**Debounce and throttling**

Debounce and throttling are techniques used in web development and user interface design to control the frequency of certain actions, particularly those triggered by events like user interactions or API requests. These techniques help optimize performance, prevent excessive resource consumption, and improve user experience.

**\*\*Debounce:\*\***

Debounce delays the execution of a function until after a period of inactivity, often used to control events that trigger rapidly in succession.

**Example 1: Search Input**

Suppose you have a search input field that fetches search results from an API as the user types. Without debounce, the API might be hit with multiple requests in quick succession as the user types each character. By applying debounce, you delay the API request until the user pauses typing, ensuring that only one request is made after a brief period of inactivity.

**\*\*Throttling:\*\***

Throttling limits the rate at which a function or action can be executed, preventing it from being triggered too frequently.

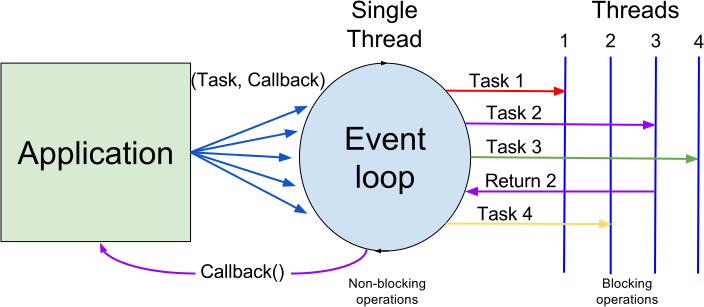
Throttling involves limiting the rate at which a function or action can be executed. This prevents a function from being executed more frequently than a specified interval. Throttling is useful to prevent an action from being performed too often, which could lead to performance issues or unwanted behavior.

