🔢 Arithmetic Function

Defines an arrow function to add two numbers and return the result.

typescript
```
const addNumbers = (num1: number, num2: number): number => {
    return num1 + num2;
};

const result: number = addNumbers(20, 22);
console.log(result);
```

📦 Variable Types

Demonstrates TypeScript's basic types: number, string, boolean, and `any`.

typescript
```
const num1: number = 100;
const num2: number = 10.30;
const str1: string = "100";
const str2: string = '100';
const str3: string = `Manas Pandey Class 12 F`;
const isTrue: boolean = true;
const anyNum: any = 1813749;
const anyStr: any = "1813749";
const anyBool: any = true;

console.log(num1, num2, str1, str2, str3, isTrue, anyNum, anyStr, anyBool);
```

🧾 JSON Object

Creates a simple JSON object and accesses its properties.

typescript
```
const user = {
    name: "Manas",
    email: "Manas@goodManas.com",
    age: 22
};

console.log(user.name, user.email, user.age);
```

📚 Arrays

Shows how to declare and access elements in a string array.

typescript
const people: string[] = ["Manas", "Sanam", "Pandey", "Pandey", "Stylish", "Passed with 43%", "Failed in English"];

people.forEach((person, index) => {
    console.log(`people[${index}]: ${person}`);
});

🎯 Mixed Type Logging

Prints variables of different types in a single statement.

typescript
const age: number = 23;
const name: string = "Masad";
const status: any = true;

console.log(age, name, status);

❓ Uninitialized Variable

Logs an uninitialized variable of type `any`, which results in `undefined`.

typescript
let obj: any;
console.log(obj);

🧮 Functions

Examples of functions with different return types and parameter combinations.

typescript
function add(x: number, y: number): number {
    return x + y;
}
console.log(add(10, 3497249854));

function concat(text: string, value: number): string {
    return text + value;

```
}
console.log(concat("Manas's age is: ", 22));

function logMessage(): void {
    console.log("Equivalent to Syso in Java");
}
logMessage();

function fullName(first: string, last: string): void {
    console.log(`${first} ${last}`);
}
fullName("Manas", "Pandey");
```

⚙️ Default & Optional Parameters

Shows how to use default values and optional parameters in functions.

```
typescript
function applyDiscount(price: number, discount: number = 1): number {
    return price - discount;
}
console.log(applyDiscount(100, 20));

function optionalDiscount(price: number, discount?: number): number {
    return discount ? price - (price * discount / 100) : price;
}
console.log(optionalDiscount(100));
```

🔁 Loops & Rest Parameters

Uses loops and rest parameters to handle dynamic input arrays.

```
typescript
const values: number[] = [1, 2, 3, 4, 0, 5, 6, 7];

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

function printRest(...nums: number[]): void {
    nums.forEach(num => console.log(num));
}
```

```
printRest(1, 2, 3, 4, 5);

function printMixed(...items: any[]): void {
    items.forEach(item => console.log(item));
}
printMixed("Manas", 12, true, "text", false);

function logPurchases(buyer: string, ...items: any[]): void {
    items.forEach(item => {
        console.log(`${buyer} bought: ${item}`);
    });
}
logPurchases("Manas", "coffee", "cake", 12, true);
```

✨ Template Literals

Uses backticks to embed variables inside strings.

typescript

```
function greet(name: string, city: string): void {
    console.log(`${name} is from ${city}`);
}
greet("Manas", "Bengaluru");
```

🏹 Arrow Functions

Defines concise arrow functions for various tasks.

typescript

```
const addValues = (a: number, b: number): number => a + b;
console.log(addValues(10, 343));

const sayHello = (): void => {
    console.log("Hello, World!");
};
sayHello();

const returnHundred = (): number => 100;
console.log(returnHundred());

const multiply = (a: number, b: number): number => a * b;
console.log(multiply(14, 54));
```

🧮 Arrays

Creates and iterates through a number array.

typescript
const numbers: number[] = [1, 2, 3];
numbers.forEach(num => console.log(num));

📅 Enums

Defines an enum for days of the week and accesses its values.

typescript
enum Days {
  Sun,
  Mon,
  Tue,
  Wed,
  Thu,
  Fri,
  Sat
}

console.log(Days);
console.log(Days.Sun, Days.Mon, Days.Tue, Days.Wed, Days.Thu, Days.Fri, Days.Sat);

