Untitled-7

Meet Ameen Alam:

An accomplished professional with extensive expertise in Cloud Computing and DevOps. He holds multiple certifications, including AWS Developer Associate and Kubernetes Application Developer, and has over 8 years of experience in the IT, finance, and banking industry. Ameen is currently the Founder & CTO at Doblier Inc.

- Facebook
- Instagram
- LinkedIn
- YouTube

Table of Contents:

Introduction to TypeScript:

TypeScript as a Statically Typed Superset of JavaScript:

TypeScript is a superset of JavaScript that adds optional static typing to the language, allowing code to be checked for errors during development, which helps catch bugs early on. TypeScript also provides other features, such as generics, interfaces, and type aliases, making it a more powerful and versatile language than JavaScript.

Understanding Programming Languages:

Imagine you're planning a trip. To communicate effectively with people in the places you'll visit, you might learn their languages. Similarly, programming languages are like human languages for computers, allowing us to give instructions and tell them what to do.

Why Learn Programming?

In today's digital world, programming skills are increasingly valuable. Just like knowing English opens up opportunities in various fields, programming can lead to exciting careers in technology, web development, and software engineering. It also empowers you to create your own digital solutions and automate tasks.

Why Choose TypeScript?

TypeScript is a powerful and versatile language that adds optional static typing to JavaScript, making it more reliable and maintainable. Imagine a chef using a recipe with precise measurements and instructions compared to one with vague guidelines. TypeScript is like the chef with a precise recipe, ensuring consistent and error-free outcomes.

Programming Languages Most In Demand In 2023:

Before tackling the list of the most in-demand programming languages for 2023, let's look at which ones ranked first in popularity in 2022. In a list of the top programming languages used by developers in 2022, released by Statista analysts, JavaScript came in first, followed by HTML, then SQL, Python, and TypeScript, in that order.

TypeScript Mastery: A Step-by-Step Learning Experience

Who is a Developer?

Imagine you're building a house. You have architects who design the blueprint, engineers who ensure it is structurally sound, and construction workers who put everything together. In the world of technology, developers play a similar role.

Developers:

- **Builders of the digital world:** They create the software, websites, apps, and other digital tools we use daily.
- **Translate ideas into reality:** Just like architects turn sketches into houses, developers turn ideas into working software, using programming languages to write code that instructs computers what to do.
- **Solve problems with code:** Developers solve problems creatively and efficiently, whether it's making online shopping easier or helping us navigate a new city.
- **Connect people and information:** From social media platforms to online banking systems, developers build the digital infrastructure that allows us to connect and access information.

Types of Developers:

- **Front-end developers:** Focus on the visual elements and user interface of websites and apps.
- **Back-end developers:** Work behind the scenes, building the logic and server-side functionality of software.
- **Full-stack developers:** Have expertise in both front-end and back-end development.

Becoming a Developer:

You don't need a specific degree to be a developer. Success can be achieved as a self-taught developer by mastering key skills and tools.

TypeScript Productivity Features:

- **Static types**
- **Access control keywords**
- **Concise class constructor syntax**

These features help prevent common coding errors, making development smoother.

How TypeScript Works:

TypeScript productivity features are applied to JavaScript code. The TypeScript package includes a compiler, and the compiled file can be executed by a JavaScript runtime, such as Node.js or a browser:

1. **TypeScript**
2. **TypeScript Compiler**
3. **JavaScript**

TypeScript's Advantages:

- **Maintainability:** Enhances code clarity and organization.
- **Rich IDE:** Provides comprehensive development tools.
- **Cross-Platform:** Runs on various platforms, making it versatile.
- **Object-Oriented Language:** Supports structured programming techniques.

Why Choose TypeScript Over JavaScript?

- **Optional Static Typing:** Adds type safety and reduces runtime errors.
- **IDE Support:** Provides robust development tools.
- **Object Orientation:** Encourages structured programming.
- **Readability:** Improves code clarity.
- **Community Support:** Offers a supportive network of developers.

Real-world Analogy: Baking a Cake

Think of baking a cake as an analogy for programming. The recipe is like the code, and the ingredients are like the data. Just as following a recipe with the correct ingredients leads to a delicious cake, using the right code with appropriate data produces the desired result in programming.

Saying of Sir Zia:

"TypeScript is a valuable tool for creating reliable and maintainable software applications. Its static typing, code clarity, and tooling support make it an excellent choice for beginners and experienced programmers alike. By learning TypeScript, you'll open doors to exciting career opportunities and gain the power to create your own digital solutions."

-Sir Zia Khan

Setting Up Your Environment:

To develop TypeScript applications, you need the following tools:

1. **Node.js:** A runtime environment for JavaScript, used to run TypeScript code after it has been compiled to JavaScript.

2. **Git:** A distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

3. **TypeScript:** The compiler that converts TypeScript code to JavaScript.

4. **Visual Studio Code:** A popular code editor with excellent support for TypeScript.

Installing TypeScript:

1. **Install TypeScript:**

   Open a command prompt or terminal and run the following command:

   ```
   npm install --global typescript@5.0.2
   tsc --version
   ```

2. **Install the TypeScript Extension:**

   Open VS Code and go to **Extensions** > **Search Extensions**. Search for "TypeScript: JavaScript with Language Features" and install it.

Chapter #2: Your First TypeScript Application:

1. **Your First TypeScript Program:**

   Let's write a simple "Hello, World!" program:

   ```
   console.log("Hello, World!");
   ```

   This code prints "Hello, World!" to the console.

2. **Compiling to JavaScript:**

   To compile this code to JavaScript, open a command prompt or terminal and navigate to the directory where you saved the TypeScript file. Then, run:

   ```
   tsc main.ts
   ```

Problem-Solving Power 15:

A 15-minute focused question-and-answer session:

1. **Bring your problems and questions:** Receive personalized guidance and feedback from experienced faculty.
2. **Collaborate:** Learn from fellow students' inquiries and share your own.
3. **Gain insights:** Sharpen your troubleshooting skills and understanding.
4. **Benefits:**
   - **Boost problem-solving expertise:** Develop a structured approach to analyzing and solving programming challenges.
   - **Enhance understanding:** Gain deeper insights into key programming concepts and applications.
   - **Sharpen coding skills:** Practice writing efficient and effective code through engaging exercises.
   - **Build confidence:** Feel empowered to tackle complex tasks and pursue independent learning.

PanaVerse:

Learn TypeScript 5.0+ in baby steps:

GitHub Repository

---

1. **Generate `tsconfig.json`:**

   Press **Shift + right-click** in the folder where you want to open the Command Prompt. Then, select the "Open command window here" option and run:

   ```
   tsc --init
   ```

   This creates the compiler configuration file.

