Agenda

1.What is JavaScript?

2.How JS Powers the Internet

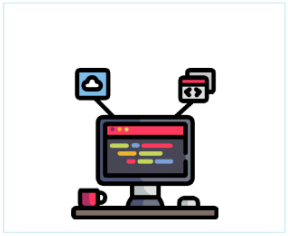
3.Variables

4.Data Types

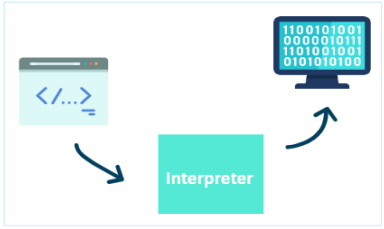
5.Type Conversion, nAn, Infinity

What is JavaScript ?

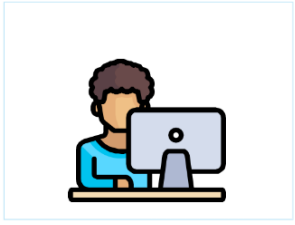
- JavaScript is a high level, Interpreted programming language used to make web pages more interactive.
- Every time a web page does more than just sit there and display static information for you to look at — Timely Updates, Alerts, Action on Button Click etc. is using JavaScript.



Interactive Web Pages



Interpreted Language



Runs on the Client's System

Why Dynamically Typed Language ?

Dynamically-typed languages are those where the interpreter assigns variables a type at runtime based on the variable's value at the time.

You do not have to explicitly declare the type of variable which is being declared.

Need for JavaScript

Drawbacks/ Restrictions of the webpage created by using HTML & CSS.

- It is just a static unresponsive HTML with images, texts, buttons & other UI features.
- No use of such web page from the perspective of a business or the customers.

This is where **JavaScript** comes into the picture, It provides the following advantages:

- It is very fast because any code can run immediately instead of having to contact the server.

- It allows you to create highly responsive interfaces to improve the user experience.
- **JavaScript** has no compilation step. Instead, an interpreter in the browser reads over the JavaScript code, interprets each line and runs it.
- It provides dynamic functionality without even having to wait for the server to react and show another page.

What makes them so well-loved by programmers

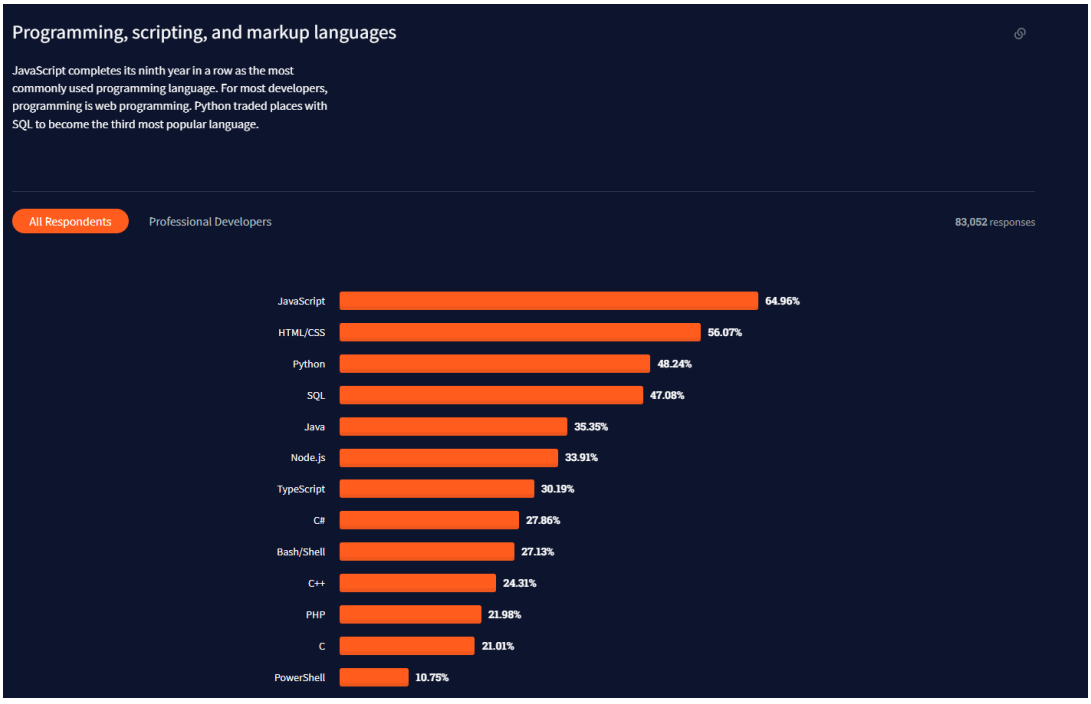
- There are 1.9 billion websites on the Internet today. And 95% of them use JavaScript in some way or the other.
- Try turning off the JavaScript access on your web browser and you'd see several sites and web apps start to crack and look dull and drab. That's because it is JavaScript that manages the interactivity of the web.
- Plus, developers love it because it is simple to learn and master. You could start with 0 coding experience and still be able to create something beautiful with just the knowledge of JavaScript. For businesses, it means never having to struggle to find able JavaScript developers.

