Typescript :

Topic : Destructuring

- Create an object `const person = { name: 'Alice', age: 30, city: 'New York' }`.
- Use object destructuring to extract the `name` and `age` properties into variables `name` and `age`.
- Print `name` and `age`.

Topic : Operator

- **Arithmetic Operators:**
  - Create two variables `a = 10` and `b = 3`.
  - Perform addition, subtraction, multiplication, division, and modulus operations using these variables.
  - Print the results of each operation.
- **Comparison Operators:**
  - Compare the variables `a` and `b` using ==, ===, `!=`, `!==`, >, <, >=, and <= operators.
  - Print the result of each comparison.
- **Logical Operators:**
  - Create two boolean variables `x = true` and `y = false`.
  - Use the `&&` (AND), `||` (OR), and `!` (NOT) operators with `x` and `y`.
  - Print the results.
- **Bitwise Operators:**
  - Use the bitwise AND (`&`), OR (`|`), XOR (`^`), and NOT (`~`) operators on `a` and `b`.
  - Use the left shift (<<) and right shift (>>) operators on `a`.
  - Print the results.
- **Assignment Operators:**
  - Create a variable `c = 5`.
  - Use +=, -=, `*=`, `/=`, and `%=` operators to modify `c` using `b`.
  - Print the result of `c` after each operation.
- **Ternary Operator:**

- Write an expression using the ternary operator to check if `a` is greater than `b`. If true, assign the result to a variable `result` with the value "a is greater", otherwise "b is greater or equal".
- Print the value of `result`.

- **Nullish Coalescing Operator:**

  - Create a variable `d` and set it to `undefined`.
  - Use the nullish coalescing operator (`??`) to assign a default value `"default value"` to a new variable `finalValue` if `d` is `null` or `undefined`.
  - Print `finalValue`.

- **Optional Chaining Operator:**

  - Create an object `const person = { name: 'Alice', address: { city: 'New York' } }`.
  - Use the optional chaining operator to safely access the `city` and `zipCode` properties of the `address` object.
  - Print the values (or `undefined` if `zipCode` does not exist).

- **Typeof and Instanceof Operators:**

  - Use the `typeof` operator to check the types of variables `a`, `x`, and `person`.
  - Use the `instanceof` operator to check if `person` is an instance of the `Object` class.
  - Print the results.

- **Advanced - Combining Operators:**

  - Write a small function `calculate(a: number, b: number): number` that:
    1. Multiplies `a` by `b` using the `*` operator.
    2. If the result is greater than 100, use the ternary operator to return 100.
    3. Otherwise, return the result.
  - Call this function with different values of `a` and `b`, and print the results.

Topic : Conditioanal Statement

- **Basic Ternary Operator:**

  - Create two variables `a = 15` and `b = 10`.
  - Use the ternary operator to compare `a` and `b`. If `a` is greater than `b`, assign the string `"a is greater"` to a variable `result`, otherwise `"b is greater or equal"`.
  - Print the value of `result`.

- **Nested Ternary Operator:**

  - Extend the previous example. Use a nested ternary operator to check if `a` is greater than `b`, if `b` is greater than 5, and assign `"a is greater and b is greater than 5"` to `result`. Otherwise, check if `b` is less than or equal to 5, and assign `"a is greater but b is 5 or less"`. If `a` is not greater than `b`, assign `"b is greater or equal"`.
  - Print the value of `result`.

- **Basic `if...else` Statement:**

  - Create a variable `score = 85`.
  - Write an `if...else` statement to print `"Pass"` if the score is 60 or above, and `"Fail"` if it is below 60.

- **Multiple `if...else if` Conditions:**

  - Modify the `score` example to include multiple conditions:
    - If `score` is 90 or above, print `"Grade A"`.
    - If `score` is between 80 and 89, print `"Grade B"`.
    - If `score` is between 70 and 79, print `"Grade C"`.
    - If `score` is between 60 and 69, print `"Grade D"`.
    - If `score` is below 60, print `"Grade F"`.

- **Switch Statement:**

  - Create a variable `day = 3`.
  - Write a `switch` statement to determine the day of the week based on the value of `day`:
    - 1 for `"Monday"`, 2 for `"Tuesday"`, 3 for `"Wednesday"`, 4 for `"Thursday"`, 5 for `"Friday"`, 6 for `"Saturday"`, 7 for `"Sunday"`.
  - Print the day of the week.