2. **Initializing the Project Folder - Node.js:**

   The `npm init` command creates a `package.json` file, which keeps track of the packages required by the project and helps configure development tools.

Chapter #4: Hands-on Exercises:

Getting started exercises with TypeScript and Node.js.

I hope this revised version meets your needs. Let me know if you'd like further adjustments or additions. Here's a refined version of your document, making it clearer and better organized:

Getting Started Exercises with TypeScript and Node.js:

Note: Try these short programs to gain firsthand experience with TypeScript and Node.js. You might want to create a new folder for each exercise to keep them organized. Create a single GitHub repository to commit the code for these exercises, and once finished, submit the URL of the repo.

TypeScript Node Projects:

- Who will be the first to complete the exercises?
  - Learn TypeScript Repository
  - TypeScript Node Projects

Project Submission:

- Submit your projects here:

  TypeScript Project Submission Form:

  (We'll share the link with you later.)

Connect with Ameen Alam:

- Facebook
- LinkedIn
- Instagram
- YouTube
- LinkTree

TypeScript Mastery: A Step-by-Step Learning Experience

Presented by Sir Ameen Alam

(Part 2)

Meet Ameen Alam

An accomplished professional with extensive expertise in Cloud Computing and DevOps. He holds multiple certifications, including AWS Developer Associate and Kubernetes Application Developer, and has over 8 years of experience in the IT, finance, and banking industry. Ameen is currently the Founder & CTO at Doblier Inc.

- Facebook
- LinkedIn
- Instagram
- YouTube

Content

- GitHub Repository
- Link Tree

Step 00: Introduction to TypeScript

- **TypeScript Mastery: A Step-by-Step Learning Experience (Part 1)**
- Part 1 Presentation

Hello World Program

Introduction to TypeScript, setting up the environment, and writing a simple Hello World program.

Follow TypeScript Mastery: A Step-by-Step Learning Experience (Part 1)

Variable in Non-Programming

Imagine you have a box where you can store your toys. You can take toys out, put different toys in, or even check to see which toy is currently inside. The box serves as a container for whatever toy you decide to place in it at any time.

Similarly, a variable in programming acts like this box. It's a container in your computer's memory where you can store information, such as numbers, text, or more complex items. Just like you can change the toy in the box, you can also change the information stored in a variable.

In programming, a variable is used to store data that can be changed or manipulated throughout the execution of a program. Variables have names, so you can refer to them and manage their contents easily.

Declares a variable named favoriteColor and assigns it the value "blue"

```
let favoriteColor = "blue";
console.log(favoriteColor);
```

Changes the value of favoriteColor to "green"

```
favoriteColor = "green";
console.log(favoriteColor);
```

In the example above, we declare a variable `favoriteColor` and initially assign it the value "blue." Later, we change the value of `favoriteColor` to "green."

Case Sensitivity Explained

- **camelCase**
- **snake_case**
- **PascalCase**

An accomplished professional with extensive expertise in Cloud Computing and DevOps. He holds multiple certifications, including AWS Developer Associate and Kubernetes Application Developer, and has over 8 years of experience in the IT, finance, and banking industry. Ameen is currently the Founder & CTO at Doblier Inc.

- Facebook
- LinkedIn
- Instagram
- YouTube

Content

- GitHub Repository
- Link Tree

Step 00: Introduction to TypeScript

- **TypeScript Mastery: A Step-by-Step Learning Experience (Part 1)**
- Part 1 Presentation

Hello World Program

Introduction to TypeScript, setting up the environment, and writing a simple Hello World program.

TypeScript's Type System

Imagine you're organizing a big event and have assigned specific roles to your team members: photographers, decorators, and security personnel. Each role has clear expectations and tasks that cannot be exchanged without causing confusion or issues. For example, asking a photographer to handle security would not be ideal, as each person's skills and tools are suited to their specific role.

This scenario is similar to how TypeScript's type system works. Just like assigning specific roles helps manage your event smoothly, TypeScript assigns types to variables to ensure they are used correctly throughout your code. This helps prevent errors, such as mixing up numbers with text or trying to perform operations on incompatible data types.

Data Types in TypeScript

Imagine you're packing for a vacation and have different types of containers for your items: a bottle for liquids (like water or shampoo), a wallet for money, and a photo album for pictures. Each container is meant for a specific type of item, and using the wrong container wouldn't make sense.

TypeScript uses data types to know what kind of data is being stored and manipulated. For example, you wouldn't store text in a variable meant for numbers, just like you wouldn't put water in a photo album.

TypeScript enhances JavaScript by adding explicit types. This allows you to specify what kind of data a variable can hold, such as a number, string, or a more complex structure like an array or object.

Basic Data Types:

- **String:** For textual data
- **Number:** For numerical values (integers and floating-point numbers)
- **Boolean:** For true/false values
- **Any:** For variables that can hold any type of data

Variables in TypeScript: `let` and `const`

Consider two scenarios involving money in your wallet:

- **Let:** The amount of money you have changes frequently. One day you might have $50, and after shopping, you might have $20. This is similar to variables declared with `let` in TypeScript, where the value can change over time.

- **Const:** Your bank account number, on the other hand, is a constant. Once it's set, it doesn't change. This mirrors the `const` declaration in TypeScript, signifying that once a variable is assigned a value, it cannot be reassigned to something else.

In TypeScript, `let` and `const` are two ways to declare variables, with key differences in their mutability and scope:

- **Using `let`:** This declares a variable that can be reassigned to a different value. It's useful for values that change over time, like counters or values that depend on conditions.
- **Using `const`:** This declares a constant that cannot be reassigned once set. It's perfect for values that should remain the same throughout the execution of your program, like configuration settings or important identifiers.

Additional Primitive Data Types

- **Undefined:** Represents a variable that has not been assigned a value or has not been initialized. It is one of JavaScript's primitive types that TypeScript adopts.
- **Unknown:** Used for variables where the type is not known at the time of writing the code. It's a safer alternative to `any`, as it requires the type to be determined before it can be used.
- **BigInt:** A data type that can store numbers larger than the maximum limit for the number type, allowing representation of very large integers.
- **Symbol:** A unique and immutable primitive value that can be used as the key of an Object property. Symbols are often used to add unique property keys to an object that won't collide with keys any other code might add to the object and which are hidden from any mechanisms other code will typically use to access the object.
- **Null:** Represents the intentional absence of any object value. It is another primitive type in JavaScript used in TypeScript to signify that a variable intentionally has no value.

