Curriculum

HTML

1. Basic Concepts of Web Dev

- Server and Client - General Terms
- Request and Response - Communication between a Client and a Server
- Internet and IP address
- Web Server, Browser and their role
- Communication between a Browser and a Web Server
    - Similar to any Client-Server communication
    - HTML rendering by the browser
    - HTTP Protocol
    - View Page source for response

2. HTML Introduction

- What is HTML and its significance?
- Basic structure of an HTML document
- Indentation and its importance
- HTML Tags, Elements and Attributes
- Self Closing Tags
- The different tags of the basic structure of an HTML document
- Using HTML
    - Editor: Ideally use a very basic text editor or Install Visual Studio Code
    - Write code on text editor and check the output on Browser.

3. Basic HTML Tags

- Elemetary Tags like: <h1> <h2> <h3> <h4> <h5> <h6>, <p>,
, <hr/>, <img>, <a>, ,
- Formatting Tags: <b>, <i> etc
- Block v/s Inline Tags: Basic Idea
- Lists: <ol>, <ul>, <dl>, Nested list
- Tables: <table>, <tr>, <th>, <td>, Border, Cellspacing, Cellpadding, Height, Width

4. Forms

- Significance of an HTML Form
- Different Input Elements: Text Box, Password, Radio Button, Checkbox, Drop Down, Text Area
- Submit Button
- Action attribute. Method attribute (only type of request)

CSS

1. CSS Introduction

- What is CSS?
- Three ways to include styling: Inline Style, Internal Styling, External CSS
- CSS syntax: Selectors, Properties, Values
- Precedence of including styling
- Folder Structure
- How CSS is loaded and rendered - Separate request while the HTML gets rendered
- Absolute or Relative URLs when including CSS file

2. CSS Selectors

- Element / Type Selectors
- Id Selectors
- Class Selectors
- Combination Selectors
- Hierarchy: Child and Descendant
- Child Selectors
- Descendant Selectors
- Grouping Selectors
- Attribute Selectors

3. Length Units & Colour

- Pixels in General
- Pixels in Web Development
- Percentage length unit
- Colors: Names, RGB, Hex

4. Basic CSS Properties

- Default Values
- Color
- Background Color
- Font-family
- Font-size
- Font-weight
- Font-style
- Text-decoration
- Text-align
- Line-height

5. Box Model

- Borders: border-style, border-width, border-color, border (shorthand property), border sub-properties (border-top, border-right, border-bottom, border-left)
- Padding: Multiple values for individual sides, padding sub-properties
- Margin: Multiple values for individual sides, margin sub-properties
- Height and width: Content Box, Box Sizing
- Default values for height and width properties

6. Inspect Element

- Dock Side, Element Tab, Left Section, Right Section, Box Model, Computed Tab, Styles Tab (edits), Toggle Device, How to Debug

7. Display

- Block, Inline, Inline-block
- Tabular comparison - previous element, next element, default width, explicit height / width
- Display: none

8. Positioning

- Static | Positioned element
- Relative | top, bottom, left, right properties
- Fixed
- Absolute
- Tabular comparison of "with respect to" and "flow gap"
- Z-index - positioned

9. Page Layouts - Structuring a webpage

- How to create a layout - box inside another box, either horizontal or vertical.
- Vertical and Horizontal alignment of elements.
- Idea about height and width. Width is generally fixed, Height is generally content-based.
- Page Layout using Display: inline-block
- Page Layout using Float
    - Idea of clearfix
- Page Layout using Flexbox

10. Flexbox

- The flex container and flex items
- The two axes of flexbox
- Flex-direction
- Flex-basis, Available and deficit space, Flex-grow, Flex-shrink, Flex shorthand
- Flex-wrap, Flex-flow shorthand
- Justify-content, Align-items, Align-self

11. Media Queries

- Explain the problem of viewing in different devices.
- Explain the possible solution - design for different views
- Viewport
- Media Query
- Inspect Element

JavaScript

1. Introduction

- What is JavaScript?
    - Introduce JavaScript as a simple programming language and not as a client-side language.
- What is Node? What is NPM? Installation
    - Run the code on Command Line or Terminal.

2. Variables, Data Types and Operators

- Variables - concept
    - let, var, const
    - "use strict"; - restrict from using older syntax
- Datatypes
    - Integer, float, string, etc
    - Variables can store data of any datatype without having to mention it.
- Operators and Expressions
    - Arithmetic Operators: + - * / % **
    - Assignment Operators: = += -= *= /= %= **=
    - Increment / Decrement Operators: ++ --
    - Comparison Operators: == === != !== < <= > >=
    - Logical Operators: && || !
    - Typeof

3. Conditionals and Loops

- Conditional Statements
    - If-statements
    - if-else statements
    - if-<else if>-else statements
    - Switch statement
- Loops
    - While
    - Do While
    - For
    - Break
    - Continue

        - For-of (revisit in Arrays)
        - For-in (revisit in Objects)
        - Foreach (revisit in Arrays)

4. Functions

- What is a function? Why it is used?
    - Syntax: the two parts, Function Definition and Function Calling
- Default Value for Arguments
- Call By Value / Reference
- Function Expressions
- Arrow functions
- Callback Functions and Higher Order Functions

5. Objects and Classes

- What is an Object?
    - Properties and Methods
- Creating objects
    - Object Initializer: Dot operator and Square Brackets
    - Using new Object()
    - Function constructor
- Class
- For-in loop

6. Arrays

- What is an Array?
- For-of Loop
- Arrays are Objects
- Array methods: foreach(), map(), filter()
- Rest Parameters
- Spread Operator

7. Strings

- What is a String?
- Concatenation
- Template Literals
- Strings as Objects
- String Methods: indexOf(), slice(), replace(), startsWith(), endsWith(), include() etc
    - Regular expressions

8. JavaScript as a Client-side Language

- Idea of a client-side language and its extent
    - JavaScript can run on a browser. This allows JavaScript to:
        - Capture any user action
        - Editing HTML elements
    - Hence, JavaScript is used to create user-interactive web pages.
- Using JavaScript along with HTML and CSS

    - Folder structure for JS files in a project.
    - How to include JS files and how they are fetched as separate requests.
    - <script> tags can also be used - should be at the end, we will learn later why.

9. Inspect Element

- Debugging using Source tab
- Anazlyzing Requests and Responses using Network Tab

10. DOM

- DOM Tree
- Document Object
- Document Object Methods: getElementById(), getElementsByClassName(), getElementsByTagName(), querySelectorAll()
- Return type: Node Object
- Node Object Properties: parentNode, childNodes, firstElementChild, etc
- Node Object Properties and Methods: innerHTML, style, etc
- Event Handlers: Click, Double Click, Mouse Enter, Key Press, Mouse Over

11. Asynchronous JavaScript

- Introduction to Asynchronous JavaScript
    - Synchronous vs Asynchronous Programming
    - The Event Loop
    - Callbacks and the "Callback Hell" Problem
- Timers
    - setTimeout
    - setInterval
    - Clearing Timers (clearTimeout, clearInterval)
- Promises
    - Introduction to Promises
    - Creating Promises
    - Chaining Promises with .then()
    - Error Handling with .catch()
    - The finally Method
    - Common Promise Methods: Promise.all, Promise.race, Promise.allSettled, Promise.any
- Async/Await
    - Introduction to async and await
    - Writing Asynchronous Functions
    - Error Handling in Async/Await
    - Combining Async/Await with Promises

12. API's

- Making HTTP Requests with Fetch
- Handling Responses (JSON, text, etc.)

- Error Handling in Fetch
- Using Fetch with Async/Await

Node + Mongo

1. Introduction to Node.js

- What is Node.js?
  - Overview of Node.js and its purpose
  - Understanding the event-driven, non-blocking I/O model
- Setting Up Node.js
  - Installing Node.js and npm (Node Package Manager)
  - Creating a simple Node.js script
- Understanding the Node.js Runtime
  - How Node.js executes JavaScript on the server
  - Differences between Node.js and traditional web servers

2. Working with Node.js Modules

- Built-in Modules
  - Introduction to core modules like fs, http, path, os
- Creating Custom Modules
  - Exporting and importing custom modules
  - Organizing code into separate files
- Using npm and Third-Party Modules
  - Introduction to npm and managing dependencies
  - Installing and using third-party modules

3. Creating a Simple Web Server

- Building a Basic HTTP Server
  - Using the http module to create a simple web server
  - Handling requests and responses
- Routing Requests
  - Implementing basic routing to handle different URLs
- Serving Static Files
  - Serving HTML, CSS, and JavaScript files using Node.js

4. Introduction to Express.js

- What is Express.js?
  - Overview of Express as a web application framework for Node.js
- Setting Up Express
  - Installing and setting up a basic Express application
- Creating Routes
  - Defining and handling GET, POST, PUT, DELETE routes

- Middleware in Express
    - Understanding middleware and its role in Express
    - Using built-in and third-party middleware

5. Working with Templates

- Introduction to Templating Engines
    - Understanding the purpose of templating engines
- Setting Up EJS (Embedded JavaScript)
    - Installing and configuring EJS in an Express app
    - Creating and rendering EJS templates
- Passing Data to Templates
    - Rendering dynamic content using data passed from the server

6. Introduction to MongoDB

- What is MongoDB?
    - Overview of MongoDB as a NoSQL database
    - Understanding document-oriented data storage
- Setting Up MongoDB
    - Installing MongoDB locally or using a cloud service like MongoDB Atlas
    - Introduction to MongoDB Shell for basic operations

7. CRUD Operations with MongoDB

- Connecting to MongoDB
    - Using the mongodb package to connect Node.js with MongoDB
- Creating Documents
    - Inserting documents into a MongoDB collection
- Reading Documents
    - Querying collections and retrieving documents
- Updating Documents
    - Updating existing documents in a collection
- Deleting Documents
    - Removing documents from a collection

8. Integrating MongoDB with Express.js

- Connecting MongoDB with an Express App
    - Setting up the connection using Mongoose or the native MongoDB driver
- Creating a Simple RESTful API
    - Building CRUD routes in Express to interact with MongoDB
- Handling Errors and Validations
    - Implementing basic error handling and input validation

9. Mongoose ODM (Object Data Modeling)

- What is Mongoose?
  - Introduction to Mongoose as an ODM for MongoDB
- Setting Up Mongoose
  - Installing and configuring Mongoose in an Express app
- Defining Mongoose Models
  - Creating schemas and models to represent MongoDB documents
- CRUD Operations with Mongoose
  - Performing CRUD operations using Mongoose models

10. Middleware and Authentication

- Custom Middleware in Express
  - Creating and using custom middleware for request handling
- User Authentication Basics
  - Implementing basic user authentication with sessions and cookies
  - Using packages like express-session and bcrypt for password hashing
- JWT (JSON Web Tokens)
  - Introduction to JWT for stateless authentication
  - Implementing JWT authentication in an Express app

11. Advanced Topics in Node.js and MongoDB

- Asynchronous Programming in Node.js
  - Understanding callbacks, promises, and async/await
- Working with Files and Streams
  - Reading and writing files asynchronously using the fs module
- Aggregation in MongoDB
  - Using MongoDB aggregation framework for advanced data processing
- Indexes in MongoDB
  - Creating and using indexes for optimizing queries

12. Building a Full-Stack Application

- Setting Up the Project Structure
  - Organizing the backend and frontend in a single project
- Integrating Frontend with Backend
  - Serving a frontend application (e.g., React) from the Node.js server
  - Connecting the frontend to the backend API
- Deploying the Application
  - Deploying the Node.js application to a cloud provider like Heroku or AWS
  - Setting up environment variables and configurations for production

13. Testing and Debugging

- Unit Testing with Mocha and Chai

    - Writing and running unit tests for Node.js modules
- Integration Testing
    - Testing API endpoints with tools like Postman or Supertest
- Debugging Node.js Applications
    - Using the Node.js debugger and logging techniques for troubleshooting

14. Security Best Practices

- Securing Express Applications
    - Implementing HTTPS, helmet, and other security measures
- Sanitizing User Input
    - Preventing SQL injection and cross-site scripting (XSS) attacks
- Managing Sensitive Data
    - Using environment variables and secure storage for secrets

HTML Tasks

HTML Introduction

1. Basic Structure of an HTML Document

Objective: Understand the basic structure of an HTML document.

a. Creating the HTML Skeleton

- Create a new HTML file named basic-structure.html.
- Define the basic structure of an HTML document, including <!DOCTYPE html>, <html>, <head>, <title>, and <body> tags.

b. Adding Metadata

- Inside the <head> tag, add a <meta> tag with the charset attribute set to "UTF-8".

c. Setting the Title

- Inside the <head> tag, add a <title> tag.
- Set the title to "Basic HTML Structure".

d. Adding Content to the Body

- Inside the <body> tag, add an <h1> tag with the content "Welcome to My HTML Document".
- Below the <h1> tag, add a <p> tag with the content "This document demonstrates the basic structure of an HTML page.".

Basic HTML Tags

2. Using Elementary Tags

Objective: Learn how to use basic HTML tags.

a. Headings and Paragraphs

- Create a new HTML file named elementary-tags.html.
- Define the basic structure of an HTML document.
- Inside the <body> tag, add <h1> to <h6> tags with different content for each heading level.
- Add a <p> tag below each heading with content describing that heading.

b. Adding Line Breaks and Horizontal Lines

- Below the paragraph tags, add a
 tag.
- Add another paragraph with the content "This is a new line after a break.".
- Below this paragraph, add an <hr> tag.

c. Inserting Images

- Below the horizontal line, add an <img> tag with the src attribute set to the URL of an image and the alt attribute set to "Sample Image".

d. Creating Links

- Below the image, add an <a> tag with the href attribute set to "https://www.example.com" and the content "Visit Example".

3. Block vs. Inline Tags

Objective: Understand the difference between block-level and inline-level tags.

a. Using Block-Level Tags

- Create a new HTML file named block-vs-inline.html.
- Define the basic structure of an HTML document.
- Inside the <body> tag, add three tags with the content "Block 1", "Block 2", and "Block 3".

b. Using Inline-Level Tags

- Below the block-level tags, add a <p> tag with the content "This is an example of using " followed by three tags with the content "inline ", "elements ", and "in a paragraph".

c. Mixing Block and Inline Tags

- Below the inline-level tags, add a tag.
- Inside the tag, add a <p> tag with the content "This is a block-level element containing an inline " followed by a tag with the content "element".

d. Observing Display Differences

- Add CSS styles to differentiate block and inline elements.
- Inside the <head> tag, add a <style> tag.
- Add styles to give a border to and tags to visualize their differences.
- Set the div border to "1px solid black" and span border to "1px dotted red".

