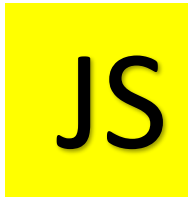

Types, Values, and Variables

Asst.Prof. Dr. Umaporn Supasitthimethee
ผศ.ดร.อุมาพร สุภสิทธิเมธี

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types

JavaScript: The Definitive Guide, Seventh Edition, by David Flanagan

Basic JavaScript Statements

- JavaScript is case-sensitive
- Semicolon in the end of statement is an optional
  - ✓console.log(x);
  - ✓console.log(y)
- Comment
  - //Single Line Comment
  - /* … */ Single or Multiple Lines Comment
- Console Printing
  - console.log (*variable*)

Reserved Words

Must not be used any of these reserved words as identifiers

as	const	export	get	null	target	void
async	continue	extends	if	of	this	while
await	debugger	false	import	return	throw	with
break	default	finally	in	set	true	yield
case	delete	for	instanceof	static	try	catch
do	from	let	super	typeof	class	else
function	new	switch	var			

JavaScript Data Types

- **Primitive values** (immutable values)
  - **Number**
    - integer between -2^{53} to 2^{53}
    - floating-point numbers (64 bits floating point format defined by IEEE754 standard)
  - **String** - support Unicode characters, use UTF-16 encoding, zero-based indexing
  - **Boolean –** true or false
  - **BigInt** - for very large integers, created by appending *n* to the end of an integer literal
  - **Symbol** - a built-in object, non-string property names, guaranteed to be unique
  - **Null** - a missing object
  - **Undefined** – a variable that has not been assigned a value
- **Objects** (mutable) - Collections of properties

Note that JavaScript does not have a special type that represents a single element like character

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures

Immutable Primitive Values

- Primitive types are **immutable**
  - No way to change a primitive value
  - For example: all string methods return a modified string are as a new string value.

Source codes: //types-variables/mutable

Mutable Object

- Object types are **mutable**
  - A value of a mutable type can change:
    - can change the values of **object properties and array elements**

```
let std = { firstname: 'Somchai', lastname: 'Rakdee' }
std.lastname = 'Deejai'
console.log(std) //{ firstname: 'Somchai', lastname: 'Deejai' }
let scores = [10, 25, 30, 50]
scores[0] = 5
scores[scores.length - 1] = 100
console.log(scores) //[ 5, 25, 30, 100 ]
```

Source codes: //types-variables/mutable

The following table summarizes the possible return values of typeof

returns the type name of the variable as a lowercase string.

Type	Result
Undefined	`"undefined"`
Null	`"object"` (see below)
Boolean	`"boolean"`
Number	`"number"`
BigInt (new in ECMAScript 2020)	`"bigint"`
String	`"string"`
Symbol (new in ECMAScript 2015)	`"symbol"`
Function object (implements [[Call]] in ECMA-262 terms)	`"function"`
Any other object	`"object"`

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof

Literals (data value)

- `15` // The number fifthteen
- `1.5` // The number one point five
- `6.53e15` // 6.52 x 10^{15}
- `"Hello World"` // A string of text
- `'SIT'` // Another string
- `` `"I ' am a student", I said` `` // Another string
- `true` // A Boolean value
- `false` // The other Boolean value
- `null` // Absence of an object, typeof null is object
- `undefined` //a value for a variable that has not been assigned a value
  // typeof undefined is undefined

Escape sequences can be used in JavaScript: \n,\t, \\, \b, …

A JavaScript Identifier

- An identifiers are used to name constants, variables, properties, functions, and classes.
- A JavaScript identifier usually starts with a letter, underscore (_), or dollar sign ($).
- Subsequent characters can also be digits (0–9).
- Because JavaScript is case sensitive, letters include the characters ***A*** through ***Z*** (uppercase) as well as ***a*** through ***z*** (lowercase).
- For examples,

```
my_variable, i, section1, $num, _value, DOLLAR_2_BAHT
```

Declarations and Types

- There is no type associated with JavaScript's variable declaration.
- JavaScript variable can hold a value of any types.

```
let x=10
x = 'hello world'
```

Source codes: //types-variables/declaration

Variable Declaration

```
var x=0                  //0
var y=1, z               //1, undefined
let i=5                  //5
let sum, avg=0           //undefined, 0
const DOLLAR_2_BAHT = 35.32  //32.5
```

JavaScript Scope

- Scope determines the accessibility (visibility) of variables.
- Before ES6 (2015), JavaScript had only Global Scope and Function Scope.
- JavaScript has **3 types of scope**:
  - **Block scope** - JavaScript block scope is bounded by { }
  - **Function scope** – Each function creates a new scope
  - **Global scope** - Variables declared Globally (outside any function or block scope) have Global Scope

Source codes: //types-variables/scopes

https://www.w3schools.com/js/js_scope.asp

Three Kinds of Variable Declaration

- **var**

  Declares a function-scoped or globally-scoped variable (**NOT have block scope)**, optionally initializing it to a value.

