Javascript

Created by	Ⓡ Rutik jadhav
Created time	@July 12, 2024 3:43 PM
Tags	

Topic 1: Static vs Dynamic Typing

Introduction

Typing systems in programming define how variables are declared and how their types are handled during code execution. Static typing and dynamic typing represent two different approaches to type-checking in programming languages. Knowing their differences can help in choosing the right language for a specific project.

Key Concepts and Definitions

- **Statically Typed Languages**: Variables have a fixed type determined at compile-time. Examples: Java, C++, Swift.
    - Example:

    ```
    int number = 10; // type is explicitly declared
    ```

- **Dynamically Typed Languages**: Variables are not bound to a specific type and are determined at runtime. Examples: Python, JavaScript.
    - Example:

    ```
    number = 10 # type is inferred dynamically
    ```

Step-by-Step Explanation

1. **Static Typing**:

    - Errors are caught at compile-time.
    - Improves performance and reduces runtime bugs.
    - Requires more boilerplate code (explicit type declarations).
2. **Dynamic Typing**:
    - Errors are detected at runtime.
    - Increases flexibility and speeds up prototyping.
    - Risk of unexpected type-related errors during execution.

Examples

- Statically Typed:

```
double price = 9.99; // Explicit type assignment
price = "ten"; // Compile-time error
```

- Dynamically Typed:

```
let price = 9.99;
price = "ten"; // No error at declaration, may fail during runtime
```

Practice Exercises

1. Identify whether the following snippets use static or dynamic typing.
2. Write examples of declaring variables in a statically typed language and a dynamically typed language.

Summary

Static typing provides safety and efficiency but requires more setup. Dynamic typing offers flexibility at the cost of potential runtime errors.

Additional Resources

- Static vs Dynamic Typing on GeeksforGeeks

- JavaScript Dynamic Typing

Topic 2: `let`, `var`, and `const` (Difference in Scopes)

Introduction

In JavaScript, variable declarations are handled using `let`, `var`, and `const`. Each has its own scope rules and use cases, influencing how variables are managed in the code.

Key Concepts and Definitions

- **`var`**: Function-scoped, can be redeclared, and prone to hoisting issues.
- **`let`**: Block-scoped, introduced in ES6, safer for modern development.
- **`const`**: Block-scoped and immutable (cannot be reassigned).

Step-by-Step Explanation

1. **Scope**:
   - `var` is function-scoped:

     ```
     function example() {
       if (true) {
         var x = 10;
       }
       console.log(x); // 10
     }
     ```

   - `let` and `const` are block-scoped:

     ```
     if (true) {
       let x = 10;
       const y = 20;
     }
     console.log(x, y); // ReferenceError
     ```

2. **Redeclaration**:

   - `var` allows redeclaration; `let` and `const` do not:

     ```
     var x = 1;
     var x = 2; // Allowed
     let y = 1;
     let y = 2; // SyntaxError
     ```

3. **Mutability**:

   - `const` prevents reassignment but does not make objects immutable:

     ```
     const obj = { name: 'Alice' };
     obj.name = 'Bob'; // Allowed
     obj = {}; // Error
     ```

Examples

- `var` hoisting issue:

  ```
  console.log(x); // Undefined
  var x = 10;
  ```

- `let` and `const` prevent this:

  ```
  console.log(y); // ReferenceError
  let y = 10;
  ```

Practice Exercises

1. Rewrite code using `var` to use `let` or `const`.
2. Explain the output of code snippets that mix scopes.

Summary

Use `let` and `const` for predictable scoping and to follow modern best practices. Avoid `var` in new codebases.

Additional Resources

- MDN Documentation on let

Topic 3: Functional Scopes

Introduction

Functional scope is the area in code where a variable is accessible. In JavaScript, variables declared with `var` are function-scoped, limiting their access to the containing function.

Key Concepts and Definitions

- **Function Scope**: Variables declared inside a function are not accessible outside of it.
  - Example:

    ```javascript
    function greet() {
      var message = 'Hello!';
      console.log(message);
    }
    console.log(message); // ReferenceError
    ```

Step-by-Step Explanation

1. **Variable Access**:
   - Only accessible within the same function.
2. **Nested Functions**:
   - Inner functions can access variables from their parent scope.

     ```javascript
     function outer() {
       var outerVar = 'Outer';
     ```

```
  function inner() {
    console.log(outerVar);
  }
  inner();
}
```

Practice Exercises

1. Write a function and declare variables within it. Test their accessibility outside the function.
2. Nest functions and observe variable scope.

Summary

Function scope ensures variables are encapsulated within functions, preventing unintended access.

Topic 4: Callback Functions and Higher-Order Functions

Learning Objectives

- Define callback and higher-order functions.
- Understand their relationship.
- Learn to use callbacks in JavaScript.

Introduction

Functions in JavaScript can be passed as arguments or returned from other functions. Callback functions and higher-order functions make this possible, providing powerful tools for asynchronous programming.

Key Concepts and Definitions

- **Callback Function**: A function passed as an argument to another function.
    - Example:

```
function greet(name) {
  console.log(`Hello, ${name}`);
}
function processUserInput(callback) {
  let name = 'Alice';
  callback(name);
}
processUserInput(greet);
```

- **Higher-Order Function**: A function that takes other functions as arguments or returns them.
  - Example:

    ```
    function higherOrder(fn) {
      fn();
    }
    higherOrder(() => console.log('Callback executed!'));
    ```

Step-by-Step Explanation

1. **Defining Callbacks**:
   - Create a function to handle a task.
2. **Passing Callbacks**:
   - Use callbacks to decouple logic.
3. **Higher-Order Functions**:
   - Functions like `map`, `filter`, and `reduce` are built-in higher-order functions.

Examples

- Using `map`:

  ```
  const numbers = [1, 2, 3];
  const doubled = numbers.map(n => n * 2);
  ```

```
console.log(doubled); // [2, 4, 6]
```

Practice Exercises

1. Write a higher-order function that accepts a callback to perform operations on numbers.
2. Use `filter` to extract even numbers from an array.

Summary

Callbacks and higher-order functions enable flexible and reusable code structures, essential for modern JavaScript programming.

Additional Resources

- MDN Documentation on Callbacks
- Eloquent JavaScript

Topic 5: Global Execution Context

Learning Objectives

- Understand the Global Execution Context (GEC) in JavaScript.
- Learn what happens during the creation and execution phases of the GEC.

Introduction

The **Global Execution Context** (GEC) is the default execution context in JavaScript. It is created when the JavaScript code is executed and manages global variables, functions, and objects like `window` or `global`.

Key Concepts and Definitions

- **Global Execution Context (GEC)**: The environment where global code (not inside any function) is executed.
- **Phases of Execution Context**:
  - **Creation Phase**:

        - Memory is allocated for variables and functions.
        - Variables are initialized with `undefined`.
        - Functions are hoisted with their full definitions.
    - **Execution Phase**:
        - Code is executed line by line, and values are assigned to variables.

