Data types in JavaScript

There are 8 data types in JavaScript:

1.	String	Primitive
2.	Number	Primitive
3.	Boolean	Primitive
4.	Undefined	Primitive
5.	Null	Primitive
6.	Object	Non-primitive / Reference
7.	BigInt	Primitive
8.	Symbol	Primitive

String

The String is used for storing **text**. In JavaScript, strings are surrounded by quotes:

- Single quotes: `'Hello'`
- Double quotes: `"Hello"`
- Backticks: `` `Hello` ``

Number

The Number represents **integer and floating numbers** (decimals and exponentials):

```
const number1 = 3;
const number2 = 3.433;
const number3 = 3e5 // 3 * 10^5
```

A number type can also be *+Infinity, -Infinity*, and *NaN* (not a number).

Boolean

This data type represents logical entities. Boolean represents one **of two values: true or false**. It is easier to think of it as a yes/no switch. For example:

```
const dataChecked = true;
const valueCounted = false;
```

Undefined

The undefined data type represents **value that is not assigned**. If a variable is declared but the value is not assigned, then the value of that variable will be undefined. For example:

```
let name;
console.log(name); // undefined
```

It is also possible to explicitly assign a variable value undefined. For example:

```
let name = undefined;
console.log(name); // undefined
```

It is recommended not to explicitly assign undefined to a variable. Usually, null is used to assign 'unknown' or 'empty' value to a variable.

Null

In JavaScript, null is a special value that represents **empty or unknown value**. For example:

```
const number = null;
```

The code above suggests that the number variable is empty.
null is not the same as NULL or Null.

Object

An object is a complex data type that allows us to store **collections of data**. For example:

```
const student = {
    firstName: 'ram',
    lastName: null,
    class: 10
};
```

Array, Object, function, RegEx, Date are types of Object. Object represents refrence / non-primitive / composite data type in JavaScript.

BigInt

In JS, Number type can only represent numbers less than (2^{53} - 1) and more than -(2^{53} - 1). However, if you need to use a larger number than that, you can use the BigInt data type.
A BigInt number is created by appending n to the end of an integer. For example:

```
// BigInt value
const value1 = 900719925124740998n;

// Adding two big integers
const result1 = value1 + 1n;
console.log(result1); // "900719925124740999n"

const value2 = 900719925124740998n;

// Error! BitInt and number cannot be added
const result2 = value2 + 1;
console.log(result2);

Output
900719925124740999n
Uncaught TypeError: Cannot mix BigInt and other types
```

BigInt was introduced in the newer version of JavaScript and is **not supported by many browsers including Safari**.

Symbol

This data type was introduced in a newer version of JavaScript (from ES2015).
A value having the data type Symbol can be referred to as a symbol value. Symbol is an immutable primitive value that is unique. For example:

```
// two symbols with the same description
const value1 = Symbol('hello');
const value2 = Symbol('hello');
```

Though `value1` and `value2` both contain `'hello'`, they are different as they are of the Symbol type.

Primitive vs non-primitive/reference data types/values

Data types in JavaScript are classified into 2 types:

- **Primitive**: - String, Boolean, Number, BigInt, Null, Undefined, Symbol
- **Non-Primitive**: - Object (array, functions) also called object references.

The fundamental difference between primitives and non-primitive is that primitives are immutable and non-primitive are mutable.

- **immutable values** are data types that cannot be changed after they are created. When you assign a new value to a variable that holds an immutable data type, a new value is created in memory, and the old value is going to garbage collection. Examples of immutable data types include numbers, booleans, and strings.

- **mutable values** are data types that can be changed after they are created. When you modify a mutable value, you are actually changing the original value in memory rather than creating a new one. Examples of mutable data types include objects and arrays.

Primitives

They are **stored in memory (normally stack**), variable stores the value itself.
Copying a variable (= assign to the different variable) copies the value.
Primitives are known as being immutable data types because there is **no way to change a primitive value once it gets created**.

```
var string = 'This is a string.';
string = 'H'console.log(string)
console.log(string) // Output -> 'This is a string.'
```

A variable that stored primitive value **can only be reassigned to a new value** (instead of changing the original value, JavaScript creates a new value – and creates a new location in memory), as shown in the example below:

```
var string = 'This is a string.';
string = 'Hello world'
console.log(string) // Output -> 'Hello world'
```

Primitives are **compared by value**. Two values are strictly equal if they have the same value.

```
var number1 = 5;
var number2 = 5;
number1 === number 2; // true
var string1 = 'This is a string.';
var string2 = 'This is a string.';
string1 === string2; // true
```

Here the variable for primitives stores the value, so they are always copied or passed by value.

