AIMERZ.ai
Aim.Act.Achieve

NOTES

Introduction to Operators

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

Operators are symbols or keywords that represent specific actions or operations to be performed on one or more operands.

```javascript
4 + 2 = 6
```

Here We have to add **4 and 2(Operands)** to get the final value. So we are using **+ (operator)** to add these two values. And the final value is **6**.

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, 4 and 2 are **operands**.

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

Different Types of Operators

1. Arithmetic Operators
2. Assignment Operator
3. Comparison Operator
4. Logical Operator
5. Unary Operator

1. 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 value together

```javascript
var num1 = 100;
var num2 = 50;

var sum = num1 + num2 // 150
```

Subtraction (-): Subtracts one value from another.

```javascript
var num1 = 100;
var num2 = 50;

var sub = num1 - num2 //50
```

Multiplication (*): Multiplies two values together.

```javascript
var num1 = 100;
var num2 = 2;

var result = num1 * num2 // 200
```

Division (/): Divides one value by another.

```javascript
var num1 = 100;
var num2 = 2;

var results = num1 / num2 // 50
```

Modulus(%): Returns the remainder of a division operation.

```javascript
var num1 = 100;
var num2 = 2;

var results = num1 % num2 // 0
```

Exponentiation(**): raises to the power of.

```javascript
var num1 = 100;
var num2 = 2;

var results = num1 ** num2 // 10000
```

Increment and Decrement(post & pre):

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

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.

```javascript
//Pre-increment

Example-1:

var num = 10;
var result = ++num

console.log(num) // 11
console.log(result) // 11

Example-2:

var num = 100;
var result = ++num

console.log(num)    //101
console.log(result) // 101
```

JavaScript

```
//Post-increment

Example -1:

var num = 10;
var result = num++

console.log(num)   //11
console.log(result) // 10

Example-2:

var num = 100;
var result = num++

console.log(num)   //101
console.log(result) //100
```

JavaScript

```
//Pre-Decrement

Example-1:

var num = 10;
var result = --num

console.log(num)   //9
console.log(result) //9
```

```
Example-2:

var num = 100;
var result = --num

console.log(num)  //99
console.log(result) //99
```

JavaScript

```
//Post-Decrement

Example-1:

var num = 10;
var result =num--

console.log(num)  //9
console.log(result) //10

Example-2:

var num = 100;
var result =num--

console.log(num)  //99
console.log(result) //100
```

Assignment Operator

Assignment operators do something with the value on the right side to set the variable on the left side or we can say Assignment operators are an essential aspect of JavaScript that allow you to assign values to variables and perform arithmetic operations in a concise manner. We can divided into two types:

a. **Simple Assignment**
b. **Compound Assignment**

Let's explore each of them with example to understand better

a. Simple Assignment(=):

The simple assignment operator is used to assign a single value to a variable. It takes the value on the right-hand side (RHS) and assigns it to the variable on the left-hand side (LHS).

Example-1:

```
JavaScript
let age = 54
console.log(age) // 54
```

In the above example, we used the simple assignment operator(=) to assign the value **54** to the variable **'age' .**

Example -2:

```
JavaScript

let x = 10;
console.log(x) //10
```

b. Compound Assignment:

Compound assignment operators combine arithmetic operations with assignment. These operators perform the specified arithmetic operation on the variable's current value and then assign the result back to the variable. They are a shorthand way of writing expressions.

Here are some common compound assignment operators in JavaScript:

- +=: Addition and assignment
- -=: Subtraction and assignment
- *=: Multiplication and assignment
- /=: Division and assignment
- %=: Modulus and assignment (remainder after division)
- **=: Exponentiation and assignment

Addition and assignment(+=)

```javascript
// Addition and assignment

let x = 6;
x += 4  // Equivalent to x = x + 4

console.log(x)   // Output: 10
```

In the example above, we used the **+= operator** to add 4 to the current value of x (which is 6) and assign the result (10) back to x.

Subtraction and assignment(-=)

```javascript
let y = 16
y -= 4   // equivalent to y = y - 4
console.log(y) //Output:12
```

In this example, we used the **-= operator** to subtract 4 from the current value of y (which is 16) and assign the result (12) back to y.

Multiplication and assignment(*=)

```javascript
let z = 3;
z *= 4;  // Equivalent to z = z * 4
console.log(z);  // Output: 12
```

In this example, we used the ***= operator** to multiply the current value of z (which is 3) by 4 and assign the result (12) back to z. After this operation, the value of z becomes 12.

Division and assignment(/=)

```javascript
let num = 15;
num /= 3;  // Equivalent to num = num / 3
console.log(num);  // Output: 5
```

In this example, we used the **/= operator** to divide the current value of num (which is 15) by 3 and assign the result (5) back to num. After this operation, the value of num becomes 5.

