Typescript Interview Question

Q.What is TypeScript and how does it compare to JavaScript?

It's a **strict syntactical superset of JavaScript**, meaning:

- Every valid JavaScript program is also a valid TypeScript program.
- TypeScript adds **optional static typing** and other features on top of JavaScript.

Key Differences: TypeScript vs JavaScript

Feature	JavaScript	TypeScript
Typing	Dynamically typed	Statically typed (with optional types)
Compilation	Interpreted by browsers	Compiled to JavaScript before running
Error Detection	Runtime (only when code executes)	Compile-time (before running)
Tooling Support	Good	Excellent (with rich IDE support like VS Code)
Learning Curve	Easier for beginners	Slightly steeper due to type annotations
Code Scalability	Less structured for large apps	Better suited for large-scale applications
Advanced Features	Limited	Interfaces, Enums, Generics, Access Modifiers

Advantages of TypeScript

- **Catch errors early** through compile-time checking.
- **Improved IDE experience** with autocompletion, refactoring, and navigation.
- **Better documentation** through type annotations.
- **More maintainable code** for large codebases.
- **Support for modern JavaScript features** and backward compatibility.

What are the key TypeScript features? (e.g. static typing, interfaces, generics)

TypeScript that make it powerful and developer-friendly — especially for building scalable, maintainable applications:

2. Static Typing

TypeScript allows you to define types (e.g., `string`, `number`, `boolean`, `any`) for variables, function parameters, and return values

```
function greet(name: string): string {
  return `Hello, ${name}`;
}
```

Benefits:

- Catch type-related bugs at compile time.
- Improves IDE support (autocomplete, hints, etc.).

2. Interfaces

Interfaces define the shape of an object. They're used to ensure consistency in object structure.

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

const user: User = { name: "Alice", age: 30 };
```

Benefits:

- Enforces structure in objects and classes.
- Supports code reuse and abstraction.

3. Generics

Generics allow you to write reusable components or functions that work with any data type.

```
function identity<T>(arg: T): T {
  return arg;
}
```

4. Type Inference

TypeScript can infer the type of a variable even if you don't explicitly type it.

```
let count = 10; // inferred as number
```

5. Enums

Enums provide named constants for easier readability and maintainability.

```
enum Direction {
  Up,
  Down,
  Left,
  Right,
}
```

6. Classes & Access Modifiers

TypeScript enhances JavaScript classes with access modifiers (`public`, `private`, `protected`) and better OOP support.

```
class Person {
  private name: string;

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

  greet() {
    return `Hi, I'm ${this.name}`;
  }
}
```

7. Union & Intersection Types

You can combine types using union (`|`) or intersection (`&`) operators.

```
function printId(id: number | string) {
  console.log("ID:", id);
}
type User = {
  id: number;
  name: string;
};
//Intersection
type Admin = User & { role: string };
```

Adds flexibility while maintaining type safety.

- `&` merges multiple types into one.
- The resulting type **must satisfy all types involved**.
- Useful for extending existing types with more properties without inheritance.

8. Type Aliases

Allows you to name complex types for easier reuse.

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

Type Alias मतलब आप TypeScript की किसी **मौजूदा type** (primitive, object, union, function, tuple इत्यादि) को एक **कस्टम नाम** दे देते हैं — जिससे कोड अधिक पठन-योग्य और reusable हो जाता है।

```
type नया_नाम = मौजूदा_टाइप;
type ID = number;
let userId: ID = 123;
type User = {
  name: string;
  age: number;
  isAdmin?: boolean;
};

let u: User = { name: "Rahul", age: 30 };
```

function overloads

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: any, b: any): any {
  return a + b;
}
```

are **overload signatures**. They tell TypeScript:

> *"There's a function called `add`. It has **two valid forms**:*
>
> - *One takes two `number`s and returns a `number`*
> - *One takes two `string`s and returns a `string`"*

This is the **implementation signature**, which **must handle all the declared overload cases**.

```
add(1, 2);         // ✅ returns number
add("hi", "bye");  // ✅ returns string
```

```
add(true, false);  // ❌ error: no overload for boolean
```

Why use TypeScript over JavaScript?

Using TypeScript over JavaScript is mainly about increasing code safety, maintainability, and developer productivity, especially in larger codebases or team environments.

