**Java Script**

JavaScript is a dynamic computer programming language. It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages. It is an interpreted programming language with object-oriented capabilities.

JavaScript was first known as **LiveScript,** but Netscape changed its name to JavaScript, possibly because of the excitement being generated by Java. JavaScript made its first appearance in Netscape 2.0 in 1995 with the name **LiveScript**. The general-purpose core of the language has been embedded in Netscape, Internet Explorer, and other web browsers.

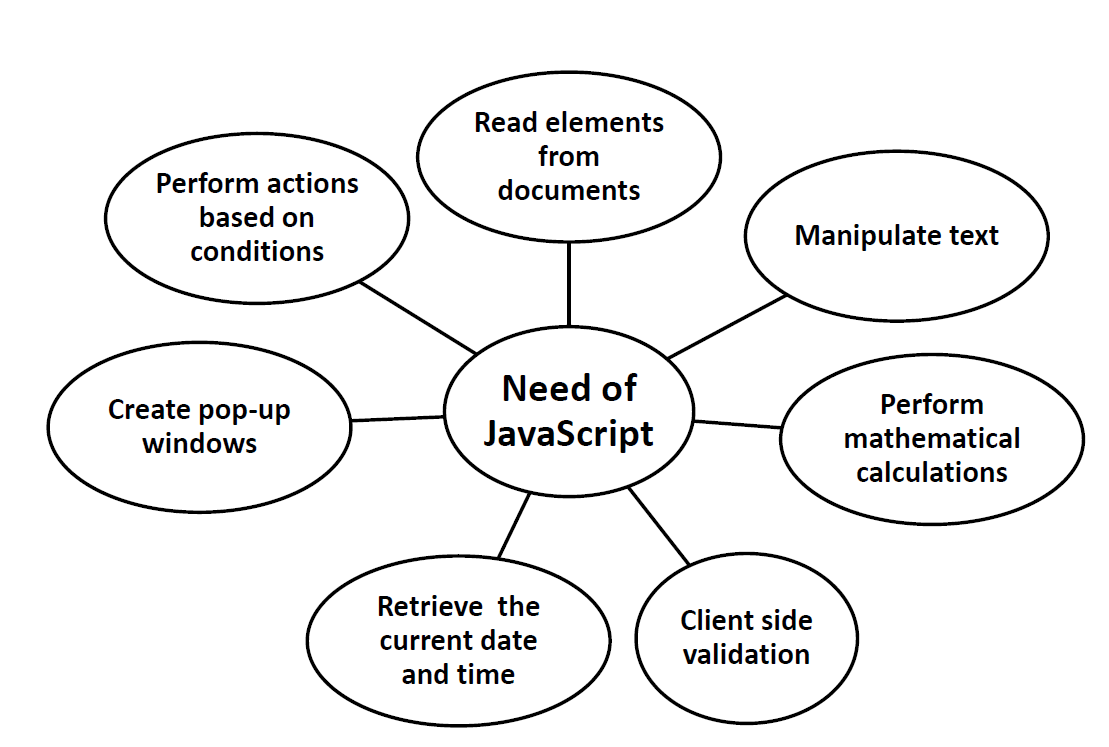
* JavaScript is a lightweight, interpreted programming language.
* Designed for creating network-centric applications.
* Complementary to and integrated with Java.
* Complementary to and integrated with HTML.
* Open and cross-platform

**Advantages of learning Javascript:**

* Javascript is the most popular **programming language** in the world and that makes it a programmer’s great choice. Once you learnt Javascript, it helps you developing great front-end as well as back-end softwares using different Javascript based frameworks like jQuery, Node.JS etc.
* Javascript is everywhere, it comes installed on every modern web browser and so to learn Javascript you really do not need any special environment setup. For example Chrome, Mozilla Firefox , Safari and every browser you know as of today, supports Javascript.
* Javascript helps you create really beautiful and crazy fast websites. You can develop your website with a console like look and feel and give your users the best Graphical User Experience.
* JavaScript usage has now extended to mobile app development, desktop app development, and game development. This opens many opportunities for you as Javascript Programmer.
* Due to high demand, there is tons of job growth and high pay for those who know JavaScript. You can navigate over to different job sites to see what having JavaScript skills looks like in the job market.
* Great thing about Javascript is that you will find tons of frameworks and Libraries already developed which can be used directly in your software development to reduce your time to market.

Applications of Javascript Programming

* **Client side validation** - This is really important to verify any user input before submitting it to the server and Javascript plays an important role in validting those inputs at front-end itself.
* **Manipulating HTML Pages** - Javascript helps in manipulating HTML page on the fly. This helps in adding and deleting any HTML tag very easily using javascript and modify your HTML to change its look and feel based on different devices and requirements.
* **User Notifications** - You can use Javascript to raise dynamic pop-ups on the webpages to give different types of notifications to your website visitors.
* **Back-end Data Loading** - Javascript provides Ajax library which helps in loading back-end data while you are doing some other processing. This really gives an amazing experience to your website visitors.
* **Presentations** - JavaScript also provides the facility of creating presentations which gives website look and feel. JavaScript provides RevealJS and BespokeJS libraries to build a web-based slide presentations.
* **Server Applications** - Node JS is built on Chrome's Javascript runtime for building fast and scalable network applications. This is an event based library which helps in developing very sophisticated server applications including Web Servers.



