# Typescript

ts / tsx

# TypeScript Fundamentals

## Course Overview

- Basic Types and Variables
- Functions and Parameters
- Arrays and Objects
- Classes and Interfaces
- Advanced Types

## Basic Types in TypeScript

### Primitive Types

```typescript
// Basic Types
let name: string = "John";
let age: number = 25;
let isStudent: boolean = true;
let u: undefined = undefined;
let n: null = null;

// Special Types
let anyType: any = "anything";
let unknownType: unknown = 4;
let voidType: void = undefined;
let neverType: never;  // Never returns

// Type Inference
let inferredString = "Hello";  // Type: string
let inferredNumber = 42;       // Type: number
```

# Variables and Constants

## Variable Declarations

```typescript
// let - block scoped, can be reassigned
let counter: number = 0;
counter = 1;  // OK

// const - block scoped, cannot be reassigned
const PI: number = 3.14159;
// PI = 3;  // Error!

// Type Assertions
let someValue: unknown = "this is a string";
let strLength: number = (someValue as string).length;
// or
let strLength2: number = (<string>someValue).length;

// Union Types
let id: string | number = 123;
id = "ABC";  // Also valid

// Type Aliases
type Point = {
    x: number;
    y: number;
};
let position: Point = { x: 10, y: 20 };
```

# Functions in TypeScript

## Function Types and Signatures

```
// Basic Function
function add(x: number, y: number): number {
    return x + y;
}

// Arrow Function
const multiply = (x: number, y: number): number => x * y;

// Optional Parameters
function greet(name: string, greeting?: string): string {
    return greeting ? `${greeting} ${name}` : `Hello ${name}`;
}

// Default Parameters
function createPoint(x: number = 0, y: number = 0): Point {
    return { x, y };
}

// Rest Parameters
function sum(...numbers: number[]): number {
    return numbers.reduce((total, n) => total + n, 0);
}

// Function Overloads
function processValue(x: number): number;
function processValue(x: string): string;
function processValue(x: any): any {
    return x;
}
```

# Arrays and Tuples

Working with Collections

```
// Array Types
let numbers: number[] = [1, 2, 3, 4, 5];
let strings: Array<string> = ["a", "b", "c"];

// Mixed Arrays
let mixed: (string | number)[] = [1, "two", 3];

// Tuple Type
let tuple: [string, number] = ["hello", 10];

// Array Methods with Types
numbers.push(6);  // OK
// numbers.push("7");  // Error!

// Array Destructuring
let [first, second, ...rest] = numbers;

// Array Methods with Types
let doubled: number[] = numbers.map(x => x * 2);
let evens: number[] = numbers.filter(x => x % 2 === 0);
let sum: number = numbers.reduce((acc, val) => acc + val, 0);

// Readonly Arrays
const readonlyNumbers: ReadonlyArray<number> = [1, 2, 3];
// readonlyNumbers[0] = 4;  // Error!
```

## Complex Types

```
// Object Type
let user: { name: string; age: number } = {
    name: "John",
    age: 30
};

// Interface
interface User {
    name: string;
    age: number;
    email?: string;  // Optional property
    readonly id: number;  // Read-only property
}

// Implementing Interface
let admin: User = {
    name: "Admin",
    age: 35,
    id: 1
};

// Index Signatures
interface StringMap {
    [key: string]: string;
}

let dictionary: StringMap = {
    "key1": "value1",
    "key2": "value2"
};

// Extending Interfaces
interface Employee extends User {
    salary: number;
    department: string;
}
```

## Object-Oriented Programming

```
class Animal {
    private name: string;
    protected age: number;

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

    public makeSound(): void {
        console.log("Some sound");
    }
}

// Inheritance
class Dog extends Animal {
    constructor(name: string, age: number) {
        super(name, age);
    }

    public makeSound(): void {
        console.log("Woof!");
    }
}

// Class with Interface
interface Pet {
    name: string;
    play(): void;
}

class Cat extends Animal implements Pet {
    play() {
        console.log("Playing with toy");
    }
}
```

## Advanced Types

### Type Manipulation

```typescript
// Union Types
type StringOrNumber = string | number;

// Intersection Types
type Employee = Person & { salary: number };

// Generic Types
function identity<T>(arg: T): T {
    return arg;
}

// Utility Types
interface Todo {
    title: string;
    description: string;
    completed: boolean;
}

// Partial
type PartialTodo = Partial<Todo>;

// Pick
type TodoPreview = Pick<Todo, "title" | "completed">;

// Omit
type TodoWithoutDescription = Omit<Todo, "description">;

// Record
type CatInfo = { age: number; breed: string };
type CatNames = "miffy" | "boris";
const cats: Record<CatNames, CatInfo> = {
    miffy: { age: 10, breed: "Persian" },
    boris: { age: 5, breed: "Maine Coon" }
};
```

## TypeScript Configuration

tsconfig.json

```
{
    "compilerOptions": {
        "target": "es6",
        "module": "commonjs",
        "strict": true,
        "esModuleInterop": true,
        "skipLibCheck": true,
        "forceConsistentCasingInFileNames": true,
        "outDir": "./dist",
        "rootDir": "./src"
    },
    "include": ["src/**/*"],
    "exclude": ["node_modules"]
}
```

Key Compiler Options:

- target: ECMAScript target version
- module: Module system
- strict: Enable strict type checking
- outDir: Output directory

# Best Practices & Tips

## TypeScript Best Practices

- Use explicit type annotations when necessary
- Leverage type inference when possible
- Prefer interfaces over type aliases for objects
- Use enums for distinct values
- Enable strict mode in tsconfig.json
- Use generics for reusable code
- Consider using unknown instead of any
- Use readonly when applicable