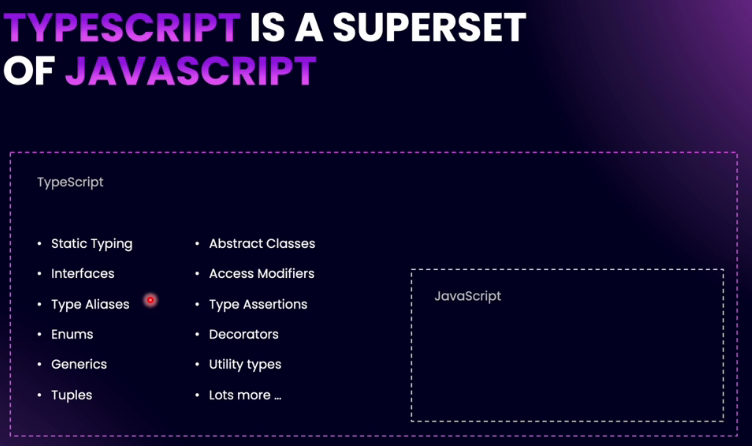
# Section 1: Introduction

5. What Is TypeScript



6. Setup Development Environment

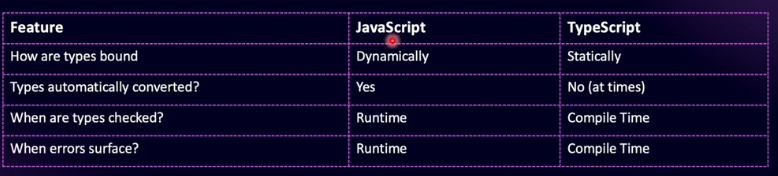
Npm i -D typescript

7. Setting Up TypeScript Compiler

8. How TypeScript Helps You Catch Errors

9. Your First TypeScript Program

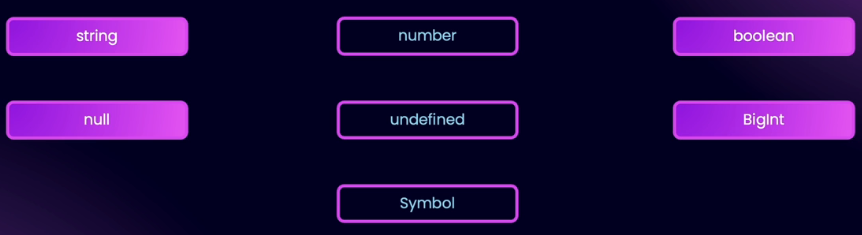
10. Deep Dive into the JavaScript File Generated by TypeScript



11. TypeScript Type System

# Section 2: Primitive Types

12. Introduction To Primitive Types



13. string, number and boolean

Type annotation

Type inference

14. null and undefined Types

15. bigint Type

const safeint = Number.MAX\_SAFE\_INTEGER;

console.log(safeint) // 9007199254740991

let bigInt1: bigint = BigInt(1234);

let bigInt2: bigint = 12341234n;

let c = bigInt1 - bigInt2;

console.log(Math.log(bigInt1))

// error: Argument of type 'bigint' is not assignable to parameter of type 'number'.ts(2345)

16. symbol Type

let id: symbol = Symbol(1234)

let alphabeticId: symbol = Symbol("id")

let user = {

    [id]: "1234",

    name: "Mark",

    getId() {

        return this[id]

    }

}

// id is private.

console.log(user.name)

console.log(user.id) // error : Property 'id' does not exist on type '{ [x: symbol]: string; name: string; }'.ts(2339)

console.log(user)

// { name: 'Mark', [Symbol(1234)]: '1234' }

console.log(user.getId())

Quiz:

1. Top of Form

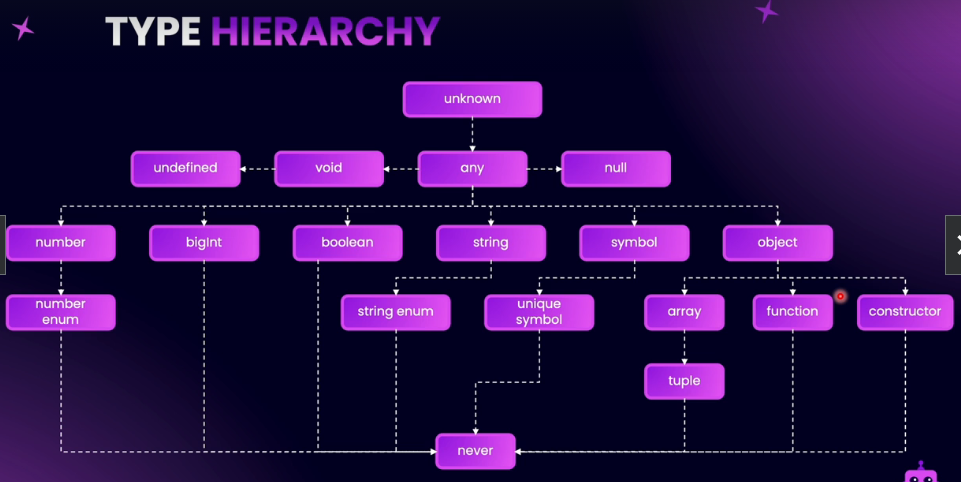
Why using the 'unknown' type is a better option than using the 'any' type?

* 'unknown' type is not better than any time. They are the same.
* **Because it forces the developer to type check before performing any operation on the value.**
* The other way around 'any' type is better than the 'unknown' type**.**

Bottom of Form

# Section 3: Exploring TypeScript's Type System

19. Introduction To Type System

 20. any Type

21. unknown type

22. Type Alias

23. Inference, Annotation, Duck Typing and Declaration

// As naming convention use Upper Camel Case or also called PascalCase

type CustomString = string;

type CustomNumberType = number;

type CustomDate = Date;

type CustomSymbol = Symbol;

// Duck Typing is used by TypeScript for inference of types

//\* "If it looks like a duck and quacks like a duck, it's a duck".

// Type declaration

let firstName: CustomString = "Mark";

let age: CustomNumberType = 32;

let today: CustomDate = new Date(); // Object instantiation hence has a constructor and new keyword

let unique: CustomSymbol = Symbol(); // Primitive does not require new keyword

// If you do not assign a type to a variable TS will still try to infer it

// If I try to assign another type to firstName it will throw error

let lastName = "Doe";

lastName = 25;

// This is an example of a declaration

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

    return a + b;

}

// TypeScript is able to infer that final result will be a number

let finalResult = addNumber(10, 15);

24. Union Types

25. Quick Conditional Types

26. Understanding Type Hierarchy

// A subtype will always exten the parent type

type check = any extends unknown ? true : false;

type check2 = string extends any ? true : false;

type check3 = number extends any ? true : false;

type check4 = {} extends Object ? true : false;

type check5 = [] extends Object ? true : false;

type check6 = Function extends Object ? true : false;

type check7 = never extends any ? true : false;

// All are true

// A funtion delclaration that never completes and returns a never type

const throwError = (errorMsg: string) => {

    throw new Error(errorMsg);

};

let strings: Object = ["a", "b"];

let myFunc: Function = () => 2;

27. Type Casting

// Types can be casted other types.

// This shoudl be done when you are sure that TypeScript is not able to infer the types correctly

let firstName = <any>"Mark";

let lastName = "Doe" as any;

let user = {

    name: "Mark",

    email: "mark@email.com",

};

type User = {

    name: string;

    email: string;

};

function fetchUser() {

    return user as User;

}

const fetchedData = fetchUser();

Quiz

let city = "New York"; // string

let population = 8400000; //number

const age = 32; // literal value of 32

let oldAge = 79 as const; //  79

let newAge = oldAge; // 79

let data = new Map(); // Map<any,any>

let score = [90, 86, 100]; // number[]

type Primitive = string | number | boolean; //string | number | boolean

type CustomName = "John" extends string ? string : "John"; //string

type CustomAge = typeof newAge extends number ? 79 : number; // 79

type CheckData = typeof data extends Object ? true : false; // true

type CheckScore = typeof score extends never ? {} : []; // []

/\*\*

 \* Are the following statements valid

 \* Check if below lines of code are valid as per TypeScript or not without uncommenting them

 \*  \*/

// age = 85; //cannot assign a constant

// score.push(10); // valid

// score.push("New Score"); // string is not assignable to number

// let customAge: CustomAge = 50; // 50 is not assignable to type 79

// let primitive: Primitive = new Date(); // type Date is not assignable to type primitive

// let years: CheckScore = []; // valid

Section 4: Objects In TypeScript

30. Introduction To Objects

31. Working With Objects

let person = {

name: "Mark",

age: 32,

};

// Car is any object without a shape or defined properties

let car: Object = {

brand: "BMW",

color: "black",

};

// The problem using teh Object type, its a super type and Arrays are also objects

let score: Object = [];

