JAVASCRIPT CHEATSHEET

JAVASCRIPT CHEATSHEET

- What’s the difference between using let and var in JavaScript?

The keywords **let** and **var** both declare new variables in JavaScript. The difference between let and var is in the scope of the variables they create:

Variables declared by let are only **available inside the block** where they’re defined.
Variables declared by var are available **throughout the function** in which they’re declared.

```
function varScoping() {
  var x = 1;

  if (true) {
    var x = 2;
    console.log(x); // will print 2
  }

  console.log(x); // will print 2
}

function letScoping() {
  let x = 1;

  if (true) {
    let x = 2;
    console.log(x); // will print 2
  }

  console.log(x); // will print 1
}
```

- What is Hoisting?

In JavaScript, hoisting is a behavior in which variable and function declarations are moved, or "hoisted," to the top of their containing scope (global or local) during the compilation phase, before the code is executed. However, this only applies to the declarations themselves, not the actual assignments or initialization.

Hoisting allows JavaScript to access variables and functions before they appear in the code, but it can sometimes lead to **unexpected behavior,** so understanding it helps in avoiding pitfalls.

Variable Hoisting: Variables declared using var are hoisted to the top of their scope. However, they are only initialized with undefined until their actual assignment is encountered.

```
console.log(x);  // Output: undefined
var x = 5;
console.log(x);  // Output: 5
```

Function Hoisting: Function declarations are also hoisted, and unlike variables, they are hoisted along with their definition. This allows functions to be called before they are declared in the code.

```
greet();  // Output: Hello!

function greet() {
    console.log("Hello!");
}
```

let and **const** Hoisting: **Variables declared with let and const are also hoisted, but they are not initialized**. They remain in a **"temporal dead zone"** (TDZ) until the code execution reaches their declaration. Accessing them before their declaration will result in a **ReferenceError.**

- What is Closures?

Grasping How Inner Functions Have Access to the Outer Enclosing Function's Scope

A closure is when an inner function "remembers" and has access to variables from its outer function scope, even after the **outer function has returned.** Closures are a powerful feature in JavaScript because they enable the creation of **private variables** and persistent state.

```
function outer() {
 let count = 0;
 return function inner() {
   count++;
   console.log(count);
 };
}
const increment = outer();  // outer function is invoked
increment();  // Output: 1
increment();  // Output: 2
```

In this example, **the inner function forms a closure**, capturing the count variable from the outer function. Even after outer has returned, the inner function retains access to count.

A closure in JavaScript is a function that has access to the variables in its parent scope, even after **the parent function has returned.** Closures are created when a function is defined within another function, and the inner function retains access to the outer function's variables.

- What is a Callback Function?

A callback function is a function that is passed as an argument to another function and is executed after the completion of some operations.

This mechanism allows JavaScript to perform tasks like reading files, making HTTP requests, or waiting for user input without blocking the execution of the program. This helps ensure a smooth user experience.

JavaScript runs in a single-threaded environment, meaning it can only execute one command at a time. **Callback functions help manage asynchronous operations,** ensuring that the code continues to run smoothly without waiting for tasks to complete. This approach is crucial for maintaining a responsive and efficient program.

```
function greet(name, callback) {
  console.log(`Hello, ${name}!`);
  callback();
}

function sayGoodbye() {
  console.log("Goodbye!");
}

greet("Alice", sayGoodbye);
```

Simple Explanation :

```
a function that is passed as an argument
              to another function.
```

Used to handle asynchronous operations:
1. Reading a file
2. Network requests
3. Interacting with databases
"Hey, when you're done, call this next".

Example 1 :

```
hello(goodbye);

function hello(callback){
   console.log("Hello!");
   callback();
}
```

```
function goodbye(){
   console.log("Goodbye!");
}
```

Example 2 :

```
function sum (callback , a , b ){
    let result = a + b
    callback(result)
}



function displayResult (result){
    console.log("result",result)
}

sum(displayResult, 5 , 10 )
```

- What is a Callback Hell ?

Callback Hell refers to a situation in JavaScript where **multiple nested callbacks are used to handle asynchronous operations,** leading to code that is **difficult to read, maintain, and debug.** This happens when callbacks are chained or nested within each other, causing the code to grow horizontally, forming a "pyramid" shape, hence the term "callback hell."

```
function fetchData(callback) {
  setTimeout(() => {
    console.log("Fetched data");
    callback();
  }, 1000);
}

function processData(callback) {
  setTimeout(() => {
    console.log("Processed data");
    callback();
  }, 1000);
}

function saveData(callback) {
  setTimeout(() => {
    console.log("Saved data");
    callback();
  }, 1000);
```

```
}

fetchData(() => {
  processData(() => {
    saveData(() => {
      console.log("All done!");
    });
  });
});
```

Problems with Callback Hell:

- **Difficult to Read:** The nested structure becomes hard to read and follow as more callbacks are added.
- **Error Handling:** Error handling becomes tricky because each callback needs to handle errors for the entire chain.
- **Unmanageable Code:** The code becomes more complex and harder to refactor or maintain.
- **Difficult Debugging:** Since the logic is deeply nested, debugging becomes a challenge, especially when an error occurs within one of the callbacks.

Solutions to Avoid Callback Hell:
Using Promises: Promises are an alternative to callbacks that allow chaining of asynchronous operations in a more readable way. Promises flatten the callback structure and improve code readability.

SOLUTION :

```
function fetchData() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        console.log("Fetched data");
        resolve();
      }, 1000);
    });
  }

  function processData() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        console.log("Processed data");
        resolve();
      }, 1000);
    });
  }
```

```
  function saveData() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        console.log("Saved data");
        resolve();
      }, 1000);
    });
  }

  fetchData()
  .then(processData)
  .then(saveData)
  .then(() => console.log("All done!"))
    .catch(error => console.error(error));
```

What is Promise ?

A promise in JavaScript is an object that represents the eventual completion or failure of an asynchronous operation.

It is used for **handling asynchronous operations, such as making API calls or reading files,** in a more organized and readable way.

A Promise is in one of these states:
Pending: The initial state, neither fulfilled nor rejected.
Fulfilled: The operation completed successfully, and the promise has a value.
Rejected: The operation failed, and the promise has a reason (error) for the failure.

```
const myPromise = new Promise((resolve, reject) => {
  let success = true;  // Simulate an asynchronous operation
  if (success) {
    resolve("Operation Successful");
  } else {
    reject("Operation Failed");
  }
});

myPromise
  .then(result => {
    console.log(result);  // This runs if the promise is fulfilled
  })
  .catch(error => {
    console.log(error);  // This runs if the promise is rejected
  });
```

How Promises Work:

then(): Used to specify what to do when the promise is fulfilled. It takes a callback function that is invoked **when the promise is resolved.**

```
promise.then(result => {
  // Handle the successful result
});
```

catch(): Used to specify what to do when the promise is rejected. It takes a callback function that is invoked **when the promise is rejected.**

```
promise.catch(error => {
  // Handle the error
});
```

finally(): (Optional) Used to specify code that runs after the promise is settled (either fulfilled or rejected), **regardless of the outcome.**

```
promise.finally(() => {
  // Code to run after promise is settled
});
```

Example of an Asynchronous Promise:

```
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => {
    const success = true;
    if (success) {
      resolve("Data fetched successfully!");
    } else {
      reject("Failed to fetch data.");
    }
  }, 2000);  // Simulate a 2-second delay (e.g., a network request)
});

fetchData
  .then(result => console.log(result))   // After 2 seconds: "Data fetched successfully!"
  .catch(error => console.error(error));
```

Chaining Promises:
You can chain multiple then() methods to handle a sequence of asynchronous tasks. Each then() returns a new promise, allowing for more sequential operations without deeply nesting callbacks.

