

TypeScript Guide

✓ Setup

1. Install Node.js

[Download Node.js](#)

2. Check Node.js version

```
node -v
```

3. Install TypeScript globally

```
npm install -g typescript
```

4. Check TypeScript version

```
tsc -v
```

First Project

5. Create a TypeScript file

```
index.ts
```

6. Check available **tsc** commands

```
tsc --help  
# or  
tsc -h
```

7. Create a TypeScript configuration file

```
tsc --init
```

This will generate a `tsconfig.json` file where you can configure TypeScript options.

8. Compile TypeScript to JavaScript

```
tsc index.ts      # Compile one file
tsc index.ts -w   # Watch file changes
tsc               # Compile all files
tsc -w            # Watch all files
```

9. Run the compiled JavaScript

```
node index.js
```

Data Types

String

```
let name: string = "Mohamed";
```

Number

```
let age: number = 35;
```

Boolean

```
let isWorking: boolean = true;
```

Any

```
let anything: any = "Name";
anything = 1;
anything = true;
```

Union (Declare multiple types for one variable)

```
let value: string | number;
value = "Elhussin";
```

```
value = 123;
```

Array

```
let numbers: number[] = [1, 2, 3];
let strings: string[] = ["a", "b"];
let mixed: (number | string)[] = [1, "a"];
let nested: (number | string | string[])[] = [1, "a", ["x", "y"]];
// Or
let nums: Array<number> = [1, 2, 3];
```

Enum (Named constants)

```
enum Direction {
  Up,
  Down,
  Left,
  Right
}
enum Scour {
  A = 1,
  B = 2,
  C = 3,
  d = options.option1 + 1, // can use other enum values
  fun = funcEnum(5) // can use function
}
```

Tuple

Tuple is an array with fixed number of elements and fixed types. Tuple allow to define the type of each element in the array.

```
let person: [string, number] = ["Ali", 30];
person[0] = "Taha";
person[1] = 35;
```

Object

Object is a collection of properties.

```
let user: object = { name: "Elhussin", age: 30 };
```

Type Alias

Type alias is a way to give a name to a type.

```
type ID = string | number;
let userId: ID = 123;
type button = {
  Up: string;
  right: string;
  down: string;
  left: string
}
```

Extend type

```
type Lest = button & {
  x: boolean
}
```

ReadOnly

```
let readonlyArray: readonly number[] = [1, 2, 3];
```

Optional Properties (?)

Optional properties are properties that are not required.

```
interface User {
  name: string;
  age?: number;
}
```

Type Assertions

Type assertions are used to tell the compiler that a value is of a specific type. Change type of variable manually.

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

Void

void is a type that represents the absence of a value.

```
function logMessage(msg: string): void {  
  console.log(msg);  
}
```

Functions

Function Annotations

Function annotations are used to tell the compiler the return type of a function.

```
function greet(name: string, show: boolean): string {  
  return show ? `Hello ${name}` : "No greeting";  
}
```

Default Parameters

Default parameters are parameters that have a default value.

```
function greet(name: string = "User"): string {  
  return `Hello ${name}`;  
}
```

Rest Parameters

Rest parameters are used to pass multiple parameters to a function.

```
function sum(name: string, ...nums: number[]): string {  
  let total = nums.reduce((a, b) => a + b, 0);  
  return `${name} - Total: ${total}`;  
}
```

Arrow Function

Arrow function is a shorter way to write a function.

```
const add = (a: number, b: number): number => a + b;
```

Anonymous Function

Anonymous function is a function without a name.

```
const multiply = function (a: number, b: number): number {  
  return a * b;  
};
```

Interface

```
interface Person {  
  readonly name: string;  
  age?: number;  
  isEmployed: boolean;  
  greet(): string;  
  salaryWithTax: (amount: number) => number;  
}  
  
// reopen interface  
interface Person {  
  role: string;  
}  
  
interface ExtendedPerson extends Person {  
  firstName: string;  
}
```

Interface VS Type Aliases

- Interface can be extended but type cannot
- Type can be used with primitive types but interface cannot
- Interface can be merged but type cannot
- Interface is more readable than type

Class

```
class User {  
  constructor(private _name: string, public age: number, public readonly employed:  
    boolean) {}  
  
  get name(): string {  
    return this._name;  
  }  
  
  set name(value: string) {
```

```
    this._name = value;
  }

  greet = () => `Hello ${this._name}`;

  salaryWithTax(amount: number): number {
    return amount * 0.8;
  }
}
```