First Java Script Code:

<html>

<body>

<script language = "javascript" type = "text/javascript">

<!--

document.write("Hello World!")

//-->

</script>

</body>

</html>

Note: Java Script is a case sensitive language.

**Comments in JS:**

* Any text between a // and the end of a line is treated as a comment and is ignored by JavaScript.
* Any text between the characters /\* and \*/ is treated as a comment. This may span multiple lines.
* JavaScript also recognizes the HTML comment opening sequence <!--. JavaScript treats this as a single-line comment, just as it does the // comment.
* The HTML comment closing sequence --> is not recognized by JavaScript so it should be written as //-->.

# JavaScript - Placement in HTML File

## JavaScript in <head>...</head> section

## If you want to have a script run on some event, such as when a user clicks somewhere, then you will place that script in the head as follows

<!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>...</body> section

If you need a script to run as the page loads so that the script generates content in the page, then the script goes in the <body> portion of the document. In this case, you would not have any function defined using JavaScript. Take a look at the following code.

<html>

<head>

</head>

<body>

<script type = "text/javascript">

<!--

document.write("Hello World")

//-->

</script>

<p>This is web page body </p>

</body>

</html>

## External JavaScript

### **External file: myScript.js**

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.

**JavaScript versions**

JavaScript was invented by Brendan Eich in 1995, and became an ECMA( European Computer Manufacturer's Association) standard in 1997.

ECMAScript is the official name of the language.

ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6.

Since 2016, versions are named by year (ECMAScript 2016, 2017, 2018, 2019, 2020).

# **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:

### **Example**

<!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 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>

The document.write() method should only be used for testing.

## Using window.alert()

You can use an alert box to display data:

### **Example**

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

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

alert(5 + 6);

## Using console.log()

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

### **Example**

<h2>Activate Debugging</h2>

<p>F12 on your keyboard will activate debugging.</p>

<p>Then select "Console" in the debugger menu.</p>

<p>Then click Run again.</p>

<script>

console.log(5 + 6);

</script>

## Semicolons

A semicolon may be omitted in most cases when a line break exists.

This would also work:

alert('Hello')

alert('World')

Here, JavaScript interprets the line break as an “implicit” semicolon. This is called an automatic semicolon insertion.

**In most cases, a newline implies a semicolon. But “in most cases” does not mean “always”!**

There are cases when a newline does not mean a semicolon. For example:

alert(3 +

1

+ 2);

The code outputs 6 because JavaScript does not insert semicolons here. It is intuitively obvious that if the line ends with a plus "+", then it is an “incomplete expression”, so a semicolon there would be incorrect. And in this case, that works as intended.

**But there are situations where JavaScript “fails” to assume a semicolon where it is really needed.**

Errors which occur in such cases are quite hard to find and fix.

**An example of an error**

If you’re curious to see a concrete example of such an error, check this code out:

alert("Hello");

[1, 2].forEach(alert);

No need to think about the meaning of the brackets [] and forEach yet. We’ll study them later. For now, just remember the result of running the code: it shows Hello, then 1, then 2.

Now let’s remove the semicolon after the alert:

alert("Hello")

[1, 2].forEach(alert);

The difference compared to the code above is only one character: the semicolon at the end of the first line is gone.

If we run this code, only the first Hello shows (and there’s an error, you may need to open the console to see it). There are no numbers any more.

That’s because JavaScript does not assume a semicolon before square brackets [...]. So, the code in the last example is treated as a single statement.

Here’s how the engine sees it:

alert("Hello")[1, 2].forEach(alert);

Looks weird, right? Such merging in this case is just wrong. We need to put a semicolon after alert for the code to work correctly.

This can happen in other situations also.

We recommend putting semicolons between statements even if they are separated by newlines. This rule is widely adopted by the community. Let’s note once again – it is possible to leave out semicolons most of the time. But it’s safer – especially for a beginner – to use them.

## 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  
  
1001

**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.

avaScript Variables can be declared in 4 ways:

* Automatically
* Using var
* Using let
* Using const

x = 5;  
y = 6;  
z = x + y;

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

The var keyword was used in all JavaScript code from 1995 to 2015.

The let and const keywords were added to JavaScript in 2015.

The var keyword should only be used in code written for older browsers.

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

const price1 = 5;  
const price2 = 6;  
let total = price1 + price2

The two variables price1 and price2 are declared with the const keyword.

These are constant values and cannot be changed.

The variable total is declared with the let keyword.

The value total can be changed.

## When to Use var, let, or const?

1. Always declare variables

2. Always use const if the value should not be changed

3. Always use const if the type should not be changed (Arrays and Objects)

4. Only use let if you can't use const

5. Only use var if you MUST support old browsers.

An **equal sign** is used to **a0ssign values** to variables.

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

let x;  
x = 6;

<h2>JavaScript Variables</h2>

<p>In this example, x is defined as a variable.