Examples

- Creation and Execution:

```
console.log(x); // undefined (hoisted)
var x = 10; // Assigned in execution phase
function greet() {
  console.log('Hello');
}
greet(); // "Hello" (function fully hoisted)
```

Practice Exercises

1. Write code that uses global variables and observe their behavior during the creation phase.
2. Predict the output of code involving variable and function declarations.

Topic 6: Functional Execution Context

Introduction

A **Functional Execution Context** (FEC) is created whenever a function is invoked. It handles the function's variables, parameters, and inner functions.

Key Concepts and Definitions

- **FEC**: Execution context specific to a function call.
- **Scope Chain**: Includes the function's local scope and its parent scopes.

Examples

- Function Call Creates FEC:

```
function add(a, b) {
  return a + b;
}
add(5, 10); // Creates a new FEC
```

Practice Exercises

1. Create a nested function to observe scope chaining within an FEC.

Topic 7: Hoisting and Temporal Dead Zone (TDZ)

Learning Objectives

- Learn how hoisting affects variables and functions.
- Understand the concept of Temporal Dead Zone (TDZ) with `let` and `const`.

Introduction

Hoisting refers to the process where variable and function declarations are moved to the top of their scope during the creation phase. The **Temporal Dead Zone (TDZ)** is the period between entering a scope and declaring a variable with `let` or `const`.

Key Concepts and Definitions

- **Hoisting**:
  - Variables declared with `var` are hoisted and initialized with `undefined`.
  - Functions are hoisted with their full definitions.
- **TDZ**:
  - Variables declared with `let` or `const` exist in the scope but cannot be accessed before declaration.

Examples

- Hoisting:

```
console.log(a); // undefined
var a = 10;

console.log(b); // ReferenceError (TDZ)
let b = 20;
```

- TDZ:

```
{
  console.log(x); // ReferenceError
  let x = 5;
}
```

Practice Exercises

1. Identify the TDZ in code with `let` and `const`.
2. Write examples to demonstrate function and variable hoisting.

Topic 8: Closures and Lexical Scoping

Introduction

A **closure** is formed when an inner function retains access to variables in its outer scope, even after the outer function has returned. **Lexical Scoping** determines how variable names are resolved based on the location of their declaration.

Key Concepts and Definitions

- **Closure**:
  - A combination of a function and its lexical scope.
  - Allows data encapsulation.

- **Lexical Scoping**:
  - Variable access is determined by the position in the source code.

Examples

- Closure:

```
function outer() {
  let counter = 0;
  return function inner() {
    counter++;
    console.log(counter);
  };
}
const increment = outer();
increment(); // 1
increment(); // 2
```

- Lexical Scoping:

```
const globalVar = 'Global';
function outer() {
  const outerVar = 'Outer';
  function inner() {
    console.log(globalVar, outerVar); // Access both varia
bles
  }
  inner();
}
outer();
```

Practice Exercises

1. Create a closure to manage a private variable.
2. Write a nested function and observe lexical scope behavior.

Topic 9: IIFE and Function Currying

Introduction

IIFE(Immediately Invoked Function Expression) and currying are functional programming concepts. An **IIFE** is a function executed immediately after its declaration. **Currying** transforms a function with multiple arguments into a series of functions, each taking a single argument.

Key Concepts and Definitions

- **IIFE**:
  - Syntax: `(function() { ... })();`
  - Used to avoid polluting the global scope.
- **Currying**:
  - Breaking down a function into smaller functions that return another function.
  - Improves reusability and composition.

Examples

- IIFE:

```
(function() {
  let message = 'IIFE executed';
  console.log(message);
})();
```

- Currying:

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

```
const double = multiply(2);
console.log(double(5)); // 10
```

Practice Exercises

1. Write an IIFE to initialize variables without affecting the global scope.
2. Create a curried function to calculate discounts on prices.

Summary

Concept	Key Idea
GEC	Default context for global code.
FEC	Context for each function call.
Hoisting	Variables/functions moved to the top of their scope.
TDZ	`let` / `const` variables inaccessible before declaration.
Closures	Inner functions retaining access to outer variables.
Lexical Scoping	Variable resolution based on source code location.
IIFE	Functions executed immediately after definition.
Currying	Breaking a function into a sequence of unary functions.

Topic 1: Static vs Dynamic Typing

Introduction

Typing systems in programming define how variables are declared and how their types are handled during code execution. Static typing and dynamic typing represent two different approaches to type-checking in programming languages. Knowing their differences can help in choosing the right language for a specific project.

Key Concepts and Definitions

- **Statically Typed Languages**: Variables have a fixed type determined at compile-time. Examples: Java, C++, Swift.

- Example:

  ```
  int number = 10; // type is explicitly declared
  ```

- **Dynamically Typed Languages**: Variables are not bound to a specific type and are determined at runtime. Examples: Python, JavaScript.
  - Example:

  ```
  number = 10 # type is inferred dynamically
  ```

Step-by-Step Explanation

1. **Static Typing**:
   - Errors are caught at compile-time.
   - Improves performance and reduces runtime bugs.
   - Requires more boilerplate code (explicit type declarations).
2. **Dynamic Typing**:
   - Errors are detected at runtime.
   - Increases flexibility and speeds up prototyping.
   - Risk of unexpected type-related errors during execution.

Examples

- Statically Typed:

  ```
  double price = 9.99; // Explicit type assignment
  price = "ten"; // Compile-time error
  ```

- Dynamically Typed:

  ```
  let price = 9.99;
  price = "ten"; // No error at declaration, may fail during
  ```

```
runtime
```

Practice Exercises

1. Identify whether the following snippets use static or dynamic typing.
2. Write examples of declaring variables in a statically typed language and a dynamically typed language.

Summary

Static typing provides safety and efficiency but requires more setup. Dynamic typing offers flexibility at the cost of potential runtime errors.

Additional Resources

- Static vs Dynamic Typing on GeeksforGeeks
- JavaScript Dynamic Typing

Topic 2: `let`, `var`, and `const` (Difference in Scopes)

Introduction

In JavaScript, variable declarations are handled using `let`, `var`, and `const`. Each has its own scope rules and use cases, influencing how variables are managed in the code.

Key Concepts and Definitions

- `var`: Function-scoped, can be redeclared, and prone to hoisting issues.
- `let`: Block-scoped, introduced in ES6, safer for modern development.
- `const`: Block-scoped and immutable (cannot be reassigned).

Step-by-Step Explanation

1. **Scope**:
    - `var` is function-scoped:

```
function example() {
  if (true) {
    var x = 10;
  }
  console.log(x); // 10
}
```

- `let` and `const` are block-scoped:

```
if (true) {
  let x = 10;
  const y = 20;
}
console.log(x, y); // ReferenceError
```

2. **Redeclaration**:

- `var` allows redeclaration; `let` and `const` do not:

```
var x = 1;
var x = 2; // Allowed
let y = 1;
let y = 2; // SyntaxError
```

3. **Mutability**:

- `const` prevents reassignment but does not make objects immutable:

```
const obj = { name: 'Alice' };
obj.name = 'Bob'; // Allowed
obj = {}; // Error
```

Examples

- `var` hoisting issue:

```
console.log(x); // Undefined
var x = 10;
```

- `let` and `const` prevent this:

```
console.log(y); // ReferenceError
let y = 10;
```

Practice Exercises

1. Rewrite code using `var` to use `let` or `const`.
2. Explain the output of code snippets that mix scopes.

Summary

Use `let` and `const` for predictable scoping and to follow modern best practices. Avoid `var` in new codebases.

