What is Programming?

1. **Definition of a Computer Program**:
    a. A computer program is a **set of instructions** that a computer follows to complete a task.
    b. It works like a recipe, guiding the computer step-by-step.
2. **Purpose of Programs**:
    a. Programs can perform a wide range of tasks:
        i. Simple tasks like calculations (e.g., adding two numbers).
        ii. Complex operations like running video games or managing databases.
3. **Examples of Programs**:
    a. Web browsers, word processors, video games, and mobile apps are all programs.
4. **Programming Languages**:
    a. Programs are written in specific languages like **Python**, **JavaScript**, or **C++**.
    b. Each language has its own syntax (rules) and is suited for particular tasks:
        i. **JavaScript**: Commonly used for web development.
        ii. **Python**: Popular for data analysis, machine learning, and more.
5. **What is Programming?**
    a. Programming (or coding) is the process of writing these instructions in a programming language.
    b. It involves logical thinking and problem-solving.
6. **Learning to Program**:
    a. Requires understanding a programming language and practicing logical problem-solving.
    b. It's similar to learning a new language, but for communicating with computers instead of people.
7. **Impact of Programming**:
    a. Every app, website, or software is created through programming.
    b. Learning programming enables you to create apps, build websites, analyse data, and much more.
8. **Importance of Programming**:
    a. Programming is a powerful skill in today's digital age.
    b. It opens opportunities in technology, innovation, and automation.

Programming Languages

1. **Purpose of Programming Languages**:

a. Designed to build **instructions** for devices to execute.
b. Serve as a communication bridge between humans and computers.

2. **Why Not Binary?**
    a. Devices understand only **binary characters** (0 and 1).
    b. Writing code in binary is **inefficient** for humans.
    c. Programming languages simplify this process by using human-readable formats.
3. **Variety of Programming Languages**:
    a. Each language has specific uses:
        i. **JavaScript**: Used mainly for **web applications**.
        ii. **Bash**: Commonly used for **operating system scripting**.
4. **Low-Level vs. High-Level Languages**:
    a. **Low-Level Languages**:
        i. Closer to machine code (binary).
        ii. Fewer steps are needed for interpretation.
        iii. Example: Assembly language.
    b. **High-Level Languages**:
        i. Easier to read and write.
        ii. More abstract and support advanced features.
        iii. Example: JavaScript.
5. **Code Comparison**:
    a. **JavaScript** (High-Level):
        i. Readable and concise.
        ii. Uses constructs like variables, loops, and statements.
        iii. Example: Fibonacci sequence implemented in a few lines of code.

Code:

```
let number = 10
let n1 = 0, n2 = 1, nextTerm;
for (let i = 1; i <= number; i++) {
 console.log(n1);
 nextTerm = n1 + n2;
  n1 = n2;
 n2 = nextTerm;
  }
```

    b. **Assembly Language** (Low-Level):
        i. Complex and harder to read.
        ii. Closer to the device's hardware instructions.
        iii. Example: The same Fibonacci sequence written with detailed hardware-level commands.

Code:

```
area ascen,code,readonly
```

```
entry
Code32
adr r0,thumb+1
bx r0
code16
thumb
mov r0,#00
sub r0,r0,#01
mov r1,#01
mov r4,#10
ldr r2,=0x40000000
back add r0,r1
str r0,[r2]
add r2,#04
mov r3,r0
mov r0,r1
mov r1,r3
sub r4,#01
cmp r4,#00
bne back
end
```

6. **Fibonacci Sequence Example**:
    a. A Fibonacci sequence starts with **0** and **1**.
    b. Each subsequent number is the **sum of the two preceding numbers**.
    c. JavaScript code is easier to understand compared to assembly code for this algorithm.
7. **Key Takeaway**:
    a. High-level languages prioritize **readability** and **developer efficiency**.
    b. Low-level languages offer greater **control** over hardware but at the cost of **complexity**.

Elements of a Program

1. **Understanding Syntax and Statements**:
    a. **Syntax**:
        i. The set of **rules** that dictate how programs are written in a particular language.

        ii. Similar to grammar rules in natural languages.
        iii. Each programming language has its own syntax, which must be followed to write valid code.
    b. **Statements**:
        i. A single instruction that a program executes.
        ii. Comparable to sentences in natural languages.
        iii. Examples include assigning values, calling functions, and controlling program flow.
        iv. Must adhere to the syntax rules of the language.
2. **Programs are Data-Driven**:
    a. Programs operate by manipulating **data**:
        i. **Input**: Data from users, files, networks, etc.
        ii. **Processing**: Manipulation and computation on the data.
        iii. **Output**: Results displayed to users or written to a file.
    b. Example:
        i. A game takes **player actions** (input), updates the **game state** (processing), and displays the **updated screen** (output).
3. **Flow Control**:
    a. **Determines the sequence** in which statements are executed in a program.
    b. Common flow control structures:
        i. **Loops**: Repeat actions (e.g., `for`, `while` loops).
        ii. **Conditionals**: Make decisions based on conditions (e.g., `if`, `else` statements).
    c. Example:
        i. An `if` statement checks whether user input is valid before processing.
4. **Error Handling**:
    a. **Deals with runtime errors**:
        i. Errors caused by unexpected user input, missing resources, or code bugs.
    b. Techniques:
        i. **Try-Catch Blocks**: Detect and handle exceptions.
        ii. **Logging**: Record errors for debugging purposes.
        iii. **Graceful Failure**: Display user-friendly messages or attempt recovery.
    c. Importance:
        i. Prevents programs from crashing.
        ii. Ensures programs handle unexpected situations **gracefully**.
5. **Key Takeaways**:

a. A program consists of well-defined **syntax**, **statements**, and **flow control**.
b. All programs manipulate **data** and rely on **error handling** for robustness.
c. Mastering these elements is crucial for writing efficient and reliable programs.

Tools of the Trade

1. **Importance of Development Tools**:
    a. Development tools enhance **speed**, **accuracy**, and **efficiency** when writing code.
    b. They provide features to aid in **formatting** and ensuring **correctness**.
2. **Development Environment**:
    a. A **customized set of tools** and features for writing software.
    b. Tailored to meet **specific needs** of a developer.
    c. Evolves with changing **work priorities**, **personal projects**, or when switching to a new programming language.
3. **Code Editors**:
    a. Central tool where developers **write** and sometimes **run** their code.
    b. Crucial features include:
        i. **Debugging**: Step through code line-by-line to identify errors.
        ii. **Syntax Highlighting**: Adds color and formatting to make code readable.
        iii. **Extensions and Integrations**: Extend functionality with tools like spell check, documentation aids, or linters.
        iv. **Customization**: Modify the editor's layout, theme, and behavior to suit personal preferences.
4. **Popular Editors**:
    a. **Visual Studio Code**:
        i. Extensions: Code Spell Checker, Live Share, Prettier - Code Formatter.
    b. **Atom**:
        i. Extensions: Spell Check, Teletype, Atom-Beautify.
5. **Benefits of Extensions**:
    a. **Code Spell Checker**: Catches typos in the code.
    b. **Live Share**: Enables real-time collaboration.
    c. **Prettier - Code Formatter**: Automatically formats code for consistency.
    d. **Spell Check**: Ensures error-free comments and documentation.
    e. **Atom-Beautify**: Improves code structure and readability.
6. **Customization**:

    a. Most editors support **custom themes**, **shortcuts**, and **personalized extensions**.
    b. Helps developers create a **unique workflow** for better productivity.
7. **Conclusion**:
    a. A well-chosen and customized development environment significantly enhances the programming experience.
    b. Features like debugging, syntax highlighting, and extensions empower developers to write clean and efficient code.