Syntax and Type Errors in TypeScript

- **Syntax Error:** Identifying and resolving syntax errors in TypeScript code.
- **Type Error:** Understanding type errors and how TypeScript's type system helps prevent them.
- **Assignability Error:** Learning about assignability errors and how to address them through type compatibility.

String Concatenation and Template Literals

```
let firstname: string = "Ameen";
let lastname: string = "Alam";

// Concatenation using the + operator
let fullName: string = firstname + " " + lastname;
console.log(fullName);

// Using template literals
fullName = `${firstname} ${lastname}`;
console.log(fullName);
```

Modules

Basics of modules, exporting, and importing modules.

Homework:

We will code a module later in our advanced session; for now, go to the advanced slides, comprehend the module, and utilize the Inquirer.

Operators in TypeScript

Operators allow us to perform operations on variables and values.

Addition (+):

You add apples to your cart. 2 apples and then 3 more apples give you 5 apples.

```
console.log(2 + 3);

let num1: number = 2;
let num2: number = 3;
let cart: number = num1 + num2;

console.log(cart);
```

Subtraction (-):

You have 5 apples in the cart, and you eat 2 apples, leaving you with 3 apples.

```
console.log(5 - 2);

let cart: number = 5;
let num3: number = 2;
let total: number = cart - num3;

console.log(total);
```

Multiplication (*):

You decide you need 4 bags of 5 apples. You now have 20 apples.

```
console.log(4 * 5);

let bags: number = 4;
let apples: number = 5;
let totalApples: number = bags * apples;

console.log(totalApples);
```

Division (/):

You decide to distribute these 20 apples equally into 4 bags. Each bag gets 5 apples.

```
console.log(20 / 4);

let bags: number = 4;
let totalApples: number = 20;
let applesPerBag: number = totalApples / bags;

console.log(applesPerBag);
```

Modulus (%):

You have 5 apples; you distribute them into 2 bags. Each bag gets 2 apples, and 1 apple remains (5 % 2 = 1).

```
console.log(5 % 2);

let apples: number = 5;
let bags: number = 2;
let remainder: number = apples % bags;

console.log(remainder);
```

Unary Operators:

Prefix and Postfix Operators:

Imagine you have a loyalty card that you use once (++) and then a coupon that you use once and throw away (--).

```
let a: number = 5;
let b: number = 2;

a++; // a becomes 6
b--; // b becomes 1

console.log(a, b);
```

Homework:

```
let a: number = 5;
let b: number = 2;
let c: number;

c = ++a + a++ + --b + b-- + a + b++ + b;

console.log(c);
```

Combining Operators:

```
let result = 3 + 4 * 5;

console.log(result); // Answer will be 23 or ??
```

TypeScript Mastery: A Step-by-Step Learning Experience

Operators in TypeScript

Operators allow us to perform operations on variables and values.

Mathematical Operators:

- **Addition (+):**

  An addition calculator:

```
let result = 3 + 4 * 5;

console.log(result); // Will be 23 or ??

import inquirer from "inquirer";

const input1 = await inquirer.prompt({
  name: "num1",
  type: "number",
  message: "Kindly enter your first number:",
});

const input2 = await inquirer.prompt({
  name: "num2",
  type: "number",
  message: "Kindly enter your second number:",
});

let total: number = input1.num1 + input2.num2;

console.log(total);
```

- **Subtraction (-):**

  You have 5 apples in the cart, and you eat 2 apples, leaving you with 3 apples.

```
console.log(5 - 2);

let cart: number = 5;
let num3: number = 2;
let total: number = cart - num3;

console.log(total);
```

- **Multiplication (*):**

  You decide you need 4 bags of 5 apples. You now have 20 apples.

```
console.log(4 * 5);

let bags: number = 4;
let apples: number = 5;
let totalApples: number = bags * apples;

console.log(totalApples);
```

- **Division (/):**

  You decide to distribute these 20 apples equally into 4 bags. Each bag gets 5 apples.

```
console.log(20 / 4);

let bags: number = 4;
let totalApples: number = 20;
let applesPerBag: number = totalApples / bags;

console.log(applesPerBag);
```

- **Exponentiation ()**:**

  A power operation:

```
let layer: number = 5;
let apple: number = 5;
let power: number = layer ** 2;

console.log(power); // 25
```

- **Modulus (%):**

  You distribute 5 apples into 2 bags. Each bag gets 2 apples, and 1 apple remains.

```
console.log(5 % 2);

let apples: number = 5;
let bags: number = 2;
let remainder: number = apples % bags;

console.log(remainder);
```

Homework:

Create an Addition, Subtraction, Multiplication, Division, Exponentiation, Modulus, and BMI Calculator using Inquirer.

```
let weightInKg = 70; // 70kg
let heightInMeters = 1.75; // 1.75m

let bmi = weightInKg / (heightInMeters * heightInMeters);
console.log(`Your BMI is ${bmi}`);
```

Assignment:

- **Assignment Operators (=):**

```
let c = 10;
c += 5; // equivalent to c = c + 5, c is now 15
```

- **Comparison Operators:**

```
let a = 5;
let b = 2;

let isEqual = a == b; // false
let isNotEqual = a != b; // true
```

```
let isGreaterThan = a > b; // true
let isLessThan = a < b; // false
```

- **Logical Operators:**

```
(5 > 0 &&
  (2 > 0)(
    // true
    5 < 0
  )) ||
  2 > 0; // true
!(5 > 0); // false
```

Logic Statements:

- **If and If-Else Statements:**

  Imagine deciding what to wear based on the weather. If it's raining, you wear a raincoat. Otherwise (else), you wear sunglasses.

```
let isRaining: boolean = true;

if (isRaining) {
  console.log("Wear a raincoat.");
} else {
  console.log("Wear sunglasses.");
}
```

Conditional Statements in TypeScript

If and If-Else Statements:

- **If:** Checking if it's raining:

```
let isRaining = true;

if (isRaining) {
  console.log("Wear a raincoat.");
} else {
  console.log("Wear sunglasses.");
}

isRaining = false;

if (isRaining) {
  console.log("Wear a raincoat.");
} else {
  console.log("Wear sunglasses.");
}
```

- **Else If:** Extending the decision-making process:

```
let weather = "cloudy";
```

```
if (weather === "raining") {
  console.log("Wear a raincoat.");
} else if (weather === "cloudy") {
  console.log("Wear a light jacket.");
} else {
  console.log("Wear sunglasses.");
}
```

Conditional Ternary Operators:

Deciding on a snack:

```
let isHungry = true;

let snack = isHungry ? "apple" : "water";

console.log(`You should have some ${snack}.`);

isHungry = false;

snack = isHungry ? "apple" : "water";

console.log(`You should have some ${snack}.`);
```

Switch Statements:

Choosing what to do on a day off based on the day:

```
let dayOff = "Sunday";

switch (dayOff) {
  case "Saturday":
    console.log("Go hiking.");
    break;
  case "Sunday":
    console.log("Read a book.");
    break;
  default:
    console.log("Work on a hobby.");
}
```

Self-Check Quiz:

A simple quiz that evaluates answers using if-else statements:

```
let answer: string = "correct";

if (answer === "correct") {
  console.log("You got it right!");
} else {
  console.log("Sorry, that's not correct.");
}
```

Evaluating a Number Game:

A simple game where the user guesses if a number is high, low, or equal to a target number.