Other than simplicity, JavaScript is also known for its speed and versatility. It can be used for the following programming projects:

- Adding interactive elements to your website
- Create web and mobile apps
- Build web servers
- Game development.

Why JavaScript is Popular ?

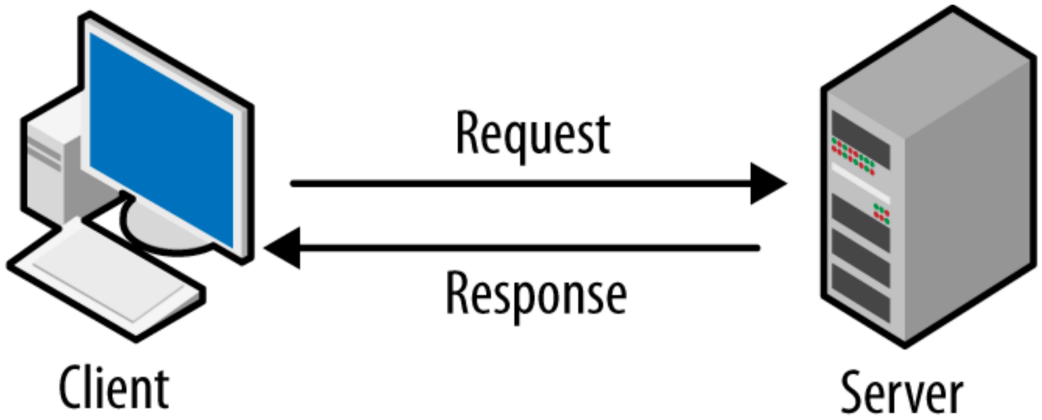
JavaScript completes its ninth year in a row as the most commonly used programming language. For most developers, programming is we programming. Python traded places with SQL to become the third most popular language.



Reference : https://insights.stackoverflow.com/survey/2021#programming-scripting-and-markup-languages

How JavaScript Powers the Internet?

Client - Server Cycle :

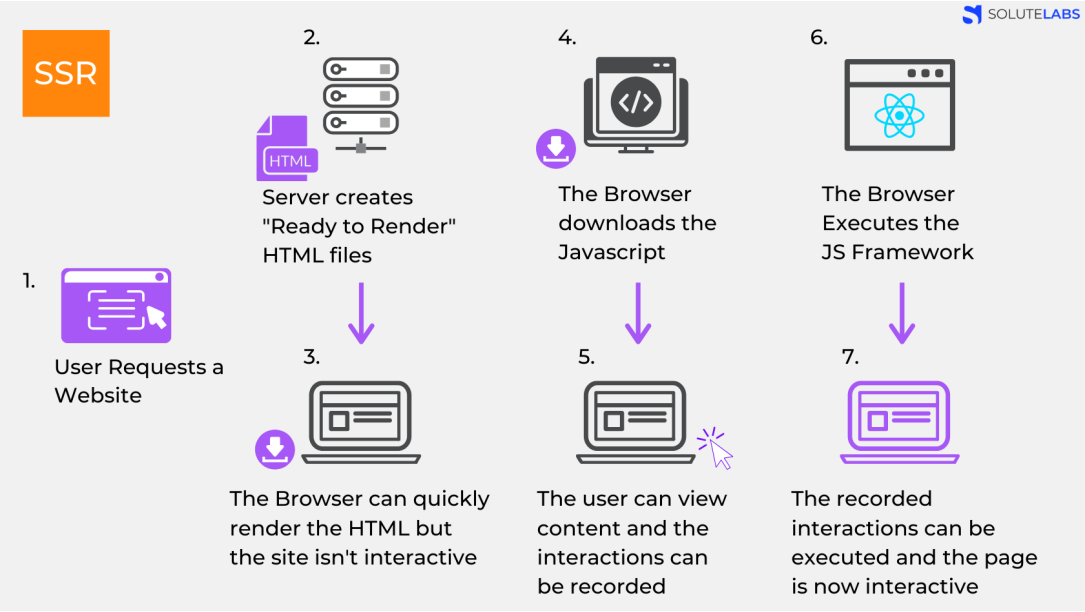


- With the power of JavaScript frameworks, it became possible to render dynamic content right from the browser by requesting just the content that was required. The server, in this scenario, only served the base HTML wrapper that was necessary.
- This transformation gave a seamless user experience to visitors since there was very little time taken for loading the web page. Moreover, once loaded, the web page did not reload itself again.