Example of chaining promises:

```
fetchData
  .then(result => {
    console.log(result);
    return "Processing data...";  // This is returned to the next then
  })
  .then(processedData => {
    console.log(processedData);
  })
  .catch(error => {
    console.error(error);  // If any error occurs, this will catch it
  });
```

Benefits of Promises:
Avoid callback hell: Promises allow for chaining, making code easier to read and manage.
Error handling: Promises have a built-in way of handling errors using catch().
Composability: Promises can be combined using methods like Promise.all() or Promise.race() for concurrent operations.

Common Promise Methods:
Promise.all(): Takes an array of promises and resolves when all of them resolve, or rejects if any one of them rejects.

```
Promise.all([promise1, promise2])
  .then(results => console.log(results))
  .catch(error => console.error(error));
```

Promise.race(): Resolves or rejects as soon as one of the promises in the array resolves or rejects.

```
Promise.race([promise1, promise2])
  .then(result => console.log(result))
  .catch(error => console.error(error));
```

Conclusion:
A Promise is a way to handle asynchronous operations in JavaScript, providing a cleaner and **more readable alternative to traditional callback functions.** It represents a future value and **helps manage asynchronous code** flows in a more structured manner.

- What is **Map** method:

The map method creates a new array by applying a provided function to each element of the original array. It doesn't modify the original array and instead returns a new array with the transformed values.

```
const newArray = array.map((element, index, array) => {
  // return a new value
});
```

Parameters:
element: The current element being processed in the array.
index: (Optional) The index of the current element.
array: (Optional) The original array.
Returns: A new array with the results of applying the function to each element.

```
const numbers = [1, 2, 3, 4];
const squared = numbers.map(num => num * num);

console.log(squared); // [1, 4, 9, 16]
```

What is **forEach** Method:

The **forEach** method is **used to execute a provided function once for each element in an array.** However, it **doesn't return a new array** — it **simply iterates** over the array and allows you to perform side effects, such as logging or modifying variables outside the array.

```
array.forEach((element, index, array) => {
  // perform side effects
});
```

Parameters:
element: The current element being processed.
index: (Optional) The index of the current element.
array: (Optional) The original array.
Returns: undefined (does not return anything).

```
const numbers = [1, 2, 3, 4];
numbers.forEach(num => console.log(num * num));
// Output: 1 4 9 16
```

Key Differences Between **map** and **forEach**:
Return Value:

- **map returns a new array with the transformed elements.**
- **forEach does not return anything; it returns undefined.**

Usage:

map is typically used when you want to **transform the elements of an array** and **store the result in a new array.**
forEach is used when you want to **perform side effects (like logging or modifying variables outside the loop)**, but you don't need to create a new array.

Immutability:

map does **not modify the original array;** it creates and returns a new array.
forEach does not modify the original array by itself, **but you can manually modify the original array elements** if needed (though this is not recommended).
Chainability:

map returns a new array, so it is chainable with other array methods (e.g., filter, reduce).
forEach returns undefined, so it cannot be chained with other array methods.

```
const numbers = [1, 2, 3, 4];
const doubled = [];
numbers.forEach(num => doubled.push(num * 2));
console.log(doubled); // [2, 4, 6, 8]
```

- What is Debouncing?

Debouncing is a programming **technique used to limit the rate at which a function gets executed.** It ensures that a function is not called too frequently in response to a series of events that happen in quick succession. **The idea is to make sure that the function is only executed once after a certain period of inactivity or delay,** which helps optimize performance in scenarios where a function might otherwise be called too many times in a short period.

How Debouncing Works:
When a series of events are triggered, such as user input, window resizing, or scrolling, debouncing delays the execution of the associated function until after a specified wait time (e.g., 300ms) has passed since the last event. If the event is triggered again before the wait time has passed, the timer is reset, and the function execution is delayed again.

Common Use Cases:

Search input boxes: To avoid firing a search query for each keystroke, debouncing waits for the user to stop typing before making a request.
Window resizing: Instead of executing a resize handler multiple times as the window is being resized, debouncing ensures that it only fires once when resizing is complete.
Button click prevention: To prevent multiple submissions or actions when a button is clicked multiple times in quick succession.

```
const readline = require('readline');

// Debounce function (same as before)
function debounce(func, delay) {
  let timer;
  return function (...args) {
    clearTimeout(timer); // Clears the previous timer if any
    timer = setTimeout(() => {
      func.apply(this, args);  // Executes the function after the delay
```

```
    }, delay);
  };
}

// Function to execute after debounce
function searchQuery(query) {
  console.log(`Searching for: ${query}`);
}

// Debounced version of the search function
const debouncedSearch = debounce(searchQuery, 300);

// Setup for capturing user input in Node.js
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

// Handle user input with debouncing
rl.on('line', (input) => {
  debouncedSearch(input);
});
```

Explanation:
Debounce Function: The debounce function works the same as in the browser, delaying the execution of the searchQuery function until after the user has stopped providing input for 300ms.

User Input Simulation:

The readline module in Node.js is used to capture input from the command line.
Every time the user presses "Enter" after typing, the debouncedSearch function is triggered.
Debouncing in Action:

If you keep entering new input within 300ms of the previous input, the search will not be executed until you stop typing for at least 300ms.

- What is Throttling ?

Throttling is a programming technique used to control the rate at which a function gets executed over time. Unlike debouncing, which postpones execution until after a certain delay has passed since the last event, **throttling ensures that a function is called at most once in a specified interval.** This can help improve performance and responsiveness in scenarios where functions are triggered frequently, such as scrolling, resizing, or keypress events.

How Throttling Works

When a throttled function is called, it will execute immediately, but it will ignore subsequent calls until the specified time interval has elapsed. After that interval, the function can be invoked again.

Common Use Cases

Window resizing: To limit how often a resize handler runs when the window is being resized.
Scroll events: To reduce the number of times a function runs while scrolling (e.g., lazy loading images or infinite scrolling).
Button clicks: To prevent users from submitting a form multiple times quickly.

```
const readline = require('readline');
function throttle(func, limit) {
  let lastFunc;
  let lastRan;

  return function (...args) {
    const context = this;

    if (!lastRan) {
      func.apply(context, args); // Execute immediately
      lastRan = Date.now(); // Update the last ran time
    } else {
      clearTimeout(lastFunc); // Clear the timeout if it's already set
      lastFunc = setTimeout(() => {
        if ((Date.now() - lastRan) >= limit) {
          func.apply(context, args); // Execute the function after the limit
          lastRan = Date.now(); // Update the last ran time
        }
      }, limit - (Date.now() - lastRan));
    }
  };
}

// Function to execute after throttling
function logInput(input) {
  console.log(`Input logged: ${input}`);
}

// Create a throttled version of the logInput function
const throttledLogInput = throttle(logInput, 1000); // Throttle to 1 second

// Setup for capturing user input

const rl = readline.createInterface({
```

```
  input: process.stdin
output: process.stdout
})



// Handle user input with throttling
console.log('Type something (you will see it logged once every second):');
rl.on('line', (input) => {
  throttledLogInput(input);
});
```

Explanation:
Throttle Function:

Similar to the previous explanation, this function accepts another function (func) and a time limit (limit). It maintains the last time the function was executed (lastRan) and allows for throttled execution of the function.
Logging Function:

The logInput function logs the input received from the console.
Using readline:

The readline module is used to read user input from the terminal.
The line event is triggered every time the user presses "Enter".
Throttled Input Handling:

The throttledLogInput function ensures that logInput is called at most once every second, regardless of how many times the user inputs text.

