**1️. Variables & Types**

**Problem:**  
**Declare the following variables with appropriate TypeScript types:**

* **A string variable userName with a value "John Doe".**
* **A number variable age with a value 25.**
* **A boolean variable isStudent with a value true.**

**Bonus: Try to assign a different type to these variables and see what happens!**

let userName: string = "John Doe";

let age: number = 25;

let isStudent: boolean = true;

// Trying to assign different types (Uncomment to see errors)

// userName = 42; // Error: Type 'number' is not assignable to type 'string'.

// age = "twenty-five"; // Error: Type 'string' is not assignable to type 'number'.

// isStudent = "yes"; // Error: Type 'string' is not assignable to type 'boolean'.

**2️. Functions & Type Annotations**

**Problem:**  
**Write a function add that takes two numbers as arguments and returns their sum.**

function add(a: number, b: number): number {

return a + b;

}

// Example usage:

console.log(add(5, 10)); // Output: 15

**3️. Function with Optional Parameters**

**Problem:**  
**Modify the function below to make the second parameter optional. If the message is not provided, default to "Hello".**

function greet(name: string, message: string = "Hello"): string {

return `${message}, ${name}!`;

}

// Example usage:

console.log(greet("Alice")); // Output: "Hello, Alice!"

console.log(greet("Bob", "Good morning")); // Output: "Good morning, Bob!"

**4️. Union Types**

**Problem:  
Define a function displayId that accepts either a number or a string as an argument and logs it.**

function displayId(id: number | string): void {

console.log(`ID: ${id}`);

}

// Example usage:

displayId(123); // Output: ID: 123

displayId("ABC123"); // Output: ID: ABC123

**5️. Arrays & Tuples**

**Problem:**

* **Declare an array numbers that only holds numbers.**
* **Create a tuple person that contains a string (name) and a number (age).**

// Array of numbers

let numbers: number[] = [10, 20, 30];

// Tuple with name (string) and age (number)

let person: [string, number] = ["Alice", 30];

// Example usage:

console.log(numbers); // Output: [10, 20, 30]

console.log(person); // Output: ["Alice", 30]

**6️. Interfaces**

**Problem:  
Define an interface Person with the following properties:**

* **name (string)**
* **age (number)**
* **email (string)**

**Create a variable user that implements this interface.**

interface Person {

name: string;

age: number;

email: string;

}

const user: Person = {

name: "John Doe",

age: 25,

email: "john@example.com"

};

// Example usage:

console.log(user);

**7️. Type Aliases**

**Problem:  
Create a type alias Employee that includes:**

* **id (number)**
* **name (string)**
* **role ("admin" | "manager" | "employee")**

**Define an employee variable of type Employee.**

type Employee = {

id: number;

name: string;

role: "admin" | "manager" | "employee";

};

const employee: Employee = {

id: 101,

name: "Alice",

role: "manager"

};

// Example usage:

console.log(employee);

**8️. Enum**

**Problem:**  
**Define an enum Color with values "Red", "Green", and "Blue".  
Create a variable favoriteColor of type Color.**

enum Color {

Red = "Red",

Green = "Green",

Blue = "Blue"

}

let favoriteColor: Color = Color.Green;

// Example usage:

console.log(favoriteColor); // Output: Green

**9️. Readonly & Partial**

**Problem:  
Create a readonly object for user information.  
Use Partial<T> to make a type where all properties are optional.**

interface User {

id: number;

name: string;

email: string;

}

// Readonly object

const user: Readonly<User> = { id: 1, name: "Alice", email: "alice@example.com" };

// user.name = "Bob"; Error: Cannot assign to 'name' because it is a read-only property.

let newUser: Partial<User> = { name: "Bob" }; // Allowed

console.log(user);

console.log(newUser);

**10. Classes in TypeScript**

**Problem:  
Create a class Car with the following properties:**

* **brand (string)**
* **model (string)**
* **year (number)**

**Add a method getCarInfo() that returns a string like "Toyota Corolla (2020)".**

class Car {

brand: string;

model: string;

year: number;

constructor(brand: string, model: string, year: number) {

this.brand = brand;

this.model = model;

this.year = year;

}

getCarInfo(): string {

return `${this.brand} ${this.model} (${this.year})`;

}

}

// Example usage:

const myCar = new Car("Toyota", "Corolla", 2020);

console.log(myCar.getCarInfo()); // Output: Toyota Corolla (2020)