Then, x is assigned the value of 6:</p>

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

<script>

let x;

x = 6;

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

</script>

## JavaScript Identifiers

All JavaScript **variables** must be **identified** with **unique names**.

These unique names are called **identifiers**.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

The general rules for constructing names for variables (unique identifiers) are:

* Names can contain letters, digits, underscores, and dollar signs.
* Names must begin with a letter.
* Names can also begin with $ and \_ (but we will not use it in this tutorial).
* Names are case sensitive (y and Y are different variables).
* Reserved words (like JavaScript keywords) cannot be used as names.

## JavaScript Data Types

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

In programming, text values are called text strings.

JavaScript 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.

## JavaScript Dollar Sign $

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

### **Example**

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

### **JavaScript has 8 Datatypes**

**1. String  
2. Number**

Besides regular numbers, there are so-called “special numeric values” which also belong to this data type: Infinity, -Infinity and NaN.

Infinity represents the mathematical [Infinity](https://en.wikipedia.org/wiki/Infinity) ∞. It is a special value that’s greater than any number.

We can get it as a result of division by zero:

alert( 1 / 0 ); // Infinity

Or just reference it directly:

alert( Infinity ); // Infinity

NaN represents a computational error. It is a result of an incorrect or an undefined mathematical operation, for instance:

alert( "not a number" / 2 ); // NaN, such division is erroneous

NaN is sticky. Any further mathematical operation on NaN returns NaN:

alert( NaN + 1 ); // NaN

alert( 3 \* NaN ); // NaN

alert( "not a number" / 2 - 1 ); // NaN

So, if there’s a NaN somewhere in a mathematical expression, it propagates to the whole result (there’s only one exception to that: NaN \*\* 0 is 1).

**3. Bigint**

In JavaScript, the “number” type cannot safely represent integer values larger than (253-1) (that’s 9007199254740991), or less than -(253-1) for negatives.

(let x = BigInt("123456789012345678901234567890");)

A BigInt value is created by appending n to the end of an integer:

// the "n" at the end means it's a BigInt

const bigInt = 1234567890123456789012345678901234567890n;

**4. Boolean**  (x == y)       // Returns true  
 (x == z)       // Returns false  
5. **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

**6. Null  
7. Symbol  
8. Object**

# **JavaScript Operators**

There are different types of JavaScript operators:

* Arithmetic Operators
* Assignment Operators
* Comparison Operators
* String Operators
* Logical Operators
* Bitwise Operators
* Ternary Operators
* Type Operators

## Arithmetic Operator

## Operator Description

## + Addition

## - Subtraction

## \* Multiplication

## \*\* Exponentiation (ES2016)

## / Division

## % Modulus (Division Remainder)

## ++ Increment

## -- Decrement

## 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

## String Comparison

## let text1 = "A"; let text2 = "B"; let result = text1 < text2;

## Note that strings are compared alphabetically:

## String Addition

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

### **Example**

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

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

### **Example**

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

## [Comparison of different types](https://javascript.info/comparison" \l "comparison-of-different-types)

When comparing values of different types, JavaScript converts the values to numbers.

For example:

alert( '2' > 1 ); // true, string '2' becomes a number 2

alert( '01' == 1 ); // true, string '01' becomes a number 1

For boolean values, true becomes 1 and false becomes 0.

For example:

alert( true == 1 ); // true

alert( false == 0 ); // true

**A funny consequence**

It is possible that at the same time:

* Two values are equal.
* One of them is true as a boolean and the other one is false as a boolean.

For example:

let a = 0;

alert( Boolean(a) ); // false

let b = "0";

alert( Boolean(b) ); // true

alert(a == b); // true!

From JavaScript’s standpoint, this result is quite normal. An equality check converts values using the numeric conversion (hence "0" becomes 0), while the explicit Boolean conversion uses another set of rules.

## [Strict equality](https://javascript.info/comparison" \l "strict-equality)

A regular equality check == has a problem. It cannot differentiate 0 from false:

alert( 0 == false ); // true

The same thing happens with an empty string:

alert( '' == false ); // true

This happens because operands of different types are converted to numbers by the equality operator ==. An empty string, just like false, becomes a zero.

What to do if we’d like to differentiate 0 from false?

**A strict equality operator === checks the equality without type conversion.**

In other words, if a and b are of different types, then a === b immediately returns false without an attempt to convert them.

Let’s try it:

alert( 0 === false ); // false, because the types are different

There is also a “strict non-equality” operator !== analogous to !=.

The strict equality operator is a bit longer to write, but makes it obvious what’s going on and leaves less room for errors.

## [Comparison with null and undefined](https://javascript.info/comparison" \l "comparison-with-null-and-undefined)

There’s a non-intuitive behavior when null or undefined are compared to other values.

**For a strict equality check ===**

These values are different, because each of them is a different type.

alert( null === undefined ); // false

**For a non-strict check ==**

There’s a special rule. These two are a “sweet couple”: they equal each other (in the sense of ==), but not any other value.

alert( null == undefined ); // true

**For maths and other comparisons < > <= >=**

null/undefined are converted to numbers: null becomes 0, while undefined becomes NaN.

Now let’s see some funny things that happen when we apply these rules. And, what’s more important, how to not fall into a trap with them.