// If you see just because car is a Object I can re-assign it as an array

// ! This is a problem

car = [21, 32, 48];

// Defining an object with the litral syntax

let newCar: {} = {

brand: "BMW",

color: "black",

};

// The problem with usign the litral object {} is that now properties are not defined and can vary

// This can lead to errors in the application

newCar = {

brand: "BMW",

};

32. Type Alias For Objects

// A post can be strictly typed using type annotations

// let post: {

// title: string; // Type Annotations using semi-colons instead of commas

// content: string;

// date: Date;

// } = {

// title: "This is a blog post",

// content: "Content of the post",

// date: new Date(),

// };

// We can use Type declarations using Type Alias

type Post = {

title: string; // Type Annotations using semi-colons instead of commas

content: string;

date: Date;

};

let post: Post = {

title: "This is a blog post",

content: "Content of the post",

date: new Date(),

};

// Advantage of type declaration, can use the same type again for a new object

let post2: Post = {

title: "This is a blog post 2",

content: "Content of the post 2",

date: new Date(),

};

33. Nested Objects

// One of having nested objects

// type Post = {

// title: string; // Type Annotations using semi-colons instead of commas

// content: string;

// date: Date;

// author: {

// name: string;

// age: number;

// email: string;

// };

// };

// Better to declare a separate type for Author

type Author = {

name: string;

age: number;

email: string;

};

type Post = {

title: string; // Type Annotations using semi-colons instead of commas

content: string;

date: Date;

author: Author; // Assign Author type to author property on Post type

};

let post: Post = {

title: "This is a blog post",

content: "Content of the post",

date: new Date(),

author: {

name: "John",

age: 32,

email: "john@doe.com",

},

};

34. Index Signatures With Objects

// One of having nested objects

// type Post = {

// title: string; // Type Annotations using semi-colons instead of commas

// content: string;

// date: Date;

// author: {

// name: string;

// age: number;

// email: string;

// };

// };

// Better to declare a separate type for Author

type Author = {

name: string;

age: number;

email: string;

};

// Declare a new type for Awards

// Initial declaration

// type Awards = {

// [keyof: string]: {

// name: string;

// date: Date;

// };

// };

type AwardDetails = {

name: string;

date: Date;

};

type Awards = {

[keyof: string]: AwardDetails;

};

type Post = {

title: string; // Type Annotations using semi-colons instead of commas

content: string;

date: Date;

author: Author; // Assign Author type to author property on Post type

awards: Awards;

};

let post: Post = {

title: "This is a blog post",

content: "Content of the post",

date: new Date(),

author: {

name: "John",

age: 32,

email: "john@doe.com",

},

awards: {

web: {

name: "Wed Awards",

date: new Date(),

},

web3: {

name: "Web 3",

date: new Date(),

},

},

};

35. Optional And Readonly Properties

// One of having nested objects

// type Post = {

// title: string; // Type Annotations using semi-colons instead of commas

// content: string;

// date: Date;

// author: {

// name: string;

// age: number;

// email: string;

// };

// };

// Better to declare a separate type for Author

type Author = {

name: string;

age: number;

email: string;

readonly type: "human" | "ai";

};

// Declare a new type for Awards

// Initial declaration

// type Awards = {

// [keyof: string]: {

// name: string;

// date: Date;

// };

// };

type AwardDetails = {

name: string;

date: Date;

};

type Awards = {

[keyof: string]: AwardDetails;

};

type Post = {

title: string; // Type Annotations using semi-colons instead of commas

content: string;

date: Date;

author: Author; // Assign Author type to author property on Post type

awards: Awards;

category?: string;

};

let post: Post = {

title: "This is a blog post",

content: "Content of the post",

date: new Date(),

category: "javascript",

author: {

name: "John",

age: 32,

email: "john@doe.com",

type: "human",

},

awards: {

web: {

name: "Wed Awards",

date: new Date(),

},

web3: {

name: "Web 3",

date: new Date(),

},

},

};

let post2: Post = {

title: "This is a blog post",

content: "Content of the post",

date: new Date(),

author: {

name: "John",

age: 32,

email: "john@doe.com",

type: "human",

},

awards: {

web: {

name: "Wed Awards",

date: new Date(),

},

},

};

// Readonly properties once created cannot be re-assigned

post2.author.type = "ai";

36. Union Types With Objects

// Declare a type for the Dog

type Dog = {

name: string;

barks: boolean;

wags: boolean;

};

// Declare a type for the Cat

type Cat = {

name: string;

purrs: boolean;

};

// Create a new type which is a union of Dog and Cat

type DogAndCatUnion = Dog | Cat;

// All Dog properties

let dog: DogAndCatUnion = {

name: "Buddy",

barks: true,

wags: true,

};

// All Cat properties

let cat: DogAndCatUnion = {

name: "Bella",

purrs: true,

};

// All Dog and partial cat properties

let dogAndCat: DogAndCatUnion = {

name: "Hybrid",

barks: true,

wags: true,

purrs: true,

};

// Cannot contain partial Properties of one of the types

let partialDog: DogAndCatUnion = {

name: "Hybrid",

barks: true,

};

37. Discriminating Unions

// To Discriminate Unions All the three types must have the state property

type NetworkLoadingState = {

state: "loading";

};

type NetworkFailedState = {

state: "failed";

code: number;

};

type NetworkSuccessState = {

state: "success";

response: {

title: string;

duration: number;

summary: string;

};

};

// We Want to create a Network State with includes Loading, Failed and Success

// Create a type which represents only one of the above types

// but you aren't sure which it is yet.

type NetworkState = NetworkLoadingState | NetworkFailedState | NetworkSuccessState;

// Based on the types created now we can discriminate the network state and take action based on the state

// We need to create a logger function which logs the state of the network

function logger(state: NetworkState): string {

// Right now TypeScript does not know which of the three

// potential types state could be.

// Trying to access a property which isn't shared

// across all types will raise an error

//! state.code;

// By switching on state, TypeScript can narrow the union

// down in code flow analysis

switch (state.state) {

case "loading":

return "Downloading...";

case "failed":

// The type must be NetworkFailedState here,

// so accessing the `code` field is safe

return `Error ${state.code} downloading`;

case "success":

return `Downloaded ${state.response.title} - ${state.response.summary}`;

}

}

38. Intersection Types

type Cat = {

name: string;

purrs: boolean;

color: string;

};

type Dog = {

name: string;

barks: boolean;

color: string;

};

type HybridAnimal = Dog & Cat;

const hybridAnimal: HybridAnimal = {

name: "Max",

color: "White",

barks: true,

purrs: false,

};

Section 5: Arrays And Enums

41. Introduction To Arrays And Enums

42. Strictly Typing Arrays

// Declaring an array of numbers

let a: number[] = [1, 2, 3];

// Generic syntax of declaring array of strings

let b: Array<string> = ["a", "b", "c"];

// Array containing mutiple types

let c: (string | number)[] = ["a", "b", 1];

// Declare a Caterer type

type Caterer = {

name: string;

address: string;

phone: number;

};

// Declare a Seats type

type Seats = {

[keyof: string]: string;

};

// Declare a Airplane Type

type Airplane = {

model: string;

flightNumber: string;

timeOfDeparture: Date;

timeOfArrival: Date;

caterer: Caterer;

seats: Seats;

};

// Declarign a type for an array of Airplanes

// Example of an array of objects

type Airplanes = Airplane[];

// Assign Airplane Type to Object

let airplanes: Airplanes = [

{

model: "Airbus A380",

flightNumber: "A2201",

timeOfDeparture: new Date(),

timeOfArrival: new Date(),

caterer: {

name: "Special Food Ltd",

address: "484, Some Street, New York",

phone: 7867856751,

},

seats: {

A1: "John Doe",

A2: "Mark Doe",

A3: "Sam Doe",

},

},

];

43. Tuples

// What if the person array needs to have a fixed set of values

// We create a Tuple for such a situation

let person: [string, string, number];

person = ["John", "Doe", 18];

// A user array can have optional properties as well

let user: [string, string, number, string?];

user = ["Mark", "Doe", 21, "mark@doe.com"];

// Tuple with multiple string value which do not exist at the time of declaration

type listOfStudents = [number, boolean, ...string[]];

const passingStudents: listOfStudents = [3, true, "John", "Stella", "Mark"];

const failingStudents: listOfStudents = [1, false, "Scott"];

// Tuples with any number of value in the beginning or end

// Type Alias can be used with Tuples as well

type StringBooleansNumber = [string, ...boolean[], number];

type BooleansStringNumber = [...boolean[], string, number];

let stringBooleanNumber: StringBooleansNumber = ["string", true, false, 32];

let booleanStringNumber: BooleansStringNumber = [true, false, "string", 32];

44. Readonly Arrays and Tuples

// An array can be converted into readonly using the readopnly keyword

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

// Once an array is readonly no values can be added or removed from an array

number.push(1);

// A tuple can be read only also

type ReadOnlyTuple = readonly [string, string, number];

// For our example let's create a new Tuple which is readonly

type ReadOnlyPerson = readonly [string, string, number];

// Creating a new readonly person

const person: ReadOnlyPerson = ["John", "Smith", 32];

// There are some alternavites for creating Readonly Arrays

type a = Readonly<string[]>;

type b = ReadonlyArray<string>;

// Alternative syntax for Readonly Tuple

type c = Readonly<[string, string, number]>;

45. Enums

// Why Are enums needed?

