**📘 TypeScript Notes**

**🔹 1. What is TypeScript?**

* **TypeScript** is a programming language.
* It is built on top of **JavaScript**.
* It adds **types** to JavaScript.
* TypeScript helps catch errors **before** running the code.

💡 TypeScript files have .ts extension.

**🔹 2. Why We Use TypeScript?**

**✅ Features of TypeScript:**

1. **Static Typing**  
   You can define the type of a variable (like number, string, etc.).
2. **Type Inference :** implicit type defining
3. **Error Checking**  
   Finds mistakes in code before running it.
4. **Code Suggestions**  
   Better auto-complete and hints in VS Code or other editors.
5. **Modern JavaScript Support**  
   Allows using new JavaScript features safely.
6. **Object-Oriented Features**  
   Like classes, interfaces, inheritance.

**⭐ Benefits of TypeScript:**

* Strongly static typed
* Reduces bugs
* Makes code easier to read and manage
* Useful for big projects and teamwork
* Better developer tools and debugging
* Works with existing JavaScript code

**🔹 3. History of TypeScript**

| **Year** | **Event** |
| --- | --- |
| 2012 | TypeScript was released by **Microsoft** |
| Creator | **Anders Hejlsberg** (also created C#) |
| Current | TypeScript is widely used with Angular, React, and Node.js |

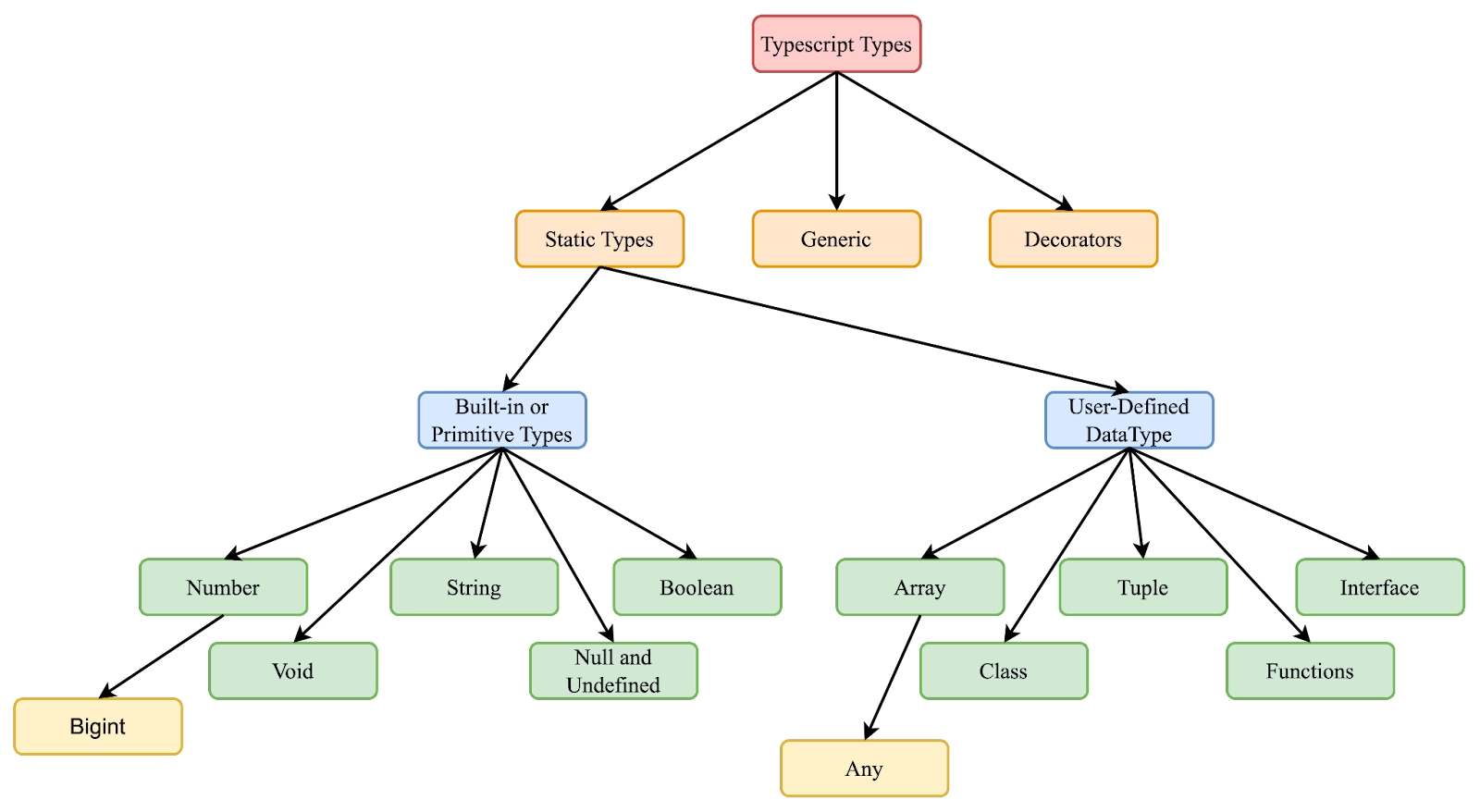
**🔹 4. Components of TypeScript**

1. **Language**  
   The TypeScript syntax and features (types, interfaces, etc.)
2. **Compiler (tsc)**  
   Converts .ts files into .js files so browsers can understand them.
3. **Type Definitions (.d.ts files)**  
   Files that describe the types of JavaScript libraries.