Modulus and assignment (remainder after division)[%=]

```javascript
let dividend = 17;
dividend %= 5;  // Equivalent to dividend = dividend % 5
console.log(dividend);  // Output: 2
```

In this example, we used the **%= operator** to find the remainder of dividing the current value of the dividend (which is 17) by 5 and assign the result (2) back to the dividend. After this operation, the value of dividend becomes 2.

Exponentiation and assignment(=)**

```javascript
let base = 2;
base **= 3;  // Equivalent to base = base ** 3
console.log(base);  // Output: 8
```

In this example, we used the ****= operator** to calculate the value of the current base (which is 2) raised to the power of 3 and assign the result (8) back to base. After this operation, the value of base becomes 8.

Comparison Operator

Comparison Operators are used to compare values and return Boolean results (true or false) based on the comparison.

Let's explore the four comparison operators: Equality (==, ===), Inequality (!=, !==), Greater than (>), Less than (<), Greater than or equal to (>=), and Less than or equal to (<=), along with the nullish coalescing operator (??).

1. **Equality (==) and Strict Equality (===):**
    - The **equality operator (==)** and **strict equality operator (===)** are used to compare two values for equality.
    - The equality operator (==) compares both the values. It returns true if both values are equal.
    - The strict equality operator (===) compares both the values and their types. It only returns true if both values and types are the same.

Let's take an example to understand better

```
JavaScript
Example-1:

let num1 = 5
let num2 = '5'

console.log(num1 == num2) // true // It only checks value
console.log(num1 === num2) //false // Both check values and
data type

Example-2:

console.log(0 === false); // Output: false
console.log(0 == false);  // Output: true

Example-3:

console.log('Prabir' == 'Prabir') //true
console.log('Prabir' === 'Prabir') //true
```

In the 2nd example,
The strict equality operator (===) compares the value and the type of the operands. In this case, 0 is a number, and false is a boolean. Since they are of different types, the comparison returns false, indicating that 0 is not strictly equal to false.

2. Inequality (!=) and Strict Inequality (!==):

The **inequality operator (!=)** compares two values to check if they are not equal and return a Boolean value.

But in the case of **Strict Inequality operator (!==)** checks both the values and their data types. It returns true only if both the values and the types are different.

Let's take an example to understand better

```
JavaScript
console.log('1' !=  1);  //false
console.log('1' !==  1);  //true
```

In this case, we are comparing the string '1' with the number 1. Even though they look similar, they are of different data types: '1' is a string, and 1 is a number. In the **Inequality Operator** only compares the value , so both the value is same ,So it gives false

But in the **Strict Inequality operator** It checks both value and data type. We are comparing the string '1' with the number 1. Since their data types are different (string vs. number), the strict inequality comparison immediately evaluates to **true.**

```
JavaScript
Example-2

console.log(true !== true); // false
console.log(true !== false); // true
```

```
console.log(true != true); // false
console.log(true != false); // true

Example-3

console.log(null !== undefined) //true
console.log(null != undefined)  //false
```

The **strict inequality operator (!==)** checks both the values and their data types. In this case, we are comparing null and undefined, which are distinct values of different types.

Since **null and undefined** are not only different values but also different types (null is of the "object" type, and undefined is of the "undefined" type), the strict inequality comparison returns **true.** This means that null is not strictly equal to undefined, and the statement null !== undefined is **true.**

Since both null and undefined are falsy values and are considered equal, So equality operator gives false.

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 behaviours due to type mismatches.

3. **Greater than (>), Less than (<), Greater than or equal to (>=), Less than or equal to (<=):**
    - The **greater than operator (>)** checks if the left operand is numerically greater than the right operand.
    - The **less than operator (<)** checks if the left operand is numerically less than the right operand.

- The **greater than or equal to operator (>=)** checks if the left operand is numerically greater than or equal to the right operand.
- The **less than or equal to operator (<=)** checks if the left operand is numerically less than or equal to the right operand.

```JavaScript
let num1 = 10;
let num2 = 5;

console.log(num1 > num2); // true, because 10 is greater than
5
console.log(num1 < num2); // false, because 10 is not less
than 5
console.log(num1 >= num2); // true, because 10 is greater than
or equal to 5
console.log(num1 <= num2); // false, because 10 is not less
than or equal to 5
```

4. **Nullish Coalescing (??):**

The nullish **coalescing operator (??)** is used to handle **null or undefined** values. It returns the **right-hand operand** if the **left-hand operand** is null or undefined; otherwise, it returns the left-hand operand.