Browser Technologies

1. **Role of Web Browsers**:
    a. Web browsers act as **clients** to run and view programs on the internet.
    b. Essential for **testing**, **debugging**, and **interacting** with web applications.
2. **Importance for Developers**:
    a. Browsers are primary tools for testing and debugging web apps.
    b. Modern browsers include powerful **developer tools** to streamline development.
3. **Browser Developer Tools Features**:
    a. **DOM Inspector**:
        i. Allows direct interaction with the HTML and CSS structure of a web page.
        ii. Enables real-time experimentation without reloading the code.
    b. **JavaScript Console**:
        i. Write, test, and debug JavaScript code within the browser.
        ii. Displays **errors** and **warnings**, aiding in issue identification and resolution.
    c. **Network Tools**:
        i. Monitor network requests and responses.
        ii. Check **load times** and optimize performance.
        iii. Debug API-related and data-fetching issues.
4. **Popular Browsers with Developer Tools**:
    a. Microsoft Edge
    b. Google Chrome
    c. Firefox
5. **Benefits of Browser Developer Tools**:
    a. Real-time **feedback** and debugging.
    b. Detailed insights into **performance** and **network behaviour**.
    c. Interactive features for faster **experimentation** and **problem-solving**.
6. **Conclusion**:
    a. Browser technologies are vital for running and refining web applications.

b. Developer tools simplify the debugging process and help ensure applications perform as expected.

Command-Line Tools

1. **Definition**:
   a. Command-line tools are text-based interfaces used to interact with a computer's operating system and manage development tasks.
2. **Why Use Command-Line Tools?**
   a. **Preference**: Some developers favour a minimalist, text-only interface for programming.
   b. **Better Workflow**:
      i. Typing-centric tasks reduce reliance on graphical elements.
      ii. Keyboard shortcuts improve efficiency for multitasking and managing files.
   c. **Ergonomics**: Reduces strain caused by frequent switching between mouse and keyboard.
   d. **Configurability**:
      i. Custom configurations can be created and saved.
      ii. Easily transferred across different development environments.
3. **Command-Line Options by OS**:
   a. **Windows**:
      i. PowerShell (💻 Preinstalled)
      ii. Command Prompt (CMD) (💻 Preinstalled)
      iii. Windows Terminal
      iv. Mintty
   b. **macOS**:
      i. Terminal (💻 Preinstalled)
      ii. iTerm
      iii. PowerShell
   c. **Linux**:
      i. Bash (💻 Preinstalled)
      ii. PowerShell
4. **Popular Command-Line Tools**:
   a. **Git**: Version control (💻 Preinstalled on most operating systems).
   b. **NPM**: Node.js package manager for JavaScript.

    c. **Yarn**: Fast, reliable JavaScript dependency manager.
5. **Benefits of Command-Line Tools**:
    a. **Efficiency**: Streamlined workflows for typing-heavy tasks.
    b. **Flexibility**: Supports scripting and automating repetitive tasks.
    c. **Portability**: Configurations can be reused across systems.
6. **Conclusion**:
    a. Command-line tools enhance productivity and are a key part of many developers' toolkits.
    b. Whether used exclusively or alongside graphical interfaces, they offer unmatched flexibility and control.

Developer Documentation (Pointers)

1. **Purpose of Developer Documentation**:
    a. Helps developers learn new tools, frameworks, or languages.
    b. Provides detailed guidance on how to use programming tools effectively.
    c. Offers deeper insights into the underlying mechanics of programming.
2. **Popular Web Development Documentation Resources**:
    a. **Mozilla Developer Network (MDN)**:
        i. Comprehensive resource for web development.
        ii. Covers HTML, CSS, JavaScript, and browser APIs.
        iii. Offers beginner-friendly guides and advanced technical references.
    b. **Frontend Masters**:
        i. Focused on advanced training in frontend development.
        ii. Includes tutorials, courses, and expert insights.
3. **Tip**:
    a. Research the differences between **web development** and **web design**:
        i. **Web Development**: Focuses on coding and building functional websites using programming languages and frameworks.
        ii. **Web Design**: Centers on aesthetics and user experience, involving tools like Adobe XD, Figma, and CSS design principles.
4. **Conclusion**:
    a. Developer documentation is an essential part of a programmer’s learning journey.
    b. By exploring resources like MDN and Frontend Masters, developers can stay updated and continually improve their skills.

1. Screen Readers

- **Purpose**: Screen readers are primarily used by individuals with vision impairments to audibly read out webpage content.
- **How they work**: They read the content from top to bottom, including text, links, and other elements. However, they cannot interpret non-text content (like images) unless you provide alternative descriptions.
- **Important Considerations**:
    - Use **alt text** for images.
    - Properly label interactive elements such as forms, buttons, and links.
    - Ensure that the page structure (headings, lists, etc.) is semantic, so the reader can interpret it correctly.
- **Popular Screen Readers**:
    - **Windows**: Narrator, JAWS, NVDA
    - **macOS/iOS**: Voiceover
- **Testing Tip**: Always test your webpage with screen readers to ensure they can read and navigate the content correctly.

2. Zooming

- **Purpose**: Zooming tools are helpful for users with vision impairments who need to magnify portions of the screen.
- **Types of Zoom**:
    - **Static Zoom**: Keyboard shortcuts (e.g., Ctrl + +) or screen resolution adjustments resize the entire page.
    - **Magnification**: Tools like **Windows Magnifier**, **ZoomText**, and **macOS/iOS Zoom** magnify portions of the screen, much like a magnifying glass.
- **Responsive Design**:
    - Ensure that your webpage uses **responsive design** principles, so the layout adapts and content remains legible when zoomed in.
    - Make sure images and text scale properly without breaking the layout.

3. Testing for Accessibility

- **Screen Reader & Zoom Testing**:
    - Test your webpage with different screen readers and zoom tools to ensure accessibility.
    - Make sure all content (including images, videos, and forms) is properly described and easy to navigate with screen readers and zoom features.

- **Browser Tools**: Some browsers have text-to-speech tools, but remember these are not full screen readers—use them for supplementary testing.

By incorporating these accessibility tools and techniques into your web development process, you help ensure that your site is usable by everyone, including those with vision impairments.

The Importance of Accessibility:

Web accessibility is about making your webpages usable by people with all abilities. As Tim Berners-Lee mentioned, the power of the web is in its universality, meaning everyone, regardless of disability, should have equal access.

1. **Tools Used by Users**: Users with disabilities often use assistive technologies like screen readers, voice commands, and keyboard navigation tools to interact with web pages.
2. **Developer Tools for Accessibility**: There are many tools available for developers to check and ensure accessibility, such as:
    a. **WAVE (Web Accessibility Evaluation Tool)**
    b. **Axe Accessibility Checker**
    c. **Lighthouse (for web performance and accessibility audits)**
3. **Skills for Accessible Web Design**:
    a. Use semantic HTML: Properly structure your HTML with elements like `<header>`, `<footer>`, `<article>`, etc., to make the content clear for screen readers.
    b. Provide text alternatives: Ensure that images, videos, and other media have descriptive `alt` text or captions.
    c. Focus on colour contrast: Ensure sufficient contrast between text and background colours for users with visual impairments.
    d. Keyboard accessibility: Ensure users can navigate your website using only the keyboard.
4. **Why Learn Accessibility Early**: It's easier to build accessible websites from the start rather than trying to retrofit them later. Making accessibility a priority will save you time and effort as you progress.

These principles will help ensure that your websites are inclusive, reaching a broader audience while adhering to web standards.

1. Screen Readers

- **Purpose**: Screen readers are primarily used by individuals with vision impairments to audibly read out webpage content.
- **How they work**: They read the content from top to bottom, including text, links, and other elements. However, they cannot interpret non-text content (like images) unless you provide alternative descriptions.
- **Important Considerations**:
    - Use **alt text** for images.
    - Properly label interactive elements such as forms, buttons, and links.
    - Ensure that the page structure (headings, lists, etc.) is semantic, so the reader can interpret it correctly.
- **Popular Screen Readers**:
    - **Windows**: Narrator, JAWS, NVDA
    - **macOS/iOS**: VoiceOver
- **Testing Tip**: Always test your webpage with screen readers to ensure they can read and navigate the content correctly.