### [Strange result: null vs 0](https://javascript.info/comparison" \l "strange-result-null-vs-0)

Let’s compare null with a zero:

alert( null > 0 ); // (1) false

alert( null == 0 ); // (2) false

alert( null >= 0 ); // (3) true

Mathematically, that’s strange. The last result states that "null is greater than or equal to zero", so in one of the comparisons above it must be true, but they are both false.

The reason is that an equality check == and comparisons > < >= <= work differently. Comparisons convert null to a number, treating it as 0. That’s why (3) null >= 0 is true and (1) null > 0 is false.

On the other hand, the equality check == for undefined and null is defined such that, without any conversions, they equal each other and don’t equal anything else. That’s why (2) null == 0 is false.

### [An incomparable undefined](https://javascript.info/comparison" \l "an-incomparable-undefined)

The value undefined shouldn’t be compared to other values:

alert( undefined > 0 ); // false (1)

alert( undefined < 0 ); // false (2)

alert( undefined == 0 ); // false (3)

Why does it dislike zero so much? Always false!

We get these results because:

* Comparisons (1) and (2) return false because undefined gets converted to NaN and NaN is a special numeric value which returns false for all comparisons.
* The equality check (3) returns false because undefined only equals null, undefined, and no other value.

### [Avoid problems](https://javascript.info/comparison" \l "avoid-problems)

Why did we go over these examples? Should we remember these peculiarities all the time? Well, not really. Actually, these tricky things will gradually become familiar over time, but there’s a solid way to avoid problems with them:

* Treat any comparison with undefined/null except the strict equality === with exceptional care.
* Don’t use comparisons >= > < <= with a variable which may be null/undefined, unless you’re really sure of what you’re doing. If a variable can have these values, check for them separately.

**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

The typeof operator returns the type of the operand.

A call to typeof x returns a string with the type name:

typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object" (1)

typeof null // "object" (2)

typeof alert // "function" (3)

The last three lines may need additional explanation:

1. Math is a built-in object that provides mathematical operations. We will learn it in the chapter [Numbers](https://javascript.info/number). Here, it serves just as an example of an object.
2. The result of typeof null is "object". That’s an officially recognized error in typeof, coming from very early days of JavaScript and kept for compatibility. Definitely, null is not an object. It is a special value with a separate type of its own. The behavior of typeof is wrong here.
3. The result of typeof alert is "function", because alert is a function. We’ll study functions in the next chapters where we’ll also see that there’s no special “function” type in JavaScript. Functions belong to the object type. But typeof treats them differently, returning "function". That also comes from the early days of JavaScript. Technically, such behavior isn’t correct, but can be convenient in practice.

**The typeof(x) syntax**

You may also come across another syntax: typeof(x). It’s the same as typeof x.

To put it clear: typeof is an operator, not a function. The parentheses here aren’t a part of typeof. It’s the kind of parentheses used for mathematical grouping.

Usually, such parentheses contain a mathematical expression, such as (2 + 2), but here they contain only one argument (x). Syntactically, they allow to avoid a space between the typeof operator and its argument, and some people like it.

Some people prefer typeof(x), although the typeof x syntax is much more common.

# **Type Conversions**

Most of the time, operators and functions automatically convert the values given to them to the right type.

For example, alert automatically converts any value to a string to show it. Mathematical operations convert values to numbers.

## There are also cases when we need to explicitly convert a value to the expected type.

## String Conversion

String conversion happens when we need the string form of a value.

For example, alert(value) does it to show the value.

We can also call the String(value) function to convert a value to a string:

let value = true;

alert(typeof value); // boolean

value = String(value); // now value is a string "true"

alert(typeof value); // string

String conversion is mostly obvious. A false becomes "false", null becomes "null", etc.

## [Numeric Conversion](https://javascript.info/type-conversions" \l "numeric-conversion)

Numeric conversion in mathematical functions and expressions happens automatically.

For example, when division / is applied to non-numbers:

alert( "6" / "2" ); // 3, strings are converted to numbers

We can use the Number(value) function to explicitly convert a value to a number:

let str = "123";

alert(typeof str); // string

let num = Number(str); // becomes a number 123

alert(typeof num); // number

Explicit conversion is usually required when we read a value from a string-based source like a text form but expect a number to be entered.

If the string is not a valid number, the result of such a conversion is NaN. For instance:

let age = Number("an arbitrary string instead of a number");

alert(age); // NaN, conversion failed

Numeric conversion rules:

| **Value** | **Becomes…** |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true and false | 1 and 0 |
| string | Whitespaces (includes spaces, tabs \t, newlines \n etc.) from the start and end are removed. If the remaining string is empty, the result is 0. Otherwise, the number is “read” from the string. An error gives NaN. |

Examples:

alert( Number(" 123 ") ); // 123

alert( Number("123z") ); // NaN (error reading a number at "z")

alert( Number(true) ); // 1

alert( Number(false) ); // 0

Please note that null and undefined behave differently here: null becomes zero while undefined becomes NaN.

Most mathematical operators also perform such conversion, we’ll see that in the next chapter.