- **Combining Conditions:**

  - Create a function `checkEligibility(age: number, isEmployed: boolean): string` that:
    - Returns `"Eligible for loan"` if `age` is greater than 18 and `isEmployed` is `true`.
    - Returns `"Not eligible for loan"` otherwise.
  - Use an `if...else` statement to implement this logic.
  - Test the function with different values and print the results.

- **Ternary Operator with Function Call:**

  - Write a function `getDiscount(isMember: boolean): number` that:
    - Returns `10` if `isMember` is `true`, otherwise returns `0`.
  - Use the ternary operator to call this function based on a boolean variable `isCustomerMember = true` and assign the result to a variable `discount`.
  - Print the value of `discount`.

- **Advanced - Conditional Types (TypeScript Specific):**

  - Create a function `checkType<T>(value: T): string` that:
    - Uses conditional types to return `"string"` if the type of `value` is a string, `"number"` if it is a number, and `"other"` for any other type.
  - Use the ternary operator in conjunction with `typeof` to implement this logic.
  - Test the function with different types of values (e.g., `checkType('Hello')`, `checkType(42)`, `checkType(true)`).

Topic : Looping Statement

- **Basic `for` Loop:**

  - Write a `for` loop that prints numbers from `1` to `10` to the console.

- **`for` Loop with Array:**

  - Create an array `const fruits = ['apple', 'banana', 'cherry', 'date'].`
  - Use a `for` loop to print each fruit in the array.

- **`while` Loop:**

  - Write a `while` loop that prints numbers from `10` down to `1`.
  - Ensure that the loop terminates correctly.

- **`do...while` Loop:**

  - Write a `do...while` loop that prints numbers from `1` to `5`.
  - Ensure that the loop body executes at least once, even if the condition is initially false.

- **`for...of` Loop:**

  - Create an array of objects `const users = [{ name: 'Alice', age: 25 }, { name: 'Bob', age: 30 }, { name: 'Charlie', age: 35 }].`
  - Use a `for...of` loop to print the `name` and `age` of each user.

- **`for...in` Loop:**

  - Create an object `const car = { brand: 'Toyota', model: 'Corolla', year: 2021 }.`
  - Use a `for...in` loop to print each key and its corresponding value in the object.

- **Nested `for` Loops:**

  - Write a nested `for` loop to print a multiplication table from `1` to `5`. Each row should represent the multiples of a number (e.g., `1 x 1`, `1 x 2`, ..., `5 x 5`).

- **Loop with Break and Continue:**

  - Write a `for` loop that prints numbers from `1` to `10`.
  - Use the `continue` statement to skip printing the number `5`.
  - Use the `break` statement to stop the loop when the number `8` is reached.

- **`for...of` with String:**

  - Create a string `const message = 'Hello, TypeScript!'`.
  - Use a `for...of` loop to print each character in the string.

- **Advanced - Looping with Conditional Logic:**

  - Create a function `processNumbers(numbers: number[]): void` that:
    - Uses a `for...of` loop to iterate over an array of numbers.
    - If the number is even, print `"Even: [number]"`.
    - If the number is odd, print `"Odd: [number]"`.
    - Use the `continue` statement to skip processing numbers less than `0`.

- **Advanced - `for...in` with Arrays:**

  - Create an array `const colors = ['red', 'green', 'blue']`.
  - Use a `for...in` loop to print each index and the corresponding color.
  - Note that `for...in` loops over the indices of the array, not the elements themselves.

Topic Function :

- **Function Declaration:**

  - Write a function `greet(name: string): string` that takes a `name` parameter and returns a greeting message `"Hello, [name]!"`.
  - Call this function with a sample name and print the result.

- **Function Expression:**

  - Create a function expression `const add = function(x: number, y: number): number` that takes two numbers and returns their sum.
  - Call this function with two numbers and print the result.

- **Arrow Function:**

  - Write an arrow function `multiply = (a: number, b: number): number => a * b` that multiplies two numbers.
  - Call this function with two numbers and print the result.

- **Optional Parameters:**

  - Write a function `printDetails(name: string, age?: number): void` where `age` is an optional parameter.
  - If `age` is provided, print `"Name: [name], Age: [age]"`; otherwise, print `"Name: [name]"`.
  - Call this function with and without the `age` parameter and print the results.

