# Module 8 : Bootstrap Basic & Advanced

1. **What are the advantages of Bootstrap?**

**Ans :-**  Bootstrap is a web development framework that has many advantages, including.

1. Responsive Design: Bootstrap enables developers to create mobile-friendly and responsive websites that adapt to various screen sizes and devices.

2. Fast Development: Bootstrap provides pre-built CSS and HTML templates, reducing development time and effort.

3. Consistency: Bootstrap ensures consistency in design and layout across different browsers and devices.

4. Easy Maintenance: Bootstrap's modular structure makes it easy to update and maintain websites.

5. Large Community: Bootstrap has an extensive community of developers, ensuring extensive support and resources.

6. Customizable: Bootstrap allows for easy customization to fit specific project needs.

7. Cross-Browser Compatibility: Bootstrap ensures compatibility with various browsers, reducing compatibility issues.

8. Extensive Documentation: Bootstrap offers comprehensive documentation, making it easier for developers to learn and use.

9. Pre-built Components: Bootstrap includes pre-built UI components, such as navigation bars, alerts, and modals, saving development time.

10. Free and Open-Source: Bootstrap is free to use and distribute, with an open-source license.

1. **What is a Bootstrap Container, and how does it work?**

**Ans :-** In Bootstrap, a container is a fundamental element that wraps the content of a webpage, providing a fixed width and padding. It's used to create a responsive and consistent layout.

**Grid System:** The container is the foundation for Bootstrap's grid system, allowing for flexible and responsive column layouts.

**Responsive:** The container's width adjusts automatically based on the screen size, ensuring a responsive design.

**Content Wrapper:** The container wraps the content, keeping it organized and contained within the specified width.

**Padding:** The container adds padding to the content, creating a buffer zone between the content and the edge of the screen.

**Fixed Width:** The container has a fixed width, which varies based on the screen size (e.g., 1200px for large screens, 960px for medium screens, etc.)

There are two types of containers in Bootstrap:-

**- \*.container\*:** A fixed-width container that changes width at specific breakpoints.

**- \*.container-fluid\*:** A full-width container that takes up the entire width of the screen.

1. **What are the default Bootstrap text settings?**

**Ans :-** Bootstrap sets default text settings to provide a consistent and readable typography.

1. Font Weight: 400 (normal) for the base font weight.

2. Line Height: 1.5, which translates to 24px (1.5rem) for the base line height.

3. Color: #212529, a dark gray color for the base text color.

4. Font Family: Open Sans, a clean and modern sans-serif font.

5. Font Size: 16px (1rem) for the base font size.

Additionally, Bootstrap sets default settings for headings (h1-h6), which have:

- Larger font sizes (from 36px to 24px)

- Bold font weights (700)

- Margin bottom for spacing

These default text settings can be overridden using Bootstrap's utility classes or custom CSS to fit your specific design needs.

1. **What do you know about the Bootstrap Grid System?**

**Ans :-** The Bootstrap Grid System is a powerful and flexible layout framework that allows developers to create responsive and adaptive layouts.

**Containers**: Define the fixed-width or full-width layout of your content. Use .container for fixed-width or .container-fluid for full-width layouts.

**Rows**: Organize columns within a container. Each row must be placed inside a container. Rows use negative margins to align columns correctly.

**Columns**: Define the content's width and layout within a row. Columns are created using classes like .col, .col-md-6, etc., where md specifies the breakpoint and 6 is the number of grid columns (out of 12).

**Breakpoints**: Bootstrap's grid system is responsive and includes five breakpoints: xs (extra small), sm (small), md (medium), lg (large), and xl (extra large).

**Auto Layout**: Columns can adjust their size automatically to fill available space, or you can set specific column widths using classes like .col-auto, .col-3, etc.

**Nesting**: You can nest rows and columns inside other columns to create more complex layouts.

**Offsetting**: Use classes like .offset-md-3 to create space between columns by offsetting them from the start of the row.

1. **What is the difference between Bootstrap 4 and Bootstrap 5.**

**Ans :-** Bootstrap 5 introduced several updates and changes compared to Bootstrap 4.

**New Components**: Bootstrap 5 adds new components such as the offcanvas component and updates existing ones for enhanced functionality and better design.