**🔹 5. JavaScript vs TypeScript**

| **Feature** | **JavaScript** | **TypeScript** |
| --- | --- | --- |
| Type Checking | No (dynamic typing) | Yes (static typing) |
| File Extension | .js | .ts |
| Compilation | Not needed | Compiled to JavaScript |
| Error Detection | At runtime | At compile time |
| Learning Curve | Easy to start | Slightly harder (types, rules) |
| IDE Support | Basic | Advanced (auto-suggestions, errors) |

## Typescript Types:



**Type Annotations vs Type Inference**

**🧾 1. Type Annotations**

**🔹 Definition:**

Type annotations are **explicit declarations** where you tell TypeScript the type of a variable, parameter, or return value.

**🧠 Syntax:** let variableName: type = value;

**📌 Examples of Type Annotations**

| **Type** | | **Example** |
| --- | --- | --- |
| **number** | let count: number = 10; | |
| **string** | let message: string = "Hello"; | |
| **boolean** | let isReady: boolean = true; | |
| **array** | let items: string[] = ["apple", "banana"]; | |
| **object** | let user: { name: string; age: number } = { name: "Alice", age: 30 }; | |
| **function** | function add(x: number, y: number): number { return x + y; } | |

**🧠 2. Type Inference**

**🔹 Definition:**

Type inference is when **TypeScript automatically determines** the type based on the assigned value or usage context, without needing an explicit type.

**📌 Examples of Type Inference**

| **Type** | **Code** | **Inferred Type** |
| --- | --- | --- |
| **number** | let count = 10; | count: number |
| **string** | let message = "Hello"; | message: string |
| **boolean** | let isReady = true; | isReady: boolean |
| **array** | let items = ["apple", "banana"]; | items: string[] |
| **object** | let user = { name: "Alice", age: 30 }; | user: { name: string; age: number } |
| **function** | Possible only when default parameter is there or usage context should be there(for ex.- map, filter – Ts will understand have to return array) | Based on default parameter and usage context |

function multiply(x = 2, y = 3) {

return x \* y;

}

// Inferred: x: number, y: number, return: number

**🆚 3. Annotations vs Inference: Summary Table**

| **Feature** | **Type Annotation** | **Type Inference** |
| --- | --- | --- |
| **Definition** | You **explicitly declare** the type | TypeScript **guesses** the type automatically |
| **Syntax** | let name: string = "Alice"; | let name = "Alice"; |
| **Control** | High (manual typing) | Medium (automatic) |
| **Ease** | Verbose but clear | Concise and faster |
| **Function Use** | Must annotate parameters | Return type can be inferred |

**🔹 1. Promise: Type Annotation vs Type Inference**

| **Feature** | **Type Annotation** | **Type Inference** |
| --- | --- | --- |
| Syntax | function foo(): Promise<Type> {} | function foo() {} |
| Example | function getData(): Promise<string> { return Promise.resolve("Hi"); } | function getData() { return Promise.resolve("Hi"); } |
| Control | High (you define the return type) | Low (TypeScript guesses the type) |
| Use Case | Exported functions, complex returns | Simple or quick functions |
| Benefit | More clarity, useful in large codebases | Less typing, faster to write |

**🔹 2. Passing Object in Function**

**✅ Required Properties**

function printUser(user: { name: string; age: number }) {

console.log(`${user.name} is ${user.age} years old`);

}

printUser({ name: "Alice", age: 25 });

* All properties in the object **must be provided**.

**🔹 3. Passing Object with Optional Properties**

**✅ Optional Property (using ?)**

function printUser(user: { name: string; age?: number }) {

console.log(`Name: ${user.name}`);

if (user.age !== undefined) {

console.log(`Age: ${user.age}`);

}

}

printUser({ name: "Bob" }); // age is optional

printUser({ name: "Alice", age: 30 }); // age is present

* age?: number → **optional**
* Always check if optional properties exist before using them.

**🔹 4. Optional Entire Object Parameter**

function greet(user?: { name?: string }) {

console.log(`Hello, ${user?.name ?? "Guest"}`);

}

greet(); // Hello, Guest

greet({ name: "Sam" }); // Hello, Sam

* user?: → entire object is optional.
* user?.name ?? "Guest" → safe way to handle undefined values.

