// 1️⃣ Product Type & Discounted Price  
interface Product {  
    id: number;  
    name: string;  
    price: number;  
    isAvailable: boolean;  
}  
function getDiscountedPrice(product: Product, discount: number): number {  
    return product.price - (product.price \* discount) / 100;  
}

// 2️⃣ Union & Intersection Types  
function formatInput(input: string | number | string[]): string {  
    if (typeof input === 'string') {  
        return input.toUpperCase();  
    } else if (typeof input === 'number') {  
        return `$${input}`;  
    } else {  
        return input.join(', ');  
    }  
}

// 3️⃣ Tuples & Enums  
enum DriverStatus {  
    Available,  
    OnTrip,  
    Offline  
}  
type DriverInfo = [number, string, DriverStatus];  
const driver: DriverInfo = [101, "John Doe", DriverStatus.Available];

// 4️⃣ Interfaces & Generics  
interface User {  
    id: number;  
    name: string;  
    role: 'Admin' | 'Customer';  
}  
function getUserDetails<T extends User>(user: T): string {  
    return `User ${user.name} is a ${user.role}`;  
}

// 5️⃣ Class Implementation & Abstract Classes  
abstract class BankAccount {  
    constructor(public accountNumber: number, public balance: number) {}  
    abstract withdraw(amount: number): void;  
}  
class SavingsAccount extends BankAccount {  
    withdraw(amount: number): void {  
        if (this.balance >= amount) {  
            this.balance -= amount;  
            console.log(`Withdrawal successful. New balance: $${this.balance}`);  
        } else {  
            console.log("Insufficient balance");  
        }  
    }  
}  
class CurrentAccount extends BankAccount {  
    withdraw(amount: number): void {  
        if (this.balance - amount >= -500) {  
            this.balance -= amount;  
            console.log(`Withdrawal successful. New balance: $${this.balance}`);  
        } else {  
            console.log("Overdraft limit exceeded");  
        }  
    }  
}

// 6️⃣ Utility Types & Mapped Types  
interface Employee {  
    id: number;  
    name: string;  
    position: string;  
    salary: number;  
}  
type ReadonlyEmployee = Readonly<Employee>;  
type PartialEmployee = Partial<Employee>;  
function updateEmployeeInfo(emp: PartialEmployee) {  
    console.log("Updated Employee Info:", emp);  
}

// 7️⃣ Type Guards & Type Narrowing  
interface TextMessage {  
    type: 'text';  
    content: string;  
}  
interface ImageMessage {  
    type: 'image';  
    imageUrl: string;  
}  
type Message = TextMessage | ImageMessage;  
function isTextMessage(msg: Message): msg is TextMessage {  
    return msg.type === 'text';  
}  
function handleMessage(msg: Message) {  
    if (isTextMessage(msg)) {  
        console.log(`Text: ${msg.content}`);  
    } else {  
        console.log(`Image: ${msg.imageUrl}`);  
    }  
}

// 8️⃣ Decorators & Metadata  
function LogClass<T extends { new (...args: any[]): {} }>(constructor: T) {  
    return class extends constructor {  
        constructor(...args: any[]) {  
            super(...args);  
            console.log(`Class ${constructor.name} is instantiated`);  
        }  
    };  
}  
@LogClass  
class OrderService {  
    constructor() {  
        console.log("OrderService initialized");  
    }  
}  
const orderService = new OrderService();

// 9️⃣ Asynchronous TypeScript & Promises  
async function getWeather(city: string): Promise<number> {  
    return new Promise((resolve, reject) => {  
        setTimeout(() => {  
            const temperature = Math.floor(Math.random() \* 35) + 10; // Simulated temperature  
            resolve(temperature);  
        }, 2000);  
    });  
}  
async function fetchWeather() {  
    try {  
        const nyTemp = await getWeather("New York");  
        console.log(`New York Temperature: ${nyTemp}°C`);  
        const londonTemp = await getWeather("London");  
        console.log(`London Temperature: ${londonTemp}°C`);  
    } catch (error) {  
        console.error("Error fetching weather data", error);  
    }  
}  
fetchWeather();

// 🔟 TypeScript with MongoDB  
interface Product {  
    \_id: string;  
    name: string;  
    price: number;  
    stock: number;  
    category: string;  
}  
async function findProductById(id: string): Promise<Product | null> {  
    return new Promise((resolve) => {  
        setTimeout(() => {  
            const mockDatabase: Product[] = [  
                { \_id: "1", name: "Laptop", price: 1000, stock: 5, category: "Electronics" },  
                { \_id: "2", name: "Phone", price: 500, stock: 10, category: "Electronics" }  
            ];  
            const product = mockDatabase.find(p => p.\_id === id) || null;  
            resolve(product);  
        }, 2000);  
    });  
}  
findProductById("1").then(product => console.log(product));

