JAVASCRIPT

Module Objectives:

JavaScript is used by programmers across the world **to create dynamic and interactive web content like applications and browsers**. JavaScript is so popular that it's the most used programming language in the world, used as a client-side programming language by 97.0% of all websites.

Overview :

This course provides you hands-on experience and exposure to develop JavaScript based web application. This course builds strong foundation of JavaScript which will help developer to apply JavaScript concepts for responsive web frontend and backend development.

Detailed course contents:

- JS INTRODUCTION
- JS Where To
- JS Output
- JS Statement
- JS Syntax
- JS Comments
- JS Variables
- JS Let
- JS Const
- JS Operators
- JS Arithmetic
- JS Assignment
- JS Precedence
- JS Comparisons
- JS Data Types
- JS Functions
- JS Objects
- JS Events
- JS Strings
- JS Conditions
- JS Switch
- JS Loops

JS INTRODUCTION

JavaScript is the world's most popular programming language.

JavaScript is the programming language of the Web.

JavaScript is easy to learn.

This tutorial will teach you JavaScript from basic to advanced.

Why Study JavaScript?

JavaScript is one of the **3 languages** all web developers **must** learn:

1. **HTML** to define the content of web pages
2. **CSS** to specify the layout of web pages
3. **JavaSccript** to program the behavior of web pages

JavaScript Can Change HTML Content

One of many JavaScript HTML methods is `getElementById()`.

The example below "finds" an HTML element (with id="demo"), and changes the element content (innerHTML) to "Hello JavaScript":

Example

```
document.getElementById("demo").innerHTML = "Hello JavaScript";
```

JavaScript Can Change HTML Styles (CSS)

Changing the style of an HTML element, is a variant of changing an HTML attribute:

Example

```
document.getElementById("demo").style.fontSize = "35px";
```

JavaScript Can Hide HTML Elements

Hiding HTML elements can be done by changing the `display` style:

```
<!DOCTYPE html>

<html>

<body>


<h2>What Can JavaScript Do?</h2>


<p id="demo">JavaScript can hide HTML elements.</p>
```

```
<button type="button"
onclick="document.getElementById('demo').style.display='none'">Click
Me!</button>

</body>

</html>
```

JavaScript Can Show HTML Elements

Showing hidden HTML elements can also be done by changing
the `display` style:

```
<!DOCTYPE html>

<html>

<body>


<h2>What Can JavaScript Do?</h2>


<p>JavaScript can show hidden HTML elements.</p>


<p id="demo" style="display:none">Hello JavaScript!</p>


<button type="button"
onclick="document.getElementById('demo').style.display='block'">Click
Me!</button>


</body>

</html>
```

JavaScript Where To

The <script> Tag

In HTML, JavaScript code is inserted between `<script>` and `</script>` tags.

```
<!DOCTYPE html>

<html>

<body>


<h2>JavaScript in Body</h2>


<p id="demo"></p>


<script>

document.getElementById("demo").innerHTML = "My First JavaScript";

</script>


</body>

</html>
```

JavaScript in <head> or <body>

You can place any number of scripts in an HTML document.

Scripts can be placed in the `<body>`, or in the `<head>` section of an HTML page, or in both.

JavaScript in <head>

In this example, a JavaScript `function` is placed in the `<head>` section of an HTML page.

The function is invoked (called) when a button is clicked:

```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>
</head>
<body>

<h2>Demo JavaScript in Head</h2>

<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>

</body>
</html>
```

JavaScript in <body>

In this example, a JavaScript `function` is placed in the `<body>` section of an HTML page.

The function is invoked (called) when a button is clicked:

```
<!DOCTYPE html>
<html>
<body>

<h2>Demo JavaScript in Body</h2>

<p id="demo">A Paragraph</p>

<button type="button" onclick="myFunction()">Try it</button>

<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>

</body>
</html>
```

External JavaScript

Scripts can also be placed in external files:

JavaScript files have the file extension **.js**.

To use an external script, put the name of the script file in the `src` (source) attribute of a `<script>` tag:

```
<script src="myScript.js"></script>
```

You can place an external script reference in `<head>` or `<body>` as you like.

External JavaScript Advantages

Placing scripts in external files has some advantages:

- It separates HTML and code
- It makes HTML and JavaScript easier to read and maintain
- Cached JavaScript files can speed up page loads

To add several script files to one page - use several script tags:

```
<script src="myScript1.js"></script>
<script src="myScript2.js"></script>
```

External References

An external script can be referenced in 3 different ways:

- With a full URL (a full web address)
- With a file path (like /js/)
- Without any path

This example uses a **full URL** to link to myScript.js:

```
<script src="https://www.w3schools.com/js/myScript.js"></script>
```

This example uses a **file path** to link to myScript.js:

```
<script src="/js/myScript.js"></script>
```

This example uses no path to link to myScript.js:

```
<script src="myScript.js"></script>
```

JavaScript Output

JavaScript Display Possibilities

JavaScript can "display" data in different ways:

- Writing into an HTML element, using `innerHTML`.
- Writing into the HTML output using `document.write()`.
- Writing into an alert box, using `window.alert()`.
- Writing into the browser console, using `console.log()`.

Using innerHTML

To access an HTML element, JavaScript can use
the `document.getElementById(id)` method.

The `id` attribute defines the HTML element. The `innerHTML` property defines the HTML content:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My First Paragraph</p>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>
```

```
</body>
</html>
```

Using document.write()

For testing purposes, it is convenient to use `document.write()`:

EXAMPLE:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Web Page</h1>
<p>My first paragraph.</p>

<script>
document.write(5 + 6);
</script>

</body>
</html>
```

Using console.log()

For debugging purposes, you can call the `console.log()` method in the browser to display data.

```
<!DOCTYPE html>
<html>
<body>

<script>
console.log(5 + 6);
</script>
```

```
</body>
</html>
```

JavaScript Print

JavaScript does not have any print object or print methods.

You cannot access output devices from JavaScript.