Additional Resources

- MDN Documentation on let

Topic 3: Functional Scopes

Introduction

Functional scope is the area in code where a variable is accessible. In JavaScript, variables declared with `var` are function-scoped, limiting their access to the containing function.

Key Concepts and Definitions

- **Function Scope**: Variables declared inside a function are not accessible outside of it.
  - Example:

```javascript
function greet() {
  var message = 'Hello!';
  console.log(message);
}
console.log(message); // ReferenceError
```

Step-by-Step Explanation

1. **Variable Access**:
   - Only accessible within the same function.
2. **Nested Functions**:
   - Inner functions can access variables from their parent scope.

   ```javascript
   function outer() {
     var outerVar = 'Outer';
     function inner() {
       console.log(outerVar);
     }
     inner();
   }
   ```

Practice Exercises

1. Write a function and declare variables within it. Test their accessibility outside the function.
2. Nest functions and observe variable scope.

Summary

Function scope ensures variables are encapsulated within functions, preventing unintended access.

Topic 4: Callback Functions and Higher-Order Functions

Learning Objectives

- Define callback and higher-order functions.
- Understand their relationship.
- Learn to use callbacks in JavaScript.

Introduction

Functions in JavaScript can be passed as arguments or returned from other functions. Callback functions and higher-order functions make this possible, providing powerful tools for asynchronous programming.

Key Concepts and Definitions

- **Callback Function**: A function passed as an argument to another function.
    - Example:

    ```
    function greet(name) {
      console.log(`Hello, ${name}`);
    }
    function processUserInput(callback) {
      let name = 'Alice';
      callback(name);
    }
    processUserInput(greet);
    ```

- **Higher-Order Function**: A function that takes other functions as arguments or returns them.
    - Example:

    ```
    function higherOrder(fn) {
      fn();
    }
    higherOrder(() => console.log('Callback executed!'));
    ```

Step-by-Step Explanation

1. **Defining Callbacks**:
   - Create a function to handle a task.
2. **Passing Callbacks**:
   - Use callbacks to decouple logic.
3. **Higher-Order Functions**:
   - Functions like `map`, `filter`, and `reduce` are built-in higher-order functions.

Examples

- Using `map`:

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

Practice Exercises

1. Write a higher-order function that accepts a callback to perform operations on numbers.
2. Use `filter` to extract even numbers from an array.

Summary

Callbacks and higher-order functions enable flexible and reusable code structures, essential for modern JavaScript programming.

Additional Resources

- MDN Documentation on Callbacks
- Eloquent JavaScript

Topic 5: Global Execution Context

Learning Objectives

- Understand the Global Execution Context (GEC) in JavaScript.
- Learn what happens during the creation and execution phases of the GEC.

Introduction

The **Global Execution Context** (GEC) is the default execution context in JavaScript. It is created when the JavaScript code is executed and manages global variables, functions, and objects like `window` or `global`.

Key Concepts and Definitions

- **Global Execution Context (GEC)**: The environment where global code (not inside any function) is executed.
- **Phases of Execution Context**:
  - **Creation Phase**:
    - Memory is allocated for variables and functions.
    - Variables are initialized with `undefined`.
    - Functions are hoisted with their full definitions.
  - **Execution Phase**:
    - Code is executed line by line, and values are assigned to variables.

Examples

- Creation and Execution:

```
console.log(x); // undefined (hoisted)
var x = 10; // Assigned in execution phase
function greet() {
  console.log('Hello');
}
greet(); // "Hello" (function fully hoisted)
```

Practice Exercises

1. Write code that uses global variables and observe their behavior during the creation phase.
2. Predict the output of code involving variable and function declarations.

Topic 6: Functional Execution Context

Introduction

A **Functional Execution Context** (FEC) is created whenever a function is invoked. It handles the function's variables, parameters, and inner functions.

Key Concepts and Definitions

- **FEC**: Execution context specific to a function call.
- **Scope Chain**: Includes the function's local scope and its parent scopes.

Examples

- Function Call Creates FEC:

```
function add(a, b) {
  return a + b;
}
add(5, 10); // Creates a new FEC
```

Practice Exercises

1. Create a nested function to observe scope chaining within an FEC.

Topic 7: Hoisting and Temporal Dead Zone (TDZ)

Learning Objectives

- Learn how hoisting affects variables and functions.
- Understand the concept of Temporal Dead Zone (TDZ) with `let` and `const`.

Introduction

Hoisting refers to the process where variable and function declarations are moved to the top of their scope during the creation phase. The **Temporal Dead Zone (TDZ)** is the period between entering a scope and declaring a variable with `let` or `const`.

Key Concepts and Definitions

- **Hoisting**:
  - Variables declared with `var` are hoisted and initialized with `undefined`.
  - Functions are hoisted with their full definitions.
- **TDZ**:
  - Variables declared with `let` or `const` exist in the scope but cannot be accessed before declaration.

Examples

- Hoisting:

  ```
  console.log(a); // undefined
  var a = 10;

  console.log(b); // ReferenceError (TDZ)
  let b = 20;
  ```

- TDZ:

  ```
  {
    console.log(x); // ReferenceError
    let x = 5;
  }
  ```

Practice Exercises

1. Identify the TDZ in code with `let` and `const`.

2. Write examples to demonstrate function and variable hoisting.

Topic 8: Closures and Lexical Scoping

Introduction

A **closure** is formed when an inner function retains access to variables in its outer scope, even after the outer function has returned. **Lexical Scoping** determines how variable names are resolved based on the location of their declaration.

Key Concepts and Definitions

- **Closure**:
  - A combination of a function and its lexical scope.
  - Allows data encapsulation.
- **Lexical Scoping**:
  - Variable access is determined by the position in the source code.

Examples

- Closure:

```
function outer() {
  let counter = 0;
  return function inner() {
    counter++;
    console.log(counter);
  };
}
const increment = outer();
increment(); // 1
increment(); // 2
```

- Lexical Scoping:

```
const globalVar = 'Global';
function outer() {
  const outerVar = 'Outer';
  function inner() {
    console.log(globalVar, outerVar); // Access both varia
bles
  }
  inner();
}
outer();
```

Practice Exercises

1. Create a closure to manage a private variable.
2. Write a nested function and observe lexical scope behavior.

Topic 9: IIFE and Function Currying

Introduction

IIFE(Immediately Invoked Function Expression) and currying are functional programming concepts. An **IIFE** is a function executed immediately after its declaration. **Currying** transforms a function with multiple arguments into a series of functions, each taking a single argument.