1. Early Error Detection :-TypeScript catches type-related and logic errors at compile time, not during runtime like JavaScript.
2. Improved Code Readability and Self-Documentation :-With types explicitly defined, code is more understandable without having to infer what each variable or function is doing.

```
function calculateTotal(price: number, taxRate: number): number
```

3. Powerful Autocomplete and IntelliSense :-Editors like VS Code use TypeScript types to provide context-aware suggestions, navigation, and refactoring tools.
4. Scalability :-TypeScript's static typing, modular structure, interfaces, and classes make it ideal for large applications and teams.
5. Better Refactoring Support :-With static typing, tools can safely rename variables, extract methods, and restructure code without breaking it.
6. Modern JavaScript Features (with Backward Compatibility):-TypeScript supports the latest JavaScript features (e.g., optional chaining, async/await) and compiles them to older JavaScript for browser compatibility.
7. Optional Adoption :-You can incrementally adopt TypeScript into existing JavaScript projects. Files can mix `.js` and `.ts`, so you don't have to rewrite everything.
   Bottom Line:
   Use JavaScript if you're building something quick and simple.
   Use TypeScript if you want fewer bugs, better collaboration, and scalable, maintainable code.

How do you declare variables (`let`, `const`, `var`)?

```
let score: number = 100;
const isAdmin: boolean = true;
var greeting: string = "Hello";
```

Summary Table

Type	Description	Example
`number`	Any numeric value	`let x: number = 3.14;`
`string`	Sequence of characters	`let msg: string = "Hello";`
`boolean`	Logical true or false	`let flag: boolean = true;`
`null`	Explicitly no value	`let val: null = null;`
`undefined`	Variable declared but not assigned	`let y: undefined = undefined;`
`symbol`	Unique identifier	`let id = Symbol('key');`
`bigint`	Arbitrarily large integer	`let big = 123456789n;`

What is `any`, `unknown`, `never`, and `void`?

In TypeScript, the types `any`, `unknown`, `never`, and `void` are special utility types used for different situations where stricter typing either isn't needed, isn't possible, or should be avoided. Here's a clear breakdown

1. 🔄 `any`

- Disables type checking completely for a variable.
- You can assign *anything* to it, and call anything on it.

```
let value: any = 5;
value = "hello";
value.toUpperCase(); // ✅ No error, even if it might fail at runtime
```

Use it when:

- Migrating JS to TS gradually.
- You don't know the type (yet).
  ⚠️ Avoid overusing — it removes the benefits of TypeScript.

2. ❓ `unknown`
   - Like `any`, but safer.
   - You must check the type before using it.

```
let value: unknown = "hello";
// value.toUpperCase(); ❌ Error
if (typeof value === "string") {
  console.log(value.toUpperCase()); // ✅ Safe
}
```

Use it when:

- Receiving external input (e.g., API data).
- You want **type safety with uncertainty**.

3. 🚫 `never`
   Represents values that **never occur**:

- A function that always throws an error.
- A function that never returns (infinite loop).

```
 function fail(message: string): never {
  throw new Error(message);
}

function infiniteLoop(): never {
  while (true) {}
}
```

Use it when:

- Indicating functions that cannot return.
- Exhaustive checks (e.g., switch statements with enums).

4. 🕳️ `void`
   Used when a function **does not return a value**.

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

Use it when:

- A function performs a side effect (like logging or modifying data) but doesn't return anything.

> ***Tip**: The actual value of a `void` function is usually `undefined`.*

Summary Table

Type	Description	Use Case
`any`	Bypasses all type checking	Use sparingly; last resort
`unknown`	Like `any`, but requires type checking	Safer external or dynamic values

Type	Description	Use Case
`never`	Function never returns or code unreachable	Error functions, exhaustive checking
`void`	Function doesn't return anything	Logging, side-effect functions

How does type inference work?

Type inference is one of TypeScript's most powerful features. It means that TypeScript can automatically determine a variable's type based on how it's declared and used — without you needing to explicitly annotate it.