`setInterval` and throttling are both techniques used to manage how often a piece of code runs, but they serve different purposes and are applied in different contexts. Here's a detailed comparison of the two:

setInterval

- **Purpose**:
  - `setInterval` is a built-in JavaScript function that repeatedly executes a specified function at a defined interval (in milliseconds).
- **Usage**:
  - It is commonly used for tasks that need to happen at regular intervals, such as updating a UI element, polling a server, or creating animations.
- **Behavior**:

- The function specified in `setInterval` will be called repeatedly with the specified time delay, regardless of how long the execution of the function takes. This means that if the function takes longer than the interval to execute, it may start stacking calls, leading to multiple executions overlapping.

- **Example**:

```
setInterval(() => {
    console.log('This message is logged every 1 second.');
}, 1000);
```

- **Stopping the Interval**:
  - You can stop the interval by calling `clearInterval()` and passing the interval ID returned by `setInterval`.

```
const intervalId = setInterval(() => {
    console.log('This will keep running.');
}, 1000);

// To stop the interval after 5 seconds
setTimeout(() => {
    clearInterval(intervalId);
    console.log('Interval stopped.');
}, 5000);
```

Throttling

- **Purpose**:
  - Throttling is a technique that limits how often a function can be executed over time, ensuring that it is called at most once in a specified period. It is often used to improve performance in scenarios where functions are triggered frequently, such as scrolling or resizing events.

- **Usage**:
  - Commonly used in event handling where a function is triggered multiple times in rapid succession (like scrolling), allowing it to execute only at set intervals to reduce the number of calls.

- **Behavior**:
  - The throttled function will execute immediately the first time it is called and will ignore further calls until the specified time interval has passed. If more calls occur within that interval, they will be ignored until the next allowed execution.

- **Example**:

```
function throttle(func, limit) {
    let lastFunc;
    let lastRan;
```

```
    return function (...args) {
        const context = this;

        const now = Date.now();
        if (!lastRan || (now - lastRan) >= limit) {
            func.apply(context, args);
            lastRan = now;
        } else {
            clearTimeout(lastFunc);
            lastFunc = setTimeout(() => {
                if ((Date.now() - lastRan) >= limit) {
                    func.apply(context, args);
                    lastRan = Date.now();
                }
            }, limit - (now - lastRan));
        }
    };
}

// Throttling example usage
window.addEventListener('scroll', throttle(() => {
    console.log('Scroll event fired!');
}, 1000)); // Throttle to 1 second
```

Key Differences

Feature	`setInterval`	Throttling
Execution Frequency	Executes at regular intervals regardless of execution time.	Executes at most once in a specified period, allowing the first call to happen immediately.
Use Cases	Regular tasks, polling, animations, timed updates.	Handling high-frequency events (e.g., scroll, resize) efficiently.
Handling Overlap	Can lead to overlapping executions if the function takes longer than the interval.	Prevents overlapping executions, ensuring that calls are spaced out.
Stopping Execution	Uses `clearInterval()` to stop.	No built-in method; typically controlled by how you set the throttled function.

When to Use Which?

- **Use `setInterval`** when you want to perform a task at a consistent interval, regardless of other factors (like user interactions).

- **Use throttling** when dealing with events that can fire frequently and you want to ensure a function is executed at controlled intervals, improving performance and avoiding overwhelming the system.

What is Pure ?

A **pure function** in JavaScript (and functional programming in general) is a function that adheres to two main principles:

1. **Deterministic**: Given the same input, a pure function will always return the same output. This means the function's result is solely determined by its input values and does not depend on any external state or variables.
2. **No Side Effects**: A pure function does not cause any observable side effects outside of the function itself. This means it does not modify any external state, such as global variables, or perform actions like modifying data, interacting with I/O (e.g., network requests, file systems), or altering input arguments.

Characteristics of Pure Functions

- **Immutability**: Pure functions typically work with immutable data, meaning they do not alter the original data but instead return new data structures.
- **Referential Transparency**: This means that a function call can be replaced with its return value without changing the program's behavior. This property allows for easier reasoning about code and can lead to more effective optimizations by the compiler or runtime.

Example of a Pure Function

Here’s a simple example of a pure function:

```
function add(a, b) {
  return a + b; // Always returns the same output for the same inputs
}

// Usage
const result1 = add(2, 3); // Returns 5
const result2 = add(2, 3); // Also returns 5
console.log(result1 === result2); // true
```

Example of an Impure Function

To illustrate the difference, here’s an example of an impure function:

```
let count = 0;

function increment() {
  count += 1; // Modifies the external variable 'count', causing a side effect
  return count;
```

```
}

// Usage
console.log(increment()); // Returns 1
console.log(increment()); // Returns 2, but depends on the external state
```

Advantages of Pure Functions

1. **Easier Testing**: Pure functions are easier to test because you can predict their output given specific inputs without worrying about hidden states.
2. **Higher Order Functions**: They can be passed as arguments or returned from other functions, facilitating functional programming techniques like map, filter, and reduce.
3. **Concurrency**: Since they do not have side effects, pure functions can be executed in parallel without concerns about race conditions.
4. **Caching and Memoization**: Because they always return the same output for the same inputs, pure functions are ideal candidates for caching, improving performance.

What is High Order Functions :

Higher-order functions are a fundamental concept in functional programming, including JavaScript. A higher-order function is a function that meets one or both of the following criteria:

1. **Takes one or more functions as arguments**: It can accept other functions as parameters.
2. **Returns a function as its result**: It can produce a new function as its output.

Characteristics of Higher-Order Functions

- They enable **abstraction** in programming by allowing you to treat functions as first-class citizens, meaning functions can be passed around just like any other data type (like strings or numbers).
- They are useful for creating **composable** code, where you can build complex operations from simpler ones.

Common Examples of Higher-Order Functions in JavaScript

1. **Array Methods**: Many built-in array methods in JavaScript are higher-order functions. For instance, `map`, `filter`, and `reduce` accept a function as an argument.
    - **`map`**: Transforms each element of an array using a provided function.

      ```
      const numbers = [1, 2, 3, 4];
      const doubled = numbers.map(num => num * 2);
      console.log(doubled); // [2, 4, 6, 8]
      ```

    - **`filter`**: Filters elements of an array based on a provided condition.