Key Concepts and Definitions

- **IIFE**:
  - Syntax: `(function() { ... })();`
  - Used to avoid polluting the global scope.
- **Currying**:
  - Breaking down a function into smaller functions that return another function.

- Improves reusability and composition.

Examples

- IIFE:

```
(function() {
  let message = 'IIFE executed';
  console.log(message);
})();
```

- Currying:

```
function multiply(a) {
  return function(b) {
    return a * b;
  };
}
const double = multiply(2);
console.log(double(5)); // 10
```

Practice Exercises

1. Write an IIFE to initialize variables without affecting the global scope.
2. Create a curried function to calculate discounts on prices.

Summary

Concept	Key Idea
GEC	Default context for global code.
FEC	Context for each function call.
Hoisting	Variables/functions moved to the top of their scope.
TDZ	`let` / `const` variables inaccessible before declaration.
Closures	Inner functions retaining access to outer variables.

Lexical Scoping	Variable resolution based on source code location.
IIFE	Functions executed immediately after definition.
Currying	Breaking a function into a sequence of unary functions.

Understanding Asynchronous JavaScript, Concurrency, and Single-Threaded Nature

1. JavaScript's Single-Threaded Nature

JavaScript operates on a **single-threaded execution model**, meaning it can execute one piece of code at a time on the main thread. This approach simplifies memory management but raises challenges for handling tasks like I/O operations, animations, and fetching data from servers, as these might block the main thread and lead to unresponsive applications.

To address this limitation, JavaScript uses **asynchronous programming** facilitated by the **event loop** and **Web APIs**. These mechanisms allow JavaScript to perform tasks in the background without halting the execution of the main thread.

2. The JavaScript Concurrency Model

Key Components

1. **Call Stack**:

    - The call stack executes functions in a **last in, first out (LIFO)** manner.
    - Synchronous code is pushed and popped directly on the call stack.
    - Asynchronous tasks are deferred to **Web APIs** or other threads.

    Example:

    ```
    function greet() {
      console.log('Hello!');
    }
    greet(); // "Hello!" is executed immediately.
    ```

2. **Web APIs**:

- Web APIs (provided by the browser or Node.js) handle asynchronous tasks like `setTimeout`, `fetch`, and DOM events.
- When a task completes, its callback is pushed into the **callback queue**.

3. **Callback Queue**:
   - Holds completed asynchronous tasks.
   - Tasks are moved to the call stack when it is empty.
4. **Event Loop**:
   - Continuously checks if the call stack is empty.
   - If the call stack is empty, the event loop moves the next task from the callback queue to the call stack for execution.
   - So Basically , The event loop is a mechanism in JavaScript that continuously monitors the call stack and the callback/microtask queues. It ensures that tasks from the callback queue (macrotasks) or microtask queue are pushed onto the call stack when it is empty, enabling non-blocking asynchronous execution.

---

3. How Web APIs Handle Asynchronous Operations

Web APIs play a crucial role in enabling JavaScript to perform asynchronous tasks. Let's see how they interact with the call stack and event loop using `setTimeout`.

Example: Understanding Web APIs and Event Loop

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

setTimeout(() => {
  console.log('Inside setTimeout');
}, 2000);

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

Execution Flow:

1. **Call Stack**:
    - Logs "Start" and "End".
2. **Web API**:
    - `setTimeout` sends the callback to the browser's Web API with a timer of 2 seconds.
3. **Callback Queue**:
    - After 2 seconds, the callback (`console.log('Inside setTimeout')`) moves to the callback queue.
4. **Event Loop**:
    - Pushes the callback to the call stack after the main thread finishes.

Output:

```
Start
End
Inside setTimeout
```

---

4. SetTimeout and SetInterval

What are `setTimeout` and `setInterval`?

- `setTimeout`: Executes a function once after a specified delay.
- `setInterval`: Executes a function repeatedly at specified intervals.

How They Work:

- Both are part of **Web APIs**.
- They schedule tasks asynchronously, so the main thread remains non-blocking.

Examples

1. **Using `setTimeout`:**

```
setTimeout(() => {
  console.log('Executed after 1 second');
}, 1000);
```

2. **Using `setInterval`:**

```
let count = 0;
const intervalId = setInterval(() => {
  console.log(`Interval count: ${++count}`);
  if (count === 5) {
    clearInterval(intervalId); // Stop after 5 intervals
  }
}, 1000);
```

---

5. Promises

Promises are the backbone of asynchronous programming in JavaScript. A **Promise** represents a value that may be available now, later, or never.
Another definition can be , A Promise is an object that represents the eventual completion or failure of an asynchronous operation. It has three states: **Pending**, **Fulfilled**, or **Rejected**, and provides methods like `.then`, `.catch`, and `.finally` to handle these states.

Promise has three states:

1. **Pending**: The initial state.
2. **Fulfilled**: When the operation is successful.
3. **Rejected**: When the operation fails.

Creating and Handling Promises

```
const promiseExample = new Promise((resolve, reject) => {
  let success = true; // Simulate success or failure
```

```
  setTimeout(() => {
    if (success) {
      resolve('Operation successful');
    } else {
      reject('Operation failed');
    }
  }, 2000);
});

promiseExample
  .then((message) => console.log(message)) // Logs "Operation successful" after 2 seconds
  .catch((error) => console.error(error));
```

Promise Chaining

```
new Promise((resolve) => {
  setTimeout(() => resolve(5), 1000);
})
  .then((value) => value * 2)
  .then((result) => console.log(result)); // Logs 10 after 1 second
```

---

6. Async/Await

The `async` and `await` keywords simplify handling Promises, allowing asynchronous code to appear synchronous.

Example: Async/Await

```
async function fetchData() {
  try {
    const response = await fetch('<https://jsonplaceholder.typicode.com/posts/1>');
    const data = await response.json();
```

```
    console.log(data);
  } catch (error) {
    console.error('Fetch failed:', error);
  }
}
fetchData();
```

How It Works:

- The `await` keyword pauses the execution until the Promise resolves or rejects.
- The `try-catch` block handles errors gracefully.

---

7. Example: Promises with SetTimeout

Here's a practical example that combines Promises with `setTimeout` to simulate an asynchronous task.