Syntax:

```JavaScript
const result = someValue ?? defaultValue;
```

- **someValue:** This is the value that you want to check for null or undefined.

- **defaultValue:** This is the value that will be returned if someValue is either null or undefined.

Let's take an example to understand better:

```javascript
const firstName = null;
const lastName = "Kumar";

const fullName = firstName ?? "Prabir";

console.log(fullName); // Output: "Prabir" (since firstName is null, the defaultValue "Prabir" is used)
```

In this example, we have a variable firstName set to null and a variable lastName set to "Kumar". We use the nullish coalescing operator to assign a default value of "Prabir" to the variable fullName if firstName is null or undefined.

Since firstName is null, the nullish coalescing operator returns the defaultValue "Prabir", and as a result, the value of fullName becomes "Prabir".

Let's take another example to understand better

```javascript
const name = null;
const emptyText = ""; // falsy
const num = 34;

const value1 = name ?? "Prabir";
const value2 = emptyText ?? 'default value';
const value3 = num ?? 0;
```

```
console.log(value1); // Output: "Prabir" (name is null, so the
default value "Prabir" is used)
console.log(value2); // Output: "" (emptyText is an empty
string, which is not null or undefined, so it is used)
console.log(value3); // Output: 34 (num is 34, which is not
null or undefined, so it is used)
```

In this example, we have three variables: **name**, **emptyText**, and **num.**

- **const name = null;**
  The name variable is explicitly set to null.

- **const emptyText = "";**
  The emptyText variable is set to an empty string "", which is considered falsy in JavaScript, but it is not null or undefined.

- **const num = 34;**
  The num variable is set to 34.

Now, let's use the nullish coalescing operator (??) to set default values for each variable and examine the **output:**

- **const value1 = name ?? "Prabir";**
  The nullish coalescing operator checks the value of name. Since name is null, it returns the default value "Prabir". So, value1 is set to "Prabir".

- **const value2 = emptyText ?? 'default value';**
  The nullish coalescing operator checks the value of emptyText. Since emptyText is an empty string "", which is not null or undefined, the nullish coalescing operator does not replace it. So, value2 remains an empty string "".

- **const value3 = num ?? 0;**
  The nullish coalescing operator checks the value of num. Since num is 34, which is not null or undefined, the nullish coalescing operator does not replace it. So, value3 is set to 34.

Logical operators

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

There are **three** main logical operators: **AND (&&), OR (||), and NOT (!).**

i. AND Operator(&&):

- 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 2**
true	true	**true**
false	true	false
true	false	false
false	false	false

Let's take an example to understand how it is work

Example-1:

```javascript
var num1 = 20;
var num2 = 40;
var num3 = 20;
var result = num1 >= num3 && num1 == num3
console.log(result) // true
var results = num1 >= num2 && num1 == num3
console.log(results) //false
```

In this code, we have three variables: **num1, num2, and num3**, each assigned a numeric value.

- **var num1 = 20;**
- **var num2 = 40;**
- **var num3 = 20;**

Now, let's go through each statement:

1. var result = num1 >= num3 && num1 == num3;

In this line, we use the AND (&&) operator to combine two conditions:

- **num1 >= num3:** This condition checks if num1 is greater than or equal to num3, which is true (20 is greater than or equal to 20).
- **num1 == num3:** This condition checks if num1 is equal to num3, which is also true (both are 20).
- Since both conditions are true, the overall result of the AND operator is true. Therefore, result is assigned the value true.

2. var results = num1 >= num2 && num1 == num3;

In this line, we again use the AND (&&) operator to combine two conditions:

- **num1 >= num2:** This condition checks if num1 is greater than or equal to num2, which is false (20 is not greater than or equal to 40).
- **num1 == num3:** This condition checks if num1 is equal to num3, which is true (both are 20).
- Since one of the conditions (the first one) is false, the overall result of the AND operator is false. Therefore, the result is assigned the value false.

Let's take another example to checks whether a person is eligible to drive based on their age and whether they have a driving license:

```javascript
const age = 25;
const hasDrivingLicense = true;

const isEligibleToDrive = age >= 18 && hasDrivingLicense;

console.log(isEligibleToDrive); //true
```

- **const age = 25;**
  This line sets the value of the variable age to 25, representing the age of a person.

- **const hasDrivingLicense = true;**
  This line sets the value of the variable hasDrivingLicense to **true**, indicating that the person has a driving license.