**Icons**: Bootstrap 5 no longer includes Glyphicons by default. Instead, it recommends using external icon libraries like Font Awesome or Bootstrap Icons.

**Enhanced Customization**: Bootstrap 5 offers improved theming capabilities with CSS variables, allowing for more straightforward customization of styles without needing to override CSS.

**Dropped Internet Explorer 10 and 11 Support**: Bootstrap 5 no longer supports Internet Explorer 10 and 11, focusing instead on modern browsers.

**Utility API**: Bootstrap 5 introduces a utility API for customizing and generating utilities, making it easier to create and manage custom utility classes.

**Updated Grid System**: Bootstrap 5 includes a new .g- class for gutters, allowing for more flexible spacing adjustments. It also introduces the row-cols-\* classes for more responsive column layouts.

**JQuery Dependency**: Bootstrap 5 has removed the requirement for jQuery. It now relies solely on vanilla JavaScript, which can lead to better performance and smaller file sizes.

1. **What is a Button Group, and what is the class for a basic Button Group?**

**Ans :-** In Bootstrap, a Button Group is a component that groups multiple buttons together, making it easier to display related actions or options. It's a container element that wraps multiple buttons, allowing them to be displayed together and share a common styling.

The class for a basic Button Group is .btn-group.

**Example:**

<div class="btn-group">

<button class="btn btn-primary">Button 1</button>

<button class="btn btn-primary">Button 2</button>

<button class="btn btn-primary">Button 3</button>

</div>

**Features :**

**Alignment**: Buttons within the group will align horizontally by default.

**Button Variants**: You can use any of Bootstrap’s button classes (e.g., .btn-primary, .btn-secondary) within the group.

**Additional Styles**: You can add classes like .btn-group-vertical to stack the buttons vertically.

1. **How can you use Bootstrap to make thumbnails?**

**Ans :-** To create thumbnails using Bootstrap, you can use the .img-thumbnail class. This class adds a simple border and padding to an image, making it look like a thumbnail.

**Basic Thumbnail Example :**

<img src="path/to/image.jpg" class="img-thumbnail" alt="Thumbnail">

**Thumbnail Grid :**

You can also create a grid of thumbnails using Bootstrap’s grid system:

<div class="container">

<div class="row">

<div class="col-md-4">

<img src="path/to/image1.jpg" class="img-thumbnail" alt="Thumbnail 1">

</div>

<div class="col-md-4">

<img src="path/to/image2.jpg" class="img-thumbnail" alt="Thumbnail 2">

</div>

<div class="col-md-4">

<img src="path/to/image3.jpg" class="img-thumbnail" alt="Thumbnail 3">

</div>

</div>

</div>

**Features :**

**Border and Padding**: .img-thumbnail adds a border and padding around the image.

**Responsive**: Thumbnails are responsive and will adjust their size based on the viewport.

1. **In Bootstrap 4, what is flexbox?**

**Ans :-**  In Bootstrap 4, Flexbox is a layout model that provides a more efficient way to distribute space and align items within a container. It simplifies the creation of complex layouts and responsive designs.

Bootstrap 4 uses Flexbox to create a flexible grid system, navigation, and other components.

**1. Create flexible containers:** That can expand or shrink to fit their content.

**2. Align items horizontally or vertically:** With ease, using justify-content and align-items properties.

**3. Distribute space between items:** Using the flex-grow, flex-shrink, and flex-basis properties.

**4. Create responsive layouts:** That adapt to different screen sizes and devices.

Common Bootstrap 4 Flexbox classes include:

**.d-flex:** Enables Flexbox layout for a container.

**.flex-row:** Creates a horizontal Flexbox layout (default).

**.flex-column:** Creates a vertical Flexbox layout.

**.justify-content-\*:** Aligns items horizontally (e.g., .justify-content-center).

**.align-items-\*:** Aligns items vertically (e.g., .align-items-center).

**.flex-wrap:** Wraps items to a new line if there's not enough space.

**.flex-grow-\*, .flex-shrink-\*, and .flex-basis-\*:** Controls item flexibility and size.

1. **How can one create an alert in Bootstrap?**

**Ans :-** In Bootstrap, you can create an alert using the .alert class, along with optional modifier classes to customize its appearance and behavior.

**Example :**

<div class="alert alert-primary" role="alert">

This is a primary alert!

