Strict Equality) are both comparison operators, but they behave differently in terms of type conversion.

1. == (Loose Equality)

- Type Conversion: The == operator compares two values for equality after converting them to a common type, if necessary. This is known as type coercion.

```
• console.log(5 == '5');    // Output: true
• console.log(true == 1);   // Output: true
```


2. === (Strict Equality)

- No Type Conversion: The === operator compares two values for equality without converting them to a common type. The values must be of the same type and have the same value to be considered equal.

```
• console.log(5 === '5');   // Output: false
• console.log(true === 1);  // Output: false
```

In these examples, 5 == '5' returns true because the string '5' is converted to the number 5 before comparison. Similarly, true == 1 returns true because true is converted to 1. However, when using strict equality, 5 === '5' returns false because one is a number and the other is a string, so they are not of the same type. Likewise, true === 1 returns false because true is a Boolean and 1 is a number.

Q4. Predict the output of the following expressions and explain your reasoning



```
console.log(0 == false);  
console.log(0 === false);  
console.log('' == 0);  
console.log('' === 0);
```

`Console.log(0 == false);`

Output – here false is converted into 0 , on 0 comparison with 0 we get result as true.

`Console.log(0 === false);`

Output – here strict comparison is done without having any type conversion hence it results into false.

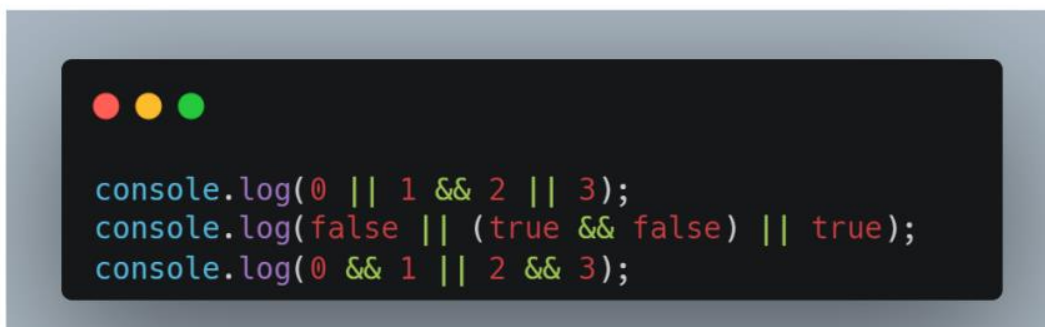
`Console.log('' == 0);`

Output – when comparing an empty string with something , the empty string is converted into 0 and it results in as true.

`Console.log('' === 0);`

Output – here no type conversions are done hence it results as false.

Q5. Given the following code, what will be the output and why?



```
console.log(0 || 1 && 2 || 3);  
console.log(false || (true && false) || true);  
console.log(0 && 1 || 2 && 3);
```

```
Console.log( 0 || 1 && 2 || 3);
```

Output –

Evaluate 1 && 2 first: Since both 1 and 2 are truthy, the result of 1 && 2 is 2. Now the expression becomes 0 || 2 || 3. Evaluate 0 || 2: 0 is falsy, so the result is 2. Evaluate 2 || 3: 2 is truthy, so the result is 2.

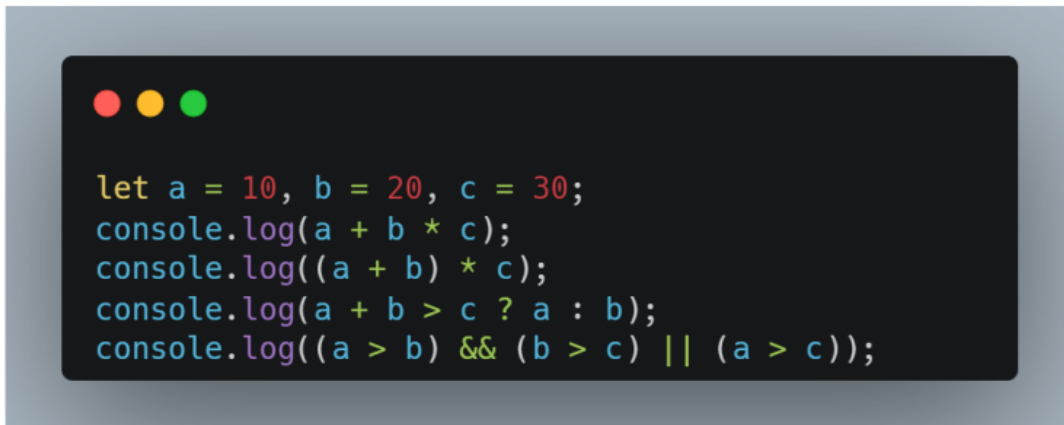
```
Console.log(0 == false);
```

Output –

```
Console.log("" == 0);
```

```
Console.log("" === 0);
```

Q6. Predict the output of the following expressions and explain your reasoning



```
let a = 10, b = 20, c = 30;
console.log(a + b * c);
console.log((a + b) * c);
console.log(a + b > c ? a : b);
console.log((a > b) && (b > c) || (a > c));
```