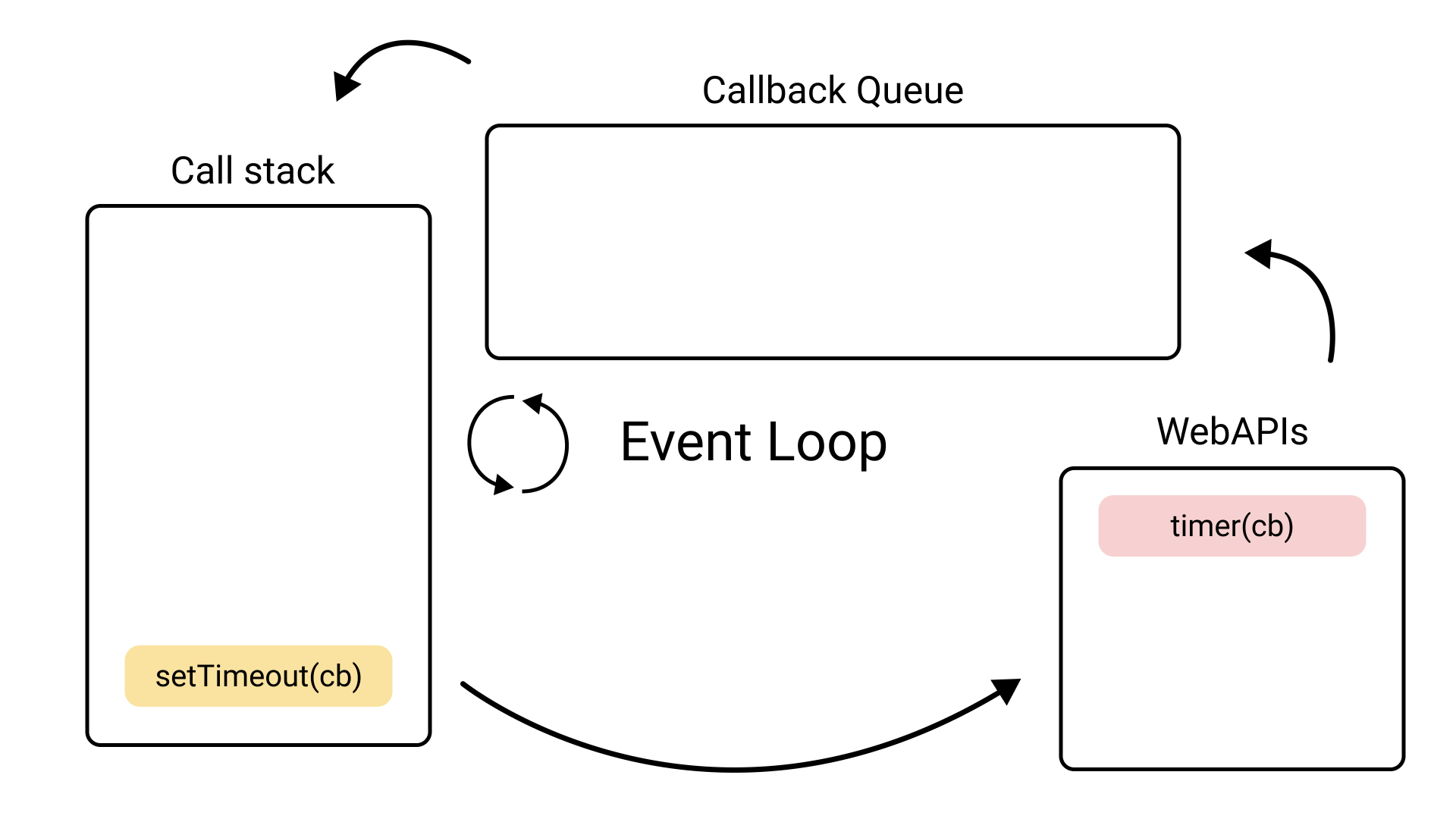
**Example 2: Scrolling Event**

Consider a scenario where you're implementing an infinite scrolling feature on a webpage, where more content is loaded as the user scrolls down. Throttling can be applied to the scroll event handler to ensure that content is loaded at a controlled rate. Without throttling, the event could fire rapidly as the user scrolls, potentially causing excessive network requests and slowing down the page.

Both debounce and throttling are important tools to optimize the performance of web applications and provide a smoother user experience by controlling the frequency of certain actions.

**How event loop works ?**

****

****

In JavaScript, the event loop is responsible for executing code, handling events, and managing tasks in a non-blocking manner. It ensures that while one piece of code is being executed, other events or tasks can still be processed without blocking the main thread.

Here's a simplified explanation of how the event loop works:

**Call Stack:**The call stack is a data structure that keeps track of the currently executing functions in a program. When a function is called, it is pushed onto the call stack, and when the function completes its execution, it is popped off the stack. The call stack operates on a last-in, first-out (LIFO) basis. It maintains the order of function calls and helps manage the flow of execution in a synchronous manner.

**Web APIs and Callback Queue:** When you perform asynchronous operations like setTimeout, DOM events, or AJAX requests, these operations are moved to the Web APIs provided by the browser. Once the Web API has completed the task, it sends the corresponding callback function (if any) to the Callback Queue(also known as the event queue or message queue).

Callback Queue is a data structure that holds tasks (usually functions or messages) that are scheduled to be executed in response to asynchronous events. These events can include user interactions, timer expirations, or AJAX responses. Asynchronous tasks are placed in the event queue when they are ready to be processed, and they await execution.

The event queue is managed by the event loop, which continuously checks the state of the call stack. When the call stack is empty, the event loop dequeues tasks from the event queue and places them onto the call stack for execution. This ensures that asynchronous operations can be handled without blocking the main thread of execution.

**Event Loop:** The event loop continuously checks the call stack and the callback queue. If the call stack is empty, it takes the first function from the callback queue and pushes it onto the call stack, initiating its execution.

**Execution and Completion:** The function on top of the call stack is executed. If it contains asynchronous code (like a setTimeout callback), the asynchronous operation is offloaded to the Web API, and the function completes. The event loop then waits for the call stack to be empty again.

**Rinse and Repeat:** This process repeats, allowing asynchronous operations to be handled while keeping the main thread available for other tasks.

Note : The call stack keeps track of the current execution context and function calls, ensuring synchronous execution.

The event queue holds asynchronous tasks that are waiting to be executed in response to events, managed by the event loop.

# **Explain Scope and Scope Chain in JavaScript(Needs to Explore)**

**Global Execution Context (GEC)**

Whenever the JavaScript engine receives a script file, it first creates a default Execution Context known as the Global Execution Context (GEC).

The GEC is the base/default Execution Context where all JavaScript code that is not inside of a function gets executed.

For every JavaScript file, there can only be one GEC.

**Function Execution Context (FEC)**

Whenever a function is called, the JavaScript engine creates a different type of Execution Context known as a Function Execution Context (FEC) within the GEC to evaluate and execute the code within that function.

Since every function call gets its own FEC, there can be more than one FEC in the run-time of a script.

Reference : <https://www.freecodecamp.org/news/execution-context-how-javascript-works-behind-the-scenes/#:~:text=Whenever%20the%20JavaScript%20engine%20receives%20a%20script%20file%2C,JavaScript%20file%2C%20there%20can%20only%20be%20one%20GEC.>

**Garbage Collector (Needs to Explore)**

**what is the difference between bind apply and call in javascript**

In JavaScript, `bind`, `apply`, and `call` are three methods that allow you to manipulate how a function is invoked and control its context (the value of `this` inside the function). While they serve similar purposes, they have distinct differences in terms of their syntax and usage:

1. \*\*`bind` Method:\*\*

The `bind` method creates a new function with a specified `this` value and any provided arguments. It doesn't immediately invoke the function but returns a new function that can be called later.

Syntax:

const newFunction = originalFunction.bind(thisValue, arg1, arg2, ...);

Example:

function greet(name) {

console.log(`Hello, ${name}! I'm ${this.title}.`);

}

const person = {

title: 'Mr.'

};

const greetPerson = greet.bind(person, 'John');

greetPerson(); // Output: Hello, John! I'm Mr.

2. \*\*`apply` Method:\*\*

The `apply` method allows you to call a function with a specified `this` value and an array or array-like object of arguments. It immediately invokes the function.

Syntax:

```javascript

originalFunction.apply(thisValue, [arg1, arg2, ...]);

```

Example:

```javascript

function sum(a, b) {

return a + b;

}

const result = sum.apply(null, [5, 3]);

console.log(result); // Output: 8

```

3. \*\*`call` Method:\*\*

The `call` method is similar to `apply`, but it allows you to pass arguments directly instead of an array. Like `apply`, it immediately invokes the function.

Syntax:

```javascript

originalFunction.call(thisValue, arg1, arg2, ...);

```

Example:

```javascript

function introduce(name, age) {

console.log(`My name is ${name} and I am ${age} years old. Nice to meet you!`);

}

introduce.call(null, 'Alice', 30); // Output: My name is Alice and I am 30 years old. Nice to meet you!

```

Key differences:

- `bind` returns a new function with the specified `this` value and arguments, while `apply` and `call` immediately invoke the function.

- `apply` and `call` both accept the function arguments differently. `apply` uses an array or array-like object, while `call` accepts arguments directly.

- All three methods allow you to control the `this` value of a function during its execution.

These methods are powerful tools for managing function context and behavior, and their choice depends on your specific use case and preferences.

**What is memory leak in javascript and example ?**

A memory leak in JavaScript occurs when a program unintentionally retains references to objects in memory that are no longer needed, preventing the JavaScript engine's garbage collector from reclaiming that memory. Over time, these memory leaks can lead to increased memory consumption and degraded performance of your application.

Here's a simple example of a memory leak in JavaScript:

let element = document.getElementById("myElement");

function handleClick() {

// Some code...

}

element.addEventListener("click", handleClick);

In this example, an event listener is attached to an HTML element. However, if you repeatedly remove the element from the DOM without also removing the event listener, a memory leak can occur. The event listener holds a reference to the `handleClick` function, which in turn holds a reference to the `element`. Even if the element is no longer visible or used, it cannot be garbage collected because the event listener keeps the reference alive.

