Assignment Question (1)

1. Difference between HTTP and HTTPS

HTTP (HyperText Transfer Protocol) and HTTPS (HTTP Secure) are protocols for transferring data over the web. HTTP is the foundational protocol for web communication, while HTTPS is its secure version, adding encryption to protect data in transit. HTTPS is essentially HTTP layered over SSL/TLS (Secure Sockets Layer/Transport Layer Security).

Key Differences

Aspect	**HTTP**	**HTTPS**
Security	No encryption; data is sent in plain text, making it vulnerable to interception (e.g., eavesdropping, man-in-the-middle attacks).	Encrypts data using SSL/TLS, ensuring confidentiality, integrity, and authenticity.
Encryption	None.	Uses certificates (e.g., from Certificate Authorities like Let's Encrypt) to establish a secure connection.
Port	Default port 80.	Default port 443.
URL Scheme	Starts with http://.	Starts with https://.
Performance	Faster (no encryption overhead).	Slightly slower due to encryption/decryption, but modern optimizations (e.g., HTTP/2) minimize this.
Trust & SEO	Not trusted for sensitive data; browsers may warn users.	Browsers show a padlock icon; required for e-commerce, logins, etc. Search engines (e.g., Google) favor HTTPS for ranking.

Use Cases	Simple, non-sensitive sites (e.g., static blogs).	Any site handling personal data, payments, or requiring privacy (e.g., banking, social media).

How They Work

1.

- **HTTP**: When you visit a site, your browser sends a request to the server, which responds with data. No security layer, so anyone on the network can read the traffic.
- **HTTPS**: Before data exchange, a "handshake" occurs: The server presents a certificate; the browser verifies it, and a shared encryption key is established. All subsequent communication is encrypted.

Why HTTPS Matters

1.

- Protects against data breaches (e.g., stolen passwords or credit cards).
- Required by modern web standards; many browsers block HTTP content on HTTPS pages.
- Transitioning from HTTP to HTTPS is straightforward with free tools like Cerbo for certificates.

2. Define web architecture

Definition

1.

- Web architecture refers to the overall structure, design, and organization of web-based systems, including websites, web applications, and services. It encompasses the frameworks, technologies, and patterns used to build, deploy, and maintain these systems, ensuring they are scalable, secure, efficient, and user-friendly. This concept draws from software architecture principles but focuses on the web's distributed, client-server model.

Key Components

1.

   a. Web architecture typically includes the following layers and elements:

      - **Client-Side (Frontend)**: Handles user interaction via browsers. Technologies include HTML, CSS, JavaScript, and frameworks like React or Angular. This layer manages UI rendering and user inputs.
      - **Server-Side (Backend)**: Processes requests, manages data, and generates responses. Common technologies are languages like Python (with Django), JavaScript (Node.js), or Java (Spring), along with databases (e.g., SQL like MySQL or NoSQL like MongoDB).
      - **Network Layer**: Involves protocols like HTTP/HTTPS for communication, APIs (e.g., REST or GraphQL) for data exchange, and CDNs for content delivery to optimize performance.
      - **Infrastructure**: Includes servers, cloud platforms (e.g., AWS, Azure), load balancers, and security measures like SSL/TLS encryption.
      - **Data Layer**: Manages storage and retrieval, often with databases, caching (e.g., Redis), and data processing pipelines.

Types of Web Architectures

1.

   -

      - **Monolithic Architecture**: A single, unified application where all components are tightly coupled. Simple for small projects but harder to scale.

- **Microservices Architecture**: Breaks the system into small, independent services that communicate via APIs. Offers better scalability and fault isolation, as seen in platforms like Netflix.
- **Serverless Architecture**: Relies on cloud providers to handle servers, focusing on functions (e.g., AWS Lambda). Ideal for event-driven apps with variable traffic.
- **Progressive Web Apps (PWAs)**: Hybrid architectures that combine web and native app features, enabling offline access and push notifications.

3. Difference between ARIA and WA ARIA

Definitions

1. **. ARIA**: Short for Accessible Rich Internet Applications, a set of attributes and roles defined in the WAI-ARIA specification. It enhances web content accessibility for users with disabilities by providing semantic information to assistive technologies like screen readers.

   .WAI-ARIA: The full name of the specification, developed by the World Wide Web Consortium (W3C) under the Web Accessibility Initiative (WAI). It includes ARIA as its core component, along with guidelines for implementation.

In essence, ARIA and WAI-ARIA refer to the same technology—ARIA is the acronym, and WAI-ARIA is the complete title.

Key Differences (or Similarities)

1. Since they are synonymous, there are no substantive differences. However, here's a comparison for clarity:

Aspect	ARIA (as a component)	WAI-ARIA (full spec)
Scope	Focuses on the attributes, roles, and states (e.g., role="button", aria-label) used to make dynamic web content accessible.	Encompasses ARIA plus broader guidelines, authoring practices, and API mappings for various technologies.
Usage	Commonly used in code (e.g., HTML attributes like aria-expanded="true").	Refers to the entire W3C recommendation, including ARIA, which is updated periodically (e.g., WAI-ARIA 1.2 in 2019).
Examples	<button aria-label="Close menu">X</button> (provides screen reader context).	The spec includes ARIA examples plus rules for keyboard navigation and focus management.
Purpose	Improves accessibility for assistive tech, ensuring compliance with standards like WCAG.	Same as ARIA, but framed within the WAI's mission to make the web accessible to all

4. Behind the screen how web works

The web operates on a client-server model where users interact with websites or apps through browsers, and servers handle requests behind the scenes. This process involves multiple layers of technology, protocols, and infrastructure. Below, I'll break it down step-by-step, from a user's perspective to underlying mechanics.