2. Zooming

- **Purpose**: Zooming tools are helpful for users with vision impairments who need to magnify portions of the screen.
- **Types of Zoom**:
    - **Static Zoom**: Keyboard shortcuts (e.g., Ctrl + +) or screen resolution adjustments resize the entire page.
    - **Magnification**: Tools like **Windows Magnifier**, **ZoomText**, and **macOS/iOS Zoom** magnify portions of the screen, much like a magnifying glass.
- **Responsive Design**:
    - Ensure that your webpage uses **responsive design** principles, so the layout adapts and content remains legible when zoomed in.
    - Make sure images and text scale properly without breaking the layout.

3. Testing for Accessibility

- **Screen Reader & Zoom Testing**:
    - Test your webpage with different screen readers and zoom tools to ensure accessibility.

    - Make sure all content (including images, videos, and forms) is properly described and easy to navigate with screen readers and zoom features.
- **Browser Tools**: Some browsers have text-to-speech tools, but remember these are not full screen readers—use them for supplementary testing.

By incorporating these accessibility tools and techniques into your web development process, you help ensure that your site is usable by everyone, including those with vision impairments.

Ensuring Accessibility with Developer Tools

Testing your webpage for accessibility ensures that all users, regardless of disability, can access and interact with it. Developer tools like contrast checkers and Lighthouse help in identifying accessibility issues that might not be obvious through manual testing. Here's a breakdown of tools to ensure your webpage is accessible:

1. Contrast Checkers

- **Purpose**: Ensure that your webpage's colours are readable by people with colour blindness or those with low vision. The W3C has established a contrast ratio system to help identify sufficient colour contrast between text and its background.
- **Tools to Test Colour Contrast**:
    - **Palette Generation Tools**:
        - **Adobe Colour**: Interactive tool for testing colour combinations.
        - **Colour Safe**: Generates text colours based on a chosen background colour for readability.
    - **Compliance Checkers**:
        - **Edge**: WCAG Colour Contrast Checker extension.
        - **Firefox**: WCAG Contrast Checker extension.
        - **Chrome**: Colour Contrast Checker extension.
    - **Applications**:
        - **Colour Contrast Analyser (CCA)**: A desktop tool for checking colour contrast.
        - **Lighthouse**: A Google tool integrated into most browsers that analyses web pages, including accessibility features.

These tools help ensure that text and other important elements are distinguishable for colour-blind users or those with low vision.

2. Lighthouse Tool

- **Purpose**: Lighthouse is a popular tool developed by Google that audits web pages for performance, SEO, and accessibility. It gives an accessibility score and suggests areas of improvement.
- **How to Use Lighthouse**:
    - **Open Developer Tools**: Press F12 to open the developer tools in your browser.
    - **Access Lighthouse**: Click on the chevron (>>) icon to reveal hidden tabs and select "Lighthouse".
    - **Generate Report**: Under the "Categories" section, uncheck all options except "Accessibility", select "Desktop" under "Device", and click "Generate report".
    - **Interpret Results**: Lighthouse will provide a score, highlighting potential accessibility issues and offering suggestions for improvement.

Note: While Lighthouse provides a helpful accessibility score, it's not the final authority. It should be used as a starting point to identify issues, and further manual testing may be needed.

3. Practical Exercise:

- **Test Your Page's Accessibility**:
    - Open your browser's developer tools (F12).
    - Go to the Microsoft homepage (or any webpage you want to test).
    - Use Lighthouse to generate an accessibility report and note the score and suggestions for improving accessibility.

By using these tools regularly, you ensure that your webpage is not only functional but also accessible to users with different disabilities, helping you create a more inclusive web experience.

Designing for Accessibility

- **Start with Accessibility in Mind**
    - Designing an accessible page from the start is easier than fixing it later.
- **Use HTML as Intended**
    - HTML provides built-in functionality for buttons, links, and forms.
    - Avoid using `<span>` or `<div>` to mimic buttons or links.

    - Semantic elements improve screen reader compatibility and keyboard navigation.
- **Use Proper Headings (`<h1>` to `<h6>`)**
    - Headings provide structure and help screen reader users navigate.
    - Avoid using `<div>` or `<span>` for styling instead of semantic headings.
- **Provide Visual Cues**
    - Users rely on visual indicators like underlined links and focus outlines.
    - Avoid removing default styles (e.g., focus outlines) without alternatives.
- **Consider Keyboard Navigation**
    - Some users rely solely on keyboards to navigate.
    - Ensure a logical tabbing order by structuring HTML elements properly.
    - Use CSS for layout without disrupting natural keyboard flow.
- **Test Keyboard Accessibility Manually**
    - Ensure all interactive elements can be accessed via `Tab` and activated with `Enter` or `Spacebar`.
    - WebAIM provides guidelines on keyboard navigation testing.

Overview of Variables in JavaScript

- **Importance of Variables:**
    - Essential for interactivity in web development.
    - Help track state and store data dynamically.
- **Usage of Variables:**
    - Store user selections and preferences.
    - Hold results of calculations or data for future reference.
- **Data Types in JavaScript:**
    - Define the shape and size of stored data.
    - Common types include:
        - **Numbers** (e.g., `10`, `3.14`)
        - **Strings** (e.g., `"Hello"`, `'JavaScript'`)
        - **Boolean** (e.g., `true`, `false`)
        - **Objects & Arrays** for complex data structures.
- **Comparison with Other Languages:**
    - Similarities exist with other programming languages.
    - Syntax might differ, but fundamental concepts remain consistent.
- **Beginner-Friendly Learning:**
    - No prior coding experience required.
    - Concepts are broken down for easy understanding.

Using Variables to Remember Values in JavaScript

1. Variables in JavaScript

- Variables store values that can be used and changed throughout the code.
- They help remember values for later use, such as user inputs or calculations.

2. Declaring Variables

- Syntax: `[keyword] [variableName];`
- Example: `var aVariable;`

- `var` is used to declare a variable (older way).

3. Using `let` for Block Scope (Recommended)

- Introduced in **ES6**.
- Provides **block scope**, making it a better choice than `var`.
- Example: `let myVariable;`

- **Scope Differences:**
  - `var` variables are function-scoped.
  - `let` variables are block-scoped (available only within `{}` they are declared in).

4. Assigning Values to Variables

- Using the **assignment operator (=)**: `myVariable = 123;`

- **Difference between = and == or ===**:
  - = assigns a value.
  - == checks equality (type conversion allowed).
  - === checks strict equality (no type conversion).

5. Explicit Initialization

- Declaring and assigning a value at the same time: `let pokerChips = 100;`
- Useful for setting default values.

6. Changing Variable Values

- Variables declared with `let` can be reassigned: `let myVariable = 123;`
  `myVariable = 321;  // Value changed`

7. Constants (`const`)

- Constants store fixed values that **cannot be reassigned**.
- Declared using `const`: `const PI = 3.14;`
- **Key rules for `const`:**
  - Must be **initialized** at declaration.
  - Cannot be **reassigned**.

8. Constants with Objects

- **Reassigning a constant reference is not allowed:** `const obj = { a: 3 };`
  `obj = { b: 5 };  // ❌ Not allowed`
- **Modifying object properties is allowed:** `const obj = { a: 3 };`
  `obj.a = 5;  // ✅ Allowed`

Conclusion

- Use `let` for variables that can change.
- Use `const` for fixed values.
- Avoid `var` unless necessary (due to function scoping issues).
- Understanding variable scope and assignment helps in writing efficient JavaScript code.