## Data types:

|  |  |
| --- | --- |
| Built In / Primitive data types | User defined |
| * string * number * boolean * null * undefined * bigInt * symbol | * object * array * function * tuple |

**📘 TypeScript Notes: Objects, Type Inference, Annotations & Functions**

**🔹 1. Object Literals**

let obj = {};

* An empty object with no defined structure.
* Type is {} — can’t add properties directly unless specified.

**🔹 2. Object with Type Inference**

let userInference = {

ename: "Rinki",

age: 19,

salary: 50000,

city: 'delhi'

};

console.log(userInference);

**✅ TypeScript infers the type:**

{

ename: string;

age: number;

salary: number;

city: string;

}

* You **can update existing properties**.
* ❌ You **cannot add new properties** not already inferred.

**🔹 3. Object with Type Annotation**

let userAnnotation: {

ename: string;

age: number;

salary: number;

city: string;

isPaid: boolean;

} = {

ename: "Rinki",

age: 19,

salary: 50000,

city: 'delhi',

isPaid: true,

};

console.log(userAnnotation);

**✅ Explicitly defining structure using type annotation**

* Allows **clear structure**, optional properties, and stricter control.
* Adding or removing keys not listed will cause an error.

**🔹 4. Updating Values in Annotated Object**

let user: { firstName: string; lastName: string } = {

firstName: "sushant",

lastName: "Nayak"

};

user.firstName = "yogesh";

user.lastName = "kundra";

console.log(user);

✅ You can update existing values  
❌ You **cannot add new properties** that are not defined in the type

**🔹 5. Using Object in Arrow Function (Parameter & Return)**

**✅ Arrow function with object as parameter and return type as object**

let y = (user: { firstName: string; lastName: string }): object => {

return user;

};

let result = y({ firstName: "rohit", lastName: "soni" });

console.log(result);

* Works fine, but return type object is generic — doesn’t tell what’s inside.

**❌ Arrow function with {} as return type**

let y = (user: { firstName: string; lastName: string }): {} => {

return true; // ❌ Error: `true` is not an object

};

* {} means a non-null object.
* return true causes an error because it's a boolean, not an object.

**✅ Arrow function with specific object structure as return type**

let y = (user: { firstName: string; lastName: string }): { firstName: string; lastName: string } => {

return user;

};

let result = y({ firstName: "rohit", lastName: "soni" });

console.log(result);

✅ This is the correct way to ensure both **input and return types** are well-defined.

**🔴 What You Can’t Do**

| **❌ Not Allowed** | **Reason** |
| --- | --- |
| Add new property to inferred object | Not part of the inferred type |
| Return true when return type is {} | true is a boolean, not an object |
| Add property not defined in type | Type mismatch |
| Leave out required properties in function argument | Violates the type contract |

**🧠 Summary**

| **Concept** | **Works** | **Notes** |
| --- | --- | --- |
| Type Inference | ✅ | Auto-detects structure |
| Type Annotation | ✅ | Explicit, safer |
| Update values | ✅ | Must match type |
| Add new keys | ❌ | Only if declared |
| Function param (object) | ✅ | Structure must match |
| Arrow function return {} | ❌ | Too generic / error if mismatched |

**📘 Type Aliases in TypeScript**

**✅ What is a Type Alias?**

A **Type Alias** lets you **create a custom name** for a type. You can use this alias anywhere you would normally use a type.

**🧱 1. Type Alias with Object**

**🔹 Concept:**

You define an object structure once, give it a name using type, and then reuse that type multiple times — **especially useful for code reusability and readability**.

**🔸 Syntax:**

type objName = {

propertyName: type;

}

**🔸 Example:**

type USER = {

firstName: string;

lastName: string;

age: number;

isFemale: boolean;

};

let user: USER = {

firstName: "Somya",

lastName: "Yadav",

age: 23,

isFemale: true,

};

console.log(user);

**⭐ Important Points:**

* All properties **must be present** unless marked optional (?).
* Type alias helps reduce **repetition** in function parameters, classes, etc.
* You can create **nested types** using type alias too.

**🧮 2. Type Alias with Variables**

**🔹 Concept:**

Instead of writing primitive types repeatedly, define type aliases for them to improve **consistency and abstraction**.

**🔸 Syntax:**

type aliasName = actualType;

**🔸 Without Alias:**

function getProfile(userName: string, password: string, age: number, isMale: boolean) {

console.log(userName, password, age, isMale);

}

getProfile('somya123', 'password0000', 20, true);

**🔸 With Alias:**

type userName = string;

type password = string;

type age = number;

type isMale = boolean;

function getProfile(userName: userName, password: password, age: age, isMale: isMale) {

console.log(userName, password, age, isMale);

}

getProfile('somya123', 'password0000', 20, true);

**⭐ Important Points:**

* Useful in **large codebases** or where you want to give **semantic meaning** to primitive types.
* Increases **code readability**.

