**converting numbers to string**

**code:**

console.log(Number('23'));

console.log(+'23');

**parsing**

console.log(Number.parseInt('30px', 10));

**Number.parseInt:**

* **it gets rid of unnecessary symbols which are not number..but remember that it should start with number**
* **10 is the radix i.e the base number of decimal**
* **in case of hexadecimal we can pass the radix as 16**

**Number.parseFloat:**

* **it gets rid of unnecessary symbols which are not floating point number..but remember that it should start with number**

**Note:** **Number is a global object and it contains different methods dont confuse this with Number primitive or constructor**

**isFinite** is used to check if values is number or string(practically this is used to check if it is a number or not)

**console.log(Number.isFinite(20));**

**the window object is specific to the browser environment and is not part of the ECMAScript specification, which defines the core JavaScript language. The window object is a feature provided by web browsers to expose interfaces and functionalities related to the browser and the document object model (DOM) to JavaScript.**

**In environments outside the browser, such as Node.js, you won't find a window object. Instead, there are other global objects like global in Node.js. The specific global object and its features can vary depending on the JavaScript environment.**

**Math functions**

**In JavaScript, the Math object is not a global object in the same sense as window or document, but it is a built-in object that is readily available in the global scope**

**Similar to the window object, the document object is also part of the browser environment and is not a core feature of the ECMAScript specification. The document object represents the HTML document in a web page and provides an interface for interacting with the DOM (Document Object Model).**

console.log(Math.trunc(23.3)); //retrieves only the integer part

console.log(Math.round(23.3)); //to the nearest integer

console.log((2.7).toFixed(3)); //to fixed returns value in string

console.log(+(2.7).toFixed(3)); //converting string to number

implementing remainder operator

const isEven = *n* => n % 2 === 0;

**we might need numbers bigger than Number.MAX\_SAFE\_INTEGER**

**that’s when bigInt comes into picture**

//BigInt

console.log(3244444444444444444444444444444444444444444444444444444n); //to store big numbers accurately we need to add n to the last number

console.log(BigInt(12321321));

//opertions

console.log(100000000000000000n + 100000000000000n);

console.log(100000000000000n \* 10000000000n);

// console.log(10000000000n \* 100); //cannot mix bigInt and other types

console.log(20n > 15); //this works as js will do type coercion (exception)

console.log(20n == 20);

console.log(20n + ' is really Big');

//Math fns wont work on BigInt

//Divisions

console.log(10n / 3n); //cuts the decimal part and output will be:3n

**Create a date**

**Method 1:** **using constructor**

const now = new Date(); //current date and time

console.log(new Date('Sep 12 2023 17:40:55')); //passing a string as argument

//new Date(year,month,day,hour,mins,seconds)

console.log(new Date(2037, 10, 30,15,23,5));

console.log(new Date(0)); //0 milliseconds after unix time i.e jan 1st 1970

**THE DATES WE HAVE CREATED ARE OBJECTS HENCE THERE WILL BE METHODS:**

const future = new Date(2037, 10, 19, 15, 23);

console.log(future);

console.log(future.getFullYear()); //never use getYear() method always use getFullYear

console.log(future.getMonth()); //zero based month

console.log(future.getDate()); //day of the month

console.log(future.getDay()); //gets the day of the week..0 is sunday

console.log(future.getHours());

console.log(future.getMinutes());

console.log(future.getSeconds());

console.log(future.toISOString());

**Internationalization of dates:**

**`Intl.DateTimeFormat` is a built-in JavaScript object that allows you to format dates and times according to various locales and options. It's a part of the Internationalization API (Intl) introduced in ECMAScript 2015 (ES6) and provides a convenient way to display dates and times in a user-friendly and culturally appropriate manner. `Intl.DateTimeFormat` helps you handle date and time formatting in a way that respects the user's locale and preferences.**

**Here's a detailed explanation of how `Intl.DateTimeFormat` works and its various use cases:**

**### Basic Usage**

**You can create an `Intl.DateTimeFormat` object by passing one or more arguments to its constructor. The most basic usage involves specifying the desired locale:**

**```javascript**

**const dateFormatter = new Intl.DateTimeFormat('en-US');**

**const date = new Date();**

**console.log(dateFormatter.format(date));**

**```**

**In this example, we create a `dateFormatter` object for the 'en-US' (English, United States) locale and then format the current date using this formatter. The output will be something like "9/13/2023" in the US date format.**

**### Locales and Options**

**`Intl.DateTimeFormat` supports various options and configurations for formatting dates and times. Here are some common options:**