1. User Interaction (The Front End)

1. You type a URL (e.g., www.example.com) into your browser or click a link.
   The browser (client) interprets this as a request to fetch and display content. It uses HTML, CSS, and JavaScript to render the page visually.
   Browsers like Chrome or Firefox act as interpreters, executing code to create interactive elements (e.g., buttons, forms).

2. Domain Name Resolution (DNS)

1.URLs aren't directly usable by computers; they need IP addresses (e.g., 192.0.2.1).
The browser queries on a Domain Name System (DNS) server to translate the domain name into an IP address. This is like looking up a phone number in a directory.
DNS is hierarchical: local DNS → ISP DNS → root servers → authoritative servers. It typically takes milliseconds and can be cached for speed.

3. Establishing a Connection (Network Layer)

1. Once the IP is resolved, the browser initiates a connection using the Transmission Control Protocol (TCP) over the Internet Protocol (IP).
For security, HTTPS (HTTP Secure) is used, which encrypts data via SSL/TLS certificates. This prevents eavesdropping or tampering.
The connection is established through a three-way handshake: SYN → SYN-ACK → ACK.

4. Sending the Request (HTTP/HTTPS Protocol)

1. The browser sends an HTTP request to the server, specifying the method (e.g., GET for fetching a page, POST for submitting data) and headers (e.g., user agent, cookies).
   Example: A GET request for a homepage might look like: **GET /index.html HTTP/1.1 Host: www.example.com**.
   APIs (e.g., REST or GraphQL) often handle dynamic requests, allowing apps to fetch data without reloading the page.

5. Server Processing (The Back End)

1. The server (e.g., running Apache, Nginx, or cloud services like AWS) receives the request.
   It processes it: routing to the right application, querying databases (e.g., MySQL for structured data or MongoDB for flexible data), and generating a response.
   Web applications (built with frameworks like Node.js, Django, or Ruby on Rails) handle logic, such as user authentication or e-commerce transactions.

Content Delivery Networks (CDNs) like Cloudflare may serve cached static assets (images, CSS) from nearby servers to speed things up.

6. Response and Rendering

1. The server sends back an HTTP response, including status codes (e.g., 200 OK, 404 Not Found), headers, and the content (HTML, JSON, etc.).
   The browser parses the HTML, applies CSS for styling, and runs JavaScript for interactivity.
   If the page includes resources (e.g., images, scripts), the browser makes additional requests (parallel for efficiency).
   Rendering engines (e.g., Blink in Chrome) convert code into the visual page you see.

Additional Layers and Technologies

1. **Databases and Storage**: Store user data, content, and sessions. Relational (SQL) for structured data; NoSQL for scalability.
   Caching: Browsers and servers cache data to reduce load (e.g., browser cache for faster reloads).
   Security: Firewalls, encryption, and tools like CAPTCHA protect against threats.
   Scalability: Load balancers distribute traffic; microservices break apps into manageable parts.
   Evolution: The web started with static HTML in the 1990s (e.g., Tim Berners-Lee's invention) and evolved to dynamic, real-time apps using WebSocket for live updates.

Real-World Example

When you search on Google:

1. Browser resolves google.com to an IP.
2. Send a secure request.
3. Google's servers process the query, search indexes, and return results.
4. Your browser renders the page with ads, links, and scripts.

5. Define CDN

Definition

- A Content Delivery Network (CDN) is a distributed network of servers strategically placed around the world to deliver web content (such as images, videos, stylesheets, and scripts) more efficiently to users. It acts as a middleman between the origin server (where the content is hosted) and the end-user device, reducing latency and improving performance.

How It Works

- **Caching and Distribution**: When a user requests content, the CDN routes the request to the nearest server (edge server) instead of the origin server. If the content is cached there, it's served directly; otherwise, it's fetched from the origin and cached for future requests.
- **Key Technologies**: Uses protocols like HTTP/HTTPS, DNS-based routing, and load balancing. CDNs often integrate with web architectures to handle traffic spikes.
- **Global Reach**: Servers are located in data centers worldwide (e.g., hundreds of points of presence), minimizing data travel distance.

Benefitis

- **Faster Load Times**: Reduces round-trip time (RTT) by serving content from nearby locations—e.g., a user in Europe gets data from a European server rather than one in the US.
- **Scalability**: Handles high traffic volumes, preventing origin server overload during events like viral videos or Black Friday sales.
- **Reliability and Security**: Provides DDoS protection, SSL/TLS encryption, and redundancy. It can also compress data and optimize images.
- **Cost Efficiency**: Offloads bandwidth from the origin server, potentially lowering hosting costs.

Examples

- **Popular CDNs**: Cloudflare, Akamai, Amazon CloudFront, and Fastly. For instance, Netflix uses CDNs to stream videos globally without buffering.
- **Use Case**: A website like YouTube relies on CDNs to deliver video files quickly, ensuring smooth playback even for millions of concurrent viewers.
  6. Define DNS

What is DNS?

- DNS stands for **Domain Name System**. It's a hierarchical and decentralized naming system that translates human-readable domain names (like www.example.com) into numerical IP addresses (like 192.0.2.1) that computers use to identify each other on the internet. This process is essential for web browsing, email, and other online services, as it allows users to access websites without memorizing complex numbers.