Texas Hold'em game state model in **JavaScript**:

app.js

```
// Constants for game setup
const STARTING_POKER_CHIPS = 100; // Each player's initial chips
const PLAYERS = 3; // Number of players
const NO_OF_STARTER_CARDS = 2; // Number of starter cards

// Variables to track each player's current points
let playerOnePoints = STARTING_POKER_CHIPS;
let playerTwoPoints = STARTING_POKER_CHIPS;
let playerThreePoints = STARTING_POKER_CHIPS;

// Simulating a game round
playerOnePoints -= 50; // Player One bets 50 chips
playerTwoPoints -= 25; // Player Two bets 25 chips
playerThreePoints += 75; // Player Three wins and gets the pot

// Logging the game state
console.log("Player One Points:", playerOnePoints);
console.log("Player Two Points:", playerTwoPoints);
console.log("Player Three Points:", playerThreePoints);
```

Breakdown of Key Concepts:

1. **Constants (`const`) for Fixed Game Rules**
    a. `STARTING_POKER_CHIPS`, `PLAYERS`, and `NO_OF_STARTER_CARDS` are **constants** because they **do not change** throughout the game.
2. **Variables (`let`) for Changing Values**
    a. `playerOnePoints`, `playerTwoPoints`, and `playerThreePoints` are **variables** because they **change** as the game progresses.
3. **Game Round Simulation**
    a. Players bet some points, and the winner collects the total pot.
    b. We **update** their points using `-=` and `+=` operators.
4. **Logging the Game State**
    a. `console.log()` helps track the game progress and test the logic.

app.js

```
// Number Data Type
let age = 25;
let pi = 3.14159;
let negativeNumber = -10;

// Arithmetic Operations
let sum = 5 + 10;       // 15
let difference = 10 - 5; // 5
let product = 4 * 5;     // 20
let quotient = 10 / 2;   // 5
let remainder = 10 % 3;  // 1

console.log("Sum:", sum);
console.log("Difference:", difference);
console.log("Product:", product);
console.log("Quotient:", quotient);
console.log("Remainder:", remainder);

// String Data Type
let firstName = "Aaryan";
let lastName = "Verma";

// String Concatenation
let fullName = firstName + " " + lastName;
console.log("Full Name:", fullName);

// Template Literals (Recommended for formatting)
let greeting = `Hello, ${firstName}! Welcome to JavaScript.`;
console.log(greeting);

// Boolean Data Type
let isStudent = true;
let hasGraduated = false;

console.log("Is Student:", isStudent);
console.log("Has Graduated:", hasGraduated);

// Type Coercion
console.log("1 + 1 =", 1 + 1);      // 2 (Number)
console.log("'1' + '1' =", '1' + '1'); // "11" (String
```

```
Concatenation)
console.log("'1' + 1 =", '1' + 1);   // "11" (String Concatenation)
console.log("1 + '1' =", 1 + '1');   // "11" (String Concatenation)

// Truthy and Falsy Values
console.log("Boolean(0):", Boolean(0));  // false
console.log("Boolean(1):", Boolean(1));  // true
console.log("Boolean(''):", Boolean(''); // false
console.log("Boolean('Hello'):", Boolean('Hello')); // true
console.log("Boolean(null):", Boolean(null)); // false
console.log("Boolean(undefined):", Boolean(undefined)); // false

// Undefined Data Type
let notDefined;
console.log("Not Defined:", notDefined);

// Symbol Data Type (Unique Values)
let symbol1 = Symbol("A unique value");
let symbol2 = Symbol("A unique value");

console.log("Symbol1 === Symbol2:", symbol1 === symbol2); // false
(Each symbol is unique)
```

Breakdown of Key Concepts:

1. **Number Data Type**
    a. Supports **integers, decimals, negative numbers**
    b. Performs **arithmetic operations** (+, -, *, /, %)
2. **String Data Type**
    a. Defined with **single ' ' or double " " quotes**
    b. Use **+ for concatenation**
    c. **Template literals** (backticks ` `) allow better formatting
3. **Boolean Data Type**
    a. `true` or `false` values
    b. Used for **decision-making**
4. **Type Coercion in JavaScript**
    a. `"1" + 1` results in `"11"` because **string concatenation takes priority**
5. **Truthy & Falsy Values**
    a. `0`, `""`, `null`, `undefined`, `NaN` → **Falsy**

   b. **Everything else → Truthy**
6. **Undefined & Symbol Data Type**
   a. `undefined` → A variable that hasn't been assigned a value
   b. `Symbol()` → Creates a **unique, immutable identifier**

app.js

```
// Constants
const STARTING_POKER_CHIPS = 100; // Each player starts with 100
chips
const PLAYERS = 3;
const NO_OF_STARTER_CARDS = 2;

// Boolean to track game status
let gameHasEnded = false;

// Player Names
let playerOneName = "Chloe";
let playerTwoName = "Jasmine";
let playerThreeName = "Jen";

// Game Introduction
console.log(`Welcome to Texas Hold'em. The championship title will
be awarded to one of these three players: ${playerOneName},
${playerTwoName}, and ${playerThreeName}. Each player has
${STARTING_POKER_CHIPS} in their pot. We have an exciting game ahead
of us. May the best player win!`);

// Player Points (Chips)
let playerOnePoints = STARTING_POKER_CHIPS;
let playerTwoPoints = STARTING_POKER_CHIPS;
let playerThreePoints = STARTING_POKER_CHIPS;

// Game Simulation: Players lose or win chips
playerOnePoints -= 50; // Chloe loses 50 chips
playerTwoPoints -= 25; // Jasmine loses 25 chips
playerThreePoints += 75; // Jen wins 75 chips
```

```
// Check if the game has ended (one player has all the chips)
gameHasEnded = ((playerOnePoints + playerTwoPoints) == 0) || //
Player Three has won
               ((playerTwoPoints + playerThreePoints) == 0) ||  //
Player One has won
               ((playerOnePoints + playerThreePoints) == 0);  //
Player Two has won

console.log("Game has ended:", gameHasEnded);
```

How to Run the Game

1. Save the file as `app.js`.
2. Open the terminal and navigate to the file location.
3. Run the following command: `node app.js`

4. Expected Output:
```
Welcome to Texas Hold'em. The championship title will be awarded to one of these three players: Chloe, Jasmine, and Jen. Each player has 100 in their pot. We have an exciting game ahead of us. May the best player win!
Game has ended: false
```

Test Different Scenarios

Try setting:

```
playerOnePoints = 0;
playerTwoPoints = 0;
```

This should make `gameHasEnded` return `true`.

Overview of Functions

- Functions are reusable blocks of code that perform specific tasks.
- They help avoid code duplication, making maintenance easier.

- Functions can be called from other functions for modular programming.
- Naming functions properly acts as a form of documentation.

Creating and Calling a Function

- Functions take inputs (parameters) and return outputs.
- Components of a function:
    - **Function Body** – Block of code executed when the function is called.
    - **Parameters** – Inputs passed to the function (optional).
    - **Return Value** – Output generated by the function (optional).

Syntax:

```
function functionName(parameter) {
  // Function body
}
```

Example:

```
function displayGreeting() {
  console.log('Hello, world!');
}

// Calling the function
displayGreeting();
```

Methods vs. Functions

- A **function** is a standalone block of code.
- A **method** is a function attached to an object (e.g., `console.log`).

Best Practices

- Use **descriptive names** to clarify the function's purpose (e.g., `displayGreeting` instead of `greet`).
- Follow **camelCase** for readability (`displayGreeting` instead of `displaygreeting`).
- Keep functions **focused on a single task** for better reusability and debugging.
- Add **comments** (`//`) to describe the function's purpose.

