### **Course Outline**

### 

### **Module 1: Introduction to TypeScript**

* What is TypeScript?
* Why TypeScript? Benefits over JavaScript.
* Installing TypeScript and Setting Up the Environment (Node.js, TypeScript Compiler, IDE).
* First TypeScript Program and Compilation (Transpiling TS to JS).

### **Module 2: TypeScript Basics**

* Type Annotations: *String, Number, Boolean, Array, Object, Tuple, Enum.*
* Type Inference and Type Assertions.
* Union and Intersection Types.
* Literal Types and Aliases.
* Nullable Types and Null Handling.

### **Module 3: Functions and Interfaces**

* Function Types and Optional/Default Parameters.
* Arrow Functions and Return Type Annotations.
* Understanding **Interfaces** and their Use Cases.
* Extending Interfaces and Implementing Interfaces in Classes.

### **Module 4: Object-Oriented Programming in TypeScript**

* Classes and Access Modifiers (*public, private, protected*).
* Class Inheritance, Polymorphism, and Abstract Classes.
* Constructors, Getters, Setters, and Static Properties.
* Interfaces vs Abstract Classes.

### **Module 5: Advanced TypeScript Concepts**

* Generics: Using Generics in Functions, Classes, and Interfaces.
* Type Guards and Conditional Types.
* Utility Types (*Partial, Readonly, Record, Pick, Omit*).
* Advanced Type Manipulation: Keyof, Mapped Types.
* Type Declaration Files and Modules (*.d.ts Files*).

### **Module 6: Working with TypeScript and JavaScript**

* Integrating TypeScript into Existing JavaScript Projects.
* TypeScript Configuration (*tsconfig.json*).
* Debugging TypeScript Code.
* Handling Errors and Best Practices.

### **Module 7: Testing and Deployment**

* Unit Testing in TypeScript (Jest, Mocha).
* Type-Safe Code with ESLint and Prettier.
* Compiling, Building, and Deploying TypeScript Projects.

### **Module 1: Introduction to TypeScript**

This module provides a foundational understanding of TypeScript, its purpose, benefits, and how to set up a working environment to start writing and compiling TypeScript code.

### **1. What is TypeScript?**

**TypeScript** is a **superset of JavaScript** developed and maintained by Microsoft. It adds **static typing** to JavaScript, which helps developers catch errors at compile time rather than runtime. TypeScript retains all features of JavaScript and compiles down to plain JavaScript, making it compatible with any JavaScript engine (browsers, Node.js, etc.).

**Key Features of TypeScript:**

* **Static Typing**: Enforces type checking at compile time.
* **Type Inference**: Automatically infers types even if not explicitly declared.
* **Object-Oriented Programming (OOP)**: Supports classes, interfaces, and inheritance.
* **Enhanced Tooling Support**: Better autocompletion, refactoring, and bug detection in IDEs.

### **2. Why TypeScript? Benefits over JavaScript**

TypeScript offers several advantages that make it a preferred choice for modern web development:

1. **Early Error Detection**:  
   * TypeScript catches common errors during development (at compile time), reducing runtime bugs.
   * Example: Declaring a variable as number and mistakenly assigning it a string will trigger a compile-time error.
2. **Improved Code Readability and Maintainability**:  
   * The use of types makes the code self-documenting and easier to understand.
3. **Enhanced IDE Support**:  
   * Provides better autocompletion, inline documentation, and error detection in IDEs like Visual Studio Code.
4. **Supports Modern JavaScript Features**:  
   * TypeScript supports ES6+ features (like arrow functions, destructuring, and async/await) and transpiles them for wider browser compatibility.
5. **Scalability for Large Projects**:  
   * Useful for large-scale applications where static typing helps manage complex codebases and avoid type-related bugs.

### **3. Installing TypeScript and Setting Up the Environment**

To get started with TypeScript, you’ll need to install and configure your development environment.

**Prerequisites:**

* **Node.js**: TypeScript requires Node.js to run the TypeScript compiler (tsc).

**Step 1: Install Node.js**

* Download and install Node.js from<https://nodejs.org/>.

After installation, check if Node.js and npm (Node Package Manager) are installed by running  
node -v

npm -v

**Step 2: Install TypeScript** TypeScript can be installed globally using npm:

npm install -g typescript

After installation, verify that TypeScript is installed:

tsc -v

This command will display the installed TypeScript version.

**Step 3: Setting Up an IDE**

* It’s recommended to use **Visual Studio Code (VS Code)**, which has excellent TypeScript support.  
  + Download and install from<https://code.visualstudio.com/>.
  + Install the **TypeScript Extension** (optional) for enhanced autocompletion.

### **4. First TypeScript Program and Compilation (Transpiling TS to JS)**

Once TypeScript is installed, follow these steps to write and compile your first TypeScript program:

**Step 1: Create a TypeScript File**

1. Open VS Code and create a new folder for your project.
2. Inside the folder, create a file named app.ts.

**Step 2: Write Your First TypeScript Code**

let message: string = "Hello Student, Welcome to TypeScript World!";

console.log(message);

**Explanation:**

* let message: string: Declares a variable message of type string.
* The console.log() function prints the message to the console.

**Step 3: Compile TypeScript to JavaScript** Use the TypeScript compiler to transpile app.ts into JavaScript:

tsc app.ts

This will generate a JavaScript file app.js in the same folder. You can then run the generated JavaScript file using Node.js:

node app.js

**Expected Output:**

Hello, TypeScript!