// We ofetn declare constants in JavaScript for eg.

const STATUS\_LOADING = "loading";

const STATUS\_STOPPED = "stopped";

// The intension of declaring these constants is that we do not want the value to change

// because these cannot be reassiged to developers bny mistake cannot do this

// JavaScript will not throw an error but will not reassign the constant it will always remain loading

STATUS\_LOADING = "stopped";

// Auto incrementing values given to enums

// Single enum can contain all values

enum Direction {

Up,

Down,

Left,

Right,

}

// Assigning the first number and the rest would auto increment

enum Direction2 {

Up = 1,

Down,

Left,

Right,

}

// String Enums

export enum Roles {

admin = "admin",

author = "author",

editor = "editor",

}

// Use case for enums

type Person = {

name: string;

email: string;

password: string;

role: Roles;

};

const person: Person = {

name: "John",

email: "john@email.com",

password: "password",

role: Roles.admin,

};

// Enums can be hetrogeneous as well

// Assigning the first number and the rest would auto increment

enum Direction3 {

Up = 1,

Down = "Down",

Left = "Left",

}

46. Enums are Available At Runtime

// Auto incrementing values given to enums

enum Direction {

Up,

Down,

Left,

Right,

}

// Assigning the first number and the rest would auto increment

enum Direction2 {

Up = 1,

Down,

Left,

Right,

}

// String Enums

export enum Roles {

admin = "admin",

author = "author",

editor = "editor",

}

// Use case for enums

type Person = {

name: string;

email: string;

password: string;

role: Roles;

};

const person: Person = {

name: "John",

email: "john@email.com",

password: "password",

role: Roles.admin,

};

// Enums can be hetrogeneous as well

// Assigning the first number and the rest would auto increment

enum Direction3 {

Up = 1,

Down = "Down",

Left = "Left",

}

// Enums are available in JavaScript as Objects

console.log(Roles);

47. Enums Vs Objects

// Redeclaring the same using Enum

// const Enum, is not compiled in JavaScript as an Object but only the value is used

const enum EDirection {

Up,

Down,

Left,

Right,

}

// Show the use of value

let eDirection = EDirection.Up;

// Declaring an object with same values as a constant

// Typescript sets each property as readonly

const ODirection = {

Up: 0,

Down: 1,

Left: 2,

Right: 3,

} as const;

// Now this acts as an enum because you cannot change the value of properties

console.log("Object as const", ODirection.Up);

ODirection.Up = "newValue";

48. Computed Enums

// Auto incrementing values given to enums

enum Direction {

Up,

Down,

Left,

Right,

}

// Assigning the first number and the rest would auto increment

enum Direction2 {

Up = 1,

Down,

Left,

Right,

}

// String Enums

export enum Roles {

admin = "admin",

author = "author",

editor = "editor",

}

// Use case for enums

type Person = {

name: string;

email: string;

password: string;

role: Roles;

};

const person: Person = {

name: "John",

email: "john@email.com",

password: "password",

role: Roles.admin,

};

// Enums can be hetrogeneous as well

// Assigning the first number and the rest would auto increment

enum Direction3 {

Up = 1,

Down = "Down",

Left = "Left",

}

// Enums are available in JavaScript as Objects

console.log(Roles);

// Redeclaring the same using Enum

// const Enum, is not compiled in JavaScript as an Object but only the value is used

const enum EDirection {

Up,

Down,

Left,

Right,

}

// Show the use of value

let eDirection = EDirection.Up;

// Declaring an object with same values as a constant

// Typescript sets each property as readonly

const ODirection = {

Up: 0,

Down: 1,

Left: 2,

Right: 3,

} as const;

// Now this acts as an enum because you cannot change the value of properties

console.log("Object as const", ODirection.Up);

ODirection.Up = "newValue";

// Enums can contain computed values as well

enum AccessPermission {

None = 0,

Read = 1,

Write = 2,

ReadWrite = Read + Write,

Delete = 4,

All = ReadWrite | Delete,

}

console.log(AccessPermission.ReadWrite);

console.log(AccessPermission.All);

49. Enums as Unions and Types

// We have an enume for shapes

enum ShapeKind {

Circle = "circle",

Square = "sqaure",

}

// A circle type would have additional properties

type Circle = {

kind: ShapeKind.Circle;

radius: number;

};

// A sqaure type would have additional properties

type Square = {

kind: ShapeKind.Square;

sideLength: number;

};

// Now enuma act as types as TS is able to identify that square enum cannot be assigned to circle type

let circle: Circle = {

kind: ShapeKind.Square,

radius: 100,

};

// Enums automatically become union of each of its members

// Here the ShapeKind Enum is acting as a union of ""

function printShape(shape: ShapeKind /\* same as "Circle"| "Square" \*/) {

// We can check teh shape in step one

console.log(`Shape is: ${shape}`);

// Also an or comparison TS will say that this is already an enum so this comparison is not needed

if (ShapeKind.Circle || ShapeKind.Square) {

}

}

// We can invoke the function

printShape(ShapeKind.Circle);

Quiz

// Practice Questions

//\* 1. Create an array numbers that only accepts numbers and another array strings that only accepts strings.

let numbers: number[] = [1, 2, 3];

let strings: string[] = ["a", "b", "c"];

//\* 2. Create a tuple person that holds a string (name) and a number (age).

let person: [string, number] = ["Alice", 30];

//\* 3. Create a readonly array colors that holds strings and a readonly tuple point that holds two numbers (x, y). Attempt to modify their elements and observe the TypeScript error.

const colors: readonly string[] = ["red", "green", "blue"];

const point: readonly [number, number] = [10, 20];

//\* 4. Create an enum called StatusEnum that should 3 properties Active, Inactive, Pending

enum StatusEnum {

Active = "active",

Inactive = "inactive",

Pending = "pending",

}

//\* 5. Create an object as const called Status with the same structure as an StatusEnum

const Status = {

Active: "active",

Inactive: "inactive",

Pending: "pending",

} as const;

Section 6: Functions In TypeScript

52. Declaring Functions

// Defining a named function in TypeScript

function intro(name: string, age: number): string {

return `My name is ${name} and I am ${age} years old`;

}

// Using a fucntion expression

const intro2 = function (name: string, age: number): string {

return `My name is ${name} and I am ${age} years old`;

};

// Using the arrow function syntax

const intro3 = (name: string, age: number): string => {

return `My name is ${name} and I am ${age} years old`;

};

53. Default and Optional Parameters

// Defining a named function in TypeScript

// Functions often need optional params

// We can add optional params by using ? just like we do with objects, PAUSE AND PRACTICE

function intro(name: string, age: number, country?: string): string {

if (country) {

return `My name is ${name} and I am ${age} years old, I live in ${country}`;

}

return `My name is ${name} and I am ${age} years old`;

}

// TypeScript will throw an error if all defualt params are not added as arguments

// The error displays while you are programming and not at runtime

//! intro("John");

intro("John", 32);

54. Custom Parameters And Return Types

enum AgeUnit {

Years = "years",

Months = "months",

}

type Person = {

name: string;

age: number;

ageUnit: AgeUnit;

country: string;

};

let person: Person = {

name: "Scott",

age: 30,

ageUnit: AgeUnit.Years,

country: "USA",

};

function convertAgeToMonths(person: Person): Person {

if (person.ageUnit === AgeUnit.Years) {

person.age = person.age \* 12;

person.ageUnit = AgeUnit.Months;

}

return person;

}

console.log(convertAgeToMonths(person));

55. Function Call Signatures

enum AgeUnit {

Years = "years",

Months = "months",

}

// Greeting can be defined as a type as well

type GreetingFunction = (greeting: string /\* can have additional params \*/) => string;

type Person = {

name: string;

age: number;

ageUnit: AgeUnit;

country: string;

// greet: Function;

greet: GreetingFunction;

};

let person: Person = {

name: "Scott",

age: 30,

ageUnit: AgeUnit.Years,

country: "USA",

greet: (greeting) => {

return `${greeting} ${person.name}`;

},

};

function convertAgeToMonths(person: Person): Person {

if (person.ageUnit === AgeUnit.Years) {

person.age = person.age \* 12;

person.ageUnit = AgeUnit.Months;

}

return person;

}

console.log(convertAgeToMonths(person));

console.log(person.greet("Hello"));

56. Anonymous Functions

const students = ["Alice", "Bob", "Mark"];