Number Guessing Game:

A simple game to guess a number:

```
let guess: number = 7;
let target: number = 5;

if (guess < target) {
  console.log("Your guess is too low.");
} else if (guess > target) {
  console.log("Your guess is too high.");
} else {
  console.log("You guessed correctly!");
}
```

Friend Checker Game:

A game to determine if someone is a friend based on their name:

```
let isFriend: string = "Ameen";

if (isFriend === "Ameen" || isFriend === "Daniyal") {
  console.log(`${isFriend} is your friend.`);
} else {
  console.log(`${isFriend} is not your friend.`);
}

isFriend = "Hamzah";

if (isFriend === "Ameen" || isFriend === "Daniyal") {
  console.log(`${isFriend} is your friend.`);
} else {
  console.log(`${isFriend} is not your friend.`);
}
```

Using Inquirer to interactively check if someone is a friend:

```
import inquirer from "inquirer";

let friendCheck = await inquirer.prompt([
  {
    name: "name",
    type: "string",
    message: "Enter your friend's name:",
  },
]);

if (friendCheck.name === "Ameen" || friendCheck.name === "Daniyal") {
```

```
  console.log(`${friendCheck.name} is your friend.`);
} else {
  console.log(`${friendCheck.name} is not your friend.`);
}
```

Rock Paper Scissors Game:

A simple implementation of Rock, Paper, Scissors:

```
let player1: string = "Rock";
let player2: string = "Scissors";

if (player1 === player2) {
  console.log("It's a tie!");
} else if (
  (player1 === "Rock" && player2 === "Scissors") ||
  (player1 === "Scissors" && player2 === "Paper") ||
  (player1 === "Paper" && player2 === "Rock")
) {
  console.log("Player 1 wins!");
} else {
  console.log("Player 2 wins!");
}
```

Homework: Create a Calculator:

Create a calculator using conditional statements, operators, template literals, Inquirer, and manage the logic efficiently.

```
import inquirer from "inquirer";

async function startCalculator() {
  let response = await inquirer.prompt([
    { name: "num1", type: "number", message: "Enter the first number:" },
    {
      name: "operation",
      type: "list",
      choices: ["+", "-", "*", "/", "**", "%"],
      message: "Choose an operation:",
    },
    { name: "num2", type: "number", message: "Enter the second number:" },
  ]);

  let result;
  switch (response.operation) {
    case "+":
      result = response.num1 + response.num2;
      break;
    case "-":
      result = response.num1 - response.num2;
      break;
    case "*":
      result = response.num1 * response.num2;
```

```
      break;
    case "/":
      result = response.num1 / response.num2;
      break;
    case "**":
      result = response.num1 ** response.num2;
      break;
    case "%":
      result = response.num1 % response.num2;
      break;
  }

  console.log(`The result is ${result}`);
}

startCalculator();
```

This script prompts the user for two numbers and an operation, then calculates and displays the result based on their choices.

Functions in TypeScript

Functions are a fundamental building block of any application in JavaScript.

Basic Functions:

A function to make a half-fried egg:

```
function halfFryEgg(): number {
  let cooked = 1 + 1.5 + 2; // Egg + Butter + Salt
  return cooked;
}

// Invoking the function
let response: number = halfFryEgg();
console.log(response);
```

Parameters and Arguments:

A function is like a recipe:

- **Recipe:** The function name.
- **Ingredients:** The parameters.
- **Steps:** The code inside the function.

Parameters are like ingredients, such as "butter" and "salt." The actual values passed to the function are the arguments:

```
function halfFryEgg(egg: number, butter: number, salt: number): number {
  return egg + butter + salt;
}
```

```
let response: number = halfFryEgg(1, 1.5, 2);
console.log(response);
```

Another function example to add numbers:

```
function addNumbers(a: number, b: number): number {
  return a + b;
}

let response = addNumbers(5, 3);
console.log(response);
```

A function to calculate the area of a rectangle:

```
function calculateArea(width: number, height: number): number {
  return width * height;
}

let response: number = calculateArea(100, 50);
console.log(response);
```

Default Parameters:

If you forget an ingredient, the function has a backup:

```
function halfFryEgg(
  egg: number = 1,
  butter: number = 1.5,
  salt: number = 2
): number {
  return egg + butter + salt;
}

let response: number = halfFryEgg();
console.log(response);
```

Rest Parameters:

Accepting an unknown number of ingredients:

```
function halfFryEgg(egg: number = 1, ...ingredients: number[]): number {
  console.log(egg);
  console.log(ingredients);
  return egg + ingredients.reduce((acc, val) => acc + val, 0);
}

halfFryEgg(1, 1.5, 2, 5);
```

Spread Operator:

Spreading out ingredients:

```
function halfFryEgg(egg: number = 1, ...ingredients: number[]): number {
  console.log(egg);
```

```
  console.log(...ingredients);
  return egg + ingredients.reduce((acc, val) => acc + val, 0);
}

halfFryEgg(1, 1.5, 2, 5);
```

Arrow Functions:

A shorthand way of writing a recipe:

```
let halfFryEgg = () => 1 + 1.5 + 3; // egg + butter + salt

let response: number = halfFryEgg();
console.log(response);

let halfFryEgg = (egg: number, butter: number, salt: number): number =>
  egg + butter + salt;

let response: number = halfFryEgg(1, 1.5, 2);
console.log(response);
```

Top AI Tools for YouTubers:

If you're looking to start and grow a YouTube channel in 2024, here are five essential AI tools:

1. **InVideo AI:** This tool can turn a simple text prompt into a complete video, including a script, media, music, subtitles, and even a realistic-sounding voiceover. You can also clone your own voice, saving you over 100 hours when creating videos.

2. **Vocal Remover:** This tool removes vocals from trending audio tracks, allowing you to use them seamlessly for your YouTube Shorts and Reels.

