Numbers,Dates ,Timers etc..

String to Number

JavaScript provides multiple ways to convert a string into a number. Some common approaches include:

a. Number() Constructor

- Converts any value to a number.
- Returns NaN if the string cannot be converted.

```
console.log(Number("123"));       // Output: 123
console.log(Number("123.45"));    // Output: 123.45
console.log(Number("abc"));       // Output: NaN
```

b. parseInt() and parseFloat()

- parseInt parses a string into an integer, ignoring decimals.
- parseFloat parses a string into a floating-point number.

```
console.log(parseInt("123.45"));    // Output: 123
console.log(parseFloat("123.45")); // Output: 123.45
console.log(parseInt("123abc"));   // Output: 123 (parses up to non-numeric part)
```

c. Unary + Operator

- A shorthand way to convert a string to a number.

```
console.log(+"123");      // Output: 123
console.log(+"123.45");   // Output: 123.45
console.log(+"abc");      // Output: NaN
```

Number to String

Converting numbers into strings is straightforward using:

a. String() Constructor

- Converts any value into a string.

```
console.log(String(123));      // Output: "123"
console.log(String(123.45));   // Output: "123.45"
```

b. **toString() Method**

- A method available for numbers.
- Does not work for null or undefined.

```
console.log((123).toString());      // Output: "123"
console.log((123.45).toString());   // Output: "123.45"
```

c. Template Literals

- Using backticks ${} to embed numbers in strings.

```
let num = 123;
console.log(`${num}`); // Output: "123"
```

d. **String Concatenation**

- Adding an empty string to a number converts it into a string.

```
console.log(123 + "");    // Output: "123"
console.log(123.45 + ""); // Output: "123.45"
```

parseInt()

parseInt(string, radix)

- Parses the string and converts it to an integer.
- radix specifies the base (e.g., base 10 for decimal).
- Ignore trailing non-numeric characters.

Examples:

```
console.log(parseInt("123"));        // 123 (base 10)
console.log(parseInt("123", 8));     // 83 (base 8)
console.log(parseInt("0xF", 16));    // 15 (hexadecimal)
console.log(parseInt("abc123"));     // NaN
console.log(parseInt("101", 2));   // Output: 5 (binary to decimal)
```

Practical Use Case

Extracting an integer from user input:

```
let userInput = "42px";
let size = parseInt(userInput);
console.log(size); // Output: 42
```

isNaN()

- Checks if a value is NaN (Not-a-Number).
- Returns true for NaN and false otherwise.

```
console.log(isNaN("abc"));   // Output: true
console.log(isNaN(123));     // Output: false
console.log(isNaN(NaN));     // Output: true
console.log(isNaN(undefined)); // true
console.log(isNaN("123"));   // false
```

Practical Use Case

Validating numeric input:

```
let userInput = "abc";
if (isNaN(userInput)) {
   console.log("Please enter a valid number.");
}
```

isFinite()

- Checks if a value is a finite number.
- Returns false for NaN, Infinity, and -Infinity.

```
console.log(isFinite(123));      // Output: true
console.log(isFinite("123"));    // Output: true (string is converted)
console.log(isFinite(NaN));      // Output: false
console.log(isFinite(Infinity)); // Output: false
```

Practical Use Case

Avoiding calculations with infinite or invalid numbers:

```
let num = 1 / 0;
if (!isFinite(num)) {
   console.log("Invalid calculation.");
}
```

Number.isInteger()

- Checks if a value is an integer.
- Does not convert types.

```
console.log(Number.isInteger(123));      // Output: true
console.log(Number.isInteger(123.45));   // Output: false
console.log(Number.isInteger("123"));    // Output: false
```

Practical Use Case

Validating age input:

```
let age = 25.5;
if (Number.isInteger(age)) {
   console.log("Valid age.");
} else {
   console.log("Age must be a whole number.");
}
```

Math and Rounding

1. Math.sqrt()

The Math.sqrt() method returns the square root of a given number.

- **Input:** A non-negative number.
- **Output:** The square root of the number. For negative numbers, it returns NaN.

