Lesson:

Operators and Its Type

PW SKILLS

Topics Covered

1. Introduction to operators.
2. Operators & Operands.
3. Types of Operators in JavaScript.

Introduction to operators.

Basically, programming is done for a variety of reasons and one of the important reasons why programming is done is to operate on the values in the variables with high accuracy and faster speed.

Operators and operands

To perform these operators and operands. Operators, defines operation and the operation is performed on operands.

```
5 + 3 = 8
```

Here, we have to add **5 and 3 (operands)** to get the final value. So we are using **+ (operator)** to add these two values. And the final value is 8.

In the above example, the **+** sign performs an arithmetic operation to add these two values. So this **+** sign is the operator here. In modern programming languages, we use this **+** sign to add two values.

An **operand** is a data value that the operator will carry out the actions. It is the values on which we operate. So, in the above example, 5 and 3 are operands.

Operators are used in programming to perform operations on variables and values.

Types of Operators in JavaScript

1. **Assignment Operators**
2. **Arithmetic Operators**
3. **Comparison (Relational) Operators**
4. **Logical Operators**
5. **Bitwise Operators**

Assignment Operator

Assignment operators are used to assign values to variables.

Example:

```
var name = "PW Skills";

// Assignment Operator
var name = "PW Skills"; // name = "PW Skills"
```

The commonly used assignment operator is =. We will understand other assignment operators such as +=, -=, *=, etc. once we learn arithmetic operators.

Arithmetic Operators

We use arithmetic operators to do mathematical operations like addition, subtraction, multiplication, division, etc. It simply takes numerical values as operands, performs an arithmetic operation, and returns a numerical value.

- **Addition (+):** Adds two values together.
- **Subtraction (-):** Subtracts one value from another.
- **Multiplication (*):** Multiplies two values together.

- **Division (/):** Divides one value by another.
- **Modulus(%):** Returns the remainder of a division operation.
- **Exponentiation(**):** Raises to the power of.
- **Increment Operator(++):** Increases the value by 1.
- **Decrement Operator(--):** Decreases the value by 1.

In JavaScript, the **"++"** operator is used for both **pre-increment** and **post-increment** operations.

Pre-increment operator increments the value and returns the incremented value immediately. Post increment operator returns the original value itself and increments the value later. The same follows with Pre decrement and Post decrement operators.

```
// Arithmetic Operators

var num1 = 100;
var num2 = 2;

// 1. Addition
var result = num1 + num2; // 102

// 2. Subtraction
var result = num1 - num2; // 98

// 3. Multiplication
var result = num1 * num2; // 200

// 4. Division
var result = num1 / num2; // 50

// 5. Modulus
var result = num1 % num2; // 0
```

```
// 6. Exponential
var result = num1 ** num2; // 10000

// 7. Pre Increment
var num = 10;
var result = ++num; // num = 11 ; result = 11

// 8. Post Increment
var num = 10;
var result = num++; // result = 10 ; num = 11

// 9. Pre Decrement
var num = 10;
var result = --num; // num = 9 ; result = 9

// 10. Post Decrement
var num = 10;
var result = num--; // result = 10 ; num = 9
```

Adding two numbers, will return the sum, but adding a number and a string will return a string:

```
// Adding Two Numbers
num1 = 10;
num2 = 5;
result = num1 + num2; // 15
result_type = typeof result; // number

// Adding A Number and a string
num1 = 100;
str1 = "Welcome ";
result = str1 + num1; // Welcome 100
result_type = typeof result; // string

// Adding A Number and a string
num1 = 100;
str1 = "10";
result = str1 + num1; // 10100
result_type = typeof result; // string
```

Shorthands For Arithmetic and Assignment

We have shorthand operators for arithmetic and assignment as follows,

```
// 1. Add & Assign
var students = 100; // students = 100
students += 50; // students = 150

// 2. Subtract & Assign
var students = 100; // students = 100
students -= 50; // students = 50

// 3. Multiply & Assign
var students = 100; // students = 100
students *= 50; // students = 5000

// 4. Divide & Assign
var students = 100; // students = 100
students /= 50; // students = 2

// 5. Modulus & Assign
var students = 100; // students = 100
students %= 50; // students = 0

// 6. Exponential & Assign
var students = 100; // students = 100
students **= 2; // students = 10000
```

First arithmetic operation will be performed then the result will be assigned to the variable on the left hand side.

Relational Operators

Relational/Comparison operators compare two values and return a boolean value, either true or false.

The following are the comparison operators in javascript:

- **Equal (==):** Compares two values for equality, returns true if they are equal in value and false if they are not.
- **Strict equal (===):** Compares two values for equality and type, returns true if they are equal in value and of the same data type, and false if they are not.

- **Not equal (!=):** Compares two values for inequality, returns true if they are not equal and false if they are.
- **Strict not equal (!==):** Compares two values for inequality or type, returns true if they are not equal in value or not of the same data type, and false if they are.

Note: It is always advisable to use **strict equal(===)** and **strict not equal(!==)**, for equality because it checks types also, and makes sure that our code is safe from unexpected behaviors due to type mismatches.