Non-Primitives / Reference

Stored in memory (heap) variable stores a pointer (address) to a location in memory.
Copying a variable (= assign to a different variable) copies the pointer/reference.
Non-primitive values are mutable data types. The **value of an object can be changed** after it gets created, the location in memory is still the same.

```
var arr = [ 'one', 'two', 'three', 'four', 'five' ];
arr[1] = 'TWO';
console.log(arr)
// [ 'one', 'TWO', 'three', 'four', 'five' ];
```

Objects are **not compared by value**. This means that even if two objects have the same properties and values, they are not strictly equal. The same goes for arrays. Even if they have the same elements that are in the same order, they are not strictly equal.

```
var obj1 = { 'cat': 'playful' };
var obj2 = { 'cat': 'playful' };
obj1 === obj2;  // falsevar arr1 = [ 1, 2, 3, 4, 5 ];
var arr2 = [ 1, 2, 3, 4, 5 ];
arr1 === arr2;  // false
```

Non-primitive values can also be referred to as reference types because they are being **compared by reference instead of value**. Two objects are only strictly equal if they refer to the same underlying object.

```
var obj3 = { 'car': 'purple' }
var obj4 = obj3;
obj3 === obj4;  // true
```

Summary

- Primitive values are immutable
- Primitive values compared by value
- Non-primitive values are mutable
- Non-primitive compare by reference not value

Data type conversion

JavaScript is a dynamically typed language. This means you don't have to specify the data type of a variable when you declare it. It also means that data types are automatically converted as-needed during script execution.

Numbers and the '+' operator

In expressions involving numeric and string values with the + operator, JavaScript converts numeric values to strings.

```
x = "The answer is " + 42; // "The answer is 42"
y = 42 + " is the answer"; // "42 is the answer"
z = "37" + 7; // "377"
```

With all other operators, JavaScript does not convert numeric values to strings. For example:

```
"37" - 7; // 30
"37" * 7; // 259
```

Converting strings to numbers

In the case that a value representing a number is in memory as a string, there are methods for conversion.

```
parseInt()
parseFloat()
```

`parseInt` only returns whole numbers, so its use is diminished for decimals.

An alternative method of retrieving a number from a string is with the + (unary plus) operator:

```
"1.1" + "1.1" // '1.11.1'
(+"1.1") + (+"1.1"); // 2.2
// Note: the parentheses are added for clarity, not required.
```

Literals

Literals represent values in JavaScript. **Literals are constant values that can be assigned to the variables** that are called literals or constants. This section describes the following types of literals:

- Array literals
- Boolean literals
- Numeric literals
- Object literals
- RegExp literals
- String literals

Array literals

An array literal is a list of zero or more expressions, each of which represents an array element, enclosed in square brackets ([]). When you create an array using an array literal, it is initialized with the specified values as its elements, and its length is set to the number of arguments specified.

The following example creates the coffees array with three elements and a length of three:

```
const coffees = ["French Roast", "Colombian", "Kona"];
```

Extra commas in array literals

If you put two commas in a row in an array literal, the array leaves an empty slot for the unspecified element. The following example creates the fish array:

```
const fish = ["Lion", , "Angel"];
```

When you log this array, you will see:

```
console.log(fish);
// [ 'Lion', <1 empty item>, 'Angel' ]
```

Note that the second item is "empty", which is not exactly the same as the actual undefined value. When using array-traversing methods like `Array.prototype.map`, empty slots are skipped. However, index-accessing fish[1] still returns undefined.

In the following example, the length of the array is four, and `myList[1]` and `myList[3]` are missing. Only the last comma is ignored.

```
const myList = ["home", , "school", ,];
```

However, when writing your own code, you should explicitly declare the missing elements as undefined, or at least insert a comment to highlight its absence. Doing this increases your code's clarity and maintainability.

```
const myList = ["home", /* empty */, "school", /* empty */, ];
```

Boolean literals

The Boolean type has two literal values: `true` and `false`.

Numeric literals

JavaScript numeric literals include integer literals in different bases as well as floating-point literals in base-10.

Integer literals
Integer and BigInt literals can be written in decimal (base 10), hexadecimal (base 16), octal (base 8) and binary (base 2).