The only exception is that you can call the `window.print()` method in the browser to print the content of the current window.

```
<!DOCTYPE html>

<html>

<body>


<h2>The window.print() Method</h2>


<p>Click the button to print the current page.</p>


<button onclick="window.print()">Print this page</button>


</body>

</html>
```

JavaScript Statements

JavaScript statements are composed of:

Values, Operators, Expressions, Keywords, and Comments.

This statement tells the browser to write "Hello Dolly." inside an HTML element with id="demo":

```
<script>

document.getElementById("demo").innerHTML = "Hello Dolly.";

</script>.
```

Semicolons ;

Semicolons separate JavaScript statements.

Add a semicolon at the end of each executable statement:

```
<p id="demo1"></p>

<script>

let a, b, c;

a = 5; b = 6; c = a + b;

document.getElementById("demo1").innerHTML = c;

</script>
```

JavaScript Code Blocks

JavaScript statements can be grouped together in code blocks, inside curly brackets {...}.

The purpose of code blocks is to define statements to be executed together.

```
<p id="demo1"></p>

<p id="demo2"></p>



<script>

function myFunction() {
  document.getElementById("demo1").innerHTML = "Hello Dolly!";

  document.getElementById("demo2").innerHTML = "How are you?";

}
```

JavaScript Keywords:

JavaScript statements often start with a **keyword** to identify the JavaScript action to be performed.Here is a list of some of the keywords you will learn about in this tutorial:

Keyword	**Description**
var	Declares a variable
let	Declares a block variable
const	Declares a block constant
if	Marks a block of statements to be executed on a condition
switch	Marks a block of statements to be executed in different cases
for	Marks a block of statements to be executed in a loop
function	Declares a function

return	Exits a function
try	Implements error handling to a block of statements

JavaScript Syntax

```
// How to create variables:
var x;
let y;

// How to use variables:
x = 5;
y = 6;
let z = x + y;
```

JavaScript Values

The JavaScript syntax defines two types of values:

- Fixed values
- Variable values

Fixed values are called **Literals**.

Variable values are called **Variables**.

JavaScript Literals

The two most important syntax rules for fixed values are:

1. **Numbers** are written with or without decimals:

Number can be written with or without decimals.

```
10.50

1001
```

```
<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = 10.50;
</script>
```

2. **Strings** are text, written within double or single quotes:

```
"John Doe"

'John Doe'
```

JavaScript Variables

In a programming language, **variables** are used to **store** data values.

JavaScript uses the keywords `var`, `let` and `const` to **declare** variables.

An **equal sign** is used to **assign values** to variables.

```
let x;
x = 6;
```

x is defined as a variable. Then, x is assigned (given) the value 6

JavaScript Operators

JavaScript uses **arithmetic operators** (`+ - * /`) to **compute** values:

JavaScript uses an **assignment operator** (`=`) to **assign** values to variables:

```
let x, y;
x = 5;
y = 6;
```

JavaScript Expressions

An expression is a combination of values, variables, and operators, which computes to a value.

The computation is called an evaluation.

For example, 5 * 10 evaluates to 50:

JavaScript Keywords

JavaScript **keywords** are used to identify actions to be performed.

The `let & var` keywords tells the browser to create variables:

JavaScript Comments

Not all JavaScript statements are "executed".

Code after double slashes `//` or between `/*` and `*/` is treated as a **comment**.

Comments are ignored, and will not be executed:

```
let x = 5;   // I will be executed

// x = 6;   I will NOT be executed
```

JavaScript Identifiers / Names

Identifiers are JavaScript names.

Identifiers are used to name variables and keywords, and functions.

The rules for legal names are the same in most programming languages.

A JavaScript name must begin with:

- A letter (A-Z or a-z)
- A dollar sign ($)
- Or an underscore (_)

JavaScript is Case Sensitive

All JavaScript identifiers are **case sensitive**.

The variables `lastName` and `lastname`, are two different variables:

```
let lastname, lastName;
lastName = "Doe";
lastname = "Peterson";
```

Single Line Comments

JavaScript comments can be used to explain JavaScript code, and to make it more readable.

Single line comments start with //.

Any text between // and the end of the line will be ignored by JavaScript (will not be executed).

This example uses a single-line comment before each code line:

Multi-line Comments

Multi-line comments start with /* and end with */.

Any text between /* and */ will be ignored by JavaScript.

JavaScript Variables

4 Ways to Declare a JavaScript Variable:

- Using `var`
- Using `let`
- Using `const`
- Using nothing

Variable

Variables are containers for storing data (storing data values).

```
var x = 5;

var y = 6;
var z = x + y;
```

In this example, `x`, `y`, and `z`, are variables, declared with the `var` keyword:

```
let x = 5;
let y = 6;
let z = x + y;
```

In this example, `x`, `y`, and `z`, are variables, declared with the `let` keyword:

In this example, `x`, `y`, and `z`, are undeclared variables:

```
x = 5;
y = 6;
z = x + y;
```

JavaScript Let

The `let` keyword was introduced in ES6 (2015).

Variables defined with `let` cannot be Redeclared.

Variables defined with `let` must be Declared before use.

Variables defined with `let` have Block Scope.

Cannot be Redeclared

Variables defined with `let` cannot be **redeclared**.

You cannot accidentally redeclare a variable.

With `let` you can not do this:

```
let x = "John Doe";

let x = 0;

// SyntaxError: 'x' has already been declared
```

With `var` you can:

```
var x = "John Doe";

var x = 0;
```

Variables declared with the `var` keyword can NOT have block scope.

Variables declared inside a { } block can be accessed from outside the block.

```
{
  var x = 2;
}
// x CAN be used here only with in the scope only
```

Redeclaring Variables

Redeclaring a variable using the `var` keyword can impose problems.

Redeclaring a variable inside a block will also redeclare the variable outside the block:

```
var x = 10;
// Here x is 10

{
var x = 2;
// Here x is 2
}

// Here x is 2
```

Redeclaring a variable using the `let` keyword can solve this problem.