4. Using Formatting Tags

Objective: Understand how to use basic formatting tags.

a. Using Bold and Italic Tags

- Create a new HTML file named formatting-tags.html.
- Define the basic structure of an HTML document.
- Inside the <body> tag, add a <h2> tag with the content "Formatting Tags".

- Below the <h2> tag, add a <p> tag with the content "This is an example of using " followed by a <b> tag with the content "bold" and a <i> tag with the content "italic".

b. Using the Underline Tag

- Below the previous <p> tag, add another <p> tag with the content "This text is " followed by a <u> tag with the content "underlined".

c. Using the Strikethrough Tag

- Below the previous <p> tag, add another <p> tag with the content "This text has " followed by a <s> tag with the content "strikethrough".

d. Combining Formatting Tags

- Below the previous <p> tag, add another <p> tag.
- Use a combination of <b>, <i>, <u>, and <s> tags to format the text "This is bold, italic, underlined, and strikethrough".

5. Creating Lists

Objective: Understand how to create and use ordered and unordered lists.

a. Creating an Unordered List

- Create a new HTML file named lists.html.
- Define the basic structure of an HTML document.
- Inside the <body> tag, add a <h2> tag with the content "Unordered List".
- Below the <h2> tag, add a <ul> tag.
- Inside the <ul> tag, add three <li> tags with the content "Item 1", "Item 2", and "Item 3".

b. Creating an Ordered List

- Below the unordered list, add a <h2> tag with the content "Ordered List".
- Below the <h2> tag, add an <ol> tag.
- Inside the <ol> tag, add three <li> tags with the content "First", "Second", and "Third".

c. Creating a Description List

- Below the ordered list, add a <h2> tag with the content "Description List".
- Below the <h2> tag, add a <dl> tag.
- Inside the <dl> tag, add three <dt> tags with the content "Term 1", "Term 2", and "Term 3".
- Add a <dd> tag below each <dt> tag with the description "Description for Term 1", "Description for Term 2", and "Description for Term 3".

d. Nesting Lists

- Inside one of the <li> tags in the unordered list, add another <ul> tag.
- Inside this nested <ul> tag, add two <li> tags with the content "Subitem 1" and "Subitem 2".

Forms

6. Creating Forms

Objective: Learn how to create and use HTML forms.

a. Creating a Basic Form

- Create a new HTML file named forms.html.
- Define the basic structure of an HTML document.
- Inside the <body> tag, add a <h2> tag with the content "Basic Form".
- Below the <h2> tag, add a <form> tag with the action attribute set to "#" and the method attribute set to "post".

b. Adding Input Fields

- Inside the <form> tag, add a <label> tag with the content "Name:".
- Below the <label> tag, add an <input> tag with the type attribute set to "text" and the name attribute set to "name".
- Add a
 tag for line break.

- Add another <label> tag with the content "Email:".
- Below the <label> tag, add an <input> tag with the type attribute set to "email" and the name attribute set to "email".

c. Adding a Submit Button

- Below the email input field, add a <button> tag with the type attribute set to "submit" and the content "Submit".

d. Adding a Textarea

- Inside the <form> tag, below the email input field, add a <label> tag with the content "Message:".
- Below the <label> tag, add a <textarea> tag with the name attribute set to "message" and set the rows attribute to "4" and the cols attribute to "50".

7. Creating a Contact Form

Objective: Learn how to use various input types in an HTML form.

a. Creating the Contact Form

- Create a new HTML file named contact-form.html.
- Define the basic structure of an HTML document.
- Inside the <body> tag, add a <h2> tag with the content "Contact Form".
- Below the <h2> tag, add a <form> tag with the action attribute set to "#" and the method attribute set to "post".

b. Adding Text and Email Input Fields

- Inside the <form> tag, add a <label> tag with the content "Full Name:".
- Below the <label> tag, add an <input> tag with the type attribute set to "text" and the name attribute set to "fullname".
- Add a
 tag for line break.
- Add another <label> tag with the content "Email Address:".
- Below the <label> tag, add an <input> tag with the type attribute set to "email" and the name attribute set to "email".

c. Adding Radio Buttons and Checkboxes

- Below the email input field, add a <label> tag with the content "Gender:".
- Below the <label> tag, add a <input> tag with the type attribute set to "radio" and the name attribute set to "gender" and the value set to "male". Add a label with the content "Male".
- Add another <input> tag with the type attribute set to "radio" and the name attribute set to "gender" and the value set to "female". Add a label with the content "Female".
- Below the radio buttons, add a <label> tag with the content "Subscribe to Newsletter:".
- Below the <label> tag, add an <input> tag with the type attribute set to "checkbox" and the name attribute set to "newsletter".

d. Adding a Submit Button

- Below the checkbox, add a <button> tag with the type attribute set to "submit" and the content "Submit".

Projects

8. Project: Personal Portfolio

Objective: Create a personal portfolio website using only HTML to display your profile, projects, and contact information.

Project Structure: Create an HTML page named index.html to include the following:

a. Homepage

- Basic page structure with <html>, <head>, and <body> tags.
- <title> tag: "Personal Portfolio"
- Header section with a <h1> tag: "Welcome to My Portfolio"
- Navigation menu with links to "About Me", "Projects", and "Contact" sections using <a> tags.

b. About Me Section

- Add an <h2> tag: "About Me"
- Add a <p> tag with a brief introduction about yourself.
- Include an <img> tag with your photo and a brief description using the alt attribute.

c. Projects Section

- Add an <h2> tag: "Projects"
- Create an unordered list (<ul>) with list items (<li>) to display your projects.
- Each project should have a title using <h3>, a brief description using <p>, and a link using <a> to the project's repository or live demo.

d. Contact Section

- Add an <h2> tag: "Contact"
- Create a contact form using <form>, <label>, <input>, and <textarea> tags.
- Include fields for "Name", "Email", and "Message", and a submit button using <button>.

9. Project: Simple Blog

Objective: Create a simple blog using only HTML to display posts and comments.

Project Structure:

a. Homepage

- Basic page structure with <html>, <head>, and <body> tags.
- <title> tag: "Simple Blog"
- Header section with a <h1> tag: "My Blog"
- Navigation menu with links to "Home", "Blog Posts", and "About" sections using <a> tags.

b. Home Section

- Add a <section> with the id attribute set to "home".

- Inside the section, add a <h2> tag: "Welcome to My Blog"
- Add a <p> tag with a brief introduction to your blog.

c. Blog Posts Section

- Add a <section> with the id attribute set to "posts".
- Inside the section, add a <h2> tag: "Blog Posts"
- Create a series of blog posts using <article> tags. Each post should have a <h3> for the title, a <p> for the content, and a <footer> for the author and date.

d. About Section

- Add a <section> with the id attribute set to "about".
- Inside the section, add a <h2> tag: "About Me"
- Add a <p> tag with a brief description about yourself and the purpose of the blog.

e. Comments Section

- Below each blog post, add a comments section using a <form> for users to submit comments.
- The form should include fields for "Name", "Email", and "Comment", and a submit button.

CSS Tasks

Selectors

10. Styling a Personal Profile

Objective: Learn to use basic CSS selectors.

a. Basic Structure and Linking CSS

- Create a new HTML file named profile.html.
- Define the basic structure of an HTML document.
- Create a new CSS file named profile.css.

- Link the CSS file to the HTML file inside the <head> tag using the <link> tag.

b. Styling by Element Selector

- Add a <h1> tag with the content "John Doe".
- Add a <p> tag with a brief introduction.
- In the CSS file, use the element selector to style the <h1> tag with a different font size and color.
- Style the <p> tag with a different font size and line height.

c. Styling by Class Selector

- Add a section with the class name "about-me".
- Inside this section, add a <h2> tag with the content "About Me".
- Add another <p> tag with more detailed information.
- In the CSS file, use the class selector to style the "about-me" section with padding and background color.

d. Styling by ID Selector

- Add an image of John Doe with the ID "profile-pic".
- In the CSS file, use the ID selector to set the width and border radius of the image.

11. Creating a Product Card

Objective: Learn to use different types of CSS selectors.

a. Basic Structure and Linking CSS

- Create a new HTML file named product.html.
- Define the basic structure of an HTML document.
- Create a new CSS file named product.css.
- Link the CSS file to the HTML file inside the <head> tag using the <link> tag.

b. Styling by Element and Descendant Selector

- Add a element with the class "product-card".
- Inside this , add an <h3> tag with the product name.
- Add a <p> tag with the product description.
- In the CSS file, use the descendant selector to style the <h3> and <p> tags inside the "product-card" class.

c. Styling by Attribute Selector

- Add a <button> element with the attribute type="button" and the content "Add to Cart".
- In the CSS file, use the attribute selector to style the button with a background color and padding.

d. Styling by Pseudo-classes

- Add a list of product features using an unordered list <ul> and list items <li>.
- In the CSS file, use the pseudo-class :hover to change the background color of the list items when hovered over.

12. Styling a Navigation Menu

Objective: Learn to use advanced CSS selectors.

a. Basic Structure and Linking CSS

- Create a new HTML file named navigation.html.
- Define the basic structure of an HTML document.
- Create a new CSS file named navigation.css.
- Link the CSS file to the HTML file inside the <head> tag using the <link> tag.

b. Styling by Element, Class, and ID Selectors

- Add a <nav> element with an unordered list <ul> and list items <li>.
- Add <a> tags inside each list item for the menu links.
- In the CSS file, use the element selector to remove the default list style of the unordered list.

- Use the class selector to style the <nav> element with a background color and padding.
- Use the ID selector to style a specific link differently.

c. Styling by Pseudo-elements

- Use the ::before pseudo-element to add a decorative element before each list item.
- In the CSS file, style the ::before pseudo-element with content, color, and margin.

d. Styling by Combinators

- Add a submenu inside one of the list items with another unordered list <ul> and list items <li>.
- Use the child combinator to style only the direct children of the main <ul> list.
- Use the adjacent sibling combinator to style a list item that follows another list item.

Basic CSS Properties

13. Styling a Personal Biography

Objective: Learn to use basic CSS properties.

a. Basic Structure and Linking CSS

- Create a new HTML file named biography.html.
- Define the basic structure of an HTML document.
- Create a new CSS file named biography.css.
- Link the CSS file to the HTML file inside the <head> tag using the <link> tag.

b. Setting Text Color and Background Color

- Add an <h1> tag with the content "John Doe".
- In the CSS file, set the text color of the <h1> tag to a color of your choice.

- Set the background color of the <body> tag to a light color.

c. Font-family and Font-size

- Add a <p> tag with a brief biography of John Doe.
- In the CSS file, set the font-family of the <p> tag to Arial.
- Set the font-size of the <p> tag to 16px.

d. Font-weight and Font-style

- Add another <p> tag with a quote by John Doe.
- In the CSS file, set the font-weight of the quote to bold.
- Set the font-style of the quote to italic.

14. Styling a Travel Blog Post

Objective: Learn to use various CSS properties.

a. Basic Structure and Linking CSS

- Create a new HTML file named travel-blog.html.
- Define the basic structure of an HTML document.
- Create a new CSS file named travel-blog.css.
- Link the CSS file to the HTML file inside the <head> tag using the <link> tag.

b. Setting up the Blog Post Content

- Inside the <body> tag, add an <h1> tag with the title "My Amazing Journey".
- Add a element with the class "post-content" to wrap the content.
- Inside the , add an <h2> tag with the subtitle "Exploring the Wonders of the World".
- Add a <p> tag with a brief introduction to the blog post.
- Add another <p> tag with a detailed description of the journey.
- Add a quote within the detailed description using a <blockquote> tag.

c. Applying CSS Properties

- In the CSS file, set the text color of the <h1> tag to a color of your choice.
- Set the background color of the <body> tag to a light color.
- Set the font-family of the entire document to "Arial, sans-serif".
- Set the font-size of the <h1> tag to 36px and the font-weight to bold.
- Set the font-size of the <h2> tag to 28px and the font-weight to bold.
- Set the font-size of the <p> tags to 16px and the line-height to 1.8.
- Set the font-style of the <blockquote> tag to italic.
- Set the text-decoration of the <h1> and <h2> tags to underline.
- Set the text-align of the <h1> tag to center and the <h2> tag to left.
- Set the background color of the .post-content div to a light shade and add some padding around it.

15. [Changed] Styling a News Article

Objective: Learn to use basic CSS properties.

a. Basic Structure and Linking CSS

- Create a new HTML file named news.html.
- Define the basic structure of an HTML document.
- Create a new CSS file named news.css.
- Link the CSS file to the HTML file inside the <head> tag using the <link> tag.

b. Setting Up the Article Structure

- Inside the <body> tag, add a element with the class "article-container".
- Inside the "article-container" div, add a <h1> tag with the title of the news article: "Breaking News: Major Event Unfolds".
- Below the title, add a element with the class "article-meta".

- Inside the "article-meta" div, add a tag for the author: "By Jane Smith".
- Add another tag for the publication date: "August 10, 2024".
- Below the "article-meta" div, add a <p> tag with a summary of the news article.
- Add another <p> tag with the detailed content of the article.

c. Applying Color and Background Color

- In the CSS file, set the text color of the <h1> tag to a dark color.
- Set the background color of the "article-container" to a light shade.
- Set the text color of the "article-meta" to a gray shade.

d. Font-family and Font-size

- Set the font-family of the entire document to "Georgia, serif".
- Set the font-size of the <h1> tag to 28px.
- Set the font-size of the "article-meta" to 14px.
- Set the font-size of the <p> tags to 18px.

e. Font-weight and Font-style

- Set the font-weight of the <h1> tag to bold.
- Set the font-weight of the "article-meta" to normal.
- Set the font-style of the "article-meta" to italic.

f. Text-decoration and Text-align

- Set the text-decoration of the "article-meta" to none.
- Set the text-align of the "article-container" to justify.

g. Line-height and Spacing

- Set the line-height of the <p> tags to 1.6.
- Add padding around the "article-container" to ensure the content does not touch the edges.