- **Default Parameters:**

  - Write a function `createGreeting(name: string, greeting: string = 'Hello'): string` that returns a greeting message with a default greeting value of `"Hello"`.
  - Call this function with a specific greeting and with no greeting provided, and print the results.

- **Function with Return Type:**

  - Create a function `isEven(number: number): boolean` that returns `true` if the number is even and `false` otherwise.
  - Call this function with different numbers and print whether each number is even.

- **Function Overloading:**

  - Write a function `concatenate(value1: string, value2: string): string` that concatenates two strings.
  - Overload this function to also accept two numbers and return their sum as a string.
  - Test both versions of the function with appropriate inputs and print the results.

- **Rest Parameters:**

  - Write a function `sumNumbers(...numbers: number[]): number` that takes any number of numeric arguments and returns their sum.
  - Call this function with multiple numbers and print the result.

- **Function as a Parameter:**

  - Write a function `applyOperation(x: number, y: number, operation: (a: number, b: number) => number): number` that applies a given operation (e.g., addition or multiplication) to two numbers.
  - Define addition and multiplication functions and pass them to `applyOperation`, then print the results.

- **Recursive Function:**
  - Write a recursive function `factorial(n: number): number` that calculates the factorial of a given number `n`.
  - Call this function with different numbers and print the results.
- **Advanced - Function Types:**
  - Define a type alias for a function `type Comparator = (a: number, b: number) => boolean`.
  - Write a function `sortArray(array: number[], comparator: Comparator): number[]` that sorts an array using the provided comparator function.
  - Test the `sortArray` function with different comparator functions (e.g., ascending and descending order) and print the results.

Topic Class Object and Constructor:

- **Basic Class Definition:**
  - Define a class `Person` with properties `name` (string) and `age` (number).
  - Create a method `introduce` that returns a string `"Hello, my name is [name] and I am [age] years old."`.
  - Instantiate an object of the `Person` class, set its properties, and call the `introduce` method to print the result.
- **Constructor Usage:**
  - Modify the `Person` class to include a constructor that initializes `name` and `age` properties.
  - Create an instance of the `Person` class using the constructor and print the introduction message.
- **Access Modifiers:**
  - Update the `Person` class to use access modifiers:
    - `public` for `name` and `age`.
    - Add a `private` property `socialSecurityNumber`.
  - Create methods `getSocialSecurityNumber` and `setSocialSecurityNumber` to access and modify the `socialSecurityNumber` property.
  - Instantiate the class and demonstrate the use of these methods.
- **Getter and Setter Methods:**
  - Add a `fullName` property to the `Person` class that returns the `name` property in uppercase.
  - Create a setter for `fullName` that updates the `name` property based on the given value.
  - Demonstrate the use of `fullName` getter and setter methods.

- **Inheritance:**
  - Define a subclass `Employee` that extends `Person` and adds a new property `jobTitle` (string).
  - Add a method `getJobDescription` to the `Employee` class that returns `"I am a [jobTitle]"`.
  - Instantiate an `Employee` object, set its properties, and call the `getJobDescription` method.
- **Constructor Inheritance:**
  - Modify the `Employee` class to call the `Person` class constructor using `super()` to initialize the `name` and `age` properties.
  - Demonstrate creating an `Employee` object with all properties initialized through the constructor.
- **Static Methods:**
  - Add a static method `createPerson` to the `Person` class that takes `name` and `age` parameters and returns a new `Person` object.
  - Use this static method to create a new `Person` instance and print the introduction message.
- **Abstract Classes:**
  - Define an abstract class `Animal` with an abstract method `makeSound()`.
  - Create a subclass `Dog` that extends `Animal` and implements `makeSound` to return `"Woof!"`.
  - Instantiate a `Dog` object and call the `makeSound` method.
- **Interfaces and Classes:**
  - Define an interface `Identifiable` with a method `getId(): string`.
  - Implement this interface in the `Person` class by adding a `getId` method that returns a unique identifier.
  - Demonstrate using the `Identifiable` interface to interact with `Person` objects.
- **Advanced - Class Composition:**
  - Create a class `Address` with properties `street`, `city`, and `postalCode`.
  - Update the `Person` class to include an `Address` property.
  - Add methods to the `Person` class to set and get the address details.
  - Instantiate a `Person` object with an `Address` and demonstrate setting and getting address details.