## [Boolean Conversion](https://javascript.info/type-conversions" \l "boolean-conversion)

Boolean conversion is the simplest one.

It happens in logical operations (later we’ll meet condition tests and other similar things) but can also be performed explicitly with a call to Boolean(value).

The conversion rule:

* Values that are intuitively “empty”, like 0, an empty string, null, undefined, and NaN, become false.
* Other values become true.

For instance:

alert( Boolean(1) ); // true

alert( Boolean(0) ); // false

alert( Boolean("hello") ); // true

alert( Boolean("") ); // false

**Please note: the string with zero "0" is true**

Some languages (namely PHP) treat "0" as false. But in JavaScript, a non-empty string is always true.

alert( Boolean("0") ); // true

alert( Boolean(" ") ); // spaces, also true (any non-empty string is true)

## 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}

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

let year = prompt('In which year was ECMAScript-2015 specification published?', '');

## if (year == 2015) alert( 'You are right!' );

## 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}

## [Conditional operator ‘?’](https://javascript.info/ifelse" \l "conditional-operator)

## let result = condition ? value1 : value2;

# The "switch" statement

switch(x) {

case 'value1': // if (x === 'value1')

...

[break]

case 'value2': // if (x === 'value2')

...

[break]

default:

...

[break]

## }

* The value of x is checked for a strict equality to the value from the first case (that is, value1) then to the second (value2) and so on.

**Any expression can be a switch/case argument**

Both switch and case allow arbitrary expressions.

For example:

let a = "1";

let b = 0;

switch (+a) {

case b + 1:

alert("this runs, because +a is 1, exactly equals b+1");

break;

default:

alert("this doesn't run");

}

Here +a gives 1, that’s compared with b + 1 in case, and the corresponding code is executed.

## [Type matters](https://javascript.info/switch" \l "type-matters)

Let’s emphasize that the equality check is always strict. The values must be of the same type to match.

For example, let’s consider the code:

let arg = prompt("Enter a value?");

switch (arg) {

case '0':

case '1':

alert( 'One or zero' );

break;

case '2':

alert( 'Two' );

break;

case 3:

alert( 'Never executes!' );

break;

default:

alert( 'An unknown value' );

}

# Loops: while and for

## Everything works the same way including break and continue as C programming .

## [Labels for break/continue](https://javascript.info/while-for" \l "labels-for-break-continue)

## Sometimes we need to break out from multiple nested loops at once.

Outer: for (let i = 0; i < 3; i++) {

for (let j = 0; j < 3; j++) {

let input = prompt(`Value at 9coords (${i},${j})`, '');

// what if we want to exit from here to Done (below)?

if (!input) break outer;

// do something with the value...

}

}

## Function

function name(parameter1, parameter2, ... parameterN) {

// body

## }

function showMessage() {

alert( 'Hello everyone!' );

}

showMessage();

## showMessage();

## [Local variables](https://javascript.info/function-basics" \l "local-variables)

A variable declared inside a function is only visible inside that function.

For example:

function showMessage() {

let message = "Hello, I'm JavaScript!"; // local variable

alert( message );

}

showMessage(); // Hello, I'm JavaScript!

alert( message ); // <-- Error! The variable is local to the function

## [Outer variables](https://javascript.info/function-basics" \l "outer-variables)

A function can access an outer variable as well, for example:

let userName = 'John';

function showMessage() {

let message = 'Hello, ' + userName;

alert(message);

}

showMessage(); // Hello, John

The function has full access to the outer variable. It can modify it as well.

For instance:

let userName = 'John';

function showMessage() {

userName = "Bob"; // (1) changed the outer variable

let message = 'Hello, ' + userName;

alert(message);

}

alert( userName ); // John before the function call

showMessage();

alert( userName ); // Bob, the value was modified by the function

The outer variable is only used if there’s no local one.

If a same-named variable is declared inside the function then it *shadows* the outer one. For instance, in the code below the function uses the local userName. The outer one is ignored:

let userName = 'John';

function showMessage() {

let userName = "Bob"; // declare a local variable

let message = 'Hello, ' + userName; // Bob

alert(message);

}

// the function will create and use its own userName

showMessage();

alert( userName ); // John, unchanged, the function did not access the outer variable

**Global variables**

Variables declared outside of any function, such as the outer userName in the code above, are called *global*.

Global variables are visible from any function (unless shadowed by locals).

It’s a good practice to minimize the use of global variables. Modern code has few or no globals. Most variables reside in their functions. Sometimes though, they can be useful to store project-level data.

## [Parameters](https://javascript.info/function-basics" \l "parameters)

We can pass arbitrary data to functions using parameters.

In the example below, the function has two parameters: from and text.

function showMessage(from, text) { // parameters: from, text

alert(from + ': ' + text);

}

showMessage('Ann', 'Hello!'); // Ann: Hello! (\*)

showMessage('Ann', "What's up?"); // Ann: What's up? (\*\*)

Here’s one more example: we have a variable from and pass it to the function. Please note: the function changes from, but the change is not seen outside, because a function always gets a copy of the value:

function showMessage(from, text) {

from = '\*' + from + '\*'; // make "from" look nicer

alert( from + ': ' + text );

}

let from = "Ann";

showMessage(from, "Hello"); // \*Ann\*: Hello

// the value of "from" is the same, the function modified a local copy

alert( from ); // Ann

When a value is passed as a function parameter, it’s also called an argument.