Box Model

16. [New] Exploring Inspect Element

Objective: Use the browser's "Inspect Element" tool to explore HTML elements, CSS styles, and the box model in detail, focusing on the Elements tab, Styles tab, and Computed tab.

a. Basic Structure and Linking CSS

- Create a new HTML file named inspect-element.html.
- Define the basic structure of an HTML document.
- Create a new CSS file named inspect-element.css.
- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Creating a Sample Structure

- Inside the <body>, add a element with the class "content-container".
- Inside this "content-container" div, add an <h1> tag with the content "Inspect Element Demo".
- Below the <h1> tag, add a <p> tag with the content "This paragraph is used to demonstrate the 'Inspect Element' tool in detail.".
- Below the <p> tag, add a element with the class "nested-box".
- Inside this "nested-box" div, add a <p> tag with the content "Nested Box with a paragraph.".

c. Styling the Elements

- In the CSS file, set the border of the "content-container" class to 2px solid #333.
- Set the padding to 15px.
- Set the margin to 20px.
- Set the background color to #f0f0f0.
- Set the font-family of the entire document to "Arial, sans-serif".
- For the "nested-box" class, set the border to 1px solid #007BFF.

- Set the padding to 10px.
- Set the margin to 10px 0.

d. Exploring the Elements Tab

- Right-click on the "content-container" div and select "Inspect" to open the Elements tab.
- Observe the HTML structure and how the elements are nested.
- Expand and collapse elements to understand the document structure.

e. Examining the Styles Tab

- In the Elements tab, click on the "content-container" div.
- In the Styles tab, observe the CSS properties applied to this element.
- Note the inheritance and overriding of styles. Observe how specific styles are crossed out when overridden by other rules.

f. Analyzing the Computed Tab

- Click on the Computed tab for the "content-container" div.
- Observe the list of all CSS properties computed for this element.
- Pay special attention to the box model diagram at the top, which displays the content, padding, border, and margin.

g. Observing the Box Model Diagram

- In the Computed tab, hover over different areas of the box model diagram to see the corresponding area highlighted in the browser.
- Note the specific dimensions for content, padding, border, and margin as displayed.
- Compare these values with the ones specified in the CSS file and observe how they contribute to the total space occupied by the element.

h. Comparing Elements

- Use the same process to inspect the "nested-box" div and its content.

- Compare the styles and computed values with those of the "content-container" div.
- Observe any differences in the box model, styles, and overall layout.

17. [New] Nested Box Elements

Objective: Understand the CSS Box Model by creating nested and independent elements, applying borders, padding, and margins to visualize the box structure.

a. Basic Structure and Linking CSS

- Create a new HTML file named nested-box-elements.html.
- Define the basic structure of an HTML document.
- Create a new CSS file named nested-box-elements.css.
- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Creating the Outer Container

- Inside the <body>, add a element with the class "outer-container".
- Inside this div, add a <h2> tag with the content "Outer Container".
- In the CSS file, set the border of the "outer-container" class to 3px solid #000. Set the padding to 20px and set the margin to 20px auto.

c. Adding Nested Element

- Inside the "outer-container", add a element with the class "nested-container-1".
- Inside this div, add a <h3> tag with the content "Nested Container 1".
- In the CSS file, set the border of the "nested-container-1" class to 2px solid #007BFF. Set the padding to 15px and set the margin to 10px.
- Inside the "nested-container-1" div, add another element with the class "nested-container-2".
- Inside this "nested-container-2" div, add a <p> tag with the content "This is Nested Container 2, inside Nested Container 1."

- In the CSS file, set the border of the "nested-container-2" class to 1px solid #FF5733. Set the padding to 10px and set the margin to 5px.

d. Creating Independent Boxes

- Below the "outer-container", add a element with the class "independent-box-1".
- Inside this div, add a <p> tag with the content "Independent Box 1".
- In the CSS file, set the border of the "independent-box-1" class to 2px dashed #00CC66. Set the padding to 10px and set the margin to 15px 0.
- Below "independent-box-1", add another element with the class "independent-box-2".
- Inside this div, add a <p> tag with the content "Independent Box 2".
- In the CSS file, set the border of the "independent-box-2" class to 2px dotted #FFCC00. Set the padding to 10px and set the margin to 15px 0.

e. Observing the Box Model using Inspect Element

- Use the browser's "Inspect Element" tool to observe that every HTML element is a box.
- Note the application of borders, padding, and margins.
- Observe how each element's content area, padding, border, and margin are displayed and how they affect the total space occupied by the element.

18. [Changed] Designing a Call-to-Action Button

Objective: Understand the CSS Box Model, focusing on box-sizing.

a. Basic Structure and Linking CSS

- Create a new HTML file named cta-button.html.
- Define the basic structure of an HTML document.
- Create a new CSS file named cta-button.css.
- Link the CSS file to the HTML file inside the <head> tag using the <link> tag.

b. Div and Heading for First Button

- Inside the <body> tag, add a element with the class "button-container-1".
- Inside this div, add a <h2> tag with the content "Button 1".
- Below the heading, add a <button> element with the class "cta-button-1" and the content "Click Here".

c. Div and Heading for Second Button

- Below the first div, add another element with the class "button-container-2".
- Inside this div, add a <h2> tag with the content "Button 2".
- Below the heading, add a <button> element with the class "cta-button-2" and the content "Click Here".

d. Applying Basic Box Model Properties

- In the CSS file, define common styles for the "cta-button-1" and "cta-button-2" classes.
- Set the border to 2px solid #007BFF.
- Set the padding to 10px 20px.
- Set the margin to 20px 0.
- Set the width to 150px and the height to 50px.
- Set the background color to #007BFF and text color to #ffffff.
- Add a cursor property set to pointer and center the text.

e. Box-Sizing Property

- For the "cta-button-1" class, set the box-sizing property to border-box.
- For the "cta-button-2" class, set the box-sizing property to content-box.

f. Observe using Inspect Element

- Use the browser's "Inspect Element" tool to observe the details of both buttons.

- Check the computed styles to see how the box-sizing property affects the total size of the buttons, especially how padding and border are calculated differently in border-box versus content-box.
- Compare the dimensions, padding, and border areas as displayed in the "Inspect Element" box model visualization.

19. [Changed] Creating a Box Sizing Demonstration

Objective: Understand the CSS Box Model, focusing on box-sizing.

a. Basic Structure and Linking CSS

- Create a new HTML file named box-sizing.html.
- Define the basic structure of an HTML document.
- Create a new CSS file named box-sizing.css.
- Link the CSS file to the HTML file inside the <head> tag using the <link> tag.

b. Setting up the Box Model Demonstration

- Inside the <body> tag, add a element with the class "box-demo-1".
- Inside this div, add a <h3> tag with the content "Box 1: Border-Box".
- Below the first div, add another element with the class "box-demo-2".
- Inside this div, add a <h3> tag with the content "Box 2: Content-Box".

c. Applying Common Box Model Properties

- In the CSS file, define common styles for both "box-demo-1" and "box-demo-2" classes.
- Set the border to 2px solid #333.
- Set the height to 150px and the width to 300px.
- Set the padding to 20px.
- Set the margin to 20px auto for centering.
- Set the background color to #f0f0f0.

d. Box-Sizing Property

- For the "box-demo-1" class, set the box-sizing property to border-box.
- For the "box-demo-2" class, set the box-sizing property to content-box.

e. Observe using Inspect Element

- Use the browser's "Inspect Element" tool to observe and document the differences between the two divs.
- Check the computed styles to see how the box-sizing property affects the total size of the divs.
- Note how padding and border are included in the total width and height for border-box and how they are added on top of the width and height for content-box.

Display

20. Exploring the CSS Display Property

Objective: Understand the different values of the CSS display property, focusing on block, inline, inline-block, and none, and observe their effects on layout, spacing, and element behavior.

a. Basic Structure and Linking CSS

- Create a new HTML file named display-property.html.
- Define the basic structure of an HTML document.
- Create a new CSS file named display-property.css.
- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Display: Block

- Inside the <body>, add a element with the class "block-element" containing the text "Block Element".
- Below the "block-element" div, add another element with the same class "block-element" containing the text "Another Block Element".
- In the CSS file, set the display property of the "block-element" class to block.

- Set a border to 2px solid #333 and add some padding with 10px.
- Set margin-bottom to 10px.

c. Display: Inline

- Below the block elements, add three elements with the class "inline-element" containing the text "Inline Element 1", "Inline Element 2", and "Inline Element 3".
- In the CSS file, set the display property of the "inline-element" class to inline.
- Set a border to 1px dashed #007BFF and add some padding with 5px.

d. Display: Inline-Block

- Below the inline elements, add three elements with the class "inline-block-element" containing the text "Inline-Block 1", "Inline-Block 2", and "Inline-Block 3".
- In the CSS file, set the display property of the "inline-block-element" class to inline-block.
- Set a border to 1px solid #FF5733 and add some padding with 5px.
- Set a width to 100px.

e. Display: None

- Below the inline-block elements, add a element with the class "none-element" containing the text "This element is hidden".
- In the CSS file, set the display property of the "none-element" class to none.

f. Observing the Differences

- Observe that block elements take up the full width available and start on a new line.
- Observe that inline elements sit next to each other horizontally and only take up as much width as necessary.
- Observe that inline-block elements sit next to each other like inline elements but can have set widths and heights like block elements.

- Observe that elements with display: none do not appear on the page and do not take up any space.

21. Width for Different Display Values

Objective: Understand how different display values (block, inline, inline-block) affect the width of elements, including default behavior and explicit width settings.

a. Basic Structure and Linking CSS

- Create a new HTML file named display-width.html.
- Define the basic structure of an HTML document.
- Create a new CSS file named display-width.css.
- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Display: Block

- Add a element with the class "block-element-default" containing the text "Block Element (Default Width)".
- Below it, add another element with the class "block-element-fixed" containing the text "Block Element (Fixed Width)".
- In the CSS file, set the display property of both elements to block.
- Set the border for both classes to 2px solid #333 and add padding with 10px.
- Set margin-bottom to 10px.
- For the "block-element-default" class, do not set any width. For the "block-element-fixed" class, set the width explicitly to 300px.

c. Display: Inline

- Add three elements with the class "inline-element-default" containing the text "Inline Element (Default Width)".
- Below them, add three elements with the class "inline-element-fixed" containing the text "Inline Element (Fixed Width)".
- In the CSS file, set the display property of both sets of elements to inline.

- Set the border for both classes to 1px dashed #007BFF and add padding with 5px.
- For the "inline-element-default" class, do not set any width. For the "inline-element-fixed" class, set the width explicitly to 100px.

d. Display: Inline-Block

- Add three elements with the class "inline-block-element-default" containing the text "Inline-Block Element (Default Width)".
- Below them, add three elements with the class "inline-block-element-fixed" containing the text "Inline-Block Element (Fixed Width)".
- In the CSS file, set the display property of both sets of elements to inline-block.
- Set the border for both classes to 1px solid #FF5733 and add padding with 5px.
- For the "inline-block-element-default" class, do not set any width. For the "inline-block-element-fixed" class, set the width explicitly to 150px.

e. Observing the Differences in Width

- Observe that the "block-element-default" takes up the full width of its parent container by default (100% width).
- The "block-element-fixed" takes up exactly 300px as set. There is space remaining on its right, still, it doesn't allow another element to sit next to it.
- Observe that "inline-element-default" takes up only as much width as the content requires (auto width).
- The "inline-element-fixed" does not respect the width: 100px setting as inline elements ignore width settings, but they still display the padding and border.
- Observe that "inline-block-element-default" takes up only as much width as the content requires (auto width).
- The "inline-block-element-fixed" respects the width: 150px setting and adjusts accordingly.

22. Default Display Values for HTML Elements

Objective: Demonstrate the default display property values for different HTML elements, such as , , <a>, <p>, <img>, and <li>. Observe how these default values affect the layout and behavior of elements.

a. Basic Structure and Linking CSS

- Create a new HTML file named default-display-values.html.
- Create a new CSS file named default-display-values.css.
- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Display: Block Elements

- Add a element with the class "block-element" containing the text "This is a div (block element)".
- Add a <p> element with the class "block-element" containing the text "This is a paragraph (block element)".
- Set a border for the "block-element" class to 2px solid #333.
- Set the padding to 10px.
- Set the margin-bottom to 10px.

c. Display: Inline Elements

- Add a element with the class "inline-element" containing the text "This is a span (inline element)".
- Add an <a> element with the class "inline-element" containing the text "This is a link (inline element)".
- Set a border for the "inline-element" class to 1px dashed #007BFF.
- Set the padding to 5px.

d. Display: Inline-Block

- Add an <img> element with the class "inline-block-element" and a placeholder image source [src = "https://via.placeholder.com/100"]. Set the alt attribute to "This is an image (inline-block element)".

- For the "inline-block-element" class, set the border to 1px solid #FF5733 and add padding with 5px.

e. List Item Elements

- Add an unordered list <ul> with three <li> elements, each with the class "list-item-element" and containing text like "List item 1", "List item 2", and "List item 3".
- For the "list-item-element" class, set the border to 1px dotted #00CC66 and add padding with 5px.

f. Observing the Default Display Values

- Observe that, for the and the <p> elements, we have not set any width or display, but still they take up the full width available and start on a new line. This is because their default display value is block.
- Observe that, the and the <a> elements, sit next to each other horizontally and only take up as much width as necessary. This is because their default display value is inline.
- Observe that images, by default, behave like inline elements but can be treated as block elements with set width and height. This is because the default value for <img> is inline-block.
- Observe that list items in an unordered list (<ul>) have a block-like behavior with automatic new lines, but are styled to appear as a list.

23. Centering Elements

Objective: Learn how to center inline, inline-block, and block elements using CSS properties like text-align and margin: auto.

a. Basic Structure and Linking CSS

- Create a new HTML file named centering-elements.html.
- Create a new CSS file named centering-elements.css.
- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Text Align for Inline Elements

- Inside the <body>, add a element with the class "inline-container" containing two elements with the class "inline-element" and the text "Inline Element 1" and "Inline Element 2".
- For the "inline-container" class, set the text-align property to center.
- For the "inline-element" class, set the border to 1px solid #007BFF and add padding with 5px.

c. Text Align for Inline-Block Elements

- Below the "inline-container" div, add another element with the class "inline-block-container" containing two elements with the class "inline-block-element" and the text "Inline-Block Element 1" and "Inline-Block Element 2".
- For the "inline-block-container" class:
    1. Set the text-align property to center.
- For the "inline-block-element" class:
    1. Set the display property to inline-block.
    2. Set the border to 1px solid #FF5733
    3. Set the padding to 10px.

d. Margin Auto for Block Elements

- Below the "inline-block-container" div, add a element with the class "block-container" containing a element with the class "block-element" and the text "Block Element (Centered)".
- For the "block-element" class:
    1. Set the margin property to 10px auto.
    2. Set the width property to 200px.
    3. Set the border to 2px solid #333
    4. Set padding to 10px.

e. Observing Centering Techniques

- Observe how the text-align: center property in the "inline-container" class centers the inline elements horizontally within the container.
- Observe how the text-align: center property in the "inline-block-container" class centers the inline-block elements, which respect width and height properties.
- Observe how the margin: auto property in the "block-element" class centers the block element horizontally, as text-align does not apply to block elements.