Server-side Rendering :

As discussed above, the traditional way of rendering dynamic web content follows the below steps:

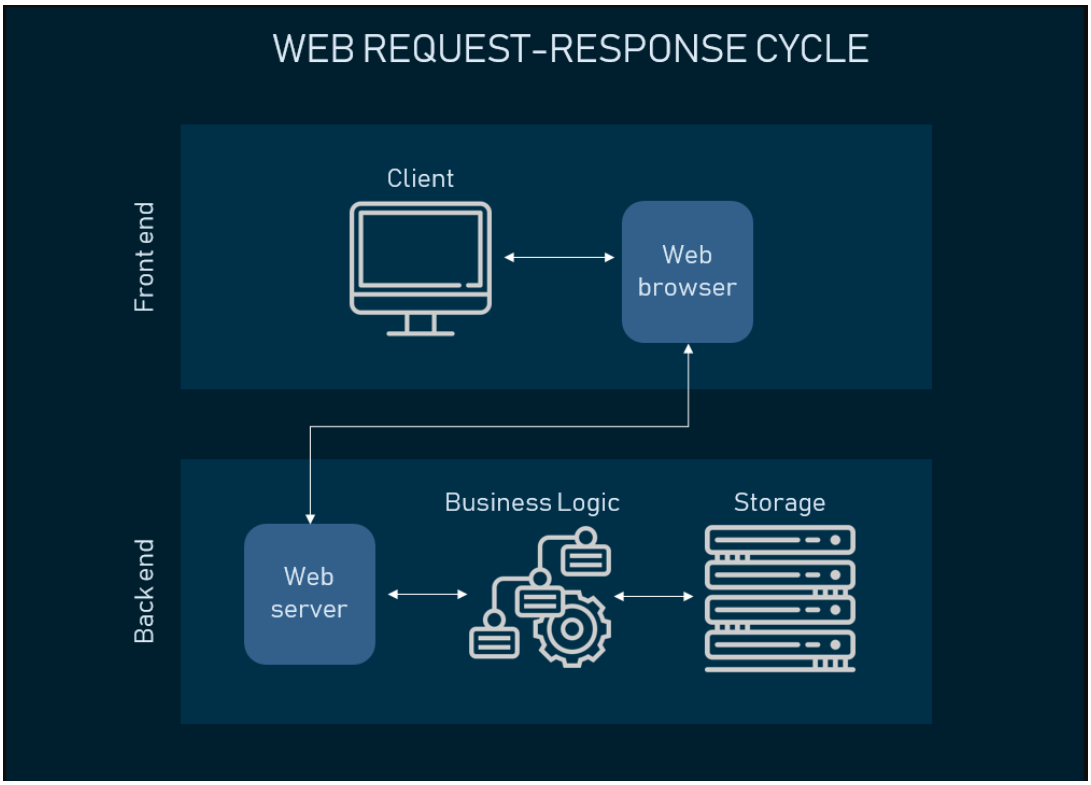
- The user sends a request to a website (usually via a browser)
- The server checks the resource, compiles and prepares the HTML content after traversing through server-side scripts lying within the page.
- This compiled HTML is sent to the client's browser for further rendering and display.
- The browser downloads the HTML and makes the site visible to the end-user
- The browser then downloads the Javascript (JS) and as it executes the JS, it makes the page interactive



Client-side Rendering :

he normal flow of web page rendering for a client-side rendering scenario follows these steps:

- The user sends a request to a website (usually via a browser).
- Instead of a server, a CDN (Content Delivery Network) can be used to serve static HTML, CSS, and supporting files to the user.
- The browser downloads the HTML and then the JS. Meanwhile, the user sees a loading symbol.
- After the browser fetches the JS, it makes API requests via AJAX to fetch the dynamic content and processes it to render the final content.
- After the server responds, the final content is rendered using DOM processing in the client's browser.



Usability of JavaScript in Front-end :

- A front-end web developer's role is to create the code and mark-up that is rendered by a web browser when you visit a site (read: they control what you see when you visit a webpage).
- There are three main components when it comes to front-end development: HTML, CSS and JavaScript. Each are critical for making a webpage what it is. HTML is the structure and content of the site, CSS (Cascading Style Sheets) makes it look pretty, and, lastly, JavaScript is what powers its interactivity. Each work hand-in-hand when it comes to building websites, but the focus of this blog post is on JavaScript and how it's used.

INTERACTIVITY

JavaScript is a very powerful tool that can do many things for a website. For one, it powers the site's general interactivity. JavaScript makes it possible build rich UI components such as image sliders, pop-ups, site navigation mega menus, form validations, tabs, accordions, and much more.

CROSS-BROWSER COMPATIBILITY AND STANDARDS COMPLIANCE

There are quite a few web browsers out there: Firefox, Chrome, Safari, Opera, Internet Explorer (10, 11, Edge)… and they all run on different operatir systems and devices. From time to time, each of these browsers has its own unique bugs and quirks. Nothing is perfect, unfortunately. While W3 compliance standards continue to improve across the board, there still comes a time when a front-end web developer needs to resolve issues b leveraging JavaScript.