**Step 4: Configuring TypeScript with tsconfig.json (Optional)** For larger projects, you can create a tsconfig.json file to manage TypeScript configuration options:

tsc --init

This command generates a tsconfig.json file with customizable options, such as target JavaScript version, module resolution, and more.

### **Create Project Folder Structure**

**Step 1:**

1. **Project Structure**:

Create the following folder structure in your project directory:  
  
 pgsql  
  
my-typescript-project/

├── src/ → Contains TypeScript files

│ └── app.ts → Sample TypeScript file

├── dist/ → Will contain the compiled JavaScript files (after running the compiler)

├── tsconfig.json → TypeScript configuration file

└── package.json → For npm configurations (optional)

**Add a Sample TypeScript File**:  
 Add the app.ts file inside the src folder with the following code:  
  
let greeting: string = "Hello from the src folder!";

console.log(greeting);

### **Step 2: Configure tsconfig.json for src and dist**

**Generate the tsconfig.json File (if not created yet)**:  
 Run the following command to create the tsconfig.json file:  
  
 bash  
  
tsc --init

1. This generates a tsconfig.json file with default settings.

**Modify tsconfig.json to Use src and dist:** Update the following options inside tsconfig.json to specify src as the input folder and dist as the output folder:  
  
 **Updated tsconfig.json:** json  
  
{

"compilerOptions": {

"target": "es6", // Target JavaScript version (ES6)

"module": "commonjs", // Module system

"outDir": "./dist", // Output directory for compiled JavaScript files

"rootDir": "./src", // Root directory for TypeScript source files

"strict": true, // Enable strict type checking

"esModuleInterop": true // Allow interop between CommonJS and ES modules

},

"include": ["src/\*\*/\*"], // Include all files in src folder

"exclude": ["node\_modules"] // Exclude node\_modules folder

}

1. **Explanation of Key Options:**
   * "rootDir": "./src": Specifies that TypeScript source files are located inside the src folder.
   * "outDir": "./dist": Specifies that the compiled JavaScript files should be output to the dist folder.
   * "include": ["src/\*\*/\*"]: Includes all TypeScript files inside the src folder and its subfolders.
   * "exclude": ["node\_modules"]: Ensures that the node\_modules folder is ignored by the TypeScript compiler.

### **Step 3: Compile the TypeScript Code**

1. Save the changes to tsconfig.json.

Run the TypeScript compiler:

tsc

1. This command reads the tsconfig.json file, compiles all .ts files from the src folder, and outputs the compiled .js files in the dist folder.

### **Step 4: Run the Compiled JavaScript**

After running the TypeScript compiler, navigate to the dist folder and execute the compiled JavaScript using Node.js:

node dist/app.js

**Expected Output:**

Hello from the src folder!

### **Final Folder Structure (After Compilation)**

After following the above steps, your project folder will look like this:

my-typescript-project/

├── src/

│ └── app.ts → Original TypeScript file

├── dist/

│ └── app.js → Compiled JavaScript file

├── tsconfig.json → TypeScript configuration file

└── package.json → Optional npm configuration file (if applicable)

### **Module 2: TypeScript Basics**

In this module, we dive into the core TypeScript concepts that are essential for working with types and writing type-safe code. You will learn how to use type annotations, explore type inference and assertions, and handle more advanced types like unions, intersections, and null values.

### **1. Type Annotations**

Type annotations allow you to explicitly declare the data type of variables, function parameters, and return values in TypeScript. Below are some of the common types used in TypeScript:

#### **1.1 Primitive Types**

**String**: Used for text values.  
  
  
let username: string = "John Doe";

**Number**: Used for integers and floating-point numbers.  
  
  
let age: number = 30;

**Boolean**: Used for true/false values.  
  
  
let isActive: boolean = true;

#### **1.2 Complex Types**

**Array**: Declares an array of specific types.  
  
  
let numbers: number[] = [1, 2, 3, 4];

let fruits: string[] = ["Apple", "Banana", "Cherry"];

**Object**: Defines objects with specific properties and types.  
  
  
let user: { name: string; age: number; isAdmin: boolean } = {

name: "Alice",

age: 25,

isAdmin: false

};

**Tuple**: A fixed-length array with different types in each position.  
  
  
let person: [string, number] = ["John", 40]; // First element is string, second is number.

**Enum**: Defines a set of named constants.  
  
enum Role {

Admin,

User,

Guest

}

let userRole: Role = Role.Admin; // Assigns the Admin value from the Role enum.

### **2. Type Inference and Type Assertions**

#### **2.1 Type Inference**

TypeScript can infer types automatically based on the assigned value, which means you don’t always need to specify the type explicitly.

Example:

typescript

let city = "Lagos"; // Inferred as a string.

let count = 10; // Inferred as a number.

If TypeScript can infer the type, explicit type annotations may not be necessary.

#### **2.2 Type Assertions**

Type assertions are used when you know more about the type of a value than TypeScript does. It allows you to override TypeScript’s inferred type.

Example of Type Assertion:

typescript

let value: any = "Hello, TypeScript!";

let stringLength: number = (value as string).length; // Assert that value is a string.

Alternative Syntax (using angle brackets):

typescript

let stringLength: number = (<string>value).length;

### **3. Union and Intersection Types**

#### **3.1 Union Types**

Union types allow a variable to have more than one possible type. This is useful when a value can be one of several types.