Redeclaring a variable inside a block will not redeclare the variable outside the block:

```
let x = 10;
// Here x is 10

{
let x = 2;
// Here x is 2
}

// Here x is 10
```

With `let`, redeclaring a variable in the same block is NOT allowed:

```
{
let x = 2;    // Allowed
let x = 3;    // Not allowed
}
```

Redeclaring a variable with `let`, in another block, IS allowed:

```
let x = 2;    // Allowed

{
let x = 3;    // Allowed
}

{
let x = 4;     // Allowed
}
```

JavaScript Const

The `const` keyword was introduced in ES6 (2015).

Variables defined with `const` cannot be Redeclared.

Variables defined with `const` cannot be Reassigned.

Variables defined with `const` have Block Scope.

Cannot be Reassigned

A `const` variable cannot be reassigned:

```
const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
PI = PI + 10;   // This will also give an error
```

Must be Assigned

JavaScript `const` variables must be assigned a value when they are declared:

Correct : `const PI = 3.14159265359;`

Incorrect :

```
const PI;
PI = 3.14159265359;
```

Constant Arrays

You can change the elements of a constant array:

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];

// You can change an element:
cars[0] = "Toyota";

// You can add an element:
cars.push("Audi");
```

But you can NOT reassign the array:

```
const cars = ["Saab", "Volvo", "BMW"];

cars = ["Toyota", "Volvo", "Audi"];    // ERROR
```

Constant Objects

You can change the properties of a constant object:

```
// You can create a const object:
const car = {type:"Fiat", model:"500", color:"white"};

// You can change a property:
car.color = "red";

// You can add a property:
car.owner = "Johnson";
```

But you can NOT reassign the object:

```
const car = {type:"Fiat", model:"500", color:"white"};

car = {type:"Volvo", model:"EX60", color:"red"};    // ERROR
```

Block Scope

Declaring a variable with `const` is similar to `let` when it comes to **Block Scope**.

The x declared in the block, in this example, is not the same as the x declared outside the block:

```
const x = 10;
// Here x is 10

{
const x = 2;
// Here x is 2
}

// Here x is 10
```

Redeclaring

Redeclaring a JavaScript `var` variable is allowed anywhere in a program:

```
var x = 2;     // Allowed
var x = 3;     // Allowed
x = 4;         // Allowed
```

Redeclaring an existing `var` or `let` variable to `const`, in the same scope, is not allowed:

```
var x = 2;     // Allowed
const x = 2;   // Not allowed

{
let x = 2;     // Allowed
const x = 2;   // Not allowed
}

{
const x = 2;   // Allowed
const x = 2;   // Not allowed
}
```

Reassigning an existing `const` variable, in the same scope, is not allowed:

```
const x = 2;      // Allowed
x = 2;            // Not allowed
var x = 2;        // Not allowed
let x = 2;        // Not allowed
const x = 2;      // Not allowed

{
  const x = 2;    // Allowed
  x = 2;          // Not allowed
  var x = 2;      // Not allowed
  let x = 2;      // Not allowed
  const x = 2;    // Not allowed
}
```

Redeclaring a variable with `const`, in another scope, or in another block, is allowed:

```
const x = 2;       // Allowed

{
  const x = 3;    // Allowed
}

{
  const x = 4;    // Allowed
}
```

Const Hoisting

Variables defined with `var` are **hoisted** to the top and can be initialized at any time.

Meaning: You can use the variable before it is declared:

```
carName = "Volvo";
var carName;
```

JavaScript Operators

There are different types of JavaScript operators:

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- Logical Operators
- Conditional Operators
- Type Operators

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

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
**=	x **= y	x = x ** y

The **addition assignment** operator (`+=`) adds a value to a variable.

```
let x = 10;
x += 5;
```

Adding JavaScript Strings

The + operator can also be used to add (concatenate) strings.

```
let text1 = "John";
let text2 = "Doe";
let text3 = text1 + " " + text2;
```

The += assignment operator can also be used to add (concatenate) strings:

```
let text1 = "What a very ";
text1 += "nice day";
```

RESULT:

```
What a very nice day
```

Adding Strings and Numbers

Adding two numbers, will return the sum, but adding a number and a string will return a string:

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

The result of *x*, *y*, and *z* will be:

```
10
55
Hello5
```

JavaScript Comparison Operators

Operator	**Description**
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

JavaScript Logical Operators

Operator	Description
&&	logical and
||	logical or
!	logical not

JavaScript Type Operators

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

JavaScript Bitwise Operators

Bit operators work on 32 bits numbers.

Any numeric operand in the operation is converted into a 32 bit number. The result is converted back to a JavaScript number.

Operator	Description	Example	Same as	Result	Decimal
&	AND	5 & 1	0101 & 0001	0001	1
|	OR	5 | 1	0101 | 0001	0101	5
~	NOT	~ 5	~0101	1010	10
^	XOR	5 ^ 1	0101 ^ 0001	0100	4
<<	left shift	5 << 1	0101 << 1	1010	10
>>	right shift	5 >> 1	0101 >> 1	0010	2
>>>	unsigned right shift	5 >>> 1	0101 >>> 1	0010	2

Arithmetic Operations

A typical arithmetic operation operates on two numbers.

The two numbers can be literals:

```
let x = 100 + 50;
```

or variables:

```
let x = a + b;
```

or expressions:

```
let x = (100 + 50) * a;
```

Operators and Operands

The numbers (in an arithmetic operation) are called **operands**.

The operation (to be performed between the two operands) is defined by an **operator**.

Operand	Operator	Operand
100	+	50

Adding

The **addition** operator (`+`) adds numbers:

```
let x = 5;
let y = 2;
let z = x + y;
```

Subtracting

The **subtraction** operator (-) subtracts numbers.

```
let x = 5;
let y = 2;
let z = x - y;
```

Multiplying

The **multiplication** operator (*) multiplies numbers.

```
let x = 5;
let y = 2;
let z = x * y;
```

Dividing

The **division** operator (/) divides numbers.

```
let x = 5;
let y = 2;
let z = x / y;
```

Remainder

The **modulus** operator (%) returns the division remainder.