Example:

```
console.log(Math.sqrt(16)); // Output: 4
console.log(Math.sqrt(25)); // Output: 5
console.log(Math.sqrt(-1)); // Output: NaN (square root of a negative number is not defined)
```

```
console.log(Math.sqrt(16)); //4
console.log(Math.sqrt(null)); //0
console.log(Math.sqrt(undefined)); //NaN
console.log(Math.sqrt(NaN)); //NaN
console.log(Math.sqrt(' ')); //0
```

---

2. Math.max()

The Math.max() method returns the largest value from a set of numbers.

- **Input:** A list of numbers.
- **Output:** The largest number. If no arguments are passed, it returns -Infinity.

Example:

```
console.log(Math.max(5, 10, 15, 20)); // Output: 20
console.log(Math.max(-5, -10, -15));  // Output: -5
console.log(Math.max());              // Output: -Infinity
```

---

3. Math.min()

The Math.min() method returns the smallest value from a set of numbers.

- **Input:** A list of numbers.
- **Output:** The smallest number. If no arguments are passed, it returns Infinity.

Example:

```
console.log(Math.min(5, 10, 15, 20)); // Output: 5
console.log(Math.min(-5, -10, -15));  // Output: -15
console.log(Math.min());              // Output: Infinity
```

---

4. Math.PI

The Math.PI property represents the ratio of a circle's circumference to its diameter, approximately 3.14159.

Example:

```
console.log(Math.PI); // Output: 3.141592653589793

// Calculate the circumference of a circle with radius 5
const radius = 5;
const circumference = 2 * Math.PI * radius;
console.log(circumference); // Output: 31.41592653589793
```

---

5. Math.random()

The Math.random() method generates a pseudo-random number between **0 (inclusive) and 1** (exclusive).

- Use it to generate random integers by scaling and truncating.

Example:

```
console.log(Math.random()); // Output: Random number between 0 and 1
console.log(Math.random() * 10); // Output: Random number between 0 and 10

// Generate a random integer between 1 and 100
const randomInt = Math.floor(Math.random() * 100) + 1;
console.log(randomInt); // Output: Random integer between 1 and 100
```

---

6. Math.trunc()

The Math.trunc() method removes the fractional part of a number and returns the integer part.

- It does not round the number, simply truncates it.

Example:

```
console.log(Math.trunc(4.9));   // Output: 4
console.log(Math.trunc(-4.9));  // Output: -4
console.log(Math.trunc(0.5));   // Output: 0
```

---

7. Math.floor()

1. For **positive numbers**, it simply truncates the decimal part.
   Example: Math.floor(4.9) → 4
2. For **negative numbers**, it rounds away from zero to the next smaller integer.
   Example: Math.floor(-4.9) → -5

Example:

```
// Positive numbers
console.log(Math.floor(4.9));  // Output: 4
console.log(Math.floor(4.1));  // Output: 4
console.log(Math.floor(0.9));  // Output: 0

// Negative numbers
console.log(Math.floor(-4.9)); // Output: -5
console.log(Math.floor(-4.1)); // Output: -5
console.log(Math.floor(-0.9)); // Output: -1

// Edge cases
console.log(Math.floor(0));    // Output: 0
console.log(Math.floor(-0));   // Output: -0
```

---

8. Math.round()

The Math.round() method rounds a number to the nearest integer.

- Values of .5 or higher round up, others round down.

Example:

```
console.log(Math.round(4.5));   // Output: 5
console.log(Math.round(4.4));   // Output: 4
console.log(Math.round(-4.5));  // Output: -4
```

---

9. Math.ceil()

The Math.ceil() method in JavaScript **rounds a number up** to the nearest integer. It always rounds towards positive infinity, regardless of whether the number is positive or negative.

1. For **positive numbers**, it rounds up to the next largest integer.
   Example: Math.ceil(4.1) → 5
2. For **negative numbers**, it rounds "up" towards zero (less negative).
   Example: Math.ceil(-4.9) → -4

Example:

```
// Positive numbers
console.log(Math.ceil(4.1));  // Output: 5
console.log(Math.ceil(4.9));  // Output: 5
console.log(Math.ceil(0.1));  // Output: 1

// Negative numbers
console.log(Math.ceil(-4.1)); // Output: -4
console.log(Math.ceil(-4.9)); // Output: -4
console.log(Math.ceil(-0.1)); // Output: 0

// Edge cases
console.log(Math.ceil(0));    // Output: 0
console.log(Math.ceil(-0));   // Output: -0
```

---

10. toFixed()

The toFixed() method formats a number to a fixed number of decimal places.