Example:

typescript

let score: number | string; // Can be a number or a string.

score = 100;

score = "A+";

#### **3.2 Intersection Types**

Intersection types combine multiple types into one. A variable with an intersection type must satisfy all the included types.

Example:

typescript

interface Employee {

employeeId: number;

department: string;

}

interface Manager {

managerLevel: number;

}

type SeniorManager = Employee & Manager; // Must have properties from both Employee and Manager.

let seniorManager: SeniorManager = {

employeeId: 123,

department: "Engineering",

managerLevel: 2

};

### **4. Literal Types and Type Aliases**

#### **4.1 Literal Types**

Literal types allow variables to hold specific, predefined values. This is often used for strict value constraints.

Example:

typescript

let trafficLight: "red" | "yellow" | "green";

trafficLight = "red"; // This is allowed.

// trafficLight = "blue"; // Error: "blue" is not assignable to the literal type.

#### **4.2 Type Aliases**

Type aliases give a custom name to a type, making the code more readable and maintainable.

Example:

typescript

type Status = "active" | "inactive" | "suspended";

let userStatus: Status = "active";

type Point = { x: number; y: number };

let coordinate: Point = { x: 10, y: 20 };

### **5. Nullable Types and Null Handling**

In TypeScript, you can handle null and undefined explicitly by using union types or strict null checks.

**Allowing Null Values**:  
 You can explicitly allow null or undefined in type definitions.  
  
 typescript  
  
let userName: string | null = null; // Can be a string or null.

userName = "Alice";

**Null Checking with Optional Chaining (ES2020)**:  
 Optional chaining helps safely access deeply nested properties without risking null reference errors.  
  
 typescript  
  
let user = { address: { city: "Ilorin" } };

console.log(user?.address?.city); // Outputs: Ilorin (safe null check)

**Nullish Coalescing Operator (??)**:  
 Provides a fallback value when a variable is null or undefined.  
  
 typescript  
  
let input: string | null = null;

let defaultInput = input ?? "Default Value"; // Returns "Default Value" if input is null.

### **Module 3: Functions and Interfaces**

This module covers the advanced usage of **functions** and the power of **interfaces** in TypeScript. You will learn how to define function types, handle optional and default parameters, and leverage interfaces to create structured, type-safe code.

### **1. Function Types and Optional/Default Parameters**

In TypeScript, functions can be assigned specific types, including the types of parameters and return values.

#### **Function Types**

You can explicitly define the parameter and return types of a function.

Example:

typescript

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

return a + b;

}

let sum: number = add(10, 20); // Correct usage

// add("10", 20); // Error: Argument of type 'string' is not assignable to type 'number'

#### **Optional Parameters**

You can make function parameters optional by adding a ? after the parameter name.

Example:

typescript

function greet(name: string, age?: number): string {

return age ? `Hello, ${name}. You are ${age} years old.` : `Hello, ${name}.`;

}

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

console.log(greet("Bob", 30)); // Output: Hello, Bob. You are 30 years old.

#### **Default Parameters**

You can provide default values to parameters, which will be used if no value is passed.

Example:

typescript

function greetUser(name: string = "Guest"): void {

console.log(`Welcome, ${name}!`);

}

greetUser(); // Output: Welcome, Guest!

greetUser("John"); // Output: Welcome, John!

### **2. Arrow Functions and Return Type Annotations**

Arrow functions provide a concise way to define functions. They also support type annotations for parameters and return values.

Example:

typescript

const multiply = (a: number, b: number): number => a \* b;

console.log(multiply(5, 10)); // Output: 50

Key points about arrow functions:

* Arrow functions don’t have their own this context, which makes them useful in callbacks and event handlers.
* They can be combined with other TypeScript features like optional parameters, default parameters, and function types.

### **3. Understanding Interfaces and Their Use Cases**

**Interfaces** in TypeScript define the structure of objects and help ensure that objects follow a consistent shape.

#### **Defining an Interface**

An interface specifies the properties and methods an object should have.

Example:

typescript

interface User {

name: string;

age: number;

isActive: boolean;

}

let user1: User = {

name: "Alice",

age: 25,

isActive: true,

};

console.log(user1); // Output: { name: "Alice", age: 25, isActive: true }

Key Benefits of Using Interfaces:

* **Type Safety**: Prevents accidental type mismatches.
* **Code Readability**: Makes it clear what structure an object must follow.
* **Reusability**: Interfaces can be reused across different parts of the code.

### **4. Extending Interfaces and Implementing Interfaces in Classes**

#### **Extending Interfaces**

You can create new interfaces by extending existing ones, which allows for inheritance and code reuse.

Example:

typescript

interface Person {

name: string;

age: number;

}

interface Employee extends Person {

employeeId: number;

position: string;

}

const employee1: Employee = {

name: "John",

age: 30,

employeeId: 1234,

position: "Manager",

};

console.log(employee1); // Output: { name: "John", age: 30, employeeId: 1234, position: "Manager" }

#### **Implementing Interfaces in Classes**

Classes can implement interfaces to enforce that they follow a specific structure.

Example:

typescript

interface Shape {

area(): number;

}

class Circle implements Shape {

radius: number;

constructor(radius: number) {

this.radius = radius;

}

area(): number {

return Math.PI \* this.radius \* this.radius;

}

}

const circle = new Circle(5);

console.log(`Circle Area: ${circle.area()}`); // Output: Circle Area: 78.53981633974483

Key Takeaways:

* The implements keyword enforces that the class adheres to the interface.
* Any class that implements an interface must define the methods and properties described in the interface.