Page Structuring

24. Page Layout with Inline-Block

Objective: Learn to structure a web page layout with rows and columns using display: inline-block.

a. Create Basic HTML Structure

- Create a new HTML file named display-layout.html.
- Create a new CSS file named display-layout.css.
- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Add Header, Main, and Footer Sections

- Inside the <body>, add a element with the class "header" and content "Header".
- Add a element with the class "main-content".
- Add a element with the class "footer" and content "Footer".

c. Create Rows and Columns in the Main Section

- Inside the "main-content" element, add a element with the class "row".
- Inside this "row" div, add two elements with the class "column-2" and content "Column 1" and "Column 2".
- Inside the "main-content" element, add another element with the class "row".
- Inside this second "row" div, add three elements with the class "column-3" and content "Column 1", "Column 2", and "Column 3".

d. Styling the columns:

- In the CSS file, set the display property of the "column-2" and "column-3" classes to inline-block.
- For the "column-2" class, set the width to 49%.
- For the "column-3" class, set the width to 32%.
- Set a height of 200px to both classes.
- Set background colors for both classes to #f0f0f0 to visually differentiate sections.
- Set a border for both classes to 1px solid #ddd and add padding with 10px.
- Add box-sizing: border-box to ensure padding and border are included in the width.

e. Styling the header, footer, main-content and row:

- Set a border for header, footer, main-content to 2px solid black to make them visually distinct.
- Set background colors for header and footer to #4CAF50 to visually differentiate sections.
- Set a height of 50px to the header and the footer. Set a line-height to 50px.
- Set the text-align property for header and footer to center.
- Add a margin-bottom of 20px to the "row" class to separate rows.

f. Observe the Page Layout

- Open the HTML file in a web browser and examine the layout.
- Notice the clear structure of the header, rows, columns, and footer, emphasized by the set heights and borders.
- Understand that creating a structured layout like this is the first step before placing the actual content.
- Observe how display: inline-block allows the columns to sit next to each other, forming the desired layout.

- Pay attention to the gaps between columns and understand why width: 49% is used for .column-2 instead of width: 50%.
- Try changing the .column-2 width to 50% and observe how the layout breaks, causing the columns to wrap to the next line.

25. Float Property

Objective: Learn what the float property is, how it works

a. a. Basic Structure and Linking CSS

- Create a new HTML file named float-basics.html.
- Create a new CSS file named float-basics.css.
- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Exploring Float: Left

- Inside the <body>, add a element with the class "container".
- Inside the "container" div, add an <img> element with the class "float-left" and a placeholder image source [src = "https://via.placeholder.com/150"].
- Add a <p> element with the class "first-paragraph" after the image with some text.
- In the CSS file, set the float property of the "float-left" class to left.

c. Exploring Float: Right

- Below the previous <p> element, add another <img> element with the class "float-right" and a placeholder image source [src = "https://via.placeholder.com/150"].
- Add another <p> element with the class "second-paragraph" after the second image with some placeholder text.
- In the CSS file, set the float property of the "float-right" class to right.

d. Observing the Effects of Float Using Inspect Element

- Open the HTML file in a web browser and observe how the images are positioned within the container.

- Use the inspect element tool to observe how the first image floats to the left.
- The first paragraph, even though it is a block element, starts immediately after the image and flows around it.
- Observe how the first paragraph takes up the entire space on the right side of the container.
- The second image floats to the right but starts after the first paragraph, as the paragraph takes up the entire space on the right.
- The second paragraph starts on the left but after the first image.
- Observe how both floating images are removed from the normal flow—the paragraph elements extend from the left end to the right end of the container but appear immediately after the images due to their floated positioning.

e. Experimenting with the Clear Property

- In the CSS file, apply clear: left; to the "second-paragraph" class. The second paragraph starts below the first floated image, which is on the left. This is because clear: left; prevents the paragraph from appearing beside the left-floated element, pushing it down below the image.
- Change the clear property to clear: right. The second paragraph moves slightly down, clearing the second image on the right and starting below it. The paragraph doesn't appear beside the right-floated element and instead clears that space.
- Finally, change the clear property to clear: both;. While there may be no additional visible change from clear: right, this setting ensures that the second paragraph clears both the left and right-floated images. This ensures that the paragraph starts on a completely new line, below all floated elements, even if there were more floated elements on the left or right.

26. Page Layout with Float Left

Objective: Learn how to create a page layout using float: left and understand the issues with float and how to fix them.

a. Basic Structure and Linking CSS

- Create a new HTML file named float-layout.html.
- Create a new CSS file named float-layout.css.

- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Add Header, Main, and Footer Sections

- Inside the <body>, add a element with the class "header" and content "Header".
- Add a element with the class "main-content".
- Add a element with the class "footer" and content "Footer".

c. Create Rows and Columns in the Main Section

- Inside the "main-content" element, add a element with the class "row".
- Inside this "row" div, add two elements with the class "column-2" and content "Column 1" and "Column 2".
- Inside the "main-content" element, add another element with the class "row".
- Inside this second "row" div, add three elements with the class "column-3" and content "Column 1", "Column 2", and "Column 3".

d. Styling the columns:

- In the CSS file, set the float property of the "column-2" and "column-3" classes to left.
- For the "column-2" class, set the width to 50%.
- For the "column-3" class, set the width to 33.3%.
- Set a border for both classes to 1px solid #ddd and add padding with 10px.
- Add box-sizing: border-box for both classes to ensure padding and border are included in the width.
- Set a height of 200px for both classes.
- Set background colors for both classes to #f0f0f0.

e. Styling the header, footer, main-content and row:

- Set a border for the "header", "footer", and "main-content" to 2px solid black to make them visually distinct.

- Set the background color for the "header" and "footer" to #4CAF50 to visually differentiate sections.
- Set a height of 50px for the "header" and "footer", and set line-height to 50px.
- Set the text-align property for the "header" and "footer" to center.
- Add a margin-bottom of 20px to the "row" class to separate rows.

f. Observing the Effects of Float

- Open the HTML file in a web browser and observe the layout.
- Notice how the layout is created using float: left instead of display: inline-block, and the columns are similarly aligned next to each other.
- Observe how the width is set to 50% for "column-2" and 33.3% for "column-3", and it works without any gaps.
- Open the browser's inspect element tool and observe how the "main-content" and "row" elements do not wrap around the floated columns, leading to the collapse:
    1. Notice that the parent "row" elements have zero height, causing the subsequent content to overlap with the floated elements.
    2. This happens because floating an element removes it from the normal document flow, so the parent container does not expand to contain the floated elements.

g. Fixing Float Issues with Clearfix

- Add the following code to the CSS:

```
.row::after {
    content: "";
    display: block;
    clear: both;
}
```

- This CSS rule works by using the ::after pseudo-element to insert a block-level element after the floated elements within each .row. The content: ""; declaration creates an empty pseudo-element, and display: block; makes it a block element. The clear: both; declaration clears both left and right floats, ensuring that the parent .row container expands to contain its floated child elements, thus preventing the collapse.

- By applying this clearfix method directly to the .row class, the parent container is forced to recognize the height of the floated elements within it, maintaining the proper structure and layout without collapsing.

Flexbox

27. Page Layout with Flexbox

Objective: Learn how to use Flexbox to create a basic page layout similar to one previously created with float.

a. Basic Structure and Linking CSS

- Create a new HTML file named flexbox-layout.html.
- Create a new CSS file named flexbox-layout.css.
- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Add Header, Main, and Footer Sections

- Inside the <body>, add a element with the class "header" and content "Header".
- Add a element with the class "main-content".
- Add a element with the class "footer" and content "Footer".

c. Creating Rows and Columns in the Main Section

- Inside the "main-content" element, add a element with the class "row".
- Inside this "row" div, add two elements with the class "column-2" and content "Column 1" and "Column 2".
- Inside the "main-content" element, add another element with the class "row".
- Inside this second "row" div, add three elements with the class "column-3" and content "Column 1", "Column 2", and "Column 3".

d. Styling the Columns

- In the CSS file, set the display property of the "row" class to flex.

- For the "column-2" class, set the width to 50%.
- For the "column-3" class, set the width to 33.3%.
- Set a border for both classes to 1px solid #ddd and add padding with 10px.
- Add box-sizing: border-box for both classes to ensure padding and border are included in the width.
- Set a height of 200px for both classes.
- Set background colors for both classes to #f0f0f0.

e. Styling the Header, Footer, Main-Content, and Row

- Set a border for the "header", "footer", and "main-content" to 2px solid black to make them visually distinct.
- Set the background color for the "header" and "footer" to #4CAF50 to visually differentiate sections.
- Set a height of 50px for the "header" and "footer", and set line-height to 50px.
- Set the text-align property for the "header" and "footer" to center.
- Add a margin-bottom of 20px to the "row" class to separate rows.

f. Observing the Effects of Flexbox

- Open the HTML file in a web browser and observe the layout.
- Notice how the layout is created using display: flex instead of float: left, and the columns are similarly aligned next to each other.
- Open the browser's inspect element tool and observe how the parent "row" elements do not collapse, as Flexbox ensures that the parent container expands to contain its children, preserving the layout's integrity.

28. Flex-basis, Flex-grow, and Flex-shrink

Objective: Learn how to use the flex-basis, flex-grow, and flex-shrink properties in Flexbox to control the size of flex items. Understand the concepts of available space and deficit space in a flex container.

a. Basic Structure and Linking CSS

- Create a new HTML file named flex-properties.html.
- Create a new CSS file named flex-properties.css.
- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Setting Up the Flex Container and Items

- Inside the <body>, add a element with the class "container".
- Inside the "container" div, add three elements with the classes "box box1", "box box2", and "box box3". Set the content of these elements as "Box 1", "Box 2", and "Box 3" respectively.
- In the CSS file, for the "container" class, set the display property to flex.
- For the "container" class, set a border property of 2px dashed black, width of 500px, and margin: auto to center the container.
- For the "box" class, set box-sizing to border-box, background-color to #c3dbeb, border to 2px solid #608BA8, and height to 100px;
- Open the HTML file in a web browser and observe how each box is taking up only the space defined by its content.

c. Initial Setup with Flex-Basis and Available Space.

- Set the flex-basis property of the "box1", "box2", and "box3" classes to 100px.
- Open the HTML file in a web browser and using inspect element observe how each box is allocated 100px of space, leaving 200px of available space within the 500px container.

d. Introducing Flex-Grow

- Set the flex-grow property for the "box1", "box2", and "box3" classes to 2, 1 and 1 respectively.
- Observe the layout in the browser. Notice how the available 200px is distributed among the boxes in the ratio of 2:1:1, which is 100px, 50px and 50px, resulting in final widths of 200px (100px+100px) for "Box 1", and 150px (100px+50px) for "Box 2" and "Box 3".
- Understand that flex-grow allows boxes to increase their size proportionally to fill the available space when there is extra space in the container.

- Experiment by changing the flex-grow values for each box to different numbers. For example, set flex-grow: 3 for box1, flex-grow: 0 for box2, and flex-grow: 2 for box3.

e. Exploring Deficit Space with Flex-Basis and Flex-Shrink

- Change the flex-basis of each box to 200px.
- Set the flex-shrink property of each box to 0.
- Open the HTML file in a web browser and observe how the boxes now exceed the container width (a total width of 600px in a 500px container), creating a deficit of 100px.
- Understand that this happened because the boxes were not allowed to shrink (flex-shrink: 0), which led to them spilling out of the container.

f. Introducing Flex-Shrink to Handle Deficit Space

- Set the flex-shrink property for the "box1", "box2", and "box3" classes to 1, 1 and 2 respectively.
- Observe the layout in the browser. Notice how the deficit of 100px is distributed among the boxes in the ratio of 1:1:2, which is 25px, 25px and 50px, resulting in final widths of 175px (200px-25px) for "Box 1", 175px (200px-25px) for "Box 2", and 150px (200px-50px) for "Box 3`.
- Understand that flex-shrink allows boxes to reduce their size proportionally to fit within the available space when there is a deficit.
- Experiment by changing the flex-shrink values for each box to different numbers. For example, set flex-shrink: 0 for box1, flex-shrink: 2 for box2, and flex-shrink: 3 for box-3.

29. Flex-wrap

Objective: Learn how to use the flex-wrap property in Flexbox to manage the layout of flex items when there is a deficit of space. Understand how flex-wrap interacts with flex-grow and flex-shrink properties in the presence of both available and deficit space.

a. Basic Structure and Linking CSS

- Create a new HTML file named flex-wrap.html.
- Create a new CSS file named flex-wrap.css.

- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Setting Up the Flex Container and Items

- Inside the <body>, add a element with the class "container".
- Inside the "container" div, add three elements with the classes "box box1", "box box2", and "box box3". Set the content of these elements as "Box 1", "Box 2", and "Box 3" respectively.
- In the CSS file, for the "container" class, set the display property to flex.
- For the "container" class, set a border property of 2px dashed black, width of 500px, and margin: auto to center the container.
- For the "box" class, set box-sizing to border-box, background-color to #c3dbeb, border to 2px solid #608BA8, and height to 100px.
- Open the HTML file in a web browser and observe how each box is taking up only the space defined by its content.

c. Initial Setup with Flex-Basis and Deficit Space

- Set the flex-basis property of the "box1", "box2", and "box3" classes to 200px.
- Set the flex-shrink property of each box to 0.
- Open the HTML file in a web browser and observe how the boxes now exceed the container width (a total width of 600px in a 500px container), creating a deficit of 100px.
- Understand that this happened because the boxes were not allowed to shrink (flex-shrink: 0), which led to them spilling out of the container.

d. Introducing Flex-Shrink to Handle Deficit Space

- Set the flex-shrink property for the "box1", "box2", and "box3" classes to 0, 1, and 3 respectively.
- Observe the layout in the browser. Notice how the deficit of 100px is distributed among the boxes in the ratio of 0:1:3, resulting in final widths of 200px (200px-0px) for "Box 1", 175px (200px-25px) for "Box 2", and 125px (200px-75px) for "Box 3`.