- **Input:** An integer specifying the number of decimal places.
- **Output:** A string representation of the number.

The result is always a string, so you may need to convert it back to a number if necessary:

Example:

Rounding Rules:

- Values ≥ .5 round **up**.
- Values < .5 round **down**.

For positive numbers, toFixed() rounds up or down to the specified decimal places as per the rounding rules (≥ .5 rounds up, < .5 rounds down).

```
// Positive numbers
const positiveNumber1 = 5.6789;
const positiveNumber2 = 5.1234;

console.log(positiveNumber1.toFixed(2));  // Output: "5.68" (rounded up)
console.log(positiveNumber2.toFixed(2));  // Output: "5.12" (rounded down)
```

- **5.6789** → Rounded to two decimal places, it becomes "5.68" because the third decimal is 8 (≥ .5, rounds up).
- **5.1234** → Rounded to two decimal places, it becomes "5.12" because the third decimal is 3 (< .5, rounds down).

Negative Numbers

For negative numbers, toFixed() still follows the rounding rules, but since it's rounding towards zero, the behavior might seem different (rounds "up" towards zero).

```
// Negative numbers
const negativeNumber1 = -5.6789;
const negativeNumber2 = -5.1234;

console.log(negativeNumber1.toFixed(2));  // Output: "-5.68" (rounded up)
console.log(negativeNumber2.toFixed(2));  // Output: "-5.12" (rounded down)
```

- **-5.6789** → Rounded to two decimal places, it becomes "-5.68" because the third decimal is 8 (≥ .5, rounds up towards zero).
- **-5.1234** → Rounded to two decimal places, it becomes "-5.12" because the third decimal is 3 (< .5, rounds down).

Padding with Zeros:

If the specified digits is greater than the number of actual decimal places, the result will be padded with 0s:

Formatting Prices or Currency

Ensure consistent formatting when displaying prices:

```
const price = 19.9;
```

```
console.log(`$${price.toFixed(2)}`); // Output: "$19.90"
```

Displaying Percentage Values

Convert fractions to percentage values with two decimal places:

```
const fraction = 0.12345;
console.log(`${(fraction * 100).toFixed(2)}%`); // Output: "12.35%"
```

Rounding to Fixed Precision for Calculations

Use toFixed() to round values for comparison or further processing:

```
const result = (10 / 3).toFixed(2);
console.log(result); // Output: "3.33"
```

Formatting User Input

Ensure numerical input adheres to a specific decimal format:

```
let input = 45.6789;
console.log(parseFloat(input.toFixed(1))); // Output:

console.log((5).toFixed(2)); // Output: "5.00"
console.log((123.4).toFixed(4)); // Output: "123.4

// Converting back to a number
const fixedNum = parseFloat(num.toFixed(2));
console.log(fixedNum); // Output: 123.46
```

Remainder Operator

```
console.log(5 % 2); //1
console.log(5 / 2); // output =2.5 //5 = 2 * 2 + 1
console.log(8 % 3); //2
console.log(8 / 3); // 2.6666666666// 8= 3 *2 + 2
console.log(7 % 2); //1
console.log(7 / 2); //3.5

const isEven = num => (num % 2 === 0 ? 'Even' : 'Odd');
```

```
console.log(isEven(22)); //even
console.log(isEven(23)); //odd
```

Numeric Separator

The numeric separator (_) is a feature in JavaScript that allows you to make large numbers more readable by separating groups of digits. This feature was introduced in ECMAScript 2021 (ES12) and works similarly to how commas or spaces are used to separate large numbers in some cultures.

Purpose

The main purpose of the numeric separator is to improve t**he readability of large numbers by inserting underscores between groups of digits without affecting the actual value of the number.**

const num = 1_000_000; //1000000

Valid Use Cases

You can insert the numeric separator anywhere between digits of a numeric literal, including:

- Between digits in integers.
- Between digits in floating-point numbers.
- Between digits in binary, octal, or hexadecimal numbers.