- **const isEligibleToDrive = age >= 18 && hasDrivingLicense;**
  This line uses the AND (&&) operator to combine two conditions:

- **age >= 18:** Checks if the value of age is greater than or equal to 18. In this case, it's **true** since age is 25.
- **hasDrivingLicense:** Checks if the value of hasDrivingLicense is true.
- Since both conditions are **true**, the AND operator returns **true**, and the variable isEligibleToDrive is assigned the value true.

ii. Logical OR(||) Operator:

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 2**
true	true	true
false	true	true
true	false	true
false	false	**false**

Let's take an example to understand better:

Example-1:

```javascript
var num1 = 20;
var num2 = 40;
var num3 = 20;

var result1 = num1 >= num3 || num1 == num3;
console.log(result1)// true
var result2 = num1 >= num2 || num1 == num3;
console.log(result2)// true
var result3= num1 >= num2 || num1 > num3;
console.log(result3)// false
```

- **var result1 = num1 >= num3 || num1 == num3;**
  In this line, We use the OR (||) operator to combine two conditions:
- **num1 >= num3:** This condition checks if num1 is greater than or equal to num3, which is true (20 is greater than or equal to 20).
- **num1 == num3:** This condition checks if num1 is equal to num3, which is also true (both are 20).
- Since at least one of the conditions is true, the OR operator returns true, and result1 is assigned the value **true**.

- **var result2 = num1 >= num2 || num1 == num3;**
  In this line, you again use the OR (||) operator to combine two conditions:

- **num1 >= num2:** This condition checks if num1 is greater than or equal to num2, which is false (20 is not greater than or equal to 40).
- **num1 == num3:** This condition checks if num1 is equal to num3, which is true (both are 20).

- Since at least one of the conditions is true, the OR operator returns true, and result2 is assigned the value **true.**

- **var result3 = num1 >= num2 || num1 > num3;**
  In this line, the OR (||) operator combines two conditions:

- **num1 >= num2:** This condition checks if num1 is greater than or equal to num2, which is false (20 is not greater than or equal to 40).
- **num1 > num3:** This condition checks if num1 is greater than num3, which is also false (20 is not greater than 20).
- Since both conditions are false, the OR operator returns false, and result3 is assigned the value **false.**

Example-2:

```javascript
// Example 2: Checking Product Availability
const productInStock = true;
const productOnSale = false;

const isProductAvailable = productInStock || productOnSale;

console.log(isProductAvailable); // Output: true (since
productInStock is true)
```

In this code, We're using the OR (||) operator to determine whether a product is available for purchase. Here's what's happening step by step:

- **const productInStock = true;**
  This line sets the variable productInStock to true, indicating that the product is currently in stock.

- **const productOnSale = false;**
  This line sets the variable productOnSale to false, indicating that the product is not currently on sale.

- **const isProductAvailable = productInStock || productOnSale;**
  Here, you use the OR operator to combine the conditions:

- **productInStock:** This condition is true, as indicated by the value true.
- **productOnSale:** This condition is false, as indicated by the value false.
- Since at least one of the conditions is true (productInStock), the OR operator returns true. Therefore, the variable isProductAvailable is assigned the value **true.**

iii. 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.

```javascript
let num1 = 10
let num2 = 10

let result = !(num1 == num2)
console.log(result)  //false
```

We're using the NOT (!) operator to negate the result of a comparison between num1 and num2.

- **let num1 = 10;**
  This line initializes the variable num1 with the value 10.
- **let num2 = 10;**
  This line initializes the variable num2 with the value 10.
- **let result = !(num1 == num2);**
  In this line, we're using the NOT operator (!) to negate the result of the comparison num1 == num2. The comparison checks if num1 is equal to num2, which is true since both variables have the value 10. The NOT operator negates this true result, so the value of result will be **false**.

Example-2:

```javascript
const isDarkModeEnabled = false;

const isDarkModeDisabled = !isDarkModeEnabled;

console.log(isDarkModeDisabled); // Output: true (since
isDarkModeEnabled is false)
```

In this code, we're using the NOT (!) operator to negate the value of the isDarkModeEnabled variable. Here's what's happening step by step:

- **const isDarkModeEnabled = false;**
  This line initializes the variable isDarkModeEnabled with the value false, indicating that the user has not enabled dark mode.

- **const isDarkModeDisabled = !isDarkModeEnabled;**
  In this line, you're using the NOT operator (!) to negate the value of isDarkModeEnabled. Since isDarkModeEnabled is false, the NOT operator negates it to true. Therefore, the variable isDarkModeDisabled will be assigned the value **true.**

Topics Covered:

1. Unary Operator
2. Miscellaneous Operator
3. Number System
4. Bit-wise Operator

Unary Operator

Unary operators are operators that perform actions on a single operand. Let’s explore some of the Unary operators:

i. Unary Plus(‘+’):

The unary plus operator is used to convert its operand into a number. While this might seem redundant, it becomes valuable when dealing with user inputs that need to be explicitly converted to numbers.

