Basics of JS

JS DATA TYPES

There are **7 data types** available in JavaScript. You need to remember that JavaScript is **loosely typed**(or **dynamic language**), so any value can be assigned to variables in JavaScript.

According to latest ECMAScript standard there are **6 primitive data types** and **1 object** -

- **Number** - represents integer and floating values
- **String** - represents textual data
- **Boolean** - logical entity with values as 'true' and 'false'
- **Undefined** - represents variable whose value is not yet assigned
- **Null** - represents intentional absence of value
- **Symbol** - represents unique value
- **Object** - represents key value pair

All of them except Object have **immutable values**, i.e. the values which cannot be changed.

Some important information about the data types have been described below -

- **Number** represents variables whose value is either an integer or float. It can have numbers in range between **$-(2^{53}-1)$** and **$(2^{53}-1)$**. Other than integer and float numbers, it has three symbolic values: `+Infinity`, `-Infinity`, and `NaN`. It uses 2 constants - `Number.MAX_VALUE` and `Number.MIN_VALUE`. Both of these constants lie between +Infinity and -Infinity. Eg., 10, -25.2, 8.321.

- **String** represents textual data. String contains elements that can be accessed using the **index**. The **first element has index 0**. Eg., "hello", "1234", "12here". You can access each element of string like - `str="HelloWorld"`, then `str[1]` will output 'e' on the console.

SOME IMPORTANT VALUES

Below are some important values that are defined in brief-

- **undefined**

  'undefined' is the value assigned to the variable that has not yet been assigned any value. We can also **explicitly assign 'undefined' value** to a variable, but that does not make any sense due to its meaning.
  Eg., `var a;` defines a variable that has not been assigned any value. If you write on the console as - `console.log(a);`, then **'undefined'** is printed.

- **null**

  'null' is the value that represents a reference that points to a non existent object or address. This means that there is an absence of a value. The **data type for null value is "Object".**

- **NaN**

  'NaN' means **Not-A-Number**. So, if any e***xpression fails to return a number***, then ***'NaN'*** is printed on the console. Eg., (`12 - "abc"`) cannot be evaluated to a number, so 'NaN' is printed on the console. The ***data type for 'NaN' is "Number".***

- **Infinity**

  Infinity is a variable in global scope. Its ***value is greater than any other number***. It literally works as Infinity, i.e. it has the same property as in Mathematics. Any numeric expression evaluated to a number outside the range of the number, prints 'Infinity' on the console.

TYPEOF OPERATOR

'typeof' operator is used to ***return the name of the data type to which the value belongs to***. It returns the name of the data type in the form of a string. There are two syntax for using **typeof** operator -

```
typeof operand
```

OR

```
typeof(operand)
```

The parentheses are optional. Operand can be a ***variable*** or ***some value*** or ***an expression.*** Below are some of the data types and their return values of 'typeof' -

S.No.	Data Type	Return value of 'typeof'
1.	Number	"number"
2.	String	"string"
3.	Boolean	"boolean"
4.	Undefined	"undefined"
5.	Null	"object"
6.	Symbol	"symbol"
7.	Object	"object"

EXTRA:
You can read typeof in detail from the below link -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof

OPERATORS IN JS

Operators are used to perform some action on operands. Operand can be of any data type, eg. number, boolean, string, etc..

Some of the operators are described below -

- **Arithmetic Operators**

  Arithmetic operators take numerical values as operands and evaluate them to a single numerical value. Some of the arithmetic operators are -

- **Addition(+)** - It adds the numerical operands and is also used for string concatenation. It will add numerical and string operand to a number if the string can be converted to a number. Else it concatenates them.
- **Subtraction(-)** - It subtracts the two numeric operands. If any one of them is not a number or cannot be converted to a number, then **'NaN'** is printed.
- **Division(/)** - It divides the first operand with the second operand. If the second operator is '+0' or '-0', then **'Infinity'** and **'-Infinity** are printed respectively. If they are not divisible **'NaN'** is printed. Try dividing 0 by 0 and see what happens!
- **Multiplication(*)** - It multiplies the two numeric operands. Any number multiplied with Infinity prints **'Infinity'**. See what happens when **Infinity** is multiplied with **0**.
- **Remainder(%)** - It finds the remainder left after division. If one of the operands is **'Infinity'** or **'NaN'**, then **'NaN'** is printed.

EXTRA:

You can learn more about arithmetic operator from this link - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators

- **Assignment Operators**

Assignment operators are used to **assign value of the right operand/expression to the left operand.** The simplest assignment operator **is equal (=),** which assigns the right operand value to left operand.
Eg., `var x=10; var y=x;` will assign value 10 to y.