**- `locale`: Specifies the locale (language and region) for formatting. It's passed as the first argument to the constructor.**

**- `options`: An object containing formatting options like `year`, `month`, `day`, `hour`, `minute`, `second`, `weekday`, and more. You can use these options to customize the formatting to your needs.**

**```javascript**

**const dateFormatter = new Intl.DateTimeFormat('fr-FR', {**

**year: 'numeric',**

**month: 'long',**

**day: 'numeric',**

**});**

**const date = new Date();**

**console.log(dateFormatter.format(date));**

**```**

**In this example, we create a formatter for the 'fr-FR' (French, France) locale and specify that we want the full month name, numeric day, and full numeric year in the output.**

**### Custom Formatting**

**You can also use `Intl.DateTimeFormat` to customize the date and time format even further by using options like `hour`, `minute`, and `second`. For example:**

**```javascript**

**const dateFormatter = new Intl.DateTimeFormat('de-DE', {**

**year: 'numeric',**

**month: '2-digit',**

**day: '2-digit',**

**hour: '2-digit',**

**minute: '2-digit',**

**});**

**const date = new Date();**

**Certainly! `Intl.NumberFormat` is a built-in JavaScript object that provides a way to format numbers according to the user's locale, which includes settings like the language, country, and preferred number formatting conventions. It is part of the Internationalization API (Intl) introduced in ECMAScript Internationalization API Specification.**

**Here's a detailed explanation of how to use `Intl.NumberFormat` and its various use cases in JavaScript:**

**### Basic Usage:**

**You can create an `Intl.NumberFormat` object by providing one or more arguments, such as the locale and an options object:**

**```javascript**

**const number = 1234567.89;**

**const formatter = new Intl.NumberFormat('en-US'); // Format as per US English locale**

**const formattedNumber = formatter.format(number);**

**console.log(formattedNumber); // Output: "1,234,567.89"**

**```**

**In this example, we created a formatter for the US English locale, which formats the number with a comma as a thousands separator and a period as a decimal separator.**

**### Locale:**

**The first argument to `Intl.NumberFormat` is the locale string. It specifies the language and region for formatting. Examples of locale strings are `'en-US'` (English in the United States), `'fr-FR'` (French in France), `'de-DE'` (German in Germany), and so on.**

**### Options Object:**

**The second argument is an optional options object that allows you to customize the formatting. Some commonly used options include:**

**- `style`: Specifies the number style. Values can be `'decimal'` (default), `'currency'`, or `'percent'`.**

**- `currency`: If the style is `'currency'`, you can specify the currency code (e.g., `'USD'`, `'EUR'`) using this option.**

**- `minimumFractionDigits` and `maximumFractionDigits`: Control the number of decimal places.**

**- `minimumIntegerDigits`: Minimum number of integer digits.**

**- `useGrouping`: A boolean to enable/disable grouping of thousands separators.**

**- `currencyDisplay`: Specifies how the currency should be displayed. Values can be `'symbol'`, `'code'`, or `'name'`.**

**Here's an example using some of these options:**

**```javascript**

**const number = 12345.67;**

**const options = {**

**style: 'currency',**

**currency: 'EUR',**

**minimumFractionDigits: 2,**

**maximumFractionDigits: 2,**

**};**

**const formatter = new Intl.NumberFormat('de-DE', options);**

**const formattedNumber = formatter.format(number);**

**console.log(formattedNumber); // Output: "12.345,67 €"**

**```**

**### Advanced Use Cases:**

**`Intl.NumberFormat` is versatile and can handle more complex formatting requirements. For example, you can use it to format percentages:**

**```javascript**

**const percentage = 0.4567;**

**const options = {**

**style: 'percent',**

**minimumFractionDigits: 2,**

**maximumFractionDigits: 2,**

**};**

**const formatter = new Intl.NumberFormat('en-US', options);**

**const formattedPercentage = formatter.format(percentage);**

**console.log(formattedPercentage); // Output: "45.67%"**

**```**

**You can also use it to format large numbers, such as millions or billions, in a human-readable way:**

**```javascript**

**const bigNumber = 1000000;**

**const options = {**

**notation: 'compact',**

**compactDisplay: 'short',**

**};**

**const formatter = new Intl.NumberFormat('en-US', options);**

**const formattedBigNumber = formatter.format(bigNumber);**

**console.log(formattedBigNumber); // Output: "1M"**

**```**

**### Browser Compatibility:**

**`Intl.NumberFormat` is well-supported in modern browsers. However, it might not be available in older browsers. To ensure compatibility, you can use feature detection or consider using a polyfill library like `intl-messageformat` or `format-number`.**