PLUGINS

There are various plugins and that run off of JavaScript. If you've ever visited a site that features banner ads, has chat support, suggests content, ha forms, or offers social sharing, there's a good chance it's powered by a 3rd party JavaScript plugin. Usually, these plugins have configurable options th need additional set-up to function properly. Understanding the configurable options of these plugins is essential. Plugins are generally intended to b easily dropped into a site with little modification.

FRAMEWORKS

- A JavaScript framework can be a powerful tool you can use to help render the page. These are typically only used when there are complex dynamic interactions that need to occur.
- One example of this is if you have a multi-step form-fill. In this case, the form fill process has certain steps that only occur based on previously entered information. Also, certain data gets populated for certain inputs as well as previous inputs. Doing this without a framework can be very difficult task to achieve. Things can get problematic, and this can happen fast.
- Using a JavaScript framework helps resolve these issues so you can complete your wonderful form, and make your clients happy. While there are dozens, the most popular ones (as of this writing) are Google's Angular, Facebook's React and the open source Vue.js.

- JavaScript is a very important tool for a front-end web developer. Without it, webpages wouldn't have become the dynamic web applications they are today. There would be no image carousels. There would be no partial page reloads that keep your spot on the page.

Usability of JavaScript in Back-end :

The popularity of Node.js has definitely boosted the use of JavaScript as a backend language, and in order to get started with JavaScript in the backen you need to know some basics and general rules of this language.

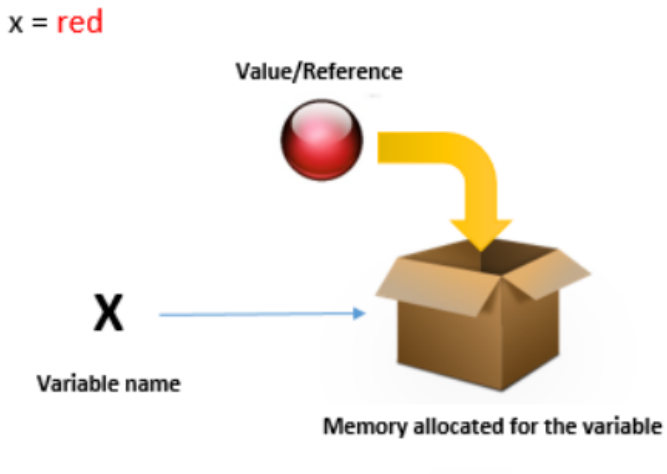
JavaScript Engine :

Each browser has its own JavaScript engine which is used to support the JavaScript scripts in order for them to work properly. The basic job of JavaScript engine is to take the JavaScript code, then convert it into a fast, optimized code that can be interpreted by a browser. Below are the names the JavaScript engines used in some of the most popular browsers out there.

- Chrome: V8
- Firefox: SpiderMonkey
- Safari: JavaScript Core
- Microsoft Edge/ Internet Explorer: Chakra/ChakraCore.

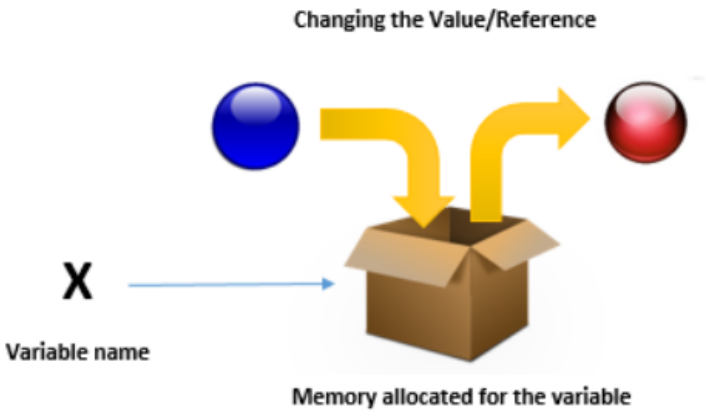
What is a Variable :

Variable is a name given to a memory location that acts as a container for storing data temporarily. They are nothing but reserved memory locations store values.



Now setting the new value;