How DNS Works

1. **Query Initiation**: When you type a URL into a browser, your device sends a DNS query to a DNS resolver (often provided by your ISP or a service like Google Public DNS).
2. **Resolution Process**:
   - The resolver checks its cache for the IP address.
   - If not found, it queries root servers, then top-level domain (TLD) servers (e.g., .com), and finally authoritative name servers for the specific domain.
   - This hierarchical lookup ensures efficiency and scalability.
3. **Response**: The IP address is returned, allowing your device to connect to the server hosting the website.

Key Components

- **DNS Servers**: Include recursive resolvers, root servers, TLD servers, and authoritative servers.
- **Records**: DNS stores various records like A (address), CNAME (alias), MX (mail exchange), and TXT (text) to provide different types of information.
- **Protocols**: Primarily uses UDP for speed, with TCP as a fallback for larger responses.

Assignment Question (2):

1. Explain what is the smallest js file which you can excite in web

The Smallest Executable JS File

- The **smallest possible JS file** that can be executed in a web browser is an **empty file** (0 bytes). Here's why:
    - **File Content**: Nothing (literally empty).
    - **Execution**: Browsers load and parse it without issues. It runs successfully but performs no actions, logs nothing, and doesn't alter the page.
    - **Example Usage**: If you have an HTML file like this:

```
<!DOCTYPE html>
<html>
<head>
  <script src="empty.js"></script> <!-- empty.js is 0 bytes -->
</head>
<body>
  <p>Hello World</p>
</body>
</html>
```

The page loads fine, and the empty JS executes (doing nothing).

This is trivial and not very useful, but it's technically the smallest. If the file contained invalid JS (e.g., just **{**), it would throw a syntax error and fail to execute.

Smallest Non-Trivial JS File (That Does Something) If you want the smallest JS that actually performs an action (e.g., logs to console or shows an alert), here are minimal examples:

1. **Smallest File That Logs to Console** (7 bytes):
    - Content: **console.log (1);**
    - Execution: Opens the browser console and prints **1.** This is a single statement.
    - In HTML: **<script>console.log (1) ;</script>**

2. **Smallest File That Shows an Alert** (9 bytes):

- Content: **alert (1);**
- Execution: Pops up an alert box with "1".
- In HTML: **<script>alert (1) ;</script>**

3. **Even Smaller Valid Code** (1 byte):

   - Content: (a single semicolon).
   - Execution: Does nothing but valid JS (an empty statement). Browsers execute it without errors.
   - File size: 1 byte (in UTF-8 encoding).

These are the tiniest valid, executable snippets. JS engines (like V8 in Chrome) are forgiving and can run very minimal code.

Why This Matters Browser Compatibility: Modern browsers (Chrome, Firefox, Safari, Edge) handle these without issues. Performance: Tiny files load instantly, but real-world JS is larger for functionality. Edge **Cases**: If the JS file has encoding issues or is corrupted, execution fails. Always ensure it's served with the correct MIME type (**application/JavaScript** or **text/JavaScript**).

Assignment question (3):

1. 1. How arithmetic operations and concatenation work in browser?

 Web browsers execute JavaScript (JS) code, which handles arithmetic operations and string concatenation. These behaviors are defined by the ECMAScript specification, implemented in engines like V8 (Chrome), Spider Monkey (Firefox), or JavaScript Core (Safari). Operations occur in the browser's runtime environment, often triggered by user interactions, events, or scripts loaded in HTML.

2. Arithmetic Operations
3. Arithmetic operators (+, -, *, /, %, **) work primarily on numbers. JS uses floating-point arithmetic (IEEE 754 double-precision), which can lead to precision issues (e.g., 0.1 + 0.2 !== 0.3 due to binary representation).

   a. **Addition (+)**: Adds numbers. If operands are strings, it concatenates instead (see below).

b. **Subtraction (-)**: Subtracts numbers. Converts strings to numbers if possible (e.g., "5" - 2 = 3).
c. **Multiplication (*)**: Multiplies of numbers.
d. **Division (/)**: Divides numbers.
e. **Modulo (%)**: Returns remainder of division.
f. **Exponentiation ()** **: Raises power (ES2016+).

Type coercion applies: Non-numeric values are converted to numbers (e.g., **true** becomes 1, **false** or **null** becomes 0, **undefined** becomes NaN).

Examples in Browser Console (open DevTools with F12):

```
5 + 3;        // 8 (number addition)
"5" + 3;      // "53" (concatenation, see below)
5 - "3";       // 2 (string coerced to number)
10 * 2;        // 20
10 / 3;        // 3.3333333333333335
10 % 3;    // 1
2 ** 3;    // 8
```

String Concatenation

1. Concatenation combines strings using the + operator. It's not a separate operator but an overload of +.

   a. If both operands are strings, + join them.
   b. If one is a string and the other isn't, JS coerces the non-string to a string and concatenates.
   c. Other operators (-, *, /, etc.) don't concatenate; they attempt numeric operations and may coerce strings to numbers.

Examples:

```
"Hello" + " World";  // "Hello World"
"Hello" + 42;        // "Hello42" (42 coerced to string)
42 + "Hello";        // "42Hello" (same result, order doesn't matter for coercion)
5 + 5 + "5";         // "105" (left-associative: (5+5)="10", then "10"+"5")
```

Type Coercion Details

1.
   a. **To String ()**: For concatenation, numbers, Booleans, etc., are converted to strings (e.g., 123 → "123").

b. **To Number ()**: For arithmetic, strings are parsed as numbers (e.g., "123" → 123; "abc" → NaN).
c. **Edge Cases**: **null + "text"** → "null text"; **undefined + 1** → NaN (since undefined → NaN in arithmetic).

