1. What is the difference between null and undefined in javascript?

In JavaScript, Null and Undefined are two distinct types that represent the absence of a value or a non-existent value, but they are used in different contexts and have different implications.

Null is a primitive type, but it's considered a special value in JavaScript.
It represents the intentional absence of any object value. It is explicitly assigned by the programmer to indicate "no value" or "empty."
It is used to explicitly indicate that a variable should have no value.

undefined is a primitive type.
It represents a variable that has been declared but not assigned a value.
It is the default value for uninitialized variables, function parameters that are not provided, and missing object properties.

—----------

2. console.log('10' + 5);

> **Explanation**: The `+` operator is used for both string concatenation and addition. When one of the operands is a string, JavaScript converts the other operand to a string and concatenates them.
>
> **Output**: '105'

console.log('10' - 5);

> **Explanation: The `-` operator is only used for arithmetic subtraction. JavaScript converts the string `"10"` to the number `10` and performs the subtraction.**
>
> **Output: 5**

> **console.log(true + 2);**
>
> **Explanation: In arithmetic operations, `true` is converted to the number `1`. So the expression becomes 1 + 2.**
>
> **Output: 3**

conole.log(false + undefined);

Explanation: In arithmetic operations, `false` is converted to the number `0`. However, `undefined` cannot be converted to a number in a meaningful way, so the result is `NaN` (Not-a-Number).

Output: NaN

—----------

3. What is the difference between == and === ? provide with an example.

`==` (Loose Equality)

- **Type Coercion: The `==` operator compares two values for equality after converting both values to a common type. This process is called type coercion.**
- **Usage: It's useful when you want to compare values without worrying about their types.**

`===` (Strict Equality)

- **No Type Coercion: The `===` operator compares two values for equality without converting them to a common type. It checks both the value and the type.**
- **Usage: It's useful when you want to ensure that the values being compared are of the same type and value.**

Using `==`:

javascript

Copy code

```
console.log(5 == "5");
```

- **Explanation: The string `"5"` is converted to the number `5` before the comparison, so the result is `true`.**

console.log(null == undefined);

Result is false

console.log(true == 1);

Using `===`:

javascript

Copy code

```
console.log(5 === "5");
```

- **Explanation: No type conversion occurs, so the number `5` and the string `"5"` are considered different types, resulting in `false`.**

console.log(5 === "5"); —--------->false

console.log(null === undefined); —--------------------> false

console.log(true === 1); —-------------------------> false

—--

4.

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

- **Explanation: The `==` operator performs type coercion before comparing the values. In this case, `false` is converted to `0`, so the comparison is between `0` and `0`.**
- **Output: true**

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

- **Explanation: The `===` operator does not perform type coercion. It checks both the type and the value. Here, `0` is a number and `false` is a boolean, so they are different types.**
- **Output: false**

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

- **Explanation: The `==` operator performs type coercion. In this case, the empty string `' '` is converted to the number `0` before the comparison. So, the comparison is between `0` and `0`.**
- **Output: true**

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

- **Explanation: The `===` operator does not perform type coercion. It checks both the type and the value. Here, `' '` is a string and `0` is a number, so they are different types.**
- **Output: false**

—----------

5.

console.log(0| | 1&&2 | |3))→ 2

1&&2 → 2

0 || 2 || 3 → 2

0 || 2 would return 2

console.log(false || (true && false) || true)→true

&& has higher precedence than ||

true && false would return false.

false || false would return false

false || true would return true

console.log(0 && 1 || 2 && 3)

&& has higher precedence than ||

0&&1→ returns to 0

2&&3→ returns to 3

0 || 3 → returns to 3

—----------

6.