Rules for Using the Numeric Separator

Cannot Begin or End with an Underscore: You cannot use an underscore at the beginning or end of a number.

const invalidStart = **_1000;** // SyntaxError

const invalidEnd = 1000_; // SyntaxError

Cannot Have Two Underscores Together: There cannot be consecutive underscores within a number.
const invalid = **1__000; // SyntaxError**

No Separators for Decimal Points: You cannot insert an underscore immediately before or after a decimal point.
const invalidFloat = **3._14;** // SyntaxError

```
const billion = 1_000_000_000;
console.log(billion); // Output: 1000000000

const million = 1_000_000;
console.log(million); // Output: 1000000

const pi = 3.141_592_653_589_793;
console.log(pi); // Output: 3.141592653589793

const binary = 0b1010_1101_0011_0100;
console.log(binary); // Output: 43924 (binary value 1010110100110100)

const octal = 0o123_456_701;
console.log(octal); // Output: 34239105 (octal value 123456701)

const hexadecimal = 0xa0_bc_5f_33;
console.log(hexadecimal); // Output: 2674659795 (hexadecimal value A0BC5F33)
```

While JavaScript allows underscores in numeric literals, functions like Number() or parseInt() **do not support underscores in strings** that represent numbers.

Invalid Example with Number()

```
const numWithSeparator = "1_000_000";
const num = Number(numWithSeparator);
console.log(num); // Output: NaN (Not a Number)
```

BigInt

BigInt was officially introduced in ECMAScript 2020 (ES11). It became available in modern JavaScript engines around 2020.

BigInt is a built-in JavaScript type that was introduced to handle arbitrarily large integers, which are beyond the range of the traditional Number type. The Number type in JavaScript can only safely represent integers up to 2^53 - 1 (9007199254740991) or as low as -(2^53 - 1). BigInt allows you to work with integers of any size.

Arbitrary Precision: BigInt can represent integers larger than the limit of Number, allowing for precise calculations with very large integers.

Syntax: You can create a BigInt by appending n to the end of an integer or by using the BigInt() constructor.

const bigIntFromLiteral = 123456789012345678901234567890123456789**0n**; // BigInt literal
const bigIntFromConstructor = BigInt("123456789012345678901234567890123456789"); // BigInt constructor

```
let a = 92345678901234567890123456789012345678901n;
let b = 98765432109876543210987654321098765432109n;
let sum = a + b;
console.log(sum); // 191111111011111111101111111110111111111100n

let c = 98765432109876543210987654321098765432109n;
let d = 12345678901234567890123456789012345678901n;
let difference = c - d;
console.log(difference); // 864197532086419753208642975320864197532 0n

let x = 12345678901234567890123456789012345678901n;
let y = 2n;
console.log(typeof y);//bigint
let product = x * y;
console.log(product); // 24691357802469135780246913578024691357802n

let p = 12345678901234567890123456789012345678901n;
let q = 98765432109876543210987654321098765432109n;
console.log(p < q); // true
console.log(p === q); // false
console.log(p == 12345678901234567890123456789012345678901n); // true
console.log(20n == '20');//true
```

Creating Dates

The Date object is used to work with dates and times. It provides methods for handling dates, times, and time zones, including creating, parsing, formatting, and manipulating date and time values.

We can create a Date object in several ways:

i) Using the new Date() Constructor

Creates a Date object with the current date and time.

```
let now = new Date();
console.log(now);// Mon Dec 16 2024 12:52:11 GMT+0530 (India Standard Time)

// Current date and time
```

ii)Using new Date(year, month, day, hours, minutes, seconds, milliseconds)

Creates a Date object with specific components. Note:

- The year is the full year (e.g., 2024).
- The month is zero-based (0 = January, 11 = December).
- The day is the day of the month (1–31).
- The hours, minutes, seconds, and milliseconds are optional (default to 0).

```
let specificDate = new Date(2024, 11, 25, 10, 30, 15, 500);
console.log(specificDate); // Wed Dec 25 2024 10:30:15 GMT+0530 (India Standard Time)
```

iii)Using new Date(milliseconds)

Creates a Date object using the number of milliseconds since // Jan 01 1970 05:30:00 GMT+0530 (India Standard Time)

```
let epochTime = new Date(0);
console.log(epochTime); // Thu Jan 01 1970 05:30:00 GMT+0530 (India Standard Time)

let oneDayLater = new Date(24 * 60 * 60 * 1000);
console.log(oneDayLater); // Fri Jan 02 1970 05:30:00 GMT+0530 (India Standard Time)
```

iv) Using new Date(dateString)

