

# JS Fundamentals Lesson 1

Mohamed Emary

August 16, 2023

## Main Notes

**Return to all the hard DOM Manipulation code snippets in all articles mentioned in JS Sections**  
*First of all read the highlighted notes from the [JavaScript.info](#) book.*

## Difference between `let`, `var`, and `const`

Keyword	Scope	Can be re-assigned	Can be re-declared	Hoisted	Temporal Dead Zone	Advantages	Disadvantages
<code>var</code>	<b>Function</b>	Yes	Yes	Yes	No	Can be useful for <b>hoisting</b>	Can lead to unexpected behavior due to hoisting and function scope
<code>let</code>	Block	Yes	No	No	Yes	Block scoping can prevent <b>variable leakage</b>	Cannot be hoisted, which can lead to <b>reference errors</b>
<code>const</code>	Block	No	No	No	Yes	Prevents accidental reassignment	Cannot be reassigned, which can be inconvenient in some cases

**The Temporal Dead Zone (TDZ)** is a behavior in JavaScript associated with the use of `let` and `const` variables. It's the period between entering the scope of the variable and the line where the variable is declared. During this period, any reference to the variable will result in a `ReferenceError`.

```
1 console.log(myVar); // ReferenceError: myVar is not defined
2 let myVar = 5;
```

**Variable leakage**, or global namespace pollution, happens in JavaScript when variables are unintentionally declared in the global scope. This usually occurs when a variable is defined without the `var`, `let`, or `const` keyword inside a function, making it automatically a global variable.

# MAIN NOTES

---

```
1 function example() {  
2     leakyVar = "I'm global!";  
3 }  
4  
5 example();  
6 console.log(leakyVar); // Outputs: "I'm global!"
```

## Some JavaScript Notes

- "use strict"; prevents the use of undeclared variables. like `x = 3.14;` which will throw an error. because it wasn't declared first. So it has to be `let x = 3.14;`
- Constant that are known prior to execution are named in UPPERCASE. like `const PI = 3.14;` while constants that are calculated at run time are named in lowercase. like `const area = PI * radius * radius;`
- Notice the difference between pre-increment and post-increment:

```
let x = 5;  
console.log(x++); //5 because x++ is post increment  
console.log(x);   //6 now x is already incremented  
console.log(++x); //7 because ++x is pre increment
```

- In JavaScript, you can do this `m = Math;` then use `m.sqrt(2)`. What you see is that we assigned the Math object to the variable m. So we can use it as a shortcut so we just type m instead of Math.
- Some info related to numbers representation:
  - JavaScript has only one type of numbers. Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc. JavaScript Numbers are *Always 64-bit* Floating Point (Double Precision). It uses the international IEEE 754 standard 1 bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa.
  - Floating point arithmetic is not always 100% accurate. Test this `let x = 0.2 + 0.1;` If you print x it will give you `0.30000000000000004`. This is because of IEEE 754 standard for storing and doing calculations on floating point numbers. We can also solve the problem if we multiply and divide: `let x = (0.2 * 10 + 0.1 * 10) / 10;`
- If you add a number and a string, the result will be a string concatenation:

## Some notes on numbers

**A common mistake is to expect this result to be 30:**

```
let x = 10;  
let y = 20;  
let z = "The result is: " + x + y; // The result is: 1020
```

**A common mistake is to expect this result to be 102030:**

```
let x = 10;  
let y = 20;
```

## MAIN NOTES

---

```
let z = "30";
let result = x + y + z; // 3030
```

JavaScript will try to convert strings to numbers in all numeric operations:

```
let x = "100";
let y = "10";
let z = x / y; // 10
```

This is a NaN: `let x = 100 / "Apple";`

```
let x = NaN;
let y = "5";
let z = x + y; // NaN5
```

NaN is a number: `typeof NaN` returns number.

Never write a number with a leading zero (like 07). Some JavaScript versions interpret numbers as octal if they are written with a leading zero.

The `toString()` method can output numbers from base 2 to 36:

Decimal 32 different representations:

Code will be like: `MyNumber.toString(base);`

Base	Base Name	Result	Conversion
36	Hexatrigesimal	w	$w = 32$
32	Duotrigesimal	10	$1 \times 32 + 0 \times 1 = 32$
16	Hexadecimal	20	$2 \times 16 + 0 \times 1 = 32$
12	Duodecimal	28	$2 \times 12 + 8 \times 1 = 32$
10	Decimal	32	$3 \times 10 + 2 \times 1 = 32$
8	Octal	40	$4 \times 8 + 0 \times 1 = 32$
2	Binary	100000	$1 \times 32 + 0 \times 16 + 0 \times 8 + 0 \times 4 + 0 \times 2 + 0 \times 1 = 32$

For example:

```
1 let x = 15;
2 x.toString(2); // returns 1111
```

Numbers can be defined as objects with the new keyword: `let y = new Number(123);`. However, avoid new keyword. It complicates the code, slows down execution, and produce unexpected results.

```
let myNumber = "74";
myNumber += 3;
console.log(myNumber); // 743
typeof myNumber; // string not number
```

The + operator

```
// No effect on numbers
let x = 1;
alert( +x ); // 1
```

```
let y = -2;
alert( +y ); // -2

// Converts non-numbers
alert( +true ); // 1
alert( +"" ); // 0
```

## Operators in JS

Comparison between == and === operators in JS

Operator	Name	Description	Example
==	Equality	Test whether the values are the same but not whether the values' datatypes are the same	5 == "5" returns true because JavaScript converts the string "5" to the number 5 before making the comparison.
===	Strict Equality	Test the equality of both the values and their datatypes	5 === "5" returns false because the operands are of different types.

*The strict versions tend to result in fewer errors, so we recommend you use them.*

**Comparing two JavaScript objects always returns false.**

```
x = new Number(500);
y = new Number(500);
console.log(x == y); // false
console.log(x === y); // false
```

**The comma operator allows us to evaluate several expressions, dividing them with a comma ,. Each of them is evaluated but only the result of the last one is returned.**

```
let a = (1 + 2, 3 + 4);
alert( a ); // 7 (the result of 3 + 4)
```

Please note that the comma operator has very low precedence, lower than =, so parentheses are important in the example above.

Without them: `a = 1 + 2, 3 + 4` evaluates + first, summing the numbers into `a = 3, 7`, then the assignment operator = assigns `a = 3`, and the rest is ignored. It's like `(a = 1 + 2), 3 + 4`.

[See this exercise](#)

**Review MDN Articles Hard JS Code**