2)Programs on if, if else, nested if, switch. looping conditions. For loop

If conditions

1. Check if a Number is Positive

```
num = float (input ("Enter a number: "))

if num > 0:

   print ("The number is positive.")
```

2. Determine if a Number is Even or Odd

```
num = int (input("Enter an integer: "))

if num % 2 == 0:

   print ("The number is even.")

else:

   print ("The number is odd.")
```

3. Grade Based on Score

```
score = int (input ("Enter your score (0-100): "))

if score >= 90:

   print ("Grade: A")

elif score >= 80:
   print("Grade: B")
elif score >= 70:
   print("Grade: C")
else:
   print ("Grade: F")
```

4. Check Voting Eligibility

```
age = int(input ("Enter your age: "))
if age >= 18:
    print ("You are eligible to vote.")
else:
    print ("You are not eligible to vote.")
```

5. Find the Largest of Two Numbers

```
a = float (input ("Enter first number: "))
b = float (input ("Enter second number: "))
if a > b:
    print(f"{a} is larger.")
elif b > a:
    print(f"{b} is larger.")
else:
    print ("Both numbers are equal.")
```

6. Check if a Year is a Leap Year

```
year = int(input("Enter a year: "))
```

```
if (year % 4 == 0 and year % 100!= 0) or (year % 400 == 0):
   print ("It's a leap year.")
else:
   print ("It's not a leap year.")
```

7. Simple Login Check

```
username = input("Enter username: ")
password = input("Enter password: ")
if username == "admin" and password == "pass123":
   print("Login successful.")
else:
   print("Invalid credentials.")
```

if else condition

1. Check if a Number is Positive or Negative

```
num = float (input("Enter a number: "))
if num > 0:
   print("The number is positive.")
else:
   print("The number is negative or zero.")
```

2. Determine if a Number is Even or Odd

```
num = int(input("Enter an integer: "))
if num % 2 == 0:
   print("The number is even.")
```

```
else:
    print("The number is odd.")
```

3. Grade Based on Score

```
score = int(input("Enter your score (0-100): "))
if score >= 50:
    print("You passed.")
else:
    print("You failed.")
```

4. Check Voting Eligibility

```
age = int(input("Enter your age: "))
if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote.")
```

5. Find the Largest of Two Numbers

```
a = float(input("Enter first number: "))
b = float(input("Enter second number: "))
if a > b:
    print(f"{a} is larger.")
else:
    print(f"{b} is larger or equal.")
```

6. Check if a Year is a Leap Year

```
year = int(input("Enter a year: "))
if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
    print("It's a leap year.")
```

```
else:
    print("It's not a leap year.")
```

7. Simple Login Check

```
username = input("Enter username: ")
password = input("Enter password: ")
if username == "admin" and password == "pass123":
    print("Login successful.")
else:
    print("Invalid credentials.")
```

nested if condition

1. Check Number Type (Positive/Negative and Even/Odd)

```
num = int(input("Enter an integer: "))
if num >= 0:
    if num == 0:
        print("The number is zero.")
    else:
        print("The number is positive.")
        if num % 2 == 0:
            print("It is also even.")
        else:
            print("It is also odd.")
else:
    print("The number is negative.")
```

2. Grade with Detailed Feedback

```
score = int(input("Enter your score (0-100): "))

if score >= 50:

   if score >= 90:

      print("Grade: A - Excellent!")

   elif score >= 80:

      print("Grade: B - Good job!")

   else:

      print("Grade: C - Satisfactory.")

else:

   if score >= 30:

      print("Grade: D - Needs improvement.")

   else:

      print("Grade: F - Failed.")
```

3. Age-Based Eligibility for Activities

```
age = int(input("Enter your age: "))

if age >= 18:

   if age >= 65:

      print("You are eligible for senior discounts.")

   else:

      print("You are eligible to vote and drive.")

else:

   if age >= 13:

      print("You are a teenager; eligible for some activities.")

   else:

      print("You are a child; limited eligibility.")
```

4. Weather Decision Maker

```
temp = float(input("Enter temperature in Celsius: "))

if temp > 20:

   if temp > 30:

      print("It's hot; stay indoors.")

   else:

      print("It's warm; go for a walk.")

else:

   if temp < 10:

      print("It's cold; wear a jacket.")

   else:

      print("It's mild; enjoy the weather.")
```

5. Simple User Authentication with Role Check

```
username = input("Enter username: ")

password = input("Enter password: ")

if username == "admin":

   if password == "pass123":

      print("Admin login successful.")

   else:

      print("Invalid password for admin.")

else:

   if username == "user":

      if password == "user123":

         print("User login successful.")

      else:

         print("Invalid password for user.")
```

```
    else:

        print("Unknown username.")
```

6. Leap Year with Century Check

```
year = int(input("Enter a year: "))

if year % 4 == 0:

    if year % 100 == 0:

        if year % 400 == 0:

            print("It's a leap year (century leap).")

        else:

            print("It's not a leap year.")

    else:

        print("It's a leap year.")

else:

    print("It's not a leap year.")
```