3. **WriteSonic AI:** This AI can write your YouTube scripts and can be trained to write like you, cutting your effort in half.

4. **Merlin AI:** This plugin can be used on your Google Chrome browser to summarize and transcribe YouTube videos directly on the platform.

5. **CleanVoice AI:** This tool removes background noise, filler sounds, and "uhs" and "ums" from your audio and video, making it sound professional.

AI has come a long way, and these tools can help you create high-quality content with ease.

5 Simple Hacks to Read Like a CEO (60 books per year):

An average CEO reads around 60 books per year. Here are some hacks to help you reach that number:

1. **Drop Uninteresting Books:** If a book isn't engaging, don't hesitate to set it aside. Pick up something you genuinely enjoy, making reading a habit.

2. **Set Tiny Goals:** Instead of aiming to read a book per week, set an embarrassingly small goal, like reading half a page before bed. This makes it easier to stay consistent.

3. **Change Your Screensaver:** Set a screensaver that says "Read more" or "Read a book instead." This way, instead of scrolling through social media, you'll be reminded to read.

4. **Read in Multiple Languages:** Reading in different languages not only makes it exciting, but studies show it also speeds up cognitive abilities.

5. **Read in Unusual Places:** Force yourself to read in inconvenient situations, like in the shower. This makes reading in a relaxed setting, like on a sofa by the fireplace, feel like a piece of cake. Here's a refined version of your text, making it clearer and more concise:

---

3 Amazing Websites to Read Books for FREE! | Ishan Sharma #shorts

Here are three websites where you can get any ebook for free:

1. **PDFDrive.com:** It has a vast collection of books that you can download in ePub or PDF format.

2. **OpenLibrary.org:** It offers millions of ebooks that you can download and read in any format you prefer.

3. **Gutenberg.org:** It has a collection of over 60,000 freely available ebooks that you can download and start reading.

Bonus: Check out **LibriVox,** which has over 15,000 audiobooks available for download and listening.

So, which book will you read next? Let me know below, and follow me for more!

Here's a revised version of the TypeScript document, making it clearer and correcting some formatting issues:

Variable Scope:

The current context of code, which determines the accessibility of variables to JavaScript.

Global Variables:

Ingredients available in your entire kitchen.

```
let globalVar = "Accessible everywhere";
```

Local Variables:

Ingredients used within a recipe.

```
function showExample() {
  let localVar = "Accessible only inside this function";
  console.log(globalVar); // Works
}

console.log(localVar); // Error: localVar is not defined
```

Hoisting in JavaScript with `let` and `const` - How it Differs from `var`:

Refer to [this FreeCodeCamp article] for more details.

Anonymous Function:

A way to define a function without giving it a name.

```
let halfFryEgg = function () {
  let cooked = 1 + 1.5 + 2; // Egg + Butter + Salt
  console.log(cooked);
};

halfFryEgg();
```

Immediately Invoked Function Expression (IIFE):

A recipe that you prepare immediately after testing, without planning to make it again.

```
(function () {
  console.log("Runs immediately");
})();
```

Recursive Functions:

A recipe that repeats a step (the function) until the dish is done.

```
function countdown(number: number): void {
  if (number <= 0) {
    console.log("Done!");
    return;
  }
  console.log(number);
  countdown(number - 1);
}

countdown(5);
```

Execution:

To compile and run the TypeScript code:

```
D:\code\typescript\classcode>tsc
D:\code\typescript\classcode>node main.js
```

Output:

```
5
4
3
2
1
Done!
```

I hope this revised version meets your expectations. Let me know if you'd like further adjustments. Here's a revised version of your TypeScript document with improved clarity and structure:

Recursive Function:

A function that calls itself to solve a problem:

```
function factorial(n: number): number {
  if (n === 0) {
    return 1;
  }
  return n * factorial(n - 1);
}

let response = factorial(5);
console.log(response); // 120
```

Nested Functions:

A recipe within a recipe:

```
function outerFunction() {
  const innerFunction = function () {
    console.log("Hello from inside!");
  };
  innerFunction();
}

outerFunction();
```

Callback Functions:

Functions that take other functions as arguments:

```
function processUserInput(callback: (name: string) => void) {
  let name = "Sir Ameen Alam";
  callback(name);
}

processUserInput(function (name: string) {
  console.log("Hello, " + name);
});
```

Set Timeout Order:

Scheduling tasks like setting a timer for baking:

```
setTimeout(() => {
  console.log("Cake is ready!");
}, 2000); // Waits 2 seconds
```

Self-Check Quiz:

Creating a function that quizzes the user and checks the answer:

```
import inquirer from "inquirer";

let input1 = await inquirer.prompt([
  {
    name: "userAnswer",
    type: "string",
    message: "What is the capital of France?",
  },
]);

function quiz(question: string, correctAnswer: string) {
  if (input1.userAnswer.toLowerCase() === correctAnswer.toLowerCase()) {
    console.log("Correct!");
  } else {
    console.log("Wrong answer. Try again.");
  }
}

quiz("What is the capital of France?", "Paris");
```

I hope this refined version meets your needs. Let me know if you'd like further adjustments or additions. Here's a revised version of your document with a clear and organized presentation for the TypeScript code:

Homework Assignments:

1. **Basic Function Creation:**

Create a function named `calculateProduct` that takes two parameters, multiplies them together, and returns the result.

```
function calculateProduct(a: number, b: number): number {
  return a * b;
}

const result = calculateProduct(5, 10);
console.log(result); // Should print 50
```

---

2. **Using Default Parameters:**

Define a function `greet` that takes two parameters, `name` and `greeting`, where `greeting` has a default value of "Hello". The function should return a greeting message.

```
function greet(name: string, greeting: string = "Hello"): string {
  return `${greeting}, ${name}!`;
}

console.log(greet("Ameen")); // Should print "Hello, Ameen!"
console.log(greet("Zia", "Hi")); // Should print "Hi, Zia!"
```

3. **Arrow Function Conversion:**

Convert the following traditional function into an arrow function:

```
const add = (a: number, b: number): number => a + b;
```

---

4. **Implementing a Rest Parameter:**

Write a function `sumAll` that uses a rest parameter to take any number of arguments and returns their sum.

```
function sumAll(...numbers: number[]): number {
  return numbers.reduce((acc, num) => acc + num, 0);
}

console.log(sumAll(1, 2, 3)); // Should print 6
console.log(sumAll(10, 20, 30, 40, 50)); // Should print 150
```

