**Numbers and dates**

* [« Previous](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators)
* [Next »](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Text_formatting)

This chapter introduces the concepts, objects and functions used to work with and perform calculations using numbers and dates in JavaScript. This includes using numbers written in various bases including decimal, binary, and hexadecimal, as well as the use of the global [Math](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math) object to perform a wide variety of mathematical operations on numbers.

[**Numbers**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#numbers)

In JavaScript, numbers are implemented in [double-precision 64-bit binary format IEEE 754](https://en.wikipedia.org/wiki/Double-precision_floating-point_format) (i.e., a number between ±2^−1022 and ±2^+1023, or about ±10^−308 to ±10^+308, with a numeric precision of 53 bits). Integer values up to ±2^53 − 1 can be represented exactly.

In addition to being able to represent floating-point numbers, the number type has three symbolic values: +[Infinity](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Infinity), -[Infinity](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Infinity), and [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN) (not-a-number).

See also [JavaScript data types and structures](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures) for context with other primitive types in JavaScript.

You can use four types of number literals: decimal, binary, octal, and hexadecimal.

[**Decimal numbers**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#decimal_numbers)

1234567890

42

// Caution when using leading zeros:

0888 // 888 parsed as decimal

0777 // parsed as octal in non-strict mode (511 in decimal)

Copy to Clipboard

Note that decimal literals can start with a zero (0) followed by another decimal digit, but if every digit after the leading 0 is smaller than 8, the number gets parsed as an octal number.

[**Binary numbers**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#binary_numbers)

Binary number syntax uses a leading zero followed by a lowercase or uppercase Latin letter "B" (0b or 0B). If the digits after the 0b are not 0 or 1, the following [SyntaxError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError) is thrown: "Missing binary digits after 0b".

const FLT\_SIGNBIT = 0b10000000000000000000000000000000; // 2147483648

const FLT\_EXPONENT = 0b01111111100000000000000000000000; // 2139095040

const FLT\_MANTISSA = 0B00000000011111111111111111111111; // 8388607

Copy to Clipboard

[**Octal numbers**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#octal_numbers)

The standard syntax for octal numbers is to prefix them with 0o. For example:

const a = 0o10; // 8

Copy to Clipboard

There's also a legacy syntax for octal numbers — by prefixing the octal number with a zero: 0644 === 420 and "\045" === "%". If the digits after the 0 are outside the range 0 through 7, the number will be interpreted as a decimal number.

const n = 0755; // 493

const m = 0644; // 420

Copy to Clipboard

[Strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode) forbids this octal syntax.

[**Hexadecimal numbers**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#hexadecimal_numbers)

Hexadecimal number syntax uses a leading zero followed by a lowercase or uppercase Latin letter "X" (0x or 0X). If the digits after 0x are outside the range (0123456789ABCDEF), the following [SyntaxError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError) is thrown: "Identifier starts immediately after numeric literal".

0xFFFFFFFFFFFFFFFFF // 295147905179352830000

0x123456789ABCDEF // 81985529216486900

0XA // 10

Copy to Clipboard

[**Exponentiation**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#exponentiation)

1E3 // 1000

2e6 // 2000000

0.1e2 // 10

Copy to Clipboard

[**Number object**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#number_object)

The built-in [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number) object has properties for numerical constants, such as maximum value, not-a-number, and infinity. You cannot change the values of these properties and you use them as follows:

const biggestNum = Number.MAX\_VALUE;

const smallestNum = Number.MIN\_VALUE;

const infiniteNum = Number.POSITIVE\_INFINITY;

const negInfiniteNum = Number.NEGATIVE\_INFINITY;

const notANum = Number.NaN;

Copy to Clipboard

You always refer to a property of the predefined Number object as shown above, and not as a property of a Number object you create yourself.

The following table summarizes the Number object's properties.

| **Property** | **Description** |
| --- | --- |
| [Number.MAX\_VALUE](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MAX_VALUE) | The largest positive representable number (1.7976931348623157e+308) |
| [Number.MIN\_VALUE](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MIN_VALUE) | The smallest positive representable number (5e-324) |
| [Number.NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/NaN) | Special "not a number" value |
| [Number.NEGATIVE\_INFINITY](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/NEGATIVE_INFINITY) | Special negative infinite value; returned on overflow |
| [Number.POSITIVE\_INFINITY](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/POSITIVE_INFINITY) | Special positive infinite value; returned on overflow |
| [Number.EPSILON](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/EPSILON) | Difference between 1 and the smallest value greater than 1 that can be represented as a [Number](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number) (2.220446049250313e-16) |
| [Number.MIN\_SAFE\_INTEGER](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MIN_SAFE_INTEGER) | Minimum safe integer in JavaScript (−2^53 + 1, or −9007199254740991) |
| [Number.MAX\_SAFE\_INTEGER](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MAX_SAFE_INTEGER) | Maximum safe integer in JavaScript (+2^53 − 1, or +9007199254740991) |

| **Method** | **Description** |
| --- | --- |
| [Number.parseFloat()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/parseFloat) | Parses a string argument and returns a floating point number. Same as the global [parseFloat()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseFloat) function. |
| [Number.parseInt()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/parseInt) | Parses a string argument and returns an integer of the specified radix or base. Same as the global [parseInt()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt) function. |
| [Number.isFinite()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/isFinite) | Determines whether the passed value is a finite number. |
| [Number.isInteger()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/isInteger) | Determines whether the passed value is an integer. |
| [Number.isNaN()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/isNaN) | Determines whether the passed value is [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN). More robust version of the original global [isNaN()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN). |
| [Number.isSafeInteger()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/isSafeInteger) | Determines whether the provided value is a number that is a *safe integer*. |