Example - 1:

```javascript
let numString = "42";
let num = +numString; // Converts "42" to the number 42
console.log(num);
```

We are declaring a variable **numString** and assigning it the value "42", which is a string representing the characters "4" and "2".

Then, We use the **unary plus operator (+)** on numString to convert it into a **number.** This unary plus operation constrain the string "42" into the numeric value 42.

Finally, you assign the result of the unary plus operation to the variable num and then log its value using console.log(num), which will output 42 to the console.

Moreover , we use the unary operator to explicitly convert a string containing a numeric value into an actual number which can be useful when we want to ensure consistent data types for mathematical operations or comparison.

Example-2

```javascript
let strNumber = "5";
let number = +strNumber; // Unary plus converts the string "5"
to the number 5
console.log("Original String:", strNumber);
console.log("Converted Number:", number);
```

Output:

```
JavaScript
Original String: 5
Converted Number: 5
```

In this example, we have a strNumber variable containing the string "5". When we apply the unary plus operator to strNumber, it converts the string into a number. The converted value is stored in the number variable.

ii. Unary negation('-')

The unary negation operator is used to negate the value of its operand, effectively changing its sign.

Example-1:

```
JavaScript
let value = 10;
let negValue = -value;
console.log(negValue); // Output: -10
```

We are declaring a variable value and assigning it the value 10.

Then, We use the **unary negation operator (-)** on value to negate its value, effectively changing its sign from positive to negative.

Finally, you assign the negated result to the variable negValue and then log its value using console.log(negValue), which will output -10 to the console.

The unary negation operator simply changes the sign of a number. If the number is positive, it becomes negative, and if it's negative, it becomes positive.

Example -2

```javascript
let debt = -500; // Represents a debt of $500
let assets = 1000; // Represents assets worth $1000
let netWorth = debt + assets;
console.log(netWorth); // Output: 500 (positive net worth)
```

We have two variables: **debt and assets.** The variable debt represents a debt of $500, and the variable assets represents assets worth $1000.

Then, we calculate the netWorth by adding the debt and assets. Since the debt is represented as a negative value, adding it to the positive assets value will result in a positive net worth.

Finally, we log the netWorth using console.log(netWorth), which outputs 500, indicating a positive net worth of $500.

This example illustrates how the unary negation operator can be used to represent and manipulate financial values, such as debts and assets, in a real-world context.

iii. Logical NOT(‘!’):

The logical NOT operator is used to negate the Boolean value of its operand. It converts true to false and vice versa. It is a unary operator.We have already known the ‘**Logical NOT’ operator** in the previous section.

```javascript
let isSunny = false;
let isRainy = !isSunny; // Reverses the value of isSunny
console.log(isRainy); // Output: true
```

We are declaring a variable isSunny and assigning it the value false.

Then, we use the logical NOT operator (!) on isSunny. Since the initial value of isSunny is false, applying the logical NOT operator negates it, changing it to true.

Finally, We assign the negated value to the variable isRainy and log its value using console.log(isRainy), which outputs true.

This example demonstrates how the logical NOT operator can be used to reverse the truthiness of a Boolean value. When you negate false using the logical NOT operator, it becomes true

iv. Increment (++) and Decrement (--) Operators

The increment (++) and decrement (--) operators are used to increase or decrease the value of a variable by 1, respectively.

```javascript
//increment Operator

let count = 5;
count++; // Increment count by 1
console.log(count); // Output: 6
```

```javascript
// Decrement Operator

let inventory = 10;
inventory--; // Decrease inventory by 1
console.log(inventory); // Output: 9
```

v. typeof Operator

The **typeof operator** is used to determine the data type of its operand or we can say it allows checking the data type of a given variable.

```javascript
let name = 'Prabir'
console.log(typeof(name)) // string

var age =45
console.log(typeof(age)) //number

var couponCode = null
console.log(typeof(couponCode)) //object

var endDate;
console.log(typeof(endDate)) //undefined

var enrolledToFSWD = true;
console.log(typeof(enrolledToFSWD)) //boolean
```

vi. void operator

The void operator is used to evaluate an expression and return undefined.

In JavaScript, the void operator is used to explicitly return undefined. It's a unary operator, meaning only one operand can

be used with it. You can use it like shown below — standalone or with a parenthesis.

```javascript
void expression;
void(expression);
```

Lets take some examples to understand better

```javascript
void 0; //returns undefined
void(1); //returns undefined

void 'hello'; //undefined
void {}; //undefined
void []; //undefined
```

In The above ,

- **void 0:** This takes the value 0, but when we use the void operator on it, it returns undefined. This can be useful when you want to ensure that a numeric value like 0 doesn't have any unintended consequences.

