

Data-Types, Primitives & Variables

JavaScript



What are Data-Types?

Most any programming language has a set of “Data-Types.” This is the type of data that can be inputted, stored, used, outputted, or handled by programs written in that language.

For example, you might want to store some text for a user’s name. Or a floating-point (decimal) number for the price of a product!

Different types of data can often be used in specific ways. Think about it... if you have a name, you might want to put that in a sentence: “Hello, Bob!” If you have a number, you might want to add that to another number: $5.05 + 2.50 = 7.55$.

You typically wouldn’t want to apply mathematical addition to some text. Programming languages often enforce rules to ensure we are dealing with data correctly. JavaScript isn’t particularly strict on this, so as developers we have to be careful with our data to ensure our operations don’t lead to unexpected results.

Data-Types in JavaScript

Primitives are data-types that are not considered “objects” in the traditional sense (you do not need to use the “new” keyword to instantiate them) and can be distinguished using the “typeof” operator.

Primitives in JavaScript include:

- undefined
- Boolean
- Number
- String
- BigInt
- Symbol

Structural types are often a bit more complex, requiring the “new” keyword to create one out of predefined classes or sometimes the “function” keyword.

Structural types include:

- Objects
- Functions

Lastly, there is structural root primitive—of which there is only one:

- null

How do we work with
data?

Variables and Data/Values

If we want to keep track of a value for use later in our code, we use what is called a “[variable](#).”

Think of a variable as a name or label, representing a value. See the following example:

```
myName = “Theodore”
```

This isn’t real code, but it helps illustrate what some real code might look like. The pseudo-code above suggests that the label myName represents the value: “Theodore”.

Now if I were to ask for the value of myName we would expect to see “Theodore”!

We can actually try this
out in the Web Console.

Assigning a Variable a Value

The “=” sign is the assignment operator. It is used exclusively to assign a value to a variable by name.

If you open your Web Console and try out the previous pseudo-code...

```
myName = "Theodore"
```

The name “Theodore” will be stored in your variable: myName

You can confirm this by outputting the myName variable: output(myName). The Web Console will output the name stored in that variable.

```
>> myName = "Theodore"  
← "Theodore"  
  
>> myName  
← "Theodore"
```

Declarations

When you are making a new variable and assigning it, there are special keywords we should be using. Each one works a little bit differently—let's go over each now!

- **var**

Var is no longer regularly used, it is seen as bad practice with few exceptions. Variables declared using the var keyword are function-scoped, and can be reassigned. This is how all variables used to be declared.

- **let**

Variables declared with the let keyword are block-scoped (this means they are more conservative with memory usage), and can be reassigned. If you know your variable might need to be reassigned, this is the way to go!

- **const**

Variables declared with the const keyword are block-scoped, and cannot be reassigned. If the value of the variable will never be reassigned, use this. Attempting to re-assign a const variable will result in an error.

Let's try that Assignment Again

We should always use declaration keywords. Let's get in the habit of thinking which keyword to use!

Previously we typed...

```
myName = "Theodore"
```

This isn't good code! Following proper JavaScript syntax and convention it is expected that we use a declaration keyword, and end our line with a semicolon.

When in doubt, try using `const`. If you run into an error later, you can always change your code to make use of `let` instead!

```
const myName = "Ted";
```

```
>> const myName = "Ted";  
← undefined  
  
>> myName  
← "Ted"
```

Variable Scope

Scope determines the visibility of variables to other parts of the program.

Variables declared inside a `{ }` block cannot be accessed by code outside of the block! For example:

```
{  
  
    let myNum = 1;  
  
    let myNum2 = myNum;  
  
    //this is valid code as both  
    variables are in the same block  
  
}  
  
let myNum3 = myNum; //this is invalid  
code, as myNum is in a block and  
cannot be accessed by myNum3, which is  
outside of the block
```

Let's have a closer look at
some of the most common
data-types.

Strings

Strings are just text; our previous examples were strings!

A string is a collection of zero or more characters.

Note that text characters can be letters, numbers, spaces, or even symbols like an exclamation mark!

Whenever you use a string in an expression or you assign one as a value, you must wrap the text in quotes. Single or double quotes will do.