// Lets assume that you are looping through the students array

// Since students is an array fo strings even when using annonymous function like this

// TypeScript is able to correctly infer the type of each student

students.map((student) => {

console.log(student);

});

// This also works with the function defined using the function keyword and not just the arrow functions

students.map(function (student) {

console.log(student);

});

57. void and never Types

// At times there are functions which do not return anything

// this function does not return anything

// Hence for rerturn type we can use a special TypeScript type called void

function writeToDatabase(value: string): void {

console.log("Writing to database:", value);

}

// This is different from void because this function never completes execution

function throwError(error: string): never {

throw new Error(error);

}

// We check these types and hence void can be used in place of never but not visa versa

type check = never extends void ? true : false;

type checks = void extends never ? true : false;

58. Async Functions

// Async fucntion in JavaScript always return a promise

// Declaration of async function using the function keyword

async function fetchFromDatabase(id: number): Promise<any> {}

// Declaration of async fucntion using the arrow function syntax

const anotherAsyncFunction = async (): Promise<any> => {};

// Async fucntion as a function express

const fucntionExpression = async function (): Promise<any> {};

// Setting return types apart from any

async function returnString(id: number): Promise<string> {

return Promise.resolve("string");

}

type User = {

name: string;

age: number;

};

// If User type is nto returned TS throws an error

async function returnObject(id: number): Promise<User> {

return Promise.resolve({ name: "John", age: 21 });

}

59. Rest Parameters And Arguments

// Unlimited function params using the

function multiplyBy(by: number, ...numbers: number[]) {

return numbers.map((eachNumber) => by \* eachNumber);

}

// Calling the function

console.log(multiplyBy(2, 3, 4, 5));

console.log(multiplyBy(2, 3, 4));

// Strictly typing rest parameters

const args = [8, 5];

// We get this erro because if we see the signature of the function it takes in 2 arguments

// Here TS s not sure that args will always contain 2 arguments

const angle = Math.atan2(...args);

// The solution is to define args as a tuple

const args1 = [8, 5] as const;

// Now TypeScript will not thwo an error because it will now know that args are always of a fixed length because its a tuple

const angle1 = Math.atan2(...args1);

// We can define type annotate it like this as well

const args2: [number, number] = [8, 5]; // Here using as const is not needed

const angle2 = Math.atan2(...args2);

60. Parameter Destructuring

// Parmaeter destructuring is also possible just like JavaScript

type Numbers = {

a: number;

b: number;

c: number;

};

// Declare and object of numbers

let numbers: Numbers = {

a: 2,

b: 3,

c: 4,

};

// create a function to print numbers

// function sum(numbers: Numbers) {

// console.log(numbers);

// }

// Destructure numbers

// TS will infer each of the destructured params correctly as number types

function sum({ a, b, c }: Numbers) {

return a + b + c;

}

// While invoking the function you pass the numbers object and

// destructuring with type inference will play the role

console.log(sum(numbers));

// wrong type will lead to an error

console.log(sum({ a: 3, b: 4, c: "c" }));

61. Introduction to Function Overloading

// Example if function overloading

// JavaScritp functions as Dynamic in natrue they can accept different arguments and based on that return results

// Here is an example of the same

// Show the slice method signature to show optional params

const str = "Hello, World!";

const part1 = str.slice(7);

const part2 = str.slice(7, 10);

// See this returns a different result and its up to you how you invoke the function

// Whether you use single argument or two JS functions are Dynamic to return results

// based on the arguments passed to the function

console.log(part1);

console.log(part2);

// What if we hd to struictly type such function overloads?

// Hypothtical AirTicket Reservation System

type Reservation = {

departureDate: Date;

returnDate: Date;

departingFrom: string;

destination: string;

};

// Declare a call signature

// type Reserve = (

// departureDate: Date,

// returnDate: Date,

// departingFrom: string,

// destination: string

// ) => Reservation;

// Using a more explicit syntax for call signatures

// This is considered as an object with keys as function params

type Reserve = {

(departureDate: Date, returnDate: Date, departingFrom: string, destination: string): Reservation;

};

62. Function Overloading In TypeScript

// Example if function overloading

// JavaScritp functions as Dynamic in natrue they can accept different arguments and based on that return results

// Here is an example of the same

// Show the slice method signature to show optional params

const str = "Hello, World!";

const part1 = str.slice(7);

const part2 = str.slice(7, 10);

// See this returns a different result and its up to you how you invoke the function

// Whether you use single argument or two JS functions are Dynamic to return results

// based on the arguments passed to the function

console.log(part1);

console.log(part2);

// What if we hd to struictly type such function overloads?

// Hypothtical AirTicket Reservation System

type Reservation = {

departureDate: Date;

returnDate?: Date;

departingFrom: string;

destination: string;

};

// Declare a call signature

// type Reserve = (

// departureDate: Date,

// returnDate: Date,

// departingFrom: string,

// destination: string

// ) => Reservation;

// Using a more explicit syntax for call signatures

// This is considered as an object with keys as function params

type Reserve = {

(departureDate: Date, returnDate: Date, departingFrom: string, destination: string):

| Reservation

| never;

(departureDate: Date, departingFrom: string, destination: string): Reservation | never;

};

const reserve: Reserve = (

departureDate: Date,

returnDateOrDepartingFrom: Date | string,

departingFromOrDestination: string,

destination?: string

) => {

if (returnDateOrDepartingFrom instanceof Date && destination) {

return {

departureDate: departureDate,

returnDate: returnDateOrDepartingFrom,

departingFrom: departingFromOrDestination,

destination: destination,

};

} else if (typeof returnDateOrDepartingFrom === "string") {

return {

departureDate: departureDate,

departingFrom: returnDateOrDepartingFrom,

destination: departingFromOrDestination,

};

}

throw new Error("Please provide valid details to reserve a ticket");

};

// Try making a reservation

console.log(reserve(new Date(), new Date(), "New York", "Washington"));

console.log(reserve(new Date(), "New York", "Washington"));

Quiz

/\*\*

\* Practice Excercise for functions

\*/

//\* 1. Declare a function named greet that takes a string parameter name and returns a greeting message.

function greet(name: string): string {

return `Hello, ${name}!`;

}

//\* 2. Define an type Product with properties id (number) and name (string). Create a function named getProduct that takes an id parameter and returns a Product.

interface Product {

id: number;

name: string;

}

function getProduct(id: number): Product {

return { id, name: `Product ${id}` };

}

//\* 3. Declare a function signature named Calculator as a type that takes two numbers and returns a number. Implement two functions add and subtract that match this signature.

type Calculator = (a: number, b: number) => number;

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

const subtract: Calculator = (a, b) => a - b;

//\* 4. Create a function named logMessage that takes a string message and logs it to the console, returning void. Also, create a function named throwError that takes a string message and throws an error, returning never.

function logMessage(message: string): void {

console.log(message);

}

function throwError(message: string): never {

throw new Error(message);

}

Section 7: Generics In TypeScript

65. What are Generics

// Let us look at this simple function which just returns the param

//\* function returnParam(param) {

//\* return param;

//\* }

// TS has a problem and is warning that param is of any type or is implicitly declared at any

//\* function returnParam(param: any) {

//\* return param;

//\* }

// While using any is certainly generic in that it will cause the function to accept any and all types for the type of arg, we actually are losing the information about what that type was when the function returns. If we passed in a number, the only information we have is that any type could be returned.

// Since now we do not know the return type of this function we end up usign the return type as

// any as well. We are back to JS behaviour

//\* function returnParam(param: any): any {

//\* return param;

//\* }

// Here, we will use a type variable, a special kind of variable that works on types rather than values. This is a generic variable we can pass to our function to retain the type of the value used

// This is how it is done in case of functions

function returnParam<Type>(param: Type): Type {

return param;

}

// With this generic in place the value of the type used to invoke the function is retained

// thi is one way of invoking the function

let stringOutput = returnParam<string>("string");

//Here we explicitly set Type to be string as one of the arguments to the function call, denoted using the <> around the arguments rather than ().