### **Best Practices for Functions and Interfaces**

* **Use explicit return types** for functions when the return value isn’t obvious.
* **Use optional and default parameters** to handle flexible function arguments.
* **Leverage interfaces** to enforce consistent structure across objects and classes.
* **Extend interfaces** for inheritance and to keep your code DRY (Don't Repeat Yourself).

### **Module 4: Classes and Access Modifiers in TypeScript**

In this module, we delve into **object-oriented programming (OOP)** concepts in TypeScript, focusing on classes, access modifiers, inheritance, polymorphism, abstract classes, and static members. These OOP principles improve code organization, scalability, and reusability.

### **1. Classes and Access Modifiers (public, private, protected)**

A **class** in TypeScript acts as a blueprint for creating objects. It can define properties and methods that describe the behavior of those objects.

#### **Defining a Class**

typescript

CopyEdit

class Person {

name: string; // Property

constructor(name: string) {

this.name = name; // Constructor initializing 'name'

}

greet(): void { // Method

console.log(`Hello, my name is ${this.name}.`);

}

}

const person1 = new Person("Alice");

person1.greet(); // Output: Hello, my name is Alice.

#### **Access Modifiers**

Access modifiers control the visibility of class properties and methods:

**public (default)**: Accessible from anywhere (inside or outside the class).  
  
 typescript  
CopyEdit  
class Car {

public brand: string;

constructor(brand: string) {

this.brand = brand;

}

public drive(): void {

console.log(`${this.brand} is driving.`);

}

}

const car = new Car("Toyota");

car.drive(); // Output: Toyota is driving.

**private**: Accessible only within the class (not visible outside).  
  
 typescript  
CopyEdit  
class BankAccount {

private balance: number; // Private property

constructor(initialBalance: number) {

this.balance = initialBalance;

}

deposit(amount: number): void {

this.balance += amount;

console.log(`New balance: ${this.balance}`);

}

}

const account = new BankAccount(1000);

account.deposit(500); // Output: New balance: 1500

// console.log(account.balance); // Error: Property 'balance' is private

**protected**: Accessible within the class and its subclasses.  
  
 typescript  
CopyEdit  
class Employee {

protected salary: number;

constructor(salary: number) {

this.salary = salary;

}

}

class Manager extends Employee {

showSalary(): void {

console.log(`Manager's salary is: ${this.salary}`);

}

}

const manager = new Manager(50000);

manager.showSalary(); // Output: Manager's salary is: 50000

### **2. Class Inheritance, Polymorphism, and Abstract Classes**

#### **Inheritance**

Inheritance allows one class to inherit the properties and methods of another class using the extends keyword.

Example:

typescript

CopyEdit

class Animal {

move(): void {

console.log("The animal is moving.");

}

}

class Dog extends Animal {

bark(): void {

console.log("The dog is barking.");

}

}

const dog = new Dog();

dog.move(); // Output: The animal is moving.

dog.bark(); // Output: The dog is barking.

#### **Polymorphism**

Polymorphism allows methods in child classes to have different implementations than their parent class methods.

Example:

typescript

CopyEdit

class Animal {

makeSound(): void {

console.log("The animal makes a sound.");

}

}

class Cat extends Animal {

makeSound(): void {

console.log("The cat meows.");

}

}

const animal: Animal = new Cat();

animal.makeSound(); // Output: The cat meows (polymorphic behavior)

#### **Abstract Classes**

Abstract classes provide a base class that cannot be instantiated directly. They can include abstract methods that must be implemented by subclasses.

Example:

typescript

CopyEdit

abstract class Shape {

abstract area(): number; // Abstract method (must be implemented in subclass)

}

class Circle extends Shape {

radius: number;

constructor(radius: number) {

super();

this.radius = radius;

}

area(): number {

return Math.PI \* this.radius \* this.radius;

}

}

const circle = new Circle(5);

console.log(`Circle Area: ${circle.area()}`); // Output: Circle Area: 78.54

### **3. Constructors, Getters, Setters, and Static Properties**

#### **Constructors**

A constructor is a special method used to initialize objects when they are created.

Example:

typescript

CopyEdit

class Person {

name: string;

constructor(name: string) {

this.name = name;

}

}

const person = new Person("Alice");

console.log(person.name); // Output: Alice

#### **Getters and Setters**

Getters and setters allow controlled access to private properties.

Example:

typescript

CopyEdit

class Student {

private \_name: string;

constructor(name: string) {

this.\_name = name;

}

get name(): string {

return this.\_name;

}

set name(value: string) {

if (value.length < 3) {

console.log("Name is too short.");

} else {

this.\_name = value;

}

}

}

const student = new Student("John");

console.log(student.name); // Output: John

student.name = "Al"; // Output: Name is too short.

student.name = "Alice";

console.log(student.name); // Output: Alice

#### **Static Properties and Methods**

Static members belong to the class itself, not to any specific object instance.