**1️1.Strongly Typed Variables & Functions**

**Scenario:**

**You're building a simple e-commerce application and need to store product details.**

**👉 Exercise:**

**Define a Product type with properties:**

* **id (number)**
* **name (string)**
* **price (number)**
* **isAvailable (boolean)**

**Create a function getDiscountedPrice(product: Product, discount: number): number that calculates the discounted price.**

// Define the Product type

type Product = {

id: number;

name: string;

price: number;

isAvailable: boolean;

};

// Function to calculate the discounted price

function getDiscountedPrice(product: Product, discount: number): number {

return product.price - (product.price \* discount) / 100;

}

// Example usage:

const laptop: Product = {

id: 101,

name: "Gaming Laptop",

price: 1500,

isAvailable: true

};

console.log(getDiscountedPrice(laptop, 10)); // Output: 1350

**12️. Union & Intersection Types**

**Scenario:**

**You need to create a function that formats user inputs, which can be a string, a number, or an array of strings.**

**👉 Exercise:**

**Implement a function formatInput(input: string | number | string[]): string**

* **If it's a string, return it in uppercase.**
* **If it's a number, return it as a string with a $ prefix.**
* **If it's an array of strings, return a comma-separated string.**

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(", ");

}

}

// Example usage:

console.log(formatInput("hello")); // "HELLO"

console.log(formatInput(50)); // "$50"

console.log(formatInput(["apple", "banana", "cherry"])); // "apple, banana, cherry"

**13️. Tuples & Enums**

**Scenario:**

**You're working on a ride-sharing app and need to store driver status and location.**

**👉 Exercise:**

* **Define an enum DriverStatus with values: Available, OnTrip, Offline.**
* **Create a tuple type for DriverInfo:**
  + **First element: driverId (number)**
  + **Second element: driverName (string)**
  + **Third element: status (DriverStatus)**
* **Declare a driver: DriverInfo tuple with example values.**

// Define the DriverStatus enum

enum DriverStatus {

Available = "Available",

OnTrip = "OnTrip",

Offline = "Offline"

}

// Define the DriverInfo tuple type

type DriverInfo = [number, string, DriverStatus];

// Example driver data

const driver: DriverInfo = [101, "John Doe", DriverStatus.OnTrip];

console.log(driver); // Output: [101, "John Doe", "OnTrip"]

**14️. Interfaces & Generics**

**Scenario:**

**You're developing a user management system that supports both customers and admins.**

**👉 Exercise:**

* **Define an interface User with:**
  + **id (number)**
  + **name (string)**
  + **role (Admin | Customer)**
* **Make a generic function getUserDetails<T extends User>(user: T): string**
  + **It should return "User [name] is a [role]".**

// Define User interface

interface User {

id: number;

name: string;

role: "Admin" | "Customer";

}

// Generic function to get user details

function getUserDetails<T extends User>(user: T): string {

return `User ${user.name} is a ${user.role}`;

}

// Example usage:

const admin: User = { id: 1, name: "Alice", role: "Admin" };

const customer: User = { id: 2, name: "Bob", role: "Customer" };

console.log(getUserDetails(admin)); // "User Alice is a Admin"

console.log(getUserDetails(customer)); // "User Bob is a Customer"

**15️. Class Implementation & Abstract Classes**

**Scenario:**

**You need to create a Bank Account System with different account types.**

**👉 Exercise:**

* **Create an abstract class BankAccount with:**
  + **accountNumber (number)**
  + **balance (number)**
  + **Abstract method withdraw(amount: number): void**
* **Implement two classes:**
  + **SavingsAccount (allows withdrawals if the balance is sufficient)**
  + **CurrentAccount (allows overdrafts up to $500)**
* **Create instances and test withdrawals.**

// Abstract class for Bank Account

abstract class BankAccount {

constructor(public accountNumber: number, public balance: number) {}

abstract withdraw(amount: number): void;

deposit(amount: number): void {

this.balance += amount;

}

}

// Savings Account (No overdraft)

class SavingsAccount extends BankAccount {

withdraw(amount: number): void {

if (amount > this.balance) {

console.log("Insufficient funds");

} else {

this.balance -= amount;

console.log(`Withdrawn: $${amount}, New Balance: $${this.balance}`);

}

}

}

// Current Account (Allows overdraft up to $500)

class CurrentAccount extends BankAccount {

withdraw(amount: number): void {

if (amount > this.balance + 500) {

console.log("Overdraft limit exceeded");