**🧠 3. Arrow Functions with Alias**

**🔹 Concept:**

Use type aliases to define a **function signature** and reuse it across different function declarations.

**🔸 Syntax:**

type FunctionType = (param1: type1, param2: type2) => returnType;

**🔸 Example:**

type cal = (x: number, y: number) => number;

let sum: cal = (x, y) => x + y;

let mul: cal = (x, y) => x \* y;

let sub: cal = (x, y) => x - y;

console.log(sum(10, 20));

console.log(mul(10, 20));

console.log(sub(10, 20));

**⭐ Important Points:**

* Ensures **type safety** for all functions using this alias.
* Encourages **DRY principle** (Don’t Repeat Yourself).
* Makes code easier to test and maintain.

**📝 Final Notes**

* Type aliases are a **core feature** of TypeScript.
* You can alias:
  + Objects
  + Primitives
  + Function signatures
  + Union and intersection types
* Use them to **write cleaner and scalable code**.

**✅ 1. Simple Arrays with Types**

let langs: string[] = ['js', 'python'];

let nums: number[] = [2, 5];

console.log(langs); // ['js', 'python']

console.log(nums); // [2, 5]

🟢 Arrays store multiple values of the **same type**.

**Syntax:** let arr: type[] = [val1, val2];

**🔁 2. Array Depth ([][][])**

let nestedArr: string[][][] = [[[]]];

console.log(nestedArr); // Output: [[[]]]

Each [] adds a **level of nesting**:

* string[]: one-dimensional
* string[][]: two-dimensional
* string[][][]: three-dimensional

**🧩 3. 3D Array Example (String)**

let mixArray: string[][][] = [

[["john"], ["f"]],

[["cena"], ["g"]],

[["alice"], ["x"]],

];

console.log(mixArray);

✅ This is a **multi-dimensional array of strings**  
Useful for representing:

* Grid structures
* Hierarchies
* Grouped data

**🏷️ 4. Tuples in TypeScript**

A **tuple** is an array with a **fixed number of elements**, where **each element has a specific type**.

**🔹 Example:**

let user: [string, number] = ['Amit', 25];

console.log(user); // ['Amit', 25]

**✅ 5. Tuple with Type Alias**

type Lang = [string, number];

let language: Lang = ['js', 85];

console.log(language); // ['js', 85]

🧠 Tuples are strict in **order and length**:

let wrong: Lang = [85, 'js']; // ❌ Error: Order matters

let tooMany: Lang = ['js', 85, 'extra']; // ❌ Error: Exceeds defined length

**✨ 6. Optional and Rest Elements in Tuples**

type OptionalTuple = [string, number?];

let val: OptionalTuple = ['Amit']; // ✅ Second value is optional

type RestTuple = [string, ...number[]];

let scores: RestTuple = ['Math', 85, 90, 92];

✅ Tuples support:

* Optional values
* Rest values (...)

**🛠️ 7. Function Returning an Array**

let addNum = (a: number, b: number): number[] => {

return [a, b];

};

console.log(addNum(10, 20)); // [10, 20]

✅ You can define:

* Parameter types
* Array return types
* Consistent structure with tuples

**🧠 Quick Recap Table**

| **Feature** | **Syntax** | **Notes** |
| --- | --- | --- |
| Typed array | let x: string[] = ['a'] | Homogeneous types |
| Nested array | let x: number[][][] = [[[1]]] | Use depth as per structure |
| Tuple | [string, number] | Fixed length, ordered types |
| Tuple with alias | type Lang = [string, number] | Clean and reusable |
| Optional tuple | [string, number?] | Last element can be skipped |
| Rest tuple | [string, ...number[]] | Combines fixed + flexible |
| Function returns array | (a: number, b: number): number[] | Return a typed array |

**🧪 Trainer Tips**

* 🔎 **Use tuples** when position and type matter (e.g., [name, age])
* 🔄 **Use arrays** for collections of similar items (string[], number[])
* 🛡️ TypeScript ensures safety by catching invalid lengths/types early
* 📚 Pair tuples with type for cleaner and reusable code structures

**📘 TypeScript: Union, Tagged Union, Narrowing, Intersection**

**🔶 1. Union Type (|)**

Union allows a variable to hold **more than one type** — but **only one at a time**.

**✅ Syntax:**

let value: string | number;

**✅ Example:**

let user: string | number | boolean;

user = "Sushant";

user = 100;

user = true;

**✅ Key Points:**

* Use | to declare multiple allowed types.
* Often used for **API responses**, **user input**, **optional configs**.
* Type **must be narrowed** before accessing type-specific methods.

**🔷 2. Tagged Union (Discriminated Union)**

A **type-safe way** to model different object shapes using a **common literal property** (tag) to distinguish them.