Example:

typescript

CopyEdit

class MathUtils {

static PI = 3.14;

static calculateCircleArea(radius: number): number {

return this.PI \* radius \* radius;

}

}

console.log(MathUtils.PI); // Output: 3.14

console.log(MathUtils.calculateCircleArea(5)); // Output: 78.5

### **4. Interfaces vs. Abstract Classes**

Both interfaces and abstract classes are used to define structures, but they have key differences:

| **Feature** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| **Purpose** | Defines a contract for objects. | Serves as a base class with partial implementation. |
| **Inheritance** | A class can implement multiple interfaces. | A class can only extend one abstract class. |
| **Abstract Methods** | All methods are abstract. | Can have both abstract and non-abstract methods. |
| **Properties** | Only method signatures and properties. | Can include fields and methods with logic. |
| **Example Usage** | Best for defining object structure. | Best for providing a common base class. |

**Interface Example:**

typescript

CopyEdit

interface Animal {

name: string;

makeSound(): void;

}

class Dog implements Animal {

name: string;

constructor(name: string) {

this.name = name;

}

makeSound(): void {

console.log("Bark!");

}

}

**Abstract Class Example:**

typescript

CopyEdit

abstract class Animal {

abstract makeSound(): void;

move(): void {

console.log("The animal moves.");

}

}

### **Best Practices and Key Takeaways**

* Use **interfaces** to define the structure of objects and ensure type consistency.
* Use **abstract classes** for base classes that may include shared functionality.
* Leverage **access modifiers** (public, private, protected) to control access to class members.
* Use **getters and setters** to encapsulate and control property access.
* Use **static members** for shared utility methods or constants.

### **Module 5: Advanced TypeScript Concepts**

This module dives into advanced TypeScript concepts, including generics, type guards, utility types, advanced type manipulation, and Type Declaration Files. These topics help in building robust, reusable, and scalable TypeScript applications.

### **1. Generics: Using Generics in Functions, Classes, and Interfaces**

**Generics** in TypeScript allow you to create reusable components (functions, classes, interfaces) that work with different types. Instead of specifying exact types, you can define a **type placeholder** that can be replaced with different types later.

#### **1.1 Generics in Functions**

Generics make functions flexible while still maintaining type safety.

Example:

typescript

CopyEdit

function identity<T>(value: T): T {

return value;

}

console.log(identity<string>("Hello")); // Output: Hello

console.log(identity<number>(42)); // Output: 42

Explanation:

* <T> is a type parameter that acts as a placeholder for any type.
* When calling identity, TypeScript infers the type based on the argument or it can be specified explicitly.

#### **1.2 Generics in Classes**

Classes can also be made generic, allowing them to handle various data types.

Example:

typescript

CopyEdit

class Box<T> {

private value: T;

constructor(value: T) {

this.value = value;

}

getValue(): T {

return this.value;

}

}

const stringBox = new Box<string>("A String");

console.log(stringBox.getValue()); // Output: A String

const numberBox = new Box<number>(123);

console.log(numberBox.getValue()); // Output: 123

#### **1.3 Generics in Interfaces**

You can use generics in interfaces to define flexible and reusable contracts.

Example:

typescript

CopyEdit

interface Result<T> {

data: T;

error: string | null;

}

const successResult: Result<string> = { data: "Success", error: null };

const failureResult: Result<number> = { data: 0, error: "An error occurred" };

### **2. Type Guards and Conditional Types**

Type guards and conditional types allow TypeScript to narrow down and infer types at runtime based on certain conditions.

#### **2.1 Type Guards**

Type guards are used to refine types inside conditional blocks by using typeof, instanceof, or custom type predicates.

Example:

typescript

CopyEdit

console.log(typeof “")

function printLength(value: string | number): void {

if (typeof value === "string") {

console.log(`String length: ${value.length}`); // Type narrowed to string

} else {

console.log(`Number value: ${value}`); // Type narrowed to number

}

}

* **typeof Type Guard**: Used to check primitive types like string, number, and boolean.

**instanceof Type Guard**: Used to check if an object is an instance of a class.  
 Example:  
  
 typescript  
CopyEdit  
class Animal {}

class Dog extends Animal {}

function checkAnimal(animal: Animal) {

if (animal instanceof Dog) {

console.log("This is a dog.");

}

}

#### **2.2 Custom Type Guards (Type Predicates)**

You can define your own type guard function using a **type predicate**.

Example:

typescript

CopyEdit

interface Cat {

name: string;

meow: () => void;

}

interface Dog {

name: string;

bark: () => void;

}

function isCat(animal: Cat | Dog): animal is Cat {

return (animal as Cat).meow !== undefined;

}

const pet: Cat | Dog = { name: "Fluffy", meow: () => console.log("Meow!") };

if (isCat(pet)) {

pet.meow(); // TypeScript knows this is a Cat

}

### **3. Utility Types (Partial, Readonly, Record, Pick, Omit)**

TypeScript provides built-in **utility types** that help transform and manipulate types.

#### **3.1 Partial<Type>**

Makes all properties of a type optional.

Example:

interface User {

name: string;

age: number;

email: string;

}

let partialUser: Partial<User> = { name: "Alice" }; // Other properties are optional

#### **3.2 Readonly<Type>**

Makes all properties of a type read-only.

Example:

typescript

CopyEdit

const user: Readonly<User> = { name: "Alice", age: 25, email: "alice@example.com" };

// user.age = 30; // Error: Cannot assign to 'age' because it is a read-only property

#### **3.3 Record<Keys, Type>**

Creates a type with a set of keys and a specific type for the values.