} else {

this.balance -= amount;

console.log(`Withdrawn: $${amount}, New Balance: $${this.balance}`);

}

}

}

// Example usage:

const savings = new SavingsAccount(101, 1000);

savings.withdraw(1100); // "Insufficient funds"

savings.withdraw(500); // "Withdrawn: $500, New Balance: $500"

const current = new CurrentAccount(102, 300);

current.withdraw(700); // "Withdrawn: $700, New Balance: -$400"

current.withdraw(200); // "Overdraft limit exceeded"

**6️. Utility Types & Mapped Types**

**Scenario:**

**You're building an API where some fields should be optional and others readonly.**

**👉 Exercise:**

* **Given an Employee interface:**

**typescript**

**CopyEdit**

// Define Employee interface

interface Employee {

id: number;

name: string;

position: string;

salary: number;

}

// Readonly version

type ReadonlyEmployee = Readonly<Employee>;

// Partial version

type PartialEmployee = Partial<Employee>;

// Function to update employee details

function updateEmployeeInfo(emp: PartialEmployee): void {

console.log("Updated Employee Info:", emp);

}

// Example usage:

const employee: ReadonlyEmployee = {

id: 1,

name: "Alice",

position: "Manager",

salary: 5000

};

// employee.salary = 6000; // Error: Cannot assign to 'salary' because it is a read-only property

const newEmployee: PartialEmployee = { name: "Bob" };

updateEmployeeInfo(newEmployee); // Output: Updated Employee Info: { name: "Bob" }

**18️. Decorators & Metadata**

**Scenario:**

**You need to log API requests in a Node.js backend.**

**👉 Exercise:**

* **Write a class decorator @LogClass that logs "Class [className] is instantiated" when an object is created.**
* **Apply it to a class OrderService.**
* **Create an instance of OrderService and verify the decorator works.**

function LogClass(constructor: Function) {

console.log(`Class ${constructor.name} is instantiated`);

}

@LogClass

class OrderService {

constructor() {

console.log("OrderService initialized!");

}

}

// Example usage:

const orderService = new OrderService();

// Output:

// Class OrderService is instantiated

// OrderService initialized!

**19️. Asynchronous TypeScript & Promises**

**Scenario:**

**You’re working on a weather app that fetches live temperature data from an API.**

**👉 Exercise:**

* **Write an async function getWeather(city: string): Promise<number>**
  + **Simulates fetching weather data after 2 seconds.**
* **Use await to get the weather for "New York" and "London", then log it.**
* **Implement proper error handling.**

function fetchWeatherData(city: string): Promise<number> {

return new Promise((resolve, reject) => {

setTimeout(() => {

const temperatures: Record<string, number> = {

"New York": 22,

"London": 18,

"Chennai": 34,

};

if (temperatures[city] !== undefined) {

resolve(temperatures[city]);

} else {

reject(new Error("City not found"));

}

}, 2000);

});

}

async function getWeather(city: string): Promise<void> {

try {

const temperature = await fetchWeatherData(city);

console.log(`The temperature in ${city} is ${temperature}°C`);

} catch (error) {

console.error(error.message);

}

}

// Example usage:

getWeather("New York"); // Logs: The temperature in New York is 22°C (after 2 sec)

getWeather("Paris"); // Logs: City not found

**20. TypeScript with MongoDB**

**Scenario:**

**You're building an inventory management system where products are stored in a MongoDB database.**

**👉 Exercise:**

* **Define a MongoDB schema type in TypeScript:**

**typescript**

// Define Product interface

interface Product {

\_id: string;

name: string;

price: number;

stock: number;

category: string;

}

// Simulated database

const productsDB: Product[] = [

{ \_id: "101", name: "Laptop", price: 1500, stock: 10, category: "Electronics" },

{ \_id: "102", name: "Smartphone", price: 800, stock: 20, category: "Electronics" },

];

async function findProductById(id: string): Promise<Product | null> {

return new Promise((resolve) => {

setTimeout(() => {

const product = productsDB.find((p) => p.\_id === id);

resolve(product || null);

}, 1000);

});

}

// Example usage:

async function main() {

const product = await findProductById("101");

if (product) {

console.log("Product Found:", product);

} else {

console.log("Product not found!");

}

}

main();

// Output (after 1 sec):

// Product Found: { \_id: '101', name: 'Laptop', price: 1500, stock: 10, category: 'Electronics' }