User-Defined Types in TypeScript

1. Problem Statement

You are building a patient management system for a hospital.

- Each patient record must track personal details, vital signs, and medical history.
- Different types of staff (doctors, nurses, admins) interact with the system, each with different permissions and data needs.
- The system must handle collections of patients, structured medical records, and specialized data like test results or medication schedules.
- You need to ensure that all data is organized, type-safe, and easy to extend as requirements change.

The challenge:
How do you use TypeScript's user-defined types (arrays, tuples, enums, classes, interfaces) to model complex, real-world data in a way that is safe, maintainable, and scalable?

2. Learning Objectives

- Create and use arrays, tuples, and enums in TypeScript.
- Define custom types using interfaces and classes.
- Model real-world data structures with type safety.
- Use user-defined types to organize and scale your codebase.

3. Concept Introduction with Analogy

Analogy: The Hospital Filing Cabinet

A hospital's filing cabinet contains:

- **Folders** for each patient (arrays).
- **Tabs** for specific info (tuples).
- **Color-coded labels** for patient status (enums).
- **Standard forms** for personal info and medical history (interfaces).
- **Specialized folders** for doctors, nurses, and admins (classes).

This system keeps patient data organized, consistent, and easy to find-just like user-defined types in TypeScript.

4. Technical Deep Dive

Arrays

- Store collections of values of the same type.

- Can use `type[]` or `Array<type>` syntax.

```
let patientIds: number[] = [101, 102, 103];
let patientNames: Array<string> = ["Alice", "Bob", "Carol"];
```

Tuples

- Arrays with a fixed number of elements, each with a specific type.

```
let vitalSigns: [number, number] = [120, 80]; // [systolic, diastolic]
let patientInfo: [string, number] = ["Alice", 30];
```

Enums

- Named set of related values (numeric or string).

```
enum PatientStatus { Admitted, Discharged, UnderObservation }
let status: PatientStatus = PatientStatus.Admitted;
```

Interfaces

- Define the shape of an object.

```
interface Patient {
  id: number;
  name: string;
  age: number;
  status: PatientStatus;
  vitals: [number, number];
}
```

Classes

- Blueprint for creating objects with data and behavior.

```
class Doctor {
  constructor(public name: string, public specialty: string) {}
  prescribe(medication: string): void {
    console.log(`${this.name} prescribes ${medication}`);
  }
}
```

5. Step-by-Step Data Modeling & Code Walkthrough

1. **Define enums and interfaces:**

   ```
   enum PatientStatus { Admitted, Discharged, UnderObservation }

   interface Patient {
     id: number;
     name: string;
     age: number;
     status: PatientStatus;
   ```

```
    vitals: [number, number];
}
```

2. **Create an array of patients:**

```
let patients: Patient[] = [
  { id: 1, name: "Alice", age: 30, status: PatientStatus.Admitted, vitals: [120, 80] },
  { id: 2, name: "Bob", age: 45, status: PatientStatus.UnderObservation, vitals: [130, 85] }
];
```

3. **Define a class for staff:**

```
class Nurse {
  constructor(public name: string) {}
  takeVitals(patient: Patient, vitals: [number, number]): void {
    patient.vitals = vitals;
    console.log(`${this.name} updated vitals for ${patient.name}`);
  }
}
```

4. **Use tuples and enums for structured data:**

```
let newVitals: [number, number] = [118, 76];
let nurse = new Nurse("Carol");
nurse.takeVitals(patients[0], newVitals);
```

6. Interactive Challenge

Your Turn!

- Define an enum `Role` for staff roles (Doctor, Nurse, Admin).
- Create an interface `Staff` with fields for `id`, `name`, and `role`.
- Create an array of staff members using the interface and enum.
- Write a function that prints a summary of all staff, showing their name and role.

7. Common Pitfalls & Best Practices

- **Use interfaces to define object shapes for clarity and safety.**
- **Prefer enums for related constants instead of plain numbers or strings.**
- **Use tuples only when element order and type are fixed and meaningful.**
- **Organize related data in arrays for easy processing.**
- **Use classes for objects with both data and behavior.**

8. Quick Recap & Key Takeaways

- User-defined types (arrays, tuples, enums, interfaces, classes) help you model real-world data safely and clearly.
- They make your code more maintainable, scalable, and expressive.
- Organize your data structures with type safety for fewer bugs and easier collaboration.

9. Optional: Programmer’s Workflow Checklist

- Use enums for related sets of constants.
- Define interfaces for all complex objects.
- Use arrays for collections of similar items.
- Use tuples for fixed-structure, ordered data.
- Use classes for objects that combine data and methods.
- Always annotate types for clarity and safety.