VARIABLE AND OPERATORS

A. Variable

```
let a;  // declare a variable
a = 'Hello!';  // assignment

console.log(a) ; // shows the variable content
alert(message); // shows the variable content
```

Let : we can change the value (of variable)
Const : unchanging value (of variable)
Var(the old-school declaring variable) is almost same with let. (avoid var).

1. Variable naming

> There are two limitations on variable names in JavaScript:
>
> 1. The name must contain only letters, digits, or the symbols `$` and `_`.
> 2. The first character must not be a digit.

khusus untuk const:

> Diberi nama dg huruf kapital(utk nilai yg sulit diingat) ketika nilainya sudah diketahui sebelum web loading, misal: const COLOR_WHITE = 00F
>
> Diberi nama dg huruf biasa(camelCase) ketika nilainya diketahui setelah web loading, misal: const pageLoadTime

B. Numbers

1. Arithmethic

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Remainder)
++	Increment
--	Decrement

1) Operator Precedence

> Multiplication (`*`) and division (`/`) have higher **precedence** than addition (`+`) and subtraction (`-`).
>
> And (as in school mathematics) the precedence can be changed by using parentheses.

2. Numbers

JS has only one type of number. Numbers can be written with or without decimals.
Extra large or extra small numbers can be written with scientific (exponent) notation:

```
let x = 123e5;    // 12300000
let y = 123e-5;   // 0.00123
```

- Integers (numbers without a period or exponent notation) are accurate up to 15 digits.
- The maximum number of decimals is 17.
- Floating point arithmetic isn't always 100% accurate:

```
4  let x = 0.2 + 0.1;   //not always 100% accurate:
5  let x = (0.2 * 10 + 0.1 * 10) / 10;   //To solve the problem
```

1. Numbers & String

WARNING: JS use '+' for both addition and concatenation. Numbers are added. Strings are concatenated. If you add number and string, the result will be string concatenation.

- A common Mistake

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

The result is: 1020 (not 30) because z is add string with number(concat)

```
let x = 10;
let y = 20;
let z = "30";
let result = x + y + z;
```

The result is 3030(not 102030) because x + y is addition and then + z is concatenation

- JS will try to convert strings to numbers in all numeric operations

```
4  let x = "100";
5  let y = "10";
6  let a = x / y; // This will work
7  let b = x * y; // This will work
8  let c = x - y; // This will work
9  let d = x + y; // This will not work because JS use '+' for concat
```

- NaN – Not a Number

Trying to do arithmetic with a non-numeric string will result in NaN

```
let x = 100 / "Apple";
```

However, if the string is numeric, the result will be a number:

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

If you use NaN in a mathematical operation, the result will also be NaN:

```
let x = NaN;
let y = 5;
let z = x + y;
```

Or the result might be a concatenation like NaN5:

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

- Infinity (or -Infinity)

```
4  // Execute until Infinity
5  while (myNumber != Infinity) {
6    myNumber = myNumber * myNumber;
7  }
8  // Division by 0 (zero) also generates Infinity
9  let x =  2 / 0;
10 let y = -2 / 0;
```

- Hexadecimal
  JS interprets hexadecimal if they are preceded by 0x. example `let x = 0xFF;`
  toString() method to output numbers from base 2 to 36

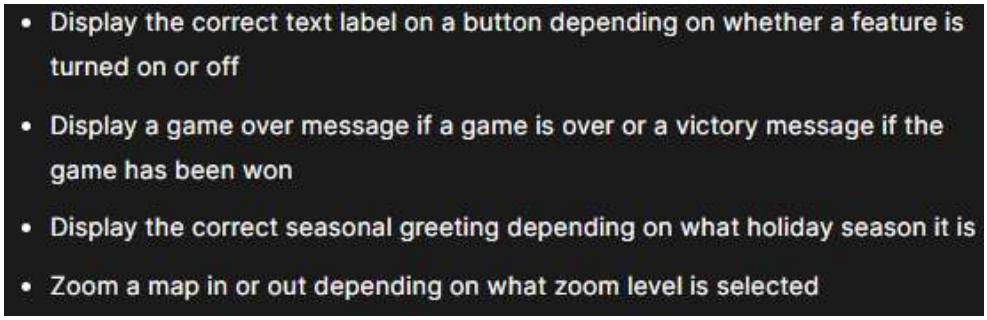
```
let myNumber = 100;
myNumber.toString(32);
myNumber.toString(16);  // Hexadecimal is base 16.
myNumber.toString(12);
myNumber.toString(10);  // Decimal is base 10.
myNumber.toString(8);   // Octal is base 8.
myNumber.toString(2);   // Binary is base 2.
```

- JS Numbers as Objects
  Note: don't create number objects `let y = new Number(123);`
  The 'new' keyword complicates the code and slows down execution speed. Number Objects can produce unexpected results.

2. Useful number methods
   - toFixed(2) : untuk membuat angka desimal menjadi 2 angka dibelakang koma(dengan pembulatan.
   - Number("3") : converts its argument to number data types(if it can).
   - String(3) : converts its argument to string.

3. Strict & Loose Equality
   is operator checks whether its two operands are equal, returning a Boolean result.
   Strict equality : ' === ' (value and data type must be the same)
   Loose equality : ' == ' (value must be the same, but data type does not have)

- Common use (booleans) for conditional



C. OPERATORS

1. Unary & Binay
   - An operator is unary if it has a single operand (i.e -3).
   - An operator is binary if it has two operands.

- Numeric conversion, unary +

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

```
let apples = "2";
let oranges = "3";

// both values converted to numbers before the binary plus
alert( +apples + +oranges ); // 5
```

Note: unary plus is higher precendence than any general operators (like unary negation, exponentioal, multiplication).

2. Assignment

```
let c = 3 - (a = b + 1);

alert( a ); // 3
alert( c ); // 0
```

sometimes we see it in JS libraries. Don't write the code like that, because it's make code hard to read.

- Chaining Assignment

```
a = b = c = 2 + 2;  // a=4, b=4, c=4
```

Once again, for the purposes of readability it's better to split such code into few lines:

```
c = 2 + 2;
b = c;
a = c;
```

3. Increment & Decrement (++ and --)

Note: Increment/decrement can only be applied to variables. Trying to use it on a value like 5++ will give an error.

1) prefix form (++a) : return new value.
2) postfix form (a++) : return old value.

The difference between them

- If the result of increment/decrement is not used, there is no difference in which form to use:

```
let counter = 0;
counter++;
++counter;
alert( counter ); // 2, the lines above did the same
```

- If we'd like to increase a value and immediately use the result of the operator, we need the prefix form:

```
let counter = 0;
alert( ++counter ); // 1
```

- If we'd like to increment a value but use its previous value, we need the postfix form:

```
let counter = 0;
alert( counter++ ); // 0
```

4. Comma

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)
```

Why do we need an operator that throws away everything except the last expression?

```
// three operations in one line
for (a = 1, b = 3, c = a * b; a < 10; a++) {
 ...
}
```