Topic : Encapsulation

- **Basic Encapsulation with Access Modifiers:**
  - Define a class `Account` with private properties `accountNumber` (string) and `balance` (number).
  - Create a public method `deposit(amount: number): void` to increase the balance.
  - Create a public method `withdraw(amount: number): boolean` to decrease the balance if sufficient funds are available; otherwise, return `false`.
  - Create a public method `getBalance(): number` to return the current balance.
  - Demonstrate creating an `Account` object, depositing and withdrawing funds, and checking the balance.

- **Getter and Setter Methods:**
  - Modify the `Account` class to include a getter and setter for `balance`. The setter should include validation to ensure the balance is not negative.
  - Use the getter and setter methods to modify and access the `balance` property.
  - Demonstrate the use of these getter and setter methods.

- **Private and Protected Access:**
  - Update the `Account` class to include a protected method `logTransaction(message: string): void` that logs transaction details.
  - Create a subclass `SavingsAccount` that extends `Account` and uses the `logTransaction` method to log deposits and withdrawals.
  - Demonstrate creating a `SavingsAccount` object and performing transactions.

- **Encapsulation with Interfaces:**
  - Define an interface `AccountOperations` with methods `deposit(amount: number): void`, `withdraw(amount: number): boolean`, and `getBalance(): number`.
  - Implement this interface in the `Account` class.
  - Create an object of type `AccountOperations` and use it to interact with an `Account` instance.

Topic : Association

- **Composition (Strong Association):**
  - Define a class `Engine` with properties `horsepower` (number) and `type` (string).
  - Define a class `Car` that has a private property `engine` of type `Engine` and a property `model` (string).
  - Create a method `start()` in the `Car` class that prints a message including the engine type and horsepower.

  - Demonstrate creating a `Car` object with an `Engine` and calling the `start` method.

- **Aggregation (Weaker Association):**

  - Define a class `Author` with properties `name` (string) and `birthYear` (number).
  - Define a class `Book` with properties `title` (string), `isbn` (string), and `author` (an instance of `Author`).
  - Create a method `getAuthorInfo()` in the `Book` class that returns information about the author.
  - Demonstrate creating a `Book` object with an `Author` and calling `getAuthorInfo`.

- **Association (Loose Coupling):**

  - Define a class `Person` with properties `name` (string) and `age` (number).
  - Define a class `Company` with properties `companyName` (string) and `employees` (array of `Person`).
  - Create methods `addEmployee(person: Person)` and `listEmployees()` in the `Company` class.
  - Demonstrate creating `Person` objects, adding them to a `Company`, and listing the employees.

- **Bidirectional Association:**

  - Define a class `Department` with properties `name` (string) and `employees` (array of `Person`).
  - Modify the `Person` class to include a `department` property (an instance of `Department`).
  - Ensure that when a `Person` is added to a `Department`, the department is set on the `Person` object.
  - Demonstrate creating `Department` and `Person` objects, adding employees to a department, and setting the department for each person.

<u>Topic : Polymorphism</u>

Polymorphism with Method Overriding:

- Define a base class `Shape` with a method `draw()` that returns a string `"Drawing shape"`.
- Define two subclasses `Circle` and `Rectangle` that override the `draw()` method to return `"Drawing circle"` and `"Drawing rectangle"`, respectively.
- Create instances of `Circle` and `Rectangle`, and demonstrate calling the `draw()` method on each object.

Polymorphism with Method Overloading:

- Define a class `Display` with an overloaded `show()` method:
    - `show(message: string): void` to display a string message.
    - `show(value: number): void` to display a number.
- Create instances of `Display` and call the `show()` method with different types of arguments.

Method Overloading in Inheritance:

- Define a base class `Shape` with an overloaded `area()` method:
    - `area(radius: number): number` for circles.
    - `area(width: number, height: number): number` for rectangles.
- Define subclasses `Circle` and `Rectangle` that inherit from `Shape` and use the `area()` method appropriately.
- Demonstrate creating `Circle` and `Rectangle` objects and calculating their area.

Polymorphism with Method Overloading and Interfaces:

- Define an interface `Printer` with overloaded methods `print()`.
    - `print(content: string): void`
    - `print(content: number): void`
- Implement the `Printer` interface in a class `TextPrinter` and provide implementations for both method signatures.
- Demonstrate using `TextPrinter` to print both strings and numbers.