---

5. **Function Returning Another Function:**

Create a function `multiplier` that takes a number as its argument and returns another function. The returned function should take a single number as its argument and return the product of its argument and the argument of the first function.

```
function multiplier(x: number) {
  return (y: number) => x * y;
}

const triple = multiplier(3);
console.log(triple(5)); // Should print 15
```

---

6. **Recursive Function - Factorial:**

Write a recursive function to calculate the factorial of a number. The factorial of a number `n` is the product of all positive integers less than or equal to `n`.

```
function factorial(n: number): number {
  if (n === 0) return 1;
  return n * factorial(n - 1);
}

console.log(factorial(5)); // Should print 120
```

---

7. **Nested Functions - Scoping:**

Write a function that contains two nested functions. The outer function should accept a parameter `x`, and its nested functions should increment and then triple `x`. The outer function should return the result of the tripled value after incrementing.

```
function outerFunction(x: number): number {
  function increment() {
    return x + 1;
  }
```

```
  function triple(y: number): number {
    return y * 3;
  }

  const incremented = increment();
  return triple(incremented);
}

console.log(outerFunction(4)); // Should print 15
```

---

8. **Anonymous Function and Callbacks:**

Create an anonymous function that takes an array of numbers and a callback function. The anonymous function should apply the callback function to each element of the array and return a new array with the results.

```
function applyCallback(
  arr: number[],
  callback: (n: number) => number
): number[] {
  return arr.map(callback);
}

const numbers = [1, 2, 3];
const doubledNumbers = applyCallback(numbers, (x) => x * 2);
console.log(doubledNumbers); // Should print [2, 4, 6]
```

---

9. **Set Timeout Exercise:**

Use `setTimeout` within a function to simulate a delay in processing (e.g., retrieving data from a database). The function should accept a callback and invoke it after a delay of 2 seconds.

```
function simulateDelay(callback: () => void): void {
  setTimeout(callback, 2000);
}

simulateDelay(() => console.log("Data retrieved"));
```

Submission Instructions:

- Write your TypeScript code in a clear and organized manner.
- Comment on your code to explain your logic and the steps you've taken to solve each problem.
- After completing the exercises, review your solutions to ensure they meet the requirements and test them to ensure they work as expected.

Let me know if you'd like further adjustments or additions. Here's a revised version of your TypeScript document with improvements for clarity and organization:

Basic Objects in TypeScript:

Think of an object as a file cabinet where each drawer is labeled and contains specific information. In TypeScript, an object is a collection of key-value pairs, where each key (also known as a property) is associated with a value. This structure is similar to the file cabinet, where each drawer's label is a key, and the contents of the drawer are the value.

Defining a Basic Object:

```
let person = {
  name: "Ameen Alam",
};

console.log(person.name); // Output: Ameen Alam
```

Defining a Detailed Object:

```
let person: { name: string; age: number; address: string } = {
  name: "Ameen Alam",
  age: 24,
  address: "123 ABC Street",
};

console.log(person.name); // Output: Ameen Alam
console.log(person.age); // Output: 24
console.log(person.address); // Output: 123 ABC Street
```

Modifying Properties:

```
person.age = 18;
console.log(person.age); // Output: 18
```

Adding New Properties:

```
// person.email = "alice@example.com";
// Error: Property 'email' does not exist on type 'person'
```

Type Alias in TypeScript:

Imagine you have a favorite smoothie recipe that includes a specific combination of ingredients: bananas, strawberries, and almond milk. Instead of listing these ingredients every time you talk about your favorite smoothie, you simply start calling it "MySmoothie." Here, "MySmoothie" is a nickname or alias for the combination of bananas, strawberries, and almond milk.

Type Alias Code:

```
type User = {
  name: string;
  age: number;
  hasPet: boolean;
};
```

```
// Using the 'User' type alias to define objects
let user1: User = {
  name: "Ameen Alam",
  age: 24,
  hasPet: true,
};

type Operation = (x: number, y: number) => number;

const add: Operation = (x, y) => x + y;
const subtract: Operation = (x, y) => x - y;

console.log(add(5, 3)); // Output: 8
console.log(subtract(10, 4)); // Output: 6
```

Type Literals:

Imagine you go to a coffee shop where you can only order drinks in three specific sizes: "Small", "Medium", or "Large". You cannot order a "Mega" or "Mini" because the shop doesn't recognize those sizes. In this scenario, the drink sizes offered are like type literals in programming—specific, predefined values that are acceptable.

Type Literal Code:

```
let drinkSize: "Small" | "Medium" | "Large";

drinkSize = "Medium"; // Valid
drinkSize = "Small"; // Valid

drinkSize = "Mega";
// Error: Type '"Mega"' is not assignable to type '"Small" | "Medium" | "Large"'
```

Type Unions:

A union type allows a variable to take on one of several defined types, making it easier to manage variables with varying structures.

I hope this refined version meets your needs. Let me know if you'd like further adjustments or additions. Here's a revised version of your document with improved clarity and structure:

Union Types:

A Real-World Analogy:

Imagine you're packing for a vacation where you plan to spend time both at the beach and hiking in the mountains. You decide to bring items useful for either activity: sunglasses (for the beach), sunscreen (for both), and a water bottle (for the mountains).

This bag represents a **Union type**: it can contain items useful for the beach, the mountains, or both, offering flexibility in different scenarios.

Union Types: TypeScript Code:

```
let mixedBag: string | number;

mixedBag = "Sunscreen"; // OK
mixedBag = 30; // OK, maybe representing the SPF of the sunscreen

// mixedBag = true; // Error: Type 'boolean' is not assignable to type 'string |
```

Intersection Types:

A Real-World Analogy:

Now, think about a multi-tool that you're bringing along on your trip. This multi-tool includes a knife for hiking and a bottle opener for the beach. It serves a purpose in both scenarios simultaneously.

This represents an **Intersection type**: it combines features from multiple sources into one entity, meeting several criteria at once.

Intersection Types: TypeScript Code:

```
type BeachGear = {
  sunscreen: boolean;
  towel: boolean;
};

type MountainGear = {
  waterBottle: boolean;
  map: boolean;
};

type AdventureGear = BeachGear & MountainGear;

let myGear: AdventureGear = {
  sunscreen: true,
  towel: true,
  waterBottle: true,
  map: true,
};
```

Arrays and Their Properties:

Imagine an array as a row of mailboxes along a street. Each mailbox is assigned a number (its index) and contains mail (values). Just as you can add, remove, or check mail in these mailboxes, you can do the same with values in an array. The properties of an array, such as its length, help you understand how many items are in the array.