// The second way is also perhaps the most common. Here we use type argument inference — that is, we want the compiler to set the value of Type for us automatically based on the type of the argument we pass in:

let stringOutput2 = returnParam("string");

let numberOutput = returnParam(100);

let numberArray = returnParam([1, 2, 3]);

let objectOutput = returnParam({ name: "Mark", age: 21 });

// Generic function declaration as an arrow function

// using a call signature

const myParam: <T>(param: T) => T = (param) => param;

// Using a function expression

const myParam2 = function <U>(param: U): U {

return param;

};

// Using a call signature in an object

type ObjectType = {

myParam: <V>(param: V) => V;

};

66. Generic Function Declarations

// Declare Generic Type

type MyParam = <AnyName>(param: AnyName) => AnyName;

// First declare a generics function signature

type GetFirstElement = <T>(arr: T[]) => T;

// A generic array function that gets first element of every type of array

const getFirstElement: GetFirstElement = (arr) => {

return arr[0];

};

// We declare two different tyopes of array

const numberArray = [1, 2, 3];

const stringArray = ["a", "b", "c"];

// Typescript is correctly able to infer the value that will be return by expression

// Even though the function is the same the returned type is different based on the input value

let stringOutput = getFirstElement(stringArray);

let numberOutput = getFirstElement(numberArray);

// Where do declare generic dictates when typescript will binc a concrete type to a generic

// what if the above function was declared with a different placement of generic

type FirstElement<T> = (arr: T[]) => T;

67. Generic and Constraints With Arrays

// First declare a generics function signature

type GetFirstElement = <T>(arr: T[]) => T;

// A generic array function that gets first element of every type of array

const getFirstElement: GetFirstElement = (arr) => {

return arr[0];

};

// We declare two different tyopes of array

const numberArray = [1, 2, 3];

const stringArray = ["a", "b", "c"];

// Typescript is correctly able to infer the value that will be return by expression

// Even though the function is the same the returned type is different based on the input value

let stringOutput = getFirstElement(stringArray);

let numberOutput = getFirstElement(numberArray);

// Where do declare generic dictates when typescript will binc a concrete type to a generic

// what if the above function was declared with a different placement of generic

type FirstElement<T> = (arr: T[]) => T;

// Here if the generic type is not passed at the time of function declaration TS will throw and error

// So now you need to tell TypeScript which types can we used with this fucntion which solves a completely different purpose from the function that has been declared above

// Hover over the function param and you will see that TS now identifies that param will be an

// array of strings

const firstElement: FirstElement<string> = (arr) => {

return arr[0];

};

// Generics can have constraints as well

type HasLength = {

length: number;

};

function logLength<T extends HasLength>(item: T): void {

console.log(item.length);

}

// Any array like value that has a length property on it will be accepted as an argument

logLength(numberArray);

logLength(stringArray);

logLength("Any String");

// But if used for an object it will throw an error

logLength({ name: "John", length: 12 });

68. Generics With Objects

type KeyValuePair<K, V> = {

key: K;

value: V;

};

const stringNumberPair: KeyValuePair<string, number> = {

key: "age",

value: 30,

};

const numberBooleanPair: KeyValuePair<number, boolean> = {

key: 1,

value: true,

};

console.log(stringNumberPair); // Output: { key: 'age', value: 30 }

console.log(numberBooleanPair); // Output: { key: 1, value: true }

/\*\*

\* Generics Constraints With Objects

\*/

type HasId = {

id: number;

};

function printId<T extends HasId>(obj: T): void {

console.log(obj.id);

}

const user = {

id: 1,

name: "Alice",

};

printId(user); // Output: 1

const product = {

id: 101,

name: "Laptop",

};

printId(product); // Output: 101

69. keyof Type Operator

type Events = {

id: number;

date: Date;

type: "indoor" | "outdoor";

};

// The keyof operator takes an object type and produces a string or numeric literal union of its keys. The following type P is the same type

// -> "id" | "date" | "type"

type UnionOfKeysOfEvents = keyof Events;

// You see these are literally the union of name of the keys of the Events object

let idOfEvent: UnionOfKeysOfEvents = "id";

let dateOfEvent: UnionOfKeysOfEvents = "date";

// If index signatures where keys are defined as numeric properties

type Numeric = {

[key: number]: string;

};

type NumericKeyOf = keyof Numeric;

type NumberAndString = {

[key: string]: string;

};

// We get a union of numbers as well as a string because this is how JavaScript objects work behind the scenes

// NumberAndString is string | number — this is because JavaScript object keys are always coerced to a string, so obj[0] is always the same as obj["0"].

type NumberAndStringKeyoff = keyof NumberAndString;

let stringObject: NumberAndString = {

0: "first",

second: "first",

};

// Accessing the object proerty with the index of the property

console.log(stringObject["0"]);

// Declaring partial types using generics and keyof

type Person = {

name: string;

age: number;

address: string;

};

// Creating a type where the keys are the same as Person but the values are optional and nullable

// Hover over PartialPerson to see how TypeScript is inferring it

type PartialPerson = {

[K in keyof Person]?: Person[K] | null;

};

let partial: PartialPerson = {

    name: "John"

}

console.log(partial)

// {name:”John”}

70. Generic Default Values

// Define a generic function to fetch data with a default type

async function fetchData<T = any>(url: string): Promise<T> {

const response = await fetch(url);

const data: T = await response.json();

return data;

}

// Using the fetchData function with the default type (any)

async function fetchDefault() {

const data = await fetchData("https://jsonplaceholder.typicode.com/posts/1");

console.log(data); // Output: any data structure, depends on the response

}

fetchDefault();

// Using the fetchData function with a specified type

// Lets declare a type based on the response that we get from the above fake API

interface Post {

userId: number;

id: number;

title: string;

body: string;

}

async function fetchPost() {

const post = await fetchData<Post>("https://jsonplaceholder.typicode.com/posts/1");

console.log(post); // Output: { userId: 1, id: 1, title: "...", body: "..." }

}

fetchPost();

71. Implementing A Polymorphic Function

// Trying to create a simple implementation of JavaScript's own filter method

const filter = (array: any[], predicate: Function) => {

let result: any[] = [];

for (let i = 0; i < array.length; i++) {

let item = array[i];

if (predicate(item)) {

result.push(item);

}

}

return result;

};

let numbers = [1, 3, 4, 6, 9, 7, 10, 12];

// Predicate to filter all numbers greater than 7

function predicate(item: number) {

return item > 7;

}

let animals = ["cat", "bat", "rat", "mat"];

// Predicate to filter all cats from animals array

function filterCats(item: string) {

return item === "cat";

}

// Result of invoking the function

console.log(filter(numbers, predicate));

console.log(filter(animals, filterCats));

72. Problems With Function Overloads

type Filter = {

(array: number[], predicate: (item: number) => boolean): number[];

(array: string[], predicate: (item: string) => boolean): string[];

(array: object[], predicate: (item: object) => boolean): object[];

};

// Trying to create a simple implementation of JavaScript's own filter method

const filter = (array: any[], predicate: Function) => {

let result: any[] = [];

for (let i = 0; i < array.length; i++) {

let item = array[i];

if (predicate(item)) {

result.push(item);

}

}

return result;

};

let numbers = [1, 3, 4, 6, 9, 7, 10, 12];

// Predicate to filter all numbers greater than 7

function predicate(item: number) {

return item > 7;

}

let animals = ["cat", "bat", "rat", "mat"];

// Predicate to filter all cats from animals array

function filterCats(item: string) {

return item === "cat";

}

// Result of invoking the function

console.log(filter(numbers, predicate));

console.log(filter(animals, filterCats));

73. Using Generics Instead Of Function Overloads

type Filter = {

(array: number[], predicate: (item: number) => boolean): number[];

(array: string[], predicate: (item: string) => boolean): string[];

(array: object[], predicate: (item: object) => boolean): object[];

};

// Trying to create a simple implementation of JavaScript's own filter method

const filter = <T>(array: T[], predicate: (item: T) => boolean): T[] => {

let result: any[] = [];

for (let i = 0; i < array.length; i++) {

let item = array[i];

if (predicate(item)) {

result.push(item);

}

}

return result;

};

let numbers = [1, 3, 4, 6, 9, 7, 10, 12];

// Predicate to filter all numbers greater than 7

function predicate(item: number) {

return item > 7;

}

let animals = ["cat", "bat", "rat", "mat"];

// Predicate to filter all cats from animals array

function filterCats(item: string) {

return item === "cat";

}

// Result of invoking the function

console.log(filter(numbers, predicate));

console.log(filter(animals, filterCats));

Quiz

Implement a map function

const map = <T, U>(array: T[], func: (item: T) => U): (U | T)[] => {

if (array.length === 0) {

return array;

}

let result: U[] = [];

for (let i = 0; i < array.length; i++) {

result.push(func(array[i]));

}

return result;

};

let numbers = [4, 5, 6, 7, 8, 9];

const converted = map(numbers, (num) => num.toString());

console.log(converted);

Section 8: Classes In TypeScript

76. What Are Classes

77. Running TypeScript In Browser

class User {

// This is how you define the properties of a class

name = "John";

email = "john@email.com";

greet() {

return `Hello John`;

}

}

// A class can be used to create an instance of itself

// An instance is an object that you create from a class

const user = new User();

const user2 = new User();

// This class generates the same object everytime

// We will see how to solve this in the next lecture

console.log(user);

console.log(user2);

console.log(user.greet());

78. Creating First Class And Instance

class User {

// This is how you define the properties of a class

name = "John";

email = "john@email.com";

greet() {

return `Hello John`;

}

}

// A class can be used to create an instance of itself

// An instance is an object that you create from a class

const user = new User();

const user2 = new User();

// This class generates the same object everytime

// We will see how to solve this in the next lecture

console.log(user);

console.log(user2);

console.log(user.greet());