</div>

**Alert Contextual Classes**

Bootstrap provides several contextual classes to denote different types of alerts:

.alert-primary for primary alerts

.alert-secondary for secondary alerts

.alert-success for success messages

.alert-danger for error messages

.alert-warning for warnings

.alert-info for informational messages

.alert-light for light alerts

.alert-dark for dark alerts

**Dismissible Alerts**

To create an alert that can be dismissed, add the .alert-dismissible class and include a close button:

**Example :**

<div class="alert alert-warning alert-dismissible fade show" role="alert">

This is a warning alert—check it out!

<button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Close"></button>

</div>

1. **What is a bootstrap card and how would you create one?**

**Ans :-** A card in Bootstrap 5 is a bordered box with some padding around its content. It includes options for headers, footers, content, colors, etc.

To create a basic Bootstrap card, you can use the following HTML structure:

<div class="card">

<img class="card-img-top" src="image.jpg" alt="Card image cap">

<div class="card-body">

<h5 class="card-title">Card title</h5>

<p class="card-text">Some quick example text to build on the card title and make up the bulk of the card's content.</p>

<a href="#" class="btn btn-primary">Go somewhere</a>

</div>

</div>

This code creates a basic card with:

- An image at the top (.card-img-top)

- A body section (.card-body) containing:

- A title (.card-title)

- Text content (.card-text)

- A button (.btn)

You can customize the card further by adding additional classes, such as:

- .card-header for a header section

- .card-footer for a footer section

- .card-border to add a border

- .card-inline to make the card display inline

- And many more!

# Advance JavaScript for Front-End Introduction and Code Quality

1. **Write a program to Show an alert.**

**Ans :-**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Alert Example</title>

<link rel="stylesheet" href="(link unavailable)">

</head>

<body>

<div class="container">

<div class="alert alert-primary" role="alert">

This is a primary alert!

</div>

<div class="alert alert-secondary" role="alert">

This is a secondary alert!

</div>

<div class="alert alert-success" role="alert">

This is a success alert!

</div>

<div class="alert alert-danger" role="alert">

This is a danger alert!

</div>

<div class="alert alert-warning" role="alert">

This is a warning alert!

</div>

<div class="alert alert-info" role="alert">

This is an info alert!

</div>

</div>

</body>

</html>

This program uses Bootstrap's alert classes to show different types of alerts. You can customize the alerts by using different classes, such as .alert-primary, .alert-secondary, .alert-success, .alert-danger, .alert-warning, and .alert-info.

1. **What will be the result for these expressions?**
2. **5 > 4**

**Ans :** The result of the expression "5 > 4" is **True** . Since 5 is indeed greater than 4.

1. **"apple" > "pineapple"**

**Ans :** The result of the expression ""apple" > "pineapple"" is **False .**

This is because the expression is comparing strings (text), and in alphabetical order, "apple" comes before "pineapple". In Python, when comparing strings, it checks the Unicode value of each character from left to right. Since 'a' comes before 'p', "apple" is considered less than "pineapple".

1. **"2" > "12"**

**Ans :** The result of the expression ""2" > "12"" is **True .**

This might seem counterintuitive, but when comparing strings, the comparison is done alphabetically (or more precisely, by Unicode value), not numerically. Since the string "2" has a higher Unicode value than the string "1" (the first character of "12"), the expression evaluates to True. If you wanted to compare numbers, you would need to remove the quotes, like this: 2 > 12, which would evaluate to False.

1. **undefined == null**

**Ans :** The result of the expression "undefined == null" is **True** in JavaScript.

In JavaScript, both undefined and null represent the absence of a value, and they are considered equal when compared using the loose equality operator (==). However, when using the strict equality operator (===), they are not considered equal, because undefined and null have different data types.

1. **undefined === null**

**Ans :** The result of the expression "undefined === null" is **False** in JavaScript.

In JavaScript, the strict equality operator (===) checks both the value and the data type of the operands. Since undefined and null have different data types (undefined is an undefined type, and null is an object type), the expression evaluates to False.

This is in contrast to the loose equality operator (==), which would return True for the same comparison, as it only checks the value, not the data type.