Access Modifiers

Modifier	Description
public	Accessible from anywhere
private	Accessible only within the class
protected	Accessible within the class and inheritance

Class static

Static members are members that are accessible without creating an instance of the class.

```
class TryStatic {
  static count: number = 0; // static property & can be private
  static increment(): void {
    console.log(`${this.count} objects created `)
  }

  constructor(public name: string) {
    TryStatic.count++;
  }
}
```

Class Implementation

Implementation is a way to provide the body of a method.

```
interface User {
  theme: boolean;
  font?: string;
  save(): void;
}

class UserSetting implements User {
```

```
    constructor(public username: string, public theme: boolean, public font?:
string, ){}
    save(): void {
        console.log("User setting saved");
    }
    update(): void {
        console.log("User setting updated");
    }
}
```

Abstract Class

Abstract class is a class that cannot be instantiated.

```
abstract class Food {
    constructor(public name: string) {}
    abstract make(): void;
}

class Pizza extends Food {
    constructor(public name: string) {
        super(name);
    }
    make(): void {
        console.log(`Making ${this.name}`);
    }
}
```

Class Inheritance

Inheritance is a way to create a new class from an existing class.

```
class Employee extends UserSetting {
    constructor(public username: string, public theme: boolean, public font?:
string, ){}
    save(): void {
        console.log("Employee setting saved");
    }
    update(): void {
        console.log("Employee setting updated");
    }
}
```

Class Polymorphism & Overried

Polymorphism is a way to provide the body of a method.


```
class Player {
  constructor(public name: string) {}
  play(): void {
    console.log(`${this.name} is playing`);
  }
  attack(): void {
    console.log(`Attacking Now`);
  }
}

class Goalie extends Player {
  constructor(name: string, public spears: number) {
    super(name); // call parent constructor
  }
  override attack(): void {
    // super.attack();
    console.log("Attacking With Spear")
    this.spears -= 1;
  }
}
```

Generics

Generics is a way to create reusable components that can work with different types.

Generics with multiple types

```
function testType<T>(arg: T): string {
  return `${arg} is of type ${typeof arg}`;
}
```

Generics with Arrow function

```
const genericArrow = <T>(arg: T): T => arg;
```

Generic class

```
class GenericClass<T> {
  constructor(public value: T) {}

  getValue(msg: T) {
    console.log(`${msg} : ${this.value}`);
  }
}
```

Generic type with interfaces

```
interface book {
  title: string;
  author?: string;
  price: number;
}

class collectionian<T>{
  constructor(public items: T[] = []) {}
  add(item: T): void {
    this.items.push(item);
  }
}

const addBoke = new collectionian<book>();
addBoke.add({title: "Book", author: "Elzero", price: 100});
console.log(addBoke);
```

Configuration Tip

tsconfig.json setting

```
"noImplicitAny": true
```

Enable error reporting for expressions and declarations with an implied 'any' type.

tsconfig.json setting

```
"noImplicitOverride": true
```

Ensure overriding members in derived classes are marked with an override modifier.

checkJs

```
"checkJs": true
```

Enable error reporting in type-checked JavaScript files.

Summary Table

Concept	Example	Notes
---------	---------	-------

Concept	Example	Notes
string	<code>let a: string = "hello";</code>	Text
number	<code>let a: number = 123;</code>	All numbers
boolean	<code>let a: boolean = true;</code>	true/false
any	<code>let a: any = "x"; a = 1;</code>	Any type (use with care)
union	<code>let a: string number;</code>	Multiple types
array	<code>let a: number[] = [1,2];</code>	List of elements
tuple	<code>let a: [string, number]</code>	Fixed types and order
enum	<code>enum X { A, B }</code>	Named constants
alias	<code>type ID = string number</code>	Custom name for a type
readonly	<code>readonly name: string;</code>	Cannot change after assignment
optional	<code>age?: number</code>	Not required
object	<code>{ name: string; age: number }</code>	General object
type assertion	<code>value as string</code>	Override type
void	<code>function(): void {}</code>	No return
never	<code>function(): never { throw ... }</code>	Never returns
interface	<code>interface User {}</code>	Describe object structure
class	<code>class User {}</code>	OOP model
access mods	<code>private, public, protected</code>	Scope of properties/methods
getter/setter	<code>get name() {}</code>	Control access to properties
tsconfig	<code>"noImplicitAny": true</code>	Force strict typing