**✅ Syntax:**

type Success = { status: "success" };

type Error = { status: "error"; message: string };

type Pending = { status: "pending" };

type Response = Success | Error | Pending;

**✅ Usage:**

function handleResponse(res: Response) {

switch (res.status) {

case "success":

console.log("Success!");

break;

case "error":

console.log("Error: " + res.message);

break;

case "pending":

console.log("Loading...");

break;

}

}

**✅ Key Points:**

* Makes narrowing very easy and readable.
* Ideal for handling **state machines**, **API states**, **form states**, etc.
* Common status/type/state string helps discriminate.

**🔶 3. Type Narrowing**

TypeScript can't guess the exact type of a union — you must **narrow** it using runtime checks.

**✅ Techniques:**

* typeof → for primitives (string, number, boolean)
* in → for object properties
* instanceof → for classes or custom types

**✅ Example:**

function display(value: string | number) {

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

console.log("Upper:", value.toUpperCase());

} else {

console.log("Fixed:", value.toFixed(2));

}

}

function logUser(user: { name: string } | { id: number }) {

if ("name" in user) {

console.log("User name:", user.name);

} else {

console.log("User ID:", user.id);

}

}

**✅ Key Points:**

* Narrowing ensures **type safety** at runtime.
* Reduces chances of type errors.
* Required when working with union types.

**🔷 4. Intersection Type (&)**

Combines multiple types into one. The resulting type must include **all properties** from all types.

**✅ Syntax:**

type A = { a: string };

type B = { b: number };

type Combined = A & B;

**✅ Example:**

let user: Combined = {

a: "Ravi",

b: 100

};

**✅ Key Points:**

* Use & when you want to **merge** type definitions.
* All properties must be satisfied.
* Great for **composing object models** or extending base types.

**🧾 TypeScript enum Notes**

**🔹 What is an enum?**

* enum stands for **enumeration**.
* It is a **special class-like structure** used to represent a **set of named constant values**.
* Helps make code **readable** and **easy to manage**.
* Once defined, the enum values are **immutable** (unchangeable).

**🔹 Why use Enums?**

* Improves **readability** and **intent** of the code.
* Helps prevent **magic numbers** or hardcoded strings.
* Used in **conditional logic**, like roles, states, directions, etc.

**🔹 Types of Enums**

1. **Numeric Enums** (default)
2. **String Enums**
3. **Heterogeneous Enums** (mix of string and number – not recommended)

**🔹 1. Numeric Enum (Default)**

By default, enum members are auto-incremented numbers starting from 0.

**✅ Syntax:**

enum ROLES {

user, // 0

admin, // 1

publisher // 2

}

**✅ Example 1:**

let Users = ROLES.admin;

console.log(Users); // Output: 1

**✅ Example 2:**

let userManagement = () => {

if (ROLES.admin == 1) {

return 'Admin dashboard';

} else {

return 'User dashboard';

}

}

let result = userManagement();

console.log(result); // Admin dashboard

**✅ Example 3 (With switch-case and enum parameter):**

let userManagement = (roles: ROLES) => {

switch (roles) {

case ROLES.admin:

return 'Admin dashboard';

case ROLES.user:

return 'User Dashboard';

case ROLES.publisher:

return "Publisher dashboard";

}

}

let result = userManagement(2);

console.log(result); // Publisher dashboard

**🔹 2. String Enums**

Each member must be initialized with a string literal.

ts

CopyEdit

enum DIRECTION {

UP = "UP",

DOWN = "DOWN",

LEFT = "LEFT",

RIGHT = "RIGHT"

}

let move = DIRECTION.LEFT;

console.log(move); // "LEFT"

**🔹 3. Heterogeneous Enums**

Not commonly used. Mix of numbers and strings.

ts

CopyEdit

enum Example {

YES = 1,

NO = "NO"

}

**📋 Key Points**

| **Feature** | **Description** |
| --- | --- |
| Default start value | 0 for numeric enums |
| Auto-increment | Yes (only for numeric enums) |
| Reverse Mapping | Works with numeric enums (ROLES[1] => "admin") |
| Use case | Roles, directions, permissions, state constants |
| Type-safe | Yes. Enums provide better IntelliSense and autocomplete in editors |

**🔹 TypeScript Special Types:**

**1. any**

* Use when you **don't care about the type** or are **not sure**.
* Disables all type-checking.
* Can be **anything**: number, string, object, etc.
* Avoid overuse – it removes TypeScript’s safety.

let value: any = "Hello";

value = 42; // okay

value = true; // okay

**2. unknown**

* Safer version of any.
* You can assign **any value**, but **can’t use it directly** without narrowing or type checking.
* Forces you to **check the type** before using.

let value: unknown = "Hello";

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

console.log(value.toUpperCase()); // safe

}

// value.toUpperCase(); ❌ Error: Object is of type 'unknown'