1. **null == "\n0\n"**

**Ans :** The result of the expression "null == "\n0\n"" is **False .**

In JavaScript, the loose equality operator (==) compares the values of the operands after performing type conversions if necessary. However, null is only considered equal to undefined, and not to any other value, including strings like "\n0\n".

So, even though "\n0\n" might be considered "falsy" in a boolean context (due to the leading and trailing whitespace), it's not considered equal to null. If you want to check for "falsy" values, you should use a boolean context or an explicit comparison, like !"\n0\n" or "\n0\n" === "".

1. **null === +"\n0\n"**

**Ans :** The result of the expression "null === +"\n0\n"" is **False.**

- The unary plus operator (+) is used to convert the string "\n0\n" to a number.

- The conversion results in 0, because the string is trimmed and the remaining 0 is converted to a number.

- Then, the strict equality operator (===) compares null to 0.

- Since null and 0 are of different types (null and number, respectively), the expression evaluates to False.

that if you used the loose equality operator (==) instead, the result would be True, because null is considered equal to 0 when using loose equality in JavaScript. However, the strict equality operator (===) checks both value and type, so it returns False.

1. **Will alert be shown ?**

**if ("0") { alert( 'Hello'); }**

**Ans :** Yes, the alert will be shown.

In JavaScript, when evaluating a condition inside an if statement, any non-empty string (including "0") is considered a truthy value. Therefore, since "0" is truthy, the condition evaluates to true, and the code inside the if block is executed. As a result, the alert with the message 'Hello' will be displayed.

1. **What is the code below going to output? alert( null || 2 || undefined );**

**Ans :** The code alert(null || 2 || undefined); will output 2.

- The || operator is the logical OR operator, which returns the first truthy value it encounters.

- null is a falsy value, so the operator moves on to the next value.

- 2 is a truthy value, so the operator returns 2 without evaluating the next value (undefined).

- Therefore, the alert will show 2.

Since null is falsy, the || operator will move to the next value, which is 2, and since 2 is truthy, it will be the result of the expression. Thus, 2 is the value that gets passed to the alert function, causing 2 to be displayed.Top of Form

1. **The following function returns true if the parameter age is greater than 18. Otherwise it asks for a confirmation and returns its result:**

**function**

**checkAge(age)**

**{**

**if (age> 18) { return true; }**

**else {**

**//...return confirm (‘did parents allow you?');**

**}**

**}**

**Ans :** You're close! Here's the corrected and completed function:

function checkAge(age) {

if (age > 18) {

return true;

} else {

return confirm('Did parents allow you?');

}

}

You had the if statement outside the function, but it should be inside. Also, I added the missing parentheses around the condition age > 18. Now the function should work as intended!

1. **Replace Function Expressions with arrow functions in the code below: Function ask(question, yes, no)**

**{ if (confirm(question))yes();**

**else**

**no();**

**}**

**ask("Do you agree?", function()**

**{ alert("You agreed."); },**

**function() {**

**alert("You canceled the execution."); }**

**}**

**Ans :** To replace the function expressions with arrow functions in the provided code, you need to modify the ask function and the callback functions.

**Change the ask function to use an arrow function:**

* The ask function itself remains a regular function declaration, but you can use arrow functions for the callbacks.

**Update the callback functions to use arrow functions:**

function ask(question, yes, no) {

if (confirm(question)) yes();

else no();

}

ask(

"Do you agree?",

() => { alert("You agreed."); },

() => { alert("You canceled the execution."); }

);

# Data Types and Objects

1. **Write the code, one line for each action:**

**a) Create an empty object user.**

**b) Add the property name with the value John.**

**c) Add the property surname with the value Smith.**

**d) Change the value of the name to Pete.**

**e) Remove the property name from the object.**

**Ans :** Here are the one-liners:

// a) Create an empty object user

**let user = {};**

// b) Add the property name with the value john

**user.name = 'john';**

// c) Add the property surname with the value smith

**user.surname = 'smith';**

// d) Change the value of the name to pete

**user.name = 'pete';**

// e) Remove the property name from the object

**delete user.name;**

a) let user = {}; initializes an empty object.

b) user.name = 'john'; adds a property name with the value 'john'.

c) user.surname = 'smith'; adds a property surname with the value 'smith'.

d) user.name = 'pete'; changes the value of the name property to 'pete'.

e) delete user.name; removes the name property from the object.