Code

```
function delay(seconds) {
  return new Promise((resolve, reject) => {
    if (seconds < 0) {
      reject('Time cannot be negative!');
    } else {
      setTimeout(() => {
        resolve(`Waited for ${seconds} second(s)`);
      }, seconds * 1000);
    }
  });
}

// Using the delay function
delay(2)
  .then((message) => {
    console.log(message); // Logs after 2 seconds
    return delay(1);
```

```
  })
  .then((message) => {
    console.log(message); // Logs after another 1 second
  })
  .catch((error) => {
    console.error(error); // Handles any error
  });
```

Example: Fetch with `async/await` and `try-catch`

```
async function fetchUsers() {
  const apiUrl = 'https://jsonplaceholder.typicode.com/user
s';

  try {
    console.log('Fetching data...');
    const response = await fetch(apiUrl);

    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}
`);
    }

    const users = await response.json();
    console.log('Fetched Users:', users);
  } catch (error) {
    console.error('Error occurred:', error.message);
  } finally {
    console.log('Fetch attempt complete.');
  }
}
fetchUsers();
```

---

8. Summary

Concept	Description
Single-Threaded Nature	JavaScript runs on a single thread, executing one task at a time.
Event Loop	Manages the interaction between the Call Stack, Callback Queue, and Microtasks.
Web APIs	Handles asynchronous operations like timers, HTTP requests, and DOM events.
setTimeout/setInterval	Web API functions for scheduling delayed or repeated execution.
Promises	Handle asynchronous tasks with `.then`, `.catch`, and `.finally`.
async/await	Simplifies asynchronous code, allowing `try-catch` for error handling.

1. Understanding the DOM (Document Object Model)

What is the DOM?

Definition:

The **Document Object Model (DOM)** is a programming interface for web documents. It represents the structure of a webpage as a tree of objects, allowing developers to manipulate content, structure, and styles dynamically.

Key Concepts:

- **Document**: The root object of the DOM tree, representing the entire HTML document.
- **Nodes**: Represent elements, attributes, or text in the DOM tree.
- **DOM Tree**: A hierarchical representation of the document structure.

Example of a Simple DOM Tree:

For the following HTML:

```
<html>
  <body>
```

```
    <h1 id="title">Hello World</h1>
    <p class="content">This is a paragraph.</p>
  </body>
</html>
```

The DOM tree looks like this:

```
Document
 └── <html>
      └── <body>
           ├── <h1 id="title">Hello World</h1>
           └── <p class="content">This is a paragraph.</p>
```

Accessing the DOM in JavaScript:

You can use JavaScript to select and manipulate DOM elements:

```
// Accessing elements
const title = document.getElementById('title');
const paragraphs = document.querySelectorAll('.content');

// Manipulating content
title.textContent = 'Welcome to the DOM!';
```

---

2. CRUD Operations on the DOM

CRUD (Create, Read, Update, Delete) operations are essential for interacting with the DOM.

Create

To add elements dynamically:

```
const newElement = document.createElement('div');
newElement.textContent = 'This is a new div';
document.body.appendChild(newElement);
```

Read

To access existing elements:

```
const header = document.getElementById('title');
console.log(header.textContent); // Logs: "Welcome to the DO
M!"
```

Update

To modify content or attributes:

```
header.textContent = 'Updated Title';
header.style.color = 'blue';
```

Delete

To remove elements:

```
header.remove(); // Removes the <h1> element
```

Example: Full CRUD Operation

```
// Create
const newPara = document.createElement('p');
newPara.textContent = 'This is a new paragraph.';
document.body.appendChild(newPara);

// Read
const para = document.querySelector('p');
console.log(para.textContent);

// Update
para.textContent = 'Updated paragraph content.';
para.style.fontWeight = 'bold';
```

```
// Delete
para.remove();
```

---

3. HTML Elements vs NodeList

HTML Elements

- Represent DOM elements as objects with specific properties like `id`, `className`, and `innerHTML`.
- Accessed using methods like `getElementById` or `getElementsByClassName`.

Example:

```
const header = document.getElementById('title'); // HTMLEleme
nt
console.log(header.id); // "title"
```

NodeList

- A collection of DOM nodes, which can be elements, text, or comments.
- Returned by methods like `querySelectorAll`.
- Similar to arrays but lacks many array methods.

Example:

```
const nodeList = document.querySelectorAll('p'); // NodeList
console.log(nodeList.length); // Number of <p> elements
```

Converting NodeList to Array

You can convert a NodeList to an array for more operations:

```
const paragraphs = Array.from(document.querySelectorAll
('p'));
paragraphs.forEach((p) => console.log(p.textContent));
```

---

4. High-Order Functions (HOFs): `map`, `filter`, `reduce`

Definition:

High-Order Functions are functions that take other functions as arguments or return them. They are powerful tools for working with arrays in JavaScript.

1. `map`

Creates a new array by applying a function to each element of the original array.

Syntax:

```
array.map(callback);
```

Example:

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

---

2. `filter`

Creates a new array containing elements that satisfy a given condition.

Syntax:

```
array.filter(callback);
```

Example:

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

---

3. `reduce`

Reduces an array to a single value by repeatedly applying a function to elements.

Syntax:

```
array.reduce(callback, initialValue);
```

Example:

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

---

5. Practical Example: DOM + HOFs

Here's a practical example combining the DOM and HOFs:

```
// HTML:
// <ul id="items">
//   <li>Apple</li>
//   <li>Banana</li>
//   <li>Cherry</li>
// </ul>

// JavaScript:
// Get NodeList of all list items
const items = document.querySelectorAll('#items li');

// Convert NodeList to Array
const itemTexts = Array.from(items).map((item) => item.textContent);

console.log(itemTexts); // ["Apple", "Banana", "Cherry"]

// Filter items containing the letter "a"
const filteredItems = itemTexts.filter((text) => text.toLowerCase().includes('a'));
```

```
console.log(filteredItems); // ["Banana", "Cherry"]

// Create a single string of items
const concatenatedItems = itemTexts.reduce((acc, curr) => acc
+ ', ' + curr);
console.log(concatenatedItems); // "Apple, Banana, Cherry"
```

6. Summary

Concept	Definition
DOM	A programming interface representing the structure of an HTML document.
CRUD with DOM	Operations to Create, Read, Update, and Delete elements dynamically.
HTML Elements	Represent DOM elements with properties like `id`, `className`, `innerHTML`.
NodeList	A collection of nodes that can include elements, text, or comments.
HOF (`map`, `filter`, `reduce`)	Functions that operate on arrays, transforming or reducing them into new outputs.

Applying Higher-Order Functions (HOFs) for Searching, Filtering, Pagination, and Sorting

1. Key Concepts

Higher-Order Functions (HOFs):

HOFs operate on arrays, taking a function as an argument or returning a function. They are essential for concise, expressive, and reusable code.

Function	Purpose
`map`	Transforms each element of an array into a new array.
`forEach`	Iterates over an array, typically used for side effects.

`filter`	Creates a new array with elements that satisfy a condition.
`reduce`	Reduces an array to a single value by repeatedly applying a function.
`sort`	Reorders elements of an array based on a comparator function.

2. Applying HOFs for Common Use Cases

2.1 Searching Logic

Searching involves finding elements in an array that match a certain condition or keyword.

Example: Searching for products by name or description.

```
const products = [
  { id: 1, name: 'Laptop', description: 'Portable computer' },
  { id: 2, name: 'Phone', description: 'Handheld device' },
  { id: 3, name: 'Tablet', description: 'Touchscreen device' }
];

// Search by keyword
const searchKeyword = 'laptop';

const searchResults = products.filter(product =>
  product.name.toLowerCase().includes(searchKeyword.toLowerCase()) ||
  product.description.toLowerCase().includes(searchKeyword.toLowerCase())
);

console.log(searchResults);
// Output: [{ id: 1, name: 'Laptop', description: 'Portable computer' }]
```

2.2 Filtering Logic

Filtering involves creating a subset of an array based on specific criteria.