The Number prototype provides methods for retrieving information from Number objects in various formats. The following table summarizes the methods of Number.prototype.

| **Method** | **Description** |
| --- | --- |
| [toExponential()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toExponential) | Returns a string representing the number in exponential notation. |
| [toFixed()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) | Returns a string representing the number in fixed-point notation. |
| [toPrecision()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toPrecision) | Returns a string representing the number to a specified precision in fixed-point notation. |

[**Math object**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#math_object)

The built-in [Math](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math) object has properties and methods for mathematical constants and functions. For example, the Math object's PI property has the value of pi (3.141…), which you would use in an application as

Math.PI

Copy to Clipboard

Similarly, standard mathematical functions are methods of Math. These include trigonometric, logarithmic, exponential, and other functions. For example, if you want to use the trigonometric function sine, you would write

Math.sin(1.56)

Copy to Clipboard

Note that all trigonometric methods of Math take arguments in radians.

The following table summarizes the Math object's methods.

| Methods of Math | |
| --- | --- |
| **Method** | **Description** |
| [abs()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/abs) | Absolute value |
| [sin()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/sin), [cos()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/cos), [tan()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/tan) | Standard trigonometric functions; with the argument in radians. |
| [asin()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/asin), [acos()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/acos), [atan()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/atan), [atan2()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/atan2) | Inverse trigonometric functions; return values in radians. |
| [sinh()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/sinh), [cosh()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/cosh), [tanh()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/tanh) | Hyperbolic functions; argument in hyperbolic angle. |
| [asinh()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/asinh), [acosh()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/acosh), [atanh()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/atanh) | Inverse hyperbolic functions; return values in hyperbolic angle. |
| [pow()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/pow), [exp()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/exp), [expm1()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/expm1), [log()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/log), [log10()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/log10), [log1p()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/log1p), [log2()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/log2) | Exponential and logarithmic functions. |
| [floor()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/floor), [ceil()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/ceil) | Returns the largest/smallest integer less/greater than or equal to an argument. |
| [min()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/min), [max()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/max) | Returns the minimum or maximum (respectively) value of a comma separated list of numbers as arguments. |
| [random()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random) | Returns a random number between 0 and 1. |
| [round()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/round), [fround()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/fround), [trunc()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/trunc), | Rounding and truncation functions. |
| [sqrt()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/sqrt), [cbrt()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/cbrt), [hypot()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/hypot) | Square root, cube root, Square root of the sum of square arguments. |
| [sign()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/sign) | The sign of a number, indicating whether the number is positive, negative or zero. |
| [clz32()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/clz32), [imul()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/imul) | Number of leading zero bits in the 32-bit binary representation. The result of the C-like 32-bit multiplication of the two arguments. |

Unlike many other objects, you never create a Math object of your own. You always use the built-in Math object.

[**BigInts**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#bigints)

One shortcoming of number values is they only have 64 bits. In practice, due to using IEEE 754 encoding, they cannot represent any integer larger than [Number.MAX\_SAFE\_INTEGER](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MAX_SAFE_INTEGER) (which is 253 - 1) accurately. To solve the need of encoding binary data and to interoperate with other languages that offer wide integers like i64 (64-bit integers) and i128 (128-bit integers), JavaScript also offers another data type to represent *arbitrarily large integers*: [BigInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt).

A BigInt can be defined as an integer literal suffixed by n:

const b1 = 123n;

// Can be arbitrarily large.

const b2 = -1234567890987654321n;

Copy to Clipboard

BigInts can also be constructed from number values or string values using the [BigInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt/BigInt) constructor.

const b1 = BigInt(123);

// Using a string prevents loss of precision, since long number

// literals don't represent what they seem like.

const b2 = BigInt("-1234567890987654321");

Copy to Clipboard

Conceptually, a BigInt is just an arbitrarily long sequence of bits which encodes an integer. You can safely do any arithmetic operations without losing precision or over-/underflowing.

const integer = 12 \*\* 34; // 4.9222352429520264e+36; only has limited precision

const bigint = 12n \*\* 34n; // 4922235242952026704037113243122008064n

Copy to Clipboard

Compared to numbers, BigInt values yield higher precision when representing large *integers*; however, they cannot represent *floating-point numbers*. For example, division would round to zero:

const bigintDiv = 5n / 2n; // 2n, because there's no 2.5 in BigInt

Copy to Clipboard

Math functions cannot be used on BigInt values. There is [an open proposal](https://github.com/tc39/proposal-bigint-math) to overload certain Math functions like Math.max() to allow BigInt values.