79. Constructor Function

class User {

// This is how you define the properties of a class

name = "John";

email = "john@email.com";

constructor(name: string, email: string) {

console.log(name);

console.log(email);

}

greet() {

return `Hello John`;

}

}

// A class can be used to create an instance of itself

// An instance is an object that you create from a class

const user = new User("Mark", "Mark@email.com");

const user2 = new User("Alice", "alice@email.com");

// This class generates the same object everytime

// We will see how to solve this in the next lecture

console.log(user);

console.log(user2);

console.log(user.greet());

80. this Keyword

class User {

// This is how you define the properties of a class

name: string;

email: string;

constructor(name: string, email: string) {

this.name = name;

this.email = email;

}

greet() {

return `Hello ${this.name}`;

}

}

// A class can be used to create an instance of itself

// An instance is an object that you create from a class

const user = new User("Mark", "Mark@email.com");

const user2 = new User("Alice", "alice@email.com");

// The objects and greet methods are now dynamic in nature

console.log(user);

console.log(user2);

console.log(user.greet());

console.log(user2.greet());

81. Classes as Types

class User {

// This is how you define the properties of a class

name: string;

email: string;

constructor(name: string, email: string) {

this.name = name;

this.email = email;

}

greet() {

return `Hello ${this.name}`;

}

}

// A class is used as a type by TypeScript.

// Here user is inferred as a type of User class

// Should match the shape of the User class and can be annotated as a User object

const user: User = new User("Mark", "Mark@email.com");

const user2: User = new User("Alice", "alice@email.com");

// If I try to assign a property that does not exist on User class

// TS will throw an error because it understands that user is a typeof User

user.lastName = "Doe";

// Even assiging incorrect types will not work

// Here trying to assign a number to a property of type string will throw an error

user.name = 123;

// The objects and greet methods are now dynamic in nature

console.log(user);

console.log(user2);

console.log(user.greet());

console.log(user2.greet());

82. Optional And Readonly Fields

class User {

// This is how you define the properties of a class

name: string;

// making the email as readonly property

readonly email: string;

lastName?: string;

constructor(name: string, email: string, lastName?: string) {

this.name = name;

this.email = email;

this.lastName = lastName;

}

greet() {

return `Hello ${this.name}`;

}

}

// A class is used as a type by TypeScript.

// Here user is inferred as a type of User class

// Should match the shape of the User class and can be annotated as a User object

const user: User = new User("Mark", "Mark@email.com");

const user2: User = new User("Alice", "alice@email.com");

// Create a new user with an optional propery inside the class

const user3: User = new User("John", "John@email.com", "Doe");

// If I try to assign a property that does not exist on User class

// TS will throw an error because it understands that user is a typeof User

user.lastName = "Doe";

// Even assiging incorrect types will not work

// Here trying to assign a number to a property of type string will throw an error

user.name = 123;

// Trying to change the email now will also throw an errro because email is now readonly

user.email = "another@email.com";

// The objects and greet methods are now dynamic in nature

console.log(user);

console.log(user3);

83. Inheritance With Classes

class User {

// This is how you define the properties of a class

name: string;

// making the email as readonly property

readonly email: string;

lastName?: string;

constructor(name: string, email: string, lastName?: string) {

this.name = name;

this.email = email;

this.lastName = lastName;

}

greet() {

return `Hello ${this.name}`;

}

}

class Admin extends User {

isAdmin: boolean = true;

}

// A class is used as a type by TypeScript.

// Here user is inferred as a type of User class

// Should match the shape of the User class and can be annotated as a User object

const user: User = new User("Mark", "Mark@email.com");

const admin: Admin = new Admin("John", "John@email.com");

console.log(user);

console.log(admin);

84. super Method

class User {

// This is how you define the properties of a class

name: string;

// making the email as readonly property

readonly email: string;

lastName?: string;

constructor(name: string, email: string, lastName?: string) {

this.name = name;

this.email = email;

this.lastName = lastName;

}

greet() {

return `Hello ${this.name}`;

}

}

class Admin extends User {

isAdmin: boolean = true;

usersReporting: number;

// NOw the constructor will require the super method to invoke the constructor of parent class

// Till you do not add the constructor to the Admin class super method is not needed

// But the moment you add the constructor the super mothod is needed

constructor(name: string, email: string, usersReporting: number, lastName?: string) {

// It is important to invoke the super meothod with the same signature as the

// constructor of the parent class.

super(name, email, lastName);

// Super can be invoked with a processed value within a constructor is your choice.

// Its just that it has to be invoked

// super ("John", "john@email.com");

this.usersReporting = usersReporting;

}

}

const user: User = new User("Mark", "Mark@email.com");

// Now users reporting need to be added to initialiser

const admin: Admin = new Admin("John", "John@email.com", 11);

console.log(user);

console.log(admin);

Quiz

// 1. Class Definition

class Book {

readonly ISBN: string;

title: string;

author: string;

yearPublished?: number;

// 2. Constructor Function

constructor(title: string, author: string, ISBN: string, yearPublished?: number) {

// 3. This Keyword

this.title = title;

this.author = author;

this.ISBN = ISBN;

if (yearPublished) {

this.yearPublished = yearPublished;

}

}

}

// 4. Creating First Class And Instance

const firstBook = new Book("The Great Gatsby", "F. Scott", "1234567890", 1925);

console.log(firstBook);

// 5. Classes as Types

function logBookDetails(book: Book): void {

console.log(`Title: ${book.title}`);

console.log(`Author: ${book.author}`);

console.log(`ISBN: ${book.ISBN}`);

if (book.yearPublished) {

console.log(`Year Published: ${book.yearPublished}`);

}

}

logBookDetails(firstBook);

// 6. Inheritance With Classes

class EBook extends Book {

fileSize: number;

format: string;

// 7. super Method

constructor(

title: string,

author: string,

ISBN: string,

fileSize: number,

format: string,

yearPublished?: number

) {

super(title, author, ISBN, yearPublished);

this.fileSize = fileSize;

this.format = format;

}

}

const firstEBook = new EBook("The Great Gatsby", "F. Scott", "1234567890", 2, "PDF", 1925);

console.log(firstEBook);

logBookDetails(firstEBook);

87. Access Modifiers

88. public Members

class User {

// Convert name to public

public name: string;

readonly email: string;

// If an access modifier is not mentioned the default remains public

lastName?: string;

constructor(name: string, email: string, lastName?: string) {

this.name = name;

this.email = email;

this.lastName = lastName;

}

greet() {

return `Hello ${this.name}`;

}

}

class Admin extends User {

isAdmin: boolean = true;

usersReporting: number;

constructor(name: string, email: string, usersReporting: number, lastName?: string) {

super(name, email, lastName);

this.usersReporting = usersReporting;

}

// Public properties are accessible inside the child classes also

// even methods can have access modifiers not just the properties

public printName() {

console.log(this.name);

}

}

const user: User = new User("Mark", "Mark@email.com");

const admin: Admin = new Admin("John", "John@email.com", 11);

// These properties were assigned a value and these can be seen in the console

// Changing the properties below changes the values as these are public properties

user.name = "Alice";

admin.lastName = "Doe";

console.log(user);

console.log(admin);

// Since the printname method is public it can be accessed from the object itself

// or outside the class

admin.printName();

89. protected Members

class User {

public name: string;

readonly email: string;

lastName?: string;

// Adding a protected member

protected phone: number;

constructor(name: string, email: string, phone: number, lastName?: string) {

this.name = name;

this.email = email;

this.lastName = lastName;

this.phone = phone;

}

greet() {

return `Hello ${this.name}`;

}

}

class Admin extends User {

isAdmin: boolean = true;

usersReporting: number;

constructor(

name: string,

email: string,

phone: number,

usersReporting: number,

lastName?: string

) {

super(name, email, phone, lastName);

this.usersReporting = usersReporting;

}

public printName() {

console.log(this.name);

}

// Methods can use access modifiers as well

protected printPhone() {

console.log(this.phone);

}

// Declaring a method to show that protected members can be used inside classes and child

// classes as well

public useProtectedPhone() {

this.printPhone();

}

}

// Create users with phone numbers now

const user: User = new User("Mark", "Mark@email.com", 123456);

const admin: Admin = new Admin("John", "John@email.com", 123456, 11);

user.name = "Alice";

admin.lastName = "Doe";

// Protected members cannot be access in the final objects as they are only available in the

// parent class as well as the child class.

console.log(user.phone);

console.log(admin.phone);

// even protected methods are not available to us outside the class

admin.printPhone();

// but protected methods are available inside the parent class as well as the child class

admin.useProtectedPhone();

admin.printName();

90. private Members

class User {

public name: string;

readonly email: string;

lastName?: string;

// Changing the phone to private member

private phone: number;

constructor(name: string, email: string, phone: number, lastName?: string) {

this.name = name;

this.email = email;

this.lastName = lastName;

this.phone = phone;

