JS Types, Values, and Variables

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

JavaScript | MDN (mozilla.org)

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

Basic JavaScript Statements

- Semicolon in the end of statement is an optional
  - let x=10;
  - let y=20
- Statement can take up multiple lines
- Comment
  - //Single Line Comment
  - /* … */ Single or Multiple Lines Comment
- Console Printing
  - Console.log (*variable*);

Reserved Words

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			

Types

JavaScript types can be divided into **two** categories:

1. **primitive types**
    - number - including integer and floating-point numbers between -2^{53} to 2^{53}
    - string
    - boolean

    Primitive value
    - number
    - string
    - boolean
    - null (*special type*)
    - undefined (*special type*)
    - symbol (*special type*)
2. **object types**
    - An object (that is, a member of the type object) is a collection of properties where each property has a name and a value (either a primitive value or another object)
    - a special kind of object, known as ***an array***, that represents an ordered collection of numbered values

JavaScript Data Types: numbers, string, boolean, undefined, symbol, object

//02_TypesValuesVariables/script2.js

```
let myNum = 0;
console.log(`type of myNum is ${typeof myNum}`);

let myString = 'Good';
console.log(`type of myString is ${typeof myString}`);

let myBool = true;
console.log(`type of myBool is ${typeof myBool}`);

let myUndefined;
console.log(`type of myUndefined is ${typeof myUndefined}`);

let mySymbol = Symbol();
console.log(`type of mySymbol is ${typeof mySymbol}`);

let myNull = null;
console.log(`type of myNull is ${typeof myNull}`);

let myArr = [1, 2, 3];

console.log(`myArr Length: ${myArr.length}`);
console.log(`type of myArr is ${typeof myArr}`);

let myObj = {id: 1, task: 'grading exam'};

console.log(`${JSON.stringify(myObj)}`);
console.log(`type of myObj is ${typeof myObj}`);
```

```
//output
type of myNum is number
type of myString is string
type of myBool is boolean
type of myUndefined is undefined
type of mySymbol is symbol
type of myNull is object
```

Null and undefined

- `null` is a language keyword that evaluates to a special value.
- `null` represent normal, expected absence of value and if there is no value, the value of variable can be set to `null`. If a variable is meant to later hold an object, it is recommended to initialize to `null`.
- Using the `typeof` operator on `null` returns the string `"object"` indicating that `null` can be thought of as a special object value that indicates "empty object pointer".
- JavaScript also has a second value that indicates absence of value. The `undefined` value represents **unexpected absence of value**, a deeper kind of absence.
  - the value of variables that have not been initialized
  - the value you get when you query the value of an object property or array element that does not exist.
  - value of functions that do not explicitly return a value
  - value of function parameters for which no argument is passed
- If you apply the `typeof` operator to the `undefined` value, it returns `"undefined"`, indicating that this value is the sole member of a special type.

The following table summarizes the possible return values of `typeof`.

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"`

Literals

- 15 // The number twelve
- 1.5 // The number one point two
- "Hello World" // A string of text
- 'Hi' // Another string
- ` "I' am a student“, I said ` // Another string
- true // A Boolean value
- false // The other Boolean value
- null // Absence of an object

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

Identifiers

- **Identifiers** are used to name constants, variables, properties, functions, and classes and to provide labels for certain loops in JavaScript code.
- A JavaScript identifier **must begin with a letter, an underscore (_), or a dollar sign ($).** Subsequent characters can be letters, digits, underscores, or dollar signs. (Digits are not allowed as the first character so that JavaScript can easily distinguish identifiers from numbers.)
- JavaScript is a **case-sensitive language**. This means that language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters.

let, var, const variables

- One of the features that came with ES6 is the addition of `let` and `const`, which can be used for variable declaration.
- `var` declarations are **globally scoped or function/locally scoped**.
- The scope is global when a `var` variable is declared outside a function. This means that any variable that is declared with `var` outside a function block is available for use in the whole window.
- All variables and functions declared globally with `var` **become properties and methods of the `window` object.**
- `var` is function scoped when it is declared within a function. This means that it is available and can be accessed only within that function.

var variables

// 01_BasicJS/ script5.js

```
//greeting is globally scope, it exists outside a function
var greeting = 'Hey’;

//var variables can be re-declared and updated
var greeting = 'Ho Ho’;

function greeter() {
  //msg is function scoped, we cannot access the variable msg outside of a function
  var msg = 'hello’;
}

// console.log(msg); //error: msg is not defined
console.log(greeting);
```

var variables can be re-declared and updated

This means that we can do this within the same scope and won't get an error.

```
var year = 'leap';
if (year === 'leap’)
   var greeting = 'Hey 366 days’; //re-declared
console.log(greeting);
```

It becomes a problem when you do not realize that a variable greeting has already been defined before.

let variables

- `let` is now preferred for variable declaration.
- JavaScript block of code is bounded by {}. A block lives in curly braces. Anything within curly braces is a block.
- `let` is **block scoped**, a variable declared in a block with `let` is only available for use within that block.
- `let` **can be updated but not re-declared.**

let can be updated but not re-declared.

// 01_BasicJS/ script6.js

```
/*let variables*/
//greeting is block scope
let greeting = 'Hey';
//let variables cannot be re-
declared, only can be updated
greeting = 'Ho Ho';
function greeter() {
  //msg is function scoped, we cannot access the
 variable msg outside of a function
  let msg = 'hello';
}
// console.log(msg); //error: msg is not defined
console.log(greeting);

let year = 'leap';
if (year === 'leap’)
    greeting = 'Hey 366 days';
console.log(greeting);
```

if the same variable is defined in different scopes, there will be no error. This is because both instances are treated as different variables since they have different scopes.

```
let greeting = 'Hey';
greeting = 'Ho Ho';
function greeter() {
    let greeting = 'Good morning';
    console.log(`greeting in function is ${greeting}`);
}

greeter(); console.log(greeting); //Ho Ho
```

const

- Variables declared with the `const` maintain constant values.
- `const` declarations share some similarities with `let` declarations.
- Like `let` declarations, `const` declarations can only be accessed within the block they were declared.
- `const` **cannot be updated or re-declared**
- Every `const` declaration, therefore, **must be initialized at the time of declaration**.

// 01_BasicJS/ script7.js

```
/*const variables*/
const greeting = 'Hey';
//const variables cannot be re-declared
// const greeting = 'Ho Ho';
//const variables cannot be updated
// greeting = 'Hi Hi';
```


https://www.freecodecamp.org/news/var-let-and-const-whats-the-difference/

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.

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

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:

```
//Explicit Conversions
Number('3'); //  3
String(false); // "false"
Boolean([]); //  true
```

how values convert from one type to another in JavaScript?

```
1 + ' objects'; //"1 objects":  Number 1 converts to a string
'5' * '4'; //20: both strings convert to numbers
let n = 'y' + 1; // n == NaN; string "y" can't convert to a number
```

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`**
`1` (finite, non-zero)	`"1"`		`true`
`Infinity`	`"Infinity"`		`true`
`-Infinity`	`"-Infinity"`		`true`
`NaN`	`"NaN"`		**`false`**
`[]` (empty array)	`""`	`0`	`true`

The primitive-to-primitive conversions shown in the table are relatively straightforward but Object-to-primitive conversion is somewhat more complicated

```
//examples of implicit type conversions
x + ""      // String(x)
+x          // Number(x)
x-0         // Number(x)
!!x         // Boolean(x)
```

//02_TypesValuesVariables/script5.js