- **Greater than (>):** Compares two values, returns true if the left operand is greater than the right operand, and false otherwise.
- **Greater than or equal to (>=):** Compares two values, returns true if the left operand is greater than or equal to the right operand and false otherwise.
- **Less than (<):** Compares two values, returns true if the left operand is less than the right operand and false otherwise.
- **Less than or equal to (<=):** Compares two values, returns true if the left operand is less than or equal to the right operand and false otherwise.

```
// Comparison Operators
var num1 = 10;
var num2 = 20;
var num3 = 10;

var str1 = "10";
var str2 = "20";

// 1. Equal to
var result = num1 == num2; // false
var result = num1 == num3; // true

// 2. Strict Equal
var result = num1 === num3; // true

var result = num1 === str1; // false

// 3. Not equal
var result = num1 != num2; // true

var result = num1 != num3; // false

// 4. Strict Not Equal
var result = num1 !== num3; // false

var result = num1 !== str1; // true
```

```
// 5. Greater than
var result = num1 > num3; // false
var result = num2 > num3; // true

// 6. Greater than or Equal to
var result = num1 ≥ num3; // true

var result = num2 ≥ num3; // true

// 7. Lesser than
var result = num1 < num3; // false

var result = num2 < num3; // false

// 8. Lesser than or Equal to
var result = num1 ≤ num3; // true
var result = num2 ≤ num3; // false
```

Logical Operators

Logical operators perform logical operations and return a boolean value, either true or false.

Logical operators in javascript:

- **Logical AND (&&):** This operator returns true if both operands are true, and false otherwise. It is a logical first operator, meaning that if the first operand is false, the second operand is not evaluated.

Possible cases:

operand 1	operand 2	operand1 && operand
true	true	**true**
false	true	false
true	false	false
false	false	false

- **Logical OR (||):** This operator returns true if at least one of the operands is true, and false otherwise. Like && operator, it is also a short-circuit operator.

Possible cases:

operand 1	operand 2	operand1 && operand
true	true	true
false	true	true
true	false	true
false	false	**true**

- **Logical NOT (!):** This operator inverts the Boolean value of the operand. If the operand is true, the operator returns false, and if the operand is false, the operator returns true.

Possible cases:

operand	!operand
true	false
false	true

These logical operators can be used to perform a variety of tasks, such as testing multiple conditions or combining multiple conditions in a single expression. They are often used with other techniques and operations to perform more complex tasks, such as flow control and data validation. We will be looking at their applications in further lectures.

```
// Logical Operators

var num1 = 10;
var num2 = 20;
var num3 = 10;

// 1. Logical AND
var result = num1 ≥ num3 && num1 == num3; // true
var result = num1 ≥ num2 && num1 == num3; // false

// 2. Logical OR
var result = num1 ≥ num3 || num1 == num3; // true
var result = num1 ≥ num2 || num1 > num3; // false

// 3. Logical NOT
var result = num1 == num3; // true
var result = !(num1 == num3); // false
```

Bitwise Operators: Bitwise operators perform operations on binary representations of numbers. Bitwise operators are particularly useful in low-level programming [programming that deals with the underlying hardware of a computer] and optimization, but they can also be used in a wide range of other areas, including data compression, encryption, and game development.

Bitwise operators in JavaScript are:

- **Bitwise AND (&):** This operator compares each bit of the first operand to the corresponding bit of the second operand, and if both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
- **Bitwise OR (|):** This operator compares each bit of the first operand to the corresponding bit of the second operand, and if at least one of the bits is 1, the corresponding result bit is set to 1.
- **Bitwise XOR (^):** This operator compares each bit of the first operand to the corresponding bit of the second operand, and if the bits are different, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
- **Bitwise NOT (~):** This operator inverts all the bits of the operand, effectively swapping 1s for 0s and 0s for 1s.
- **Left Shift (<<):** This operator shifts the bits of the operand to the left by the specified number of places, adding zeros to the right.
- **Right Shift (>>):** This operator shifts the bits of the operand to the right by the specified number of places, discarding the bits shifted out.

Examples

```
//1. AND (&) operator:
let a = 10;  // binary: 1010
let b = 6;   // binary: 0110
let result = a & b;  // binary: 0010 (decimal: 2)

// 2.OR (|) operator:
let a = 10;  // binary: 1010
let b = 6;   // binary: 0110
let result = a | b;  // binary: 1110 (decimal: 14)

// 3.XOR (^) operator:
let a = 10;  // binary: 1010
let b = 6;   // binary: 0110
let result = a ^ b;  // binary: 1100 (decimal: 12)

// 4.NOT (~) operator:
let a = 10;  // binary: 00000...1010(32 bits)
let result = ~a;  // binary: 11111...0101(32 bits)
(decimal: -11)

// 5.Left shift (<<) operator:
let a = 5;   // binary: 0101
let result = a << 2;  // binary: 010100 (decimal: 20)

// 6.Right shift (>>) operator:
let a = 20;  // binary: 010100
let result = a >> 2;  // binary: 0101 (decimal: 5)
```

An example use case of Bitwise operator, right shift and left shift as follows,

1. **Double any number : Left shift by 1**
2. **Half any number : Right shift by 1**

```
// Double any number : Left shift by 1
10 << 1  // 20
22 << 1 // 44

//Half any number : Right shift by 1
66 >> 1  // 33
100 << 1 // 50
```

Note: Binary operations are considered faster than decimal operations, because machines understand binary language easily.