Example: Filtering products by a price range.

```
const products = [
  { id: 1, name: 'Laptop', price: 1000 },
  { id: 2, name: 'Phone', price: 500 },
  { id: 3, name: 'Tablet', price: 700 }
];

// Filter products by price range
const minPrice = 600;
const maxPrice = 1200;

const filteredProducts = products.filter(product =>
  product.price >= minPrice && product.price <= maxPrice
);

console.log(filteredProducts);
// Output: [{ id: 1, name: 'Laptop', price: 1000 }, { id: 3,
name: 'Tablet', price: 700 }]
```

2.3 Sorting Logic

Sorting reorders elements of an array based on a specific property.

Example: Sorting products by price (ascending and descending).

```
const products = [
  { id: 1, name: 'Laptop', price: 1000 },
  { id: 2, name: 'Phone', price: 500 },
  { id: 3, name: 'Tablet', price: 700 }
];

// Ascending order
```

```
const sortedAsc = products.slice().sort((a, b) => a.price -
b.price);
console.log(sortedAsc);
// Output: [{ id: 2, name: 'Phone', price: 500 }, { id: 3, na
me: 'Tablet', price: 700 }, { id: 1, name: 'Laptop', price: 1
000 }]

// Descending order
const sortedDesc = products.slice().sort((a, b) => b.price -
a.price);
console.log(sortedDesc);
// Output: [{ id: 1, name: 'Laptop', price: 1000 }, { id: 3,
name: 'Tablet', price: 700 }, { id: 2, name: 'Phone', price:
500 }]
```

2.4 Pagination Logic

Pagination breaks a dataset into smaller chunks for easy navigation.

Example: Implementing pagination for a list of items.

```
const items = Array.from({ length: 50 }, (_, i) => `Item ${i
+ 1}`); // Example dataset

// Pagination logic
function paginate(array, pageSize, currentPage) {
  const start = (currentPage - 1) * pageSize;
  return array.slice(start, start + pageSize);
}

// Paginate items (10 items per page, page 2)
const pageSize = 10;
const currentPage = 2;

const paginatedItems = paginate(items, pageSize, currentPag
e);
```

```
console.log(paginatedItems);
// Output: ["Item 11", "Item 12", ..., "Item 20"]
```

2.5 Combining Searching, Filtering, and Sorting

Combine multiple operations for complex use cases.

Example: Search, filter by price, and sort by name.

```
const products = [
  { id: 1, name: 'Laptop', price: 1000 },
  { id: 2, name: 'Phone', price: 500 },
  { id: 3, name: 'Tablet', price: 700 },
  { id: 4, name: 'Smartwatch', price: 200 }
];

const searchKeyword = 'p';
const minPrice = 300;
const maxPrice = 1200;

// Combined operations
const result = products
  .filter(product =>
    product.name.toLowerCase().includes(searchKeyword.toLower
Case())
  )
  .filter(product => product.price >= minPrice && product.pri
ce <= maxPrice)
  .sort((a, b) => a.name.localeCompare(b.name));

console.log(result);
// Output: [
//   { id: 1, name: 'Laptop', price: 1000 },
//   { id: 3, name: 'Tablet', price: 700 }
// ]
```

3. Advantages of Using HOFs

- **Concise Code**: HOFs reduce boilerplate and improve readability.
- **Reusability**: HOF-based logic can be easily reused and extended.
- **Functional Approach**: Encourages immutability and avoids side effects.

---

4. Summary Table

Operation	HOF Used	Purpose
Searching	`filter`	Finds elements matching a condition.
Filtering	`filter`	Creates a subset of elements based on criteria.
Sorting	`sort`	Reorders elements based on a comparator function.
Pagination	`slice`	Extracts a portion of the array for a specific page.
Transforming	`map`	Transforms each element in an array.
Iteration	`forEach`	Iterates over elements, often for side effects.
Aggregation	`reduce`	Combines all elements into a single value or structure.

---

6. Additional Resources

- MDN: Array.prototype.map()
- MDN: Array.prototype.filter()
- MDN: Array.prototype.reduce()
- MDN: Array.prototype.sort()

Debouncing and Throttling

---

1. What are Debouncing and Throttling?

Debouncing

Definition:

Debouncing ensures that a function is executed only after a specified time has passed since it was last invoked. If the event occurs again within the wait time, the timer resets.

Use Case:

Debouncing is useful for scenarios where events occur rapidly, and we want to delay execution until the user stops triggering the event. Examples include:

- Searching in a text input field (to reduce API calls).
- Resizing a window.

Throttling

Definition:

Throttling ensures that a function is executed at most once in a specified time interval, no matter how many times the event occurs.

Use Case:

Throttling is useful for limiting the frequency of function calls. Examples include:

- Handling scroll events (e.g., infinite scrolling).
- Resizing the browser window.

---

2. Differences Between Debouncing and Throttling

Aspect	**Debouncing**	**Throttling**
Execution Timing	Executes after a pause in events.	Executes at regular intervals during events.
Behavior	Delays execution until the event stops firing.	Limits the rate at which the function executes.
Use Case	Search inputs, window resize.	Scrolling, API rate limiting.

---

3. Implementation in JavaScript

3.1 Debouncing

Implementation:

```
function debounce(func, delay) {
  let timer;
  return function (...args) {
    clearTimeout(timer); // Clear the existing timer
    timer = setTimeout(() => func.apply(this, args), delay);
// Set a new timer
  };
}
```

Example: Search Input

```
const searchInput = document.getElementById('search');

const fetchData = (query) => {
  console.log(`Fetching data for: ${query}`);
};

const debouncedFetch = debounce(fetchData, 500); // Wait 500m
s after user stops typing

searchInput.addEventListener('input', (event) => {
  debouncedFetch(event.target.value);
});
```

3.2 Throttling

Implementation:

```
function throttle(func, limit) {
  let lastCall = 0;
  return function (...args) {
    const now = Date.now();
    if (now - lastCall >= limit) {
      lastCall = now;
      func.apply(this, args);
```

```
    }
  };
}
```

Example: Scroll Event

```
const handleScroll = () => {
  console.log('Scroll event triggered');
};

const throttledScroll = throttle(handleScroll, 1000); // Exec
ute at most once every 1000ms

window.addEventListener('scroll', throttledScroll);
```

---

4. Real-World Examples

Debouncing

- **Search Input**: Delay API calls until the user stops typing.
- **Window Resize**: Recalculate dimensions after resizing has stopped.

```
const logResize = debounce(() => {
  console.log('Window resized');
}, 300);

window.addEventListener('resize', logResize);
```

Throttling

- **Scroll-Based Animations**: Trigger animations while scrolling at fixed intervals.
- **Infinite Scrolling**: Fetch new content at regular intervals.

```
const fetchMoreContent = throttle(() => {
  console.log('Fetching more content...');
}, 2000);

window.addEventListener('scroll', fetchMoreContent);
```

5. Summary

Concept	Definition
Debouncing	Ensures a function is executed only after a specified delay following the last event.
Throttling	Ensures a function is executed at most once every specified interval.
Debouncing Use Case	Search input, window resize.
Throttling Use Case	Scroll events, API rate limiting.

