Data Types

Type	typeof return value	Object wrapper
Null	`"object"`	N/A
Undefined	`"undefined"`	N/A
Boolean	`"boolean"`	`Boolean`
Number	`"number"`	`Number`
BigInt	`"bigint"`	`BigInt`
String	`"string"`	`String`
Symbol	`"symbol"`	`Symbol`

Important Points:

- MDN Web Doc Link: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Data_structures
- **Symbol**: new datatype introduced in ES6, A Symbol is a unique and immutable primitive value and may be used as the key of an Object property. In some programming languages, Symbols are called "atoms". The purpose of symbols is to create unique property keys that are guaranteed not to clash with keys from other code.

```
<script>
    let email = Symbol()
    let data = {
        name: "arjun",
        age: 37,
        [email]: "arjun.bala@darshan.ac.in"
    }

    console.log("Keys = ",Object.keys(data))

    console.log("Email = ", data[email])
</script>
```

- "**typeof**" method/operator will return the current type of variable
- **Exponential Notation:** we can also use exponential notations while specifying the value of a number.

```
<script>
    let a = 123e5;
    let b = 123e-5;

    console.log("a = ",a); // will return 12300000
    console.log("b = ",b); // will return 0.00123
</script>
```

- **"undefined"** : Conceptually, undefined indicates the absence of a value, while null indicates the absence of an object (which could also make up an excuse for typeof null === "object"). The language usually defaults to undefined when something is devoid of a value:
    - A return statement with no value (return;) implicitly returns undefined.

- Accessing a nonexistent object property (obj.iDontExist) returns undefined.
- A variable declaration without initialization (let x;) implicitly initializes the variable to undefined.
- Many methods, such as Array.prototype.find() and Map.prototype.get(), return undefined when no element is found.

Hoisting.

JavaScript Hoisting refers to the process whereby the interpreter appears to move the declaration of functions, variables, classes, or imports to the top of their scope, prior to execution of the code.

```
<script>
    a = 10;
    printMe();

    function printMe(){
        console.log(a);
    }
    var a;

</script>
```

Important Points:

- MDN
- Any of the following behaviours may be regarded as hoisting:
  1. Being able to use a variable's value in its scope before the line it is declared. ("Value hoisting")
  2. Being able to reference a variable in its scope before the line it is declared, without throwing a ReferenceError, but the value is always undefined. ("Declaration hoisting")
  3. The declaration of the variable causes behaviour changes in its scope before the line in which it is declared.
  4. The side effects of a declaration are produced before evaluating the rest of the code that contains it.
- Variable initializations are not hoisted, only variable declarations are hoisted
- To avoid hoisting, you can run javascript in strict mode by using “use strict” on top of the code

Difference between “ == “ and “ === “ operators

Both are comparison operators. The difference between both the operators is that “==” is used to compare values whereas, “ === “ is used to compare both values and types.

```
<script>
    a = 10
    b = "10"
    console.log(a==b) // will return true
    console.log(a===b) // will return false
</script>
```

Difference between var and let keyword

- From the very beginning, the 'var' keyword was used in JavaScript programming whereas the keyword 'let' was just added with ES6 in 2015.
- The keyword 'Var' has a function scope. Anywhere in the function, the variable specified using var is accessible but in ‘let’ the scope of a variable declared with the 'let' keyword is limited to the block in which it is declared. Let's start with a Block Scope.
- Variable declared with var will be hoisted, variable declared with let is also hoisted but inaccessible due to temporal dead zone.

Private variable in JavaScript

Private properties get created by using a hash # prefix and cannot be legally referenced outside of the class.

```
<script>
    class Darshan{
        name = "arjun";
        age = 37;
        #email="arjun.bala@darshan.ac.in";

        #getMarks(){
            // some code
        }
    }

    const obj = new Darshan();
    console.log(obj.name); // this will print arjun
    console.log(obj.#email); // will give error
    obj.#getMarks(); // will give error
</script>
```

Type Coercion

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 expression are of different data types.

Type conversion is similar to type coercion because they both convert values from one data type to another with one key difference — type coercion is implicit whereas type conversion can be either implicit or explicit.

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.

```
<script>
    let a = 10;
    let b = "20";

    let c = a + b;
    console.log(c); // this will return "1020"
    console.log(typeof c); // it will return string
</script>
```

Type coercion also takes place when using the ‘ - ‘ operator, but the difference while using ‘ - ‘ operator is that, a string is converted to a number and then subtraction takes place

```
<script>
    let a = 10;
    let b = "20";

    let c = a - b;
    console.log(c); // this will return -10
    console.log(typeof c); // it will return number
</script>
```

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.