7. Number Comparison with Three Numbers

```
a = int(input("Enter first number: "))

b = int(input("Enter second number: "))

c = int(input("Enter third number: "))

if a > b:

    if a > c:

        print(f"{a} is the largest.")

    else:

        print(f"{c} is the largest.")

else:

    if b > c:
        print(f"{b} is the largest.")
    else:
        print(f"{c} is the largest.")
```

Switch condition

1. Day of the Week Message

```
day = int(input("Enter a number (1-7) for the day: "))
if day == 1:
    print("Monday: Start of the workweek.")
elif day == 2:
    print("Tuesday: Keep going.")
elif day == 3:
    print("Wednesday: Midweek.")
elif day == 4:
    print("Thursday: Almost there.")
elif day == 5:
    print("Friday: Weekend is near.")
elif day == 6:
    print("Saturday: Relax.")
elif day == 7:
    print("Sunday: Rest day.")
else:
    print("Invalid day number.")
```

2. Calculator for Basic Operations

```
operation = input("Enter operation (+, -, *, /): ")
```

```
a = float(input("Enter first number: "))

b = float(input("Enter second number: "))

if operation == '+':

    print(f"Result: {a + b}")

elif operation == '-':

    print(f"Result: {a - b}")

elif operation == '*':

    print(f"Result: {a * b}")

elif operation == '/':

    if b != 0:

        print(f"Result: {a / b}")

    else:

        print("Division by zero error.")

else:

    print("Invalid operation.")
```

3. Month Name from Number

```
month = int(input("Enter month number (1-12): "))

if month == 1:

    print("January")

elif month == 2:

    print("February")

elif month == 3:

    print("March")

elif month == 4:

    print("April")

elif month == 5:
    print("May")
elif month == 6:
    print("June")
elif month == 7:
    print("July")
elif month == 8:
    print("August")
elif month == 9:
    print("September")
elif month == 10:
    print("October")
elif month == 11:
    print("November")
elif month == 12:
    print("December")
else:
    print("Invalid month number.")
```

4. Grade Letter to Description

```
grade = input("Enter grade letter (A, B, C, D, F): ").upper()
if grade == 'A':
    print("Excellent performance.")
elif grade == 'B':
    print("Good performance.")
elif grade == 'C':
    print("Satisfactory performance.")
elif grade == 'D':
    print("Needs improvement.")
elif grade == 'F':
    print("Failed.")
else:
    print("Invalid grade.")
```

5. Shape Area Calculator

shape = input("Enter shape (circle, square, triangle): ").lower() if shape == 'circle': r = float(input("Enter radius: ")) print(f"Area: {3.14159 * r * r}") elif shape == 'square': s = float(input("Enter side: ")) print(f"Area: {s * s}") elif shape == 'triangle': b = float(input("Enter base: ")) h = float(input("Enter height: ")) print(f"Area: {0.5 * b * h}") else: print("Invalid shape.")

6. Traffic Light Action

```
light = input("Enter light color (red, yellow, green): ").lower()
if light == 'red':
    print("Stop.")
elif light == 'yellow':
    print("Slow down.")
elif light == 'green':
    print("Go.")
else:
    print("Invalid color.")
```

7. User Role Permissions

```
role = input("Enter role (admin, user, guest): ").lower()
if role == 'admin':
    print("Full access granted.")
elif role == 'user':
    print("Limited access granted.")
elif role == 'guest':
    print("Read-only access.")
else:
    print("Unknown role.")
```

Assignment Question (4):

1)Execute do while 7 program, math object, function structure execution.

1. Sum of Squares Until Exceeds 100

```
import math

def sum_squares():
    total = 0
    num = 1
    while True:
        total += math.pow(num, 2)
        num += 1
        if total > 100:
            break
    print(f"Sum of squares: {total}")

sum_squares()
```

2. Factorial Calculation with Input Validation

```
import math


def calculate_factorial():

    while True:

        n = int(input("Enter a positive integer: "))

        if n > 0:

            fact = math.factorial(n)

            print(f"Factorial of {n} is {fact}")

            break

        else:

            print("Invalid input. Try again.")


calculate_factorial()
```

3. Sine Wave Approximation Until Precision

```
import math

def sine_approximation(): x = float(input("Enter angle in radians: ")) approx = 0 term = x n = 1
while True: approx += term term *= - (x ** 2) / ((2 * n) * (2 * n + 1)) n += 1 if abs(term) <
0.0001: break print(f"Approximated sin({x}) = {approx} (actual: {math.sin(x)})")

sine_approximation()
```

4. Prime Number Checker with Loop

```
import math


def is_prime():

    while True:
        num = int(input("Enter a number > 1: "))
        if num > 1:
            is_prime = True
            for i in range(2, int(math.sqrt(num)) + 1):
                if num % i == 0:
                    is_prime = False
                    break
            print(f"{num} is {'prime' if is_prime else 'not prime'}")
            break
        else:
            print("Invalid. Try again.")

is_prime()
```

5. Exponential Growth Simulation

```
import math


def exponential_growth():
    principal = float(input("Enter principal amount: "))
    rate = float(input("Enter annual rate (decimal): "))
    years = 0
    while True:
        amount = principal * math.pow(1 + rate, years)
        print(f"Year {years}: ${amount:.2f}")
        years += 1
        if amount >= principal * 2:
            break
```

```
exponential_growth()
```

6. Root Finding Using Bisection Method

```
import math


def bisection_root():
    a = float(input("Enter lower bound: "))
    b = float(input("Enter upper bound: "))
    while True:
        c = (a + b) / 2
        if abs(math.sin(c)) < 0.001:
            print(f"Root approximation: {c}")
            break
        if math.sin(a) * math.sin(c) < 0:
            b = c
        else:
            a = c


bisection_root()
```

7. Fibonacci Sequence Until Threshold

```
import math


def fibonacci_until():
    threshold = float(input("Enter threshold: "))
    a, b = 0, 1
```

```
  while True:

    print(a, end=" ")

    a, b = b, a + b

    if a > threshold:

      break

  print()


fibonacci_until()
```

1. 5. Assignment

1. Create an array in all the ways, work them throw all the loops

Anwser:Ways to Create Arrays in JavaScript

JavaScript arrays are dynamic, zero-indexed collections. Below are the main ways to create them, with examples. These work in browser environments (e.g., via DevTools console or scripts).