Function Parameters

- **Parameters** (also called arguments) allow functions to accept input, making them more flexible.
- They are listed inside parentheses `()` in the function definition.
- Multiple parameters can be separated by commas.

Syntax:

```
function functionName(param1, param2, param3) {
  // Function body
}
```

Example:

```
function displayGreeting(name) {
    const message = `Hello, ${name}!`;
    console.log(message);
}

// Calling the function with an argument
displayGreeting('Christopher');
// Output: "Hello, Christopher!"
```

Default Parameter Values

- You can assign **default values** to parameters in case no argument is provided.
- Syntax: `parameterName = 'defaultValue'`.

Example:

```
function displayGreeting(name, salutation = 'Hello') {
    console.log(`${salutation}, ${name}`);
}

displayGreeting('Christopher');
// Output: "Hello, Christopher"
```

```
displayGreeting('Christopher', 'Hi');
// Output: "Hi, Christopher"
```

Key Takeaways

✓ Functions can accept parameters to make them more reusable.

✓ Default values ensure functions work even if some arguments are missing.

✓ Multiple parameters should be separated by commas.

Return Values in Functions

- **Return values** allow functions to output data for use elsewhere in a program.
- The `return` keyword stops function execution and sends a value back to the caller.
- **Local variables** inside a function are not accessible outside (they are **out of scope**).

Syntax:

```
function functionName(parameter) {
    return someValue;
}
```

Example: Returning a Value

```
function createGreetingMessage(name) {
    const message = `Hello, ${name}`;
    return message;
}

// Storing the returned value in a variable
let greetingMessage = createGreetingMessage('Christopher');
console.log(greetingMessage);
// Output: "Hello, Christopher"
```

Key Takeaways

✓ Use `return` to send a value back from a function.

✓ A function without `return` outputs `undefined` by default.

✓ **Local variables** inside a function **cannot be accessed** outside.

✓ Functions with return values can be reused with different inputs for dynamic results.

This exercise demonstrates how to create and call a function in JavaScript that returns a message and displays it on a webpage. Below is a structured explanation:

Steps to Create and Display a Message Using a Function

1. **Create the HTML Structure**
    a. The `index.html` file includes a basic webpage structure.
    b. The `<script>` tag contains JavaScript code to generate and display the message.
2. **Define the Function (`getMessage`)**
    a. The function **accepts a parameter** (`name`).
    b. It **returns** a string `"Hello, " + name + "..."`.
3. **Call the Function and Display the Message**
    a. The function is called with `"Ornella"` as an argument.
    b. The returned message is stored in a variable `message`.
    c. `document.write(message)` writes the message directly on the webpage.

Full Code

```
<!DOCTYPE html>
<html>
<head>
    <title>Message</title>
</head>
<body>
    <script>
        // Function to generate a greeting message
```

```
        function getMessage(name) {
            return 'Hello, ' + name + '...';
        }

        // Calling the function and displaying the message
        const message = getMessage('Ornella');
        document.write(message);

        // TODO: Add setTimeout code
    </script>
</body>
</html>
```

How to Run the Code

1. **Open Visual Studio Code**
2. **Create a folder** `# Windows`
   `md functions && cd functions`

   `# macOS/Linux`
   `mkdir functions && cd functions`

3. **Open the folder in VS Code** `code .`

4. **Create an `index.html` file** and copy the above code.
5. **Open Live Server**
   a. Press `Ctrl + Shift + P` (Windows) or `Cmd + Shift + P` (Mac).
   b. Type `Live Server` and select **"Live Server: Open with Live Server"**.
   c. The browser will display: **"Hello, Ornella..."**.

Key Takeaways

✓ Functions make code **reusable** and **organized**.

✓ `return` allows a function to send data back to the caller.

✓ `document.write()` can directly write content to the webpage.

✓ Running with **Live Server** provides a dynamic preview.

Key Takeaways from Callbacks, Anonymous Functions, and Arrow Functions

1. Understanding Callbacks

A **callback** is a function passed as an argument to another function. The receiving function then calls the callback when a certain event occurs.

✓ **Example**: Using `setTimeout` to delay execution:

```
function displayDone() {
    console.log('Done!');
}

// Calls displayDone after 3 seconds
setTimeout(displayDone, 3000);
```

⚠️ **Important:** Do not use `setTimeout(displayDone(), 3000);` because it immediately executes `displayDone()` instead of passing it as a function reference.

2. Anonymous Functions

An **anonymous function** is a function without a name. It's useful when a function is needed only once, such as in callbacks.

✓ **Example: Using an Anonymous Function with `setTimeout`**

```
setTimeout(function() {
    console.log('Done!');
}, 3000);
```

✅ This avoids **namespace pollution** by not defining unnecessary function names.

3. Arrow Functions (ES6 Feature)

Arrow functions provide a **shorter syntax** for writing anonymous functions.

✓ **Example: Arrow Function with `setTimeout`**

```
setTimeout(() => {
    console.log('Done!');
}, 3000);
```

✅ The `() => {}` syntax makes it cleaner and easier to read.

Comparison of Different Approaches

Approach	Syntax Example	When to Use?
Named Function	`setTimeout(displayDone, 3000);`	When the function is reused.
Anonymous Function	`setTimeout(function() { console.log('Done!'); }, 3000);`	When the function is used only once.
Arrow Function	`setTimeout(() => { console.log('Done!'); }, 3000);`	When using ES6+ and prefer shorter syntax.

Best Practices

✓ Use **named functions** for reusability.

✓ Use **anonymous functions** when the function is needed only once.

✓ Use **arrow functions** for cleaner and more readable code.

🚀 **Now you can confidently use callbacks, anonymous functions, and arrow functions in JavaScript!**

This exercise demonstrates how to use an **anonymous function** inside `setTimeout` to delay execution.

Breakdown of the Code

1. Display the Initial Message

The function `getMessage(name)` returns a greeting message, which is then displayed using `document.write()`.

```
function getMessage(name) {
    return 'Hello, ' + name + '...';
}
const message = getMessage('Ornella');
document.write(message);
```

✅ This prints **"Hello, Ornella..."** immediately when the page loads.

2. Use `setTimeout` to Display a Delayed Message

We use an **arrow function** inside `setTimeout` to change the message after **2 seconds**:

```
setTimeout(
    () => { document.write('...Hello again!') },
    2000
);
```

✅ After **2 seconds**, it replaces the previous text with **"...Hello again!"**.

Potential Issue: Overwriting Content

- `document.write()` **overwrites** the existing content.
- To **append** instead of replacing, use `innerHTML`: `setTimeout(`
  `() => { document.body.innerHTML += '<p>...Hello again!</p>'; },`
  `2000`
  `);`

✅ This keeps both messages on the page.

Final Code (Best Practice - Appending Instead of Overwriting)

```
<!DOCTYPE html>
<html>
<head>
    <title>Message</title>
</head>
<body>
    <script>
        // Display the first message
        function getMessage(name) {
            return 'Hello, ' + name + '...';
        }
        const message = getMessage('Ornella');
        document.body.innerHTML = `<p>${message}</p>`;

        // Display the second message after 2 seconds
        setTimeout(() => {
            document.body.innerHTML += '<p>...Hello again!</p>';
        }, 2000);
    </script>
</body>
</html>
```

✅ This ensures both messages appear, instead of one replacing the other.

🚀 **Now your page properly handles delayed execution without overwriting content!**

◇ Boolean Operators in JavaScript

✅ **Booleans** can only have two values: `true` or `false`. They are useful for making decisions in code.

◇ Declaring Booleans

```
let myTrueBool = true;
let myFalseBool = false;
```

◇ Common Comparison Operators

Symbol	Description	Example
<	Less than	`5 < 6 // true`
<=	Less than or equal to	`5 <= 6 // true`
>	Greater than	`5 > 6 // false`
>=	Greater than or equal to	`5 >= 6 // false`
===	Strict equality (value + type)	`5 === "5" // false`
!==	Inequality	`5 !== 6 // true`

◇ Using Booleans in Assignments

```
let timeOfDay = 8;
let timeToWakeUp = timeOfDay >= 8;  // `true`
console.log(timeToWakeUp);  // Output: true
```

◇ Try It Yourself!