Arrays: TypeScript Code:

```
let fruits = ["Apple", "Banana", "Cherry"];

console.log(fruits.length); // 3
console.log(fruits[1]); // "Banana" - Accessing the second element (index starts

let fruitsArray: string[] = ["Apple", "Banana", "Cherry"];
console.log(fruitsArray.length); // 3
console.log(fruitsArray[1]); // "Banana"
```

Array Methods:

Arrays come with a toolbox of methods to manipulate their contents, such as adding or removing items:

```
let colors: string[] = ["Red", "Green", "Blue"];

colors.push("Yellow"); // Adds "Yellow" to the end
colors.pop(); // Removes the last element ("Yellow")
colors.shift(); // Removes the first element ("Red")
colors.unshift("Purple"); // Adds "Purple" to the start

console.log(colors); // ["Purple", "Green", "Blue"]
```

I hope this revised version meets your needs. Let me know if you'd like further adjustments or additions.

Here's a revised version of the TypeScript document, organized with clarity:

Multidimensional Arrays:

A multidimensional array is like a chest of drawers, where each drawer contains another set of compartments. For instance, one drawer might be for socks, with each compartment holding pairs of socks of different colors.

```
let matrix: number[][] = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
];

console.log(matrix[1][2]); // Output: 6 - Accessing the third element in the sec
```

Working with Objects and Arrays:

Imagine a library system where each book is represented by a card containing details about the book (like an object) such as its title, author, and ISBN. All these cards are then organized into a card catalog (an array), allowing you to find any book by looking through the cards.

```
type Book = {
  title: string;
  author: string;
```

```
  isbn: string;
};

let library: Book[] = [
  { title: "The Hobbit", author: "J.R.R. Tolkien", isbn: "123456789" },
  { title: "1984", author: "George Orwell", isbn: "987654321" },
];

// Adding a new book to the array
library.push({
  title: "The Catcher in the Rye",
  author: "J.D. Salinger",
  isbn: "1112131415",
});

// Finding a book by author
let foundBook = library.find((book) => book.author === "George Orwell");
console.log(foundBook); // Output: { title: "1984", author: "George Orwell", isb
```

Homework Assignments:

1. **Basic Array Operations:**

Create an array called `fruits` containing the names of four different fruits. Perform the following operations:

- Add a new fruit to the end of the array.
- Remove the first fruit from the array.
- Add a new fruit to the beginning of the array.
- Find the index of a fruit and remove it using the index.

```
let fruits = ["Apple", "Banana", "Cherry", "Mango"];

// Add a new fruit to the end
fruits.push("Pineapple");

// Remove the first fruit
fruits.shift();

// Add a new fruit to the beginning
fruits.unshift("Strawberry");

// Find and remove a fruit by index
let index = fruits.indexOf("Cherry");
if (index !== -1) fruits.splice(index, 1);

console.log(fruits);
```

---

2. **Working with Multidimensional Arrays:**

Define a 3x3 matrix of numbers as a multidimensional array. Write functions to:

- Print the diagonal elements of the matrix.

- Calculate the sum of all elements in the matrix.

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9],
];

function printDiagonal(matrix: number[][]): void {
  for (let i = 0; i < matrix.length; i++) {
    console.log(matrix[i][i]);
  }
}

function sumMatrix(matrix: number[][]): number {
  return matrix.reduce(
    (total, row) => total + row.reduce((rowTotal, num) => rowTotal + num, 0),
    0
  );
}

printDiagonal(matrix); // Prints 1, 5, 9
console.log(sumMatrix(matrix)); // Output: 45
```

---

3. **Filtering and Searching in Arrays of Objects:**

Given an array of objects where each object represents a book with properties `title`, `author`, and `yearPublished`, write functions to:

- Filter books published after the year 2000.
- Search for books by a specific author.

```
type Book = {
  title: string;
  author: string;
  yearPublished: number;
};

let books: Book[] = [
  { title: "The Hobbit", author: "J.R.R. Tolkien", yearPublished: 1937 },
  { title: "1984", author: "George Orwell", yearPublished: 1949 },
  {
    title: "The Catcher in the Rye",
    author: "J.D. Salinger",
    yearPublished: 1951,
  },
  {
    title: "Harry Potter and the Philosopher's Stone",
    author: "J.K. Rowling",
    yearPublished: 1997,
  },
  { title: "The Hunger Games", author: "Suzanne Collins", yearPublished: 2008 },
];
```

```
function filterBooksByYear(year: number): Book[] {
  return books.filter((book) => book.yearPublished > year);
}

function findBooksByAuthor(author: string): Book[] {
  return books.filter((book) => book.author === author);
}

console.log(filterBooksByYear(2000)); // Output: [{ title: "The Hunger Games", .
console.log(findBooksByAuthor("J.R.R. Tolkien")); // Output: [{ title: "The Hobb
```

---

4. **Using Array Methods:**

Create an array of numbers. Using array methods, perform the following tasks:

- Create a new array with the squares of each number.
- Filter out all numbers greater than 50.
- Use the `reduce` method to find the sum of all numbers in the array.

```
let numbers = [4, 8, 12, 25, 30, 50];

let squares = numbers.map((num) => num * num);
let filteredNumbers = numbers.filter((num) => num <= 50);
let sum = numbers.reduce((acc, num) => acc + num, 0);

console.log(squares); // Output: [16, 64, 144, 625, 900, 2500]
console.log(filteredNumbers); // Output: [4, 8, 12, 25, 30, 50]
console.log(sum); // Output: 129
```

---

5. **Advanced: Working with Nested Arrays and Objects:**

Consider an array of objects where each object represents a student. Each student object has a `name`, `id`, and an array of `grades`. Write a function that calculates the average grade for each student and adds it as a new property `averageGrade` to each student object.

```
type Student = {
  name: string;
  id: string;
  grades: number[];
  averageGrade?: number;
};

let students: Student[] = [
  { name: "John", id: "A001", grades: [85, 90, 78] },
  { name: "Jane", id: "A002", grades: [92, 88, 95] },
  { name: "Alex", id: "A003", grades: [70, 80, 65] },
];

function calculateAverageGrade(student: Student): void {
  student.averageGrade =
    student.grades.reduce((acc, grade) => acc + grade, 0) /
    student.grades.length;
}
```

```
students.forEach(calculateAverageGrade);

console.log(students);
```

Submission Instructions:

- Write your TypeScript code in a clear and organized manner.
- Comment on your code to explain your logic and the steps you've taken to solve each problem.
- After completing the exercises, review your solutions to ensure they meet the requirements and test them to ensure they work as expected.

Here's a revised version of your TypeScript document, making it clearer and better organized:

Tuples:

Imagine you're ordering a coffee and need to specify both the type of coffee and the size in a single order. This pair of information (coffee type, size) can be thought of as a tuple, which is a fixed collection of elements that may be of different types.

