**Advance JS Concepts:**

**1. Session Storage**

**Session Storage** allows you to store data for the duration of the page session. The session ends when the browser is closed or the tab is closed. Unlike localStorage, data in sessionStorage is only accessible within the same browser tab and is not shared across tabs.

* **Persistence**: Data is cleared when the session ends (e.g., when the browser or tab is closed).
* **Scope**: Data is restricted to the same tab or window, and cannot be shared between tabs or windows.
* **Storage Limit**: Typically 5MB, depending on the browser.

**Use Case**

Session storage is often used to temporarily store user information or session data that does not need to persist beyond the current session, such as a shopping cart or temporary user settings.

**2. Local Storage**

**Local Storage** is a web storage solution that allows you to store data on the client’s browser persistently, even when the browser is closed or the page is refreshed. Unlike sessionStorage, data in localStorage persists until it is explicitly deleted, making it ideal for storing information that needs to be retained across sessions.

* **Persistence**: Data is stored indefinitely until explicitly deleted.
* **Scope**: Available across browser tabs and windows, as long as they are in the same origin (domain, protocol, and port).
* **Storage Limit**: Typically 5MB per origin.

**Use Case**

Local Storage is typically used for storing non-sensitive user data, such as user preferences, authentication tokens, or settings that should persist across sessions.

**3. Cookies**

**Cookies** are small pieces of data stored by the browser on the client side. They are sent with every HTTP request made to the server and can be used to store user-specific information, such as authentication tokens or user settings.

* **Persistence**: Cookies can be set with expiration dates, after which they are deleted automatically. If no expiration is set, cookies last until the session ends.
* **Scope**: Cookies can be restricted to specific paths or domains. They are sent with every HTTP request to the server.
* **Size Limit**: Each cookie has a maximum size of 4KB.

**Use Case**

Cookies are often used for maintaining sessions on websites, like storing user login information or tracking user behavior for analytics purposes.

**4. Objects**

In JavaScript, **objects** are collections of key-value pairs, where each key is a string and each value can be any data type. Objects allow you to group related data and functionality together, making them essential for structuring data and organizing code.

* **Creation**: Objects are created using either object literals or constructor functions.
* **Accessing**: Object properties can be accessed using either dot notation or bracket notation.

**5. Classes**

**Classes** in JavaScript are syntactic sugar over JavaScript's existing prototype-based inheritance. Classes allow you to create objects that share the same properties and methods and are the blueprint for creating objects with shared behavior.

* **Constructor**: A special method for initializing a new object.
* **Inheritance**: Classes support inheritance, allowing one class (the subclass) to inherit properties and methods from another class (the superclass).
* **Methods**: Functions defined inside a class that represent the behavior of the class.

**6. let, var, const**

In JavaScript, the way you declare variables affects their scope, behavior, and mutability.

* **let**: Introduced in ES6, let allows you to declare variables with block-level scope. The variable can be reassigned, but it is not hoisted in the same way as var.
* **var**: The traditional way to declare variables, var has function-level scope and is hoisted, meaning it can be used before it is declared, though it will be undefined initially.
* **const**: Declares variables that are immutable (cannot be reassigned). const has block-level scope, and once assigned, its value cannot be changed. Note that objects and arrays declared with const can still be mutated (i.e., their properties can be changed).

var is hoisted during the execution of the js code. Their value remains undefined unless something is assigned. let and const are in **temporal** **deadzone** until they are initialized thus they remain inaccessible until their initialization.

**7. map**

The **map()** method creates a new array with the results of calling a provided function on every element in the calling array. It is often used to transform or modify the elements of an array.

* **Input**: A function that processes each element in the array and returns a new value.
* **Output**: A new array with the transformed elements.

**8. filter**

The **filter()** method creates a new array with all elements that pass a test provided by the given function.

* **Input**: A function that returns true or false for each element in the array.
* **Output**: A new array with only the elements that return true.

**9. reduce**

The **reduce()** method applies a function to each element in the array (from left to right) to reduce it to a single value, which can be of any type.

* **Accumulator**: The function takes an accumulator (initially provided or defaulted) and a current value for each iteration.
* **Output**: The final value after all iterations.

**10. forEach**

The **forEach()** method executes a provided function once for each element in the array. Unlike map(), it does not return a new array but instead performs an action on each element.

**11. Hoisting**

**Hoisting** is JavaScript's default behavior of moving declarations (but not initializations) to the top of their scope. This happens for variables declared with var, function declarations, and classes.

* **Variables**: Only the declaration is hoisted, not the initialization. Variables declared with let and const are also hoisted but remain in a "temporal dead zone" until initialized.
* **Functions**: Function declarations are hoisted fully (including the function body), while function expressions are not.

**Use Case**

Understanding hoisting is crucial to avoid unexpected behavior when accessing variables or functions before they are defined.

**12. Arrow Functions**

**Arrow functions** provide a concise syntax for writing functions. They also differ from traditional functions in how they handle the this keyword, as they do not have their own this context but instead inherit it from the surrounding scope.

* **Syntax**: Shorter syntax compared to traditional function expressions.
* **this behavior**: Arrow functions do not bind their own this; instead, they inherit it from the enclosing lexical scope.

**13. Promise**

A **Promise** is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. Promises are used to handle asynchronous operations in JavaScript, allowing for cleaner and more manageable code.

* **States**: A Promise can be in one of three states: fulfilled (operation is completed and is resolved), pending (operation still in process) or rejected (operation got failed).
* **Chaining**: Promises support chaining with .then() and .catch() methods to handle success or failure.

**Use Case**

Promises are useful for handling asynchronous code, such as making API requests or reading files.

**14. Async/Await**

**Async/Await** is syntax introduced in ES8 that makes working with Promises more readable and synchronous-like. async makes a function return a Promise, and await pauses the function execution until the Promise is resolved.

* **Async**: Marks a function as asynchronous.
* **Await**: Pauses the execution of the async function until the Promise is resolved.

**Use Case**

Async/Await simplifies handling multiple asynchronous operations and improves readability when working with Promises.

**15. Import/Export**

**Import/Export** allows JavaScript modules to share code. This modular approach helps in organizing code, especially in larger projects.

**16. = = = vs. = =**

* **= = =**(StrictEquality): Compares both value and type. It does not perform type coercion.
* **= =**(LooseEquality): Compares values but performs type coercion, which can sometimes lead to unexpected results.

**17. !== vs. !=**

* **!==**(Strict Inequality): Compares both value and type, and returns true if the values are different.
* **!=**(Loose Inequality): Compares values with type coercion.

**18. Browser Inspect Window (Developer Tools)**

The **Browser Developer Tools** (DevTools) are built-in tools in modern browsers for inspecting, debugging, and profiling web pages. These tools help developers test and optimize their websites and web applications.

**Tabs Overview:**

1. **Elements**: View and modify the HTML and CSS of the page in real-time.
2. **Console**: View JavaScript logs, errors, and run JavaScript code.
3. **Sources**: Debug JavaScript code, set breakpoints, and view call stacks.
4. **Network**: Monitor network activity, including API requests, assets, and responses.
5. **Performance**: Analyze page load performance, track events, and measure performance bottlenecks.
6. **Memory**: Monitor and optimize memory usage to prevent memory leaks.
7. **Application**: View and manage browser storage, such as cookies, localStorage, and sessionStorage.
8. **Security**: View information about the page’s SSL certificate and security status.
9. **Lighthouse**: Automatically audits the page for performance, accessibility, SEO, and more.

These tabs provide valuable insights into how the page is behaving and help optimize code and performance.