- **let**

  Declares a block-scoped, local variable, optionally initializing it to a value.

- **const**

  Declares a block-scoped, read-only named constant.

Note that

1. *In modern JavaScript (ES6 and later), using declaring variable with let and const.*
2. *When let and const declare at the top level, outside of any code blocks is called global variable or global constant and has global scope.*
3. *All var variables and functions declared globally become properties and methods of the window object.*

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var

Block Scope

- Variables (`let, const`) declared inside a { } block **cannot be accessed from outside the block**.
- Variables declared with the `var` keyword **can NOT have block scope** so variables declared inside a `{}` block **can be accessed from outside the block**.

```
// block scope
{
  let x = 2
  const y = 2
  var z = 2
}
// console.log(x) //x is not defined because cannot be used outside block
// console.log(y) //y is not defined because cannot be used outside block
console.log(z) //can use z because var variable can be accessed from outside the block

```

Function Scope

- Variables (`let, const, var`) declared within a JavaScript function, become **local to the function as local variables**
- They can only be accessed from within the function
- Local variables are created when a function starts and deleted when the function is completed.

```
1  // code here can NOT use all studentNames
2  function myFunction() {
3    let studentName1 = 'Somchai'
4    const studentName2 = 'Somsak'
5    var studentName3 = 'Somsri'
6    console.log(studentName1)
7    console.log(studentName2)
8    console.log(studentName3)
9  }
10 myFunction()
11 // code here can NOT use all studentNames
```

Global Scope

- A variable declared outside a function or a block, becomes **global**.
- Global variables can be accessed from anywhere in a JavaScript program.

```
let num1 = 10
const num2 = 20
var num3 = 30

function testGlobalScope() {
  console.log(num1)
  console.log(num2)
  console.log(num3)
  num1++
  // num2++ //constant cannot change value
  num3++
}

testGlobalScope()
console.log(num1)
console.log(num2)
console.log(num3)
```

JavaScript String

- The JavaScript type for representing text is the string.
- A string is an immutable ordered sequence of 16-bit values.
- JavaScript's strings (and its arrays) use zero-based indexing: the first 16-bit value is at position 0, the second at position 1, and so on.
- The empty string is the string of length 0.
- JavaScript does not have a special type that represents a single element of a string. To represent a single 16-bit value, simply use a string that has a length of 1.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Template Literals

```
let name = `Umaporn`
let greeting = `Hello ${ name }.`;
```

- This is more than just another string literal syntax, however, because these template literals can include arbitrary JavaScript expressions.
- Everything between the **${ }** is interpreted as **a JavaScript expression**.
- Everything **outside the curly braces** is **normal string literal text**
- The final value of a string literal in backticks is computed by
  - evaluating any included expressions,
  - converting the values of those expressions to strings and
  - combining those computed strings with the literal characters within the backticks

JavaScript implicit type conversions

Value	to String	to Number	to Boolean
`undefined`	`"undefined"`	`NaN`	`false`
`null`	`"null"`	`0`	`false`
`true`	`"true"`	`1`	
`false`	`"false"`	`0`	
`""` (empty string)		`0`	**`false`**
`"1.2"` (nonempty, numeric)		`1.2`	`true`
`"one"` (nonempty, non-numeric)		`NaN`	`true`
`0`	`"0"`		**`false`**
`-0`	`"0"`		**`false`**

Value	to String	to Number	to Boolean
`1` (finite, non-zero)	`"1"`		`true`
`Infinity`	`"Infinity"`		`true`
`-Infinity`	`"-Infinity"`		`true`
`NaN`	`"NaN"`		**`false`**
`[]` (empty array)	`""`	`0`	`true`
`[9]` (one numeric element)	`"9"`	`9`	`true`
`['a']` (any other array)	*use join() method*	`NaN`	`true`

```
// Implicit Type Conversions
const conv1 = 10 + ' rooms' //10 rooms
const conv2 = '4' * 5  //20
const conv3 = 'a' - 2  //NAN
const conv4 = !'Hello' //false
```

Source codes: //types-variables/conversion

Explicit Conversions

- Although JavaScript performs many type conversions automatically, you may sometimes need to perform an explicit conversion, or you may prefer to make the conversions explicit to keep your code clearer.
- The simplest way to perform an explicit type conversion is to use the `Boolean()`, `Number()`, and `String()` functions: