On every key press we have to validate first name, last name& middle name.

When we perform an action on html page an event while be generated, that event can handled by using js.

|  |  |
| --- | --- |
| **Window** | **Document** |
| Window object represents a browser window | Document object represents the html document |
| Window object is parent to the document object | Document object is child to window object |
| When we start a browser then the Window object is created by browser automatically | When you load html document from the browser then the document object is created by browser immediately |
| Methods= alert , open, close are window methods | Methods=getelementbyid(), getelementbyTagname() and getelementbyclass() |
|  |  |
|  |  |
|  |  |

Note: window.document and document refers same.

**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":

Syntax:

document.getElementById('demo').innerHTML = 'Hello JavaScript';

**JavaScript Can Change HTML Attribute Values**

With the help of the JavaScript we can change the attribute values

**JavaScript can hide Html Elements:**

with the help of the JavaScript we can hide the display style.

document.getElementById ("demo").style.display = "none";

**JavaScript Can Show HTML Elements**

with the help of the JavaScript we can show the display style.

document.getElementById("demo").style.display = "block";

**Did You Know?**

JavaScript and [**Java**](https://www.w3schools.com/java/default.asp)are completely different languages, both in concept and design.

JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997.

ECMA-262 is the official name of the standard. ECMAScript is the official name of the language.

**JavaScript Where To**

**The <script> Tag**

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

<script>  
document.getElementById("demo").innerHTML = "My First JavaScript";  
</script>

Old JavaScript examples may use a type attribute: <script type="text/JavaScript">.  
The type attribute is not required. JavaScript is the default scripting language in HTML.

***JavaScript Functions and Events***

A JavaScript function is a block of JavaScript code, that can be executed when "called" for.

For example, a function can be called when an event occurs, like when the user clicks a button.

**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.

<!DOCTYPE html>  
<html>

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

<h1>A Web Page</h1>  
<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>  
  
<h1>A Web Page</h1>  
<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>

Placing scripts at the bottom of the <body> element improves the display speed, because script interpretation slows down the display.

External JavaScript

Scripts can also be placed in external files:

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

External scripts are practical when the same code is used in many different web pages.

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.

The script will behave as if it was located exactly where the <script> tag is located.

External scripts cannot contain <script> tags.

**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

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

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

<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>

Changing the innerHTML property of an HTML element is a common way to display data in HTML.

**Using document.write()**

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

<!DOCTYPE html>  
<html>  
<body>  
  
<h1>My First Web Page</h1>  
<p>My first paragraph.</p>  
  
<script>  
document.write(5 + 6);  
</script>  
  
</body>  
</html>

Using document.write() after an HTML document is loaded, will delete all existing HTML:

<!DOCTYPE html>  
<html>  
<body>  
<h1>My First Web Page</h1>  
<p>My first paragraph.</p>  
<button type="button" onclick="document.write(5 + 6)">Try it</button>  
</body>  
</html>

**Using window.alert()**

You can use an alert box to display data:

<!DOCTYPE html>  
<html>  
 <body>  
 <h1>My First Web Page</h1>  
 <p>My first paragraph.</p>  
 <script>  
 window.alert(5 + 6);  
 </script>  
 </body>  
</html>

You can skip the window keyword.

In JavaScript, the window object is the global scope object, that means that variables, properties, and methods by default belong to the window object. This also means that specifying the window keyword is optional

Example

<script>

Alert(5+6)

</script>

**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.

Example

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

**JavaScript Statements**

JavaScript syntax is the set of rules, how JavaScript programs are constructed:

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

// 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

10.50

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.

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

Let x;

X = 6

**JavaScript Operators**

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

(5+6)\*10

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:

5\*10

Expressions can also contain variable values:

x \* 10

The values can be of various types, such as numbers and strings.

For example, "John" + " " + "Doe", evaluates to "John Doe":

"John" + " " + "Doe"

**JavaScript Keywords**

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

The let keyword tells the browser to create variables:

Let x, y;

X = 5 + 6;

Y = x \* 10;

The var keyword also tells the browser to create variables:

var x, y;  
x = 5 + 6;  
y = x \* 10;

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**

Identifiers are names.

In JavaScript, identifiers are used to name variables (and keywords, and functions, and labels).

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

In JavaScript, the first character must be a letter, or an underscore (\_), or a dollar sign ($).

Subsequent characters may be letters, digits, underscores, or dollar signs.

**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";

JavaScript does not interpret LET or Let as the keyword let.

**JavaScript and Camel Case**

Historically, programmers have used different ways of joining multiple words into one variable name:

**Hyphens:**

first-name, last-name, master-card, inter-city.

Hyphens are not allowed in JavaScript. They are reserved for subtractions.

**Underscore:**

first\_name, last\_name, master\_card, inter\_city.

**Upper Camel Case (Pascal Case):**

FirstName, LastName, MasterCard, InterCity.

**Lower Camel Case:**

JavaScript programmers tend to use camel case that starts with a lowercase letter:

firstName, lastName, masterCard, interCity.

JavaScript Character Set

JavaScript uses the **Unicode** character set. Unicode covers (almost) all the characters, punctuations, and symbols in the world.

**JavaScript Comments**

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

It can also be used to prevent execution, when testing alternative code.

**Single Line Comments**

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

// Change heading:  
document.getElementById("myH").innerHTML = "My First Page";  
  
// Change paragraph:  
document.getElementById("myP").innerHTML = "My first paragraph.";

This example uses a single line comment at the end of each line to explain the code:

let x = 5;      // Declare x, give it the value of 5  
let y = x + 2;  // Declare y, give it the value of x + 2

Multi-line Comments

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

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

This example uses a multi-line comment (a comment block) to explain the code:

\*  
The code below will change  
the heading with id = "myH"  
and the paragraph with id = "myP"  
in my web page:  
\*/  
document.getElementById("myH").innerHTML = "My First Page";  
document.getElementById("myP").innerHTML = "My first paragraph.";

Using Comments to Prevent Execution

Using comments to prevent execution of code is suitable for code testing.

Adding // in front of a code line changes the code lines from an executable line to a comment.

This example uses // to prevent execution of one of the code lines:

//document.getElementById("myH").innerHTML = "My First Page";  
document.getElementById("myP").innerHTML = "My first paragraph.";

**JavaScript Variables**

There are 3 ways to declare a JavaScript variable:

* Using var
* Using let
* Using const

This chapter uses var.

The let and const keywords are explained in the next chapters.

There are 3 ways to declare a JavaScript variable:

* Using var
* Using let
* Using const

This chapter uses var.

The let and const keywords are explained in the next chapters.

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

from the example above, you can expect:

* X stores the value 11
* Y stores the value 6
* Z stores the values 11

**Must like algebra**

In example , price1, price2, and total, are variables:

Var price1 = 5;

Var price2 = 6;

Var total = price1 + price2;

In programming, just like in algebra, we use variables (like price1) to hold values.

In programming, just like in algebra, we use variables in expressions (total = price1 + price2).

From the example above, you can calculate the total to be 11.

JavaScript variables are containers for strong data values.

JavaScript Data Types

JavaScript variables can hold numbers like 100 and text values like "John Doe".

In programming, text values are called text strings.

It can handle many types of data, but for now, just think of numbers and strings.

Strings are written inside double or single quotes. Numbers are written without quotes.

If you put a number in quotes, it will be treated as a text string.

Example

var pi = 3.14;  
var person = "John Doe";  
var answer = 'Yes I am!';

Declaring (Creating) JavaScript Variables:

Creating a variable in JavaScript is called “declaring” a variable.

You declare a JavaScript variable with the keywords:

Var carName;

After the declaration, the variable has no value (technically it has the value of undefined). To **assign** a value to the variable, use the equal sign:

carName = “Volvo”;

You can also assign a value to the variable when you declare it:

var carName = "Volvo";

In the example below, we create a variable called carName and assign the value "Volvo" to it.

Then we "output" the value inside an HTML paragraph with id="demo":

<p id=”demo”></P>

<script>  
var carName = "Volvo";  
document.getElementById("demo").innerHTML = carName;  
</script>

**One Statement, Many Variables**

You can declare many variables in one statement.

Start the statement with var and separate the variables by **comma**:

var person = "John Doe", carName = "Volvo", price = 200;

**A declaration can span multiple lines:**

var person = "John Doe",  
carName = "Volvo",  
price = 200;

**Value = undefined**

In computer programs, variables are often declared without a value. The value can be something that has to be calculated, or something that will be provided later, like user input.

A variable declared without a value will have the value undefined.

The variable carName will have the value undefined after the execution of this statement:

Example

var carName;

**Re-Declaring JavaScript Variables**

If you re-declare a JavaScript variable, it will not lose its value.

The variable carName will still have the value "Volvo" after the execution of these statements:

Example

var carName = "Volvo";  
var carName;

**JavaScript Arithmetic**

As with algebra, you can do arithmetic with JavaScript variables, using operators like = and +:

Example

var x = 5 + 2 + 3;

You can also add strings, but strings will be concatenated:

Example

var x = "John" + " " + "Doe";

Example

var x = "5" + 2 + 3;

If you put a number in quotes, the rest of the numbers will be treated as strings, and concatenated.

Now try this:

Example

var x = 2 + 3 + "5";

**JavaScript Dollar Sign $**

Remember that JavaScript identifiers (names) must begin with:

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

Since JavaScript treats a dollar sign as a letter, identifiers containing $ are valid variable names:

Example

var $$$ = "Hello World";  
var $ = 2;  
var $myMoney = 5;

Using the dollar sign is not very common in JavaScript, but professional programmers often use it as an alias for the main function in a JavaScript library.

In the JavaScript library jQuery, for instance, the main function $ is used to select HTML elements. In jQuery $("p"); means "select all p elements".

**JavaScript Underscore (\_)**

Since JavaScript treats underscore as a letter, identifiers containing \_ are valid variable names:

Example

var \_lastName = "Johnson";  
var \_x = 2;  
var \_100 = 5;

Using the underscore is not very common in JavaScript, but a convention among professional programmers is to use it as an alias for "private (hidden)" variables.

**JavaScript Let**

The let keyword was introduced in [ES6 (2015)](https://www.w3schools.com/js/js_es6.asp).

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:

Example

let x = "John Doe";  
  
let x = 0;  
  
// SyntaxError: 'x' has already been declared

With var you can:

Example

var x = "John Doe";  
  
var x = 0;

**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:

Example

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

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

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

Example

{  
  var x = 2;  
}  
// x CAN be used here

**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:

Example

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:

Example

let x = 10;  
// Here x is 10  
  
{  
let x = 2;  
// Here x is 2  
}  
  
// Here x is 10

Redeclaring

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

Example

var x = 2;  
// Now x is 2  
  
var x = 3;  
// Now x is 3

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

Example

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

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

Example

let x = 2;    // Allowed  
  
{  
let x = 3;    // Allowed  
}  
  
{  
let x = 4;    // Allowed  
}

Let 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:

Example

This is OK:

carName = "Volvo";  
var carName;

Variables defined with let are also hoisted to the top of the block, but not initialized.

Meaning: Using a let variable before it is declared will result in a ReferenceError:

Example

carName = "Saab";  
let carName = "Volvo";

**JavaScript Const**

The const keyword was introduced in [ES6 (2015)](https://www.w3schools.com/js/js_es6.asp).

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:

Example

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;

When to use JavaScript const?

As a general rule, always declare a variable with const unless you know that the value will change.

Use const when you declare:

* A new Array
* A new Object
* A new Function
* A new RegExp

**Constant Objects and Arrays**

The keyword const is a little misleading.

It does not define a constant value. It defines a constant reference to a value.

Because of this you can NOT:

* Reassign a constant value
* Reassign a constant array
* Reassign a constant object

But you CAN:

* Change the elements of constant array
* Change the properties of constant object

**Constant Arrays**

You can change the elements of a constant array:

Example

// 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:

Example

const cars = ["Saab", "Volvo", "BMW"];  
  
cars = ["Toyota", "Volvo", "Audi"];    // ERROR

**Constant Objects**

You can change the properties of a constant object:

Example

// 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:

Example

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:

Example

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:

Example

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:

Example

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:

Example

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:

Example

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:

Example

This is OK:

carName = "Volvo";  
var carName;

|  |  |  |
| --- | --- | --- |
| variables | let | constant |
| It can be redeclared | It cannot be redeclared | It cannot be redeclared |
| It can be reassigned | It cannot be reassigned | It cannot be reassigned |
| It allow black scope | It | It allow black scope |
|  |  |  |
|  |  |  |

**JavaScript Operators**

Example

Assign values to variables and add them together:

let x = 5;         // assign the value 5 to x  
let y = 2;         // assign the value 2 to y  
let z = x + y;     // assign the value 7 to z (5 + 2)

The **assignment** operator (=) assigns a value to a variable.

Assignment

let x = 10;

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

Adding

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

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

Multiplying

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

**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.

Assignment

let x = 10;  
x += 5;

**JavaScript string operator**

The + operator can be used to add strings.

Example

Let text1 = “john”;

Let text2 = “Deo”;

Let text3 = text1 + “ “ + text2;

The result of text3 will be:

John Deo

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

Example

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

The result of txt1 will be:

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:

Example

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 Arithmetic**

**JavaScript Arithmetic Operators**

Arithmetic operators perform arithmetic on numbers (literals or variables).

A typical arithmetic operation operates on two numbers.

The two numbers can be literals:

Example

let x = 100 + 50;

or variables:

Example

let x = a + b;

or expressions:

Example

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:

Example

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

**Subtracting**

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

Example

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

Multiplying

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

Example

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

**Dividing**

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

Example

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

**Remainder**

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

Example

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

In arithmetic, the division of two integers produces a **quotient** and a **remainder**.

In mathematics, the result of a **modulo operation** is the **remainder** of an arithmetic division.

**Incrementing**

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

Example

let x = 5;  
x++;  
let z = x;

**Decrementing**

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

Example

let x = 5;  
x--;  
let z = x;

**Exponentiation**

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

Example

let x = 5;  
let z = x \*\* 2;          // result is 25

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

Example

let x = 5;  
let z = Math.pow(x,2);   // result is 25

**Operator Precedence**

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

Example

let x = 100 + 50 \* 3;

Is the result of example above the same as 150 \* 3, or is it the same as 100 + 150?

Is the addition or the multiplication done first?

As in traditional school mathematics, the multiplication is done first.

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

And (as in school mathematics) the precedence can be changed by using parentheses:

Example

let x = (100 + 50) \* 3;

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

When many operations have the same precedence (like addition and subtraction), they are computed from left to right:

Example

let x = 100 + 50 - 3;

**JavaScript Assignment**

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 |
| >>= | 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 |

Assignment Examples

The = assignment operator assigns a value to a variable.

Assignment

let x = 10;

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

Assignment

let x = 10;  
x += 5;

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

Assignment

let x = 10;  
x -= 5;

The \*= assignment operator multiplies a variable.

Assignment

let x = 10;  
x \*= 5;

The /= assignment divides a variable.

Assignment

let x = 10;  
x /= 5;

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

Assignment

let x = 10;  
x %= 5;

**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";

Example

let x = "Volvo" + 16;

JavaScript evaluates expressions from left to right. Different sequences can produce different results:

JavaScript:

let x = 16 + 4 + "Volvo";

Result:

20Volvo

JavaScript:

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:

Example

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

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:

Example

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:

Example

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

You will learn more about strings later in this tutorial.

JavaScript Numbers

JavaScript has only one type of numbers.

Numbers can be written with, or without decimals:

Example

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:

Example

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

**JavaScript Booleans**

Booleans can only have two values: true or false.

Example

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

Booleans are often used in conditional testing.

You will learn more about conditional testing later in this tutorial.

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):

Example

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

Array indexes are zero-based, which means the first item is [0], second is [1], and so on.

You will learn more about **arrays** later in this tutorial.

**JavaScript Objects**

JavaScript objects are written with curly braces {}.

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

Example

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

The object (person) in the example above has 4 properties: firstName, lastName, age, and eyeColor.

You will learn more about **objects** later in this tutorial.

**The typeof Operator**

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

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

Example

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

Example

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.

Example

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.

Example

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

An empty value has nothing to do with undefined.

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

Example

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).

Example

function myFunction(p1, p2) {  
  return p1 \* p2;   // The function returns the product of p1 and 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.

A Function is much the same as a Procedure or a Subroutine, in other programming languages.

**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.

Example

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.

Example

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.

Example

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

Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.

Local variables are created when a function starts, and deleted when the function is completed.

**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:

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**.

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**.

JavaScript Objects

You have already learned that JavaScript variables are containers for data values.

This code assigns a **simple value** (Fiat) to a **variable** named car:

let car = "Fiat";

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"};

The values are written as **name:value** pairs (name and value separated by a colon).

Object Properties

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

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

Accessing Object Properties

You can access object properties in two ways:

*objectName.propertyName*

or

*objectName["propertyName"]*

Example1

person.lastName;

Example2

person["lastName"];

JavaScript objects are containers for **named values**called properties.

Object Methods

Objects can also have **methods**.

Methods are **actions** that can be performed on objects.

Methods are stored in properties as **function definitions**.

|  |  |
| --- | --- |
| **Property** | **Property Value** |
| firstName | John |
| lastName | Doe |
| age | 50 |
| eyeColor | blue |
| fullName | function() {return this.firstName + " " + this.lastName;} |

A method is a function stored as a property.

Example

const person = {  
  firstName: "John",  
  lastName : "Doe",  
  id       : 5566,  
  fullName : function() {  
    return this.firstName + " " + this.lastName;  
  }  
};

The **this** Keyword

In a function definition, this refers to the "owner" of the function.

In the example above, this is the **person object** that "owns" the fullName function.

In other words, this.firstName means the firstName property of **this object**.

**Accessing Object Methods**

You access an object method with the following syntax:

*objectName.methodName()*

Example

name = person.fullName();

If you access a method **without** the () parentheses, it will return the **function definition**:

Example

name = person.fullName;

**Do Not Declare Strings, Numbers, and Booleans as Objects!**

When a JavaScript variable is declared with the keyword "new", the variable is created as an object:

x = new String();        // Declares x as a String object  
y = new Number();        // Declares y as a Number object  
z = new Boolean();       // Declares z as a Boolean object

Avoid String, Number, and Boolean objects. They complicate your code and slow down execution speed.

JavaScript Events

Html events are things that happen to html elements.

When JavaScript is used in the 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:

Example

<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>

JavaScript code is often several lines long. It is more common to see event attributes calling functions:

<button onclick="displayDate()">The time is?</button>

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 |

What can JavaScript Do?

Event handlers can be used to handle and verify user input, user actions, and browser actions:

* Things that should be done every time a page loads
* Things that should be done when the page is closed
* Action that should be performed when a user clicks a button
* Content that should be verified when a user inputs data
* And more ...

Many different methods can be used to let JavaScript work with events:

* HTML event attributes can execute JavaScript code directly
* HTML event attributes can call JavaScript functions
* You can assign your own event handler functions to HTML elements
* You can prevent events from being sent or being handled
* And more

**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 switch statement is described in the next chapter.

**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*}

Note that if is in lowercase letters. Uppercase letters (If or IF) will generate a JavaScript error.

Example

Make a "Good day" greeting if the hour is less than 18:00:

if (hour < 18) {  
  greeting = "Good day";  
}

The result of greeting will be:

Good day

**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*}

Example

If the hour is less than 18, create a "Good day" greeting, otherwise "Good evening":

if (hour < 18) {  
  greeting = "Good day";  
} else {  
  greeting = "Good evening";  
}

The result of greeting will be:

Good day

**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*}

Example

If time is less than 10:00, create a "Good morning" greeting, if not, but time is less than 20:00, create a "Good day" greeting, otherwise a "Good evening":

if (time < 10) {  
  greeting = "Good morning";  
} else if (time < 20) {  
  greeting = "Good day";  
} else {  
  greeting = "Good evening";  
}

The result of greeting will be:

Good day

**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:

Monday

**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.

**Note:**If you omit the break statement, the next case will be executed even if the evaluation does not match the case.

**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

The default case does not have to be the last case in a switch block:

Example

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

If default is not the last case in the switch block, remember to end the default case with a break.

**Common Code Blocks**

Sometimes you will want different switch cases to use the same code.

In this example case 4 and 5 share the same code block, and 0 and 6 share another code block:

Example

switch (new Date().getDay()) {  
  case 4:  
  case 5:  
    text = "Soon it is Weekend";  
    break;  
  case 0:  
  case 6:  
    text = "It is Weekend";  
    break;  
  default:  
    text = "Looking forward to the Weekend";  
}

Switching Details

If multiple cases matches a case value, the **first** case is selected.

If no matching cases are found, the program continues to the **default** label.

If no default label is found, the program continues to the statement(s) **after the switch**.

**Strict Comparison**

Switch cases use **strict** comparison (===).

The values must be of the same type to match.

A strict comparison can only be true if the operands are of the same type.

In this example there will be no match for x:

Example

let x = "0";  
switch (x) {  
  case 0:  
    text = "Off";  
    break;  
  case 1:  
    text = "On";  
    break;  
  default:  
    text = "No value found";  
}

**JavaScript for loop:**

Loop while execute a block of code a number of times.

**JavaScript loops**

Loops are handy, if we want to run the same code over and over again, each time with different values.

Often this is the case when working with arrays:

text += cars[0] + "<br>";

text +=car[1]+ “<br>”;

text +=car[2] +”<br>”;

text += car[3] +”<br>”;

text += car[4] +”<br>”;

you can write:

for (let i =0; i < cars.length ; i++) {

text += Car[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 loop has the following syntax:

for (*statement 1*;*statement 2*;*statement 3*) {  
  // *code block to be executed*  
}

**Statement 1** is executed (one time) before the execution of the code block.

**Statement 2** defines the condition for executing the code block.

**Statement 3** is executed (every time) after the code block has been executed.

Example

for (let i = 0; i < 5; i++) {  
  text += "The number is " + i + "<br>";  
}

From the example above, you can read:

Statement 1 sets a variable before the loop starts (let i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5).

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

Statement 1

Normally you will use statement 1 to initialize the variable used in the loop (let i = 0).

This is not always the case, JavaScript doesn't care. Statement 1 is optional.

You can initiate many values in statement 1 (separated by comma):

Example

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

And you can omit statement 1 (like when your values are set before the loop starts):

Example

let i = 2;  
let len = cars.length;  
let text = "";  
for (; i < len; i++) {  
  text += cars[i] + "<b r>";  
}

Statement 2

Often statement 2 is used to evaluate the condition of the initial variable.

This is not always the case, JavaScript doesn't care. Statement 2 is also optional.

If statement 2 returns true, the loop will start over again, if it returns false, the loop will end.

If you omit statement 2, you must provide a **break** inside the loop. Otherwise the loop will never end. This will crash your browser. Read about breaks in a later chapter of this tutorial.

Statement 3

Often statement 3 increments the value of the initial variable.

This is not always the case, JavaScript doesn't care, and statement 3 is optional.

Statement 3 can do anything like negative increment (i--), positive increment (i = i + 15), or anything else.

Statement 3 can also be omitted (like when you increment your values inside the loop):

Example

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

**Loop Scope**

Using var in a loop:

Example

var i = 5;  
  
for (var i = 0; i < 10; i++) {  
  // some code  
}  
  
// Here i is 10

**Using let in a loop:**

Example

let i = 5;  
  
for (let i = 0; i < 10; i++) {  
  // some code  
}  
  
// Here i is 5

In the first example, using var, the variable declared in the loop redeclares the variable outside the loop.

In the second example, using let, the variable declared in the loop does not redeclare the variable outside the loop.

When let is used to declare the i variable in a loop, the i variable will only be visible within the loop.

**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*  
}

Example

const person = {fname:"John", lname:"Doe", age:25};  
  
let text = "";  
for (let x in person) {  
  text += person[x];  
}

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];  
}

Do not use **for in** over an Array if the index **order** is important.

The index order is implementation-dependent, and array values may not be accessed in the order you expect.

It is better to use a **for** loop, a **for of** loop, or **Array.forEach()** when the order is important.

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.

**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 loops through a block of code as long as a specified condition is true.

Syntax

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

Example

In the following example, the code in the loop will run, over and over again, as long as a variable (i) is less than 10:

Example

while (i < 10) {  
  text += "The number is " + i;  
  i++;  
}

If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.

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*);**

Example

The example below uses a do while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

Example

do {  
  text += "The number is " + i;  
  i++;  
}  
while (i < 10);

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 Break and Continue**

The break statement "jumps out" of a loop.

The continue statement "jumps over" one iteration in the loop.

**The Break Statement**

You have already seen the break statement used in an earlier chapter of this tutorial. It was used to "jump out" of a switch() statement.

The break statement can also be used to jump out of a loop:

Example

for (let i = 0; i < 10; i++) {  
  if (i === 3) { break; }  
  text += "The number is " + i + "<br>";  
}

In the example above, the break statement ends the loop ("breaks" the loop) when the loop counter (i) is 3.

**The Continue Statement**

The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 3:

Example

for (let i = 0; i < 10; i++) {  
  if (i === 3) { continue; }  
  text += "The number is " + i + "<br>";  
}

**JavaScript Labels**

To label JavaScript statements you precede the statements with a label name and a colon:

label:  
statements

The break and the continue statements are the only JavaScript statements that can "jump out of" a code block.

Syntax:

break *labelname*;  
  
continue *labelname*;

The continue statement (with or without a label reference) can only be used to **skip one loop iteration**.

The break statement, without a label reference, can only be used to **jump out of a loop or a switch**.

With a label reference, the break statement can be used to **jump out of any code block**:

Example

const cars = ["BMW", "Volvo", "Saab", "Ford"];  
list: {  
  text += cars[0] + "<br>";  
  text += cars[1] + "<br>";  
  break list;  
  text += cars[2] + "<br>";  
  text += cars[3] + "<br>";  
}

**JavaScript typeof**

In JavaScript there are 5 different data types that can contain values:

* string
* number
* Boolean
* object
* function

There are 6 types of objects:

* Object
* Date
* Array
* String
* Number
* Boolean

And 2 data types that cannot contain values:

* null
* undefined

The typeof Operator

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

Example

typeof "John"                 // Returns "string"  
typeof 3.14                   // Returns "number"  
typeof NaN                    // Returns "number"  
typeof false                  // Returns "boolean"  
typeof [1,2,3,4]              // Returns "object"  
typeof {name:'John', age:34}  // Returns "object"  
typeof new Date()             // Returns "object"  
typeof function () {}         // Returns "function"  
typeof myCar                  // Returns "undefined" \*  
typeof null                   // Returns "object"

Please observe:

* The data type of NaN is number
* The data type of an array is object
* The data type of a date is object
* The data type of null is object
* The data type of an undefined variable is **undefined** \*
* The data type of a variable that has not been assigned a value is also **undefined** \*

You cannot use typeof to determine if a JavaScript object is an array (or a date).

Primitive Data

A primitive data value is a single simple data value with no additional properties and methods.

The typeof operator can return one of these primitive types:

* string
* number
* boolean
* undefined

Example

typeof "John"              // Returns "string"  
typeof 3.14                // Returns "number"  
typeof true                // Returns "boolean"  
typeof false               // Returns "boolean"  
typeof x                   // Returns "undefined" (if x has no value)

Complex Data

The typeof operator can return one of two complex types:

* function
* object

The typeof operator returns "object" for objects, arrays, and null.

The typeof operator does not return "object" for functions.

Example

typeof {name:'John', age:34} // Returns "object"  
typeof [1,2,3,4]             // Returns "object" (not "array", see note below)  
typeof null                  // Returns "object"  
typeof function myFunc(){}   // Returns "function"

The typeof operator returns "object" for arrays because in JavaScript arrays are objects.

**The Data Type of typeof**

The typeofoperator is not a variable. It is an operator. Operators ( + - \* / ) do not have any data type.

But, the typeof operator always **returns a string** (containing the type of the operand).

**The constructor Property**

The constructor property returns the constructor function for all JavaScript variables.

Example

"John".constructor                // Returns function String()  {[native code]}  
(3.14).constructor                // Returns function Number()  {[native code]}  
false.constructor                 // Returns function Boolean() {[native code]}  
[1,2,3,4].constructor             // Returns function Array()   {[native code]}  
{name:'John',age:34}.constructor  // Returns function Object()  {[native code]}  
new Date().constructor            // Returns function Date()    {[native code]}  
function () {}.constructor        // Returns function Function(){[native code]}

You can check the constructor property to find out if an object is an Array (contains the word "Array"):

Example

function isArray(myArray) {  
  return myArray.constructor.toString().indexOf("Array") > -1;  
}

Or even simpler, you can check if the object is an **Array function**:

Example

function isArray(myArray) {  
  return myArray.constructor === Array;  
}

You can check the constructor property to find out if an object is a Date (contains the word "Date"):

Example

function isDate(myDate) {  
  return myDate.constructor.toString().indexOf("Date") > -1;  
}

Or even simpler, you can check if the object is a **Date function**:

Example

function isDate(myDate) {  
  return myDate.constructor === Date;  
}

**Undefined**

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

Example

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.

Example

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.

Example

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

**Null**

In JavaScript null is "nothing". It is supposed to be something that doesn't exist.

Unfortunately, in JavaScript, the data type of null is an object.

You can consider it a bug in JavaScript that typeof null is an object. It should be null.

You can empty an object by setting it to null:

Example

let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};  
person = null;    // Now value is null, but type is still an object

You can also empty an object by setting it to undefined:

Example

let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};  
person = undefined;   // Now both value and type is undefined

**Difference Between Undefined and Null**

undefined and null are equal in value but different in type:

typeof undefined           // undefined  
typeof null                // object  
  
null === undefined         // false  
null == undefined          // true

**JavaScript Errors - Throw and Try to Catch**

The try statement lets you test a block of code for errors.

The catch statement lets you handle the error.

The throw statement lets you create custom errors.

The finally statement lets you execute code, after try and catch, regardless of the result.

**Errors Will Happen!**

When executing JavaScript code, different errors can occur.

Errors can be coding errors made by the programmer, errors due to wrong input, and other unforeseeable things.

Example

In this example we misspelled "alert" as "adddlert" to deliberately produce an error:

<p id="demo"></p>  
  
<script>  
try {  
  adddlert("Welcome guest!");  
}  
catch(err) {  
  document.getElementById("demo").innerHTML = err.message;  
}  
</script>

JavaScript catches **adddlert** as an error, and executes the catch code to handle it.

**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*}

**JavaScript Throws Errors**

When an error occurs, JavaScript will normally stop and generate an error message.

The technical term for this is: JavaScript will **throw an exception (throw an error)**.

JavaScript will actually create an **Error object** with two properties: **name** and **message**.

**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.

**Input Validation Example**

This example examines input. If the value is wrong, an exception (err) is thrown.

The exception (err) is caught by the catch statement and a custom error message is displayed:

<!DOCTYPE html>  
<html>  
<body>  
  
<p>Please input a number between 5 and 10:</p>  
  
<input id="demo" type="text">  
<button type="button" onclick="myFunction()">Test Input</button>  
<p id="p01"></p>  
  
<script>  
function myFunction() {  
  const message = document.getElementById("p01");  
  message.innerHTML = "";  
  let x = document.getElementById("demo").value;  
  try {  
    if(x == "") throw "empty";  
    if(isNaN(x)) throw "not a number";  
    x = Number(x);  
    if(x < 5) throw "too low";  
    if(x > 10) throw "too high";  
  }  
  catch(err) {  
    message.innerHTML = "Input is " + err;  
  }  
}  
</script>  
  
</body>  
</html>

**HTML Validation**

The code above is just an example.

Modern browsers will often use a combination of JavaScript and built-in HTML validation, using predefined validation rules defined in HTML attributes:

<input id="demo" type="number" min="5" max="10" step="1">

You can read more about forms validation in a later chapter of this tutorial.

**The finally Statement**

The finally statement lets you execute code, after try and catch, regardless of the result:

Syntax

try {  
  *Block of code to try*}  
catch(*err*) {  
  *Block of code to handle errors*}  
finally {  
  *Block of code to be executed regardless of the try / catch result*}

Example

function myFunction() {  
  const message = document.getElementById("p01");  
  message.innerHTML = "";  
  let x = document.getElementById("demo").value;  
  try {  
    if(x == "") throw "is empty";  
    if(isNaN(x)) throw "is not a number";  
    x = Number(x);  
    if(x > 10) throw "is too high";  
    if(x < 5) throw "is too low";  
  }  
  catch(err) {  
    message.innerHTML = "Error: " + err + ".";  
  }  
  finally {  
    document.getElementById("demo").value = "";  
  }  
}

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 |

The six different values are described below.

Eval Error

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

Newer versions of JavaScript do not throw EvalError. Use SyntaxError instead.

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.

Example

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:

Example

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.

Example

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:

Example

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:

Example

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

How to handle errors in js?

With the help of try, catch, throw and finally blocks we can solve the errors.

Explain try, catch, throw, and code ?

Try— the code that may cause some error in program so prevent that we use try block.

Throw—throw will throw error massage with throw block

Catch—the catch will get the error from throw block and execute code

Finally— it display the massage finally block irrespective of error

**JavaScript Scope**

Scope determines the accessibility of variable.

JavaScript has 3 types of scope:

* Block scope
* Global scope
* Function scope

**Black scope**

Before ES6 (2015), JavaScript had only Global scope and function scope.

ES6 introduction two important new JavaScript keywords: let and const.

These two keywords provide block scope in JavaScript.

Variable declared inside a { } block scope be accessed from outside the block:

Example

{

Let x = 2;

}

// x can Not be used here

**Local scope**

Variable declare within a JavaScript function, become Local to the function.

Example

// code here cannot use carName

Function myFunction() {

Let carName = “volvo”;

// code here can use carName

}

// code here can Not use carName

**Local variable have function scope:**

They can only be accessed from within the function.

Since local variable are only recognized inside their function, variable with the same name can be used in different function.

Local variables are created when a function starts, and deleted when the function is completed.

Function scope

JavaScript has function scope: each function created 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” ;

}

**Global JavaScript variables**

A variable declared outside a function, becomes Global.

Example

Let carName =”Volvo”;

// code here can use CarName

Function myFunction () {

// code here can also use CarName

}

A global variable has Global scope:

All Scripts and function on a web page can access it.

**Global Scope**

Variable declared globally have global scope.

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

Variable declared with var, let and const are quite similar when declare outside a block.

They all have Global scope:  
var x= 2 ;

Let x = 2;

Const x= 2;

**JavaScript variables**

In JavaScript, objects and function are also variables.

**Automatically Global**

If you assign a value to a variable that has 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.

Example

myFunction() ;

// code here can use CarName

Function myFunction() {

carName = “Volvo”;

}

**Strict mode**

All modern browsers support running JavaScript in “strict mode”.

You will learn more about how to use strict mode in a later chapter of this tutorial.

Global variables in HTML

With JavaScript, the global scope is the JavaScript environment.

In HTML, the global scope is the window object.

Global variables defined with the var keyword belong to the window object:

Example

Var carName = “Volvo”;

// code here can use window.carName

Global variable defined with the let key do not belong to the window object:

Example

Let carName = “Volvo”;

// code here cannot window.carName

**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 word; a variable can be used before it has been declared.

Example

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

To understand this, you have to understand the term "hoisting".

Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function).

**The let and const Keywords**

Variables defined with let and const are hoisted to the top of the block, but not *initialized*.

Meaning: The block of code is aware of the variable, but it cannot be used until it has been declared.

Using a let variable before it is declared will result in a ReferenceError.

The variable is in a "temporal dead zone" from the start of the block until it is declared:

Example

This will result in a ReferenceError:

carName = "Volvo";  
let carName;

Using a const variable before it is declared, is a syntax errror, so the code will simply not run.

Example

This code will not run.

carName = "Volvo";  
const carName;

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:

Example

var x = 5; // Initialize x  
var y;     // Declare y  
  
elem = document.getElementById("demo"); // Find an element  
elem.innerHTML = x + " " + y;           // Display x and y  
  
y = 7;    // Assign 7 to y

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 Use Strict**

"use strict"; Defines that JavaScript code should be executed in "strict mode".

The "use strict" Directive

The "use strict" directive was new in ECMAScript version 5.

It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.

The purpose of "use strict" is to indicate that the code should be executed in "strict mode".

With strict mode, you can not, for example, use undeclared variables.

You can use strict mode in all your programs. It helps you to write cleaner code, like preventing you from using undeclared variables.

"use strict" is just a string, so IE 9 will not throw an error even if it does not understand it.

Declaring Strict Mode

Strict mode is declared by adding "use strict"; to the beginning of a script or a function.

Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode):

Example

"use strict";  
x = 3.14;       // This will cause an error because x is not declared

Example

"use strict";  
myFunction();  
  
function myFunction() {  
  y = 3.14;   // This will also cause an error because y is not declared  
}

Declared inside a function, it has local scope (only the code inside the function is in strict mode):

x = 3.14;       // This will not cause an error.  
myFunction();  
  
function myFunction() {  
  "use strict";  
  y = 3.14;   // This will cause an error  
}

The "use strict"; Syntax

The syntax, for declaring strict mode, was designed to be compatible with older versions of JavaScript.

Compiling a numeric literal (4 + 5;) or a string literal ("John Doe";) in a JavaScript program has no side effects. It simply compiles to a non existing variable and dies.

So "use strict"; only matters to new compilers that "understand" the meaning of it.

Why Strict Mode?

Strict mode makes it easier to write "secure" JavaScript.

Strict mode changes previously accepted "bad syntax" into real errors.

As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.

In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.

In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

Not Allowed in Strict Mode

Using a variable, without declaring it, is not allowed:

"use strict";  
x = 3.14;                // This will cause an error

Objects are variables too.

Using an object, without declaring it, is not allowed:

"use strict";  
x = {p1:10, p2:20};      // This will cause an error

Deleting a variable (or object) is not allowed.

"use strict";  
let x = 3.14;  
delete x;                // This will cause an error

Deleting a function is not allowed.

"use strict";  
function x(p1, p2) {};  
delete x;                // This will cause an error

Duplicating a parameter name is not allowed:

"use strict";  
function x(p1, p1) {};   // This will cause an error

Octal numeric literals are not allowed:

"use strict";  
let x = 010;             // This will cause an error

Octal escape characters are not allowed:

"use strict";  
let x = "\010";            // This will cause an error

Writing to a read-only property is not allowed:

"use strict";  
const obj = {};  
Object.defineProperty(obj, "x", {value:0, writable:false});  
  
obj.x = 3.14;            // This will cause an error

Writing to a get-only property is not allowed:

"use strict";  
const obj = {get x() {return 0} };  
  
obj.x = 3.14;            // This will cause an error

Deleting an undeletable property is not allowed:

"use strict";  
delete Object.prototype; // This will cause an error

The word eval cannot be used as a variable:

"use strict";  
let eval = 3.14;         // This will cause an error

The word arguments cannot be used as a variable:

"use strict";  
let arguments = 3.14;    // This will cause an error

The with statement is not allowed:

"use strict";  
with (Math){x = cos(2)}; // This will cause an error

For security reasons, eval() is not allowed to create variables in the scope from which it was called:

"use strict";  
eval ("let x = 2");  
alert (x);             // This will cause an error

The this keyword in functions behaves differently in strict mode.

The this keyword refers to the object that called the function.

If the object is not specified, functions in strict mode will return undefined and functions in normal mode will return the global object (window):

"use strict";  
function myFunction() {  
  alert(this); // will alert "undefined"  
}  
myFunction();

Keywords reserved for future JavaScript versions can NOT be used as variable names in strict mode.

These are:

* implements
* interface
* let
* package
* private
* protected
* public
* static
* yield

"use strict";  
let public = 1500;      // This will cause an error

**JavaScript this keyword**

What is this?

The JavaScript keyword refers to the object it belongs to.

In a method, this refers to the **owner object**.

const **person** = {  
  firstName: "John",  
  lastName : "Doe",  
  id       : 5566,  
  fullName : function() {  
    return **this**.firstName + " " + **this**.lastName;  
  }  
};

console.log(**this**.firstName + " " + **this**.last Name);

Alone, this refers to the **global object**.

Let x = this;

Console.log(x); // this object window

Alone, this refers to the **global object**.

“use strict”;

Let x =this;

// object window same as alone

In a JavaScript function, the owner of the function is the **default** binding for this.

Console.log(myFunction);

Function myFunction() {

Return this;

}

//object window

By using “use strict” inside function **this** is undefined

In an event, this refers to the **element** that received the event.

const **person** = {  
  firstName  : "John",  
  lastName   : "Doe",  
  id         : 5566,  
  myFunction : function() {  
    return **this**;  
  }  
};

console.log(function);