**3. void**

* Mostly used in **functions** that do **not return a value**.
* Tells TypeScript: “This function returns nothing”.

function greet(name: string): void {

console.log("Hello", name);

// return "hi"; ❌ Error

}

**4. never**

* Represents **a value that never occurs**.
* Used when a function **never finishes** or **always throws** an error.
* Also used in **exhaustive checks** with union types.

function throwError(msg: string): never {

throw new Error(msg); // never returns

}

function infiniteLoop(): never {

while (true) {}

}

**🔸 Difference Table**

| **Feature** | **Any** | **unknown** | **void** | **never** |
| --- | --- | --- | --- | --- |
| Assignable to | Anything | Anything | undefined, null | Nothing |
| Type checking | Not required | Required before usage | Not applicable | Not applicable |
| Use case | When unsure of type | When type is unknown but safe | Function returns nothing | Function that never returns |
| Can be assigned | Anything | Anything | undefined or no return | Not assignable |
| Example usage | let x: any = 5; | let x: unknown = 5; | function x(): void {} | function x(): never { throw ...} |
| Compile safety | ❌ Unsafe (disables checks) | ✅ Safe (requires type check) | ✅ Safe | ✅ Safe |

**📘 OOPs in JavaScript (Object-Oriented Programming)**

**🔷 What is OOP?**

OOP (Object-Oriented Programming) is a **programming paradigm** that is based on the concept of **objects** and **classes**.

**🎯 Benefits of OOP**

* Promotes **code reusability**
* Encourages **modular structure**
* Makes code more **scalable and maintainable**
* Allows for **real-world modeling** of data

**🏗️ Classes in JavaScript**

* A **class** is a **blueprint** for creating objects.
* Contains properties (**fields**) and functions (**methods**).
* Introduced in **ES6**.
* Allows creation of **reusable components**.

**🔤 Class Syntax**

class ClassName {

// fields

// constructor

// methods

}

**✅ Example:**

class Point {

x: string;

y: string;

}

let result = new Point(); // Object created using class

console.log(result); // Output: Point { x: undefined, y: undefined }

**⚙️ Constructor Function (Old Way Before Classes)**

* A constructor function is a **function used to create objects** manually (before class keyword was introduced).

function Points() {

let x;

let y;

}

let results = new Points();

console.log(results); // Points {}

🧠 No this binding in the above example, so properties are not attached to the object.

**✅ Key Terms in a Class**

| **Term** | **Description** |
| --- | --- |
| fields | Variables declared in a class (e.g., x, y) |
| methods | Functions defined in a class |
| constructor | Special method to initialize object properties |
| nested class | A class defined inside another class (not common in JS) |
| interface | Describes structure (more TypeScript-specific than JavaScript) |

**🔸 1. Class with Basic Properties and Methods**

class Person {

userName: string;

age: number;

role: string;

addProperties() {

return `${this.userName} ${this.age} ${this.role}`;

}

}

**🧷 Key Points:**

* A **class** is a blueprint for objects.
* Use this to refer to current instance properties.
* addProperties() is an **instance method**.

**🔸 2. Class Method with Parameters to Set Properties**

class Person {

userName: string;

age: number;

role: string;

addProperties(userName: string, age: number, role: string) {

this.userName = userName;

this.age = age;

this.role = role;

return `${this.userName} ${this.age} ${this.role}`;

}

}

**🧷 Key Points:**

* Methods can take **parameters** to set object properties.
* Same object (person1) is being **mutated multiple times**, so latest values overwrite previous ones.
* ❗ To avoid this, create **multiple instances**.

**🔸 3. Using Constructor to Initialize Objects**

class Person {

constructor(

public userName: string,

public age: number,

public role: string

) {}

seeProperty() {

return `${this.userName} ${this.age} ${this.role}`;

}

}

**🧷 Key Points:**

* The constructor() automatically runs when an object is created.
* Use public in constructor to **auto-declare** and **assign** properties.
* Promotes clean & scalable code.

**🔸 4. Inheritance (Reusability)**

class Person {

constructor(public firstName: string, public lastName: string) {}

get\_full\_name() {

console.log(`${this.firstName} ${this.lastName}`);

}

}

class Employee extends Person {

constructor(

public age: number,

public gender: string,

firstName: string,

lastName: string

) {

super(firstName, lastName); // calls parent constructor

}

}

**🧷 Key Points:**

* Use extends keyword to create a **child class**.
* super() is used to call the parent constructor.
* Child class inherits **all methods** and **properties** of parent.

**🔸 5. Method Overriding (Polymorphism)**

class RBIBANK {

rateOfInterest: number = 7;

roi(): number {

return this.rateOfInterest;

}

}

class SBI extends RBIBANK {

roi(): number {

return this.rateOfInterest + 2.5;

}

}

**🧷 Key Points:**

* Child class can **override** parent methods.
* This is an example of **runtime polymorphism**.
* Enables **dynamic behavior** depending on the object.