}

greet() {

return `Hello ${this.name}`;

}

public printPhone() {

console.log(this.phone);

}

}

class Admin extends User {

isAdmin: boolean = true;

usersReporting: number;

constructor(

name: string,

email: string,

phone: number,

usersReporting: number,

lastName?: string

) {

// You can still use the super method to create a new user and assign the private

// member its value while instantiating the object from the class

super(name, email, phone, lastName);

this.usersReporting = usersReporting;

}

public printName() {

console.log(this.name);

}

// You will now see the error in TS

// This is bacause protected memebers can't be access even in child classes

// they can only be access in parent classes

protected printPhone() {

console.log(this.phone);

}

}

// Create users with phone numbers now

const user: User = new User("Mark", "Mark@email.com", 123456);

const admin: Admin = new Admin("John", "John@email.com", 123456, 11);

user.name = "Alice";

admin.lastName = "Doe";

// You will see that phone number can't be accessed on any of the objects

console.log(user.phone);

console.log(admin.phone);

admin.printName();

// You will see the method that is inside the class has the access to the

// private members of the class

console.log(user.printPhone());

91. Which Access Modifier to Use?

92. Overriding Methods In Child Classes

class User {

public name: string;

readonly email: string;

lastName?: string;

// Changing the phone to private member

private phone: number;

constructor(name: string, email: string, phone: number, lastName?: string) {

this.name = name;

this.email = email;

this.lastName = lastName;

this.phone = phone;

}

greet(): string {

return `Hello ${this.name}`;

}

}

class Admin extends User {

isAdmin: boolean = true;

usersReporting: number;

constructor(

name: string,

email: string,

phone: number,

usersReporting: number,

lastName?: string

) {

super(name, email, phone, lastName);

this.usersReporting = usersReporting;

}

// A greet methods can override the parent class method

// but the condition is that the child class methods must have the same signature

greet(): string {

return `Hello ${this.name}! I am the admin.`;

}

}

// Create users with phone numbers now

const user: User = new User("Mark", "Mark@email.com", 123456);

const admin: Admin = new Admin("John", "John@email.com", 123456, 11);

// Greet methods on both objects would return a different greeting

console.log(user.greet());

console.log(admin.greet());

93. Shorthand For Constructor

class User {

constructor(

public name: string,

readonly email: string,

private phone: number,

public lastName?: string

) {}

greet(): string {

return `Hello ${this.name}`;

}

}

class Admin extends User {

public isAdmin: boolean = true;

constructor(

name: string,

email: string,

phone: number,

public usersReporting: number,

lastName?: string

) {

super(name, email, phone, lastName);

this.usersReporting = usersReporting;

}

// A greet methods can override the parent class method

// but the condition is that the child class methods must have the same signature

greet(): string {

return `Hello ${this.name}! I am the admin.`;

}

}

// Create users with phone numbers now

const user: User = new User("Mark", "Mark@email.com", 123456);

const admin: Admin = new Admin("John", "John@email.com", 123456, 11);

// Greet methods on both objects would return a different greeting

console.log(user.greet());

console.log(admin.greet());

94. More Control Over Classes

class Person {

constructor(public firstname: string, public lastname: string, public age: number) {

// We want the age to be between a valid human age value

// It cannot be negative or above human life expectancy

// Writing conditional logic inside the constructor like this is not a good idea

if (age > 200 || age < 0) {

throw new Error("The age must be within range of 0-200");

}

}

public fullname() {

return this.firstname + this.lastname;

}

}

const john: Person = new Person("John", "Doe", 45);

const mark: Person = new Person("Mark", "Doe", 35);

// I can set the age to a negative value if age is remains as a public property

john.age = 210;

mark.age = -10;

console.log(john);

console.log(mark);

// Also if I need to combine properties like first and last name

// a method to do so is an ot an elegant solution

console.log(john.fullname());

// Something like john.fullname would be a better option to have

95. Using Mutators - Setters

class Person {

// This \_age property is for the internal use of the class and not exposed to outside world

private \_age?: number;

constructor(public firstname: string, public lastname: string) {}

// I need a better way to set the age of the person

// The logic needs to be encapsulated inside a method separate from constructor

public set age(age: number) {

if (age > 200 || age < 0) {

throw new Error("The age must be within range of 0-200");

}

this.\_age = age;

}

public fullname() {

return this.firstname + this.lastname;

}

}

const john: Person = new Person("John", "Doe");

const mark: Person = new Person("Mark", "Doe");

// Age can nw be set like a property and providing an invalid age will throw an error

john.age = 210;

mark.age = -10;

console.log(john);

console.log(mark);

// Also if I need to combine properties like first and last name

// a method to do so is an ot an elegant solution

console.log(john.fullname());

// Something like john.fullname would be a better option to have

96. Using Accessors - Getters

class Person {

// This \_age property is for the internal use of the class and not exposed to outside world

private \_age?: number;

constructor(public firstname: string, public lastname: string) {}

// I need a better way to set the age of the person

// The logic needs to be encapsulated inside a method separate from constructor

public set age(age: number) {

if (age > 200 || age < 0) {

throw new Error("The age must be within range of 0-200");

}

this.\_age = age;

}

public get age() {

if (this.\_age === undefined) {

throw new Error("The age property has not been set as yet");

}

return this.\_age;

}

public get fullname() {

return `${this.firstname} ${this.lastname}`;

}

}

const john: Person = new Person("John", "Doe");

const mark: Person = new Person("Mark", "Doe");

john.age = 50;

mark.age = 30;

// We can access the age like a property even though there is a method behind the scenes

// that is working to fetch the age for us using one of the private properties of a class

console.log(john.age);

console.log(mark.age);

// Now full name can be access on a class as if if was the property of the class

console.log(john.fullname);

97. Static Members

/\*\*

\*\* 1. Static members (fields and methods) belong to the class itself rather than to any instance of the class.

\*

\*\* 2. Static members are accessed using the class name, not the instance.

\*/

class Counter {

static count = 0; // Static field

static increment() {

// Static method

Counter.count++;

}

}

// Static members of a class can be accessed without instantiating the class

Counter.increment();

console.log(Counter.count); // Output: 1

// Static members are not available in the instances of classes

const counter = new Counter();

// Static members cannot be access on the instances of classes, TS throws an error

counter.increment();

98. Understanding Static Blocks

/\*\*

\*

\* ! Key Differences Between Static Blocks and Constructor Methods

\*\* Purpose:

Static Blocks: Used for one-time class-level initialization. They can set static properties and perform any setup that should happen once when the class is loaded.

Constructor Methods: Used for instance-level initialization. They set up instance properties and perform any setup needed each time an instance of the class is created.

\*\* Scope:

Static Blocks: Operate at the class level. They cannot directly access or initialize instance properties since they run before any instances are created.

Constructor Methods: Operate at the instance level. They can access and initialize instance properties using the this keyword.

\*\* Execution Timing:

Static Blocks: Executed once when the class is first loaded and defined, before any instances are created.

Constructor Methods: Executed each time a new instance of the class is created.

\*\* Access:

Static Blocks: Can access and modify static properties and call static methods.

Constructor Methods: Can access and modify instance properties and methods, and can also access static properties and methods using the class name.

\*

\*/

class Counter {

static count = 0; // Static field

static increment() {

// Static method

Counter.count++;

}

static {

// Static block

console.log("Initializing Counter class");

const initialCount = Counter.loadInitialCount();

Counter.count = initialCount;

}

static loadInitialCount(): number {

// Simulate loading initial count from an external source

// This is a simulation of data that is coming from an outside source like an external API

// This method could be inside as well as outside the class

return 5;

}

}

// The month the class is first Initialized with static members the static block triggers

// Which sets the initial count value of the class to 5

console.log(Counter.count); // Output: 5

// Then we can increment the count value and print it once again

Counter.increment();

console.log(Counter.count); // Output: 6

99. Generics With Classes

class Box<T> {

private \_value: T;

constructor(value: T) {

this.\_value = value;

}

get value(): T {

return this.\_value;

}

set value(newValue: T) {

this.\_value = newValue;

}

}

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

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

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

console.log(stringBox.value); // Output: Hello

100. Generics Use Case

type Identifiable = {

id: number;

};

// We need to declare a generic class that can hold items which have id's

// It can be a repository of any entity that has an id property associated with it

// The repository needs to have methods for adding, removing and getting entities

class Repository<T extends Identifiable> {

private items: T[] = [];

add(item: T): void {

this.items.push(item);

}

getById(id: number): T | undefined {

return this.items.find((item) => item.id === id);

}

getAll(): T[] {

return this.items;

}

removeById(id: number): void {

this.items = this.items.filter((item) => item.id !== id);

}

}

101. Concrete Implementation With User Type

type Identifiable = {

id: number;

};