1. Understanding the DOM (Document Object Model)

What is the DOM?

Definition:

The **Document Object Model (DOM)** is a programming interface for web documents. It represents the structure of a webpage as a tree of objects, allowing developers to manipulate content, structure, and styles dynamically.

Key Concepts:

- **Document**: The root object of the DOM tree, representing the entire HTML document.
- **Nodes**: Represent elements, attributes, or text in the DOM tree.
- **DOM Tree**: A hierarchical representation of the document structure.

Example of a Simple DOM Tree:

For the following HTML:

```html
<html>
  <body>
    <h1 id="title">Hello World</h1>
    <p class="content">This is a paragraph.</p>
  </body>
</html>
```

The DOM tree looks like this:

```
Document
 └── <html>
      └── <body>
           ├── <h1 id="title">Hello World</h1>
           └── <p class="content">This is a paragraph.</p>
```

Accessing the DOM in JavaScript:

You can use JavaScript to select and manipulate DOM elements:

```javascript
// Accessing elements
const title = document.getElementById('title');
const paragraphs = document.querySelectorAll('.content');

// Manipulating content
title.textContent = 'Welcome to the DOM!';
```

---

2. CRUD Operations on the DOM

CRUD (Create, Read, Update, Delete) operations are essential for interacting with the DOM.

Create

To add elements dynamically:

```
const newElement = document.createElement('div');
newElement.textContent = 'This is a new div';
document.body.appendChild(newElement);
```

Read

To access existing elements:

```
const header = document.getElementById('title');
console.log(header.textContent); // Logs: "Welcome to the DO
M!"
```

Update

To modify content or attributes:

```
header.textContent = 'Updated Title';
header.style.color = 'blue';
```

Delete

To remove elements:

```
header.remove(); // Removes the <h1> element
```

Example: Full CRUD Operation

```
// Create
const newPara = document.createElement('p');
newPara.textContent = 'This is a new paragraph.';
document.body.appendChild(newPara);

// Read
const para = document.querySelector('p');
console.log(para.textContent);
```

```
// Update
para.textContent = 'Updated paragraph content.';
para.style.fontWeight = 'bold';

// Delete
para.remove();
```

---

3. HTML Elements vs NodeList

HTML Elements

- Represent DOM elements as objects with specific properties like `id`, `className`, and `innerHTML`.
- Accessed using methods like `getElementById` or `getElementsByClassName`.

Example:

```
const header = document.getElementById('title'); // HTMLEleme
nt
console.log(header.id); // "title"
```

NodeList

- A collection of DOM nodes, which can be elements, text, or comments.
- Returned by methods like `querySelectorAll`.
- Similar to arrays but lacks many array methods.

Example:

```
const nodeList = document.querySelectorAll('p'); // NodeList
console.log(nodeList.length); // Number of <p> elements
```

Converting NodeList to Array

You can convert a NodeList to an array for more operations:

```
const paragraphs = Array.from(document.querySelectorAll
('p'));
paragraphs.forEach((p) => console.log(p.textContent));
```

---

4. High-Order Functions (HOFs): `map`, `filter`, `reduce`

Definition:

High-Order Functions are functions that take other functions as arguments or return them. They are powerful tools for working with arrays in JavaScript.

1. `map`

Creates a new array by applying a function to each element of the original array.

Syntax:

```
array.map(callback);
```

Example:

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

---

2. `filter`

Creates a new array containing elements that satisfy a given condition.

Syntax:

```
array.filter(callback);
```

Example:

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

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

---

3. `reduce`

Reduces an array to a single value by repeatedly applying a function to elements.

Syntax:

```
array.reduce(callback, initialValue);
```

Example:

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

---

5. Practical Example: DOM + HOFs

Here’s a practical example combining the DOM and HOFs:

```
// HTML:
// <ul id="items">
//   <li>Apple</li>
//   <li>Banana</li>
//   <li>Cherry</li>
// </ul>

// JavaScript:
// Get NodeList of all list items
const items = document.querySelectorAll('#items li');

// Convert NodeList to Array
const itemTexts = Array.from(items).map((item) => item.textCo
ntent);
```

```
console.log(itemTexts); // ["Apple", "Banana", "Cherry"]

// Filter items containing the letter "a"
const filteredItems = itemTexts.filter((text) => text.toLowerCase().includes('a'));
console.log(filteredItems); // ["Banana", "Cherry"]

// Create a single string of items
const concatenatedItems = itemTexts.reduce((acc, curr) => acc + ', ' + curr);
console.log(concatenatedItems); // "Apple, Banana, Cherry"
```

---

6. Summary

Concept	Definition
DOM	A programming interface representing the structure of an HTML document.
CRUD with DOM	Operations to Create, Read, Update, and Delete elements dynamically.
HTML Elements	Represent DOM elements with properties like `id`, `className`, `innerHTML`.
NodeList	A collection of nodes that can include elements, text, or comments.
HOF (`map`, `filter`, `reduce`)	Functions that operate on arrays, transforming or reducing them into new outputs.

Applying Higher-Order Functions (HOFs) for Searching, Filtering, Pagination, and Sorting

1. Key Concepts

Higher-Order Functions (HOFs):

HOFs operate on arrays, taking a function as an argument or returning a function. They are essential for concise, expressive, and reusable code.

Function	Purpose
`map`	Transforms each element of an array into a new array.
`forEach`	Iterates over an array, typically used for side effects.
`filter`	Creates a new array with elements that satisfy a condition.
`reduce`	Reduces an array to a single value by repeatedly applying a function.
`sort`	Reorders elements of an array based on a comparator function.

---

2. Applying HOFs for Common Use Cases

2.1 Searching Logic

Searching involves finding elements in an array that match a certain condition or keyword.

Example: Searching for products by name or description.

```
const products = [
  { id: 1, name: 'Laptop', description: 'Portable computer'
},
  { id: 2, name: 'Phone', description: 'Handheld device' },
  { id: 3, name: 'Tablet', description: 'Touchscreen device'
}
];

// Search by keyword
const searchKeyword = 'laptop';

const searchResults = products.filter(product =>
  product.name.toLowerCase().includes(searchKeyword.toLowerCa
se()) ||
  product.description.toLowerCase().includes(searchKeyword.to
LowerCase())
);
```

```
console.log(searchResults);
// Output: [{ id: 1, name: 'Laptop', description: 'Portable c
omputer' }]
```

2.2 Filtering Logic

Filtering involves creating a subset of an array based on specific criteria.