x = blue

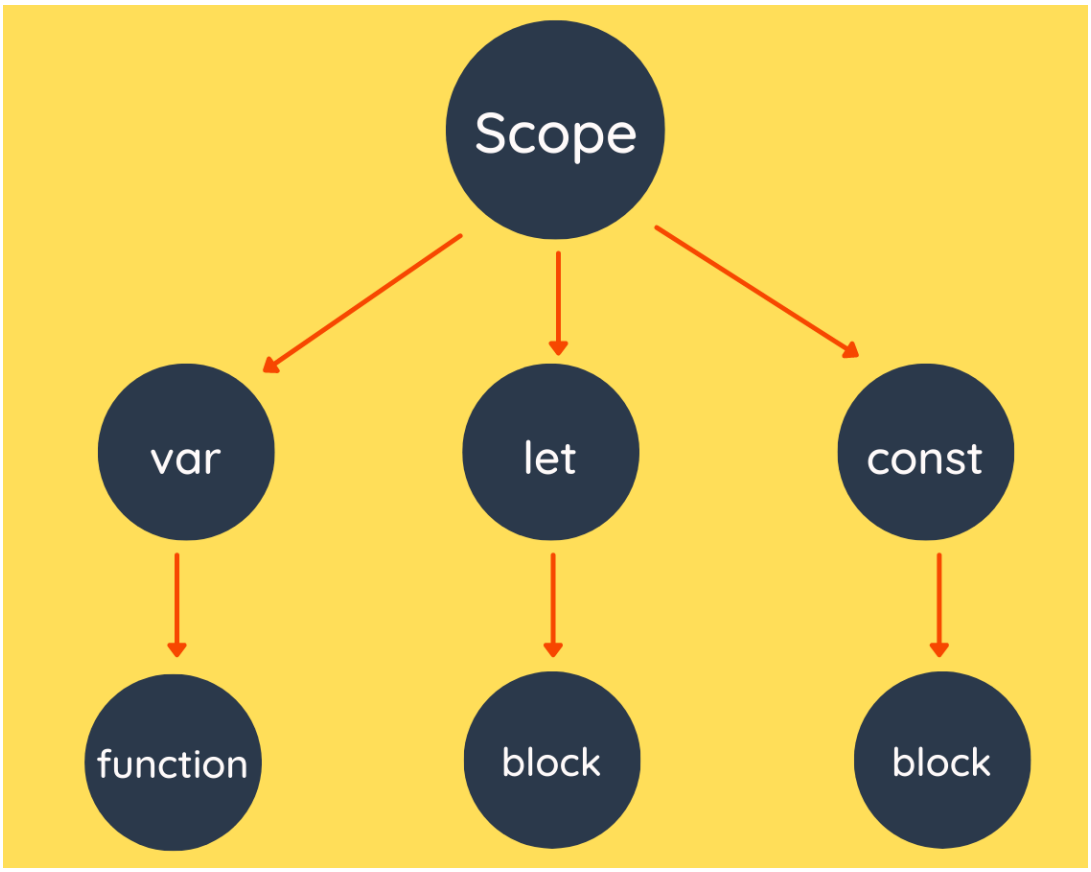


JavaScript Variable

How to declare variable in JavaScript ?

There are three ways of declaring variables in JS

- var (declares mutable variables Before ES6)
- let (declares mutable variables)
- const (declares immutable variables)



The *var* Keyword in JavaScript

In JavaScript, `var` is a reserved keyword which is followed by a reference variable name. The name defined after the keyword can then be used as pointer to the data in-memory.

Using `var` is the oldest method of **variable declaration** in JavaScript. Let's declare a variable and **initialize it** by assigning a value to it using th assignment operator (`=`):

```
// Declaration and initialization

var a = "John Doe" ;
```

Alternatively,

```
// Declaration

var a;

// Initialization

a = "John Doe";
```

Issues with var :

Var is not block-scoped.

When you declare a variable within a code block, using curly braces (`{}`), its scope "flows out" of the block! For instance:

```
var a = "John Doe";

var someBool = true;

if (someBool) {

var a = "Daniel Joan";

}


console.log(a);
```

The `name` that points to "John Doe" is global, and the `name` that points to "Daniel Joan" is defined within a block. However, when we try printir the `name` that's within scope, we run into :

```
Daniel Joan
```

> var is not block-scoped. We may think that we've defined a local var name to point to "Daniel Joan", but what we've done in reality is overwrite the var name that points to "John Doe".

The *let* Keyword in JavaScript

The `let` declaration was introduced with ES6 and has since become the preferred method for variable declaration. It is regarded as an improveme over `var` declarations and *is* block-scoped (variables that can be accessed only in the immediate block), circumventing the main issue that can aris with using `var` .

Scope of *let*

A variable defined with the `let` keyword has a scope limited to the block or function in which it is defined:

```
let firsta = "John";

let lasta = "Doe";

let someBool = true;

if(someBool){

let firsta = "Jane";

console.log(firsta);

}

console.log(firsta);
```

This time around - the `firsta` referring to "Jane" and the `firsta` referring to "John" don't overlap! The code results in:

```
Jane
```

```
John
```

The `firsta` declared within the block is limited to the block in scope and the one declared outside the block is available globally. Both instance of `firsta` are treated as different variable references, since they have different scopes.