Examples of strings...

```
>> ""  
← ""  
  
>> "Testing123!"  
← "Testing123!"  
  
>> "1"  
← "1"  
  
>> " "  
← " "  
  
>> "This is a string."  
← "This is a string."
```

Numbers

Numbers can be integers (whole numbers) or floating-point (decimal) numbers.

Numbers are... well... numbers!

They're especially useful when we need to perform math operations.

Note that, unlike strings, we don't wrap numbers with quotations.

Some examples of numbers...

```
>> 3.14
< 3.14
>> 1
< 1
>> -3
< -3
>> 0
< 0
```

Booleans

Booleans are a binary data-type—there are only two possible states!

A boolean can either be true or false, nothing else!

Note that boolean values are also written without quotation marks.

Boolean values are capital-sensitive, which means you must specifically express them as true and false. Something like True or FALSE would result in an error, as the value is invalid.

```
>> true
< true

>> false
< false

>> TrUe
Uncaught ReferenceError: TrUe is not defined
    <anonymous> debugger eval code:1
    [Learn More]

>> FALSE
Uncaught ReferenceError: FALSE is not defined
    <anonymous> debugger eval code:1
    [Learn More]
```

null

When you encounter the need for an intentional absence of a value or any particular type, null is the available option.

You can assign a variable the value of null to communicate that there is no intended value or alternative type you'd like to give it.

This is about as empty and void of meaning you can make a value outside of not defining it at all.

null is capital-sensitive.

```
>> null
< null

>> let myNewVar = null;
< undefined

>> myNewVar
< null
```

undefined

When a variable exists, but there are no values assigned to it yet, you'll be met with the undefined value to let you know!

Try declaring a variable with something like...

```
const myTestVar;
```

or...

```
let test;
```

Notice there's no assignment! If you call upon these named variables later in your program, you'll be met simply with undefined to let you know.

undefined is capital-sensitive.

```
>> let test;  
← undefined  
  
>> test  
← undefined
```


Are there ways of telling
what data-type a variable
or value is?

The typeof Operator

If you place `typeof` before a value, JavaScript will return to you a string describing the data-type.

Give it a try in your Web Console!

`typeof` is capital-sensitive.

Note: Do to legacy (backwards-compatibility) reasons, `null` identifies as an object.

```
>> typeof "Is this a string?"  
← "string"  
>> typeof null  
← "object"  
>> typeof true  
← "boolean"  
>> typeof false  
← "boolean"  
>> typeof "34"  
← "string"  
>> typeof 34  
← "number"  
>> typeof undefined  
← "undefined"
```

Working With Primitives

JavaScript



Dealing with Strings.

Concatenation

We can “glue” two pieces of text together into one string using concatenation.

`+` is the concatenation operator. When it is found between strings, it will glue the text together!

Especially once variables get involved, this is extremely common.

```
>> "Hello, " + "World!"  
← "Hello, World!"
```

toUpperCase

A method of strings, capable of turning text into a new uppercase copy of the string.

To use a string method, follow a string value or variable with a period, the name of the method, and finally a pair of parentheses.

See the pattern in action below with the [toUpperCase](#) string method!

`"Hello, World!".toUpperCase()`

This example will return to you the text: "HELLO, WORLD!"

Don't forget that these methods will all be capital-sensitive—it's the JavaScript way!

```
>> "Hello, World!".toUpperCase()  
← "HELLO, WORLD!"
```

toLowerCase

A method of strings, capable of turning text into a new lowercase copy of the string.

As you can likely guess after trying out toUpperCase, the [toLowerCase](#) method does the opposite: a lower-cased version of the provided string will be returned to you.

`"Hello, World!".toLowerCase()`

```
>> "Hello, World!".toLowerCase()  
← "hello, world!"
```

includes

A method of strings, it checks if the string has a set of characters inside of it or not.

You enter a string as an argument to the [includes](#) method, and it will determine if that text exists inside of the string you're operating on.

The following will result in false, as "Z" is not inside of "Hello, World!".

```
"Hello, World!".includes( "Z" )
```

The following will result in true, as "or" does exist inside of "Hello, World!".

```
"Hello, World!".includes( "or" )
```

```
>> "Hello, World!".includes( "Z" )  
← false  
  
>> "Hello, World!".includes( "or" )  
← true
```


slice

A method of strings, it provides a cut version of the string containing only some of the text.

slice takes two different arguments: the position of the beginning letter and the position of the last letter you'd like to cut out.