Other assignment operators are ***shorthand operations of other operators*.** They are called **compound assignment operators.** Some of them with their meaning (i.e. the extended version of these operations) are provided below -

S.No.	Name	Shorthand Operator	Meaning
1.	Addition Assignment	x +=y	x = x + y

2.	Division Assignment	x /=y	x = x / y
3.	Exponentiation Assignment	x **=y	x = x ** y
4.	Right Shift Assignment	x >>=y	x = x >> y
5.	Bitwise XOR Assignment	x ^=y	x = x ^ y

***EXTRA*:**

To look for other assignment operators, visit the link below - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Overview

- **Increment and Decrement Operator**

The **increment operator** increments the value of the numeric operand by one.
The **decrement operator** decreases the value of the numeric operand by one.

There are two ways to use increment and decrement -

- **Using postfix (x++ or x--)** - the value is returned first and then the value is incremented or decremented.
- **Using prefix (++x or --x)** - the value is first incremented or decremented and then returned.

Eg., `var x=10; console.log(++x);` will print **11** on the console.

- **Comparison Operators**

The **comparison operators** are used to ***compare two values with each other.***
The **equality operator (==)** is used to ***compare the two values***, if they are equal or not. But the values are not directly compared. First, ***they are converted to the same data type and then the converted content is compared.***
Eg., `"1" == 1` evaluates to `true`, even though the first one is a 'string' and the other is a 'number'.

There is another comparison operator **(===)** known as a **strict equality operator**. It ***checks both the data type and the content***. If the data type is not equal, it returns false.
So `"1" === 1` now evaluates to `false`.

Other comparison operators are -

- **Inequality (!=)** - It returns the opposite result of the equality operator.
- **Strict Inequality (!==)** - It returns the opposite result of the strict equality
- **Greater Than (>)** - It returns true if left operand is greater than the right one
- **Greater Than or Equal (>=)** - It returns true if the left operand is greater than or equal to the right one.
- **Less Than (<)** - It returns true if left operand is less than the right one
- **Less Than or Equal (<=)** - It returns true if the left operand is less than or equal to the right one.

EXTRA:

You can get a more detailed comparison and working of equality operators from the link below -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators

- **Logical Operators**

The **logical operators** use boolean values as their operands. These operands are mostly expressions that evaluate to ***'true'*** or ***'false'***.

There are three logical operators -

- **Logical AND (&&)** - returns ***'true'*** if both operands/expression are true, else ***'false'***. If the first expression is false, the second expression is not evaluated and ***'false'*** is returned. Eg., `false && true` returns `false`.

- **Logical OR (||)** - returns ***'true'*** if any of the operands/expression is true, else ***'false'.*** Eg., `false || true` returns `true`.

- **Logical NOT (!)** - returns the opposite boolean value to which the expression is evaluated to. Eg., , `!false` returns `true`.

EXTRA:

The below link provides a detailed view of the logical operators - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators

- **Bitwise Operators**

The **bitwise operators** treat operands as sequences of 32 bits binary representation(0 and 1).

The several bitwise operators are -

- **Bitwise AND (&)** - returns 1 for each bit position where both bits are 1s. Eg., (5 & 13 = 5) is evaluated as (0101 & 1101 = 0101).
- **Bitwise OR (|)** - returns 1 for each bit position where either bit is 1. Eg., (5 | 13 = 13) is evaluated as (0101 | 1101 = 1101).
- **Bitwise XOR (^)** - returns 1 for each bit position where either bit is 1 but not both. Eg., (5 ^ 13 = 8) is evaluated as (0101 ^ 1101 = 1000).
- **Bitwise NOT (~)** - returns the inverted bits of operand. This means that 0 becomes 1 and vice versa.
- **Left Shift (<<)** - shifts bits to the left and inserts 0s from right.
- **Sign-propagating Right Shift (>>)** - shifts bits to the right and inserts either 0s or 1s from the left according to the sign of the number ('0' for positive and '1' for negative).
- **Zero-fill Right Shift (>>>)** - shifts bits to the right and inserts 0s from left.

EXTRA:

You can get details of bitwise operator from the below link - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators

NOTE: *You need to be careful while working with operators because they work in precedence. So you might use round brackets to group them according to your priority of execution. Also, the round brackets (called as* ***grouping operator****) has the highest precedence. You can check the link below to see the precedence of the operators -* *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence#Table*

EXTRA:

You can read about all the operators from the link below - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Operators

CONDITIONALS

Conditional statements are used to ***run different sets of statements according to the conditions.*** This means that based upon different conditions, we can perform different actions. These are the same as used in other languages.

Some of the conditional statements have been described below -

- **If statements**

  The **'if'** statement specifies a block of statements (JavaScript code) that gets executed when the **'condition'** specified in the 'if' statement evaluates to true.