// 1️⃣ Strongly Typed Variables & Functions  
type Product = {  
    id: number;  
    name: string;  
    price: number;  
    isAvailable: boolean;  
};  
function getDiscountedPrice(product: Product, discount: number): number {  
    return product.price - (product.price \* discount) / 100;  
}

// 2️⃣ Union & Intersection Types  
function formatInput(input: string | number | string[]): string {  
    if (typeof input === "string") {  
        return input.toUpperCase();  
    } else if (typeof input === "number") {  
        return `$${input}`;  
    } else {  
        return input.join(", ");  
    }  
}

// 3️⃣ Tuples & Enums  
enum DriverStatus {  
    Available,  
    OnTrip,  
    Offline  
}  
type DriverInfo = [number, string, DriverStatus];  
const driver: DriverInfo = [1, "John Doe", DriverStatus.Available];

// 4️⃣ Interfaces & Generics  
interface User {  
    id: number;  
    name: string;  
    role: "Admin" | "Customer";  
}  
function getUserDetails<T extends User>(user: T): string {  
    return `User ${user.name} is a ${user.role}`;  
}

// 5️⃣ Class Implementation & Abstract Classes  
abstract class BankAccount {  
    constructor(public accountNumber: number, public balance: number) {}  
    abstract withdraw(amount: number): void;  
}  
class SavingsAccount extends BankAccount {  
    withdraw(amount: number): void {  
        if (this.balance >= amount) {  
            this.balance -= amount;  
            console.log(`Withdrawal successful. New balance: ${this.balance}`);  
        } else {  
            console.log("Insufficient funds.");  
        }  
    }  
}  
class CurrentAccount extends BankAccount {  
    private overdraftLimit = 500;  
    withdraw(amount: number): void {  
        if (this.balance + this.overdraftLimit >= amount) {  
            this.balance -= amount;  
            console.log(`Withdrawal successful. New balance: ${this.balance}`);  
        } else {  
            console.log("Overdraft limit exceeded.");  
        }  
    }  
}

// 6️⃣ Utility Types & Mapped Types  
interface Employee {  
    id: number;  
    name: string;  
    position: string;  
    salary: number;  
}  
type ReadonlyEmployee = Readonly<Employee>;  
type PartialEmployee = Partial<Employee>;

function updateEmployeeInfo(emp: PartialEmployee): void {  
    console.log("Updated Employee Info:", emp);  
}

// 7️⃣ Type Guards & Type Narrowing  
type TextMessage = { content: string };  
type ImageMessage = { imageUrl: string };  
type Message = TextMessage | ImageMessage;

function isTextMessage(msg: Message): msg is TextMessage {  
    return "content" in msg;  
}  
function handleMessage(msg: Message): void {  
    if (isTextMessage(msg)) {  
        console.log(`Text: ${msg.content}`);  
    } else {  
        console.log(`Image: ${msg.imageUrl}`);  
    }  
}

// 8️⃣ Decorators & Metadata  
function LogClass<T extends { new (...args: any[]): {} }>(constructor: T) {  
    return class extends constructor {  
        constructor(...args: any[]) {  
            super(...args);  
            console.log(`Class ${constructor.name} is instantiated`);  
        }  
    };  
}  
@LogClass  
class OrderService {  
    constructor() {  
        console.log("Order Service Created");  
    }  
}  
const orderService = new OrderService();

// 9️⃣ Asynchronous TypeScript & Promises  
async function getWeather(city: string): Promise<number> {  
    return new Promise((resolve, reject) => {  
        setTimeout(() => {  
            const temperature = Math.random() \* 40;  
            resolve(temperature);  
        }, 2000);  
    });  
}  
(async () => {  
    try {  
        const nyWeather = await getWeather("New York");  
        console.log(`New York Weather: ${nyWeather}°C`);  
        const londonWeather = await getWeather("London");  
        console.log(`London Weather: ${londonWeather}°C`);  
    } catch (error) {  
        console.error("Error fetching weather data", error);  
    }  
})();

// 🔟 TypeScript with MongoDB (for Full-Stack Developers)  
interface Product {  
    \_id: string;  
    name: string;  
    price: number;  
    stock: number;  
    category: string;  
}  
async function findProductById(id: string): Promise<Product | null> {  
    return new Promise((resolve) => {  
        setTimeout(() => {  
            const mockProduct: Product | null = Math.random() > 0.5 ? { \_id: id, name: "Laptop", price: 1000, stock: 5, category: "Electronics" } : null;  
            resolve(mockProduct);  
        }, 1000);  
    });  
}  
(async () => {  
    const product = await findProductById("12345");  
    if (product) {  
        console.log("Product Found:", product);  
    } else {  
        console.log("Product not found");  
    }  
})();