In other words, to put these terms straight:

* A parameter is the variable listed inside the parentheses in the function declaration (it’s a declaration time term).
* An argument is the value that is passed to the function when it is called (it’s a call time term).

We declare functions listing their parameters, then call them passing arguments.

In the example above, one might say: "the function showMessage is declared with two parameters, then called with two arguments: from and "Hello"".

## [Default values](https://javascript.info/function-basics" \l "default-values)

If a function is called, but an argument is not provided, then the corresponding value becomes undefined.

For instance, the aforementioned function showMessage(from, text) can be called with a single argument:

showMessage("Ann");

That’s not an error. Such a call would output "\*Ann\*: undefined". As the value for text isn’t passed, it becomes undefined.

We can specify the so-called “default” (to use if omitted) value for a parameter in the function declaration, using =:

function showMessage(from, text = "no text given") {

alert( from + ": " + text );

}

showMessage("Ann"); // Ann: no text given

Now if the text parameter is not passed, it will get the value "no text given".

The default value also jumps in if the parameter exists, but strictly equals undefined, like this:

showMessage("Ann", undefined); // Ann: no text given

Here "no text given" is a string, but it can be a more complex expression, which is only evaluated and assigned if the parameter is missing. So, this is also possible:

function showMessage(from, text = anotherFunction()) {

// anotherFunction() only executed if no text given

// its result becomes the value of text

}

**Evaluation of default parameters**

In JavaScript, a default parameter is evaluated every time the function is called without the respective parameter.

In the example above, anotherFunction() isn’t called at all, if the text parameter is provided.

On the other hand, it’s independently called every time when text is missing.

### [Alternative default parameters](https://javascript.info/function-basics" \l "alternative-default-parameters)

Sometimes it makes sense to assign default values for parameters at a later stage after the function declaration.

We can check if the parameter is passed during the function execution, by comparing it with undefined:

function showMessage(text) {

// ...

if (text === undefined) { // if the parameter is missing

text = 'empty message';

}

alert(text);

}

showMessage(); // empty message

…Or we could use the || operator:

function showMessage(text) {

// if text is undefined or otherwise falsy, set it to 'empty'

text = text || 'empty';

...

}

Modern JavaScript engines support the [nullish coalescing operator](https://javascript.info/nullish-coalescing-operator) ??, it’s better when most falsy values, such as 0, should be considered “normal”:

function showCount(count) {

// if count is undefined or null, show "unknown"

alert(count ?? "unknown");

}

showCount(0); // 0

showCount(null); // unknown

showCount(); // unknown

## [Returning a value](https://javascript.info/function-basics" \l "returning-a-value)

A function can return a value back into the calling code as the result.

The simplest example would be a function that sums two values:

function sum(a, b) {

return a + b;

}

let result = sum(1, 2);

alert( result ); // 3

The directive return can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code (assigned to result above).

There may be many occurrences of return in a single function. For instance:

function checkAge(age) {

if (age >= 18) {

return true;

} else {

return confirm('Do you have permission from your parents?');

}

}

let age = prompt('How old are you?', 18);

if ( checkAge(age) ) {

alert( 'Access granted' );

} else {

alert( 'Access denied' );

}

It is possible to use return without a value. That causes the function to exit immediately.

For example:

function showMovie(age) {

if ( !checkAge(age) ) {

return;

}

alert( "Showing you the movie" ); // (\*)

// ...

}

In the code above, if checkAge(age) returns false, then showMovie won’t proceed to the alert.

**A function with an empty return or without it returns undefined**

If a function does not return a value, it is the same as if it returns undefined:

function doNothing() { /\* empty \*/ }

alert( doNothing() === undefined ); // true

An empty return is also the same as return undefined:

function doNothing() {

return;

}

alert( doNothing() === undefined ); // true

**Never add a newline between return and the value**

For a long expression in return, it might be tempting to put it on a separate line, like this:

return

(some + long + expression + or + whatever \* f(a) + f(b))

That doesn’t work, because JavaScript assumes a semicolon after return. That’ll work the same as:

return;

(some + long + expression + or + whatever \* f(a) + f(b))

So, it effectively becomes an empty return.

If we want the returned expression to wrap across multiple lines, we should start it at the same line as return. Or at least put the opening parentheses there as follows:

return (

some + long + expression

+ or +

whatever \* f(a) + f(b)

)

And it will work just as we expect it to.

## [Naming a function](https://javascript.info/function-basics" \l "function-naming)

Functions are actions. So their name is usually a verb. It should be brief, as accurate as possible and describe what the function does, so that someone reading the code gets an indication of what the function does.

It is a widespread practice to start a function with a verbal prefix which vaguely describes the action. There must be an agreement within the team on the meaning of the prefixes.

For instance, functions that start with "show" usually show something.

Function starting with…

* "get…" – return a value,
* "calc…" – calculate something,
* "create…" – create something,
* "check…" – check something and return a boolean, etc.