```
let x = 5;
let y = 2;
let z = x % y;
```

Incrementing

The **increment** operator (++) increments numbers.

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

Decrementing

The **decrement** operator (`--`) decrements numbers.

```
let x = 5;
x--;
let z = x;
```

Exponentiation

The **exponentiation** operator (`**`) raises the first operand to the power of the second operand.

```
let x = 5;
let z = x ** 2;
```

x ** y produces the same result as `Math.pow(x,y)`:

```
let x = 5;
let z = Math.pow(x,2);
```

Operator Precedence

Operator precedence describes the order in which operations are performed in an arithmetic expression.

```
let x = 100 + 50 * 3;
```

JavaScript Assignment Operators

Assignment operators assign values to JavaScript variables.

The = Operator

The `=` assignment operator assigns a value to a variable.

Simple Assignment

```
let x = 10;
```

The += Operator

The `+=` assignment operator adds a value to a variable.

Addition Assignment

```
 let x = 10;
x += 5;
```

The -= Operator

The `-=` assignment operator subtracts a value from a variable.

Subtraction Assignment

```
let x = 10;
x -= 5;
```

The *= Operator

The *= assignment operator multiplies a variable.

Multiplication Assignment

```
let x = 10;
x *= 5;
```

The /= Operator

The /= assignment divides a variable.

Division Assignment

```
let x = 10;
x /= 5;
```

The %= Operator

The %= assignment operator assigns a remainder to a variable.

Remainder Assignment

```
let x = 10;
x %= 5;
```

The <<= Operator

The <<= assignment operator left shifts a variable.

Left Shift Assignment

```
let x = -100;
x <<= 5;
```

The >>= Operator

The >>= assignment operator right shifts a variable (signed).

Right Shift Assignment

```
let x = -100;
x >>= 5;
```

The >>>= Operator

The >>>= assignment operator right shifts a variable (unsigned).

Unsigned Right Shift Assignment

```
let x = -100;
x >>>= 5;
```

The &= Operator

The &= assignment operator ANDs a variable.

Bitwise AND Assignment

```
let x = 10;
x &= 5;
```

The != Operator

The `!=` assignment operator ORs a variable.

Bitwise OR Assignment

```
let x = 10;
x != 5;
```

JavaScript Operator Precedence

Operator precedence describes the order in which operations are performed in an arithmetic expression.

Multiplication (`*`) and division (`/`) have higher **precedence** than addition (`+`) and subtraction (`-`).

As in traditional mathematics, multiplication is done first:

```
let x = 100 + 50 * 3;
```

As in traditional mathematics, the precedence can be changed by parentheses.

When using parentheses, operations inside the parentheses are computed first:

```
let x = (100 + 50) * 3;
```

Operations with the same precedence (like * and /) are computed from left to right:

```
let x = 100 / 50 * 3;
```

JavaScript Data Types

JavaScript variables can hold different data types: numbers, strings, objects and more:

```
let length = 16;                               // Number
let lastName = "Johnson";                      // String
let x = {firstName:"John", lastName:"Doe"};    // Object
```

The Concept of Data Types

In programming, data types is an important concept.

To be able to operate on variables, it is important to know something about the type.

Without data types, a computer cannot safely solve this:

```
let x = 16 + "Volvo";
```

Does it make any sense to add "Volvo" to sixteen? Will it produce an error or will it produce a result?

JavaScript will treat the example above as:

```
let x = "16" + "Volvo";
```

When adding a number and a string, JavaScript will treat the number as a string.

EXAMPLE :

`let x = 16 + "Volvo";` `O/P -` 16Volvo

```
let x = 16 + 4 + "Volvo";
```

Result:

```
20Volvo
let x = "Volvo" + 16 + 4;
```

Result:

```
Volvo164
```

In the first example, JavaScript treats 16 and 4 as numbers, until it reaches "Volvo".

In the second example, since the first operand is a string, all operands are treated as strings.

JavaScript Types are Dynamic

JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

```
let x;           // Now x is undefined
x = 5;           // Now x is a Number
x = "John";      // Now x is a String
```

JavaScript Strings

A string (or a text string) is a series of characters like "John Doe".

Strings are written with quotes. You can use single or double quotes:

```
let carName1 = "Volvo XC60";   // Using double quotes
let carName2 = 'Volvo XC60';   // Using single quotes
```

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

```
let answer1 = "It's alright";       // Single quote inside double quotes
let answer2 = "He is 'Johnny'"; // Single quotes inside double quotes
let answer3 = 'He is "Johnny"';    // Double quotes inside single quotes
```

JavaScript Numbers

JavaScript has only one type of numbers.

Numbers can be written with, or without decimals:

```
let x1 = 34.00;     // Written with decimals

let x2 = 34;        // Written without decimals
```

Extra large or extra small numbers can be written with scientific (exponential) notation:

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

JavaScript Booleans

Booleans can only have two values: `true` or `false`.

```
let x = 5;
let y = 5;
let z = 6;
(x == y)       // Returns true
(x == z)       // Returns false
```

JavaScript Arrays

JavaScript arrays are written with square brackets.

Array items are separated by commas.

The following code declares (creates) an array called `cars`, containing three items (car names):

```
const cars = ["Saab", "Volvo", "BMW"];
```

JavaScript Objects

JavaScript objects are written with curly braces `{}`.

Object properties are written as name:value pairs, separated by commas.

```
const person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"};
```

The typeof Operator

You can use the JavaScript `typeof` operator to find the type of a JavaScript variable.

The `typeof` operator returns the type of a variable or an expression:

```
typeof ""             // Returns "string"
typeof "John"         // Returns "string"
typeof "John Doe"     // Returns "string"

typeof 0              // Returns "number"
typeof 314            // Returns "number"
typeof 3.14           // Returns "number"
typeof (3)            // Returns "number"
typeof (3 + 4)        // Returns "number"
```

Undefined

In JavaScript, a variable without a value, has the value `undefined`. The type is also `undefined`.

```
let car;    // Value is undefined, type is undefined
```

Any variable can be emptied, by setting the value to `undefined`. The type will also be `undefined`.

```
car = undefined;    // Value is undefined, type is undefined
```

Empty Values

An empty value has nothing to do with `undefined`.

An empty string has both a legal value and a type.

```
let car = "";    // The value is "", the typeof is "string"
```

JavaScript Functions

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

```
// Function to compute the product of p1 and p2
function myFunction(p1, p2) {
  return p1 * p2;
}
```

JavaScript Function Syntax

A JavaScript function is defined with the `function` keyword, followed by a **name**, followed by parentheses **()**.

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:
(*parameter1, parameter2, ...*)

The code to be executed, by the function, is placed inside curly brackets: **{}**

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

Function **parameters** are listed inside the parentheses () in the function definition.

Function **arguments** are the **values** received by the function when it is invoked.

Inside the function, the arguments (the parameters) behave as local variables.

Function Invocation

The code inside the function will execute when "something" **invokes** (calls) the function:

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self invoked)

Function Return

When JavaScript reaches a `return` statement, the function will stop executing.

If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

Functions often compute a **return value**. The return value is "returned" back to the "caller":

Example

Calculate the product of two numbers, and return the result:

```
let x = myFunction(4, 3);   // Function is called, return value will end
up in x

function myFunction(a, b) {
  return a * b;             // Function returns the product of a and b
}
```

The result in x will be:

```
12
```

Why Functions?

You can reuse code: Define the code once, and use it many times.

You can use the same code many times with different arguments, to produce different results.

Convert Fahrenheit to Celsius:

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
document.getElementById("demo").innerHTML = toCelsius(77);
```

The () Operator Invokes the Function

Using the example above, `toCelsius` refers to the function object, and `toCelsius()` refers to the function result.

Accessing a function without () will return the function object instead of the function result.

```
function toCelsius(fahrenheit) {
  return (5/9) * (fahrenheit-32);
}
document.getElementById("demo").innerHTML = toCelsius;
```

Functions Used as Variable Values

Functions can be used the same way as you use variables, in all types of formulas, assignments, and calculations.

Instead of using a variable to store the return value of a function:

```
let x = toCelsius(77);
let text = "The temperature is " + x + " Celsius";
```

You can use the function directly, as a variable value:

```
let text = "The temperature is " + toCelsius(77) + " Celsius";
```

Local Variables

Variables declared within a JavaScript function, become **LOCAL** to the function.

Local variables can only be accessed from within the function.

```
// code here can NOT use carName

function myFunction() {
  let carName = "Volvo";
  // code here CAN use carName
}

// code here can NOT use carName
```

JavaScript Objects

Real Life Objects, Properties, and Methods

In real life, a car is an **object**.

A car has **properties** like weight and color, and **methods** like start and stop:

Object	Properties	Methods

	car.name = Fiat
	car.model = 500
	car.weight = 850kg
	car.color = white
		car.start()
		car.drive()
		car.brake()
		car.stop()

All cars have the same **properties**, but the property **values** differ from car to car.

All cars have the same **methods**, but the methods are performed **at different times**.

Objects are variables too. But objects can contain many values.

This code assigns **many values** (Fiat, 500, white) to a **variable** named car:

```
const car = {type:"Fiat", model:"500", color:"white"};
```

Object Definition

You define (and create) a JavaScript object with an object literal:

```
const person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"};
```

Object Properties

The **name:values** pairs in JavaScript objects are called **properties**:

Property	**Property Value**
firstName	John
lastName	Doe
age	50
eyeColor	blue

JavaScript Events

HTML events are **"things"** that happen to HTML elements.

When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.

HTML Events

An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Often, when events happen, you may want to do something.

JavaScript lets you execute code when events are detected.

HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

With single quotes:

```
<element event='some JavaScript'>
```

With double quotes:

```
<element event="some JavaScript">
```

In the following example, an `onclick` attribute (with code), is added to a `<button>` element:

```
<button onclick="document.getElementById('demo').innerHTML = Date()">The
time is?</button>
```

In the example above, the JavaScript code changes the content of the element with id="demo".

```
<button onclick="this.innerHTML = Date()">The time is?</button>
```

the code changes the content of its own element (using `this.innerHTML`):

Common HTML Events

Here is a list of some common HTML events:

Event	**Description**
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

JavaScript Strings

JavaScript strings are for storing and manipulating text.

A JavaScript string is zero or more characters written inside quotes.

```
let text = "John Doe";
```

You can use single or double quotes:

```
let carName1 = "Volvo XC60";  // Double quotes
let carName2 = 'Volvo XC60';  // Single quotes
```

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

```
let answer1 = "It's alright";
let answer2 = "He is called 'Johnny'";
let answer3 = 'He is called "Johnny"';
```

String Length

To find the length of a string, use the built-in `length` property:

```
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
let length = text.length;
```

Escape Character

Because strings must be written within quotes, JavaScript will misunderstand this string:

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

The string will be chopped to "We are the so-called ".

The solution to avoid this problem, is to use the **backslash escape character**.

The backslash (`\`) escape character turns special characters into string characters:

Code	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

The sequence `\"` inserts a double quote in a string:

Example

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

The sequence `\'` inserts a single quote in a string:

Example

```
let text= 'It\'s alright.';
```

The sequence `\\` inserts a backslash in a string:

Example

```
let text = "The character \\ is called backslash.";
```

Six other escape sequences are valid in JavaScript:

Code	Result
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Horizontal Tabulator
\v	Vertical Tabulator

JavaScript if, else, and else if

Conditional statements are used to perform different actions based on different conditions.

Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

The if Statement

Use the `if` statement to specify a block of JavaScript code to be executed if a condition is true.

Syntax

```
if (condition) {
  //  block of code to be executed if the condition is true
}
```

The else Statement

Use the `else` statement to specify a block of code to be executed if the condition is false.

```
if (condition) {
  //  block of code to be executed if the condition is true
} else {
  //  block of code to be executed if the condition is false
}
```

```
<script>
const hour = new Date().getHours();
let greeting;

if (hour < 18) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
document.getElementById("demo").innerHTML = greeting;
</script>
```

The else if Statement

Use the `else if` statement to specify a new condition if the first condition is false.

Syntax

```
if (condition1) {
  //  block of code to be executed if condition1 is true
} else if (condition2) {
  //  block of code to be executed if the condition1 is false and
condition2 is true
} else {
  //  block of code to be executed if the condition1 is false and
condition2 is false
}
```

```
<script>

const time = new Date().getHours();

let greeting;

if (time < 10) {

  greeting = "Good morning";

} else if (time < 20) {

  greeting = "Good day";

} else {

  greeting = "Good evening";

}

document.getElementById("demo").innerHTML = greeting;

</script>
```

JavaScript Switch Statement

The `switch` statement is used to perform different actions based on different conditions.

The JavaScript Switch Statement

Use the `switch` statement to select one of many code blocks to be executed.

Syntax

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

This is how it works:

- The switch expression is evaluated once.
- The value of the expression is compared with the values of each case.
- If there is a match, the associated block of code is executed.
- If there is no match, the default code block is executed.

Example

The `getDay()` method returns the weekday as a number between 0 and 6.

(Sunday=0, Monday=1, Tuesday=2 ..)

This example uses the weekday number to calculate the weekday name:

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
     day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
}
```

The result of day will be:

```
Thursday
```

The break Keyword

When JavaScript reaches a `break` keyword, it breaks out of the switch block.

This will stop the execution inside the switch block.

It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.

The default Keyword

The `default` keyword specifies the code to run if there is no case match:

Example

The `getDay()` method returns the weekday as a number between 0 and 6.

If today is neither Saturday (6) nor Sunday (0), write a default message:

```
switch (new Date().getDay()) {
  case 6:
    text = "Today is Saturday";
    break;
  case 0:
    text = "Today is Sunday";
    break;
  default:
    text = "Looking forward to the Weekend";
}
```

The result of text will be:

```
Looking forward to the Weekend
```

JavaScript For Loop

Loops can execute a block of code a number of times.

JavaScript Loops

Loops are handy, if you want to run the same code over and over again, each time with a different value.

Often this is the case when working with arrays:

```
for (let i = 0; i < cars.length; i++) {
  text += cars[i] + "<br>";
}
```

Different Kinds of Loops

JavaScript supports different kinds of loops:

- `for` - loops through a block of code a number of times
- `for/in` - loops through the properties of an object
- `for/of` - loops through the values of an iterable object
- `while` - loops through a block of code while a specified condition is true
- `do/while` - also loops through a block of code while a specified condition is true

- ## The For Loop
- The `for` statement creates a loop with 3 optional expressions:

-
  ```
  for (expression 1; expression 2; expression 3) {
    // code block to be executed
  }
  ```
- **Expression 1** is executed (one time) before the execution of the code block.
- **Expression 2** defines the condition for executing the code block.
- **Expression 3** is executed (every time) after the code block has been executed.

JavaScript For In

The For In Loop

The JavaScript `for in` statement loops through the properties of an Object:

Syntax

```
for (key in object) {
  // code block to be executed
}
```

```
<script>

const person = {fname:"John", lname:"Doe", age:25};


let txt = "";

for (let x in person) {

  txt += person[x] + " ";

}

document.getElementById("demo").innerHTML = txt;

</script>
```

Example Explained

- The **for in** loop iterates over a **person** object
- Each iteration returns a **key** (x)
- The key is used to access the **value** of the key
- The value of the key is **person[x]**

For In Over Arrays

The JavaScript `for in` statement can also loop over the properties of an Array:

Syntax

```
for (variable in array) {
  code
}
```

Example

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
for (let x in numbers) {
  txt += numbers[x];
}
```

Array.forEach()

The `forEach()` method calls a function (a callback function) once for each array element.

Example

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
numbers.forEach(myFunction);

function myFunction(value, index, array) {
  txt += value;
}
```

Note that the function takes 3 arguments:

- The item value
- The item index
- The array itself

The example above uses only the value parameter. It can be rewritten to:

Example

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
numbers.forEach(myFunction);

function myFunction(value) {
  txt += value;
}
```

JavaScript For Of

The For Of Loop

The JavaScript `for of` statement loops through the values of an iterable object.

It lets you loop over iterable data structures such as Arrays, Strings, Maps, NodeLists, and more:

Syntax

```
for (variable of iterable) {
  // code block to be executed
}
```

variable - For every iteration the value of the next property is assigned to the variable. *Variable* can be declared with `const`, `let`, or `var`.

iterable - An object that has iterable properties.

Browser Support

For/of was added to JavaScript in 2015 (ES6)

Safari 7 was the first browser to support for of:

Looping over an Array

Example

```
const cars = ["BMW", "Volvo", "Mini"];

let text = "";
for (let x of cars) {
  text += x;
}
```

Looping over a String

Example

```
let language = "JavaScript";

let text = "";
for (let x of language) {
text += x;
}
```

JavaScript While Loop

Loops can execute a block of code as long as a specified condition is true.

The While Loop

The `while` loop loops through a block of code as long as a specified condition is true.

Syntax

```
while (condition) {
  // code block to be executed
}
```

The Do While Loop

The `do while` loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {
  // code block to be executed
}
while (condition);
```

Comparing For and While

If you have read the previous chapter, about the for loop, you will discover that a while loop is much the same as a for loop, with statement 1 and statement 3 omitted.

The loop in this example uses a `for` loop to collect the car names from the cars array:

Example

```
const cars = ["BMW", "Volvo", "Saab", "Ford"];
let i = 0;
let text = "";

for (;cars[i];) {
  text += cars[i];
  i++;
}
```

The loop in this example uses a `while` loop to collect the car names from the cars array:

Example

```
const cars = ["BMW", "Volvo", "Saab", "Ford"];
let i = 0;
let text = "";
```

```
while (cars[i]) {
  text += cars[i];
  i++;
}
```

JavaScript Type Conversion

- Converting Strings to Numbers
- Converting Numbers to Strings
- Converting Dates to Numbers
- Converting Numbers to Dates
- Converting Booleans to Numbers
- Converting Numbers to Booleans

JavaScript Type Conversion

JavaScript variables can be converted to a new variable and another data type:

- By the use of a JavaScript function
- **Automatically** by JavaScript itself

Converting Strings to Numbers

The global method `Number()` converts a variable (or a value) into a number.

A numeric string (like "3.14") converts to a number (like 3.14).

An empty string (like "") converts to 0.

A non numeric string (like "John") converts to `NaN` (Not a Number).

These will convert:

```
Number("3.14")
Number(Math.PI)
Number(" ")
Number("")
```

These will not convert:

```
Number("99 88")
Number("John")
```

Number Methods

In the chapter Number Methods, you will find more methods that can be used to convert strings to numbers:

Method	**Description**
Number()	Returns a number, converted from its argument
parseFloat()	Parses a string and returns a floating point number
parseInt()	Parses a string and returns an integer

The Unary + Operator

The **unary + operator** can be used to convert a variable to a number:

Example

```
let y = "5";       // y is a string
let x = + y;       // x is a number
```

If the variable cannot be converted, it will still become a number, but with the value NaN (Not a Number):

Example

```
let y = "John";   // y is a string
let x = + y;      // x is a number (NaN)
```

Converting Numbers to Strings

The global method String() can convert numbers to strings.

It can be used on any type of numbers, literals, variables, or expressions:

Example

```
String(x)         // returns a string from a number variable x
String(123)       // returns a string from a number literal 123
String(100 + 23)  // returns a string from a number from an expressio
```

The Number method toString() does the same.

Example

```
x.toString()
(123).toString()
(100 + 23).toString()
```

Converting Dates to Numbers

The global method Number() can be used to convert dates to numbers.

```
d = new Date();
Number(d)          // returns 1404568027739
```

The date method getTime() does the same.

```
d = new Date();
d.getTime()        // returns 1404568027739
```

Converting Dates to Strings

The global method `String()` can convert dates to strings.

```
String(Date())  // returns "Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe
Daylight Time)"
```

The Date method `toString()` does the same.

Example

```
Date().toString()  // returns "Thu Jul 17 2014 15:38:19 GMT+0200 (W.
Europe Daylight Time)"
```

Method	**Description**
getDate()	Get the day as a number (1-31)
getDay()	Get the weekday a number (0-6)
getFullYear()	Get the four digit year (yyyy)
getHours()	Get the hour (0-23)
getMilliseconds()	Get the milliseconds (0-999)
getMinutes()	Get the minutes (0-59)

getMonth()	Get the month (0-11)
getSeconds()	Get the seconds (0-59)
getTime()	Get the time (milliseconds since January 1, 1970)

Converting Booleans to Numbers

The global method `Number()` can also convert booleans to numbers.

```
Number(false)     // returns 0
Number(true)      // returns 1
```

Converting Booleans to Strings

The global method `String()` can convert booleans to strings.

```
String(false)      // returns "false"
String(true)       // returns "true"
```

The Boolean method `toString()` does the same.

```
false.toString()   // returns "false"
true.toString()    // returns "true"
```

Automatic Type Conversion

When JavaScript tries to operate on a "wrong" data type, it will try to convert the value to a "right" type.

The result is not always what you expect:

```
5 + null    // returns 5         because null is converted to 0
"5" + null  // returns "5null"   because null is converted to "null"
"5" + 2     // returns "52"      because 2 is converted to "2"
"5" - 2     // returns 3         because "5" is converted to 5
```

```
"5" * "2"   // returns 10        because "5" and "2" are converted to 5
and 2
```

Automatic String Conversion

JavaScript automatically calls the variable's `toString()` function when you try to "output" an object or a variable:

```
document.getElementById("demo").innerHTML = myVar;

// if myVar = {name:"Fjohn"}  // toString converts to "[object Object]"
// if myVar = [1,2,3,4]       // toString converts to "1,2,3,4"
// if myVar = new Date()      // toString converts to "Fri Jul 18 2014
09:08:55 GMT+0200"
```

Numbers and booleans are also converted, but this is not very visible:

```
// if myVar = 123             // toString converts to "123"
// if myVar = true            // toString converts to "true"
// if myVar = false           // toString converts to "false"
```

JavaScript Errors

Throw, and Try...Catch...Finally

The `try` statement defines a code block to run (to try).

The `catch` statement defines a code block to handle any error.

The `finally` statement defines a code block to run regardless of the result.

The `throw` statement defines a custom error.

JavaScript try and catch

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The JavaScript statements `try` and `catch` come in pairs:

```
try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
```

The throw Statement

The `throw` statement allows you to create a custom error.

Technically you can **throw an exception (throw an error)**.

The exception can be a JavaScript `String`, a `Number`, a `Boolean` or an `Object`:

```
throw "Too big";    // throw a text
throw 500;          // throw a number
```

If you use `throw` together with `try` and `catch`, you can control program flow and generate custom error messages.

The Error Object

JavaScript has a built in error object that provides error information when an error occurs.

The error object provides two useful properties: name and message.

Error Object Properties

Property	**Description**
name	Sets or returns an error name
message	Sets or returns an error message (a string)

Error Name Values

Six different values can be returned by the error name property:

Error Name	**Description**
EvalError	An error has occurred in the eval() function
RangeError	A number "out of range" has occurred
ReferenceError	An illegal reference has occurred
SyntaxError	A syntax error has occurred
TypeError	A type error has occurred

URIError	An error in encodeURI() has occurred

Eval Error

An EvalError indicates an error in the eval() function.

Range Error

A RangeError is thrown if you use a number that is outside the range of legal values.

For example: You cannot set the number of significant digits of a number to 500.

```
let num = 1;
try {
  num.toPrecision(500);   // A number cannot have 500 significant digits
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

Reference Error

A ReferenceError is thrown if you use (reference) a variable that has not been declared:

```
let x = 5;
try {
  x = y + 1;   // y cannot be used (referenced)
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

Syntax Error

A `SyntaxError` is thrown if you try to evaluate code with a syntax error.

```
try {
  eval("alert('Hello)");   // Missing ' will produce an error
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

Type Error

A `TypeError` is thrown if you use a value that is outside the range of expected types:

```
let num = 1;
try {
  num.toUpperCase();   // You cannot convert a number to upper case
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

URI (Uniform Resource Identifier) Error

A `URIError` is thrown if you use illegal characters in a URI function:

```
try {
  decodeURI("%%%");   // You cannot URI decode percent signs
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

JavaScript Scope

Scope determines the accessibility (visibility) of variables.

JavaScript has 3 types of scope:

- Block scope
- Function scope
- Global scope

Block Scope

Before ES6 (2015), JavaScript had only **Global Scope** and **Function Scope**.

ES6 introduced two important new JavaScript keywords: `let` and `const`.

These two keywords provide **Block Scope** in JavaScript.

Variables declared inside a { } block cannot be accessed from outside the block:

```
{
  let x = 2;
}
// x can NOT be used her
```

Variables declared with the `var` keyword can NOT have block scope.

Variables declared inside a { } block can be accessed from outside the block.

```
{
  var x = 2;
}
// x CAN be used her
```

Local Scope

Variables declared within a JavaScript function, become **LOCAL** to the function.

```
// code here can NOT use carName

function myFunction() {
  let carName = "Volvo";
  // code here CAN use carName
}

// code here can NOT use carName
```

Function Scope

JavaScript has function scope: Each function creates a new scope.

Variables defined inside a function are not accessible (visible) from outside the function.

Variables declared with `var`, `let` and `const` are quite similar when declared inside a function.

They all have **Function Scope**:

```
function myFunction() {
  var carName = "Volvo";   // Function Scope
}
```

```
function myFunction() {
  let carName = "Volvo";   // Function Scope
}
```

```
function myFunction() {
  const carName = "Volvo";   // Function Scope
}
```

Global JavaScript Variables

A variable declared outside a function, becomes **GLOBAL**.

```
let carName = "Volvo";
// code here can use carName

function myFunction() {
// code here can also use carName }
```

Global Scope

Variables declared **Globally** (outside any function) have **Global Scope**.

Global variables can be accessed from anywhere in a JavaScript program.

Variables declared with `var`, `let` and `const` are quite similar when declared outside a block.

They all have **Global Scope**:

```
var x = 2;       // Global scope

let x = 2;       // Global scope

const x = 2;       // Global scope
```

JavaScript Variables

In JavaScript, objects and functions are also variables.

Automatically Global

If you assign a value to a variable that has not been declared, it will automatically become a **GLOBAL** variable.

This code example will declare a global variable `carName`, even if the value is assigned inside a function.

```
myFunction();

// code here can use carName

function myFunction() {
  carName = "Volvo";
}
```

JavaScript Hoisting

Hoisting is JavaScript's default behavior of moving declarations to the top.

JavaScript Declarations are Hoisted

In JavaScript, a variable can be declared after it has been used.

In other words; a variable can be used before it has been declared.

Example 1 gives the same result as **Example 2**:

Example 1

```
x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                     // Display x in the element

var x; // Declare x
```

Example 2

```
var x; // Declare x
x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                     // Display x in the element
```

JavaScript Initializations are Not Hoisted

JavaScript only hoists declarations, not initializations.

Example 1 does **not** give the same result as **Example 2**:

Example 1

```
var x = 5; // Initialize x
var y = 7; // Initialize y

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y
```

Example 2

```
var x = 5; // Initialize x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x + " " + y;           // Display x and y

var y = 7; // Initialize y
```

Does it make sense that y is undefined in the last example?

This is because only the declaration (var y), not the initialization (=7) is hoisted to the top.

Because of hoisting, y has been declared before it is used, but because initializations are not hoisted, the value of y is undefined.

Example 2 is the same as writing:

Declare Your Variables At the Top !

Hoisting is (to many developers) an unknown or overlooked behavior of JavaScript.

If a developer doesn't understand hoisting, programs may contain bugs (errors).

To avoid bugs, always declare all variables at the beginning of every scope.

Since this is how JavaScript interprets the code, it is always a good rule.

JavaScript JSON

JSON is a format for storing and transporting data.

JSON is often used when data is sent from a server to a web page.

What is JSON?

- JSON stands for **J**ava**S**cript **O**bject **N**otation
- JSON is a lightweight data interchange format
- JSON is language independent *****
- JSON is "self-describing" and easy to understand

* The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only. Code for reading and generating JSON data can be written in any programming language.

JSON Example

This JSON syntax defines an employees object: an array of 3 employee records (objects):

JSON Example

```
{
"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]
}
```

JSON Syntax Rules

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

JSON Data - A Name and a Value

JSON data is written as name/value pairs, just like JavaScript object properties.

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

```
"firstName":"John"
```

JSON names require double quotes. JavaScript names do not.

JSON Objects

JSON objects are written inside curly braces.

Just like in JavaScript, objects can contain multiple name/value pairs:

```
{"firstName":"John", "lastName":"Doe"}
```

JSON Arrays

JSON arrays are written inside square brackets.

Just like in JavaScript, an array can contain objects:

```
"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]
```

In the example above, the object "employees" is an array. It contains three objects.

Each object is a record of a person (with a first name and a last name).

Converting a JSON Text to a JavaScript Object

A common use of JSON is to read data from a web server, and display the data in a web page.

For simplicity, this can be demonstrated using a string as input.

First, create a JavaScript string containing JSON syntax:

```
let text = '{ "employees" : [' +
'{ "firstName":"John" , "lastName":"Doe" },' +
'{ "firstName":"Anna" , "lastName":"Smith" },' +
'{ "firstName":"Peter" , "lastName":"Jones" } ]}';
```

Then, use the JavaScript built-in function `JSON.parse()` to convert the string into a JavaScript object:

```
const obj = JSON.parse(text);
```

Finally, use the new JavaScript object in your page:

```
<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML =
obj.employees[1].firstName + " " + obj.employees[1].lastName;
</script>
```