Count the letters in the string and you'll see!

```
"Hello, World!".slice( 2, 5 )
```

This cuts out and returns to you just "llo", from the original string "Hello, World!".

Note that letter position starts at zero, not one!

Try changing the numbers between the parentheses, what happens?

```
>> "Hello, World!".slice( 2, 5 )  
← "llo"
```

replace

A method of strings, it replaces target text with new text.

replace takes two arguments: the text you'd like to target in the string, and the text you'd like to have take its place! It will return to you, a copy of the string with the replacement completed (if it found the target match, of course!)

Try this out...

```
"Hello, World!".replace( "Hello", "Goodbye" )
```

You'll get "Goodbye, World!" after the replacement has done its magic.

```
>> "Hello, World!".replace( "Hello", "Goodbye" )  
← "Goodbye, World!"
```

Regular Expressions with replace

The power of pattern-matching within strings can be quickly multiplied with the use of regular expressions ("RegEx".)

Note that one of the limitations (see figure below) of replace when passing a string in as your target, is that it will only replace the first match.

You can target more than one match using advanced pattern-matching via [Regular Expressions](#), or, RegEx.

Try the following:

```
"Hello, World!".replace( /l/g, "Y" )
```

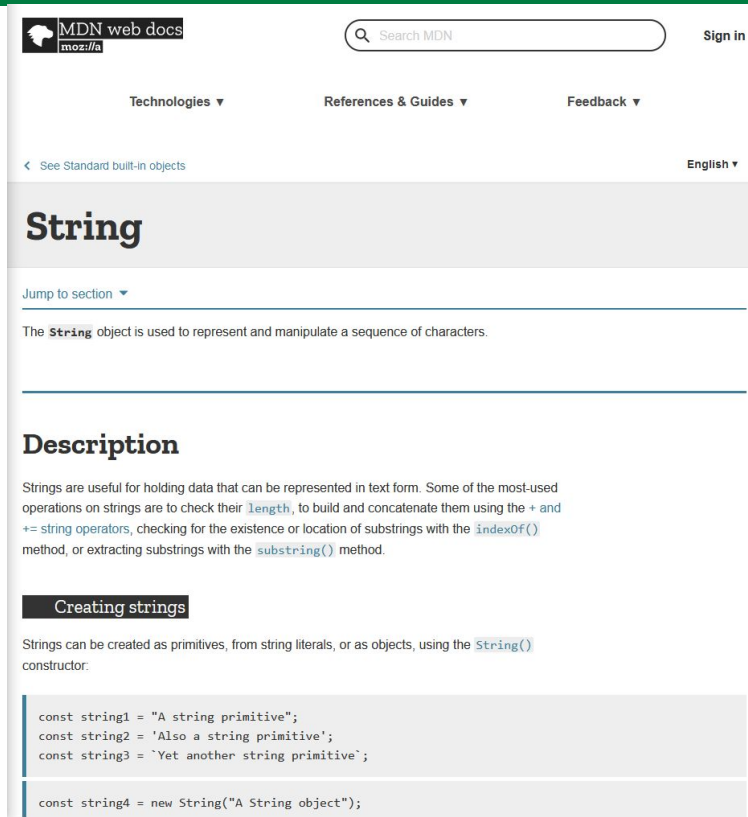
This will return to you the updated string "HeYYo WorYd!"

```
>> "Hello, World!".replace( "l", "Y" )  
← "HeYlo, World!"  
  
>> "Hello, World!".replace( /l/g, "Y" )  
← "HeYYo, WorYd!"
```

String Properties and Methods

There are plenty of more ways you can work with strings via their properties and methods.

[Peruse the full list here.](#)



The screenshot shows the MDN web docs page for the `String` object. The page has a green header with the MDN logo and a search bar. The main content area is white with a green sidebar on the left. The title "String" is prominently displayed in a large, bold, black font. Below the title, there is a section titled "Description" which explains that strings are useful for holding data that can be represented in text form. It mentions common operations like checking length, concatenation, and substring extraction. Below the description, there is a section titled "Creating strings" which explains that strings can be created as primitives or objects using the `String()` constructor. This section includes a code block with JavaScript code examples for creating strings in different ways.

MDN web docs

Search MDN

Sign in

Technologies ▼

References & Guides ▼

Feedback ▼

See Standard built-in objects

English ▼

String

Jump to section ▼

The **String** object is used to represent and manipulate a sequence of characters.