Interview Point.

1. **Describe the difference between loose equality (==) and strict equality (===) operators in JavaScript.**

Ans: The difference between loose equality (==) and strict equality (===) operators in JavaScript lies in how they handle the comparison of values, taking into account both the value and the data type.

- **Loose Equality (==):**
  The loose equality operator performs type coercion, meaning it attempts to convert the operands to the same type before making the comparison. If the operands are of different types, JavaScript will try to make them the same type and then compare their values. This can lead to unexpected results, especially when comparing values of different types.

```
"5" == 5; // true (after type coercion, the string is
converted to a number)
false == 0; // true (false is coerced to 0)
null == undefined; // true (both are loosely equal)
```

- **Strict Equality (===):**

  The strict equality operator, on the other hand, compares both value and type without performing type coercion. It returns true only if both the value and the type of the operands are the same.

```
"5" = = = 5; // false (strict equality checks both value
and type)
false = = = 0; // false (boolean and number have
different types)
null = = = undefined; // false (null and undefined are of
different types)
```

It is recommended to use strict equality (===) unless there is a specific need for type coercion. Strict equality helps avoid subtle bugs that may arise from unexpected type conversions, making the code more reliable and easier to understand. Using strict equality also promotes better code quality and reduces the likelihood of errors related to implicit type coercion.

2. Why does typeof NaN return 'number'? How can you reliably check if a value is NaN?

Ans: The typeof operator in JavaScript returns 'number' for NaN because, according to the ECMAScript specification, NaN is considered a numeric value. It's important to note that NaN stands for "Not-a-Number," and its behavior as a numeric value might seem counterintuitive.

```
console.log(typeof NaN); // Output: 'number'
```

To reliably check if a value is NaN, you can use the isNaN function, which is specifically designed for this purpose. It returns true if the provided value is NaN, and false otherwise.

```
console.log(isNaN(NaN)); // Output: true
console.log(isNaN(42));  // Output: false
console.log(isNaN('hello')); // Output: true
```

3. How does the + operator behave when used with strings in JavaScript?

Ans: In JavaScript, the + operator behaves differently depending on the types of the operands involved. When used with strings, the + operator is overloaded to perform string concatenation.

- **String Concatenation:**

  If at least one of the operands is a string, the + operator concatenates the strings together, converting non-string operands to strings if necessary.

```
let str1 = "Hello";
let str2 = "World";
let result = str1 + str2;
console.log(result); // Output: "HelloWorld"
```

In this example, the + operator concatenates the strings "Hello" and "World" to produce the string "HelloWorld".

- **Type Coercion:**

  If one of the operands is not a string, it gets automatically converted to a string before concatenation.

```
let num = 42;
let str = "The answer is: " + num;
console.log(str); // Output: "The answer is: 42"
```

In this example, the number 42 is automatically converted to a string and then concatenated with the string.

- **Expression Involving Strings and Other Types:**

  If one of the operands is an expression, and it involves a string, the other operand(s) will be automatically converted to strings for concatenation.

```
let result = "The answer is: " + 42 + 7;
console.log(result); // Output: "The answer is: 427"
```

Here, the expression "The answer is: " + 42 is first evaluated, resulting in the string "The answer is: 42". Then, the next + 7 is also treated as string concatenation.

4. What are the edge cases to be aware of typeof?

Ans: While the typeof operator is a useful tool for checking the type of a value in JavaScript, there are some edge cases and nuances to be aware of:

- **typeof null returns 'object':**

  One of the most common surprises is that typeof null returns 'object'. This is considered a historical artifact and a mistake in the language. It doesn't mean that null is an object; it's a special primitive value. So, when using typeof, be cautious when checking for 'object' as it includes null.

```
console.log(typeof null); // Output: 'object'
```

- **typeof NaN returns 'number':**

  As mentioned earlier, typeof NaN returns 'number'. This might seem counterintuitive since NaN stands for "Not-a-Number," but it is considered a numeric value in JavaScript.

```
console.log(typeof NaN); // Output: 'number'
```

- **No differentiation between various objects:**

  typeof does not differentiate between different types of objects. All objects (including arrays and custom objects) are reported as 'objects'.

```
console.log(typeof [1, 2, 3]); // Output: 'object'
```

- **typeof undefined returns 'undefined':**

  The result is straightforward, but it's essential to remember that this check is not needed in most cases since accessing an undefined variable or property directly will not throw an error.

```
console.log(typeof undefined); // Output: 'undefined'
```

- **typeof with undeclared variables:**

  When using typeof with undeclared variables, it returns 'undefined'. This can be misleading, as it might suggest the variable has been declared but hasn't been assigned a value.

```
console.log(typeof name); // Output: 'undefined'
```