      ```
      const numbers = [1, 2, 3, 4, 5];
      const evens = numbers.filter(num => num % 2 === 0);
      ```

```
console.log(evens); // [2, 4]
```

- `reduce`: Reduces the array to a single value using a provided function.

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // 10
```

2. **Function Returning a Function**: You can create functions that return other functions.

```
function makeMultiplier(multiplier) {
  return function (x) {
    return x * multiplier;
  };
}

const double = makeMultiplier(2);
console.log(double(5)); // 10
const triple = makeMultiplier(3);
console.log(triple(5)); // 15
```

3. **Using Callbacks**: Higher-order functions often use callbacks to perform operations.

```
function executeFunction(func) {
  func(); // Calls the function passed as an argument
}

executeFunction(() => {
  console.log("This is a callback function!");
});
```

Benefits of Higher-Order Functions

1. **Code Reusability**: They promote reusable code by allowing you to create generic functions that can work with different logic by passing different functions as arguments.
2. **Improved Abstraction**: Higher-order functions enable a higher level of abstraction, allowing developers to focus on what to do with data rather than how to do it.
3. **Functional Composition**: You can compose functions to create more complex functionality in a clear and concise manner.

Summary

Higher-order functions are a powerful feature of JavaScript that enhance its flexibility and expressiveness. They enable developers to write cleaner, more modular code and leverage the full power of functional programming principles. Understanding and using higher-order functions can lead to more efficient and maintainable code.

What is Memoization ?

Memoization is an optimization technique used primarily in programming to improve the performance of functions by caching their results. When a function is called with the same arguments multiple times, instead of recalculating the result, a memoized function returns the cached result for those arguments. This can significantly reduce the time complexity of certain algorithms, especially those that involve recursive calculations or repeated computations.

How Memoization Works

1. **Store Results**: The function stores the results of expensive function calls in a data structure (commonly an object or a Map).
2. **Check Cache**: When the function is called, it first checks if the result for the given input exists in the cache.
3. **Return Cached Result**: If the result is found, it is returned directly from the cache. If not, the function computes the result, stores it in the cache, and then returns the result.

Benefits of Memoization

- **Performance Improvement**: By avoiding repeated calculations, memoization can greatly reduce execution time, especially for functions with exponential time complexity.
- **Efficient Resource Usage**: It reduces CPU time and can also lead to lower energy consumption.

Example of Memoization in JavaScript

Here's a simple example of how you can implement memoization for a function that calculates Fibonacci numbers, which is a classic use case due to its exponential time complexity when implemented recursively.

```
function memoize(fn) {
  const cache = {}; // Object to store cached results

  return function (...args) {
    const key = JSON.stringify(args); // Create a unique key for the cache based on arguments
    if (cache[key]) {
      // Check if the result is already in the cache
      return cache[key]; // Return the cached result
    }
    const result = fn.apply(this, args); // Call the original function
    cache[key] = result; // Store the result in the cache
    return result; // Return the computed result
  };
}
```

```
// Example of a memoized Fibonacci function
const fibonacci = memoize((n) => {
  if (n <= 1) return n; // Base case
  return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
});

// Usage
console.log(fibonacci(40)); // Computes and caches results
console.log(fibonacci(40)); // Returns cached result
```

Explanation of the Example

- **Memoization Function**: The `memoize` function takes another function (`fn`) as an argument and returns a new function that has memoization capabilities.
- **Cache Storage**: An object named `cache` is created to store results. The key is generated by converting the arguments to a JSON string to ensure it's unique for different inputs.
- **Checking the Cache**: Before executing the original function, the memoized function checks if the result is already cached.
- **Recursive Fibonacci**: The Fibonacci function is recursively defined, and memoization prevents recalculating the Fibonacci numbers for inputs that have already been computed.

Advanced Memoization

1. **Handling Multiple Arguments**: The example above uses a JSON string to create a unique key, which works well for simple data types but may not be efficient for large objects or arrays. A more sophisticated key generation mechanism might be required for complex inputs.
2. **Memory Management**: Over time, the cache can grow large, leading to increased memory usage. You can implement cache eviction strategies to limit the size of the cache (e.g., using Least Recently Used (LRU) caching).
3. **Custom Cache Logic**: You might want to implement more advanced caching strategies, such as time-limited caching, where results expire after a certain period.

The **Event Loop** and **Concurrency Model** are foundational concepts in JavaScript, crucial for understanding how JavaScript handles asynchronous operations and executes code. Let's break these concepts down:

What is JavaScript Concurrency Model

1. **Single-Threaded**: JavaScript is a single-threaded language, which means it can execute only one block of code at a time. This design allows for simplicity in the execution of code but can lead to issues if long-running tasks block the main thread.
2. **Non-Blocking I/O**: JavaScript's design allows for non-blocking input/output operations. When an I/O operation (like reading a file or making a network request) is initiated, JavaScript doesn't wait for

the operation to complete. Instead, it continues executing other code. Once the operation is complete, a callback or promise resolves, notifying JavaScript to handle the result.

3. **Concurrency Model**: JavaScript achieves concurrency through its event-driven model. While the code execution is single-threaded, it can handle multiple tasks by using callbacks, promises, and asynchronous functions. This model allows for handling tasks concurrently, making JavaScript highly efficient for I/O-bound operations.

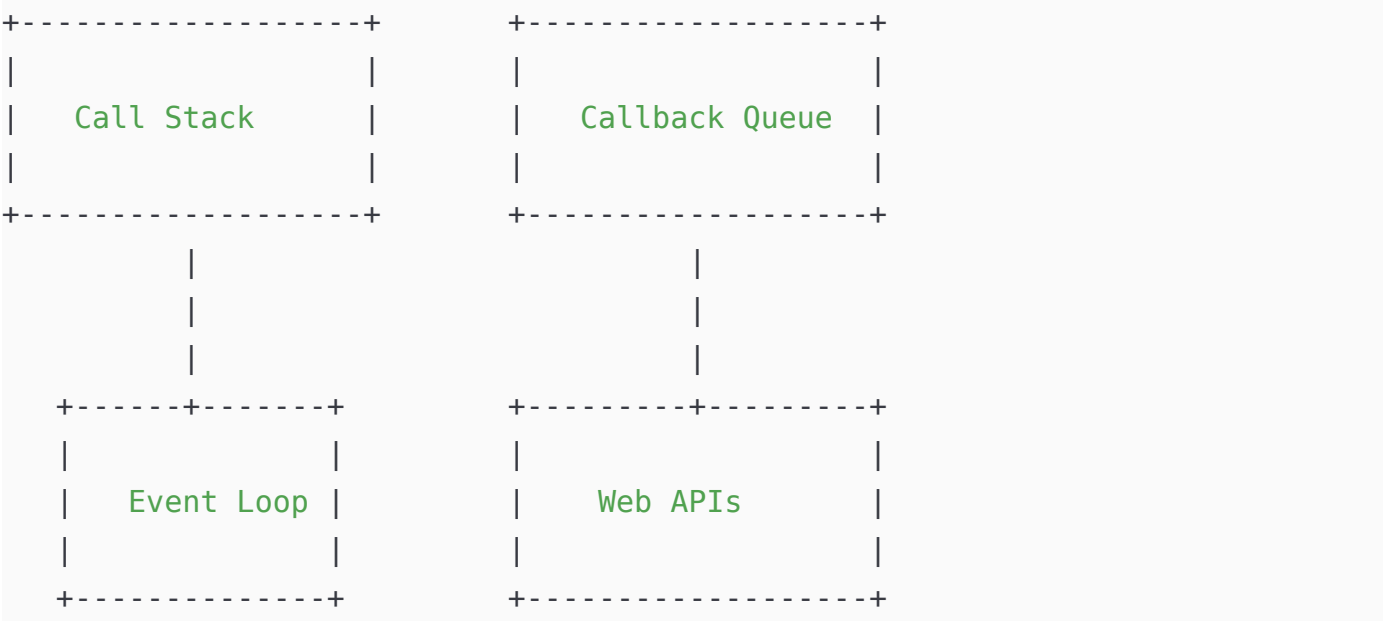
The Event Loop

The **Event Loop** is the mechanism that allows JavaScript to perform non-blocking operations despite being single-threaded. Here's how it works:

1. **Call Stack**: The call stack is where JavaScript keeps track of function execution. When a function is called, it's added to the top of the stack, and when it returns, it's removed.
2. **Web APIs**: For tasks like timers (`setTimeout`, `setInterval`), HTTP requests (using `fetch` or `XMLHttpRequest`), and DOM events, JavaScript relies on the browser's Web APIs. These tasks run outside the main thread and are asynchronous.
3. **Callback Queue**: When a Web API operation is complete (e.g., an HTTP request has finished), it pushes its callback function to the **callback queue**.
4. **Event Loop**: The event loop continuously checks the call stack and the callback queue:
    - If the call stack is empty, it takes the first callback from the queue and pushes it onto the stack for execution.
    - This process continues, allowing JavaScript to process asynchronous tasks without blocking the execution of code.

Visual Representation of the Event Loop

Here's a simplified diagram illustrating the event loop:



Example of the Event Loop in Action

Here's a simple example to illustrate how the event loop works:

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout 1");
}, 0);

setTimeout(() => {
  console.log("Timeout 2");
}, 100);

Promise.resolve().then(() => {
  console.log("Promise 1");
});

console.log("End");
```