1. **Is array copied?**

**let fruits = ["Apples", "Pear", "Orange"]; // push a new value into the "copy" let shoppingCart = fruits; shoppingCart.push("Banana"); // what's in fruits?**

**alert( fruits.length ); // ?**

**Ans :** No, the array is not copied. Instead, shoppingCart becomes a reference to the original fruits array.

So, when you push a new value into shoppingCart, you're actually modifying the original fruits array.

Therefore, alert( fruits.length ); will show 4, because the fruits array now contains:

1. "Apples"

2. "Pear"

3. "Orange"

4. "Banana"

If you want to create a copy of the array, you can use the spread operator ([...fruits]) or the slice() method (fruits.slice()), like this:

let shoppingCart = [...fruits]; or let shoppingCart = fruits.slice();

Then, modifying shoppingCart won't affect the original fruits array.

1. **Map to names**

**let john = { name: "John", age: 25 }; let pete = { name: "Pete", age: 30 }; let mary = { name: "Mary", age: 28 }; let users = [ john, pete, mary ]; let names = /\* ... your code \*/ alert( names ); // John, Pete, Mary**

**Ans :** To map the users array to an array of names, you can use the map() method:

let names = users.map(user => user.name);

This will create a new array names containing only the name properties of each user object.

So, alert(names) will show: John, Pete, Mary

**Example :**

let john = { name: "John", age: 25 };

let pete = { name: "Pete", age: 30 };

let mary = { name: "Mary", age: 28 };

let users = [ john, pete, mary ];

let names = users.map(user => user.name);

alert(names); // John, Pete, Mary

1. **Map to objects**

**let john = { name: "John", surname: "Smith", id: 1 }; let pete = { name: "Pete", surname: "Hunt", id: 2 }; let mary = { name: "Mary", surname: "Key", id: 3 }; let users = [ john, pete, mary ]; let usersMapped = /\* ... your code ... \*/ /\***

**usersMapped = [**

**{ fullName: "John Smith", id: 1 },**

**{ fullName: "Pete Hunt", id: 2 },**

**{ fullName: "Mary Key", id: 3 }**

**]**

**\*/ alert( usersMapped[0].id ) // 1 alert( usersMapped[0].fullName ) // John Smith**

**Ans :** To map the users array to a new array of objects where each object contains fullName and id properties, you can use the map method.

**Example :**

let john = { name: "John", surname: "Smith", id: 1 };

let pete = { name: "Pete", surname: "Hunt", id: 2 };

let mary = { name: "Mary", surname: "Key", id: 3 };

let users = [ john, pete, mary ];

let usersMapped = users.map(user => ({

fullName: `${user.name} ${user.surname}`,

id: user.id

}));

alert(usersMapped[0].id); // 1

alert(usersMapped[0].fullName); // John Smith

1. **Sum the properties There is a salaries object with arbitrary number of salaries. Write the function sumSalaries(salaries) that returns the sum of all salaries using Object.values and the for..of loop.If salaries is empty, then the result must be 0.**

**let salaries = {**

**"John": 100,**

**"Pete": 300,**

**"Mary": 250**

**};**

**alert( sumSalaries(salaries) ); // 650**

**Ans :** To calculate the sum of all salaries from the salaries object using Object.values and a for..of loop, follow these steps:

1. Use Object.values(salaries) to get an array of the salary values.
2. Iterate over this array using a for..of loop.
3. Accumulate the sum of these values.
4. Return the total sum.

**the function that accomplishes this:**

function sumSalaries(salaries) {

let total = 0;

for (let salary of Object.values(salaries)) {

total += salary;

}

return total;

}

let salaries = {

"John": 100,

"Pete": 300,

"Mary": 250

};

alert(sumSalaries(salaries)); // 650

1. **Destructuring assignment We have an object: Write the Destructuring assignment that reads:**

**a) Name property into the variable name.**

**b) Year’s property into the variable age.**

**c) isAdmin property into the variable isAdmin (false, if no such property)**

**d) let user = { name: "John", years: 30};**

**Ans :** To use destructuring assignment for the given user object and extract the properties into variables, you can do the following:

1. Extract name into the variable name.
2. Extract years into the variable age.
3. Extract isAdmin into the variable isAdmin with a default value of false if the property does not exist.