- Understand that flex-shrink allows boxes to reduce their size proportionally to fit within the available space when there is a deficit.

e. Introducing Flex-Wrap

- Add the flex-wrap property to the "container" class and set it to wrap.
- Open the HTML file in a web browser and observe how the boxes now wrap onto the next line when there is a deficit of space in the container.
- Notice that with flex-wrap enabled, the boxes wrap to the next line instead of shrinking further, effectively overriding the flex-shrink property.

f. Exploring Available Space After Wrapping

- Set the flex-grow property for the "box1", "box2", and "box3" classes to 2, 1, and 1 respectively.
- Observe how the boxes on the wrapped lines expand according to their flex-grow values.
- On the first line, "Box 1" and "Box 2" share the available space (100px) in a 2:1 ratio. "Box 1" grows by 67px (200px + 67px), and "Box 2" grows by 33px (200px + 33px).
- On the second line, "Box 3" is the only box present, so it takes up all the available space on that line (300px), resulting in a final width of 500px (200px + 300px).
- Understand that when elements wrap to different lines, the flex-grow property applies to each line independently, distributing available space according to the flex-grow ratios of the boxes on that specific line.

30. Justify-content and Align-items

Objective: Learn how to use the justify-content and align-items properties in Flexbox to control the horizontal and vertical alignment of flex items. Understand how justify-content distributes available space and how align-items handles vertical alignment.

a. Basic Structure and Linking CSS

- Create a new HTML file named flex-alignment.html.
- Create a new CSS file named flex-alignment.css.

- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Setting Up the Flex Container and Items

- Inside the <body>, add a element with the class "container".
- Inside the "container" div, add three elements with the classes "box box1", "box box2", and "box box3". Set the content of these elements as "Box 1", "Box 2", and "Box 3" respectively.
- In the CSS file, for the "container" class, set the display property to flex.
- For the "container" class, set a border property of 2px dashed black, width of 600px, height of 300px, and margin: auto to center the container.
- For the "box" class, set box-sizing to border-box, background-color to #c3dbeb, border to 2px solid #608BA8, width to 100px, and height to 100px.

c. Exploring justify-content

- In the CSS file, set the justify-content property for the "container" class to *flex-start*. Open the HTML file in a web browser and observe how the boxes are aligned to the start of the container with no space distributed between them.
- Change the justify-content property to *center* and observe how the boxes are centered within the container, with equal space on either side.
- Change the justify-content property to *space-between* and observe how the available space is distributed evenly between the boxes, with the first box aligned to the start and the last box aligned to the end of the container.
- Change the justify-content property to *space-around* and observe how the available space is distributed both between the boxes and on the outside edges, creating equal space around each box.
- Change the justify-content property to *space-evenly* and observe how the available space is distributed evenly between and around the boxes, with equal space before the first box, after the last box, and between each box.

d. Observing the Effect of Flex-Grow on justify-content

- Set the flex-grow property of "box1" to 1.
- Open the HTML file in a web browser and observe how "Box 1" grows to take up all the available space.

- Notice how the justify-content property no longer makes sense because "Box 1" is growing to fill the space, leaving no extra space to distribute among the other items.

e. Exploring align-items

- In the CSS file, set the align-items property for the "container" class to *flex-start*. Observe how the boxes are aligned to the top of the container (the start of the cross-axis).
- Change the align-items property to *center* and observe how the boxes are vertically centered within the container.
- Change the align-items property to *flex-end* and observe how the boxes are aligned to the bottom of the container (the end of the cross-axis).

f. Experimenting with Different Heights for Each Element

- Modify the height of each box in the CSS file. Set the height of "box1" to 100px. Set the height of "box2" to 150px. Set the height of "box3" to 200px.
- Change the align-items property to *flex-start* and observe how the boxes are aligned to the top of the container.
- Change the align-items property to center and observe how the boxes are vertically centered within the container.
- Change the align-items property to flex-end and observe how the boxes are aligned to the bottom of the container.

Media Queries

31. Introduction to Media Queries

Objective: Learn the basics of media queries in CSS by applying simple style changes based on screen width. Understand how media queries adapt web content to different screen sizes using the inspect element tool for detailed observation.

a. Basic Structure and Linking CSS

- Create a new HTML file named basic-media-queries.html.
- Create a new CSS file named basic-media-queries.css.

- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Setting Up the Basic Page Structure

- Inside the <body>, add a element with the class "container".
- Inside the "container" div, add an <h1> element with the content "Media Queries Basics".
- Below the <h1> element, add a <p> element with the content "Resize the browser window to see the effect of media queries.".
- Below the <p> element, add a element with the class "box" and content "I am a box!".

c. Applying Basic Styling

- Set a max-width of 1200px and margin: auto for the "container" class to center it on the page.
- For the "box" class, set a background-color to #c3dbeb, text-align to center, border to 2px solid #608BA8, margin-top to 20px, width to 300px, and height to 200px.
- Set the line-height property of the "box" class to 200px to center the text vertically.
  When the line-height is equal to the height of the element, the text aligns vertically in the middle because the space above and below the text is distributed evenly within the element.

d. Introducing Media Queries

- In the CSS file, add a media query for screens with a max-width of 768px.
- Inside the media query, change the background-color of the "box" class to #ffcccb.
- Change the width of the 'box' class to 50% to make it occupy half of the container's width on smaller screens.
- Add another media query for screens with a max-width of 480px.
- Inside this media query, change the background-color of the "box" class to #90ee90.

- Set the border-radius of the "box" class to 15px to give it rounded corners.
- Change the font size of the <h1> element to 18px.

e. Observing the Effects of Media Queries

- Open the HTML file in a web browser and observe the initial layout.
- Resize the browser window to different widths (e.g., desktop, tablet, and mobile widths) by dragging the edges of the browser window.
- Observe the changes in the 'box' element's appearance as the screen width decreases, particularly as it goes below 768px and below 480px.
- Understand that media queries allow you to apply different CSS styles based on the screen size, ensuring your content looks good on all devices.

32. Responsive Layouts with Media Queries

Objective: Learn how to use media queries in CSS to create responsive page layouts that adapt to different screen sizes, such as desktop and mobile.

a. Basic Structure and Linking CSS

- Create a new HTML file named responsive-layout.html.
- Create a new CSS file named responsive-layout.css.
- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Add Header, Main, and Footer Sections

- Inside the <body>, add a element with the class "header" and content "Responsive Layout".
- Add a element with the class "main-content".
- Add a element with the class "footer" and content "Footer".

c. Creating Rows and Columns in the Main Section

- Inside the "main-content" element, add a element with the class "row".
- Inside this "row" div, add two elements with the class "column-2" and content "Column 1" and "Column 2".

- Inside the "main-content" element, add another element with the class "row".
- Inside this second "row" div, add three elements with the class "column-3" and content "Column 1", "Column 2", and "Column 3".

d. Styling the Columns

- In the CSS file, set a height of 200px for both classes "column-2" and "column-3".
- Set a border for both classes to 1px solid #ddd and add padding with 10px.
- Add box-sizing: border-box for both classes to ensure padding and border are included in the width.
- Set background colors for both classes to #f0f0f0.

e. Styling the Header, Footer, Main-Content, and Row

- Set a border for the "header", "footer", and "main-content" to 2px solid black to make them visually distinct.
- Set the background color for the "header" and "footer" to #4CAF50 to visually differentiate sections.
- Set a height of 50px for the "header" and "footer, and set line-height to 50px`.
- Set the text-align property for the "header" and "footer" to center.
- Add a margin-bottom of 20px to the "row" class to separate rows.

f. Styling the Desktop Layout

- Set the max-width of the "main-content" class to 1200px and margin: auto to center it on the page.
- Set the display property of the "row" class to flex.
- For the "column-2" class, set the width to 50%.
- For the "column-3" class, set the width to 33.3%.

g. Applying Media Queries for Mobile Layout

- In the CSS file, add a media query for screens with a max-width of 768px.

- Inside the media query, set the flex-direction of the "row" class to column to stack the "column-2" and "column-3" elements vertically.
- Set the width of the "column-2" and "column-3" classes to 100% to make them full-width on smaller screens.
- Adjust the padding and margins as needed to improve readability on mobile devices.

h. Observing the Responsive Layout in Action

- Open the HTML file in a web browser and observe the layout on a desktop screen.
- Open the browser's inspect element tool, make it dock to right and resize the browser window to trigger the media query.
- Observe how the layout changes from a multi-column layout on desktop to a stacked layout on mobile devices.

Projects

33. Project: Cloning the YouTube Website

Objective: Recreate the YouTube interface using HTML and CSS, focusing on layout techniques, responsiveness, and UI styling. This project will help you master CSS Grid, Flexbox, media queries, and modern CSS features while paying attention to design details.

a. Basic Structure and Linking CSS

- Create a new HTML file named youtube-clone.html.
- Create a new CSS file named youtube-clone.css.
- Link the CSS file to the HTML file using the <link> tag inside the <head> section.

b. Recreate the Header Section

- Replicate YouTube's header, including the logo, search bar, and user profile area.
- Pay attention to the placement, size, and spacing of each element.
- Ensure the header remains fixed at the top of the page as you scroll.

c. Create the Sidebar Navigation

- Build the sidebar navigation similar to YouTube's, including links like "Home", "Trending", "Subscriptions", and others.
- Ensure the sidebar is styled to match YouTube's design, including colors, font sizes, and spacing.
- Make sure the sidebar can be collapsed or hidden on smaller screens to save space.

d. Build the Main Content Area

- Design the main content area to display video thumbnails in a grid layout.
- Each video thumbnail should include an image, video title, channel name, and view count, styled to mimic YouTube’s layout.
- Ensure the grid is responsive, adjusting the number of columns based on screen size.

e. Design the Footer Section

- Replicate YouTube’s footer, including links like "About", "Press", "Copyright", and "Contact".
- Ensure the footer is horizontally aligned and styled consistently with YouTube’s design.

f. Implement Responsiveness

- Use media queries to make the YouTube clone responsive, ensuring it looks good on different screen sizes from desktops to mobile devices.
- Ensure that as the screen size changes, elements adjust appropriately, such as the sidebar collapsing or the video grid displaying fewer columns.
- Test the design thoroughly to ensure it remains functional and visually consistent on all devices.

g. Final Review and Refinement

- Open the HTML file in a web browser and compare it side-by-side with the actual YouTube site.
- Make adjustments to ensure that every element, from fonts to spacing to colors, closely matches the original design.

- Ensure all components are responsive and adapt well to different screen sizes, maintaining the integrity of the design.

34. Project: Responsive Single-Page Portfolio Website

Objective: Objective: Create a responsive single-page portfolio website using HTML and CSS. This project will help you demonstrate your ability to structure a webpage, design various sections, and ensure the page is responsive across different devices.

a. Basic Structure and Linking CSS

- Create a new HTML file named portfolio.html.
- Create a new CSS file named portfolio.css.
- Link the CSS file to the HTML file.

b. Creating the Header Section

- Design a header section that includes your name or the name of your portfolio, and a navigation menu with links to the different sections of the page.
- Ensure that the navigation links smoothly scroll to the corresponding sections on the page.

c. About Me Section

- Add an "About Me" section that provides a brief introduction about yourself, including your background, skills, and interests.
- Include a profile picture or avatar, and format the content in a visually appealing way.

d. Portfolio Section

- Create a "Portfolio" section where you showcase your work or projects.
- For each project, include a title, a brief description, and a link to the project or a live demo.
- Display the projects in a grid or list layout.

e. Contact Section

- Add a "Contact" section where visitors can find your contact information.

- Include a contact form with fields for the user's name, email, and message.
- Provide additional ways to contact you, such as social media links or an email address.

f. Footer Section

- Design a footer that includes any additional information or links, such as copyright information, terms of service, or a link back to the top of the page.
- Ensure that the footer is consistent with the overall design of the website.

g. Implementing Responsiveness

- Ensure that the entire website is responsive and adapts well to different screen sizes, from large desktop monitors to small mobile devices.
- Use media queries to adjust the layout, font sizes, and other elements as needed to maintain a good user experience across all devices.

h. Final Review and Refinement

- Open the HTML file in a web browser and test the responsiveness of your design by resizing the browser window.
- Ensure that all sections are accessible, visually appealing, and function well on different devices.
- Refine any elements that need adjustment to ensure a polished, professional-looking portfolio website.

JavaScript Tasks

Each numbered one is a task. The alphabet ones are sub-tasks.

One separate JavaScript file should be created for each task.

After you have created a task, spend some time observing your code.

Functions

35. Greet Function

a. Greet Function:

- Define a function named *greet()* that takes a parameter name.
- Inside the function, return a greeting message using the provided name. For example, if the name is "Alice", return "Hello, Alice!".
- Call the greet function with your name as the argument.
- Log the result to the console.

b. GreetDefault Function:

- Define another function similar to greet but name it *greetDefault*.
- This function should have a default parameter name set to "Guest".
- If no name is provided when calling this function, it should return "Hello, Guest!".
- Call the greetDefault function without any arguments and log the result to the console.

c. Rewrite Greet as a Function Expression:

- Rewrite the greet function as a function expression and store it in a variable called *greetFunction*.
- Call the greetFunction with your name as the argument and log the result to the console.

d. Rewrite Greet as an Arrow Function:

- Rewrite the greet function as an arrow function with a separate name *greetArrow*.
- Call the greetArrow() function with your name as the argument and log the result to the console.

36. Calculate Area

a. Calculate Area Function:

- Define a function named *calculateArea* that takes two parameters: width and height.
- Inside the function, return the area of a rectangle by multiplying width and height.

- Call the calculateArea function with width = 5 and height = 10.
- Log the result to the console.

b. Modify Calculate Area Function with Default Values:

- Define a new function named *calculateAreaWithDefaults* that has default values of 1 for both width and height.
- This means that if no arguments are provided, the function should return 1.
- Call the calculateAreaWithDefaults function without any arguments and log the result to the console.
- Call the calculateAreaWithDefaults function with width = 5 and height = 10, and log the result to the console.

c. Rewrite Calculate Area as a Function Expression:

- Rewrite the calculateArea function as a function expression and store it in a variable called *calculateAreaFunction*.
- Call the calculateAreaFunction with width = 5 and height = 10, and log the result to the console.

d. Rewrite Calculate Area as an Arrow Function:

- Rewrite the calculateArea function as an arrow function with a separate name *calculateAreaArrow*.
- Call the calculateAreaArrow function with width = 5 and height = 10, and log the result to the console.

