Day 3: Classes, Enums, and Union/Intersection Types

Objective: To explore object-oriented programming with classes, learn about enums, and combine types using union and intersection.

Classes in TypeScript (25 minutes)

Classes are a fundamental part of object-oriented programming (OOP). They act as blueprints for creating objects that have their own properties (data) and methods (functions).

1. Defining Classes, Properties, Constructors, and Methods

A class definition includes:

- **Properties:** Variables that belong to the class.
- **Constructor:** A special method for creating and initializing an object instance of the class. It's called when you use the `new` keyword.
- **Methods:** Functions that define the behavior of the objects created from the class.

```
class Vehicle {
  // Properties
  make: string;
  model: string;
  year: number;

  // Constructor
  constructor(make: string, model: string, year: number) {
    this.make = make;
    this.model = model;
    this.year = year;
  }

  // Method
  getDetails(): string {
    return `${this.year} ${this.make} ${this.model}`;
  }
}
```

2. Access Modifiers: `public`, `private`, and `protected`

TypeScript provides keywords to control the visibility of class members:

- **`public` (default):** The member can be accessed from anywhere.
- **`private`:** The member can only be accessed from within the class itself.
- **`protected`:** The member can be accessed from within the class and by any subclasses that extend it.

```ts
class Employee {
  public name: string;
  private salary: number;

  constructor(name: string, salary: number) {
    this.name = name;
    this.salary = salary;
  }

  public getAnnualSalary(): number {
    return this.salary * 12;
  }
}

const emp = new Employee("John", 50000);
console.log(emp.name); // OK
// console.log(emp.salary); // Error: Property 'salary' is private.
```

3. Implementing Interfaces in Classes

You can use an interface to ensure a class has a specific structure. The `implements` keyword is used to check if the class meets the contract of the interface.

```ts
interface ILoggable {
  log(): void;
}

class Product implements ILoggable {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  log(): void {
    console.log(`Product: ${this.name}`);
  }
}
```

4. Hands-on: Creating a Class with Properties, Methods, and Access Modifiers

Let's create a `Student` class. Add this to your `index.ts` file:

```ts
class Student {
  public readonly studentId: number;
  private grades: number[] = [];

  constructor(public name: string) {
    // A parameter property like 'public name' is a shorthand
    // for declaring and initializing a property.
    this.studentId = Math.floor(Math.random() * 1000);
  }
```

```
  addGrade(grade: number): void {
    if (grade >= 0 && grade <= 100) {
      this.grades.push(grade);
    }
  }

  getAverageGrade(): number {
    if (this.grades.length === 0) return 0;
    const sum = this.grades.reduce((total, grade) => total + grade, 0);
    return sum / this.grades.length;
  }
}

const student1 = new Student("Alice");
student1.addGrade(95);
student1.addGrade(88);
console.log(
  `${student1.name}'s average grade is ${student1.getAverageGrade()}`
);
```

Enums (10 minutes)

Enums (enumerations) allow you to define a set of named constants. They make your code more readable and less prone to errors from using magic numbers or strings.

1. What Are Enums and When to Use Them?

Use enums when you have a value that can only be one of a small set of possible values. Examples include user roles (Admin, Editor, Viewer), status codes (Pending, Approved, Rejected), or directions (North, South, East, West).

2. Numeric and String Enums

- **Numeric Enums:** By default, enums are number-based. The first value is `0`, and the rest auto-increment. You can also set the starting value.
- **String Enums:** You can also use strings for your enum values. This can be more descriptive.

```
// Numeric Enum
enum OrderStatus {
  Pending, // 0
  Shipped, // 1
  Delivered, // 2
  Cancelled, // 3
}

// String Enum
enum UserRole {
  Admin = "ADMIN",
  Editor = "EDITOR",
  Viewer = "VIEWER",
}
```

3. Activity: Defining and Using an Enum

Let's use the `OrderStatus` enum. Add this to your `index.ts`:

```
let myOrderStatus: OrderStatus = OrderStatus.Pending;
console.log(`My order status is: ${myOrderStatus}`); // Outputs: 0

myOrderStatus = OrderStatus.Shipped;
console.log(`My order status is now: ${myOrderStatus}`); // Outputs: 1

function processOrder(status: OrderStatus) {
  if (status === OrderStatus.Pending) {
    console.log("Processing a pending order.");
  }
}

processOrder(myOrderStatus);
```

Union and Intersection Types (20 minutes)

These advanced types provide more flexibility in how you define your data structures.

1. Union Types (|)

A union type allows a variable to be one of several types. The vertical bar | acts as an "OR".

```
let id: string | number;

id = 101; // OK
id = "abc-123"; // OK
// id = false; // Error: Type 'boolean' is not assignable to type 'string | number'.
```

2. Intersection Types (&)

An intersection type combines multiple types into a single type with all the properties of the combined types. The ampersand & acts as an "AND".

```
interface Draggable {
  drag(): void;
}

interface Resizable {
  resize(): void;
}

type UIWidget = Draggable & Resizable;

let widget: UIWidget = {
  drag: () => console.log("Dragging..."),
  resize: () => console.log("Resizing..."),
};
```

3. Type Narrowing with `typeof` and `instanceof`

When you have a union type, you often need to check what the type currently is before you can perform certain operations. This is called type narrowing.

- Use `typeof` for primitive types (`string`, `number`, `boolean`).
- Use `instanceof` for checking class instances.

4. Hands-on: Writing a Function That Accepts a Union Type

Let's write a function that can accept either a single string or an array of strings and prints them. This demonstrates a union type and type narrowing. Add this to your `index.ts`:

```
function printItems(items: string | string[]): void {
  if (typeof items === "string") {
    // Here, TypeScript knows 'items' is a string
    console.log(items);
  } else {
    // Here, TypeScript knows 'items' is a string[]
    items.forEach((item) => console.log(item));
  }
}

printItems("Hello"); // Works with a single string
printItems(["Apple", "Banana", "Cherry"]); // Works with an array of strings
```

Q&A and Wrap-up (5 minutes)

- **Review of Key Concepts:**
  - **Classes** provide a blueprint for creating objects with properties and methods, supporting OOP principles.
  - **Enums** give friendly names to a set of related constants.
  - **Union (`|`)** and **Intersection (`&`)** types allow for more flexible and composite type definitions.
- **Preview of Day 4:**
  - Tomorrow, in our final session, we will cover two powerful features: **Generics**, for writing reusable, type-safe components, and built-in **Utility Types** that help you transform existing types.