**Example :**

let user = { name: "John", years: 30 };

let { name, years: age, isAdmin = false } = user;

console.log(name); // "John"

console.log(age); // 30

console.log(isAdmin); // false

* name will be "John".
* age will be 30.
* isAdmin will be false because the isAdmin property is not present in the user object.

1. **Turn the object into JSON and back Turn the user into JSON and then read it back into another variable.**

**user = { name: "John Smith", age: 35};**

**Ans :** how you can turn the user object into JSON and then read it back into another variable:

1.Convert the object into a JSON string using JSON.stringify().

2.Parse the JSON string back into an object using JSON.parse().

**how you can do it:**

// Original object

let user = { name: "John Smith", age: 35 };

// Convert the object to a JSON string

let jsonString = JSON.stringify(user);

// Parse the JSON string back into an object

let parsedUser = JSON.parse(jsonString);

console.log(parsedUser); // { name: "John Smith", age: 35 }

- JSON.stringify(user) converts the user object into a JSON string representation. This string can be stored or transmitted as needed.

- JSON.parse(jsonString) converts the JSON string back into a JavaScript object.

# New Request

* 1. **What is JSON .**

**Ans :** JSON JavaScript Object Notation is a lightweight, text-based data interchange format that is easy to read and write. It is a standard format for exchanging data between web servers, web applications, and mobile apps.

JSON represents data as key-value pairs, arrays, and objects, making it a versatile and widely-used format for data exchange.

**Example :**

{

"name": "John Doe",

"age": 30,

" occupation": "Developer",

"address": {

"street": "123 Main St",

"city": "Anytown",

"state": "CA",

"zip": "12345"

},

"interests": ["reading", "hiking", "coding"]

}

JSON is language-independent, meaning it can be used with many programming languages, including JavaScript, Python, Java, and more.

**Some key features of JSON include:**

**1. Human-readable:** JSON data is easy to read and understand.

**2. Platform-independent:** JSON can be used on any platform, device, or language.

**3. Lightweight:** JSON data is compact and efficient.

**4. Flexible:** JSON supports various data types, including strings, numbers, booleans, arrays, and objects.

**Common uses of JSON include:**

**1. Data exchange:** JSON is often used to exchange data between web servers, web applications, and mobile apps.

**2. Config files:** JSON is used in configuration files for many applications and frameworks.

**3. Data storage:** JSON can be used to store data in NoSQL databases, like MongoDB.

**4. API responses:** JSON is often used to format data returned from web APIs.

* 1. **What is promises .**

**Ans :** Promises are a way to handle asynchronous operations in JavaScript. They provide a way to write asynchronous code that is easier to read and maintain.

A promise represents a value that may not be available yet, but will be resolved at some point in the future.

**When a promise is created, it can be in one of three states:**

**1. Pending:** The initial state of a promise, when it is created.

**2. Fulfilled:** The promise has been resolved, and the value is available.

**3. Rejected:** The promise has been rejected, and an error has occurred.

**Example :**

const promise = new Promise((resolve, reject) => {

// Asynchronous operation (e.g., API call, database query)

setTimeout(() => {

// Resolve the promise with a value

resolve("Hello, World!");

}, 2000);

});

promise.then((value) => {

console.log(value); // Output: "Hello, World!"

});

**Promises provide several benefits, including:**

**- Improved readability:** Promises make asynchronous code easier to read and maintain.

**- Error handling:** Promises provide a way to handle errors in a centralized way.

**- Chaining:** Promises can be chained together to perform multiple asynchronous operations.

**Some common promise methods include:**

**- then:** Handles the resolved value.

**- catch**: Handles errors.

**- finally:** Executes a function regardless of the promise's outcome.

* 1. **Write a program of promises and handle that promises also**

**Ans :** Here's an example program that demonstrates promises and handling them:

**Example :**

// Create a promise that resolves after 2 seconds

const promise = new Promise((resolve, reject) => {

setTimeout(() => {

// Resolve the promise with a value

resolve("Hello, World!");

}, 2000);

});

// Handle the promise

promise

.then((value) => {

// Handle the resolved value

console.log(value); // Output: "Hello, World!"

return "Value processed";

})

.then((processedValue) => {

// Handle the processed value

console.log(processedValue); // Output: "Value processed"

})

.catch((error) => {

// Handle any errors

console.error(error);