- **void(1):** it takes the value 1, but the void operator makes it return undefined.
- **void 'hello':** It takes the string 'hello', but using the void operator on it makes it return undefined. This can be used when you want to use a string but don't need its value for any purpose.

- **void {}:** It takes an empty object {}, and again, the void operator makes it return undefined. This can be used when you want to perform some action using an object but don't need the result.

- **void []:** Similar to the previous example, it takes an empty array [], and the void operator makes it return undefined. This can be used when you want to execute code involving an array without caring about the array's contents.

Why do we need a special keyword just to return undefined instead of just returning undefined?

The reason is that before ES5 we could actually name a global variable undefined, like so: var undefined = "hello" or var undefined = 23, and most browsers would accept it; the identifier undefined was not promised to actually be undefined. So, to return the actual undefined value, the void operator is used.

Vii. delete Operator

The delete operator is used to delete an object's property or an element of an array.

Deleting an Object Property

```javascript
let person = { name: "Prabir", age: 25 };
delete person.age; // Delete the "age" property
console.log(person); // Output: { name: "Prabir" }
```

Deleting an array element

```javascript
let numbers = [1, 2, 3];
delete numbers[1]; // Delete the element at index 1
console.log(numbers); // Output: [1, empty, 3]
```

I know we have not covered the object and array part yet.. We will learn more in detail in the upcoming lecture.

Miscellaneous Operator

JavaScript provides a variety of operators that allow you to perform different operations on values. In this documentation, we will cover miscellaneous operators, including the **ternary operator**, **string operator**, **comma operator**, **member access operator**, and **instanceof**.

Ternary operator

The ternary operator is a shorthand way to write simple conditional statements.

syntax

```javascript
condition ? expr1 : expr2
```

condition: A boolean expression that is evaluated.
expr1: The value to return if the condition is true.
expr2: The value to return if the condition is false.

Example

```javascript
const age = 18;
const canVote = age >= 18 ? 'Yes' : 'No';

console.log(`Can you vote? ${canVote}`); // Output: Can you vote? Yes
```

String Operator (string concatenation +)

String operators in JavaScript are used to concatenate strings.

syntax

```javascript
str1 + str2
```

str1: The first string to concatenate.
str2: The second string to concatenate.

Example

```javascript
const firstName = 'John';
const lastName = 'Doe';
const fullName = firstName + ' ' + lastName;

console.log(fullName); // Output: John Doe
```

Comma

The comma operator allows you to evaluate multiple expressions (from left to right), returning the result of the last expression.

Syntax

```JavaScript
expr1, expr2, expr3, ..., exprN
```

expr1, expr2, expr3, ..., exprN: Expressions to evaluate, separated by commas.

Example

```JavaScript
let x = 1;
(x++, x += 2, x *= 3);

console.log(x); // Output: 12
```

Member Access Operators

Member access operators are used to access properties and methods of objects.

Dot notation:

```JavaScript
object.property
```

Bracket notation:

```javascript
object['property']
```

object: The object from which you want to access a property or method.
property: The name of the property or method.

Example

```javascript
const person = {
  firstName: 'Vishwa',
  lastName: 'Mohan',
};

console.log(person.firstName); // Output: Vishwa
console.log(person['lastName']); // Output: Mohan
```

Instance Of Operator

The instance of operator is used to check if an object is an instance of a particular class or constructor function. We will study about classes and constructor in details in later modules.

Syntax:

```javascript
object instanceof constructor
```

object: The object to test.
constructor: The constructor function to test against.

Example:

```javascript
class Animal {
  constructor(name) {
    this.name = name;
  }
}

const dog = new Animal('Rex');
const cat = { name: 'Whiskers' };

console.log(dog instanceof Animal); // Output: true
console.log(cat instanceof Animal); // Output: false
```

Number System

Numbers are a fundamental concept in mathematics and computing. In JavaScript, you encounter various number systems, and understanding them is crucial for effective programming.

What is a number system?

A number system, also known as a numeral system, is a mathematical notation for expressing numbers. It defines how we represent and manipulate numbers using symbols and rules. The two most commonly encountered number systems are the decimal system and the binary system.

Types of number system.

- **Decimal (Base 10):**
  The decimal number system is the most common number system in everyday use. It uses ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

- **Binary (Base 2):**
  The binary number system uses only two digits: 0 and 1. It is commonly used in computing and digital electronics.

- **Octal (Base 8):**
  The octal number system uses eight digits: 0, 1, 2, 3, 4, 5, 6, and 7. It's less commonly used today but can still be found in some computer systems.

- **Hexadecimal (Base 16):**
  The hexadecimal number system uses sixteen digits: 0-9 and A-F (or a-f). It's often used in computing for representing binary values more compactly.