To avoid this memory leak, you should remove the event listener before removing the element from the DOM:

```javascript

let element = document.getElementById("myElement");

function handleClick() {

// Some code...

}

element.addEventListener("click", handleClick);

// Later, when you want to remove the element and event listener

element.removeEventListener("click", handleClick);

element.parentNode.removeChild(element);

```

By removing the event listener before removing the element from the DOM, you ensure that there are no lingering references preventing the JavaScript engine from properly garbage collecting the memory.

Memory leaks can also occur in more complex scenarios, such as when closures capture references to variables that are no longer needed or when global variables are not properly cleaned up.

**Pure and Impure Function (Needs to explore)**

**Generator Function**

# Generators

Regular functions return only one, single value (or nothing).

Generators can return (“yield”) multiple values, one after another, on-demand. They work great with [iterables](https://javascript.info/iterable), allowing to create data streams with ease.

## [Generator functions](https://javascript.info/generators" \l "generator-functions)

To create a generator, we need a special syntax **function\***, so-called “generator function”.

A generator function in JavaScript is a special type of function that can be paused and resumed during its execution. It allows you to generate a sequence of values lazily, meaning you can yield values one at a time and then continue execution from where you left off. This can be particularly useful for generating sequences of data that might be memory-intensive or time-consuming to compute all at once.

To create a generator function, you use the function\* syntax. Inside a generator function, you use the yield keyword to yield values to the caller. When the generator is called, it doesn't actually execute the entire function immediately; instead, it returns an iterator object that you can use to control the execution of the generator.

Here's an example of a simple generator function that generates a sequence of numbers:

function\* numberGenerator() {

let i = 0;

while (true) {

yield i++;

}

}

const iterator = numberGenerator();

console.log(iterator.next().value); // 0

console.log(iterator.next().value); // 1

console.log(iterator.next().value); // 2

// ...

In this example, the numberGenerator generator function yields an incremented value each time it's called. The iterator.next() method is used to advance the generator and retrieve the next yielded value.

**Shallow copy and Deep copy(Needs to explore)**

**Event Delegation**

Event Delegation is basically a pattern to handle events efficiently. Instead of adding an event listener to each and every similar element, we can add an event listener to a parent element and call an event on a particular target using the .target property of the event object.

Let’s see an example with and without event delegation

const customUI = document.createElement('ul');

for (var i = 1; i <= 10; i++) {

const newElement = document.createElement('li');

newElement.textContent = "This is line " + i;

newElement.addEventListener('click', () => {

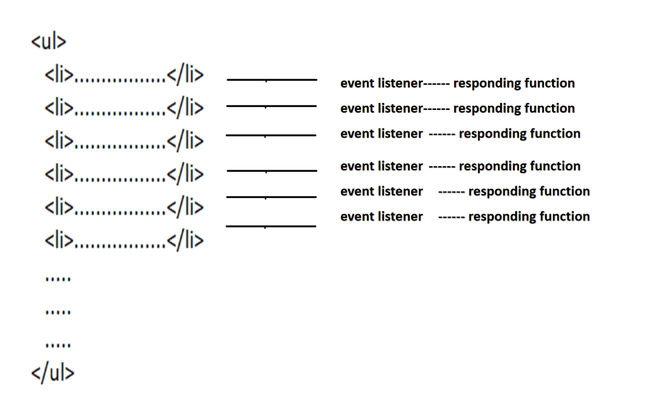
console.log('Responding')

})

customUI.appendChild(newElement);

}

The above code will associate the function with every <li> element that is shown in the below image. We are creating an <ul> element, attaching too many <li> elements, and attaching an event listener with a responding function to each paragraph as we create it.