**🏁 Summary Table**

| **Concept** | **Keyword/Technique** | **Example** |
| --- | --- | --- |
| Class | class | class Person {} |
| Object Creation | new | let p = new Person() |
| Constructor | constructor() | constructor(name) |
| Method | Function inside class | seeProperty() |
| Inheritance | extends, super() | class A extends B |
| Method Overriding | Same method in child | roi() in SBI |
| Encapsulation | Access control | private, public (default is public) |
| Polymorphism | Same method, different behavior | roi() |

**🔐 Access Modifiers in TypeScript**

Access Modifiers help **restrict access** to the properties and methods of a class.

**📌 Types of Access Modifiers**

| **Modifier** | **Accessible Inside Class** | **Derived Class** | **Outside Class** |
| --- | --- | --- | --- |
| public | ✅ | ✅ | ✅ |
| private | ✅ | ❌ | ❌ |
| protected | ✅ | ✅ | ❌ |
| readonly | ✅ (only in constructor) | ✅ | ✅ (read-only) |

**✅ public (Default)**

class Person {

public username: string = 'shashi';

public getUsername() {

console.log(this.username);

}

}

let user1 = new Person();

user1.username = 'manu'; // ✅ Can access and modify

user1.getUsername(); // ✅ Can access method

📍**Key Point:** Public members are accessible **anywhere**.

**✅ private (Fully Restricted)**

class User {

private username: string;

private age: number;

constructor(username: string, age: number) {

this.username = username;

this.age = age;

}

updateUsername() {

return this.username = 'shashi'; // ✅ allowed inside class

}

}

📍**Key Point:** private members **cannot** be accessed even in derived classes.

**✅ protected (Partial Restriction)**

class User {

protected username: string;

constructor(username: string, age: number) {

this.username = username;

}

}

class Employee extends User {

updateUsername(): string {

return this.username = 'shalni'; // ✅ allowed

}

}

📍**Key Point:** protected is useful for allowing inheritance but preventing outside access.

**✅ readonly (Immutable after initialization)**

class Employee {

readonly city: string;

constructor(city: string) {

this.city = city;

}

}

let emp = new Employee("Delhi");

console.log(emp.city); // ✅ can read

// emp.city = "Mumbai"; // ❌ Error: cannot assign

📍**Key Point:** readonly can only be **assigned once** (typically inside the constructor).

**🏗️ Multilevel Inheritance with Access Modifiers**

class QspEmployee extends Employee {

private edu: string;

private exp: string[];

constructor(username: string, age: number, salary: number, city: string, edu: string, exp: string[]) {

super(username, age, salary, city);

this.edu = edu;

this.exp = exp;

}

}

📍**Key Point:** Access control rules still apply even in multilevel hierarchy.

**⚙️ Getter & Setter in TypeScript**

A cleaner way to **access** and **update** private members.

**🎯 Syntax**

class Person {

private username: string;

get GetUsername() {

return `My name is ${this.username}`;

}

set updateUsername(name: string) {

this.username = name;

}

}

**🔍 Usage**

let person1 = new Person();

person1.updateUsername = 'Ronny'; // SETTER called

console.log(person1.GetUsername); // GETTER called

📍**Key Points:**

* get must **return** a value.
* set must accept **one parameter**.
* You can use getters and setters like **properties**, not methods.

**🧠 Summary Table**

| **Feature** | **Syntax** | **Use Case** |
| --- | --- | --- |
| Public | public name: string | Full access |
| Private | private name: string | Hide from outside & subclasses |
| protected | protected name: string | Visible in class & subclass |
| Readonly | readonly name: string | Assign once (constructor only) |
| Getter | get prop() | To retrieve private values safely |
| Setter | set prop(val) | To update private values safely |

**📘 Interface in TypeScript**

**✅ Definition:**

An **interface** defines the structure of an object. It enforces **contracts** for classes or objects to follow. Interfaces help in **type-checking** and **code organization**, especially in object-oriented programming.

**✅ Syntax:**

interface IDeveloper {

username: string;

skill: string;

}

**✅ Example Explained:**

interface IDeveloper {

username: string;

skill: string;

}

interface frontEndDev extends IDeveloper {

num\_of\_exp: number;

salary: number;

}

class FullStackDev implements frontEndDev {

username: string;

skill: string;

num\_of\_exp: number;

salary: number;

constructor(username: string, skill: string, num\_of\_exp: number, salary: number) {

this.username = username;

this.skill = skill;

this.num\_of\_exp = num\_of\_exp;

this.salary = salary;

}

}

let developer = new FullStackDev('monika', 'java', 5, 80000);

console.log(developer);

**🔍 Output:**

FullStackDev {

username: 'monika',

skill: 'java',

num\_of\_exp: 5,

salary: 80000

}

**🔑 Key Points about Interface:**

* Interfaces only define the **shape** of data.
* You can **extend** other interfaces using extends.
* Interfaces do **not generate JavaScript** code; they exist only at compile time.
* implements keyword is used by a class to follow the structure of an interface.