- **Ternary (Base 3):**

The ternary number system uses three digits: 0, 1, and 2. It's used in some mathematical and computer science contexts.

- **Quaternary (Base 4):**
  The quaternary number system uses four digits: 0, 1, 2, and 3. It's less common than binary, octal, or hexadecimal.

Each of these number systems has its own set of rules and applications. The choice of a number system depends on the specific use case and the requirements of a given problem or application.

Conversion of Binary to Decimal

Binary and decimal are two different number systems used in computing. Binary uses only two digits (0 and 1), while decimal uses ten digits (0-9). Converting binary to decimal is a fundamental skill in computer science and programming.

To convert a binary number to decimal, you follow these steps:

1. **Write Down the Binary Number:** Start with the binary number you want to convert.
2. **Assign Powers of 2:** Assign powers of 2 to each bit in the binary number, starting with 2^0 for the rightmost bit and increasing by 1 as you move to the left.

3. **Calculate the Values:** Calculate the value of each term by multiplying the bit value (0 or 1) by 2 raised to the assigned power.
4. **Sum the Values:** Add up all the values you calculated in step 3. The sum is the decimal equivalent of the binary number.

Now, let's look at some examples.

Example 1: Convert 1101 (Binary) to Decimal

1. Start with the binary number 1101.
2. Assign powers of 2 from right to left: 2^0, 2^1, 2^2, and 2^3.
3. Calculate the values: 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0.
4. Sum the values: 8 + 4 + 0 + 1 = 13.

So, **1101** in binary is equal to **13** in decimal.

Example 2: Convert 10101 (Binary) to Decimal

1. Start with the binary number 10101.
2. Assign powers of 2 from right to left: 2^0, 2^1, 2^2, 2^3, and 2^4.
3. Calculate the values: 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0.
4. Sum the values: 16 + 0 + 4 + 0 + 1 = 21.

So, **10101** in binary is equal to **21** in decimal.

Example 3: Convert 11100 (Binary) to Decimal

1. Start with the binary number 11100.
2. Assign powers of 2 from right to left: 2^0, 2^1, 2^2, 2^3, and 2^4.
3. Calculate the values: 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0.
4. Sum the values: 16 + 8 + 4 + 0 + 0 = 28.

So, **11100** in binary is equal to **28** in decimal

Conversion of decimal to binary.

The binary number system uses only two digits (0 and 1) to represent numbers, which is different from the decimal system that uses ten digits (0-9).

To convert a decimal number to binary, you can use a method known as "repeated division by 2." Here's how it works:

1. **Start with the Decimal Number:** Begin with the decimal number you want to convert.
2. **Divide by 2:** Divide the decimal number by 2.
3. **Record the Remainder:** Write down the remainder (0 or 1) that you get when you divide by 2.
4. **Repeat the Process:** Continue dividing the quotient from the previous step by 2, and record the remainders.
5. **Read the Remainders Backwards:** To get the binary representation, read the remainders from the last division to the first. This is your binary number.

Now, let's look at some examples.

Example 1: Convert 26 (Decimal) to Binary

1. Start with the decimal number 26.
2. Divide by 2: Quotient = 13, Remainder = 0.
3. Divide by 2: Quotient = 6, Remainder = 1.
4. Divide by 2: Quotient = 3, Remainder = 0.
5. Divide by 2: Quotient = 1, Remainder = 1.
6. Divide by 2: Quotient = 0, Remainder = 1.
7. Read the remainders from the bottom up: 11010.

So, 26 in decimal is equal to 11010 in binary.

Example 2: Convert 14 (Decimal) to Binary.

1. Start with the decimal number 14.
2. Divide by 2: Quotient = 7, Remainder = 0.
3. Divide by 2: Quotient = 3, Remainder = 1.
4. Divide by 2: Quotient = 1, Remainder = 1.
5. Divide by 2: Quotient = 0, Remainder = 1.
6. Read the remainders from the bottom up: 1110.

So, **14** in decimal is equal to **1110** in binary.

Example 3: Convert 37 (Decimal) to Binary.

1. Start with the decimal number 37.
2. Divide by 2: Quotient = 18, Remainder = 1.
3. Divide by 2: Quotient = 9, Remainder = 1.

4. Divide by 2: Quotient = 4, Remainder = 0.
5. Divide by 2: Quotient = 2, Remainder = 0.
6. Divide by 2: Quotient = 1, Remainder = 1.
7. Divide by 2: Quotient = 0, Remainder = 1.
8. Read the remainders from the bottom up: 100101.

So, **37** in decimal is equal to **100101** in binary.