The *const* Keyword in JavaScript

The `const` declaration was introduced with ES6, alongside `let`, and it is very similar to `let`. `const` points to data in memory that holds consta values, as the name implies. `const` reference variables cannot be reassigned to a different object in memory:

```
const a = "John";
const a = "Jane";
```

This results in:

```
Uncaught SyntaxError: Identifier 'name' has already been declared
```

Scope of *const*

The scope of a variable defined with the `const` keyword, like the scope of `let` declarations, is limited to the block defined by curly braces (a functic or a block). The main distinction is that they cannot be updated or re-declared, implying that the value remains constant within the scope:

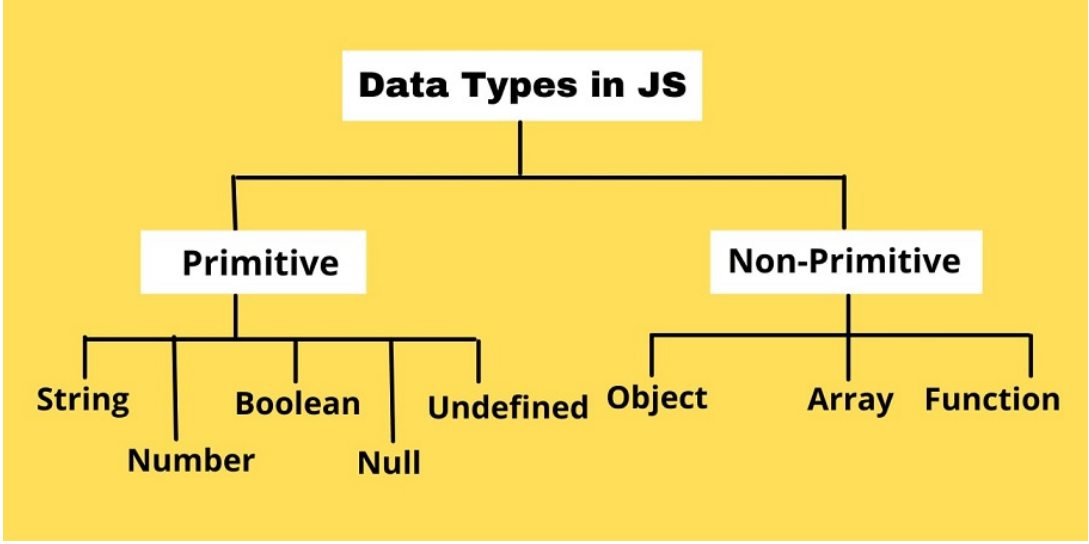
```
const a = "John";
a = "Doe";

*// Uncaught TypeError: Assignment to constant variable.*
```

Good Coding Conventions

So, what does this all mean, and which should you choose, other than the obvious requirements to avoid bugs? This can actually be boiled down to couple of good practices:

- `const` is preferred to `let`, which is preferred to `var`. Avoid using `var`.
- `let` is preferred to `const` when it's known that the value it points to will change over time.
- `const` is great for global, constant values.
- Libraries are typically imported as `const`.

Data Types :

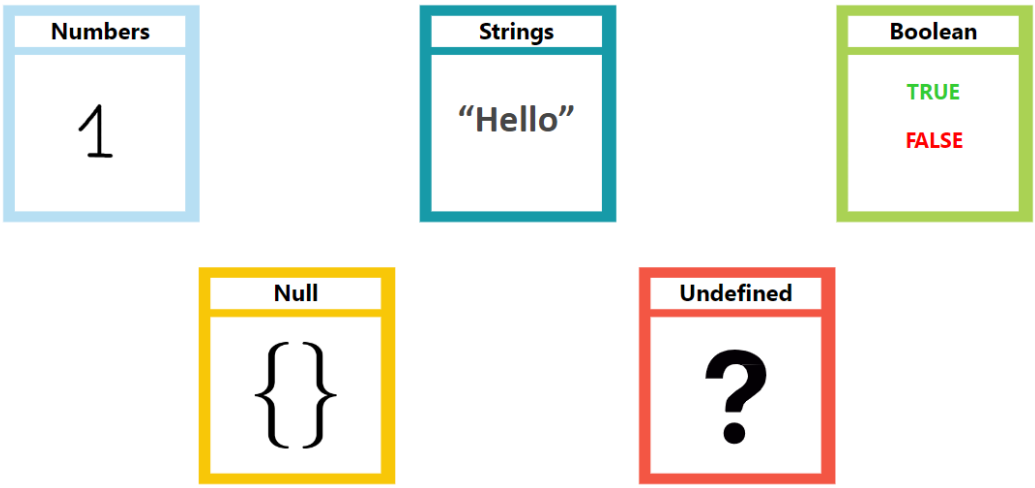


The Concept of Data Types

In programming, data types is an important concept.

To be able to operate on variables, it is important to know something about the type.

There are five types of primitive data types in JavaScript :



There are three different types of non-primitive data types in JavaScript :

- Reference data types, unlike primitive data types, are dynamic in nature. That is, they do not have a fixed size.
- Most of them are considered as objects, and therefore have methods.
- A reference type can contain other values. Since the contents of a reference type can not fit in the fixed amount of memory available for a variable, the in-memory value of a reference type is the reference itself (a memory address).
- Array
- Object
- Function

Reference types are also known as: complex types or container types.