37. Higher Order and Callback functions:

a. Define Higher Order Function:

- Define a function named *higherOrderFunction* that takes two parameters: num (a number) and callback (a function).
- Inside higherOrderFunction, call the callback function and pass num as an argument.

b. Define Callback Function and Call Higher Order Function:

- Define a function named *callbackFunction* that takes a number as an argument and logs it to the console.
- Call higherOrderFunction with a number (e.g., 5) and pass callbackFunction as the callback.

c. Call Higher Order Function with a Function Expression as Callback:

- Call higherOrderFunction with a number (e.g., 10) and a function expression as the callback.
- The function expression should log the number to the console.

d. Callback to Log Square of the Number:

- Call higherOrderFunction again with a number (e.g., 4) and a function expression as the callback.
- The new function expression should log the square of the number to the console.

e. Callback to Log Sum of Two Numbers:

- Define a new function named *newHigherOrderFunction* that takes three parameters: num1, num2, and callback.
- Inside newHigherOrderFunction, call the callback function with num1 and num2 as arguments.
- Call newHigherOrderFunction with two numbers (e.g., 3 and 7) and a function expression as the callback.
- The function expression should take two parameters and log their sum to the console.

38. Simple Mathematical Operations:

a. Define Callback Functions:

- Define a callback function named doubleNumber that takes a number and returns its double.
- Define another callback function named squareNumber that takes a number and returns its square.

- Define another callback function named incrementNumber that takes a number and returns the number incremented by one.

b. Define performOperation Function:

- Define a function named performOperation that accepts two parameters: num (a number on which to perform the operation) and operation (a callback function that specifies the operation to be performed on num).
- Inside performOperation, call the operation function and pass num as an argument.
- The function should return the result.

c. Call performOperation with Callback Functions:

- Call performOperation with a number (e.g., 5) and the doubleNumber callback function, and log the result to the console.
- Call performOperation with a number (e.g., 5) and the squareNumber callback function, and log the result to the console.
- Call performOperation with a number (e.g., 5) and the incrementNumber callback function, and log the result to the console.

d. Observe the Higher-Order Function:

- Observe the performOperation function to understand why higher-order functions are used.
- Higher-order functions like performOperation are useful because they can take other functions as arguments, allowing for more flexible and reusable code.

39. Basic Arithmetic Operations:

a. Define Callback Functions:

- Define a callback function named *addNumbers* that takes two numbers and returns their sum.
- Define a callback function named *multiplyNumbers* that takes two numbers and returns their product.
- Define a callback function named *subtractNumbers* that takes two numbers and returns the result of subtracting the second number from the first.

- Define a callback function named *divideNumbers* that takes two numbers and returns the result of dividing the first number by the second, ensuring to handle division by zero.

b. Define performArithmetic Function:

- Define a function named *performArithmetic* that accepts three parameters: num1 and num2 (the numbers on which to perform the operation) and operation (a callback function that specifies the arithmetic operation to be performed on num1 and num2).
- Inside performArithmetic, call the operation function and pass num1 and num2 as arguments.
- The function should return the result.

c. Call performArithmetic with Callback Functions:

- Call performArithmetic with two numbers (e.g., 5 and 3) and the addNumbers callback function. Log the result to the console.
- Call performArithmetic with two numbers (e.g., 5 and 3) and the multiplyNumbers callback function. Log the result to the console.
- Call performArithmetic with two numbers (e.g., 5 and 3) and the subtractNumbers callback function. Log the result to the console.
- Call performArithmetic with two numbers (e.g., 5 and 3) and the divideNumbers callback function. Log the result to the console.

d. Observe the Higher-Order Function:

- Observe the performArithmetic function to understand why higher-order functions are used.
- Higher-order functions like performArithmetic are useful because they can take other functions as arguments, allowing for more flexible and reusable code.

40. Calculate Painting Cost for Different Shapes

a. Define Callback Functions for Area Calculations:

- Define a callback function named *areaOfRectangle* that takes the length and width of a rectangle and returns the area.

- Define a callback function named *areaOfCircle* that takes the radius of a circle and returns the area. Use the formula π * radius^2.
- Define a callback function named *areaOfTriangle* that takes the base and height of a triangle and returns the area. Use the formula 0.5 * base * height.

b. Define calculatePaintingCost Function:

- Define a function named *calculatePaintingCost* that accepts three parameters: dimension1, dimension2, and calculateArea (a callback function that specifies the area calculation for a shape).
- Inside calculatePaintingCost, call the calculateArea function with dimension1 and dimension2 as arguments to get the area.
- Then, define a constant costPerUnit to represent the cost of painting per square unit (e.g., $2 per square unit).
- Calculate the total cost by multiplying the area by costPerUnit.
- The function should return the total cost.

c. Call calculatePaintingCost with Different Callback Functions:

- Call calculatePaintingCost with dimensions for a rectangle (e.g., length = 5, width = 10) and the areaOfRectangle callback function. Log the result to the console.
- Call calculatePaintingCost with dimensions for a circle (e.g., radius = 3) and the areaOfCircle callback function. Note: Pass null for the second dimension as it's not needed. Log the result to the console.
- Call calculatePaintingCost with dimensions for a triangle (e.g., base = 6, height = 8) and the areaOfTriangle callback function. Log the result to the console.

Objects and Classes

41. Student Data

a. Define and Populate the student Object:

- Define a variable named student and assign it an empty object.
- Add properties to the student object for name, email, and age. Set their values to your own name, email, and age.

- Log the name property of the student object to the console.

b. Update the student Object:

   - Update the age property of the student object to a random value, say 10.
   - Log the age property of the student object to the console to see the updated property.

c. Add Method and Nested Object to student:

   - Add a method called greet to the student object that logs a greeting message using the name property, e.g., "Hello, Alice!". Call the greet method to see the greeting message.
   - Add a new object called address inside the student object as its property. The address object should have properties for country, city, and pin_code. Set their values to your address details.
   - Log the country property of the address object to the console.
   - Update the pin_code property of the address object to a new pin code. Log the address object to the console to see the updated pin_code property.

d. Create and Populate the friend Object:

   - Create a new object named friend with the same properties as the student object, but with values representing a friend's name, email, age, and address. The object should also have the greet method.
   - Call the greet method and log the friend object to the console.

e. Create and Populate the topper Object:

   - Create a new object named topper with the same properties as the student object, but with values representing a topper’s name, email, age, and address. The object should also have the greet method.
   - Call the greet method and log the topper object to the console.

f. Define and Use the Student Class:

   - Define a class called Student that takes parameters for name, email, age, country, city, and pin_code.

- Inside the class's constructor, set these parameters as properties of the new object. The country, city, and pin_code properties should be added inside the address object in a nested way.
- Add the greet method inside the class.
- Add a method called *getFullAddress* to the Student class that returns the full address of the student in this format (India, Bangalore - 560038).

g. Create and Log Student Objects:

- Create objects of the Student class for yourself, your friend, and another student. Log these objects to the console to see the created objects.

h. Call the greet Method and getFullAddress Method on Student Objects:

- Call the greet method on each student object and log the result to the console.
- Call the getFullAddress method on each Student object and log the result to the console.

42. Employee Data

a. Define the Employee Class:

- Define a class named Employee that takes parameters for name, email, age, department, position, and salary.
- Inside the class's constructor, set these parameters as properties of the new object.

b. Add Methods to the Employee Class:

- Add a method named introduce inside the class that logs a greeting with the employee's name and position, e.g., "Hello, I am Sam, Software Developer".
- Add a method named displaySalary inside the class that logs the employee's salary formatted as a string, e.g., "Salary: $5000".

c. Create and Log employee Objects:

- Create an object named newEmployee with properties representing a new employee's name, email, age, department, position, and salary. Log this object to the console.

- Create another object named manager with properties representing a manager. Log this object to the console.

d. Call Methods on employee Objects:

- Call the introduce method on the newEmployee object to log a greeting with the employee's name and position.
- Call the displaySalary method on the newEmployee object to log the salary details.
- Call the introduce method on the manager object to log a greeting with the employee's name and position.
- Call the displaySalary method on the manager object to log the salary details.

43. Book Data

a. Define the Book Class:

- Define a class named Book that takes parameters for title, author, publisher, year, and genre.
- Inside the class's constructor, set these parameters as properties of the book object.

b. Add Methods to the Book Class:

- Add a method named *describe* inside the class that logs a brief description of the book, combining the title, author, and year, e.g., "Five Point Someone - Chetan Bhagat (2004)".
- Add a method named *displayGenre* inside the class that logs the genre of the book formatted as a string, e.g., "Genre: Fiction".

c. Create and Log book Objects:

- Create an object named classicBook with properties representing a classic novel's details. For example, "Pride and Prejudice" by Jane Austen, published by T. Egerton in 1813, genre "Classic".
- Create an object named sciFiBook with properties representing a science fiction book. For example, "Dune" by Frank Herbert, published by Chilton Books in 1965, genre "Science Fiction".

- Log both objects to the console.

d. Call Methods on book Objects:

- Call the describe method on the classicBook object to see a summary of the book.
- Call the displayGenre method on the classicBook object to log the genre.
- Call the describe method on the sciFiBook object to see a summary of the book.
- Call the displayGenre method on the sciFiBook object to log the genre.

Arrays

44. Colors

a. Define and Log the colors Array:

- Define a variable named colors and assign it an array containing strings representing different colors (e.g., "red", "green", "blue").
- Log the first element of the colors array to the console.

b. Modify and Add color elements:

- Change the second element of the colors array to "yellow". Log the second element of the colors array to the console to verify the change.
- Add a new color to the end of the colors array. Log the last element of the colors array to the console.

c. Iterate using Loops over the colors Array:

- Use a for loop to iterate over the colors array and log each color to the console.
- Use a while loop to iterate over the colors array and log each color to the console.
- Use a for...of loop to iterate over the colors array and log each color to the console.

d. Check Array Properties:

- Use the typeof operator to check the type of colors array and log it to the console.
- Log the length property of the colors array to the console to see the number of elements in the array.

e. Array Methods:

- Use the push method to add another color to the end of the colors array.
- Use the pop method to remove the last color from the colors array.
- Use the indexOf method to find the index of a specific color (e.g., "blue") in the colors array.

f. Add and Iterate Over Properties:

- Add a property to the colors array called owner and set its value to your name.
- Log the colors array to see if the owner property is added.
- Use a for...in loop to iterate over the properties of the colors array and log each property name and the associated value to the console.

45. Array Methods - Foreach, Map, Filter

a. Define and Initialize the Array:

- Define a variable named numbers and assign it an array containing some numbers (e.g., 1, 2, 3, 4, 5).

b. Iterate and Log Using forEach():

- Call the forEach method to iterate over the numbers array and log each number to the console. Write the callback function as a function expression.
- Call the forEach method to iterate over the numbers array and log each number multiplied by 2 to the console. Write the callback function as a function expression.

c. Iterate and Create a New Modified Array Using map():

- Call the map method to create a new array called squaredNumbers that contains the square of each number in the numbers array. Write the callback function as a function expression. Log to the console the squaredNumbers.

- Rewrite the call to the above map method to do the same thing, but change the callback function to use arrow function syntax. Store the result in the same squaredNumbers array and log it to the console.

d. Iterate and Create a New Filtered Array Using filter():

- Call the filter method to create a new array called evenNumbers that contains only the even numbers from the numbers array. Write the callback function as a function expression. Log the evenNumbers array to the console.

- Rewrite the call to the above filter method to do the same thing, but change the callback function to use arrow function syntax. Store the result in the same evenNumbers array and log it to the console.

46. Manipulating Temperatures

a. Define and Initialize the Array:

- Define a variable named temperatures and assign it an array containing several temperature readings in Celsius (e.g., -3, 14, 22, 5, -10).

b. Iterate and Log Using forEach():

- Call the forEach method to iterate over the temperatures array and log each temperature to the console. Write the callback function as a function expression.

- Rewrite the call to the above foreach method to do the same thing, but change the callback function to use arrow function syntax.

- Call the forEach method to iterate over the temperatures array and convert each temperature to Fahrenheit using the formula (temperature * 9/5) + 32. Log each converted temperature to the console. Write the callback function as a function expression.

- Rewrite the call to the above foreach method to do the same thing, but change the callback function to use arrow function syntax.

c. Iterate and Create a New Modified Array Using map():

- Call the map method to create a new array called temperaturesInFahrenheit that contains the Fahrenheit equivalent of each temperature in the temperatures array. Write the callback function as a function expression. Log the temperaturesInFahrenheit array to the console.

- Rewrite the call to the above map method to do the same thing, but change the callback function to use arrow function syntax. Store the result in the same temperaturesInFahrenheit array and log it to the console.

d. Iterate and Create a New Filtered Array Using filter():

- Call the filter method to create a new array called belowFreezing that contains only the temperatures from the temperatures array that are below 0°C. Write the callback function as a function expression. Log the belowFreezing array to the console.

- Rewrite the call to the above filter method to do the same thing, but change the callback function to use arrow function syntax. Store the result in the same belowFreezing array and log it to the console.

47. Operations on Fruits

a. Define and Initialize the Array:

- Define a variable named fruits and assign it an array containing the names of several fruits (e.g., "apple", "banana", "cherry", "date").

b. Iterate and Log Using forEach():

- Call the forEach method to iterate over the fruits array and log each fruit name capitalized to the console. Write the callback function as a function expression.

- Rewrite the call to the above foreach method to do the same thing, but change the callback function to use arrow function syntax.

c. Calculate Total Characters Using forEach():

- Initialize a variable totalCharacters to 0.

- Use forEach to add the number of characters in each fruit name and update totalCharacters. Write the callback function as a function expression.

- Log totalCharacters after the loop.

d. Iterate and Create a New Modified Array Using map():

- Use the map method to create a new array called reversedFruits that contains each fruit name reversed. Write the callback function as a function expression. Log the reversedFruits array to the console.

- Rewrite the call to the above map method to do the same thing, but change the callback function to use arrow function syntax. Store the result in the same reversedFruits array and log it to the console.

e. Iterate and Create a New Filtered Array Using filter():

- Use the filter method to create a new array called longFruits that contains only the fruit names with more than 5 characters. Write the callback function as a function expression. Log the longFruits array to the console.
- Rewrite the call to the above filter method to do the same thing, but change the callback function to use arrow function syntax. Store the result in the same longFruits array and log it to the console.

f. Create a New Filtered and Modified Array using filter() and map():

- Use the filter method to find fruits that contain the letter 'a'. Write the callback function as a function expression.
- Use the map method to return these fruits in uppercase. Write the callback function as a function expression.
- Store the result in an array called *aFruitsUpper* and log it to the console.