1️⃣ Open **Developer Tools** in your browser (F12 → Console).

2️⃣ Type in some comparisons like:

```
console.log(10 > 5);  // true
console.log(3 === "3");  // false
console.log(7 !== 7);  // false
```

3️⃣ Observe the results—any surprises? 😜

◇ Understanding If...Else Statements in JavaScript

✅ The `if` statement **executes a block of code if the condition is `true`**. If it's `false`, the `else` block (if present) executes instead.

◇ Basic `if` Statement Syntax

```
if (condition) {
    // Runs if condition is true
}
```

📌 **Example:**

```
let isTrue = true;
if (isTrue) {
    console.log("This will run! ✅");
}
```

◇ Using a Logical Operator

📌 **Syntax:**

```
if (value operator compareToThisValue) {
    // Runs if the condition is true
}
```

📌 **Example:**

```
let currentMoney = 1000;
let laptopPrice = 1500;

if (currentMoney >= laptopPrice) {
    console.log("Getting a new laptop! 💻");
}
```

❌ **This won't execute** because `1000 >= 1500` is **false**.

◇ Adding an `else` Statement

If the `if` condition **fails**, the `else` block runs instead.

📌 **Example:**

```
let currentMoney = 1000;
let laptopPrice = 1500;

if (currentMoney >= laptopPrice) {
    console.log("Getting a new laptop! 💻");
} else {
    console.log("Can't afford a new laptop, yet! 😥");
}
```

💡 **Output:** `"Can't afford a new laptop, yet! 😥"`

◇ 🛠️ Try It Yourself!

1️⃣ Open **Developer Tools** (F12 → Console).

2️⃣ Copy-paste the code above.

3️⃣ **Modify `currentMoney`** to test different outcomes.

🔍 **Challenge:**

- What happens if `currentMoney = 2000`?
- What if `laptopPrice = 500`?

🚀 Keep experimenting!

◇ Understanding Logical Operators in JavaScript

Logical operators allow **multiple conditions** to be combined into a **single Boolean expression**.

◇ Logical Operators Overview

Symbol	Description	Example	Returns
&& (AND)	Both conditions must be true	`(5 > 6) && (5 < 6)`	`false`
`		` (OR)	At least one condition must be true
! (NOT)	Negates a Boolean value	`!(5 > 6)`	`true`

◇ Using Logical Operators in Assignments

📌 **Example: Using OR (||)**

```
let isHoliday = true;
let isMember = true;
let hasDiscount = isHoliday || isMember;

console.log(hasDiscount); // true ✅
```

💡 **Only one condition needs to be `true` for `hasDiscount` to be `true`.**

◇ Using Logical Operators in `if...else` Statements

📌 **Example: Buying a Laptop with OR (||)**

```
let currentMoney = 800;
let laptopPrice = 1000;
let laptopDiscountPrice = laptopPrice * 0.8; // 20% discount
```

```
if (currentMoney >= laptopPrice || currentMoney >=
laptopDiscountPrice) {
    console.log("Getting a new laptop! 💻");
} else {
    console.log("Can't afford a new laptop, yet! 😢");
}
```

💡 **If `currentMoney` is enough for either the full price or the discounted price, the purchase happens.**

🔹 Using the NOT (`!`) Operator

📌 **Example: Negation in `if...else`**

```
let isRaining = false;

if (!isRaining) {
    console.log("Go outside! 🌞");
} else {
    console.log("Stay indoors! ☔");
}
```

💡 **The `!` operator flips `false` to `true`, so the first block runs.**

🔹 Ternary Operator (`?` `:`)

A **ternary operator** is a **compact alternative** to `if...else`.

📌 **Syntax:**

```
let variable = condition ? <return this if true> : <return this if
false>;
```

📌 **Example: Finding the Biggest Number**

```
let firstNumber = 20;
let secondNumber = 10;
let biggestNumber = firstNumber > secondNumber ? firstNumber :
secondNumber;

console.log(biggestNumber); // 20 ✅
```

💡 **Equivalent `if...else` version:**

```
let biggestNumber;
if (firstNumber > secondNumber) {
    biggestNumber = firstNumber;
} else {
    biggestNumber = secondNumber;
}
```

🔹 🛠️ Try It Yourself!

1️⃣ Open **Developer Tools** (F12 → Console).

2️⃣ Copy-paste the code above.

3️⃣ **Modify values** to test different outcomes.

🚀 **Challenge:**

- Modify `currentMoney` and `laptopPrice` to see when the laptop is affordable.
- Use `!` to check if a person **does not** have a discount.

Happy coding! 🎯

Nice work! You've successfully implemented **Blackjack game rules** using **if...else statements and Boolean logic**. 🎉

◇ Breakdown of the Code

1️⃣ *Calculating the Player's Hand*

```
let cardOne = 7;
let cardTwo = 5;
let sum = cardOne + cardTwo; // 12
```

- The **player starts with two cards** (7 and 5).
- `sum` holds the **total points**.

2️⃣ *Drawing Another Card*

```
let cardThree = 7;
sum += cardThree; // Now sum = 19
if (sum > 21) {
  console.log('You lost');
  process.exit(1); // Exit the program
}
console.log(`You have ${sum} points`);
```

- A **third card** is drawn, increasing the total.
- If the **sum exceeds 21**, the player **loses immediately**.
- Otherwise, the score is displayed.

3️⃣ *Adding the Bank's Hand*

```
let cardOneBank = 7;
let cardTwoBank = 5;
let cardThreeBank = 6;
let cardFourBank = 4;
let bankSum = cardOneBank + cardTwoBank + cardThreeBank +
cardFourBank; // 22
```

- The **bank has four cards**.
- `bankSum` holds the **bank's total points**.

Determining the Winner

```
if (bankSum > 21 || (sum <= 21 && sum > bankSum)) {
  console.log('You win');
  process.exit(1); // Exit program
} else {
  console.log('Bank wins');
}
```

- If the **bank's total is over 21**, the player wins.
- If the **player has a higher valid score**, they win.
- Otherwise, the **bank wins**.

◇ Possible Improvements

✅ Randomized Card Draws

Instead of using fixed values, use `Math.random()`:

```
let getCard = () => Math.floor(Math.random() * 11) + 1;
let cardOne = getCard();
let cardTwo = getCard();
let sum = cardOne + cardTwo;
```

✅ Allow the Player to Draw Multiple Cards

Use a `while` loop to let the **player decide whether to draw** another card.

✅ Create a Function for Checking the Winner

Make the game **more reusable** by writing a function to check the winner.

Introduction

✅ JavaScript is one of the core technologies of the World Wide Web.

✅ It allows developers to create interactive content on websites.

✅ As a client-side scripting language, JavaScript enables code execution on users' devices.

What is an Array?

🔹 Arrays are a fundamental data structure in many programming languages, including JavaScript.

🔹 They store multiple values in a single variable.

🔹 Useful for handling collections of data (e.g., lists of names or sequences of numbers).

JavaScript Arrays

🟢 **Dynamic Size:**

- JavaScript arrays automatically adjust their size when items are added or removed.

🟢 **Array Methods:**

- JavaScript provides built-in methods to manipulate arrays:
  - **Adding/removing items**: `push()`, `pop()`, `shift()`, `unshift()`.
  - **Sorting/reversing**: `sort()`, `reverse()`.
  - **Iteration methods**: `forEach()`, `map()`, `filter()`, `reduce()`.

🟢 **Non-Homogeneous Elements:**

- JavaScript arrays can store different data types (numbers, strings, objects, or even other arrays).