1. Array : Homogeneous Data Structure

An Array is a data structure that contains a list of elements of the same data type.

```
var names = [ "Mayank", "Shubham", "Amrita"] ; // Array of strings
```

```
var ages = [ 30, 29, 33 ] ; // Array of numbers
```

2. Objects : Key Value Pairs

An Object is a standalone entity. It consists of properties & methods.

```
var student = {
roll no : 34,
name : "Mayank" ,
age : 27 ,
city : Delhi
};
```

3. Functions :

A function is a block of organized, reusable code that is used to perform a single, related actions.

```
function sum (a, b) {
       return a+b
}

sum (3,4) ; // 7
sum (5,6) ; //11
```

JavaScript Type Conversions :

In programming, type conversion is the process of converting data of one type to another. For example: converting `String` data to `Number` .

There are two types of type conversion in JavaScript.

- **Implicit Conversion** - automatic type conversion
- **Explicit Conversion** - manual type conversion

JavaScript Implicit Conversion

In certain situations, JavaScript automatically converts one data type to another (to the right type). This is known as implicit conversion.

```
// numeric string used with + gives string type
let result;

result = '3' + 2;
console.log(result) // "32"

result = '3' + true;
console.log(result); // "3true"

result = '3' + undefined;
console.log(result); // "3undefined"

result = '3' + null;
console.log(result); // "3null"
```

> **Note**: When a number is added to a string, JavaScript converts the number to a string before concatenation.

Example 2 : Implicit Conversion to Number

```
// numeric string used with - , / , * results number type
let result;
result = '4' - '2';
console.log(result); // 2
result = '4' - 2;
console.log(result); // 2
result = '4' * 2;
console.log(result); // 8
result = '4' / 2;
console.log(result); // 2
```

Example 3: Non-numeric String Results to NaN

```
// non-numeric string used with - , / , * results to NaN
let result;
result = 'hello' - 'world';
console.log(result); // NaN
result = '4' - 'hello';
console.log(result); // NaN
```

Example 4: Implicit Boolean Conversion to Number

```
// if boolean is used, true is 1, false is 0
let result;
result = '4' - true;
console.log(result); // 3
result = 4 + true;
console.log(result); // 5
result = 4 + false;
console.log(result); // 4
```

> **Note:** JavaScript considers 0 as `false` and all non-zero number as `true` . And, if `true` is converted to a number, the result is always 1.

Example 5: null Conversion to Number

```
// null is 0 when used with number

let result;

result = 4 + null;

console.log(result);  // 4

result = 4 - null;

console.log(result);  // 4
```

Example 6: undefined used with number, Boolean or null

```
**// Arithmetic operation of undefined with number, boolean or null gives NaN

let result;

result = 4 + undefined;

console.log(result);  // NaN

result = 4 - undefined;

console.log(result);  // NaN

result = true + undefined;

console.log(result);  // NaN

result = null + undefined;

console.log(result);  // NaN**
```

JavaScript Explicit Conversion

You can also convert one data type to another as per your needs. The type conversion that you do manually is known as explicit type conversion.

In JavaScript, explicit type conversions are done using built-in methods.

Here are some common methods of explicit conversions.

1. Convert to Number Explicitly

To convert numeric strings and boolean values to numbers, you can use `Number()` . For example,

```
let result;

// string to number

result = Number('324');
```

```
console.log(result); // 324

result = Number('324e-1')

console.log(result); // 32.4

// boolean to number

result = Number(true);

console.log(result); // 1

result = Number(false);

console.log(result); // 0
```

In JavaScript, empty strings and `null` values return **0**. For example,

```
let result;

result = Number(null);

console.log(result);  // 0

let result = Number(' ')

console.log(result);  // 0
```

If a string is an invalid number, the result will be `NaN` . For example,

```
let result;

result = Number('hello');

console.log(result); // NaN

result = Number(undefined);

console.log(result); // NaN

result = Number(NaN);

console.log(result); // NaN
```

> **Note**: You can also generate numbers from strings using `parseInt()` , `parseFloat()` , unary operator `+` and `Math.floor()` . For example,

```
let result;

result = parseInt('20.01');

console.log(result); // 20

result = parseFloat('20.01');
```

```
console.log(result); // 20.01

result = +'20.01';

console.log(result); // 20.01

result = Math.floor('20.01');

console.log(result); // 20
```

---

2. Convert to String Explicitly

To convert other data types to strings, you can use either `String()` or `toString()` . For example,

```
//number to string

let result;

result = String(324);

console.log(result);  // "324"

result = String(2 + 4);

console.log(result); // "6"

//other data types to string

result = String(null);

console.log(result); // "null"

result = String(undefined);

console.log(result); // "undefined"

result = String(NaN);

console.log(result); // "NaN"

result = String(true);

console.log(result); // "true"

result = String(false);

console.log(result); // "false"

// using toString()

result = (324).toString();

console.log(result); // "324"

result = true.toString();

console.log(result); // "true"
```

> **Note**: `String()` takes `null` and `undefined` and converts them to string. However, `toString()` gives error when `null` are

> passed.