**In summary, `Intl.NumberFormat` is a powerful tool for formatting numbers in JavaScript, allowing you to tailor the presentation of numbers to the user's locale and specific formatting requirements.**

**Timers:**

**two kinds of timers**

**setTimeout:**

**set timeout time runs just once after a defined time while setInterva timer keeps runnning basicaly forever until we stop it**

**setTimeout can be used to execute some code in the future**

**it is hof which receives a callback fn**

**setTimeout(() => console.log('Here is ur pizza 🍕'), 3000);**

**setTimeout calls the callback fn after 3s i.e 3000ms**

**code execution does not stop at line 553 ,it just registers the callback fn to be called later(i.e after 3s) and it simply continues executing**

**To clear a `setTimeout` timer in JavaScript and prevent its execution, you can use the `clearTimeout` function. Here's how it works:**

**1. Setting a Timeout:**

**When you use the `setTimeout` function, you provide a callback function and a specified delay (in milliseconds) before the callback is executed. The timer is created, and the JavaScript runtime schedules the callback to be executed after the specified delay.**

**For example:**

**```javascript**

**const timeoutId = setTimeout(() => {**

**console.log("This will be executed after a delay");**

**}, 1000);**

**```**

**2. Clearing a Timeout:**

**If you want to cancel the execution of the callback function before it happens, you can use the `clearTimeout` function. It takes the timer ID returned by `setTimeout` as an argument.**

**For example, to cancel the above timeout:**

**```javascript**

**clearTimeout(timeoutId);**

**```**

**After this line is executed, the callback will not be called.**

**3. How `clearTimeout` Works Internally:**

**`setTimeout` and `clearTimeout` work together to manage timers efficiently. Internally, when you call `setTimeout`, the JavaScript runtime schedules the callback function to run after the specified delay and returns a unique timer ID for that specific timeout.**

**When you call `clearTimeout` with the timer ID, it cancels the scheduled callback. It does this by removing the timer from the list of scheduled tasks. Essentially, it tells the runtime not to execute the callback when the time comes.**

**Behind the scenes, modern JavaScript engines like V8 use various mechanisms to manage timers, such as a timer queue, and `clearTimeout` removes the timer from this queue.**

**Here's a complete example:**

**```javascript**

**const timeoutId = setTimeout(() => {**

**console.log("This will be executed after a delay");**

**}, 1000);**

**// If you want to cancel the timeout before it executes:**

**clearTimeout(timeoutId);**

**```**

**This ensures that the callback function won't be executed, and the timer is effectively cleared.**

**To clear a repeating timer set by `setInterval` in JavaScript and prevent it from executing further, you can use the `clearInterval` function. Here's how it works:**

**1. Setting an Interval:**

**When you use the `setInterval` function, you provide a callback function and a specified time interval (in milliseconds). The timer is created, and the callback function is executed repeatedly at the specified interval until it is cleared.**

**For example:**

**```javascript**

**const intervalId = setInterval(() => {**

**console.log("This will be executed repeatedly at a set interval");**

**}, 1000);**

**```**

**2. Clearing an Interval:**

**To stop the execution of the callback function and prevent it from running further, you can use the `clearInterval` function. It takes the interval ID returned by `setInterval` as an argument.**

**For example, to cancel the above interval:**

**```javascript**

**clearInterval(intervalId);**

**```**

**After this line is executed, the callback will no longer be executed at the specified interval.**

**3. How `clearInterval` Works Internally:**

**`setInterval` and `clearInterval` work together to manage intervals efficiently. Internally, when you call `setInterval`, the JavaScript runtime schedules the callback function to run repeatedly at the specified interval and returns a unique interval ID for that specific interval.**

**When you call `clearInterval` with the interval ID, it cancels the scheduled repetitive execution. It does this by removing the interval from the list of scheduled tasks. Essentially, it tells the runtime not to execute the callback at the specified interval anymore.**

**Modern JavaScript engines, like V8, use timer queues and mechanisms for handling intervals, and `clearInterval` removes the interval from this queue.**

**Here's a complete example:**

**```javascript**

**const intervalId = setInterval(() => {**

**console.log("This will be executed repeatedly at a set interval");**

**}, 1000);**

**// If you want to cancel the interval:**

**clearInterval(intervalId);**

**```**

**This ensures that the callback function will no longer be executed at the specified interval, and the interval is effectively cleared.**