Conclusion

✅ Mastering arrays is essential for becoming proficient in JavaScript.

Manipulating Arrays in JavaScript

✅ Arrays help store multiple values efficiently instead of using multiple variables.

✅ Useful for storing collections, such as order items or ice cream flavors.

Declaring an Array

📌 Arrays are created using square brackets `[]`.

📌 Example:

```
let iceCreamFlavors = ["Chocolate", "Strawberry", "Vanilla",
"Pistachio", "Neapolitan"];
```

Accessing an Item

- Use an index to access elements.
- Indexing starts from **0** (first element).
- Example:

```
iceCreamFlavors[3]; // Pistachio
```

Changing a Value

- Assign a new value to an existing index.
- Example:

```
iceCreamFlavors[4] = "Butter Pecan"; // Replaces "Neapolitan"
```

Adding More Values

- Use `push()` to add new items to the array.
- Example:

```
iceCreamFlavors.push("Mint Chip");
```

Finding Array Length

- Use `.length` to count the number of elements.
- Example:

```
iceCreamFlavors.length; // Returns total number of flavors
```

Removing a Value (Keeping Index)

- Use `delete` to remove an item but keep the index (sets value to `undefined`).
- Example:

```
delete iceCreamFlavors[iceCreamFlavors.length - 1]; // Removes "Mint Chip"
```

- The last index is now `undefined`, but still exists.

Replacing a Removed Value

- Assign a new value to the deleted index.
- Example:

```
iceCreamFlavors[iceCreamFlavors.length - 1] = "Your choice";
```

Removing an Item Completely

- Use `splice()` to remove an element **and shift remaining items**.

🔹 Syntax:

```
array.splice(index, number_of_elements_to_remove);
```

🔹 Example:

```
iceCreamFlavors.splice(2, 1); // Removes "Vanilla"
```

✅ Final Array:

```
["Chocolate", "Strawberry", "Pistachio", "Neapolitan", "Mint Chip"];
```

Iterating Over Arrays in JavaScript

🔹 **Why use loops?**

Loops help in performing operations on array elements efficiently, like printing, summing, or filtering values.

1️⃣ For Loop

- **Structure:**
```
for (let i = 0; i < array.length; i++) {
  console.log(array[i]);
}
```

- **Parts:**
  - **Counter:** `let i = 0;` (starting index)
  - **Condition:** `i < array.length;` (loop stops when false)
  - **Increment:** `i++` (updates counter)
- **Example:**
```
let iceCreamFlavors = ["Chocolate", "Strawberry", "Vanilla"];
for (let i = 0; i < iceCreamFlavors.length; i++) {
  console.log(iceCreamFlavors[i]);
}
```

- ✅ Best when you need **fine control** over iterations.

- ✅ Can use break to **exit early** when a condition is met.

2️⃣ While Loop

- **Structure:**
```
let i = 0;
while (i < array.length) {
  console.log(array[i]);
  i++;
}
```

- **Best for** when you **don't know the exact number of iterations** beforehand.

3️⃣ forEach() Loop

- **Simpler way to iterate over arrays.**
- **Structure:** `array.forEach(element => console.log(element));`

- **Example:**
```
let numbers = [1, 2, 3, 4, 5];
numbers.forEach(num => console.log(num));
```

- **With Index:** `numbers.forEach((num, index) => console.log(`Number ${num} at index ${index}`));`

- ❌ Cannot use break or continue inside.

4️⃣ for-of Loop (ES6)

- **Best for** iterating **only values** (no need for index).
- **Structure:**
```
for (let item of array) {
  console.log(item);
}
```

- **Example:**
```
let fruits = ["Apple", "Banana", "Cherry"];
for (let fruit of fruits) {
  console.log(fruit);
}
```

5️⃣ map() Method

- **Returns a new array** after transformation.
- **Example:** `let squares = [1, 2, 3, 4].map(num => num * num); console.log(squares); // [1, 4, 9, 16]`

- ✅ Great for transforming arrays without modifying the original.

Choosing the Right Loop

Loop Type	Use Case
for	Precise control, can break early
while	When iterations depend on a condition
forEach	Simple iteration, no early exit
for-of	Read-only iteration over values
map	Transforming arrays into new ones

Querying Arrays in JavaScript - Key Methods

- **Finding an Item (`find()`)**
    - Returns the first matching element or `undefined` if not found.

```
let iceCreamFlavors = ["Chocolate", "Strawberry", "Vanilla",
"Pistachio", "Neapolitan", "Mint Chip"];
iceCreamFlavors.find(flavor => flavor === "Chocolate"); //
"Chocolate"
```

- **Filtering Items (`filter()`)**
    - Returns an array of items that match the condition.

```
let iceCreamFlavors = [
  { name: "Chocolate", type: "Chocolate" },
  { name: "Strawberry", type: "fruit"},
  { name: "Vanilla", type: "Vanilla"},
  { name: "Pistachio", type: "Nuts"},
  { name: "Neapolitan", type: "Chocolate"},
  { name: "Mint Chip", type: "Chocolate"}
];
iceCreamFlavors.filter(flavor => flavor.type === "Chocolate");
```

```
// Returns [{ name: "Chocolate" }, { name: "Neapolitan" }, { name:
"Mint Chip" }]
```

- **Checking a Condition (`some()`)**
    - Returns `true` if at least one element matches the condition.

```
iceCreamFlavors.some(flavor => flavor.type === "Nuts"); // true
```

- **Excluding Items (`filter()`)**
    - Returns an array excluding items that match a condition.

```
iceCreamFlavors.filter(flavor => flavor.type !== "Nuts");
// Returns all flavors except "Pistachio".
```

- **Modifying Items (`map()`)**
    - Transforms each item and returns a new array.

```
iceCreamFlavors.map(flavor => ({ ...flavor, price: 1 }));
// Adds price: 1 to each item.
```

- **Aggregating Values (`reduce()`)**
    - Reduces an array to a single value (e.g., sum, object).

```
let sales = [
  { date: '2021-05-01', amount: 2 },
  { date: '2021-05-01', amount: 1 }
];
sales.reduce((acc, curr) => acc + curr.amount, 0); // Returns total
sales sum
```

Ice Cream Shop Operations Summary

1. Build Your Business

✅ Define all ice cream flavors:

```
let iceCreamFlavors = ["Chocolate", "Strawberry", "Vanilla",
"Pistachio", "Neapolitan", "Mint Chip", "Raspberry"];
```

✅ Categorize flavors by type and price:

```
let iceCreamFlavors = [
  { name: "Chocolate", type: "Chocolate", price: 2 },
  { name: "Strawberry", type: "Fruit", price: 1 },
  { name: "Vanilla", type: "Vanilla", price: 2 },
  { name: "Pistachio", type: "Nuts", price: 1.5 },
  { name: "Neapolitan", type: "Chocolate", price: 2 },
  { name: "Mint Chip", type: "Chocolate", price: 1.5 },
  { name: "Raspberry", type: "Fruit", price: 1 }
];
```

2. Simulate Customer Transactions

✅ Create a transactions array:

```
let transactions = [];
```

✅ Record transactions:

```
transactions.push({ scoops: ["Chocolate", "Vanilla", "Mint Chip"],
total: 5.5 });
transactions.push({ scoops: ["Raspberry", "StrawBerry"], total:
2 });
transactions.push({ scoops: ["Vanilla", "Vanilla"], total: 4 });
```

3. Analyze Business Performance

✅ **Calculate total earnings:**

```
const total = transactions.reduce((acc, curr) => acc + curr.total,
0);
console.log(`You've made ${total} $ today`); // You've made 11.5
$ today
```

✅ **Track flavor popularity:**

```
let flavorDistribution = transactions.reduce((acc, curr) => {
  curr.scoops.forEach(scoop => {
```

```
    if (!acc[scoop]) {
      acc[scoop] = 0;
    }
    acc[scoop]++;
  });
  return acc;
}, {});

console.log(flavorDistribution); // { Chocolate: 1, Vanilla: 3, Mint
Chip: 1, Raspberry: 1, StrawBerry: 1 }
```

4. Business Insights

- Total revenue for the day: **$11.5**
- Most sold flavor: **Vanilla (3 scoops)**
- Least sold flavors: **Mint Chip, Raspberry, Strawberry (1 scoop each)**
- Stock up on **Vanilla** due to high demand.