  The syntax for the 'if' statement is -

```
if (condition) {
   ...STATEMENTS...
}
```

  First, the condition is evaluated and if it is `true`, then the statements inside it are executed. Else if the condition is `false`, statements inside **'if'** are not executed.

- **if-else Statements**

  The **'if'** statement specifies a block of statements (JavaScript code) that gets executed when the **'condition'** specified in the **'if'** statement evaluates to true. Else, if the ***condition is false*** then the statements inside the ***'else' block are executed.***
  The syntax for the 'if-else' statement is -

```
if (condition) {
   ...STATEMENTS...
} else {
   ...SOME OTHER STATEMENTS...
}
```

- **else-if statements**

  The above two works for only two options i.e. whether the condition is `true` or `false`. Using **else-if** we can define ***more than one conditions*** and form a chain. The first condition that returns `true` gets executed, else the last 'else' block is executed.

```
if (condition - 1) {
   ...STATEMENTS...
} else if (condition - 2) {
   ...SOME OTHER STATEMENTS...
}...
...
...
else if (condition - (n)) {
   ...SOME OTHER STATEMENTS...
} else {
   ...EXECUTE IF NO CONDITION IS SATISFIED...
}
```

  Now the first condition is checked. If it evaluates to ***true***, it gets executed and other ***'else-if'*** blocks are not checked. If the first condition is ***false***, then the second condition is checked.

  Like this, each ***'if'*** condition is checked one by one, until one is ***true*** or ***'else'*** block is reached. If any one of them is ***true***, the block associated with it gets executed. Else the ***'else'*** block is executed.

- **Switch statements**

  The **switch** statements are similar to **else-if** statements, but ***works on a single expression***. This expression evaluates to different values which decide which block of code needs to be executed.

  The syntax for 'switch' statement is **-**

```
switch (expression) {
   case choice1:
       Run
       if expression = choice1
       break;
   case choice2:
       run
       if expression = choice2
       break;
       ...
       ...
       ...
       case choice(n):
           run
       if expression = choice(n)
       break;
   default:
       run
       if no choice is found
}
```

  The expression gives some result which is checked against the choices. If any of them matches, its set of statements get executed. Else the last **'default'** block gets executed.

  We have provided a ***'break***;' statement after each case is completed, to top the execution of the ***'switch'*** block. If **break** is not present, then the case after the one that matches also gets executed until ***'break***;' is found or switch block does not end.

EXTRA:

You can read about these conditional statements from the link below - https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/conditionals

LOOPS

Loops are used to do **something repeatedly.** Let's say that you need to print 'n' numbers, then you can use a loop to do so. There are many different kinds of loops, but they almost do the same thing. Their use varies depending on the type of situation, where one loop will be easy to implement over another.

The various loops that JavaScript provide are described below -

- **for statement**
  A **for** loop is used to ***repeat something until the condition evaluates to false***. The syntax for **'for'** loop is -

```
for ( [initializationStatement] ; [condition] ; [updateStatement] )
{
   ... STATEMENTS ...
}
```

  The **`initializationStatement`** used to initialize loop counters.
  The **`condition`** is the expression that is evaluated to boolean value 'true' or 'false'.
  The **`updateStatement`** is used to update the loop counters.

  Some points to remember about for loop in JS are -
  - The **`initializationStatement`** is optional and you can initialize these statements before the 'for' loop.
  - You can provide more than one **`initializationStatement`** using **comma**(",")as separator.
  - The loop executes until the condition becomes false. If you omit the **`'condition'`**, then it is evaluated to be true.
  - The **`updateStatement`** is also optional and can be present inside the loop. If you do not provide any update, the loop will execute infinitely.

  Eg., for(var i=0; i<=10; i++) { ... }; is loop where we have initialization, condition and update statements.
  Another example where there is no initializer, the loop will look like this - for(; i<=10; i++) { ... };. Here you can see a semicolon(;) at the start and then the condition. You need to provide semicolons for each part of the loop, even if the options are missing. Eg., for(vari=0; ; i++);

- **while statement**

  The **while** statement ***executes the statements until the condition is not false***. The syntax for **'while'** loop is -

```
while (condition) {
   ...STATEMENTS...
}
```

  First, the **`condition`** is evaluated and if it is **true**, then the statements are executed and again the **`condition`** is tested. The execution stops when the **`condition`** returns **false**.

  NOTE*: You have to provide an update expression inside the loop, so that it does not repeat infinitely.*

- **do...while statement**

  The **do-while** loop is similar to while loop, except that ***the statements are executed at least once.*** The syntax for ***'do-while'*** loop is-

```
do {
   ...STATEMENTS...
} while (condition);
```

  First, the statements are executed without checking the **`condition`**. After that the **`condition`** is checked and if it is **true** the statements are executed again.