// We need to declare a generic class that can hold items which have id's

// It can be a repository of any entity that has an id property associated with it

class Repository<T extends Identifiable> {

private items: T[] = [];

add(item: T): void {

this.items.push(item);

}

getById(id: number): T | undefined {

return this.items.find((item) => item.id === id);

}

getAll(): T[] {

return this.items;

}

removeById(id: number): void {

this.items = this.items.filter((item) => item.id !== id);

}

}

// Example usage with a User type

interface User extends Identifiable {

name: string;

email: string;

}

const userRepository = new Repository<User>();

userRepository.add({ id: 1, name: "Alice", email: "alice@example.com" });

userRepository.add({ id: 2, name: "Bob", email: "bob@example.com" });

console.log(userRepository.getById(1));

// Output: { id: 1, name: 'Alice', email: 'alice@example.com' }

console.log(userRepository.getAll());

// Output: [ { id: 1, name: 'Alice', email: 'alice@example.com' }, { id: 2, name: 'Bob', email: 'bob@example.com' } ]

userRepository.removeById(1);

console.log(userRepository.getAll());

// Output: [ { id: 2, name: 'Bob', email: 'bob@example.com' } ]

102. Intro To Mixins

103. Composing New Classes With Mixins

// We can declare a separate type for the constructor to get rid of the confusion

type Constructor = new (...args: any[]) => {};

// A mixin function that adds timestamp functionality

function Timestamp<TBase extends Constructor>(Base: TBase) {

// The function returns a new class that extends the passed Base class

return class extends Base {

protected timestamp: Date = new Date();

getTimestamp() {

return this.timestamp;

}

};

}

// Base User class

class User {

constructor(public name: string) {}

}

// Composing the final class using mixins

class UserWithTimestamp extends Timestamp(User) {

constructor(name: string, public age: number) {

super(name);

}

// Since timestamp is protected we can declare a displayInfor method to

// display the required information

displayInfo() {

console.log(`Name: ${this.name}, Age: ${this.age}`);

console.log(`Timestamp: ${this.getTimestamp()}`); // Call method from Timestamp mixin

}

}

const user = new UserWithTimestamp("Alice", 30);

user.displayInfo();

Quiz

/\*\*

\* ! You are developing a simple employee management system for a company. Implement the following requirements using TypeScript:

\*

\* TODO: 1. Class Definition: Create a class Employee with the following properties:

\*\* - name (string, public)

\*\* - age (number, public)

\*\* - salary (number, private)

\*\* - id (number, protected)

\*

\* TODO: 2. Use shorthand syntax in the constructor to initialize the properties name and age.

\*

\* TODO: 3. Implement getter and setter methods for the salary property. The setter should ensure the salary is a positive number.

\*

\* TODO: 4. Add a static property companyName (string, public) and a static method getCompanyName that returns the company name.

\*

\* TODO: 5. Create a subclass Manager that extends the Employee class. Add an additional property department (string, public).

\*

\* TODO: 6. Override a method getDetails in the Manager class to include the department information along with the employee details.

\*/

class Employee {

static companyName: string = "Tech Solutions Inc."; // Static member

constructor(

public name: string, // Public member using shorthand for constructor

public age: number, // Public member using shorthand for constructor

private \_salary: number, // Private member

protected id: number // Protected member

) {}

// Getter for salary

get salary(): number {

return this.\_salary;

}

// Setter for salary

set salary(newSalary: number) {

if (newSalary > 0) {

this.\_salary = newSalary;

} else {

throw new Error("Salary must be a positive number");

}

}

// Static method to get company name

static getCompanyName(): string {

return Employee.companyName;

}

// Method to get employee details

getDetails(): string {

return `Name: ${this.name}, Age: ${this.age}, Salary: ${this.salary}`;

}

}

class Manager extends Employee {

constructor(

name: string,

age: number,

salary: number,

id: number,

public department: string // Public member using shorthand for constructor

) {

super(name, age, salary, id);

}

// Overriding getDetails method to include department information

getDetails(): string {

return `${super.getDetails()}, Department: ${this.department}`;

}

}

// Example usage

const emp1 = new Employee("Alice", 30, 50000, 1);

console.log(emp1.getDetails()); // Output: Name: Alice, Age: 30, Salary: 50000

const manager1 = new Manager("Bob", 40, 70000, 2, "Engineering");

console.log(manager1.getDetails()); // Output: Name: Bob, Age: 40, Salary: 70000, Department: Engineering

console.log(Employee.getCompanyName()); // Output: Tech Solutions Inc.

Section 9: Abstract Classes And Interfaces

106. Module Introduction - Abstract Classes And Interfaces

107. Introduction To Abstract Classes

108. Shared methods in Abstract Classes

109. Protected Constructor And Child Classes

110. Adding Holidays To Classes

111. Print Holidays Method

112. Method Overriding In Child Class

113. Introduction to Abstract Methods

114. Introduction to Interfaces

115. Introduction to Interfaces

116. Extending Interfaces using extends keyword

117. Inheriting From Multiple Interfaces

118. Interfaces and Generics

119. Assigning Generics To Interfaces

120. Using Multiple Types as Generics

121. Using interfaces with classes

122. Multiple Classes Using the Same Interface

123. Implementing Multiple Interfaces

124. Multiple inheritance in classes using interfaces

125. Interfaces and Access Modifiers

126. Declaration Merging Interfaces

127. Difference Between a Type and an Interface

128. Difference Between a Abstract Class and an Interface

129. Difference Between Interfaces and Abstract Classes 2

130. What should you use? Abstract Class or Interfaces

Section 10: The TypeScript Compiler

131. Module Introduction - The TypeScript Compiler

132. Setting up a Base Project

133. Including and Excluding Files

134. rootDir and outDir

135. Setting a Compilation Target

136. TypeScript Core Libs

137. Type Checking Options

138. JS in Browser Using Liver Server

139. Understanding Source Maps

Section 11: Prototypes And Objects

140. Module Introduction - Prototypes And Objects

141. This Keyword In JS

142. Weird Behaviour Of The This Keyword

143. Constructor Functions

144. Javascript’s Own Constructor Function

145. Understanding Prototypes

146. Prototypical Inheritance Theory

147. Inheriting The User Properties

148. Inheriting The User Prototypes

149. Alternate Methods Of Creating Objects

150. Introduction To Property Descriptors

151. defineProperty Method

152. How Classes Are Syntactic Sugar On Prototypes

Section 12: Decorators In TypeScript

153. Introduction to Decorators

154. Writing Our First Decorator

155. When Are Decorators Invoked

156. Introduction To Decorator Factories

157. Changes To Decorators In TypeScript 5

1

158. Adding To Prototype

159. Interfaces For Prototypes

160. Adding Functions to Prototypes

161. Using Same Decorator With Multiple Classes

162. Method Decorators

163. Static Method Decorators

164. Decorators For Method Parameters

165. Decorators For Class Properties And Accessors

166. Multiple Decorators And Returning Values from Class Decorators

1

167. Decorator Composition and Evaluation

Section 13: Namespaces And Modules

0 / 5 | 30 of 5 lectures completed3

168. Introduction To Namespaces

169. ing With Namespaces

170. Namespaces In Multiple Files

171. Working With ESM Modules

172. Mixed Module Interoperability

Section 14: Declaration Files

0 / 8 | 50 of 8 lectures completed5

173. Transitioning JavaScript to TypeScript

174. First Declaration File

175. Reusable Types With Interfaces

176. Function Overloads With Declaration Files

177. Using Namespaces In Declaration Files

178. Declaration For Classes

1

179. Ambient Declarations

1

180. Using @Types Packages

Section 15: Advanced Concepts And Features

0 / 20 | 1hr 40min0 of 20 lectures completed1hr 40min

181. Module Introduction - Advanced Concepts and Features

182. Understanding Subtypes And Supertypes

183. How TypeScript Checks Compatibility

184. Type Widening

185. Typecasting

186. Totality

187. Discriminated Unions

188. keying-in or Index Accessed Types

189. keyof operator

190. typeOf Operator

191. Mapped Types

192. Using Mapped Types

193. Conditional Types

194. ★ Practice: Conditional Types

195. Solution: Conditional Types

196. Constraints on conditional types

197. Inferring With Conditional Types

198. Infer The Return Type Of A Function

199. Infer Function Arguments

200. Satisfies Operator (TS Version 5 feature)

Section 16: Type Guards In TypeScript

201. Introduction To Type Guards

202. typeof Type Guards

203. Truthiness Narrowing

204. Equality Narrowing

205. in Operator narrowing

206. instanceOf Type Guards

Section 17: Built In Utility Types

0 / 10 | 30 of 10 lectures completed3

207. Intro To Built In Utility Types

208. How Do Utility Types Work Behind The Scenes

209. Awaited<Type>

210. Record<Keys, Type>

211. Pick<Type, Keys>

212. Omit<OldType, Keys>

213. Partial<Type>

214. Required<Type>

215. Readonly<Type>

216. String Manipulation Types