GitHub Copilot & Codespaces Summary

1. Introduction to GitHub Copilot

✅ **What is GitHub Copilot?**

- An AI-powered coding assistant that provides **autocomplete suggestions** while you write code.
- Uses **OpenAI Codex** to analyze your file and related files to suggest new lines or entire functions.
- Helps improve **developer workflows** in code writing, documentation, testing, and more.

✅ **What is GitHub Codespaces?**

- A **cloud-based developer environment** that runs in **Visual Studio Code**.
- Allows **customization** with dependencies, libraries, and extensions preinstalled.

2. Scenario: Improving a Project

🔹 **Objective:** Boost productivity by **typing code faster** for new and existing projects.

🔹 **How?** Use GitHub Copilot to enhance a repository by **modifying scroll behavior** and getting **live suggestions** while coding.

3. Learning Outcomes

✅ **By the end of this module, you will:**

- **Configure a GitHub repository** in Codespaces.
- **Install & use GitHub Copilot** extension.
- **Generate AI-powered code suggestions** using crafted prompts.
- **Apply Copilot to improve projects** in JavaScript.

4. Main Objective

🎯 **Customize a JavaScript project using GitHub Copilot in Codespaces** with AI-assisted prompts.

5. Prerequisites

📌 **Skills Required:**

- Basic **JavaScript** knowledge.
- Familiarity with **Git, GitHub**, and basic commands (`git add`, `git push`).

📌 **Accounts Needed:**

- **GitHub account**
- **GitHub Copilot subscription**

🚀 Now you're ready to leverage **GitHub Copilot & Codespaces** to streamline your development process!

GitHub Copilot Summary

- **What is GitHub Copilot?**

    - 🚀 AI-powered coding assistant for faster development.
    - 📌 Uses **prompts and natural language** to suggest code.
    - 📌 Helps with **syntax lookup, debugging, testing, and documentation**.
- **How Does GitHub Copilot Work?**
    - ✅ Recognizes prompts in **comments** and **Markdown files**.
    - ✅ Suggests multiple options; cycle through them using **Ctrl + Enter**.
    - ✅ Accept suggestions with the **Tab** key.
- **Setting Up GitHub Copilot**
    - 🔹 **GitHub Account** – Required for access.
    - 🔹 **Sign Up** – Subscribe to GitHub Copilot.
    - 🔹 **Enable Copilot** – Activate it in GitHub settings.
    - 🔹 **Install Extension** – Available for **VS Code, Visual Studio, and other IDEs**.

Setting Up GitHub Copilot in Visual Studio Code

1️⃣ *Accessing GitHub Copilot*

- **GitHub Account:**
    - Required to use Copilot.
    - Create one if needed.
- **Check Copilot Access:**
    - Review **pricing plans**.
    - **Copilot Free** includes:

✅ Inline code completions
✅ Multi-file editing
✅ Copilot Chat
✅ Multiple model selection
✅ Support for all editors & GitHub.com

    - **Copilot Pro** available for free to students & educators. (Details)
- **Enable Copilot:**
    - Go to **GitHub Profile > Settings > Copilot**.
    - Ensure **Copilot & Copilot Chat** are enabled.

2️⃣ *Installing GitHub Copilot Extension*

- Available for **VS Code, Visual Studio, JetBrains IDEs, VIM, and XCode**.
- Search for **GitHub Copilot** in your **IDE's extension marketplace** and install it.

3️⃣ *Setting Up the Development Environment*

- **Launch Codespaces** (preconfigured with Copilot).
- **Steps to Start Codespaces:**
    - Click **Open in GitHub Codespaces**.
    - On the **Create codespace** page, review settings.
    - Click **Create new codespace**.
    - Wait for it to start (**may take a few minutes**).

4️⃣ *Running the JavaScript Portfolio App*

- **Codespaces Terminal** appears after startup.
- All required extensions & dependencies auto-install.
- Run the web app using: `npm start`

- Once running, the terminal shows:
    - **Server running on port 1234** inside Codespaces.

⚡ **Note:**

- **GitHub Codespaces** allows **60 free hours per month** (2-core instances).
- More info: **GitHub Codespaces monthly limits**.

Using GitHub Copilot with JavaScript

1️⃣ *Developing with GitHub Copilot*

- Keeps code **fresh, updated**, and **bug-free**.
- Helps in **adding features** and **fixing issues**.
- Uses **GitHub Copilot Chat** for interactive Q&A.

2️⃣ *Prompt Engineering*

- **Prompts** guide Copilot to generate useful suggestions.
- A **clear, specific** prompt leads to better results.

✅ **Example of a bad prompt:**

```
// Create an API endpoint
```

- Too **vague** and could lead to an undesired output.

✅ **Example of a good prompt:**

```
// Create an API endpoint using the React framework that accepts a JSON payload in a POST request
```

- **Clear** and **specific**, ensuring better code generation.

3️⃣ *Best Practices for GitHub Copilot*

✅ **Start simple, then elaborate**

- **Basic prompt:** `<!-- Create an HTML form with a text field and button -->`
- **Improved prompt:** `<!-- Add an event listener to the button to send a POST request to /generate endpoint and display response in a div with id "result" -->`

✅ **Cycle through suggestions**

- Use **Ctrl + Enter** (or **Cmd + Enter** on Mac) to see multiple options.
- Use **GitHub Copilot Chat** for interactive adjustments.

✅ **Refine prompts if results aren't ideal**

- **Reword** the prompt or **start typing** to guide Copilot.

✅ **Leverage open files for context**

- Copilot analyzes **all open files** to generate better suggestions.
- Use `@workspace` in **Copilot Chat** to reference other files.

Updating a JavaScript Portfolio with GitHub Copilot

1️⃣ *Importance of a JavaScript Portfolio*

- Showcases **skills** and **experience**.
- Builds **credibility** when applying for jobs.
- Useful for **students, graduates, and professionals**.

2️⃣ *Customizing Your Portfolio*

- The portfolio is a **React-based web app**.
- Update your details in `src/App.jsx` inside `siteProps`: `const siteProps = {`

```
  name: "Your Name",
  title: "Your Role",
  email: "your.email@example.com",
  gitHub: "your-github",
  instagram: "your-instagram",
  linkedIn: "your-linkedin",
  medium: "",
  twitter: "your-twitter",
  youTube: "your-youtube",
};
```

3️⃣ *Animating Social Media Icons*

✅ **Ask Copilot for help:**

- Add this comment in `src/styles.css`: `/* add an amazing animation to the social icons */`

- Copilot suggests this CSS animation: `img.socialIcon:hover {`

```
  animation: bounce 0.5s;
  animation-iteration-count: infinite;
}

@keyframes bounce {
  0% {
    transform: scale(1);
  }
  50% {
```

```
    transform: scale(1.2);
  }
  100% {
    transform: scale(1);
  }
}
```

✅ **Press Tab** to accept the suggestion.

✅ **Hover over social media icons** to see the animation.

Summary

- **Used Copilot** to generate code interactively.
- **Enhanced UI** with animations.
- **Updated profile details** in a React app.
- **Live preview** in GitHub Codespaces.

🎉 **Congratulations! You now know how to leverage GitHub Copilot for frontend development.**