Examples of such names:

showMessage(..) // shows a message

getAge(..) // returns the age (gets it somehow)

calcSum(..) // calculates a sum and returns the result

createForm(..) // creates a form (and usually returns it)

checkPermission(..) // checks a permission, returns true/false

With prefixes in place, a glance at a function name gives an understanding what kind of work it does and what kind of value it returns.

**One function – one action**

A function should do exactly what is suggested by its name, no more.

Two independent actions usually deserve two functions, even if they are usually called together (in that case we can make a 3rd function that calls those two).

A few examples of breaking this rule:

* getAge – would be bad if it shows an alert with the age (should only get).
* createForm – would be bad if it modifies the document, adding a form to it (should only create it and return).
* checkPermission – would be bad if it displays the access granted/denied message (should only perform the check and return the result).

These examples assume common meanings of prefixes. You and your team are free to agree on other meanings, but usually they’re not much different. In any case, you should have a firm understanding of what a prefix means, what a prefixed function can and cannot do. All same-prefixed functions should obey the rules. And the team should share the knowledge.

**Ultrashort function names**

Functions that are used very often sometimes have ultrashort names.

For example, the [jQuery](https://jquery.com/) framework defines a function with $. The [Lodash](https://lodash.com/) library has its core function named \_.

These are exceptions. Generally function names should be concise and descriptive.

# **Function expressions**

There is another syntax for creating a function that is called a Function Expression.

It allows us to create a new function in the middle of any expression.

For example:

let sayHi = function() {

alert( "Hello" );

};

Here we can see a variable sayHi getting a value, the new function, created as function() { alert("Hello"); }.

As the function creation happens in the context of the assignment expression (to the right side of =), this is a Function Expression.

Please note, there’s no name after the function keyword. Omitting a name is allowed for Function Expressions.

Here we immediately assign it to the variable, so the meaning of these code samples is the same: "create a function and put it into the variable sayHi".

In more advanced situations, that we’ll come across later, a function may be created and immediately called or scheduled for a later execution, not stored anywhere, thus remaining anonymous.

## [Function is a value](https://javascript.info/function-expressions" \l "function-is-a-value)

Let’s reiterate: no matter how the function is created, a function is a value. Both examples above store a function in the sayHi variable.

We can even print out that value using alert:

function sayHi() {

alert( "Hello" );

}

## alert( sayHi ); // shows the function code

## In JavaScript, a function is a value, so we can deal with it as a value. The code above shows its string representation, which is the source code.

We can copy a function to another variable:

function sayHi() { // (1) create

alert( "Hello" );

}

let func = sayHi; // (2) copy

func(); // Hello // (3) run the copy (it works)!

sayHi(); // Hello // this still works too (why wouldn't it)

Here’s what happens above in detail:

1. The Function Declaration (1) creates the function and puts it into the variable named sayHi.
2. Line (2) copies it into the variable func. Please note again: there are no parentheses after sayHi. If there were, then func = sayHi() would write *the result of the call* sayHi() into func, not *the function* sayHi itself.
3. Now the function can be called as both sayHi() and func().

# Arrow functions, the basics

There’s another very simple and concise syntax for creating functions, that’s often better than Function Expressions.

It’s called “arrow functions”, because it looks like this:

let func = (arg1, arg2, ..., argN) => expression;

In other words, it’s the shorter version of:

let func = function(arg1, arg2, ..., argN) {

return expression;

};

Let’s see a concrete example:

let sum = (a, b) => a + b;

/\* This arrow function is a shorter form of:

let sum = function(a, b) {

return a + b;

};

\*/

alert( sum(1, 2) ); // 3

* If we have only one argument, then parentheses around parameters can be omitted, making that even shorter.

For example:

let double = n => n \* 2;

// roughly the same as: let double = function(n) { return n \* 2 }

alert( double(3) ); // 6

* If there are no arguments, parentheses are empty, but they must be present:

let sayHi = () => alert("Hello!");

sayHi();

Arrow functions can be used in the same way as Function Expressions.

For instance, to dynamically create a function:

let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?

() => alert('Hello!') :

() => alert("Greetings!");

welcome();

### **The Object Datatype**

The object data type can contain:

1. An object  
2. An array  
3. A date

# **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** |
| https://www.w3schools.com/js/objectExplained.gif | 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**.

**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).

It is a common practice to declare objects with the const keyword.

Learn more about using const with objects in the chapter: [JS Const](https://www.w3schools.com/js/js_const.asp).

## Object Definition

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

### **Example**

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

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

**Accessing Object Properties**

You can access object properties in two ways:

*objectName.propertyName*

or

*objectName["propertyName"]*

**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;}

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

In the example above, this refers to the **person object**:

**this.firstName** means the **firstName** property of **person**.

**this.lastName** means the **lastName** property of **person**.

What is **this**?

In JavaScript, the this keyword refers to an **object**.

**Which** object depends on how this is being invoked (used or called).

The this keyword refers to different objects depending on how it is used:

# **JavaScript Arrays**

An array is a special variable, which can hold more than one value:

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

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

You can also create an array, and then provide the elements:

### **Example**

const cars = [];  
cars[0]= "Saab";  
cars[1]= "Volvo";  
cars[2]= "BMW";

## Using the JavaScript Keyword new

The following example also creates an Array, and assigns values to it:

### **Example**

const cars = new Array("Saab", "Volvo", "BMW");

The two examples above do exactly the same.