```
let coffeeOrder: [string, string] = ["Cappuccino", "Medium"]; // Tuple: [Coffee
```

Enums:

Enums provide a way to define named constants:

```
enum CoffeeType {
  Espresso,
  Latte,
  Cappuccino,
  Americano,
}

let myCoffee: CoffeeType = CoffeeType.Latte;
console.log(myCoffee); // Output: 1
```

While Loop:

Imagine you're waiting for a bus. You keep checking every minute until the bus arrives. This repeated checking is like a while loop, where you continue doing something until a condition is met:

```
let minutesUntilBusArrives = 5;

while (minutesUntilBusArrives > 0) {
  console.log(`Bus arrives in ${minutesUntilBusArrives} minutes.`);
  minutesUntilBusArrives--;
}
```

Output:

```
Bus arrives in 5 minutes.
Bus arrives in 4 minutes.
Bus arrives in 3 minutes.
Bus arrives in 2 minutes.
Bus arrives in 1 minute.
```

Do-While Loop:

Even if you arrive just as the bus pulls up, you still check at least once to see if it's there. A do-while loop ensures the action is performed at least once and continues if the condition is true:

```
let minutesUntilBusArrives = 5;

do {
  console.log("Checking for the bus...");
  minutesUntilBusArrives--;
} while (minutesUntilBusArrives > 0);
```

Output:

```
Checking for the bus...
Checking for the bus...
Checking for the bus...
...
```

With a Function:

```
let minutesUntilBusArrives = 5;

function checkBusArrival(): boolean {
  minutesUntilBusArrives--;
  return minutesUntilBusArrives > 0;
}

do {
  console.log("Checking for the bus...");
} while (checkBusArrival());
```

Output:

```
Checking for the bus...
Checking for the bus...
Checking for the bus...
Checking for the bus...
false
```

For Loop:

Imagine you're watering plants in a row. You start at one end and water each plant until you reach the other end. This process is similar to a for loop, where you perform an action for each item in a sequence:

```
for (let i = 0; i < 5; i++) {
  console.log(`Watering plant ${i + 1}`);
}
```

Output:

```
Watering plant 1
Watering plant 2
Watering plant 3
Watering plant 4
Watering plant 5
```

For-In Loop:

Imagine you have a keyring with keys for different doors. You look at each key on the keyring to find the one you need. A for-in loop in programming is like examining each key (property) in an object:

```
let person: { name: string; age: number; city: string } = {
  name: "Alice",
  age: 30,
  city: "Wonderland",
};

for (let key in person) {
  console.log(`${key}: ${person[key]}`);
}
```

Output:

```
name: Alice
age: 30
city: Wonderland
```

Troubleshooting:

If you encounter an error stating that an index signature is not found:

```
// Element implicitly has an 'any' type because expression of type 'string' can'
```

Ensure the object's type is explicitly defined as shown above.

For-Of Loop:

Consider having a box of chocolates. You take out each chocolate one by one to see what kind it is. A for-of loop is used to go through each item in an array (or any iterable), similar to checking each chocolate:

```
let flavors = ["Vanilla", "Chocolate", "Strawberry", "Mint"];

for (let flavor of flavors) {
  console.log(flavor);
}
```

Output:

```
Vanilla
Chocolate
Strawberry
Mint
```

Homework Assignments:

1. **Loop through an Array with For-Of:**

Create an array of your favorite movies. Write a function that uses a for-of loop to print each movie to the console.

- Example function to print each movie.

2. **Enumerate Properties with For-In Loop:**

Given an object representing a car with properties like make, model, and year, write a function that uses a for-in loop to print each property name and its value.

- Example function to print car properties.

3. **Practicing Do-While Loop:**

Create a function that simulates a simple guessing game. This function should generate a random number between 1 and 10 and then prompt the user to guess the number. Use a do-while loop to keep asking them to guess again until they get it right.

- Note: Since TypeScript runs on Node.js for this task, consider using pseudo-code or describe the logic, as `prompt` is not available in standard Node.js without additional packages.

4. **While Loop for a Countdown:**

Write a function that takes a number as an argument and counts down to zero using a while loop, printing each number to the console.

- Example countdown function.

5. **Enums for Days of the Week:**

Define an enum for days of the week. Write a function that takes a day as an argument and returns "Weekend" if it's Saturday or Sunday, and "Weekday" for other days.

- Example enum for days and function.

6. **Tuples for RGB Colors:**

Define a tuple type for RGB color values. Write a function that takes an RGB tuple as an argument and returns a string describing the color.

- Example RGB tuple and function.

Example Solutions:

1. **Loop through an Array with For-Of:**

```
const favoriteMovies: string[] = ["Inception", "The Matrix", "Interstellar"];

function printMovies(movies: string[]): void {
  for (const movie of movies) {
    console.log(movie);
  }
}

printMovies(favoriteMovies);
```

---

2. **Enumerate Properties with For-In Loop:**

```
const car = {
  make: "Toyota",
  model: "Camry",
  year: 2020,
};

function printCarDetails(car: { [key: string]: string | number }): void {
  for (const key in car) {
    console.log(`${key}: ${car[key]}`);
  }
}

printCarDetails(car);
```

---

3. **Basic For Loop Exercise (FizzBuzz):**

```
function fizzBuzz(): void {
  for (let i = 1; i <= 100; i++) {
    let output = "";
    if (i % 3 === 0) output += "Fizz";
    if (i % 5 === 0) output += "Buzz";
    console.log(output || i);
  }
}

fizzBuzz();
```

Submission Instructions:

For each exercise, ensure your TypeScript code is well-commented to explain your logic. Adhere to best practices, such as using `const` for variables that won't change and `let` for those that will. Test your functions to make sure they work as expected. This set of homework assignments is designed to solidify your understanding of TypeScript's control flow mechanisms, enums, and tuples, as well as encourage thoughtful problem-solving and code organization.

Modules:

Basics of modules, exporting, and importing modules:

- When we transpile this program, it runs correctly.
- However, note that the transpiled JavaScript code does not use the ES Module syntax, but rather the old commonjs syntax.

Modules in Learn-TypeScript Repo:

- Learn TypeScript Repo: Step 03a Modules

ECMAScript Modules in Node.js:

- When we transpile this program, it runs correctly.
- However, note that the transpiled JavaScript code does not use the ES Module syntax, but rather the old commonjs syntax.

Native ECMAScript Modules:

- Using ES modules in TypeScript.
- Example: Importing third-party modules, such as `inquirer`.