1. `console.log(a + b * c);`

- **Order of Operations: JavaScript follows the standard order of operations (also known as PEMDAS/BODMAS), where multiplication and division are performed before addition and subtraction.**
- **Expression: `b * c` is calculated first.**
  - **`b * c` = `20 * 30` = `600`**
- **Then: Add `a`.**
  - **`a + 600` = `10 + 600` = `610`**

Output: `610`

2. `console.log((a + b) * c);`

- **Parentheses First: Operations inside parentheses are performed first.**
- **Expression: `(a + b)` is calculated first.**
  - `a + b` **=** `10 + 20` **=** `30`
- **Then: Multiply the result by `c`.**
  - `30 * 30` **=** `900`

Output: `900`

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

- **Order of Operations: First, the addition `a + b` is calculated, and then the comparison `a + b > c` is evaluated.**
- **Expression:**
  - `a + b` **=** `10 + 20` **=** `30`
  - **Now, the comparison:** `30 > c` **(i.e.,** `30 > 30`**)**
  - **The comparison** `30 > 30` **is** `false`**.**
- **Ternary Operator: The ternary operator `? :` is used to choose between two values based on a condition.**
  - **Since the condition (`a + b > c`) is `false`, the result is the second operand, `b`.**

Output: `20`

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

- **Order of Operations: `&&` (AND) has higher precedence than `||` (OR), so the expressions involving `&&` are evaluated first.**
- **Expression:**
  - `a > b` → `10 > 20` → `false`
  - `b > c` → `20 > 30` → `false`
  - **The `&&` operator between `false && false` evaluates to `false`.**
- **Now Evaluate the OR:**
  - **The expression reduces to `false || (a > c)`**
  - `a > c` → `10 > 30` → `false`
  - **Finally,** `false || false` **is** `false`**.**

Output: `false`

7.

1. `console.log([] + {});`

- Explanation:
  - The `+` operator in JavaScript, when used with non-numeric values, attempts to concatenate the values as strings.
  - An empty array `[]` when converted to a string becomes an empty string `""`.
  - An empty object `{}` when converted to a string becomes `"[object Object]"`.
  - Therefore, `[] + {}` becomes `"" + "[object Object]"` which results in `"[object Object]"`.

Output: `"[object Object]"`

2. `console.log({} + []);`

- Explanation:
  - In this case, JavaScript treats the `{}` at the beginning as a block of code rather than an object because there's no assignment or other context to suggest it should be treated as an object.
  - Thus, the code `{}` is treated as an empty block, and the remaining `+ []` results in `+[]`.
  - `+[]` converts the empty array to a number. An empty array when converted to a number results in `0`.

Output: `0`

3. `console.log([] == ![]);`

- Explanation:
  - This line involves type coercion and comparison.
  - `![]` results in `false` because the negation operator `!` converts the array to a boolean. Since an empty array is truthy, `![]` is `false`.
  - Now the expression is `[] == false`.
  - In JavaScript, when you compare an object (like `[]`) to a non-object (like `false`), the object is first converted to a primitive value.
  - An empty array when converted to a primitive value becomes an empty string `""`.
  - The comparison then becomes `"" == false`.

- `false` when converted to a string results in `""`, so the comparison is actually `"" == ""`, which is `true`.

Output: `true`

4. `console.log('' == []);`

- **Explanation:**
  - Here, we are comparing an empty string `""` with an empty array `[]`.
  - As in the previous example, the empty array `[]` is converted to an empty string `""` when it is compared with a primitive type.
  - Thus, the comparison is `"" == ""`.

Output:true

8.

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

- **Explanation:**
  - The `+` operator before a value attempts to convert the value to a number.
  - An empty string `""` when converted to a number results in `0`.

Output: `0`

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

- **Explanation:**
  - `true` when converted to a number results in `1`.
  - The `+` operator converts `true` to `1`.

Output: `1`

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

- **Explanation:**
  - `false` when converted to a number results in `0`.
  - The `+` operator converts `false` to `0`.

Output: `0`

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

- **Explanation:**
  - `null` when converted to a number results in `0`.
  - The `+` operator converts `null` to `0`.

Output: `0`

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

- **Explanation:**
  - `undefined` when converted to a number results in `NaN` (Not-a-Number).
  - The `+` operator converts `undefined` to `NaN`.

Output: `NaN`