1. Array Literal (Most Common)

Use square brackets to define elements directly.

```
let arr1 = [1, 2, 3, 4];  // [1, 2, 3, 4]
let arr2 = [];            // Empty array
let arr3 = ["a", "b", "c"];  // String elements
```

2. Array Constructor

Use **new Array()** with elements or a single length argument

```
let arr1 = new Array(1, 2, 3);  // [1, 2, 3]
let arr2 = new Array(5);        // [empty × 5] (length 5, no
elements)
```

3. Array.of()

Creates an array from arguments (avoids issues with **new Array(5)**).

```
let arr = Array.of(1, 2, 3);  // [1, 2, 3]
```

4. Array.from()

Creates an array from an iterable (e.g., another array, string, or Set).

```
let arr1 = Array.from([1, 2, 3]);     // [1, 2, 3]
let arr2 = Array.from("hello");       // ["h", "e", "l", "l", "o"]
let arr3 = Array.from(new Set([1, 2, 2, 3]));  // [1, 2, 3]
```

5. Spread Syntax

Copies or merges arrays/objects.

```
let original = [1, 2, 3];
let arr1 = [...original];             // [1, 2, 3] (shallow copy)
let arr2 = [0, ...original, 4];       // [0, 1, 2, 3, 4]
```

6. Other Methods

a. From strings: **"hello".split("") → ["h", "e", "l", "l", "o"]**
b. From objects: **Object.values({a:1, b:2}) → [1, 2]**

Iterating Through Arrays with Loops

Assuming an array like **let arr = [10, 20, 30];**, here are ways to loop through it. Each example logs elements to the console (test in browser DevTools).

1. for Loop (Index-Based)

Accesses by index; efficient and flexible.

```javascript
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);  // 10, 20, 30
}
```

2. for...in Loop (Enumerates Indices)

Iterates over enumerable properties (indices); not ideal for arrays due to potential prototype pollution.

```javascript
for (let index in arr) {
  console.log(arr[index]);  // 10, 20, 30
}
```

3. for...of Loop (Value-Based)

Iterates over values; clean and modern.

```javascript
for (let value of arr) {
  console.log(value);  // 10, 20, 30
}
```

4. forEach() Method (Functional)

Calls a function for each element; no return value.

```javascript
arr.forEach((value, index) => {
  console.log(`${index}: ${value}`);  // "0: 10", "1: 20", "2: 30"
});
```

5. while Loop

Uses a condition; manual index management.

```javascript
let i = 0;
while (i < arr.length) {
  console.log(arr[i]);  // 10, 20, 30
  i++;
}
```

6. do...while Loop

Executes at least once, then checks condition.

```
let i = 0;
do {
  console.log(arr[i]);  // 10, 20, 30
  i++;
} while (i < arr.length);
```

For more advanced iteration (e.g., mapping or filtering), use methods like **map()**, **filter()**, or **reduce()**, but these aren't traditional loops. Test these in a browser to see output in the console.

2. Explore map(), for each(), fliter(), sort() reader?

Answer:Exploring Array Methods in JavaScript

JavaScript arrays have built-in methods for manipulation and iteration. These run in browsers via the JS engine (e.g., V8 in Chrome). Below, I explore **map()**, **forEach()**, **filter()**, **sort()**, and **reduce()** (assuming "reader" is a typo for **reduce()**, a common method). Each creates a new array or processes elements without mutating the original unless specified. Test examples in browser DevTools console.

1. map()

Transforms each element and returns a new array of the same length. Useful for modifying data.

Syntax**: **array.map(callback(element, index, array))

***callback**: Function to run on each element.*

Returns: New array with transformed elements.

***Example**:*

```
let numbers = [1, 2, 3, 4];
let doubled = numbers.map(num => num * 2);
```

```
console.log(doubled);  // [2, 4, 6, 8]
console.log(numbers);  // [1, 2, 3, 4] (original unchanged)
```

2. forEach()

Executes a function for each element but doesn't return a new array. Similar to a loop for side effects (e.g., logging or modifying external variables).

Syntax**: **array.forEach(callback(element, index, array))

***callback**: Function to run on each element.*

*Returns: **undefined**.*

***Example**:*

```
let fruits = ["apple", "banana", "cherry"];
fruits.forEach((fruit, index) => {
  console.log(`${index}: ${fruit}`);  // "0: apple", "1: banana", "2: cherry"
});
```

3. filter()

*Creates a new array with elements that pass a test (return **true** from the callback).*

Syntax**: **array.filter(callback(element, index, array))

***callback**: Function returning a boolean.*

Returns: New array with matching elements.

***Example**:*

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

4. sort()

Sorts the array in place (mutates original) based on a compare function. Defaults to string comparison if no function provided.

Syntax**: **array.sort(compareFunction?)