There is no need to use new Array().

For simplicity, readability and execution speed, use the array literal method.

Accessing Array Elements

You access an array element by referring to the **index number**:

const cars = ["Saab", "Volvo", "BMW"];  
let car = cars[0];

Access entire array:

<script>

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

document.write(cars);

</script>

**What are Cookies?**

Cookies are data, stored in small text files, on your computer.

When a web server has sent a web page to a browser, the connection is shut down, and the server forgets everything about the user.

Cookies were invented to solve the problem "how to remember information about the user":

* When a user visits a web page, his/her name can be stored in a cookie.
* Next time the user visits the page, the cookie "remembers" his/her name.

Cookies are saved in name-value pairs like:

username = John Doe

## Create a Cookie with JavaScript

JavaScript can create, read, and delete cookies with the document.cookie property.

With JavaScript, a cookie can be created like this:

document.cookie = "username=John Doe";

You can also add an expiry date (in UTC time). By default, the cookie is deleted when the browser is closed:

document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC";

With a path parameter, you can tell the browser what path the cookie belongs to. By default, the cookie belongs to the current page.

document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";

## Read a Cookie with JavaScript

With JavaScript, cookies can be read like this:

let x = document.cookie;

## document.cookie will return all cookies in one string much like: cookie1=value; cookie2=value; cookie3=value;

## Change a Cookie with JavaScript

With JavaScript, you can change a cookie the same way as you create it:

document.cookie = "username=John Smith; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";

The old cookie is overwritten.

## Delete a Cookie with JavaScript

Deleting a cookie is very simple.

You don't have to specify a cookie value when you delete a cookie.

Just set the expires parameter to a past date:

document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";

You should define the cookie path to ensure that you delete the right cookie.

Some browsers will not let you delete a cookie if you don't specify the path.

## The Cookie String

The document.cookie property looks like a normal text string. But it is not.

Even if you write a whole cookie string to document.cookie, when you read it out again, you can only see the name-value pair of it.

If you set a new cookie, older cookies are not overwritten. The new cookie is added to document.cookie, so if you read document.cookie again you will get something like:

cookie1 = value; cookie2 = value;

## If you want to find the value of one specified cookie, you must write a JavaScript function that searches for the cookie value in the cookie string.

JavaScript Cookie Example

In the example to follow, we will create a cookie that stores the name of a visitor.

The first time a visitor arrives to the web page, he/she will be asked to fill in his/her name. The name is then stored in a cookie.

The next time the visitor arrives at the same page, he/she will get a welcome message.

For the example we will create 3 JavaScript functions:

1. A function to set a cookie value
2. A function to get a cookie value
3. A function to check a cookie value

## A Function to Set a Cookie

First, we create a function that stores the name of the visitor in a cookie variable:

## function setCookie(cname, cvalue, exdays) {   const d = new Date();   d.setTime(d.getTime() + (exdays\*24\*60\*60\*1000));   let expires = "expires="+ d.toUTCString();   document.cookie = cname + "=" + cvalue + ";" + expires + ";path=/"; }

**Example explained:**

The parameters of the function above are the name of the cookie (cname), the value of the cookie (cvalue), and the number of days until the cookie should expire (exdays).

The function sets a cookie by adding together the cookiename, the cookie value, and the expires string.

## A Function to Get a Cookie

Then, we create a function that returns the value of a specified cookie:

### **Example**

function getCookie(cname) {  
  let name = cname + "=";  
  let decodedCookie = decodeURIComponent(document.cookie);  
  let ca = decodedCookie.split(';');  
  for(let i = 0; i <ca.length; i++) {  
    let c = ca[i];  
    while (c.charAt(0) == ' ') {  
      c = c.substring(1);  
    }  
    if (c.indexOf(name) == 0) {  
      return c.substring(name.length, c.length);  
    }  
  }  
  return "";  
}

**Function explained:**

Take the cookiename as parameter (cname).

Create a variable (name) with the text to search for (cname + "=").

Decode the cookie string, to handle cookies with special characters, e.g. '$'

Split document.cookie on semicolons into an array called ca (ca = decodedCookie.split(';')).

Loop through the ca array (i = 0; i < ca.length; i++), and read out each value c = ca[i]).

If the cookie is found (c.indexOf(name) == 0), return the value of the cookie (c.substring(name.length, c.length).

If the cookie is not found, return "".

## A Function to Check a Cookie

Last, we create the function that checks if a cookie is set.

If the cookie is set, it will display a greeting.

If the cookie is not set, it will display a prompt box, asking for the name of the user, and stores the username cookie for 365 days, by calling the setCookie function:

### **Example**

function checkCookie() {  
  let username = getCookie("username");  
  if (username != "") {  
   alert("Welcome again " + username);  
  } else {  
    username = prompt("Please enter your name:", "");  
    if (username != "" && username != null) {  
      setCookie("username", username, 365);  
    }  
  }  
}