**📘 Abstraction in TypeScript**

**✅ Definition:**

**Abstraction** is the concept of hiding internal implementation details and showing only essential features.  
In TypeScript, abstraction is achieved using **abstract classes**.

**✅ Syntax:**

abstract class Developer {

abstract develop(): void;

workHours(): void {

console.log("Usually works 8 hours a day");

}

}

**✅ Explanation:**

* **abstract keyword** makes a class or method abstract.
* Abstract classes **cannot be instantiated directly**.
* Child classes must **implement** all abstract methods.
* Abstract classes can have **concrete methods** (with implementation) as well.

**🧠 Custom Example:**

abstract class Animal {

abstract makeSound(): void;

move(): void {

console.log("Moving...");

}

}

class Dog extends Animal {

makeSound(): void {

console.log("Bark!");

}

}

const dog = new Dog();

dog.makeSound(); // Bark!

dog.move(); // Moving...

**🔑 Key Points about Abstraction:**

* Used to **enforce implementation** in derived classes.
* Allows for **shared logic** while still requiring specific behavior in subclasses.
* Helps in **designing robust architecture** in OOP.

**📝 Summary Table:**

| **Feature** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| Inheritance | Can extend multiple interfaces | Can only extend one abstract class |
| Implementation | implements keyword | extends keyword |
| Code Presence | Only in TypeScript (compile-time) | Translates to JS (runtime) |
| Members | Only declarations | Can have both declaration + definition |
| Instantiation | Not possible | Not possible |

**🧠 TypeScript Notes: Generics, Interfaces, Classes, and Decorators**

**✅ 1. What are Generics?**

**Generics** allow you to create **reusable components** (functions, interfaces, classes) that work with **any data type** while maintaining **type safety**.

**🧾 Basic Syntax:**

ts

CopyEdit

function functionName<Type>(arg: Type): Type {

return arg;

}

**📌 Example 1: Generic Function**

ts

CopyEdit

function demoWithGeneric<Type>(args: Type): Type {

return args;

}

let output1 = demoWithGeneric<string>('modi');

let output2 = demoWithGeneric<number>(2012);

let output3 = demoWithGeneric<boolean>(true);

console.log(output1); // modi

console.log(output2); // 2012

console.log(output3); // true

**📌 Example 2: Generic with Arrays**

ts

CopyEdit

function GetFirstName<T>(arr: T[]): T {

return arr[0];

}

let firstNumber = [2, 52, 98];

let firstWord = ['java', 'python'];

console.log(GetFirstName(firstNumber)); // 2

console.log(GetFirstName(firstWord)); // java

**✅ 2. Why Use Generics?**

| **🔹 Benefit** | **🔍 Description** |
| --- | --- |
| ✅ Type Safety | Catches errors at compile time |
| 🔁 Reusability | Works for multiple data types |
| 🛠 Maintainability | Clean, flexible, and scalable code |

**✅ 3. Generic with Interface**

ts

CopyEdit

interface Metadata {

gender: string;

height: 'tall' | 'short';

favNum: number;

}

interface Person<T> {

id: number;

name: string;

age: number;

metadata: T;

}

const person1: Person<Metadata> = {

id: 1,

name: 'Ravi',

age: 30,

metadata: { gender: 'male', height: 'tall', favNum: 7 },

};

**✅ 4. Generic Class**

ts

CopyEdit

class ExampleClass<T> {

private values: T[] = [];

setValue(value: T): void {

this.values.push(value);

}

getValues(): T[] {

return this.values;

}

}

let example = new ExampleClass<number>();

example.setValue(1500);

console.log(example.getValues()); // [1500]

const exampleWithString = new ExampleClass<string>();

exampleWithString.setValue('shashi');

console.log(exampleWithString.getValues()); // ['shashi']

**✅ 5. Multiple Generics**

ts

CopyEdit

function Example<F, L>(firstName: F, lastName: L): any {

return { firstName, lastName };

}

console.log(Example<string, string>('shashi', 'kunal'));

console.log(Example<number, number>(5, 6));

console.log(Example<boolean, boolean>(true, false));

⚠️ Note: + operator cannot be used on generic types unless constrained to number or string.

**✅ 6. Bonus: Decorator Example (Class Decorator)**

Decorators are special kinds of declarations that can be attached to classes and members.

ts

CopyEdit

function Logger(constructor: Function) {

console.log(`✅ ${constructor.name} class created.`);

}

@Logger

class Product {

name: string;

price: number;

constructor(name: string, price: number) {

this.name = name;

this.price = price;

}

}

const p1 = new Product('Pen', 20);

**Output:**

kotlin

CopyEdit

✅ Product class created.