***compareFunction(a, b)**: Returns negative (a before b), positive (b before a), or 0 (equal).*

Returns: The sorted array (same reference).

***Example**:*

```
let numbers = [3, 1, 4, 1, 5];
numbers.sort((a, b) => a - b);  // Ascending
console.log(numbers);  // [1, 1, 3, 4, 5]

let strings = ["banana", "apple", "cherry"];
strings.sort();  // Alphabetical
console.log(strings);  // ["apple", "banana", "cherry"]
```

5. reduce()

Reduces the array to a single value by applying a function cumulatively.

Syntax**: **array.reduce(callback(accumulator, element, index, array), initialValue?)

***callback**: Function combining accumulator and current element.*

***initialValue**: Starting value for accumulator (optional, but recommended).*

Returns: The final accumulated value.

***Example**:*

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

let max = numbers.reduce((acc, num) => Math.max(acc, num));
console.log(max);  // 4
```

Key Notes **Immutability**: **map()**, **filter()**, and **reduce()** return new arrays; **forEach()** and **sort()** modify or don't return arrays.**Performance**: **forEach()** is for side effects; **map()** for transformations; **filter()** for selection; **sort()** for ordering; **reduce()** for aggregation.**Browser Compatibility**: All are ES5+ (widely supported); test in older browsers if needed.**Chaining**: Combine them, e.g., **arr.filter(...).map(...).reduce(...)**

3. Complete DOM manipulation ?

Answer:What is the DOM?

The Document Object Model (DOM) is a programming interface for web documents. It represents the page as a tree of objects (nodes) that you can manipulate with JavaScript in browsers. Changes update the live page. The DOM is part of the Web API, accessible via **document** in JS.

Selecting Elements

Use methods to grab elements for manipulation. These return single elements or collections.

- **By ID**: **document.getElementById('myId')** – Fast, unique.
- **By Class**: **document.getElementsByClassName('myClass')** – Returns HTMLCollection (live).
- **By Tag**: **document.getElementsByTagName('div')** – Returns HTMLCollection.
- **By Selector**: **document.querySelector('.myClass')** – First match (CSS selectors).
  document.querySelectorAll('.myClass') – All matches (NodeList, static).

Example (in HTML: **<div id="box" class="container">Hello</div>**):

```
let el = document.getElementById('box');
console.log(el.textContent);  // "Hello"
```

Modifying Elements

Change content, attributes, or styles.

- **Text Content**: **element.textContent = 'New text';** (plain text, ignores HTML).
  element.innerHTML = '<strong>New</strong> text'; (parses HTML, risky for user input).
- **Attributes**: **element.setAttribute('src', 'image.jpg');** or **element.src = 'image.jpg'**;.
- **Styles**: **element.style.color = 'red';** (inline styles). For classes: **element.classList.add('highlight');** (better for CSS).
- **Data Attributes**: **element.dataset.myKey = 'value';** (for custom data).

Example:

```
let el = document.querySelector('#box');
el.textContent = 'Updated!';
el.style.backgroundColor = 'blue';
el.classList.add('active');
```

Creating and Inserting Elements

Build new elements dynamically.

- **Create**: **let newEl = document.createElement('p');**
  Set properties: **newEl.textContent = 'New paragraph'**;.
- **Insert**:
  - Append: **parent.appendChild(newEl);** (end of parent).
  - Prepend: **parent.insertBefore(newEl, referenceEl)**;.
  - After/Before: **referenceEl.after(newEl);** or **referenceEl.before(newEl);** (modern).
  - Insert HTML: **parent.insertAdjacentHTML('beforeend', '<p>HTML</p>')**;.

Example:

```
let div = document.createElement('div');
div.innerHTML = '<h2>Title</h2>';
document.body.appendChild(div);
```

Removing Elements

Delete nodes from the DOM.

- **Remove Child**: **parent.removeChild(child)**;.
- **Self-Remove**: **element.remove();** (modern, simple).

Example:

```
let el = document.querySelector('#box');
el.remove();  // Poof!
```

Traversing the DOM

Navigate the tree.

- **Parent**: **element.parentNode** or **element.parentElement**.
- **Children**: **element.children** (HTMLCollection) or **element.childNodes**(NodeList, includes text).
- **Siblings**: **element.nextSibling** or **element.previousSibling**.

- **Closest Ancestor**: **element.closest('.class')** (matches selector upward).

Example:

```
let el = document.querySelector('li');
console.log(el.parentElement.tagName);  // "UL"
```

Event Handling

Respond to user actions (e.g., clicks) for interactive manipulation.

- **Add Listener**: **element.addEventListener('click', function(event) { ... });**.
- **Remove**: **element.removeEventListener('click', handler);**.
- **Event Object**: Access **event.target**, **event.preventDefault()**, etc.
- **Delegation**: Attach to parent for dynamic elements: **parent.addEventListener('click', (e) => { if (e.target.matches('.btn')) { ... } });**.

Example:

```
let btn = document.querySelector('#myBtn');
btn.addEventListener('click', () => {
  let newP = document.createElement('p');
  newP.textContent = 'Clicked!';
  document.body.appendChild(newP);
});
```

Best Practices and Tips

- **Performance**: Minimize DOM queries; cache selections (e.g., **let els = document.querySelectorAll('.item');**).
- **Security**: Use **textContent** over **innerHTML** to avoid XSS. Sanitize inputs.
- **Modern JS**: Prefer **querySelector** and **classList** over older methods.
- **Browser Support**: Most methods are ES5+; check for IE if needed.
- **Debugging**: Use browser DevTools (F12) to inspect and modify the DOM live.
- **Libraries**: For complex apps, consider React/Vue, but vanilla JS is fine for basics.
  4. Event key mouse, and event keys property ?Answer:Keyboard Events and the event.key Property

In JavaScript, keyboard events (like **keydown**, **keyup**, or **keypress**) provide information about which key was pressed via the **event.key** property. This is part of the **KeyboardEvent** interface and is the modern, recommended way to access key values (replacing the deprecated **event.keyCode** and **event.which**).

- **What it returns**: A string representing the key value, such as **"a"**, **"Enter"**, **"ArrowUp"**, or **"F1"**. It accounts for key modifiers (e.g., **"Shift+a"** becomes **"A"** if Shift is held).
- **Example usage**:

-
  ```
  document.addEventListener('keydown', (event) => {
      console.log('Key pressed:', event.key);
      if (event.key === 'Enter') {
          // Do something
      }
  });
  ```
- **Why it's useful**: It's more intuitive and consistent across browsers than older properties. For a full list of possible values, see the [MDN documentation on **event.key**].

Note: Mouse events (like **click**, **mousemove**) do not have a **key** property, as they relate to pointer interactions, not keyboard input. If you're trying to detect keyboard modifiers during a mouse event (e.g., Ctrl+click), use **event.ctrlKey**, **event.shiftKey**, etc.

Mouse Events and Related Properties

Mouse events in JavaScript (e.g., **click**, **mousedown**, **mousemove**) are handled via the **MouseEvent** interface. They don't have a direct "key" property, but they include properties for buttons, coordinates, and modifiers. Here's a breakdown:

- **Button-related properties**:
  - **event.button**: Indicates which mouse button was pressed (0 for left, 1 for middle, 2 for right).
  - **event.buttons**: A bitmask of all currently pressed buttons (e.g., 1 for left, 2 for right, 3 for both).
- **Position properties**:
  - **event.clientX / event.clientY**: Coordinates relative to the viewport.
  - **event.pageX / event.pageY**: Coordinates relative to the document.

    - **event.screenX** / **event.screenY**: Coordinates relative to the screen.
- **Modifier properties** (similar to keyboard events):
    - **event.ctrlKey**, **event.altKey**, **event.shiftKey**, **event.metaKey**: Booleans indicating if those keys are held during the mouse event.
- **Example usage**:

javascript
Copy code

```javascript
document.addEventListener('click', (event) => {
    console.log('Button:', event.button);  // 0 for left click
    console.log('Position:', event.clientX, event.clientY);
    if (event.ctrlKey) {
        console.log('Ctrl was held');
    }
});
```

5.Explore npm and npx?Answer:What is npm?

npm (Node Package Manager) is the default package manager for Node.js, used to install, manage, and share JavaScript packages (libraries, tools, and modules). It acts as a registry for over 1.8 million packages hosted on npmjs.com. npm is essential for JavaScript development, allowing you to handle dependencies in projects.

- **Key features**:
    - Installs packages locally (in **node_modules**) or globally.
    - Manages project dependencies via **package.json**.
    - Runs scripts defined in **package.json**.
- **Installation**: npm comes bundled with Node.js. Download Node.js from nodejs.org, and npm is included.
- **Basic commands**:
    - **npm init**: Initializes a new project with a **package.json** file.
    - **npm install <package>**: Installs a package locally.
    - **npm install -g <package>**: Installs globally (e.g., for CLI tools).
    - **npm run <script>**: Executes scripts from **package.json**.
    - **npm list**: Shows installed packages.
    - **npm update**: Updates packages to latest versions.

Example: To start a project and install Express.js:

Copy code

```
1.npm init -y
```

```
2npm install express
```

What is npx?

npx is a tool that comes with npm (since npm 5.2.0) and allows you to execute Node.js packages without installing them globally or locally. It's like a "run-once" executor for packages, making it easy to test tools or run commands from the npm registry.

- **Key features**:
  - Executes packages directly from the registry if not installed locally.
  - Useful for one-off tasks, like running build tools or generators.
  - Can run local binaries if they exist in **node_modules/.bin**.
- **Installation**: Included with npm/Node.js. No separate setup needed.
- **Basic commands**:
  - **npx <package>**: Runs the package (downloads temporarily if needed).
  - **npx <command>**: Executes a local script or binary.
  - **npx --yes <package>**: Skips prompts for yes/no questions.

Example: Create a React app without installing **create-react-app** globally:

Copy code

```
1npx create-react-app my-app
```

Key Differences Between npm and npx

Aspect	**npm**	**npx**
Purpose	Manages and installs packages	Executes packages without permanent install
Installation	Installs packages to disk (local/global)	Downloads and runs on-the-fly (cached temporarily)

Use Case	For dependencies in projects	For one-time executions or testing tools
Example	npm install lodash (adds to project)	npx cowsay "Hello" (runs without install)
Version	Part of npm CLI	Introduced in npm 5.2.0 as a companion tool

Common Use Cases and Tips

- **npm for projects**: Use npm to set up dependencies for apps (e.g., React, Vue). It ensures reproducibility via **package-lock.json**.
- **npx for utilities**: Great for tools like **npx http-server** to quickly serve files, or **npx eslint** to lint code without global installs.
- **Security note**: npx can run untrusted code, so verify packages from the registry.
- **Updating**: Run **npm install -g npm** to update npm, which includes npx.