Example:

typescript

CopyEdit

type Role = "admin" | "user" | "guest";

const userRoles: Record<Role, string> = {

admin: "Administrator",

user: "Regular User",

guest: "Guest User",

};

#### **3.4 Pick<Type, Keys>**

Creates a type by picking specific properties from an existing type.

Example:

typescript

CopyEdit

type UserSummary = Pick<User, "name" | "email">;

const summary: UserSummary = { name: "Alice", email: "alice@example.com" };

#### **3.5 Omit<Type, Keys>**

Creates a type by omitting specific properties from an existing type.

Example:

typescript

CopyEdit

type UserWithoutEmail = Omit<User, "email">;

const userWithoutEmail: UserWithoutEmail = { name: "Alice", age: 25 };

### **4. Advanced Type Manipulation: Keyof, Mapped Types**

#### **4.1 keyof Operator**

The keyof operator returns the keys of a type as a union of string literals.

Example:

typescript

CopyEdit

type UserKeys = keyof User; // "name" | "age" | "email"

#### **4.2 Mapped Types**

Mapped types allow you to create new types by transforming properties in an existing type.

Example:

typescript

CopyEdit

type OptionalUser = { [K in keyof User]?: User[K] }; // Makes all properties optional

### **5. Type Declaration Files and Modules (.d.ts Files)**

Type Declaration Files (.d.ts files) provide type information for JavaScript libraries that don't have built-in TypeScript types.

#### **5.1 Purpose of .d.ts Files**

* They allow TypeScript to understand JavaScript code by providing type information.
* Useful when integrating with JavaScript libraries.

Example Declaration File (math.d.ts):

typescript

CopyEdit

declare module "math" {

export function add(a: number, b: number): number;

}

This allows you to use the add function in a TypeScript file as if it were type-safe:

typescript

CopyEdit

import { add } from "math";

console.log(add(10, 20)); // Type-safe usage

#### **5.2 Creating Custom Declaration Files**

For custom JavaScript modules, you can create your own .d.ts files to define types.

Example:

**Create myModule.js:** javascript  
CopyEdit  
export function greet(name) {

return `Hello, ${name}`;

}

**Create myModule.d.ts:** typescript  
CopyEdit  
declare module "./myModule" {

export function greet(name: string): string;

}

**Use in TypeScript File:** typescript  
CopyEdit  
import { greet } from "./myModule";

console.log(greet("Alice")); // Type-safe

### **Summary of Key Advanced Concepts**

* **Generics** allow creating reusable components that work with multiple types.
* **Type Guards and Conditional Types** help refine types during runtime.
* **Utility Types** simplify common type transformations (e.g., Partial, Readonly, Pick, Omit).
* **Advanced Type Manipulation** (keyof, Mapped Types) enhances type flexibility.
* **Type Declaration Files (.d.ts)** provide type information for JavaScript modules, enabling seamless integration.

### **Module 6: Working with TypeScript and JavaScript**

In this module, you will learn how to integrate TypeScript into existing JavaScript projects, configure TypeScript settings, debug TypeScript code, and handle errors using best practices. Mastering these concepts will help streamline development and improve the reliability of your applications.

### **1. Integrating TypeScript into Existing JavaScript Projects**

Integrating TypeScript into an existing JavaScript project can be done incrementally. TypeScript supports working alongside JavaScript files, allowing a gradual migration.

#### **Step 1: Install TypeScript**

Install TypeScript globally or as a project dependency:

bash

CopyEdit

npm install --save-dev typescript

#### **Step 2: Initialize TypeScript Configuration**

Create a tsconfig.json file to configure the TypeScript compiler:

bash

CopyEdit

npx tsc --init

#### **Step 3: Allow JavaScript Files in the Project**

In the tsconfig.json file, enable JavaScript compatibility:

json

CopyEdit

{

"compilerOptions": {

"allowJs": true, // Allow JavaScript files

"checkJs": true, // Enable type checking for JavaScript files

"outDir": "./dist" // Specify the output folder for compiled files

},

"include": ["src/\*\*/\*"], // Include all files in the src folder

}

#### **Step 4: Migrate JavaScript to TypeScript**

1. **Rename Files:** Change .js files to .ts incrementally.
2. **Add Type Annotations:** Start adding type annotations to variables, functions, and classes.

**Leverage JSDoc Comments:** TypeScript can infer types from JSDoc comments in JavaScript files.  
 Example:  
  
 javascript  
CopyEdit  
/\*\*

\* @param {number} a

\* @param {number} b

\* @returns {number}

\*/

function add(a, b) {

return a + b;

}

### **2. TypeScript Configuration (tsconfig.json)**

The tsconfig.json file is the heart of TypeScript configuration. It controls how TypeScript compiles the code.

Example tsconfig.json:

json

CopyEdit

{

"compilerOptions": {

"target": "es6", // Set the target JavaScript version (e.g., es5, es6)

"module": "commonjs", // Specify the module system (e.g., commonjs, esnext)

"strict": true, // Enable strict type checking

"outDir": "./dist", // Output directory for compiled JavaScript files

"rootDir": "./src", // Root directory for TypeScript files

"allowJs": true, // Allow JavaScript files in the project

"esModuleInterop": true, // Enable ES module interoperability

},

"include": ["src/\*\*/\*"], // Include TypeScript files in the src folder

"exclude": ["node\_modules"] // Exclude unnecessary folders

}

#### **Key Configuration Options:**

* **target:** Specifies the ECMAScript version (e.g., ES5, ES6).
* **module:** Defines the module system (commonjs, ESNext, etc.).
* **strict:** Enables strict type-checking options (e.g., strict null checks).
* **outDir and rootDir:** Specify where the compiled JavaScript files should be placed and the root of the TypeScript files.
* **allowJs:** Allows JavaScript files to be part of the project, useful for gradual migration.
* **esModuleInterop:** Improves compatibility with ES module imports.

