JavaScript

Performance Optimization: -

Debouncing & Throttling: -

Debouncing and throttling are techniques used to control how frequently a function executes, especially for event handlers like scrolling, resizing, and typing. These methods improve performance by preventing unnecessary function calls.

Debouncing: -

Debouncing ensures that a function is executed only after a specified delay after the last event occurs. If the event occurs again within the delay period, the timer resets.

Use Cases: -

Search input (trigger API calls only when typing stops)

Window resizing (avoid multiple calculations)

Button clicks (prevent accidental double-clicks)

Example: - Debounce Function

```
function debounce(func, delay) {
   let timer;
   return function(…args) {
      clearTimeout(timer); // Clear the previous timer
      timer = setTimeout(() => func.apply(this, args), delay); // Set new timer
   };
}
function onSearch(event) {
   console.log(“Searching for:”, event.target.value);
}
const searchInput = document.getElementById(“search”);
searchInput.addEventListener(“input”, debounce(onSearch, 500)); // Waits 500ms after typing stops
```

Throttling: -

Throttling ensures that a function executes at most once in a specified time interval, regardless of how many times the event is triggered.

Use Cases: -

Scroll events (optimize infinite scrolling)

Resize events (avoid excessive reflows)

Button clicks (limit rapid clicks)

Example: - Throttle Function

```
function throttle(func, limit) {
   let lastCall = 0;
   return function(...args) {
      const now = Date.now();
      if (now - lastCall >= limit) {
         lastCall = now;
         func.apply(this, args);
      }
   };
}
function onScroll() {
   console.log("Scrolling...");
}
window.addEventListener("scroll", throttle(onScroll, 1000)); // Executes once per second while scrolling
```

Key Differences Between Debouncing & Throttling: -

Feature	**Debouncing**	**Throttling**
Execution	Delayed until user stops triggering events	Executes at fixed intervals
Frequency Control	Executes only once after the delay	Executes at most once per interval
Use Case	Typing in search bars, resizing windows	Scrolling, button clicks

Lazy Loading: -

Lazy loading is a technique that delays the loading of non-essential resources (such as images, videos, or components) until they are needed.

This improves performance and reduces initial page load time.

Benefits of Lazy Loading: -

Faster initial page load

Reduced bandwidth usage

Better user experience

Optimized performance for large-scale applications

Lazy Loading Images: -

Images can be lazy-loaded using the loading="lazy" attribute in HTML:

<img src="large-image.jpg" alt="Example Image" loading="lazy">

Browser automatically loads the image when it enters the viewport.

Lazy Loading JavaScript Files (Dynamic Import): -

Instead of loading all scripts upfront, use dynamic imports.

```
document.getElementById("btn").addEventListener("click", async () => {
   const module = await import("./heavyModule.js");
   module.default();
});
```

JavaScript files load only when the button is clicked.

Code Splitting & Tree Shaking: -

Code Splitting and Tree Shaking are optimization techniques used in JavaScript applications to improve performance by reducing the size of the final bundle.

1.Code Splitting: -

Code Splitting allows you to split your JavaScript code into smaller bundles that are loaded dynamically when needed. This helps reduce the initial load time of web applications.

Benefits: -

Improves performance by reducing initial bundle size.

Loads only necessary code.

Optimizes large-scale applications

Methods of Code Splitting: -

a) Dynamic Imports (Lazy Loading with import()):

JavaScript allows dynamically importing modules when required.

```
document.getElementById("btn").addEventListener("click", async () => {
   const module = await import("./heavyModule.js");
   module.default();
```

```
});
```

The module is only loaded when the button is clicked.

b) Code Splitting in Webpack

Webpack supports code splitting using import().

```
// main.js
import("./moduleA.js").then(module => {
    module.functionA();
});
```

Webpack automatically creates separate chunks for these modules.

```
async function loadModule() {
    const module = await import("./moduleA.js");
    module.functionA();
}
loadModule();
```

2.Tree Shaking: -

Tree Shaking is a technique used to remove unused code (dead code) from JavaScript bundles, reducing file size.

Benefits: -

Removes unused functions and variables.

Reduces bundle size.

Improves application efficiency.

How Tree Shaking Works?

Tree shaking relies on ES6 module imports (import/export) to analyze and remove unused code.

Example: -

```
// utils.js
export function usedFunction() {
    return "This is used";
}
export function unusedFunction() {
    return "This is never used";
```

```
}
// main.js
import { usedFunction } from "./utils.js";
console.log(usedFunction());
```

With tree shaking, unusedFunction will be removed from the final bundle.

Tree shaking only works with ES6 modules (import/export), not CommonJS (require()).

Use: import { funcA } from "./module";

Avoid: const funcA = require("./module");