})

.finally(() => {

// Execute a function regardless of the promise's outcome

console.log("Promise settled");

});

// Create a promise that rejects after 3 seconds

const rejectedPromise = new Promise((resolve, reject) => {

setTimeout(() => {

// Reject the promise with an error

reject("Something went wrong");

}, 3000);

});

// Handle the rejected promise

rejectedPromise

.then((value) => {

// This block won't execute because the promise was rejected

console.log(value);

})

.catch((error) => {

// Handle the error

console.error(error); // Output: "Something went wrong"

})

.finally(() => {

// Execute a function regardless of the promise's outcome

console.log("Rejected promise settled");

});

**In this example, we create two promises:**

1. The first promise resolves after 2 seconds with a value of "Hello, World!".

2. The second promise rejects after 3 seconds with an error message of "Something went wrong".

**We handle these promises using the then, catch, and finally methods:**

- then handles the resolved value or processed value.

- catch handles any errors that occur during the promise's execution.

- finally executes a function regardless of whether the promise was resolved or rejected.

* 1. **Use fetch method for calling an api** [**https://fakestoreapi.com/products**](https://fakestoreapi.com/products)

**Ans :** To call the API at https://fakestoreapi.com/products using the fetch method in JavaScript, you can follow these steps. This example demonstrates how to use fetch to retrieve data from the API, handle the response, and process it:

**Example Using fetch :**

// Define the URL of the API

const apiUrl = 'https://fakestoreapi.com/products';

// Use fetch to make a GET request to the API

fetch(apiUrl)

.then(response => {

// Check if the response is successful

if (!response.ok) {

throw new Error('Network response was not ok');

}

// Parse the JSON from the response

return response.json();

})

.then(data => {

// Handle the data from the API

console.log('API Data:', data);

})

.catch(error => {

// Handle any errors that occurred during the fetch

console.error('Fetch error:', error);

});

**Using async/await Syntax :**

For a more readable and synchronous-looking approach, you can use async/await:

// Define an async function to fetch data from the API

async function fetchProducts() {

try {

// Define the URL of the API

const apiUrl = 'https://fakestoreapi.com/products';

// Fetch data from the API

const response = await fetch(apiUrl);

// Check if the response is successful

if (!response.ok) {

throw new Error('Network response was not ok');

}

// Parse the JSON from the response

const data = await response.json();

// Handle the data from the API

console.log('API Data:', data);

} catch (error) {

// Handle any errors that occurred during the fetch

console.error('Fetch error:', error);

}

}

// Call the async function

fetchProducts();

* 1. **Display all the product from the api in your HTML page**

**Ans :** First, create a basic HTML structure. You’ll have a container element where the products will be displayed.

1. **HTML CODE**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Product List</title>

<style>

.product {

border: 1px solid #ddd;

padding: 10px;

margin: 10px;

border-radius: 5px;

box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);

}

.product img {

max-width: 100px;

height: auto;

}

</style>

</head>

<body>

<h1>Product List</h1>

<div id="product-container"></div>

<script src="app.js"></script>

</body>

</html>

**2. JavaScript Code**

// Define the URL of the API

const apiUrl = 'https://fakestoreapi.com/products';

// Function to fetch and display products

async function fetchAndDisplayProducts() {

try {

// Fetch data from the API

const response = await fetch(apiUrl);

// Check if the response is successful

if (!response.ok) {

throw new Error('Network response was not ok');

}

// Parse the JSON from the response

const products = await response.json();

// Get the container element

const container = document.getElementById('product-container');

// Create HTML for each product and add it to the container

products.forEach(product => {

// Create a product element

const productElement = document.createElement('div');

productElement.className = 'product';

// Create HTML content for the product

productElement.innerHTML = `

<h2>${product.title}</h2>

<img src="${product.image}" alt="${product.title}">

<p><strong>Price:</strong> $${product.price}</p>

<p>${product.description}</p>

`;

// Append the product element to the container

container.appendChild(productElement);

});

} catch (error) {

// Handle any errors that occurred during the fetch

console.error('Fetch error:', error);

}

}

// Call the function to fetch and display products

fetchAndDisplayProducts();

# JavaScript Essentials

1. **Calculate subtotal price of quantity in JavaScript?**

**Ans** : To calculate the subtotal price of a quantity in JavaScript, you need to multiply the unit price of an item by the quantity purchased.