Description

Strings are useful for holding data that can be represented in text form. Some of the most-used operations on strings are to check their `length`, to build and concatenate them using the `+` and `+=` [string operators](#), checking for the existence or location of substrings with the `indexOf()` method, or extracting substrings with the `substring()` method.

Creating strings

Strings can be created as primitives, from string literals, or as objects, using the `String()` constructor:

```
const string1 = "A string primitive";
const string2 = 'Also a string primitive';
const string3 = `Yet another string primitive`;

const string4 = new String("A String object");
```

Working with Numbers.

Mathematical Operators

We can do math with JavaScript! Much of it will look pretty familiar.

`+` the addition operator.

`-` the subtraction operator.

`*` the multiplication operator.

`/` the division operator.

These work the way you'd expect! Give them a try in your Web Console.

```
>> 2 + 3  
← 5  
>> 10 - 6  
← 4  
>> 8 * 2  
← 16  
>> 12 / 3  
← 4
```

Modulus Operator

This symbol may feel a little less familiar—it is just a division remainder!

% is the modulus operator.

You can get the remainder of what would-be a division operation using this.

```
>> 25 % 4  
← 1
```

As an example, is commonly used for checking if a number is even or odd.

```
>> 12 % 2  
← 0
```

parseInt, parseFloat, and Number

We can transform strings composed of numeral characters into number data-type values.

The parseFloat and Number can be used to ensure a value is returned as a number (if possible.) If there are decimal values, they will be maintained.

parseInt works much the same, but will chop off any decimal points. Note that parseInt doesn't round up or down.

You can pass strings with numeral characters, or full-on numbers into these functions.

```
>> parseFloat( "37.5" )  
← 37.5  
  
>> Number( "64.215" )  
← 64.215  
  
>> parseInt( "3.98" )  
← 3
```


Addition Versus Concatenation

You may have noticed that JavaScript uses the same symbol for both addition and concatenation, we do have to be careful.

If a value on either side of a plus sign is a string, the operation will be concatenation.

If both values are of the number data-type, the operation will be addition.

Note you can leverage `parseInt`, `parseFloat`, and `Number` to convert strings if you need to ensure addition.

Try some of your own experiments!

```
>> 1 + "2"  
← "12"  
  
>> 1 + 2  
← 3  
  
>> 3 + 3 + "3"  
← "63"  
  
>> 3 + 3 + Number( "3" )  
← 9
```

Math Round, Floor, and Ceiling

JavaScript's built-in Math class has some great methods.

Math.round is used for traditional rounding of a number to the nearest integer (whole number.)

```
>> Math.round( 3.14 )  
← 3
```

Math.floor will round down to the nearest integer.

```
>> Math.floor( 5.9 )  
← 5
```

Math.ceil will round up to the nearest integer.

```
>> Math.ceil( 5.1 )  
← 6
```

Increment, decrement, and assignment operators

The following methods can be used to increment a value, decrement a value, or assign a value to a variable.

`++` will increment a value up by 1.

```
let myNum = 1;
```

```
myNum++; //myNum is now 2
```

`--` will decrement a value by 1.

```
let myNum = 1;
```

```
myNum--; // myNum is now 0
```

`=` will assign a value to a variable.

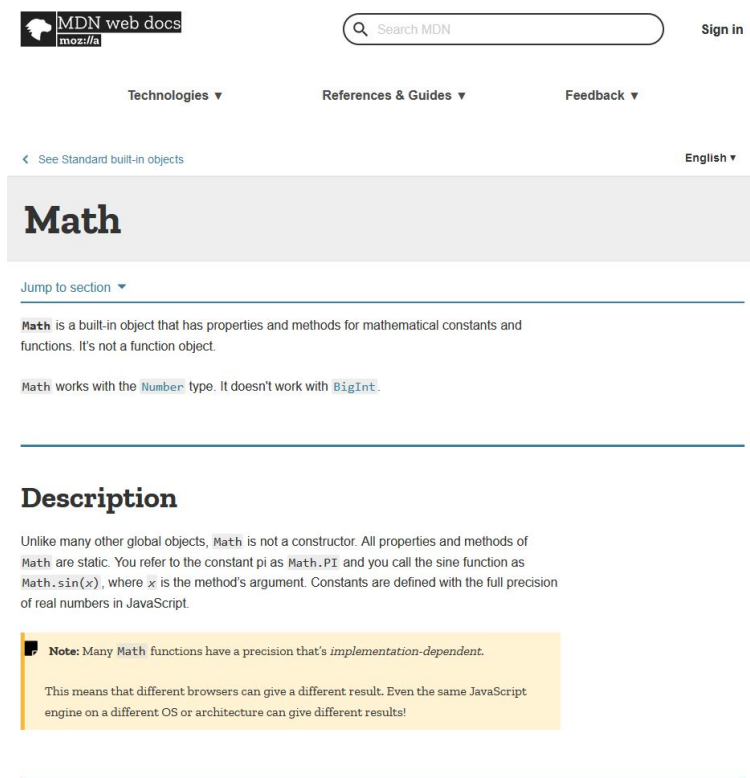
```
let myNum = 50;
```

```
let myNum2 = myNum; //myNum2 is now 50
```