🌍 Hello World & Scope Variables

Shows `let`, `const`, and `var` differences in block scope and reassignment behavior.

typescript
function testScope(): void {
  if (true) {
    let a: number = 20;
    const b: number = 30;
    var c: number = 40;
    console.log(a, b, c);
    a += 20;
    c += 20;
  }
  // 'a' and 'b' not accessible here due to block scope
  console.log(c); // Only 'var' escapes block
}
testScope();

🏛️ Class Declaration & Method

Defines a class with properties and a method using `this` to access internal data.

typescript
```
class Location {
    id: number;
    city: string;

    display(): void {
        this.id = 10;
        this.city = "Bengaluru";
        console.log(this.id, this.city);
    }
}
const loc = new Location();
loc.display();
```

⚙️ Constructor Usage

Demonstrates different constructor patterns for initializing class objects.

typescript
```
class SimpleConstructor {
    constructor() {
        console.log(1408);
    }
}
new SimpleConstructor();

class Parameterized {
    constructor(id: number, name: string) {
        console.log(`${id} ${name}`);
    }
}
new Parameterized(28, "Manas");
```

🧩 Properties with Constructor

Initializes properties using constructor parameters and accesses them outside the class.

typescript
```
class Profile {
    id: number;
```

```
    name: string;
    extra: number = 10;

    constructor(name: string, id: number) {
        this.id = id;
        this.name = name;
        console.log(id, name);
    }
}
const p = new Profile("Manas", 28);
console.log(p.id, p.name, p.extra);
```

🧠 Instance Access & Method

Accesses properties directly and via class method.

```
typescript
class User {
    name: string = "Mike";
    age: number = 22;

    show(): void {
        console.log(this.name, this.age);
    }
}
const u = new User();
console.log(u.name, u.age);
u.show();
```

📦 Object Literal from Class Interface

Creates an object with properties that mirror a class's structure—without using `new`.

```
typescript
class City {
    id: number;
    name: string;
}

const city = {
    id: 10,
    name: "Bengaluru"
};
console.log(city.id, city.name);
```

🧱 Class with Constructor Property Assignment

Assigns properties via constructor and accesses them post-instantiation.

typescript
```
class Product {
    id: number;
    constructor(id: number) {
        this.id = id;
    }
}
const prod = new Product(100);
console.log(prod.id);
```

📐 Interface & Implementation

Defines an interface and implements it with a class—shows contract enforcement.

typescript
```
interface Greet {
    name: string;
    greet(name: string): void;
}

class Speaker implements Greet {
    name: string;
    greet(name: string): void {
        console.log(name);
        console.log("Custom message here.");
    }
}

const sp = new Speaker();
sp.greet("Manas");
```

🔗 Multiple Interfaces in One Class

Implements two interfaces to demonstrate multi-interface inheritance.

typescript
```
interface Personal {
    firstName: string;
    greet1(name: string): void;
}
```

```
interface Professional {
    lastName: string;
    greet2(id: number): void;
}

class DualRole implements Personal, Professional {
    firstName: string;
    lastName: string;

    greet1(name: string): void {
        console.log(name);
    }

    greet2(id: number): void {
        console.log(id);
    }
}

const role = new DualRole();
role.greet1("Manas");
role.greet2(10);
```

⚡ Static Property & Method

Shows how static members are accessed directly via the class, not instances.

typescript

```
class Example {
    static count: number = 10;
    total: number = 20;
}

console.log(Example.count);
const ex = new Example();
console.log(ex.total);

class Utility {
    static log(): void {
        console.log("From static method");
    }
}
Utility.log();
```

🧬 Inheritance & Overriding

Extends a class and overrides its method to demonstrate polymorphism.

typescript
```
class Base {
    show(): void {
        console.log("Base Method");
    }
}
class Derived extends Base {
    show(): void {
        console.log("Method Overridden");
    }
}
const d = new Derived();
d.show();
```

🏹 Arrow Function Inside Class

Uses an arrow function inside a method to maintain scope and simplify syntax.

typescript
```
class Wrapper {
    show(): void {
        const log = () => {
            console.log(100);
        };
        log();
    }
}
new Wrapper().show();
```