48. Custom Array Methods

a. forEachArray Function:

- Define a function called forEachArray that takes two arguments: an array and a callback function.
- Replicate the behavior of JavaScript's built-in forEach function.
- The forEachArray function should apply the callback function to each element of the array.

b. mapArray Function:

- Define a function called mapArray that takes two arguments: an array and a callback function.
- Replicate the behavior of JavaScript's built-in map function.
- The mapArray function should apply the callback function to each element of the array and return a new array containing the transformed elements.

c. filterArray Function:

- Define a function called filterArray that takes two arguments: an array and a callback function.
- Replicate the behavior of JavaScript's built-in filter function.
- The function should apply the callback function to each element of the array and return a new array containing only the elements for which the callback function returns true.

Functions, Objects and Arrays

49. Calculate Average Age

a. Define the Array of People:

- Define an array called people.
- Populate the array with several objects, each representing a person with properties such as name and age.

b. Define the calculateAverageAge Function Using forEach:

- Define a function called calculateAverageAge that takes one parameter: people (the array of person objects).
- Inside the function, initialize a variable to hold the sum of all ages.
- Use the forEach method to iterate over the people array and accumulate the sum of their ages.
- Calculate the average age by dividing the total sum by the number of people in the array.
- Return the average age.

c. Call the calculateAverageAge Function:

- Call the calculateAverageAge function with the people array as an argument.
- Log the result to the console.

50. Shopping Cart

a. Define the Array of Cart Items:

- Define an array called cart.
- Populate the array with several objects, each representing an item in the shopping cart with properties such as name, price, and quantity.

b. Define the calculateTotalPrice Function Using forEach:

- Define a function called calculateTotalPrice that takes one parameter: cart (the array of item objects).
- Inside the function, initialize a variable to hold the total price.
- Use the forEach method to iterate over the cart array.
- For each item, calculate the total price by multiplying the item's price and quantity, and add it to the total price variable.
- Return the total price.

c. Call the calculateTotalPrice Function:

- Call the calculateTotalPrice function with the cart array as an argument.
- Log the result to the console.

51. Manipulating Product Data

a. Define the Array of Products:

- Define a variable named products.
- Assign it an array containing several objects, each representing a product with properties such as id, name, price, and category.

b. Define the displayProducts Function:

- Create a new function called displayProducts that takes an array as an argument.
- Inside the function, use the forEach method to iterate over the array.
- Log each product in the format: ProductName - $Price (e.g., "Banana - $1.99").

c. Calculate and Display Products with Tax:

- Assume a tax rate of 10%. Use the map method to create a new array called productsWithTax that includes each product's id, name, category, and a new priceWithTax which is the original price plus the tax.
- Call the displayProducts function to display the productsWithTax array.

d. Filter and Display Food Products:

- Use the filter method to create a new array called foodProducts that contains only the products from the category "Food".
- Call the displayProducts function to display the foodProducts array.

e. Find Affordable Products:
    - Use the filter method to find products with a price under $10. Store the filtered results in a variable called affordableProducts.
    - Use the map method on affordableProducts to return a string for each product that includes the name and price (e.g., "Banana - $1.99"). Store these strings in an array called affordableProductsStrings.
    - Log the affordableProductsStrings array to the console.

f. Calculate Total Price:
    - Initialize a variable totalPrice to 0.
    - Sum up the prices of all products and update totalPrice.
    - Log totalPrice after the loop.

52. Inventory Management System

a. Define the Array of Inventory Items:
    - Define an array called inventory.
    - Populate the array with several objects, each representing a product in the store's inventory with properties such as id, name, price, quantity, and any other relevant details.

b. Create the Product Class:
    - Create a class called Product that can be used to create new product objects, which can then be added to the inventory array.
    - The class should have a constructor that initializes the properties: id, name, price, quantity, and any other relevant details.

c. Define the displayProducts Function:
    - Define a function called displayProducts.
    - Use the forEach method to iterate over the inventory array.
    - Log each product in the format: ProductName - Price (Quantity).

d. Define the addProduct Function:
    - Define a function called addProduct that takes product details as input: id, name, price, quantity.
    - Inside the function, create a new product object using the Product class.
    - Add the new product object to the inventory array.

e. Define the updateProduct Function:

- Define a function called updateProduct that takes a product id and a quantity as arguments.
- Use the find method to locate the product with the matching id in the inventory array.
- If the product is found, update its quantity.
- Define another function called updateProductWithMap, which does the same thing as updateProduct, but uses the map function instead.

f. Define the removeProduct Function:

- Define a function called removeProduct that takes a product id as an argument.
- Use the filter method to create a new array that excludes the product with the matching id.
- Update the inventory array with the new array.

53. Expense Tracker

a. Define the Array of Expenses:

- Define an array called expenses.
- Populate the array with several objects, each representing an expense with properties such as id, name, amount, date, and any other relevant details.

b. Create the Expense Class:

- Create a class called Expense that can be used to create new expense objects, which can then be added to the expenses array.
- The class should have a constructor that initializes the properties: id, name, amount, date, and any other relevant details.

c. Define the displayExpenses Function:

- Define a function called displayExpenses.
- Use the forEach method to iterate over the expenses array.
- Log each expense in the format: ExpenseName - Amount (Date).

d. Define the addExpense Function:

- Define a function called addExpense that takes expense details as input: id, name, amount, date.
- Inside the function, create a new expense object using the Expense class.
- Add the new expense object to the expenses array.

e. Define the updateExpense Function:

- Define a function called updateExpense that takes an expense id and an amount as arguments.
- Use the find method to locate the expense with the matching id in the expenses array.
- If the expense is found, update its details.
- Define another function called updateExpenseWithMap, which does the same thing as updateExpense, but uses the map function instead.

f. Define the removeExpense Function:

- Define a function called removeExpense that takes an expense id as an argument.
- Use the filter method to create a new array that excludes the expense with the matching id.
- Update the expenses array with the new array.

54. Bookstore Management System

a. Define the Array of Inventory Items:

- Define an array called inventory.
- Populate the array with several objects, each representing a book in the bookstore's inventory with properties such as id, title, author, price, quantity, and any other relevant details.

b. Create the Book Class:

- Create a class called Book that can be used to create new book objects, which can then be added to the inventory array.
- The class should have a constructor that initializes the properties: id, title, author, price, quantity, and any other relevant details.

c. Define the displayBooks Function:

- Define a function called displayBooks.
- Use the forEach method to iterate over the inventory array.
- Log each book in the format: BookTitle - Author (Price).

d. Define the addBook Function:

- Define a function called addBook that takes book details as input: id, title, author, price, quantity.
- Inside the function, create a new book object using the Book class.
- Add the new book object to the inventory array.

e. Define the updateBook Function:
    - Define a function called updateBook that takes a book id and a quantity as arguments.
    - Use the find method to locate the book with the matching id in the inventory array.
    - If the book is found, update its details.
    - Define another function called updateBookWithMap, which does the same thing as updateBook, but uses the map function instead.

f. Define the removeBook Function:
    - Define a function called removeBook that takes a book id as an argument.
    - Use the filter method to create a new array that excludes the book with the matching id.
    - Update the inventory array with the new array.

55. Todo List Application

a. Define the Array of Tasks:
    - Define an array called tasks.
    - Populate the array with several objects, each representing a task in the to-do list with properties such as id, description, dueDate, status, and any other relevant details.

b. Create the Task Class:
    - Create a class called Task that can be used to create new task objects, which can then be added to the tasks array.
    - The class should have a constructor that initializes the properties: id, description, dueDate, status, and any other relevant details.

c. Define the displayTasks Function:
    - Define a function called displayTasks.
    - Use the forEach method to iterate over the tasks array.
    - Log each task in the format: TaskDescription - DueDate (Status).

d. Define the addTask Function:
    - Define a function called addTask that takes task details as input: id, description, dueDate, status.
    - Inside the function, create a new task object using the Task class.

- Add the new task object to the tasks array.

e. Define the updateTask Function:

- Define a function called updateTask that takes a task id and a status as arguments.
- Use the find method to locate the task with the matching id in the tasks array.
- If the task is found, update its details.
- Define another function called updateTaskWithMap, which does the same thing as updateTask, but uses the map function instead.

f. Define the removeTask Function:

- Define a function called removeTask that takes a task id as an argument.
- Use the filter method to create a new array that excludes the task with the matching id.
- Update the tasks array with the new array.

DOM

56. Fetching HTML Elements

Objective: Learn selecting HTML elements using JavaScript by fetching elements based on ID, class name, tag name, and query selectors. This will help you understand how to retrieve single or multiple elements from the DOM efficiently.

a. Create the HTML Structure:

- Create a new HTML file named fetching-elements.html.
- Define the basic structure of an HTML document with <html>, <head>, and <body> tags.
- Create a separate JavaScript file named fetching-elements.js.
- Inside the <body> of your HTML, add a <script> tag and reference the JavaScript file (fetching-elements.js).

b. Fetch Elements by ID:

- Add a element in the HTML with an ID attribute set to "myDiv" and containing the text "Hello, World!".
- In JavaScript, use the function getElementById to fetch the element with the ID "myDiv".
- Log the content of the fetched element to the console using the textContent property.

c. Fetch Elements by Class:

- Add two or more elements in the HTML that share the same class "myClass" but contain different text.
- In JavaScript, use the function getElementsByClassName to fetch all elements with the class "myClass".
- Log the contents of each fetched element to the console using the textContent property.

Observation: The getElementsByClassName function returns a collection of all matching elements, allowing you to iterate over multiple elements.

d. Fetch Elements by Tag Name:

- Add two or more <p> elements in the HTML, each containing different text.
- In JavaScript, use the function getElementsByTagName to fetch all <p> elements.
- Log the contents of each fetched <p> element to the console using the textContent property.

Observation: The getElementsByTagName function returns a collection of elements based on the specified tag name, which can include multiple elements.

e. Fetch Elements Using Query Selector All:

- Add two or more elements in the HTML, each with the class "highlight" and containing different text.
- In JavaScript, use the function querySelectorAll to fetch all elements that share the class "highlight".
- Log the contents of each fetched element to the console using the textContent property.

Observation: The querySelectorAll function returns a collection of all matching elements, allowing you to fetch multiple elements similar to getElementsByClassName but with more flexible and specific queries.

f. Fetch Elements Using Query Selector:

- In JavaScript, use the function querySelector to fetch the first element with the class "highlight".
- Log the content of the fetched element to the console using the textContent property.

Observation: The querySelector function returns only the first matching element, making it useful when you need to select a specific element without fetching all matches.

57. Traversing DOM

Objective: Learn DOM traversal techniques by fetching and iterating over the child elements using methods like firstElementChild, nextElementSibling, lastElementChild, and children. This will enhance your understanding of navigating through the DOM structure efficiently.

a. Create the HTML Structure:

- Create a new HTML file named traversing-dom.html.
- Define the basic structure of an HTML document with <html>, <head>, and <body> tags.
- Create a separate JavaScript file named traversing-dom.js.
- Inside the <body> of your HTML, add a <script> tag and reference the JavaScript file (traversing-dom.js).
- Inside the <body>, create three <ul> elements, each with its own ID ("list1", "list2", and "list3"). Add multiple <li> elements inside each <ul> with different text content.

b. Fetch the First <ul> and Iterate Over Its <li> Elements

- In JavaScript, use getElementById to fetch the first <ul> element with the ID "list1".
- Use firstElementChild to fetch the first <li> element inside this <ul>.
- Using nextElementSibling, iterate over all the <li> elements and log the content of each <li> to the console using textContent.

c. Fetch the Second <ul> and Iterate Over Its <li> Elements in Reverse Order

- In JavaScript, use getElementById to fetch the second <ul> element with the ID "list2".
- Use lastElementChild to fetch the last <li> element inside this <ul>.
- Using previousElementSibling, iterate over all the <li> elements in reverse order and log the content of each <li> using textContent.

d. Fetch the Third <ul> and Access All <li> Elements

- In JavaScript, use getElementById to fetch the third <ul> element with the ID "list3".
- Use the children property to fetch all <li> elements inside the third <ul>.
- Loop through the collection of child elements and log the content of each <li> to the console using textContent.

58. Event Listeners

Objective: Learn using addEventListener in JavaScript to handle different user interactions, such as clicks, mouseovers, and input changes, by dynamically responding to events on HTML elements.

a. Create the HTML Structure:
   - Create a new HTML file named event-listeners.html.
   - Define the basic structure of an HTML document with <html>, <head>, and <body> tags.
   - Create a separate JavaScript file named event-listeners.js.
   - Inside the <body> of your HTML, add a <script> tag and reference the JavaScript file (event-listeners.js).

b. Create a Button with a Click Event Listener:
   - Inside the <body>, add a <button> element with the text "Click Me".
   - In JavaScript, use addEventListener to attach a click event listener to the button. When the button is clicked, log "Button click event executed" to the console.

c. Create a with a Mouseover Event Listener:
   - In the same HTML file, add a element with the text "Hover over me".
   - In JavaScript, use addEventListener to attach a mouseover event listener to the . When the mouse hovers over the , log "Mouseover event on div executed" to the console.

d. Create an Input Element with an Input Event Listener:
   - Add an <input> element in the same HTML file.
   - In JavaScript, use addEventListener to attach an input event listener to the input field. Whenever the input value changes, log "Input event executed" to the console.

59. Creating and Deleting HTML Elements

Objective: Understand how to manipulate the DOM by adding and removing elements programmatically on a user action handled by event listeners.

a. Create the HTML Structure:
   - Create a new HTML file named creating-deleting-elements.html.
   - Define the basic structure of an HTML document with <html>, <head>, and <body> tags.

- Create a separate JavaScript file named creating-deleting-elements.js.
- Inside the <body> of your HTML, add a <script> tag and reference the JavaScript file (creating-deleting-elements.js).
- Inside the <body>, add a <ul> element with an ID "list". Add two buttons: One with the ID "addButton" labeled "Add Item" and another with the ID "removeButton" labeled "Remove Last Item".

b. Dynamically Create and Add <li> Items to the <ul>:

- In JavaScript, create a function called addListItem that creates a new <li> element. Set its text content to "New Item X" where X is the current number of <li> items plus 1. Append the new <li> element to the <ul> with the ID "list".
- Add an event listener to the "addButton" so that each time the button is clicked, a new <li> item is added to the list.

c. Remove the Last <li> Element:

- In JavaScript, create a function called removeLastListItem that fetches the <ul> element using getElementById. Remove the last <li> element from the <ul> (if any exist) using lastElementChild and removeChild.
- Add an event listener to the "removeButton" so that each time the button is clicked, the last <li> item is removed from the list.