Output Explanation:

1. **"Start"** is logged first.
2. The two `setTimeout` calls are scheduled but don't block the execution. They are registered with the Web APIs.
3. The `Promise` is resolved, and its `.then()` callback is scheduled to be executed after the current stack is empty.
4. **"End"** is logged next.
5. After the current execution context is complete, the event loop checks the callback queue:
   - The `Promise` callback runs first (because promises have higher priority over timers).
   - **"Promise 1"** is logged.
6. Next, the first `setTimeout` callback runs, logging **"Timeout 1"**.
7. Finally, **"Timeout 2"** is logged after 100ms.

Summary

- **Concurrency Model**: JavaScript's concurrency model allows it to handle multiple tasks simultaneously while being single-threaded. It uses non-blocking I/O operations and asynchronous programming constructs to achieve this.
- **Event Loop**: The event loop is the core mechanism that enables JavaScript to handle asynchronous tasks efficiently. It allows JavaScript to execute callback functions from the callback queue when the call stack is empty.

Understanding the event loop and the concurrency model is essential for writing efficient asynchronous code in JavaScript, especially when dealing with I/O-bound operations or complex applications.

Currying is a functional programming technique where a function is transformed into a series of functions, each taking a single argument. Instead of taking all arguments at once, a curried function takes the first argument and returns a new function that takes the next argument, and so on, until all arguments have been provided.

Benefits of Currying

1. **Partial Application**: Currying allows you to create new functions by fixing some arguments while leaving others to be specified later. This is particularly useful for creating specialized functions based on more general ones.
2. **Function Reusability**: You can create reusable and composable functions. It makes the code more modular and easier to maintain.
3. **Improved Readability**: Currying can enhance the readability of the code by making the flow of data through functions clearer.

Currying Example in JavaScript

Here's how you can implement currying in JavaScript:

Manual Currying

```
function curry(fn) {
  return function curried(...args) {
    // If the number of arguments is less than required, return a new
function
    if (args.length < fn.length) {
      return function(...args2) {
        return curried(...args.concat(args2)); // Concatenate arguments
      };
    }
    // If the required number of arguments is met, invoke the original
function
    return fn(...args);
  };
}

// Example function that takes three arguments
function add(a, b, c) {
  return a + b + c;
}

// Curried version of the add function
const curriedAdd = curry(add);

console.log(curriedAdd(1)(2)(3)); // 6
```

```
console.log(curriedAdd(1, 2)(3)); // 6
console.log(curriedAdd(1, 2, 3)); // 6
```

Explanation of the Example

1. **Curry Function**: The `curry` function takes a function `fn` as an argument and returns a new function called `curried`.
2. **Argument Checking**: Inside `curried`, it checks whether the number of arguments passed (`args`) is less than the expected number of arguments for `fn` (`fn.length`). If it is, a new function is returned, which will accept additional arguments.
3. **Concatenation of Arguments**: The new function calls `curried` again, concatenating the previously received arguments with the new ones.
4. **Final Call**: When the required number of arguments has been provided, it calls the original function `fn` with all the accumulated arguments.

Using Currying with Practical Examples

Currying is useful in various scenarios, such as creating specialized functions or event handlers.

Example: Creating a Multiplication Function

```
function multiply(a) {
  return function(b) {
    return a * b;
  };
}

const double = multiply(2); // Create a function that doubles the input
console.log(double(5)); // 10
```

Example: Combining with Higher-Order Functions

Currying is often used with higher-order functions, like when creating reusable utility functions.

```
const filterBy = (criteria) => (array) => {
  return array.filter(item => item === criteria);
};

const filterByApple = filterBy('apple'); // Create a function to filter for
'apple'
const fruits = ['apple', 'banana', 'orange', 'apple', 'grape'];

console.log(filterByApple(fruits)); // ['apple', 'apple']
```

Summary

Currying is a powerful functional programming technique that enhances the flexibility and reusability of functions in JavaScript. By transforming a multi-argument function into a series of single-argument functions, currying facilitates partial application, clearer code structure, and improved maintainability. It is widely used in functional programming and modern JavaScript development to create cleaner and more modular code.

WHAT WAS CHANGE AFTER ECMAScript 5(ES5) to ECMAScript 6 (ES6) ?