3. Convert to Boolean Explicitly

To convert other data types to a Boolean, you can use `Boolean()`.

In JavaScript, `undefined`, `null`, `0`, `NaN`, `''` converts to `false`. For example,

```
let result;

result = Boolean('');

console.log(result); // false

result = Boolean(0);

console.log(result); // false

result = Boolean(undefined);

console.log(result); // false

result = Boolean(null);

console.log(result); // false

result = Boolean(NaN);

console.log(result); // false
```

All other values give `true`. For example,

```
result = Boolean(324);

console.log(result); // true

result = Boolean('hello');

console.log(result); // true

result = Boolean(' ');

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

JavaScript Type Conversion Table

The table shows the conversion of different values to String, Number, and Boolean in JavaScript.

Value	String Conversion	Number Conversion	Boolean Conversion
1	"1"	1	`true`
0	"0"	0	`false`
"1"	"1"	1	`true`
"0"	`"0"`	0	`true`
"ten"	`"ten"`	`NaN`	`true`
`true`	"true"	1	`true`
`false`	"false"	0	`false`
`null`	"null"	0	`false`
`undefined`	"undefined"	`NaN`	`false`
''	""	0	`false`
' '	" "	0	`true`

Interview Questions

> Describe the Difference Between Var vs Let vs Const

Var

1. Function Scoped
2. Allows duplicate identifiers
3. Value can be updated
4. Hoisted and initialized with undefined.

Let

1. Block Scoped
2. Does NOT allow duplicate identifiers
3. Value can be updated
4. Hoisted BUT error if we try to access before declaration

Const

1. Block Scoped
2. Does NOT allow duplicate identifiers
3. Value cannot be updated
4. Hoisted BUT error if we try to access before declaration.

> Explain Implicit Type Coercion in javaScript.

Implicit type coercion in JavaScript is the automatic conversion of value from one data type to another. It takes place when the operands of an expressic are of different data types.

- **String coercion** String coercion takes place while using the ‘ + ‘ operator. When a number is added to a string, the number type is always converted to the string type.

```
var x = 3;
var y = "3";
x + y // Returns "33"
```

When JavaScript sees that the operands of the expression x + y are of different types (one being a number type and the other being a string type), it converts the number type to the string type and then performs the operation. Since after conversion, both the variables are of string type, the ‘ + ‘ operator outputs the concatenated string “33”

- **Boolean Coercion** Boolean coercion takes place when using logical operators, ternary operators, if statements, and loop checks. To understand boolean coercion in if statements and operators, we need to understand truthy and falsy values. Truthy values are those which will be converted (coerced) to **true**. Falsy values are those which will be converted to **false**. All values except **false, 0, 0n, -0, “”, null, undefined, and NaN** are truthy values.

```
var x = 0;
var y = 23;

if(x) { console.log(x) }   // The code inside this block will not run since the value of x is 0(Falsy)

if(y) { console.log(y) }    // The code inside this block will run since the value of y is 23 (Truthy)
```

- **Logical operators:** Logical operators in javascript, unlike operators in other programming languages, **do not return true or false. They always return one of the operands.OR (| |) operator** - If the first value is truthy, then the first value is returned. Otherwise, always the second value gets returned.**AND (&&) operator** - If both the values are truthy, always the second value is returned. If the first value is falsy then the first value is returned or if the second value is falsy then the second value is returned.

```
var x = 220;
var y = "Hello";
var z = undefined;

x | | y    // Returns 220 since the first value is truthy

x | | z   // Returns 220 since the first value is truthy

x && y    // Returns "Hello" since both the values are truthy

y && z   // Returns undefined since the second value is falsy

if( x && y ){
  console.log("Code runs" ); // This block runs because x && y returns "Hello" (Truthy)
}

if( x || z ){
  console.log("Code runs");  // This block runs because x || y returns 220(Truthy)
}
```

- **Equality Coercion** Equality coercion takes place when using ‘ == ‘ operator. As we have stated before**The ‘ == ‘operator compares values and not types.**While the above statement is a simple way to explain == operator, it’s not completely trueThe reality is that while using the ‘==’ operator, coercion takes place.The ‘==’ operator, converts both the operands to the same type and then compares them.

```
var a = 12;
var b = "12";
a == b // Returns true because both 'a' and 'b' are converted to the same type and then compared. Hence the operands are
```

```
var a = 226;
var b = "226";

a === b // Returns false because coercion does not take place and the  operands are of different types. Hence they are n
```

Thank You