Math Object

JavaScript's Math class has a lot more built-in properties and methods.

[Check out the full list to see what it is capable of!](#)



The screenshot shows the MDN web docs page for the `Math` object. At the top, there's a navigation bar with the MDN logo, a search bar, and a 'Sign in' link. Below the navigation bar, there are links for 'Technologies', 'References & Guides', and 'Feedback'. The main heading is 'Math'. Below the heading, there's a 'Jump to section' dropdown. The text describes `Math` as a built-in object for mathematical constants and functions, noting it's not a function object. It also mentions that `Math` works with the `Number` type but not with `BigInt`. A 'Description' section follows, explaining that `Math` is not a constructor and its properties and methods are static. It provides an example of using `Math.PI` and `Math.sin(x)`. A 'Note' box at the bottom states that many `Math` functions have a precision that's implementation-dependent, meaning different browsers or architectures might give different results for the same calculation.

MDN web docs
mozilla

Search MDN

Sign in

Technologies ▼ References & Guides ▼ Feedback ▼

◀ See Standard built-in objects English ▼

Math

Jump to section ▼

Math is a built-in object that has properties and methods for mathematical constants and functions. It's not a function object.

Math works with the `Number` type. It doesn't work with `BigInt`.

Description

Unlike many other global objects, **Math** is not a constructor. All properties and methods of **Math** are static. You refer to the constant pi as `Math.PI` and you call the sine function as `Math.sin(x)`, where `x` is the method's argument. Constants are defined with the full precision of real numbers in JavaScript.

Note: Many **Math** functions have a precision that's *implementation-dependent*.

This means that different browsers can give a different result. Even the same JavaScript engine on a different OS or architecture can give different results!

One More Thing!

NaN and Infinity

Another couple of results that may catch you off guard when dealing with numbers are the NaN and Infinity values.

NaN stands for “Not a Number.”

If a mathematical operation cannot be understood by JavaScript, or is impossible to express a result for within JavaScript’s ability, you’ll see this as the returned result.

JavaScript also has a representation for infinity (∞), intuitively named Infinity.

Remember: JavaScript is capital-sensitive!

```
>> Number( "This is a sentence, not a number!" );  
← NaN  
  
>> NaN  
← NaN  
  
>> 12 / 0  
← Infinity  
  
>> Infinity - Infinity  
← NaN
```

Escape Sequences

For when we want to put quotations within quotations.

When we need to put quotation marks in a string, how do we ensure that JavaScript does not take the embedded quotation mark as an end to our string? For example:

```
let text = "We are the so-called  
"Vikings" from the north.";
```

As we can see, JavaScript would assume that the string begins with “We”, and takes the quotation mark before “Vikings” as the end of the string. How we get around this is with escape sequences! In JavaScript, an escape sequence is a backslash, followed by the symbol (in this case, the quotation mark!)

```
let text = "We are the so-called  
\"Vikings\" from the north.";
```

JavaScript Worst Practices

The following are bad practices to avoid!

1. Make types explicit when possible!
This avoids implicit type conversion and unpredictable results.
2. Avoid using var! Let and const are better ways to declare variables!
3. Avoid using single character or gibberish variables. Remember that your code is likely to be read by others or yourself down the line. By making your variables descriptive, you will save both yourself and your colleagues some frustration.
4. Avoid inline code injection when possible. Don't be tempted to inject HTML and CSS into your code for convenience. Do it right!