The transition from ECMAScript 5 (ES5) to ECMAScript 6 (ES6), also known as ECMAScript 2015, introduced several significant changes and enhancements to JavaScript. Here's an overview of the key features and changes that came with ES6:

1. Block-Scoped Variables

- **`let` and `const`**: ES6 introduced two new ways to declare variables: `let` for block-scoped variables and `const` for constants.

```
let x = 10;
if (true) {
    let x = 20; // Different x
    console.log(x); // 20
}
console.log(x); // 10

const y = 30;
// y = 40; // Error: Assignment to constant variable.
```

2. Arrow Functions

- Arrow functions provide a more concise syntax for writing function expressions and lexically bind the `this` value.

```
const add = (a, b) => a + b;
console.log(add(2, 3)); // 5

const obj = {
    value: 42,
    getValue: function() {
        return () => this.value; // Lexical `this`
    }
};
const getValue = obj.getValue();
console.log(getValue()); // 42
```

3. Template Literals

- Template literals allow for multi-line strings and string interpolation using backticks (`` ` ``).

```
const name = 'World';
const greeting = `Hello, ${name}!`;
console.log(greeting); // Hello, World!
```

4. Default Parameters

- Functions can have default parameter values, making it easier to work with optional parameters.

```
function multiply(a, b = 1) {
    return a * b;
}
console.log(multiply(5)); // 5
```

5. Destructuring Assignment

- Destructuring allows unpacking values from arrays or properties from objects into distinct variables.

```
const arr = [1, 2, 3];
const [a, b] = arr; // Destructuring array
console.log(a, b); // 1 2

const obj = { x: 1, y: 2 };
const { x, y } = obj; // Destructuring object
console.log(x, y); // 1 2
```

6. Rest and Spread Operators

- The rest operator (`...`) allows collecting multiple elements into an array, while the spread operator (`...`) expands an array into individual elements.

```
function sum(...numbers) {
    return numbers.reduce((acc, num) => acc + num, 0);
}
console.log(sum(1, 2, 3)); // 6

const arr1 = [1, 2, 3];
const arr2 = [4, 5, ...arr1]; // Spread operator
console.log(arr2); // [4, 5, 1, 2, 3]
```

7. Modules

- ES6 introduced a module system that allows exporting and importing modules, enabling better organization of code.

```
// module.js
export const PI = 3.14;
export function add(a, b) {
    return a + b;
}
```

```
// main.js
import { PI, add } from './module.js';
console.log(PI); // 3.14
console.log(add(2, 3)); // 5
```

8. Classes

- ES6 introduced a class syntax for creating objects and handling inheritance, which is syntactic sugar over the existing prototype-based inheritance.

```
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a noise.`);
    }
}

class Dog extends Animal {
    speak() {
        console.log(`${this.name} barks.`);
    }
}

const dog = new Dog('Rex');
dog.speak(); // Rex barks.
```

9. Promises

- Promises are a new primitive type for handling asynchronous operations, providing a cleaner and more powerful alternative to callbacks.

```
const fetchData = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('Data received');
        }, 1000);
    });
};

fetchData().then(data => console.log(data)); // Data received
```

10. Iterators and Generators

- ES6 introduced iterators and generators, allowing for more complex control flows and iterable objects.

```
function* generator() {
    yield 1;
    yield 2;
    yield 3;
}

const gen = generator();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
```

11. Maps and Sets

- ES6 introduced new data structures: `Map` for key-value pairs and `Set` for unique values.

```
const map = new Map();
map.set('key', 'value');
console.log(map.get('key')); // value

const set = new Set();
set.add(1);
set.add(2);
set.add(1); // Duplicate will not be added
console.log(set.size); // 2
```

Summary

The upgrades from ES5 to ES6 significantly enhanced the capabilities of JavaScript, making it easier to write clean, maintainable, and modular code. With features like arrow functions, classes, modules, promises, and destructuring, ES6 has become a foundational part of modern JavaScript development, allowing developers to create more complex applications with greater ease and efficiency.

Strict mode in JavaScript is a feature introduced in ECMAScript 5 (ES5) that enables a stricter variant of JavaScript, which helps developers write more secure and error-free code. It can be applied globally or locally (inside a function), and it changes the way JavaScript code behaves by catching common coding mistakes and preventing potentially unsafe actions.

Enabling Strict Mode

Strict mode is enabled by adding the string `"use strict";` at the beginning of a script or a function.

- **Global strict mode**: Applies to the entire script.

```
"use strict";
// This entire script runs in strict mode
```

- **Local strict mode**: Applies only to the function in which it's declared.

```
function myFunction() {
    "use strict";
    // Only this function runs in strict mode
}
```

Key Differences in Strict Mode

Here are the major changes and restrictions when strict mode is enabled:

1. **Eliminates Silent Errors**: JavaScript typically allows certain errors to fail silently. In strict mode, these errors throw exceptions, making them easier to detect.
    - Example: Assigning to an undeclared variable

    ```
    "use strict";
    x = 10; // Error: x is not defined
    ```

2. **Prevents Accidental Globals**: In non-strict mode, assigning a value to an undeclared variable automatically creates a global variable. Strict mode prevents this.
    - Non-strict mode:

    ```
    y = 5; // No error, creates a global variable y
    ```

    - Strict mode:

    ```
    "use strict";
    y = 5; // Error: y is not defined
    ```

3. **Disallows Duplicates in Object Literals**: Strict mode doesn't allow duplicate property names in object literals.

```
"use strict";
const obj = { a: 1, a: 2 }; // Error: Duplicate property name 'a'
```

4. **Disallows Duplicates in Function Parameters**: Duplicate function parameters are not allowed in strict mode.

```
"use strict";
function myFunc(a, a) { // Error: Duplicate parameter name not allowed in strict mode
    return a;
}
```

5. **Prevents `this` from Defaulting to Global Object**: In non-strict mode, when calling a function without specifying an object, `this` defaults to the global object (`window` in browsers). In strict mode, `this` remains `undefined` in such cases.
    - Non-strict mode:

```
function myFunc() {
    console.log(this); // 'this' refers to the global object (window
in browser)
}
myFunc();
```

- Strict mode:

```
"use strict";
function myFunc() {
    console.log(this); // 'this' is undefined
}
myFunc(); // undefined
```

6. **Prohibits Deleting Variables or Functions**: In strict mode, you cannot delete variables, functions, or arguments.

```
"use strict";
let x = 10;
delete x; // Error: Cannot delete variable 'x'
```

7. **Restrictions on `eval`**: In strict mode, `eval` operates in its own scope, meaning it cannot create variables or functions in the enclosing scope. It also prevents access to the `caller` and `arguments` properties within functions created by `eval`.

```
"use strict";
eval("var x = 5;");
console.log(x); // Error: x is not defined
```

8. **Prohibits `with` Statement**: The `with` statement is disallowed in strict mode due to its potential to create confusion and reduce code predictability.

```
"use strict";
with (Math) { // Error: 'with' statement not allowed in strict mode
    console.log(PI);
}
```

9. **Throws Errors for `octal literals`**: Strict mode disallows octal syntax (`075`), which can be ambiguous, to avoid confusion with decimal numbers.

```
"use strict";
let num = 075; // Error: Octal literals are not allowed in strict mode
```

10. **Enhanced Security**: Strict mode also enables better security by preventing certain actions that are unsafe or might lead to security vulnerabilities, such as modifying read-only properties.

Why Use Strict Mode?

- **Error Detection**: It catches common coding errors that might otherwise go unnoticed in non-strict mode.
- **Improved Performance**: Strict mode often enables better optimizations in JavaScript engines, resulting in faster code execution.
- **Future-Proofing**: Some future versions of JavaScript are designed with strict mode in mind, so it helps in writing code that is compatible with future standards.
- **Security**: It avoids potentially dangerous actions and helps prevent bugs that might lead to security vulnerabilities.

Example of Strict Mode in Action

```
"use strict"; // Enabling strict mode globally

function myFunction() {
    z = 3; // Error: z is not defined
}

myFunction();
```

In the above example, without strict mode, `z` would have been automatically declared as a global variable. However, in strict mode, this throws an error because `z` is not explicitly declared with `let`, `const`, or `var`.