Choosing between BigInt and number depends on your use-case and your input's range. The precision of numbers should be able to accommodate most day-to-day tasks already, and BigInts are most suitable for handling binary data.

Read more about what you can do with BigInt values in the [Expressions and Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#bigint_operators) section, or the [BigInt reference](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt).

[**Date object**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#date_object)

JavaScript does not have a date data type. However, you can use the [Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date) object and its methods to work with dates and times in your applications. The Date object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.

JavaScript handles dates similarly to Java. The two languages have many of the same date methods, and both languages store dates as the number of milliseconds since January 1, 1970, 00:00:00, with a Unix Timestamp being the number of seconds since January 1, 1970, 00:00:00.

The Date object range is -100,000,000 days to 100,000,000 days relative to 01 January, 1970 UTC.

To create a Date object:

const dateObjectName = new Date([parameters]);

Copy to Clipboard

where dateObjectName is the name of the Date object being created; it can be a new object or a property of an existing object.

Calling Date without the new keyword returns a string representing the current date and time.

The parameters in the preceding syntax can be any of the following:

* Nothing: creates today's date and time. For example, today = new Date();.
* A string representing a date in the following form: "Month day, year hours:minutes:seconds." For example, let Xmas95 = new Date("December 25, 1995 13:30:00"). If you omit hours, minutes, or seconds, the value will be set to zero.
* A set of integer values for year, month, and day. For example, let Xmas95 = new Date(1995, 11, 25).
* A set of integer values for year, month, day, hour, minute, and seconds. For example, let Xmas95 = new Date(1995, 11, 25, 9, 30, 0);.

[**Methods of the Date object**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#methods_of_the_date_object)

The Date object methods for handling dates and times fall into these broad categories:

* "set" methods, for setting date and time values in Date objects.
* "get" methods, for getting date and time values from Date objects.
* "to" methods, for returning string values from Date objects.
* parse and UTC methods, for parsing Date strings.

With the "get" and "set" methods you can get and set seconds, minutes, hours, day of the month, day of the week, months, and years separately. There is a getDay method that returns the day of the week, but no corresponding setDay method, because the day of the week is set automatically. These methods use integers to represent these values as follows:

* Seconds and minutes: 0 to 59
* Hours: 0 to 23
* Day: 0 (Sunday) to 6 (Saturday)
* Date: 1 to 31 (day of the month)
* Months: 0 (January) to 11 (December)
* Year: years since 1900

For example, suppose you define the following date:

const Xmas95 = new Date('December 25, 1995');

Copy to Clipboard

Then Xmas95.getMonth() returns 11, and Xmas95.getFullYear() returns 1995.

The getTime and setTime methods are useful for comparing dates. The getTime method returns the number of milliseconds since January 1, 1970, 00:00:00 for a Date object.

For example, the following code displays the number of days left in the current year:

const today = new Date();

const endYear = new Date(1995, 11, 31, 23, 59, 59, 999); // Set day and month

endYear.setFullYear(today.getFullYear()); // Set year to this year

const msPerDay = 24 \* 60 \* 60 \* 1000; // Number of milliseconds per day

let daysLeft = (endYear.getTime() - today.getTime()) / msPerDay;

daysLeft = Math.round(daysLeft); //returns days left in the year

Copy to Clipboard

This example creates a Date object named today that contains today's date. It then creates a Date object named endYear and sets the year to the current year. Then, using the number of milliseconds per day, it computes the number of days between today and endYear, using getTime and rounding to a whole number of days.

The parse method is useful for assigning values from date strings to existing Date objects. For example, the following code uses parse and setTime to assign a date value to the ipoDate object:

const ipoDate = new Date();

ipoDate.setTime(Date.parse('Aug 9, 1995'));

Copy to Clipboard

[**Example**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Numbers_and_dates#example)

In the following example, the function JSClock() returns the time in the format of a digital clock.

function JSClock() {

const time = new Date();

const hour = time.getHours();

const minute = time.getMinutes();

const second = time.getSeconds();

let temp = String(hour % 12);

if (temp === "0") {

temp = "12";

}

temp += (minute < 10 ? ":0" : ":") + minute;

temp += (second < 10 ? ":0" : ":") + second;

temp += hour >= 12 ? " P.M." : " A.M.";

return temp;

}

Copy to Clipboard

The JSClock function first creates a new Date object called time; since no arguments are given, time is created with the current date and time. Then calls to the getHours, getMinutes, and getSeconds methods assign the value of the current hour, minute, and second to hour, minute, and second.

The following statements build a string value based on the time. The first statement creates a variable temp. Its value is hour % 12, which is hour in the 12-hour system. Then, if the hour is 0, it gets re-assigned to 12, so that midnights and noons are displayed as 12:00 instead of 0:00.

The next statement appends a minute value to temp. If the value of minute is less than 10, the conditional expression adds a string with a preceding zero; otherwise it adds a string with a demarcating colon. Then a statement appends a seconds value to temp in the same way.

Finally, a conditional expression appends "P.M." to temp if hour is 12 or greater; otherwise, it appends "A.M." to temp.