Without Event Delegation

Implementing the same functionalities with an alternate approach. In this approach, we will associate the same function with all event listeners. We are creating too many responding functions (that all actually do the exact same thing). We could extract this function and just reference the function instead of creating too many functions:

const customUI = document.createElement('ul');

function responding() {

console.log('Responding')

}

for (var i = 1; i <= 10; i++) {

const newElement = document.createElement('li');

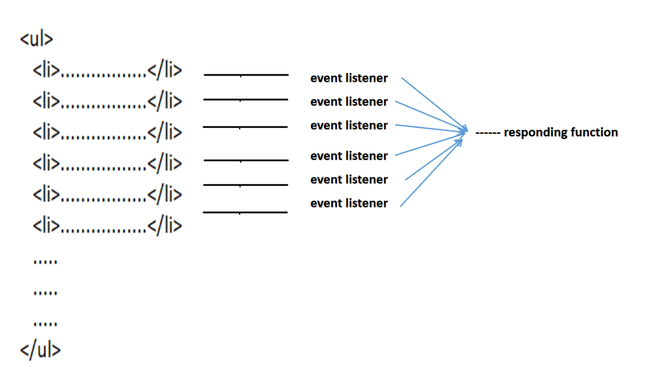
newElement.textContent = "This is line " + i;

newElement.addEventListener('click', responding)

customUI.appendChild(newElement);

}

The functionality of the above code is shown below –



Without Event Delegation

In the above approach, we still have too many event listeners pointing to the same function. Now implementing the same functionalities using a single function and single event.

const customUI = document.createElement('ul');

function responding() {

console.log('Responding')

}

for (var i = 1; i <= 10; i++) {

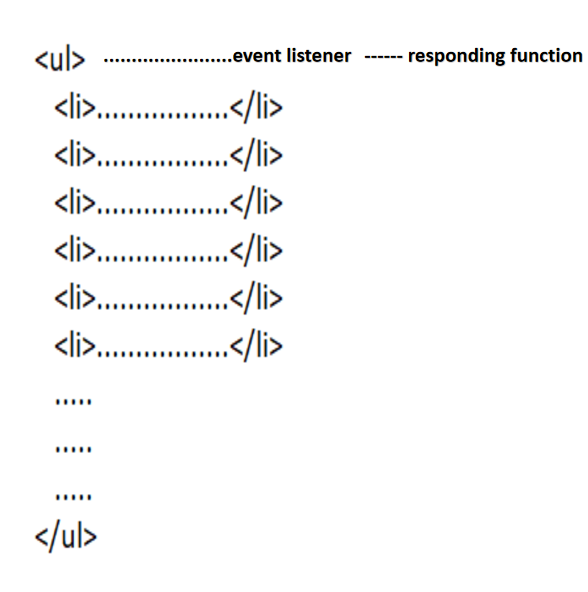
const newElement = document.createElement('li');

newElement.textContent = "This is line " + i;

customUI.appendChild(newElement);

}

customUI.addEventListener('click', responding)



Now there is a single event listener and a single responding function. In the above-shown method, we have improved the performance, but we have lost access to individual <li> elements so to resolve this issue, we will use a technique called event delegation.

The event object has a special property call .target which will help us in getting access to individual <li> elements with the help of phases.

Steps:

* <ul> element is clicked.
* The event goes in the capturing phase.
* It reaches the target (<li> in our case).
* It switches to the bubbling phase.
* When it hits the <ul> element, it runs the event listener.
* Inside the listener function event.target is the element that was clicked.
* Event.target provides us access to the <li> element that was clicked.

The .nodeName property of the .target allows us to identify a specific node. If our parent element contains more than one child element then we can identify specific elements by using the .nodeName property.

const customUI = document.createElement('ul');

function responding(evt) {

if (evt.target.nodeName === 'li')

console.log('Responding')

}

for (var i = 1; i <= 10; i++) {

const newElement = document.createElement('li');

newElement.textContent = "This is line " + i;

customUI.appendChild(newElement);

}

customUI.addEventListener('click', responding);

**Set, WeakSet, Map,WeakMap (Needs To Explore)**