Let a = 10, b = 20, c = 30;

```
Console.log( a + b * c);
```

Output – here it follows bodmos rule, first multiplication is done and then addition is done

20 * 30 = 600 and 10 + 600 results as 610

```
Console.log((a+b) * c);
```

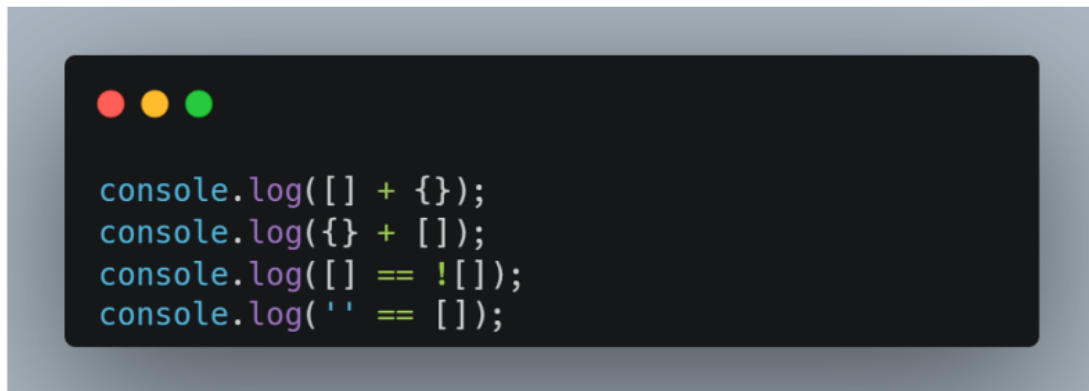
Output – here the a+b is added first as they are in inner bracket, 30 * 30 which results as 900

```
Console.log(a + b > c ? a : b);
```

Output – here a + b value is 30 its compared if its greater then c i.e, 30 since its false b is printed here as output - 20

```
Console.log((a > b) && (b > c ) || (a > c ));
```

Q7. Analyze and explain the output of the following code snippets



```
console.log([] + {});
console.log({} + []);
console.log([] == ![]);
console.log('' == []);
```

```
Console.log([] + {});
```

Output –

When using + operator with arrays and objects JavaScript first converts them into strings. Here [] is converted to an empty string and {} is converted into an empty object

Outputs as [object object]

```
Console.log({} + []);
```

Output –

JavaScript interprets {} as a block statement when it appears at the start of an expression. In this case, the {} is treated as an empty code block and not an object literal. The + [] is then evaluated as 0 because the empty array [] is coerced to 0 in numeric contexts.

```
Console.log([] == ![]);
```

Output –

Here, the ![] expression evaluates to false because [] (an empty array) is a truthy value, and ! negates it. So [] == false is evaluated next. In loose equality (==), [] is coerced to 0, and false is also coerced to 0. Hence, [] == false evaluates to true.

```
Console.log('' == []);
```

Output –

Here, the == operator is used to compare a string (") with an array ([]). The empty array [] is coerced to an empty string " when compared to a string using ==. Thus, " == [] evaluates to false because " (string) and [] (array) are not considered equal in loose equality.

Q8. What will be the output of the following code, and why?



```
console.log(+ "");  
console.log(+true);  
console.log(+false);  
console.log(+null);  
console.log(+undefined);
```

```
console.log(+ "");
```

output - The empty string "" is converted to a number and An empty string is treated as 0 when converted to a number.

```
console.log(+true);
```

output - The Boolean value true is converted to a number. true is treated as 1 when converted to a number.

```
console.log(+false);
```

output - The Boolean value false is converted to a number. false is treated as 0 when converted to a number.

```
console.log(+null);
```

output - The null value is converted to a number. null is treated as 0 when converted to a number.

```
console.log(+undefined);
```

output - The undefined value is converted to a number. undefined is treated as NaN (Not-a-Number) when converted to a number.