How to Convert Binary to Decimal and Vise-versa in JavaScript

Now let's see the conversion of binary to decimal and conversion of decimal to binary in JS using **parseInt.**

parseInt is a built-in JavaScript function that is used to parse a string and convert it into an integer (a whole number). It is often used for converting string representations of numbers, including binary, octal, decimal, or hexadecimal, into their numeric equivalents.

The basic syntax of parseInt is as follows:

Syntax

```JavaScript
parseInt(string, radix);
```

Here's an example of binary-to-decimal conversion and decimal-to-binary conversion using parseInt:

Binary to Decimal Conversion (JavaScript Example)

Ex:

```javascript
// Convert binary (base 2) to decimal (base 10)
const binaryValue = "1101";
const decimalEquivalent = parseInt(binaryValue, 2);
console.log(`Binary: ${binaryValue} => Decimal:
${decimalEquivalent}`);  // Binary: 1101 => Decimal: 13;
```

In this example:

1. We have a binary value, "1101".
2. We use parseInt(binaryValue, 2) to convert it to a decimal value. The 2 specifies that the input is in base 2 (binary).

Decimal to Binary Conversion (JavaScript Example)

Ex:

```javascript
// Convert decimal (base 10) to binary (base 2)
const decimalValue = 26;
const binaryEquivalent = parseInt(decimalValue,
10).toString(2);
console.log(`Decimal: ${decimalValue} => Binary:
${binaryEquivalent}`);  //Decimal: 26 => Binary: 11010;
```

In this example:

1. We have a decimal value, 26.
2. We use parseInt(decimalValue, 10) to ensure that JavaScript interprets it as a decimal value (base 10).

3. Then, we use .toString(2) to convert the decimal value to a binary string representation.

Bit-wise Operator

Introduction

Bitwise operators are used in programming to perform operations on individual bits of binary numbers. These operators are commonly used in tasks that require low-level manipulation of data, such as working with hardware, optimizing code, and cryptography. In this documentation, we will cover the most commonly used bitwise operators in programming.

Bitwise AND (&)

The bitwise AND operator (`&`) compares each bit of two integers and returns a new integer where each bit is set to 1 only if both corresponding bits in the original integers are 1. Otherwise, it sets the bit to 0.

Syntax

```JavaScript
result = operand1 & operand2;
```

Example

```JavaScript
const a = 12;    // binary: 1100
```

```
const b = 25;    // binary: 11001

const result = a & b;  // binary result: 1000 (decimal 8)
```

Bitwise OR (|)

The bitwise OR operator **(|)** compares each bit of two integers and returns a new integer. In the resulting integer, each bit is set to 1 if at least one of the corresponding bits in the original integers is 1.

Syntax

```
JavaScript
result = operand1 | operand2;
```

Example

```
JavaScript
const a = 12;    // binary: 1100
const b = 25;    // binary: 11001

const result = a | b;  // binary result: 11101 (decimal 29)
```

Bitwise XOR (^)

The bitwise XOR operator **(`^`)** compares each bit of two integers and returns a new integer where each bit is set to 1 if only one of the corresponding bits in the original integers is 1.

Syntax

```JavaScript
result = operand1 ^ operand2;
```

Example

```JavaScript
const a = 12;    // binary: 1100
const b = 25;    // binary: 11001

const result = a ^ b;  // binary result: 10101 (decimal 21)
```

Bitwise NOT (~)

The bitwise NOT operator (`~`) is a unary operator that flips the bits of an integer, changing 1s to 0s and 0s to 1s.

Syntax

```JavaScript
result = ~operand;
```

Example

```JavaScript
const a = 12;    // binary: 1100

const result = ~a;   // binary result: 11110011 (decimal -13)
```

Bitwise Left Shift (<<) and Right Shift (>>)

Bitwise left shift (`<<`) and right shift (`>>`) operators are used to shift the bits of an integer to the left or right by a specified number of positions.

Syntax

```javascript
result = operand << num_bits;  // Left shift
result = operand >> num_bits;  // Right shift
```

Example

```javascript
const a = 12;    // binary: 1100

const result1 = a << 2;  // Left shift by 2: binary result:
110000 (decimal 48)
const result2 = a >> 1;  // Right shift by 1: binary result:
110 (decimal 6)
```

Conclusion

Bitwise operators are essential tools for performing low-level bit manipulation in programming. They are commonly used in scenarios where you need to optimize code, work with binary data, or interface with hardware. Understanding how these operators work and when to use them is an important skill for any programmer.

THANK YOU

AIMERZ.ai
Aim.Act.Achieve

Stay updated with us!

Vishwa Mohan

Vishwa Mohan

vishwa.mohan.singh

Vishwa Mohan