- A *decimal* integer literal is a sequence of digits without a leading 0 (zero).
- A leading 0 (zero) on an integer literal, or a leading 0o (or 0O) indicates it is in *octal*. Octal integer literals can include only the digits 0 – 7.

- A leading 0x (or 0X) indicates a *hexadecimal* integer literal. Hexadecimal integers can include digits (0 – 9) and the letters a – f and A – F. (The case of a character does not change its value. Therefore: 0xa = 0xA = 10 and 0xf = 0xF = 15.)
- A leading 0b (or 0B) indicates a *binary* integer literal. Binary integer literals can only include the digits 0 and 1.
- A trailing n suffix on an integer literal indicates a BigInt literal. The integer literal can use any of the above bases. Note that leading-zero octal syntax like 0123n is not allowed, but 0o123n is fine.

Floating-point literals

A floating-point literal can have the following parts:

- An unsigned decimal integer,
- A decimal point (.),
- A fraction (another decimal number),
- An exponent.

For example:

```
3.1415926
.123456789
3.1E+12
.1e-23
```

Object literals

An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}).

The following is an example of an object literal. The first element of the car object defines a property, myCar, and assigns to it a new string, `"Saturn"`; the second element, the `getCar` property, is immediately assigned the result of invoking the `function (carTypes("Honda"));` the third element, the special property, uses an existing variable (`sales`).

```
const sales = "Toyota";

function carTypes(name) {
  return name === "Honda" ? name : `Sorry, we don't sell ${name}.`;
```

```
}

const car = { myCar: "Saturn", getCar: carTypes("Honda"), special: sales };

console.log(car.myCar); // Saturn
console.log(car.getCar); // Honda
console.log(car.special); // Toyota
```

Object property names can be any string, including the empty string. If the property name would not be a valid JavaScript identifier or number, it must be enclosed in quotes.

Property names that are not valid identifiers cannot be accessed as a dot (.) property.

```
const unusualPropertyNames = {
  '': 'An empty string',
  '!': 'Bang!'
}
console.log(unusualPropertyNames.'');   // SyntaxError: Unexpected string
console.log(unusualPropertyNames.!);    // SyntaxError: Unexpected token !

Instead, they must be accessed with the bracket notation ([]).

console.log(unusualPropertyNames[""]); // An empty string
console.log(unusualPropertyNames["!"]); // Bang!
```

Enhanced Object literals

Object literals support a range of shorthand syntaxes that include setting the prototype at construction, shorthand for foo: foo assignments, defining methods, making super calls, and computing property names with expressions.

```
const obj = {
  // __proto__
  __proto__: theProtoObj,
  // Shorthand for 'handler: handler'
```

```
    handler,
    // Methods
    toString() {
      // Super calls
      return "d " + super.toString();
    },
    // Computed (dynamic) property names
    ["prop_" + (() => 42)()]: 42,
  };
```

RegExp literals

A regex literal is a pattern enclosed between slashes. The following is an example of a regex literal.

```
const re = /ab+c/;
```

String literals

A string literal is zero or more characters enclosed in double (") or single (') quotation marks. A string must be delimited by quotation marks of the same type (that is, either both single quotation marks or both double quotation marks).

```
'foo'
"bar"
'1234'
'one line \n another line'
"Joyo's cat"
```

Template literals

Template literals are also available. Template literals are enclosed by the back-tick (`) (grave accent) character instead of double or single quotes.

Template literals provide syntactic sugar for constructing strings.

```
/ Basic literal string creation
`In JavaScript '\n' is a line-feed.`

// Multiline strings
`In JavaScript, template strings can run
 over multiple lines, but double and single
 quoted strings cannot.`

// String interpolation
const name = 'Lev', time = 'today';
`Hello ${name}, how are you ${time}?`
```

Tagged templates

A more advanced form of template literals are tagged templates.
Tags allow you to parse template literals with a function. The first argument of a tag function contains an array of string values. The remaining arguments are related to the expressions.
The tag function can then perform whatever operations on these arguments you wish, and return the manipulated string.

```
  const person = "Mike";
  const age = 28;

  function myTag(strings, personExp, ageExp) {
    const str0 = strings[0]; // "That "
    const str1 = strings[1]; // " is a "
    const str2 = strings[2]; // "."

    const ageStr = ageExp > 99 ? "centenarian" : "youngster";

    // We can even return a string built using a template literal
    return `${str0}${personExp}${str1}${ageStr}${str2}`;
  }

  const output = myTag`That ${person} is a ${age}.`;

  console.log(output);
  // That Mike is a youngster.
```