Creates a Date object from a date string. The format should follow the **ISO 8601** standard or be a format that JavaScript can recognize.

```
let isoDate = new Date('2024-12-25T10:30:00Z');
console.log(isoDate); // Wed Dec 25 2024 16:00:00 GMT+0530 (India Standard Time)
```

```
let simpleDate = new Date('December 25, 2024');
console.log(simpleDate); // Wed Dec 25 2024 00:00:00 GMT+0530 (India Standard Time)
```

v)Using Date.now()

The output 1734334204238 you see from Date.now() represents the **current timestamp** in milliseconds

```
let current = Date.now();
console.log(current);//1734334204238
```

Methods

In JavaScript, the getMonth() method returns the month as a zero-based index:

- 0 corresponds to January.
- 11 corresponds to December.

Thus, to make the month human-readable (where January = 1, February = 2, etc.), you need to add 1 to the result of getMonth().

let now = new Date();
console.log(`Month: ${now.getMonth()}`); // Month: 11 (for December)
console.log(`Month: ${now.getMonth() + 1}`); // Month: 12 (human-readable)

```
let now = new Date();

// Get components
console.log(`Year: ${now.getFullYear()}`); // Year: 2024
console.log(`Month: ${now.getMonth() + 1}`); //Month: 12 (Add 1 to match human-readable format)
console.log(`Date: ${now.getDate()}`); // Date: 16
console.log(`Day: ${now.getDay()}`); // Day: 1 (Monday)
console.log(`Hours: ${now.getHours()}`); //Hours: 13
console.log(`Minutes: ${now.getMinutes()}`); //Minutes: 5
console.log(`Seconds: ${now.getSeconds()}`); //Seconds: 9
console.log(`Milliseconds: ${now.getMilliseconds()}`); // Milliseconds: 932

// Set components
now.setFullYear(2025);
now.setMonth(0); // January
now.setDate(1);
```

```
console.log(`Updated Date: ${now}`); //Updated Date: Wed Jan 01 2025 13:05:27 GMT+0530 (India Standard Time)
```

Formatting date pattern

```
let now = new Date();
let month = now.getMonth() + 1; // Months are 0-based
let day = now.getDate();
let year = now.getFullYear();

let formattedDate = `${month}/${day}/${year}`;
console.log(formattedDate); // e.g., 12/16/2024
```

Using Template Literals for Leading Zeros

If you want to ensure that the month and day are always two digits (e.g., 12/09/2024 instead of 12/9/2024), you can pad them with leading zeros:

.padStart(2, '0')

The .padStart(targetLength, padString) method **pads** the string on the left side with the specified character ('0' in this case) until it reaches the desired length (2 in this case).

How it works:

- If the string is shorter than targetLength, it adds the padString to the left.
- If the string is already at or longer than targetLength, no padding is added.

Examples:

"1".padStart(2, '0'); // "01"

"12".padStart(2, '0'); // "12"

"123".padStart(2, '0'); // "123" (no change, already >= 2 characters)

```
let now = new Date();
let month = String(now.getMonth() + 1).padStart(2, '0');
let day = String(now.getDate()).padStart(2, '0');
let year = now.getFullYear();
```

```
let formattedDate = `${month}/${day}/${year}`;
console.log(formattedDate); // e.g., 12/09/2024
```

Internationalizing dates

Internationalizing dates in JavaScript refers to the process of displaying dates in a way that is appropriate for a specific locale or region. Different cultures and regions may have unique ways of formatting and interpreting dates, such as different date formats (e.g., day/month/year vs. month/day/year), different calendar systems, or even different names for months and days.