Conclusion

Strict mode in JavaScript is a powerful tool that helps developers write cleaner, more secure, and more predictable code. It enforces better coding practices by disallowing some of the more error-prone aspects of JavaScript, helping developers avoid common pitfalls and bugs.

What is the difference between localStorage and SessionStorage ?

In JavaScript, both **`localStorage`** and **`sessionStorage`** are part of the Web Storage API, which allows you to store key-value pairs in a web browser. They are similar in usage, but they differ in terms of their scope and lifespan.

Key Differences between **`localStorage`** and **`sessionStorage`**

Feature	**`localStorage`**	**`sessionStorage`**
Lifespan	Persists until explicitly deleted.	Persists only for the duration of the page session (until the tab is closed).
Scope	Shared across all tabs and windows of the same origin (same domain, protocol, and port).	Unique to the current tab or window. Not shared across tabs.
Storage Capacity	Typically 5–10 MB (depends on browser).	Typically 5–10 MB (depends on browser).

Feature	`localStorage`	`sessionStorage`
Accessibility	Accessible by JavaScript on the same origin (domain, protocol, and port).	Accessible by JavaScript on the same origin (domain, protocol, and port).
Use Case	Used for long-term storage (e.g., user settings, authentication tokens).	Used for session-specific data (e.g., temporary form data, per-tab user preferences).

1. `localStorage`

- Data stored in `localStorage` persists even after the browser is closed or the page is refreshed. It remains until it is explicitly deleted by the user or through code.
- Commonly used for saving data that should persist across browsing sessions, like user settings or login tokens.
- Accessible in all windows, tabs, and frames on the same domain.

Example usage:

```
// Storing data
localStorage.setItem('username', 'JohnDoe');

// Retrieving data
const username = localStorage.getItem('username');
console.log(username); // Output: JohnDoe

// Removing data
localStorage.removeItem('username');

// Clearing all data from localStorage
localStorage.clear();
```

2. `sessionStorage`

- Data stored in `sessionStorage` is cleared when the page session ends, which happens when the tab or window is closed. Even if the page is refreshed, the data will persist, but it won't be accessible across tabs or windows.
- Typically used for session-specific data, such as storing temporary form input data or user navigation history within the current session.

Example usage:

```
// Storing data
sessionStorage.setItem('theme', 'dark-mode');

// Retrieving data
const theme = sessionStorage.getItem('theme');
```

```
console.log(theme); // Output: dark-mode

// Removing data
sessionStorage.removeItem('theme');

// Clearing all data from sessionStorage
sessionStorage.clear();
```

Detailed Comparison

1. **Lifespan**:
   - `localStorage`: Data persists indefinitely until manually cleared by the user or via code. Even if the browser is closed and reopened, the data remains.
   - `sessionStorage`: Data is cleared automatically when the session ends (i.e., when the tab or window is closed). Refreshing the page does not clear it, but opening a new tab or window won't share the session data.
2. **Scope**:
   - `localStorage`: Data is shared across all windows and tabs of the same origin. This means if you store data in one tab, you can access it in another tab that is on the same domain.
   - `sessionStorage`: Each tab or window has its own unique `sessionStorage` object. Data is not shared across tabs, even if they are on the same domain.
3. **Use Cases**:
   - `localStorage`: Ideal for data that you want to persist across sessions, such as user preferences, themes, or authentication tokens.
   - `sessionStorage`: Useful for temporary data that should be cleared once the session ends, such as form data during a multi-step form process or temporary selections that should not persist across sessions.
4. **Storage Capacity**:
   Both `localStorage` and `sessionStorage` typically have the same storage limits, usually between **5MB to 10MB** depending on the browser. This is generally sufficient for most web applications, but they are not suitable for storing large amounts of data.

When to Use Which?

- **Use** `localStorage` when you need to store data that should persist between different sessions. For example, if a user sets their preferred language on a website, `localStorage` would be a good place to store that choice, as it will remain intact even if the user closes the browser and returns later.
- **Use** `sessionStorage` when you need to store temporary data specific to a particular session. For example, if a user is filling out a multi-step form and you want to save their progress temporarily as they navigate between steps, `sessionStorage` is a good option since the data will disappear after they close the tab.

Summary

- `localStorage`: Best for persistent, cross-session data storage.
- `sessionStorage`: Best for temporary, session-specific data storage.

Both are easy-to-use options for client-side storage in JavaScript, but you should choose between them based on the longevity and scope of the data you need to store.

What is the difference between localStorage and sessionStorage?

Web Storage (which includes `localStorage` and `sessionStorage`) and **Cookies** are both mechanisms for storing data on the client-side in web browsers, but they differ in terms of capabilities, purpose, and how they are handled.

Key Differences: Web Storage vs Cookies

Feature	Web Storage (`localStorage` and `sessionStorage`)	Cookies
Storage Capacity	Typically 5–10 MB (depends on browser)	Typically 4 KB per cookie (limited by browser)
Data Expiration	`localStorage`: Persists until manually deleted. `sessionStorage`: Cleared when the session ends (tab/window is closed).	Can set expiration dates. Otherwise, expires when the session ends (if not set).
Scope	Web storage is per origin (domain, protocol, port). `localStorage` data is shared across tabs. `sessionStorage` is unique to the tab or window.	Cookies are sent with every HTTP request to the server. They are shared across tabs if the same domain is accessed.
Data Sent with Requests	**Not automatically sent with HTTP requests** (stored on the client-side only).	**Automatically sent** with every HTTP request to the server. This can cause overhead, especially with large cookies.
Security	Stored on the client and not sent to the server unless explicitly done via JavaScript or API requests.	Sent with every HTTP request, which could expose data to unauthorized parties if not properly secured (e.g., `HttpOnly`, `Secure` flags).
Use Cases	Ideal for storing larger amounts of data for client-side usage only.	Often used for small data such as authentication tokens (session management), user preferences, etc.
Ease of Use	Easy to use via JavaScript APIs (`localStorage` and `sessionStorage`).	Requires more configuration (expiration, path, domain, secure, HttpOnly, etc.).
Security Enhancements	Web storage doesn't have security flags, but it's not automatically sent to the server.	Can be secured using `HttpOnly` (prevents access via JavaScript) and `Secure` (only sent over HTTPS).

1. Web Storage (localStorage and sessionStorage)

- **Storage Capacity**: Web Storage can store significantly more data (usually 5–10MB per origin), which makes it ideal for client-side storage that doesn't need to be sent to the server with every request.
- **Not Automatically Sent**: Data stored in Web Storage is **not automatically sent** to the server. This reduces the overhead in HTTP requests, making it more efficient for storing large amounts of data.
- **Simple API**: Web Storage is easy to use with `localStorage` and `sessionStorage` APIs in JavaScript. There is no need to worry about setting expiration, paths, or other metadata.

Example: Using `localStorage`

```
// Store data in localStorage
localStorage.setItem('theme', 'dark');

// Retrieve data
const theme = localStorage.getItem('theme');
console.log(theme); // 'dark'

// Remove data
localStorage.removeItem('theme');

// Clear all data
localStorage.clear();
```

When to Use Web Storage:

- Storing large amounts of client-side data (e.g., user settings, UI preferences, caching).
- Storing data that does not need to be sent to the server.
- When you need data to persist longer than a session (`localStorage`).