60. Modifying HTML Elements

Objective: Understand how to manipulate the DOM by modifying text content, styles, visibility and HTML structure programmatically on a user action handled by event listeners.

a. Create the HTML Structure:

- Create a new HTML file named modify-elements.html.
- Define the basic structure of an HTML document with <html>, <head>, and <body> tags.
- Create a separate JavaScript file named modify-elements.js.
- Inside the <body> of your HTML, add a <script> tag and reference the JavaScript file (modify-elements.js).

b. Change the Text Content of an Element on Button Click:

- Inside the <body>, add a <p> element with some initial text, e.g., "Original text", and a <button> element with the text "Change Text".

- In JavaScript, use addEventListener to attach a click event listener to the button. When the button is clicked, change the text of the <p> element to "Text has been changed!".

c. Change the Style of an Element on Mouseover:

- In the same HTML file, add a element with some text, e.g., "Hover over me".
- In JavaScript, use addEventListener to attach a mouseover event listener to the . When the mouse hovers over the , change its background color to yellow and its text color to blue.

d. Use innerHTML to Insert HTML Content Dynamically:

- Add a button with the text "Insert HTML", and an empty with an ID "htmlContent".
- In JavaScript, use addEventListener to attach a click event listener to the button. When clicked, use innerHTML to insert any combination of HTML elements (such as a heading and a paragraph) into the empty .

e. Toggle the Visibility of an Element:

- Add another <p> element and a <button> labeled "Toggle Visibility".
- In JavaScript, use addEventListener to attach a click event listener to the button. When clicked, toggle the visibility of the <p> element by switching between display: block and display: none.

61. Challenges - DOM

a. Handle Form Submission with preventDefault()

- Create an HTML file containing a form with an input field and a submit button.
- When the form is submitted, prevent the default form submission behavior and log "Form submitted!" to the console.
- Hint: Use preventDefault() to stop the form from submitting.

b. Toggle Background Color of Elements

- Create an HTML file with multiple elements, each containing different text.

- When a is clicked, its background color should toggle between red and white.
- The behavior should only affect the clicked and not others.

c. Change Background Color Using Buttons

- Create a element with an initial background color of white and four buttons, each labeled with a color (red, yellow, blue, and green).
- When a button is clicked, the background color of the should change to match the color of the button.

API's

62. Practicing API Requests with Postman

Objective: Understand how to interact with APIs by making GET and POST requests in Postman, analyzing response status codes, headers, and bodies, and exploring additional HTTP methods to manipulate resources programmatically.

About Postman: Postman is a tool used to test and interact with APIs. It allows you to send HTTP requests and view responses, helping you analyze and understand how APIs work. You can perform various request methods such as GET (retrieve data) and POST (send data). It also allows you to inspect the response status code, headers, and body, giving you a complete view of your API interaction.

a. Make a GET Request to Fetch All Posts:

- Open Postman and create a GET request to https://jsonplaceholder.typicode.com/posts
- Check the response status code. It should be 200 OK, indicating a successful request.
- Inspect the Response Headers for the Content-Type. It should be application/json; charset=utf-8, indicating the response data is in JSON format.
- Analyze the Response Body, which should contain an array of post objects. Each object includes properties like userId, id, title, and body. This response returns multiple posts as an array of objects, with each object representing one post.

b. Make a GET Request to Fetch a Specific Post

- In Postman, create another GET request to fetch a single post using the endpoint: https://jsonplaceholder.typicode.com/posts/1.
- Analyze the Response Body, which now returns a single post object instead of an array.
- In the previous request, you retrieved an array of post objects. In this request, you receive only one object representing a single post.

c. Make a POST Request to Create a New Post:

- In Postman, create a new request and select POST as the method.
- Use the endpoint https://jsonplaceholder.typicode.com/posts to send the request.
- POST requests allow you to send data to the server, unlike GET requests, which only retrieve data. In a POST request, you include data in the request body.
- In the Body tab of Postman, select raw and choose JSON format. Add the following data in the request body:

```
{
  "title": "My New Post",
  "body": "This is the content of my new post.",
  "userId": 1
}
```

- Analyze the Response Body. The API should respond with the newly created post object, which includes a new id generated by the server.

d. Explore More Requests Using the JSONPlaceholder Documentation:

- Visit the JSONPlaceholder API Documentation.
- Read through the available endpoints and explore other HTTP methods (such as PUT, PATCH, and DELETE) for different resources like posts, users, or comments.
- Try making additional requests in Postman, such as updating a post or deleting a resource.
- Analyze the responses for these new requests, and observe how they differ from GET and POST requests.

63. Fetch and Display Posts Using API

Objective: Understand how to fetch and display data from an API by making GET requests, manipulating the DOM to dynamically generate and display content, and handling errors programmatically using JavaScript.

a. Create the HTML and JavaScript Structure:

- Create a new HTML file named fetch-posts.html.
- Define the basic structure of an HTML document with <html>, <head>, and <body> tags.
- Create a separate JavaScript file named fetch-posts.js.
- Inside the <body> of your HTML, add a <script> tag and reference the JavaScript file.
- Inside the <body>, add an empty element with an ID of "posts" where all the fetched posts will be displayed.
- This will serve as the container for all the posts that will be dynamically generated through JavaScript.

b. Fetch All Posts from the JSONPlaceholder API:

- In the JavaScript file, use the fetch() function to make a GET request to the JSONPlaceholder API endpoint: https://jsonplaceholder.typicode.com/posts.
- Once the data is fetched, convert the response to JSON format using .json() and store the result.

c. Display the Posts on the Webpage:

- For each post fetched from the API, create a new element in JavaScript.
- Inside each , display the post's details: userId, id, title, and body.
- Append each to the container in the HTML.

d. Handle Errors:

- Implement error handling for the API request using .catch() to log any potential errors (e.g., network issues).

64. Fetch and Display Posts and Comments by User ID

Objective: Understand how to fetch and display related data from an API by capturing user input, making dynamic API requests, and organizing the display of posts and associated comments programmatically using JavaScript event listeners.

a. Create the HTML and JavaScript Structure:

- Create a new HTML file named user-posts-comments.html.
- Define the basic structure of an HTML document with <html>, <head>, and <body> tags.
- Create a separate JavaScript file named user-posts-comments.js and reference the JavaScript file in the HTML using a <script> tag at the end of the <body>.
- Inside the <body> of your HTML, add an input field and a button labeled "Fetch Posts" to allow the user to input the User ID.
- Create an empty with an ID "posts" where all the fetched posts and comments will be displayed.

b. Fetch and Display Posts and Comments:

- In the JavaScript file, add an event listener to the button that triggers when clicked.
- Capture the User ID from the input field.
- Use the User ID to make a GET request to fetch all posts from the endpoint https://jsonplaceholder.typicode.com/posts?userId=<inputUserId>.
- For each post, make another GET request to fetch the comments using the endpoint https://jsonplaceholder.typicode.com/comments?postId=<postId>.
- Display the posts and comments in an organized manner, ensuring each post is listed with its associated comments.

65. Challenge - Dictionary App

Problem Statement: Build a dictionary app that allows users to enter a word and fetch its meaning using the Dictionary API. You will make an API call to the given API using the fetch() method to retrieve the word's definitions and display them on the screen in a well-structured format.

API to use: https://dictionaryapi.dev/

What Needs to Be Done:

a. Create an input field where the user can enter a word.

b. When the user submits the word, make an API call to the Dictionary API using fetch().

c. Display the word's meanings and sub-meanings in a structured format, including:

- Definitions
- Parts of speech
- Example sentences (if available)
- Synonyms and antonyms (if available)

d. Make sure the UI is clean and readable.

e. The API response contains rich data about the word's meanings. It's essential to carefully understand and structure the response:

- The meanings are nested inside an array, with each meaning associated with a part of speech (e.g., noun, verb).
- Each meaning contains a list of definitions, which may also include examples, synonyms, and antonyms.
- The response might contain multiple entries for the same word if it has different usages (like a noun vs a verb). You should loop through these entries and display all the relevant information.

f. Handle errors gracefully, such as displaying a message if the word is not found or if there is an issue with the API.

Projects

66. Project: Task Management App

Problem Statement: Create a Task Management App that allows users to efficiently manage their tasks visually on the screen. The app will allow users to add, edit, delete, and mark tasks as completed, all of which will be reflected in real-time on the user interface. Additionally, users will be able to filter tasks by priority, due date, or completion status, with the option to persist tasks even after page reloads.

What Needs to Be Done:

a. User Task Input:

- Create a form where users can enter a new task with the following details: Task title, Description of the task, Due date for the task, Priority level (e.g., low, medium, high)
- Once the user submits the form, the task should appear on the screen below the form.

b. Displaying Tasks:

- Every task that is added should be displayed with all the information the user entered (title, description, due date, priority).
- Each task should have three action buttons: Edit: Allows the user to modify the task. Delete: Removes the task from the display. Mark as Completed: Moves the task to a separate section for completed tasks.

c. Managing Completed Tasks:

- When a user marks a task as completed, it should disappear from the pending tasks list and move to a Completed Tasks section below.
- The completed tasks should still display all their details but should be visually distinct from pending tasks (e.g., different color or strikethrough text).

d. Filter Tasks:

- Add the ability to filter tasks based on: Priority (e.g., view only high-priority tasks), Due Date (e.g., view tasks due within the next 7 days), Status (pending or completed tasks).
- When a filter is applied, only the relevant tasks should be shown on the screen.

e. Task Persistence (Optional):

- Ensure tasks are saved even after the user refreshes the page. (Hint: You may want to use Local Storage for this.)

67. Project: Weather Dashboard

Problem Statement: Create a Weather Dashboard where users can search for a city and view the current weather along with a 5-day forecast. The dashboard should display weather details such

as temperature, humidity, wind speed, and weather descriptions in an organized and user-friendly manner, providing real-time updates for multiple city searches.

API to use: https://openweathermap.org/api

What Needs to Be Done:

a. User Input for City Search:

- Create a form where users can enter the name of a city to retrieve its weather information.
- Once the user submits the city name, the current weather conditions and a 5-day forecast for that city should be displayed on the screen.

b. Displaying Current Weather:

- The current weather should be displayed with the following details: City name and date, Temperature, Weather description (e.g., cloudy, sunny), Humidity percentage, Wind speed, Weather icon representing current conditions (optional)

c. Displaying 5-Day Forecast:

- Below the current weather, display the 5-day forecast for the searched city.
- For each day in the forecast, show: Date, Expected temperature, Weather description (e.g., rainy, clear), A small weather icon (optional)

d. Error Handling:

- If the city is not found or the user enters an invalid input, display an appropriate error message on the screen.

Backend Tasks

Built-in Node Modules

68. File System

a. **Create a file:** Write a script that creates a new file named 'message.txt' and writes some text into it.

b. **Read from file:** Write a script that reads the content of 'message.txt' and logs it to the console.

c. **Append to a file:** Write a script that appends some text to 'message.txt'.

d. **Delete a file:** Write a script that deletes 'message.txt'.

69. HTTP Server

a. **Create a Basic Server:** Create an HTTP server. The server should respond with some text when accessed at the root URL (/).

b. **Handle Different Routes:** Update the server to handle two additional routes: '/about' and '/contact'. These routes should respond with some text.

70. Path Module

a. **Parse a file path:** Write a script that parses the path of 'message.txt' (from 1) and logs the directory, base, extension, and name.

b. **Join paths:** Write a script that joins the current directory with public and index.html to form a complete path. Log this path to the console.

71. Combined Task

a. Create a public directory with an index.html file. Update the server to serve this file when the root URL is accessed.

Installing npm Packages and Basic Functionalities

72. Npm Packages

a. Use the inquirer npm package to get user input.

b. Use the qr-image npm package to turn the user entered URL into a QR code image.

c. Create a txt file to save the user input using the native fs node module

Callbacks, Promises, and Async/Await

73. Callbacks, Promises, and Async/Await

a. Write a script that reads the content of a file named “data.txt”. The content of the file should be logged into the console. Handle any errors that may occur.

- Using Callbacks.

- Using Promises.
- Using Async/Await.

Express.js: Use EJS and Render in DOM

74. Express.js: Use EJS and Render in DOM

a. Install Express and EJS packages.

b. Write a script that sets up a basic Express server.

c. Create routes for the following:

- Root URL (/): Render an index page.
- About URL (/about): Render an about page.
- Contact URL (/contact): Render an contact page.

d. Create EJS templates for the above pages.

Establish Database connection (MongoDB)

75. Database Users

a. Create a new Node.js project and install the required dependencies.

b. Install the MongoDB package.

c. Create a database named 'test.db' .

d. Create a collection named 'users' with fields 'name' and 'email'.

e. Write a script that connects to the 'test.db' database.

f. Perform the following CRUD operations:

- Create: Insert a new document into the 'users' collection.
- Read: Retrieve all documents from the 'users' collection.
- Update: Update a user's email in the 'users' collection.
- Delete: Delete a document from the 'users' collection.

76. Database E-commerce

a. Create a new Node.js project and install the required dependencies.

b. Install the MongoDB package.

c. Create a database named 'ecommerce'.

d. Create a collection named 'catalog' with a nested structure as described here: https://cdn.shopify.com/s/files/1/0564/3685/0790/files/multiProduct.json

e. Write a script that connects to the 'ecommerce' database.

f. Perform the following CRUD operations:

- Create: Insert a new product into an existing category.
- Read: Retrieve all products from a specific category.
- Update: Write a script to update the price of a specific product.
- Delete: Write a script to delete a product from a specific category.

Sending Data Requests

77. Data Requests

a. Create a folder named 'express_request'.

b. Create a new Node.js project and install the required dependencies.

c. Create a basic Express server.

d. Use the Express body-parser middleware to handle JSON and URL-encoded data.

- Route with URL Params: Create a route that handles GET requests with URL parameters. Respond with the parameters received.
- Route with Query Params: Create a route that handles GET requests with query parameters. Respond with the query parameters received.
- Route with Body Data (JSON): Create a route that handles POST requests with JSON data in the body. Respond with the data received.

Authentication and Authorization

78. Authentication and Authorization

a. Create a new Node.js project and install the required dependencies.

b. Install Express, bcryptjs, and jsonwebtoken packages.

c. Write a script that sets up a basic Express server.

d. Use middleware to parse JSON data.

e. User Registration:

   - Create a route to handle user registration.
   - Hash the user's password before saving it to a mock database (in-memory or file-based).

f. User Login:

   - Create a route to handle user login.
   - Verify the user's password and generate a JWT upon successful authentication.

g. Protected Route:

   - Create a protected route that only allows access to authenticated users.
   - Verify the JWT in the request headers before allowing access to the protected route.