Numbers and dates

Numbers

In JavaScript, numbers are implemented in double-precision 64-bit binary format IEEE 754 (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, -Infinity, and NaN (not-a-number).

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

Decimal numbers

```
1234567890
42
```

Decimal literals can start with a zero (0) followed by another decimal digit, but if all digits after the leading 0 are smaller than 8, the number is interpreted as an octal number. This is considered a legacy syntax, and number literals prefixed with 0, whether interpreted as octal or decimal, cause a syntax error in strict mode — so, use the 0o prefix instead.

```
0888 // 888 parsed as decimal
0777 // parsed as octal, 511 in decimal
```

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 is thrown: "Missing binary digits after 0b".

```
0b10000000000000000000000000000000 // 2147483648
0b01111111100000000000000000000000 // 2139095040
0B00000000011111111111111111111111 // 8388607
```

Octal numbers

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

```
0O755 // 493
0o644 // 420
```

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
```

Strict mode forbids this octal syntax.

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 is thrown: "Identifier starts immediately after numeric literal".

```
0xFFFFFFFFFFFFFFFF // 295147905179352830000
0x123456789ABCDEF   // 81985529216486900
0XA                 // 10
```

Exponentiation

```
0e-5   // 0
```

```
0e+5   // 0
5e1    // 50
175e-2 // 1.75
1e3    // 1000
1e-3   // 0.001
1E3    // 1000
```

Number object

In JavaScript, the `Number` `object` is a **built-in object** that provides a way to work with numerical values. It provides a set of methods and properties that can be used to perform mathematical operations and manipulate numbers in various ways.

Here are some of the most commonly used methods and properties of the `Number` object:

1. `Number()`: This method is used to convert a value to a number. For example:

```
Number("10"); // returns 10
Number("10.5"); // returns 10.5
```

2. `isNaN()`: This method is used to check whether a value is NaN (Not-a-Number). For example:

```
isNaN("hello"); // returns true
isNaN(10); // returns false
```

3. `parseInt()`: This method is used to convert a string to an integer. For example:

```
parseInt("10"); // returns 10
parseInt("10.5"); // returns 10
```

4. `parseFloat()`: This method is used to convert a string to a floating-point number. For example:

```
parseFloat("10.5"); // returns 10.5
parseFloat("10"); // returns 10
```

5. `toFixed()`: This method is used to round a number to a specified number of decimal places and return the result as a string. For example:

```
let num = 10.5678;
num.toFixed(2); // returns "10.57"
```

6. `toString()`: This method is used to convert a number to a string. For example:

```
let num = 10;
num.toString(); // returns "10"
```

7. `MAX_VALUE` and `MIN_VALUE`: These properties represent the largest and smallest possible values for a number in JavaScript. For example:

```
Number.MAX_VALUE; // returns 1.7976931348623157e+308
Number.MIN_VALUE; // returns 5e-324
```

These are just a few examples of the methods and properties available on the `Number` object in JavaScript. The `Number` object can be a powerful tool for working with numerical data in your JavaScript code.

Math object

The `Math` object in JavaScript is a built-in object that provides a set of *properties* and *methods* for performing mathematical operations. It is not a constructor, and its methods and properties are static, which means they can be accessed directly without creating an instance of the object.

Some of the properties of the `Math` object include:

- `Math.PI`: represents the ratio of the circumference of a circle to its diameter, which is approximately 3.141592653589793.
- `Math.E`: represents Euler's number, which is approximately 2.718281828459045.

- `Math.LN2`: represents the natural logarithm of 2, which is approximately 0.6931471805599453.

The `Math` object also provides a set of methods for performing mathematical operations, such as:

- `Math.abs(x)`: returns the absolute value of a number `x`.
- `Math.floor(x)`: returns the largest integer less than or equal to a number `x`.
- `Math.max(x1, x2, ..., xn)`: returns the largest of the given numbers.
- `Math.pow(x, y)`: returns the result of raising `x` to the power of `y`.
- `Math.random()`: returns a random number between 0 and 1.

Here's an example of using the `Math` object to generate a random number between two values:

```
let min = 1; let max = 10;
let randomNumber = Math.floor(Math.random() * (max - min + 1) + min);
console.log(randomNumber);
```

This code generates a random number between 1 and 10 (inclusive) and logs it to the console. The `Math.random()` method returns a random number between 0 and 1, and the `Math.floor()` method is used to round the result down to the nearest integer. The formula `(max - min + 1) + min` is used to generate a random number between `min` and `max`.

BigInts

`BigInt` is a relatively new addition to the JavaScript language that provides a way to represent integers that are larger than the maximum safe integer value of `Number` type. The `BigInt` data type can represent arbitrarily large integers with precision, limited only by the amount of memory available to the JavaScript engine.

To create a `BigInt`, you simply add the letter "`n`" to the end of a numeric literal or use the `BigInt()` constructor:

```
const bigNumber = 1234567890123456789012345678901234567890n;
```

```
const anotherBigNumber = BigInt("12345678901234567890123456789O");
```

Note that you must use the `BigInt()` constructor when converting a string to a `BigInt`.

`BigInt` values can be used with standard arithmetic operators like +, -, *, /, and %, and can also be used in comparison operations like >, <, ==, and !=. However, `BigInt` values cannot be mixed with `Number` values in the same operation, and trying to do so will result in a `TypeError`:

```
const bigNumber = 12345678901234567890123456789On;
const anotherBigNumber = BigInt("98765432109876543210987654321O");
const sum = bigNumber + anotherBigNumber; // OK
const product = bigNumber * 2; // TypeError!
```

To work with `BigInt` values, you may need to use functions and libraries that support them. For example, the `Math` object does not provide functions for working with `BigInt` values. Some popular libraries that support `BigInt` include `big-integer`, `bigint-crypto-utils`, and `jsbn`.

Date object

`Date` object in JavaScript provides a way to work with dates and times. It represents a single moment in time and can be used to perform various operations on dates such as formatting, parsing, and arithmetic. Date object has a large number of methods, but it does not have any properties.

To create a new `Date` object, you can use one of the following syntaxes:

```
const now = new Date(); // current date and time
const dateFromString = new Date("2022-03-11T15:00:00Z"); // date from ISO string
const dateFromTimestamp = new Date(1647067200000); // date from Unix timestamp
const dateFromParts = new Date(2022, 2, 11, 15, 0, 0); // date from individual components
```

The first example creates a `Date` object representing the current date and time. The second example creates a `Date` object from an ISO date/time string. The third example creates a `Date` object from a Unix timestamp, which represents the number of milliseconds since January 1,

1970, 00:00:00 UTC. The fourth example creates a `Date` object from individual year, month, day, hour, minute, and second components.

The `Date` object provides a variety of methods for working with dates and times, including:

- `getFullYear()`: Returns the year of the date as a four-digit number (e.g. 2022).
- `getMonth()`: Returns the month of the date as a zero-based index (0 for January, 1 for February, etc.).
- `getDate()`: Returns the day of the month as a number (1-31).
- `getDay()`: Returns the day of the week as a zero-based index (0 for Sunday, 1 for Monday, etc.).
- `getHours()`: Returns the hour of the day as a number (0-23).
- `getMinutes()`: Returns the minute of the hour as a number (0-59).
- `getSeconds()`: Returns the second of the minute as a number (0-59).
- `getTime()`: Returns the Unix timestamp of the date in milliseconds.
- `toLocaleString()`: Returns a string representing the date in a human-readable format, according to the user's locale.
- `toISOString()`: Returns an ISO-formatted string representing the date (e.g. "2022-03-11T15:00:00.000Z").

You can also perform arithmetic operations on `Date` objects, such as adding or subtracting days, hours, or minutes:

```
const date = new Date("2022-03-11T15:00:00Z");
date.setDate(date.getDate() + 1); // add one day
date.setHours(date.getHours() - 2); // subtract two hours
```

This code creates a `Date` object representing March 11th, 2022 at 3:00pm UTC. It then adds one day to the date and subtracts two hours, resulting in a new `Date` object representing March 12th, 2022 at 1:00pm UTC.