Example: Filtering products by a price range.

```
const products = [
  { id: 1, name: 'Laptop', price: 1000 },
  { id: 2, name: 'Phone', price: 500 },
  { id: 3, name: 'Tablet', price: 700 }
];

// Filter products by price range
const minPrice = 600;
const maxPrice = 1200;

const filteredProducts = products.filter(product =>
  product.price >= minPrice && product.price <= maxPrice
);

console.log(filteredProducts);
// Output: [{ id: 1, name: 'Laptop', price: 1000 }, { id: 3,
name: 'Tablet', price: 700 }]
```

2.3 Sorting Logic

Sorting reorders elements of an array based on a specific property.

Example: Sorting products by price (ascending and descending).

```
const products = [
  { id: 1, name: 'Laptop', price: 1000 },
  { id: 2, name: 'Phone', price: 500 },
  { id: 3, name: 'Tablet', price: 700 }
];

// Ascending order
const sortedAsc = products.slice().sort((a, b) => a.price -
b.price);
console.log(sortedAsc);
// Output: [{ id: 2, name: 'Phone', price: 500 }, { id: 3, na
me: 'Tablet', price: 700 }, { id: 1, name: 'Laptop', price: 1
000 }]

// Descending order
const sortedDesc = products.slice().sort((a, b) => b.price -
a.price);
console.log(sortedDesc);
// Output: [{ id: 1, name: 'Laptop', price: 1000 }, { id: 3,
name: 'Tablet', price: 700 }, { id: 2, name: 'Phone', price:
500 }]
```

2.4 Pagination Logic

Pagination breaks a dataset into smaller chunks for easy navigation.

Example: Implementing pagination for a list of items.

```
const items = Array.from({ length: 50 }, (_, i) => `Item ${i
+ 1}`); // Example dataset

// Pagination logic
function paginate(array, pageSize, currentPage) {
  const start = (currentPage - 1) * pageSize;
  return array.slice(start, start + pageSize);
```

```
}

// Paginate items (10 items per page, page 2)
const pageSize = 10;
const currentPage = 2;

const paginatedItems = paginate(items, pageSize, currentPag
e);
console.log(paginatedItems);
// Output: ["Item 11", "Item 12", ..., "Item 20"]
```

2.5 Combining Searching, Filtering, and Sorting

Combine multiple operations for complex use cases.

Example: Search, filter by price, and sort by name.

```
const products = [
  { id: 1, name: 'Laptop', price: 1000 },
  { id: 2, name: 'Phone', price: 500 },
  { id: 3, name: 'Tablet', price: 700 },
  { id: 4, name: 'Smartwatch', price: 200 }
];

const searchKeyword = 'p';
const minPrice = 300;
const maxPrice = 1200;

// Combined operations
const result = products
  .filter(product =>
    product.name.toLowerCase().includes(searchKeyword.toLower
Case())
  )
  .filter(product => product.price >= minPrice && product.pri
ce <= maxPrice)
```

```
  .sort((a, b) => a.name.localeCompare(b.name));

console.log(result);
// Output: [
//   { id: 1, name: 'Laptop', price: 1000 },
//   { id: 3, name: 'Tablet', price: 700 }
// ]
```

---

3. Advantages of Using HOFs

- **Concise Code**: HOFs reduce boilerplate and improve readability.
- **Reusability**: HOF-based logic can be easily reused and extended.
- **Functional Approach**: Encourages immutability and avoids side effects.

---

4. Summary Table

Operation	HOF Used	Purpose
Searching	`filter`	Finds elements matching a condition.
Filtering	`filter`	Creates a subset of elements based on criteria.
Sorting	`sort`	Reorders elements based on a comparator function.
Pagination	`slice`	Extracts a portion of the array for a specific page.
Transforming	`map`	Transforms each element in an array.
Iteration	`forEach`	Iterates over elements, often for side effects.
Aggregation	`reduce`	Combines all elements into a single value or structure.

---

6. Additional Resources

- MDN: Array.prototype.map()
- MDN: Array.prototype.filter()
- MDN: Array.prototype.reduce()
- MDN: Array.prototype.sort()

Debouncing and Throttling

1. What are Debouncing and Throttling?

Debouncing

Definition:

Debouncing ensures that a function is executed only after a specified time has passed since it was last invoked. If the event occurs again within the wait time, the timer resets.

Use Case:

Debouncing is useful for scenarios where events occur rapidly, and we want to delay execution until the user stops triggering the event. Examples include:

- Searching in a text input field (to reduce API calls).
- Resizing a window.

Throttling

Definition:

Throttling ensures that a function is executed at most once in a specified time interval, no matter how many times the event occurs.

Use Case:

Throttling is useful for limiting the frequency of function calls. Examples include:

- Handling scroll events (e.g., infinite scrolling).
- Resizing the browser window.

---

2. Differences Between Debouncing and Throttling

Aspect	Debouncing	Throttling
Execution Timing	Executes after a pause in events.	Executes at regular intervals during events.

Behavior	Delays execution until the event stops firing.	Limits the rate at which the function executes.
Use Case	Search inputs, window resize.	Scrolling, API rate limiting.

3. Implementation in JavaScript

3.1 Debouncing

Implementation:

```
function debounce(func, delay) {
  let timer;
  return function (...args) {
    clearTimeout(timer); // Clear the existing timer
    timer = setTimeout(() => func.apply(this, args), delay);
// Set a new timer
  };
}
```

Example: Search Input

```
const searchInput = document.getElementById('search');

const fetchData = (query) => {
  console.log(`Fetching data for: ${query}`);
};

const debouncedFetch = debounce(fetchData, 500); // Wait 500m
s after user stops typing

searchInput.addEventListener('input', (event) => {
  debouncedFetch(event.target.value);
});
```

3.2 Throttling

Implementation:

```
function throttle(func, limit) {
  let lastCall = 0;
  return function (...args) {
    const now = Date.now();
    if (now - lastCall >= limit) {
      lastCall = now;
      func.apply(this, args);
    }
  };
}
```

Example: Scroll Event

```
const handleScroll = () => {
  console.log('Scroll event triggered');
};

const throttledScroll = throttle(handleScroll, 1000); // Execute at most once every 1000ms

window.addEventListener('scroll', throttledScroll);
```

---

4. Real-World Examples

Debouncing

- **Search Input**: Delay API calls until the user stops typing.
- **Window Resize**: Recalculate dimensions after resizing has stopped.

```
const logResize = debounce(() => {
  console.log('Window resized');
}, 300);
```

```
window.addEventListener('resize', logResize);
```

Throttling

- **Scroll-Based Animations**: Trigger animations while scrolling at fixed intervals.
- **Infinite Scrolling**: Fetch new content at regular intervals.

```
const fetchMoreContent = throttle(() => {
  console.log('Fetching more content...');
}, 2000);

window.addEventListener('scroll', fetchMoreContent);
```

5. Summary

Concept	Definition
Debouncing	Ensures a function is executed only after a specified delay following the last event.
Throttling	Ensures a function is executed at most once every specified interval.
Debouncing Use Case	Search input, window resize.
Throttling Use Case	Scroll events, API rate limiting.