### **3. Debugging TypeScript Code**

Debugging TypeScript code involves inspecting the compiled JavaScript, using source maps, and leveraging IDE features.

#### **Step 1: Enable Source Maps**

Add the following in tsconfig.json to generate source maps, which map compiled JavaScript back to the original TypeScript code:

json

CopyEdit

{

"compilerOptions": {

"sourceMap": true // Generate source maps for debugging

}

}

#### **Step 2: Debugging in Visual Studio Code (VS Code)**

1. **Open VS Code** and go to the Run and Debug tab.
2. **Add a launch configuration:**
   * Click create a launch.json file.
   * Select Node.js as the environment.

Modify the launch.json to include TypeScript debugging:  
  
 json  
CopyEdit  
{

"version": "0.2.0",

"configurations": [

{

"type": "node",

"request": "launch",

"name": "Launch Program",

"program": "${workspaceFolder}/src/index.ts",

"preLaunchTask": "tsc: build - tsconfig.json",

"outFiles": ["${workspaceFolder}/dist/\*\*/\*.js"]

}

]

}

1. **Set Breakpoints:** Set breakpoints in the TypeScript code by clicking on the left side of the line number.
2. **Start Debugging:** Press F5 to start debugging.

### **4. Handling Errors and Best Practices**

Handling errors in TypeScript improves code robustness and reliability. Below are common strategies and best practices:

#### **4.1 Error Handling with try-catch**

Use try-catch blocks to handle runtime errors. Type annotations can help detect possible error types.

Example:

typescript

CopyEdit

function parseJSON(jsonString: string): any {

try {

return JSON.parse(jsonString);

} catch (error) {

console.error("Failed to parse JSON:", error);

return null;

}

}

#### **4.2 Best Practices for TypeScript Projects**

**Enable Strict Mode:** Enabling strict mode in tsconfig.json helps catch potential issues:  
  
 json  
CopyEdit  
{

"compilerOptions": {

"strict": true

}

}

1. This includes:  
   * **strictNullChecks:** Prevents assigning null or undefined to non-nullable types.
   * **noImplicitAny:** Ensures that all variables have explicitly defined types.
2. **Use Type Annotations Wisely:**
   * Use explicit types for function parameters and return values.

Example:  
  
 typescript  
CopyEdit  
function add(a: number, b: number): number {

return a + b;

}

1. **Leverage Utility Types:** Use utility types like Partial, Readonly, Pick, and Omit to simplify type transformations.
2. **Avoid any When Possible:** Avoid the any type as it disables type checking and reduces the benefits of TypeScript.

**Use unknown Instead of any:** When handling values with unknown types, use unknown and apply type checks.  
 Example:  
  
 typescript  
CopyEdit  
function handleValue(value: unknown): void {

if (typeof value === "string") {

console.log("It's a string:", value.toUpperCase());

} else {

console.log("Unknown type");

}

}

**Organize Code with Modules:** Split large codebases into modules to improve maintainability.  
 Example:  
  
 typescript  
CopyEdit  
// math.ts

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

return a + b;

}

// index.ts

import { add } from "./math";

console.log(add(2, 3)); // Output: 5

### 

### **Summary of Key Concepts**

* **Integrating TypeScript into JavaScript Projects:** Allow JavaScript files, configure tsconfig.json, and migrate incrementally.
* **Configuring TypeScript (tsconfig.json):** Set up compiler options like strict, target, module, and outDir.
* **Debugging TypeScript:** Use source maps and IDE debugging tools to troubleshoot TypeScript code.
* **Handling Errors and Best Practices:** Use try-catch for error handling, enable strict mode, and avoid the any type for better type safety.

### **Module 7: Testing and Deployment in TypeScript**

### This module covers the essential aspects of **testing and deploying TypeScript projects**. You’ll learn how to write unit tests with Jest and Mocha, enforce type-safe and clean code using ESLint and Prettier, and understand how to compile, build, and deploy TypeScript applications.

### 

### **1. Unit Testing in TypeScript (Jest, Mocha)**

### **Unit testing** is the process of testing individual units or components of code to ensure they work as expected. Popular testing frameworks for TypeScript include **Jest** and **Mocha**.