JavaScript provides the Intl.DateTimeFormat object, which is part of the Intl (Internationalization) API, to help you format dates based on the user's locale or a specified locale.

http://www.lingoes.net/en/translator/langcode.htm

```
let now = new Date();
//(Telugu (India)
const date1 = new Intl.DateTimeFormat('te-IN').format(now); // 16/12/2024
console.log(date1);

// Get current date
const date = new Date();

// Format the date for a US locale (MM/DD/YYYY)
const usFormatted = new Intl.DateTimeFormat('en-US').format(date);
console.log(usFormatted); // Output: "12/16/2024"

// Format the date for a German locale (DD.MM.YYYY)
const deFormatted = new Intl.DateTimeFormat('de-DE').format(date);
console.log(deFormatted); // Output: "16.12.2024"

// Format the date for a Japanese locale (YYYY/MM/DD)
const jpFormatted = new Intl.DateTimeFormat('ja-JP').format(date);
console.log(jpFormatted); // Output: "2024/12/16"
```

Ex2

```
// Format with options (showing full weekday, month, and year)
const formattedDate = new Intl.DateTimeFormat('en-US', {
  weekday: 'long', // "Monday"
  year: 'numeric', // "2024"
  month: 'long', // "December"
  day: 'numeric', // "16"
}).format(date);

console.log(formattedDate); // Output: "Monday, December 16, 2024"
```

Ex3

```
const num = 56664566.45;
console.log('US : ', new Intl.NumberFormat('en-US').format(num)); //US :  56,664,566.45
console.log('Germany : ', new Intl.NumberFormat('de-DE').format(num)); //Germany :  56.664.566,45
console.log('Syria :', new Intl.NumberFormat('ar-SY').format(num)); //Syria : ٥٦٬٦٦٤٬٥٦٦٫٤٥
```

Ex

```
const options = {
  style: 'currency',
  unit: 'celsius',
  currency: 'INR',
};

console.log('US : ', new Intl.NumberFormat('en-US', options).format(num)); //US :  ₹56,664,566.45
console.log('Germany : ', new Intl.NumberFormat('de-DE', options).format(num)); //Germany :
56.664.566,45 ₹
console.log('Syria :', new Intl.NumberFormat('ar-SY', options).format(num)); //Syria : ٥٦٬٦٦٤٬٥٦٦٫٤٥ ₹
```

Ex

```
const number = 56664566.45;

// 1. Currency Formatting
const currencyOptions = {
  style: 'currency',
  currency: 'USD', // You can replace 'USD' with 'INR', 'EUR', etc.
```

```
  currencyDisplay: 'symbol', // Use 'symbol', 'narrowSymbol', or 'code'
};
console.log(
  'US Currency:',
  new Intl.NumberFormat('en-US', currencyOptions).format(number)
);
// Output: US Currency: $56,664,566.45

console.log(
  'India Currency:',
  new Intl.NumberFormat('en-IN', {
    ...currencyOptions,
    currency: 'INR',
  }).format(number)
);
// Output: India Currency: ₹5,66,64,566.45 (Indian numbering system)

// 2. Percent Formatting
const percentOptions = {
  style: 'percent',
  maximumFractionDigits: 2, // Set decimal places for percentages
};
console.log(
  'Percentage:',
  new Intl.NumberFormat('en-US', percentOptions).format(0.8543)
);
// Output: Percentage: 85.43%

// 3. Unit Formatting (e.g., Celsius, Kilometers)
const unitOptions = {
  style: 'unit',
  unit: 'celsius', // Change to 'kilometer', 'liter', etc.
  unitDisplay: 'long', // 'long', 'short', or 'narrow'
};
console.log(
  'Temperature:',
  new Intl.NumberFormat('en-US', unitOptions).format(25)
);
// Output: Temperature: 25 Celsius
```

```
const distanceOptions = {
  style: 'unit',
  unit: 'kilometer',
  unitDisplay: 'short',
};
console.log(
  'Distance:',
  new Intl.NumberFormat('en-US', distanceOptions).format(12345.67)
);
// Output: Distance: 12,345.67 km

// 4. Custom Number Formatting (Significant Digits)
const customOptions = {
  minimumSignificantDigits: 3,
  maximumSignificantDigits: 5,
};
console.log(
  'Custom Significant Digits:',
  new Intl.NumberFormat('en-US', customOptions).format(number)
);
// Output: Custom Significant Digits: 5.6665e+7

// 5. Locale-Aware Formatting (Different Regions)
console.log(
  'German Currency:',
  new Intl.NumberFormat('de-DE', {
    ...currencyOptions,
    currency: 'EUR',
  }).format(number)
);
// Output: German Currency: 56.664.566,45 €

console.log(
  'Arabic Currency:',
  new Intl.NumberFormat('ar-SY', {
    ...currencyOptions,
    currency: 'SAR',
  }).format(number)
```

```
);
// Output: Arabic Currency: ٥٦٬٦٦٤٬٥٦٦٫٤٥ ر.س

// 6. Using Browser's Locale
console.log(
  'Browser Locale:',
  new Intl.NumberFormat(navigator.language, currencyOptions).format(number)
);
// Output varies based on the user's browser settings
```

Timers

Timers in JavaScript allow you to execute code after a specified delay or repeatedly at fixed intervals. JavaScript provides two main functions for working with timers:

1) setTimeout

Executes a function after a specified delay (in milliseconds). This is useful for delayed or one-time execution.