Note: 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.

```
<script>
    let a = "darshan"
    let b = "college";
    let c = 0;

    console.log(a || b); // will return "darshan" (first value)
    console.log(a && b); // will return "college" (second value)
    console.log(c || b); // will return "college" (second value)
    console.log(c && b); // will return 0 (second value)
</script>
```

Equality Coercion

The '==' operator, converts both the operands to the same type and then compares them.

NaN property

NaN property represents the "Not-a-Number" value. It indicates a value that is not a legal number.

typeof of NaN will return a Number.

To check if a value is NaN, we use the isNaN() function,

```
<script>
    console.log(isNaN("Hello"));  // Returns true
    console.log(isNaN(345));   // Returns false
    console.log(isNaN('1'));  // Returns false, since '1' is
converted to Number
    console.log(isNaN(true)); // Returns false, since true
converted to Number type results in 1 ( a number)
    console.log(isNaN(false)); // Returns false
    console.log(isNaN(undefined)); // Returns true
</script>
```

Pass by Value and Pass by Reference

In JavaScript, primitive data types are passed by value and non-primitive data types are passed by reference.

```
<script>
    let a = 10;
    let data = {name:"arjun",age:37};

    function demoPass(a,data){
        a = 15;
        data.name = "darshan";
    }

    demoPass(a,data);

    console.log(a); // a will remain 10 only
    console.log(data.name); // name will become darshan
</script>
```

Strict Mode

In ECMAScript 5, a new feature called JavaScript Strict Mode allows you to write a code or a function in a "strict" operational environment. In most cases, this language is 'not particularly severe' when it comes to throwing errors. In 'Strict mode,' however, all forms of errors, including silent errors, will be thrown. As a result, debugging becomes a lot simpler. Thus programmer's chances of making an error are lowered.

Characteristics of strict mode in javascript:

- Duplicate arguments are not allowed by developers.
- In strict mode, you won't be able to use the JavaScript keyword as a parameter or function name.
- The 'use strict' keyword is used to define strict mode at the start of the script. Strict mode is supported by all browsers.
- You will not be allowed to create global variables in 'Strict Mode.

```
<script>
    "use strict"
    a = 10; // will throw error, Uncaught ReferenceError: a is
not defined
</script>
```

We can make a function to follow strict mode as well by just writing “use strict” at the top of the function

```
<script>
    function demoStrict(){
        "use strict"
        a = 10; // will throw error, Uncaught ReferenceError: a
is not defined
    }
</script>
```

Higher Order Functions

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.

Higher-order functions are a result of functions being first-class function in javascript.

A programming language is said to have First-class functions when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable.

```
<script>
    function getNameFunction(){
        return (
            ()=>{
                console.log("this is name");
            }
        );
    }

    const fn = getNameFunction();
    fn();
</script>
```

```
<script>
    function print(callback){
        console.log("Print Method Called");
        callback();
    }

    function hello(){
        console.log("Hello Method Called");
    }

    print(hello);
</script>
```

Default Parameters

Default function parameters allow named parameters to be initialized with default values if no value or undefined is passed.

```
<script>
    function setData(name, age=18){
        console.log(name);
        console.log(age);
    }

    setData("arjun",37); // will print arjun and 37
    setData("darshan"); // will print darshan and 18
</script>
```

Parameters are set left-to-right, overwriting default parameters even if there are later parameters without defaults.

```
<script>
    function setData(name="not specified", age){
        console.log(name);
        console.log(age);
    }

    setData("arjun",37); // will print arjun and 37
    setData(18); // will print 18 and undefined
</script>
```

Currying

Currying is an advanced technique to transform a function of arguments n, to n functions of one or fewer arguments.

```
<script>
    function add (a) {
        return function(b){
            return a + b;
        }
    }

    const ans = add(3)(4);
    console.log(ans);
</script>
```

Closures

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives a function access to its outer scope. In JavaScript, closures are created every time a function is created, at function creation time.

```
<script>
    function printName(){
        name = "arjun";

        return ()=>{
            console.log(name);
        }
    }

    const innerFunction = printName();

    innerFunction(); // this will print "arjun"
</script>
```

References

- https://www.interviewbit.com/javascript-interview-questions/
- https://developer.mozilla.org/en-US/docs/Web/JavaScript