#### **1.1 Setting Up Jest for TypeScript**

### **Jest** is a popular testing framework that works well with TypeScript.

### **Step 1: Install Required Packages**

### bash

### CopyEdit

### npm install --save-dev jest @types/jest ts-jest

### 

### **Step 2: Configure Jest** Create a configuration file jest.config.js:

### javascript

### CopyEdit

### module.exports = {

### preset: "ts-jest", // Use ts-jest to handle TypeScript files

### testEnvironment: "node", // Set the test environment to Node.js

### };

### 

### **Step 3: Write a Test in TypeScript** Example test file: math.test.ts:

### typescript

### CopyEdit

### // math.ts

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

### return a + b;

### }

### 

### // math.test.ts

### import { add } from "./math";

### 

### test("adds 2 + 3 to equal 5", () => {

### expect(add(2, 3)).toBe(5); // Assertion

### });

### 

### **Step 4: Run the Tests** Add a test script in package.json:

### json

### CopyEdit

### "scripts": {

### "test": "jest"

### }

### 

### Then, run the tests:

### bash

### CopyEdit

### npm test

### 

### 

#### **1.2 Setting Up Mocha and Chai for TypeScript**

### **Mocha** is a flexible testing framework, and **Chai** is commonly used with it for assertions.

### **Step 1: Install Mocha, Chai, and TypeScript Support**

### bash

### CopyEdit

### npm install --save-dev mocha chai ts-node @types/mocha @types/chai

### 

### **Step 2: Write a Mocha Test in TypeScript** Create a test file math.test.ts:

### typescript

### CopyEdit

### import { expect } from "chai";

### import { add } from "./math";

### 

### describe("Addition", () => {

### it("should return 5 when adding 2 and 3", () => {

### expect(add(2, 3)).to.equal(5);

### });

### });

### 

### **Step 3: Run the Tests** Add the test script to package.json:

### json

### CopyEdit

### "scripts": {

### "test": "mocha -r ts-node/register tests/\*\*/\*.test.ts"

### }

### 

### Run the tests with:

### bash

### CopyEdit

### npm test

### 

### 

### **2. Type-Safe Code with ESLint and Prettier**

### Maintaining clean, consistent, and type-safe code is crucial for any TypeScript project. **ESLint** is a popular tool for identifying and fixing problems in JavaScript and TypeScript code, and **Prettier** is used for code formatting.

#### **2.1 Setting Up ESLint for TypeScript**

### **Step 1: Install ESLint and TypeScript Plugins**

### bash

### CopyEdit

### npm install --save-dev eslint @typescript-eslint/parser @typescript-eslint/eslint-plugin

### 

### **Step 2: Configure ESLint** Create an ESLint configuration file .eslintrc.json:

### json

### CopyEdit

### {

### "parser": "@typescript-eslint/parser", // Specifies TypeScript parser

### "plugins": ["@typescript-eslint"], // Enable TypeScript rules

### "extends": [

### "eslint:recommended",

### "plugin:@typescript-eslint/recommended"

### ],

### "rules": {

### "semi": ["error", "always"], // Enforce semicolons

### "@typescript-eslint/explicit-module-boundary-types": "off"

### }

### }

### 

### **Step 3: Run ESLint** Add an ESLint script to package.json:

### json

### CopyEdit

### "scripts": {

### "lint": "eslint . --ext .ts"

### }

### 

### Run ESLint:

### bash

### CopyEdit

### npm run lint

### 

### 

#### **2.2 Setting Up Prettier for Code Formatting**

### **Step 1: Install Prettier and TypeScript Plugin**

### bash

### CopyEdit

### npm install --save-dev prettier eslint-config-prettier eslint-plugin-prettier

### 

### **Step 2: Configure Prettier** Create a .prettierrc file:

### json

### CopyEdit

### {

### "semi": true,

### "singleQuote": true

### }

### 

### **Step 3: Configure ESLint to Use Prettier** Update .eslintrc.json:

### json

### CopyEdit

### {

### "extends": [

### "eslint:recommended",

### "plugin:@typescript-eslint/recommended",

### "prettier" // Disables ESLint rules that conflict with Prettier

### ],

### "plugins": ["prettier"],

### "rules": {

### "prettier/prettier": "error" // Show Prettier errors as ESLint errors

### }

### }

### 

### **Step 4: Format Code** Add a format script to package.json:

### json

### CopyEdit

### "scripts": {

### "format": "prettier --write ."

### }

### 

### Run the formatter:

### bash

### CopyEdit

### npm run format

### 

### 

### **3. Compiling, Building, and Deploying TypeScript Projects**

### Once your TypeScript project is complete, the next step is to compile it into JavaScript, build the final output, and deploy it.

#### **3.1 Compiling TypeScript Code**

### Compile TypeScript to JavaScript using the TypeScript compiler:

### bash

### CopyEdit

### npx tsc

### 

### The compiled JavaScript files will be placed in the outDir specified in tsconfig.json.

### 

#### **3.2 Building TypeScript Projects**

### You can automate the build process by creating a custom build script.

### **Example Build Script:** Update package.json:

### json

### CopyEdit

### "scripts": {

### "build": "tsc && npm run lint && npm test"

### }

### 

### This script compiles the TypeScript code, lints the code, and runs the tests before building the project.

### 

#### **3.3 Deploying TypeScript Projects**

### **Deploying Node.js Applications (Example Deployment on Heroku):**

### **Create Procfile** (specifies how to start the app): bash CopyEdit echo "web: node dist/index.js" > Procfile

### 

### **Deploy to Heroku:** bash CopyEdit heroku login

### git init

### git add .

### git commit -m "Initial commit"

### heroku create

### git push heroku master

### 

### 

### **Best Practices for Testing, Linting, and Deployment**

### **Write Tests Early:** Write unit tests while developing to catch bugs early.

### **Use Linting and Formatting:** Maintain consistent code style with ESLint and Prettier.

### **Automate the Build Process:** Use build scripts to automate compiling, linting, and testing.

### **Enable Strict Mode:** Use strict options in TypeScript to enhance type safety.

### **Use Continuous Integration (CI):** Integrate your TypeScript project with CI tools like GitHub Actions or Jenkins to automate tests, linting, and deployment.

### 