The setTimeout() function helps the users to delay the execution of code.

Syntax:

let timeoutId = setTimeout(function, delay, arg1, arg2, ...);

function: The function to execute after the delay.
delay: Time in milliseconds (1 second = 1000 milliseconds).
arg1, arg2, ...: Optional arguments passed to the function when it executes.
timeoutId: Identifier that can be used to cancel the timer.
Ex1:
Without argos

```
setTimeout(() => { console.log('Hello after 2 seconds!'); } , 2000); //Hello after 2 seconds!(after 2 sec will get results)
```

With args

```
let zomato = setTimeout((food, price) => { console.log(`The ${food} is ${price}`); },3000,'biryani',200);
//The biryani is 200(after 3 sec will get results)
```

Cancelling setTimeout:

Use clearTimeout with the timeoutId returned by setTimeout:

```
let zomato = setTimeout(
  (food, price) => {
    console.log(`The ${food} is ${price}`);  },3000,'biryani', 200);
clearTimeout(zomato);
```

2) setInterval

Executes a function repeatedly at specified intervals (in milliseconds).

Syntax:

let intervalId = setInterval(function, interval, arg1, arg2, ...);

function: The function to execute repeatedly.

interval: Time in milliseconds between function executions.

arg1, arg2, ...: Optional arguments passed to the function when it executes.

intervalId: Identifier that can be used to cancel the timer.

```
let count = 0;

let intervalId = setInterval(() => {

  count++;

  console.log(`Count: ${count}`); //for every  1 sec it will  print until we stop.

}, 1000);
```

It will repeat to print until value ===5 and once it reaches condition we are stopping using clearInterval(intervalId);

```
let count = 0;

let intervalId = setInterval(() => {

 count++;

 console.log(`Count: ${count}`);

 if (count === 5) clearInterval(intervalId);

}, 1000);
```

Count: 1

Count: 2

Count: 3

Count: 4

Count: 5

```
let intervalId = setInterval(() => console.log('Repeating'), 1000);

setTimeout(() => clearInterval(intervalId), 5000); // Stops after 5 seconds
```

countdown timer:

Real-World Use Case

Simulate a countdown timer:

```
let countdown = 10;

let timer = setInterval(() => {

 console.log(countdown);
```

```
  countdown--;

  if (countdown < 0) {

    clearInterval(timer);

    console.log("Time's up!");

  }

}, 1000);
```

```
  10                                                    script.js:554
  9                                                     script.js:554
  8                                                     script.js:554
  7                                                     script.js:554
  6                                                     script.js:554
  5                                                     script.js:554
  4                                                     script.js:554
  3                                                     script.js:554
  2                                                     script.js:554
  1                                                     script.js:554
  0                                                     script.js:554
  Time's up!                                            script.js:558
>
```

```
let countdown = 10;

let timer;

function startTimer() {

  // Clear any existing timer

  if (timer) {

    clearInterval(timer);
```

```
  }

  // Start a new countdown timer
  timer = setInterval(() => {
    console.log(countdown); // Print current countdown value
    countdown--; // Decrease countdown
    if (countdown < 0) {
      clearInterval(timer); // Stop the timer
      console.log("Time's up!");
    }
  }, 1000); // Run every second
}

function resetTimer(newTime) {
  countdown = newTime; // Set countdown to new value
  console.log(`Timer reset to ${newTime} seconds.`);
  startTimer(); // Restart the countdown timer
}

// Start the timer
startTimer();

// Reset the timer after 4 seconds
setTimeout(() => {
  resetTimer(5); // Reset to 5 seconds
}, 4000);
```

```
10
9
8
7
Timer reset to 5 seconds.
5
4
3
2
1
0
Time's up!
```