2. Cookies

- **Limited Capacity**: Cookies have a smaller storage capacity, usually **4KB per cookie**, making them less suitable for large amounts of data.
- **Automatically Sent with Every Request**: Cookies are automatically included in every HTTP request to the server, which can increase request size and slow down performance when too many or large cookies are used.
- **Expiration & Scope**: You can set an expiration date on a cookie, and cookies can be scoped to specific paths and domains. This flexibility is useful for managing session states or tracking user activity across a domain.

Example: Using Cookies in JavaScript

```
// Set a cookie
document.cookie = "username=JohnDoe; expires=Fri, 31 Dec 2024 12:00:00 UTC;
```

```
path=/";

// Get all cookies
console.log(document.cookie);

// Delete a cookie
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC;
path=/";
```

When to Use Cookies:

- Managing **session data**, such as authentication tokens or session IDs.
- When you need to send small amounts of data to the server with each request (such as user preferences or language settings).
- If you need the ability to set expiry dates, paths, or domains for data.

Security Considerations

1. **Cookies** can be secured using flags like:
   - `HttpOnly`: Prevents JavaScript from accessing the cookie, reducing the risk of XSS (cross-site scripting) attacks.
   - `Secure`: Ensures the cookie is sent only over HTTPS.
   - `SameSite`: Helps prevent CSRF (cross-site request forgery) by restricting how cookies are sent across sites.
2. **Web Storage** is accessible via JavaScript, which makes it vulnerable to XSS attacks if the site is not properly secured.

Comparison Table: `localStorage` vs `sessionStorage` vs Cookies

Feature	`localStorage`	`sessionStorage`	Cookies
Storage Capacity	5–10 MB	5–10 MB	4 KB per cookie
Expiration	Persists until deleted	Cleared on tab/window close	Can set expiration dates, or expires on session end
Scope	Shared across tabs/windows of the same origin	Unique to each tab/window	Sent with every HTTP request, across tabs on same domain
Data Sent with Requests	Not sent with requests	Not sent with requests	Automatically sent with every HTTP request
Security	Accessible via JavaScript	Accessible via JavaScript	Can be made secure using `HttpOnly`, `Secure` flags

Summary

- **Use `localStorage` or `sessionStorage`** for storing larger client-side data that does not need to be sent to the server, with longer-lasting data stored in `localStorage` and session-specific data stored in `sessionStorage`.
- **Use Cookies** for smaller pieces of data that must be shared between the client and server, such as authentication tokens or session management data, especially if you need data to be available across requests and tabs.

Each method has its strengths depending on the use case, but for most modern web applications, **Web Storage** is preferred for client-side data that doesn't need to be sent to the server, while **Cookies** are still necessary for state management where data must travel between the client and the server.

What is Primitive Data Types & Non Primitive Data Types?

In JavaScript, data types are broadly categorized into **primitive** and **non-primitive** (or reference) data types. This distinction is crucial for understanding how data is stored and handled in memory.

1. Primitive Data Types

Primitive data types represent a single value, and they are immutable (cannot be changed once created). These types are stored directly in the stack memory, and when you assign or compare them, JavaScript works with the actual value.

Primitive Data Types in JavaScript:

1. **`string`**: Represents a sequence of characters (text).

```
let name = "John";
```

2. **`number`**: Represents both integers and floating-point numbers.

```
let age = 30;
let price = 19.99;
```

3. **`boolean`**: Represents `true` or `false`.

```
let isAvailable = true;
```

4. **`null`**: Represents an intentional absence of any object value.

```
let data = null;
```

5. **`undefined`**: Represents a variable that has been declared but has not yet been assigned a value.

```
let score; // value is undefined
```

6. **`symbol`** (introduced in ES6): Represents a unique identifier, often used for object property keys.

```
let id = Symbol('uniqueId');
```

7. **`bigint`** (introduced in ES2020): Represents large integers that are beyond the safe integer limit for the `number` type.

```
let largeNumber = 9007199254740991n;  // n denotes BigInt
```

Characteristics of Primitive Data Types:

- **Immutable**: Once a primitive value is created, it cannot be modified. For example, when you manipulate strings, JavaScript creates new strings rather than changing the original one.
- **Stored by value**: When assigning or copying a primitive value to another variable, a copy of the value is made. For instance:

```
let a = 10;
let b = a;  // b gets a copy of the value in a
a = 20;
console.log(b); // 10 (remains unchanged)
```

2. Non-Primitive Data Types

Non-primitive data types (also called reference types) are more complex structures that can store collections of values or more sophisticated objects. They are stored in heap memory, and when you assign or compare them, JavaScript handles the reference (or address) to the location in memory, not the actual value.

Non-Primitive Data Types in JavaScript:

1. **`object`**: The most common non-primitive data type. It can hold collections of key-value pairs.
   - **Objects**: General containers for any combination of data and functions.

     ```
     let person = { name: "Alice", age: 25 };
     ```

   - **Arrays**: Ordered lists of values, which are also objects.

     ```
     let fruits = ["apple", "banana", "cherry"];
     ```

   - **Functions**: A special type of object that can be invoked.

     ```
     function greet() { return "Hello!"; }
     ```

   - **Dates, Sets, Maps**: Special object types introduced in modern JavaScript.

     ```
     let today = new Date();
     let mySet = new Set([1, 2, 3]);
     ```

Characteristics of Non-Primitive Data Types:

- **Mutable**: Non-primitive data can be changed even after being created. For example, you can add or remove properties from an object or elements from an array.

- **Stored by reference**: When assigning or copying a non-primitive type, JavaScript creates a reference to the original memory location. For example:

```
let obj1 = { name: "Alice" };
let obj2 = obj1;  // obj2 points to the same memory as obj1
obj1.name = "Bob";
console.log(obj2.name); // "Bob" (both refer to the same object)
```

  - Since both `obj1` and `obj2` point to the same reference in memory, changes to one affect the other.

Example of Mutable Behavior:

```
let arr1 = [1, 2, 3];
let arr2 = arr1;
arr1.push(4);
console.log(arr2); // Output: [1, 2, 3, 4] (both point to the same array in memory)
```

Key Differences Between Primitive and Non-Primitive Types:

Feature	Primitive Data Types	Non-Primitive (Reference) Data Types
Storage	Stored by value in stack memory	Stored by reference in heap memory
Mutability	Immutable (cannot be changed)	Mutable (can be changed)
Comparison	Compared by value (actual data)	Compared by reference (memory location)
Copy Behavior	Creates a copy of the value	Copies the reference (points to the same object)
Examples	`string`, `number`, `boolean`, etc.	`object`, `array`, `function`, etc.

Summary:

- **Primitive Data Types**: Simple values like `string`, `number`, `boolean`, `null`, `undefined`, `symbol`, and `bigint`. They are immutable and stored by value.
- **Non-Primitive Data Types**: Complex types like `object`, `array`, and `function`. They are mutable and stored by reference.

Understanding the distinction between primitive and non-primitive data types helps in managing how data is copied, compared, and manipulated in JavaScript.