Here's a basic example of how you might do this:

// Define the unit price and quantity

let unitPrice = 59.98; // Price per item

let quantity = 2; // Number of items

// Calculate the subtotal

let subtotal = unitPrice \* quantity;

// Output the result

console.log("Subtotal Price: $" + subtotal.toFixed(2)); // Formatting to 2 decimal places

1. **What isJavaScript Output method?**

**Ans :** JavaScript has several output methods that allow you to display data or messages to the user.

**1. console.log():** Outputs data to the browser's console, typically used for debugging purposes.

**2. alert():** Displays a pop-up alert box with a message.

**3. document.write():** Writes HTML content directly to the document.

**4. innerHTML:** Sets the HTML content of an element.

**5. textContent:** Sets the text content of an element.

**6. console.table():** Outputs data in a tabular format to the console.

**7. console.error():** Outputs error messages to the console.

**8. console.warn():** Outputs warning messages to the console.

**9. console.info():** Outputs informational messages to the console.

**10. prompt():** Displays a pop-up prompt box that allows the user to input data.

**example of using some of these methods:**

console.log("Hello, World!"); // Outputs to console

alert("Hello, World!"); // Pop-up alert box

document.write("Hello, World!"); // Writes to document

document.getElementById("myElement").innerHTML = "Hello, World!"; // Sets HTML content

document.getElementById("myElement").textContent = "Hello, World!"; // Sets text content

These output methods are useful for displaying data, debugging, and interacting with users in JavaScript.

1. **How to used JavaScript Output method?**

**Ans :** Here are some examples of how to use JavaScript output methods:

**1. console.log()**

console.log("Hello, World!"); // Outputs "Hello, World!" to the console

**2. alert()**

alert("Hello, World!"); // Displays a pop-up alert box with "Hello, World!"

**3.document.write()**

document.write("Hello, World!"); // Writes "Hello, World!" directly to the document

**4. innerHTML**

document.getElementById("myElement").innerHTML = "Hello, World!"; // Sets the HTML content of the element with id "myElement"

**5. textContent**

document.getElementById("myElement").textContent = "Hello, World!"; // Sets the text content of the element with id "myElement"

**6. console.table()**

const data = [{ name: "John", age: 30 }, { name: "Jane", age: 25 }];

console.table(data); // Outputs the data in a tabular format to the console

**7. prompt()**

const name = prompt("Enter your name:");

console.log(name); // Outputs the user's input to the console

Remember to use these methods appropriately, considering the context and purpose of your code.

For example, console.log() is typically used for debugging, while alert() is used for user notifications. document.write() should be used with caution, as it can overwrite the entire document. innerHTML and textContent are used to update the content of HTML elements.

1. **How to used JavaScript Events to do all examples?**

**Ans :** Here are examples of how to use JavaScript events to perform various actions:

**1. Click Event**

// HTML

<button id="myButton">Click me!</button>

// JavaScript

document.getElementById("myButton").addEventListener("click", function() {

alert("Button clicked!");

});

**2. Hover Event**

// HTML

<div id="myDiv">Hover over me!</div>

// JavaScript

document.getElementById("myDiv").addEventListener("mouseover", function() {

console.log("Div hovered!");

});

**3. Key Press Event**

// JavaScript

document.addEventListener("keydown", function(event) {

console.log("Key pressed: " + event.key);

});

**4. Form Submission Event**

// HTML

<form id="myForm">

<input type="text" name="username">

<button type="submit">Submit</button>

</form>

// JavaScript

document.getElementById("myForm").addEventListener("submit", function(event) {

event.preventDefault(); // Prevent default form submission

console.log("Form submitted!");

});

**5. Page Load Event**

// JavaScript

window.addEventListener("load", function() {

console.log("Page loaded!");

});

**6. Scroll Event**

// JavaScript

window.addEventListener("scroll", function() {

console.log("Page scrolled!");

});

These examples demonstrate how to use JavaScript events to respond to user interactions and other events. You can attach event listeners to HTML elements or the window object to execute code when specific events occur.

Remember to use the addEventListener() method to attach event listeners, and pass the event type and callback function as arguments. The callback function will be executed when the event occurs.