1. Variable Declarations

```ts
let count = 10;          // inferred as number
let username = "Alice"; // inferred as string
let isOnline = true;    // inferred as boolean
```

```
   2. **Function Return Types**
      TypeScript infers the return type based on the function's `return`
value.
      ```ts
   function greet(name: string) {
  return `Hello, ${name}`; // inferred return type: string
}
```

You can still annotate explicitly if needed:

```ts
function greet(name: string): string {
  return `Hello, ${name}`;
}
```

4. Array Inference
   TypeScript infers the type of arrays based on the items in them.

```ts
let numbers = [1, 2, 3]; // inferred as number[]
let mixed = [1, "hello"]; // inferred as (number | string)[]
```

What are union and intersection types?
```

In TypeScript, union and intersection types are powerful ways to combine types, giving you flexibility and precision when defining variables, parameters, or return values.

1. Union Types (`|`)
   A union type means a value can be one of multiple types.

```
type Result = string | number;

function printId(id: number | string) {
console.log("ID:", id);
}
printId(123);       // ✅ OK
printId("ABC123");  // ✅ OK
```

2. Intersection Types (`&`)
   An intersection type combines multiple types into one. The resulting type has all the properties of the intersected types.

```
type Admin = { role: string };
type User = { name: string };
type AdminUser = Admin & User;
```

What is Type Narrowing?

> ***Type narrowing*** *is when TypeScript* ***refines a broader type*** *(like a union type) into a* ***more specific type*** *based on runtime checks.*

TypeScript uses your code logic (like `typeof`, `instanceof`, or custom checks) to "narrow" the type so you can safely access properties and methods.

What is a Type Guard?

> *A* ***type guard*** *is the* ***actual technique*** *or expression you use to* ***narrow a type****.*

It's how TypeScript knows *what* kind of value you're working with inside a block.

1. `typeof` Type Guard
   Use with primitive types: `string`, `number`, `boolean`, `symbol`, `undefined`, `bigint`

```
 function printId(id: number | string) {
if (typeof id === "string") {
  console.log(id.toUpperCase()); // ✅ id is narrowed to string
} else {
  console.log(id.toFixed(2));    // ✅ id is narrowed to number
```

```
  }
}
```

2. `instanceof` Type Guard

```
   class Dog {
  bark() {}
}

class Cat {
  meow() {}
}

function makeSound(animal: Dog | Cat) {
  if (animal instanceof Dog) {
    animal.bark(); // ✅ narrowed to Dog
  } else {
    animal.meow(); // ✅ narrowed to Cat
  }
}
```

3. `in` Operator
   Use when checking for specific property existence in objects.

```
   type Admin = { role: string };
type User = { username: string };

function greet(person: Admin | User) {
  if ("role" in person) {
    console.log(`Admin role: ${person.role}`); // ✅ Admin
  } else {
    console.log(`User: ${person.username}`);   // ✅ User
  }
}
```

What is an `enum`?

An `enum` (short for enumeration) is a custom type that consists of a set of named constants, either numeric or string-based.
It's useful when a variable can only have a fixed set of options.

1. Numeric Enums (default)

```
enum Direction {
  Up,      // 0
```

```
  Down,    // 1
  Left,    // 2
  Right,   // 3
}

let dir: Direction = Direction.Left;
console.log(dir); // 2
// Values auto-increment starting from `0`
```

2. You can also assign your own values:

```
   enum Status {
  Success = 1,
  Failure = 0,
  Pending = -1,
}
```

3. **String Enums**

```
   enum Size {
  Small = "S",
  Medium = "M",
  Large = "L",
}

let shirt: Size = Size.Medium;
console.log(shirt); // "M"
```

4. **Heterogeneous Enums** (mixing strings and numbers — ⚠️ avoid if possible)

```
enum Result {
  OK = 1,
  Error = "FAIL",
}
```

5. **Reverse Mapping (numeric enums only)**

```
   enum Role {
  Admin = 1,
  User = 2,
}
```

```
console.log(Role.Admin); // 1
console.log(Role[1]);    // "Admin" (reverse lookup)
```

⚠ ⚠️ **This does not work with string enums**

How do you define arrays, tuples, and objects?

1. **Arrays**

> *An array holds **multiple values of the same type**.*

```
let numbers: number[] = [1, 2, 3];
let names: string[] = ["Alice", "Bob"];

//Alternative syntax:
let numbers: Array<number> = [1, 2, 3];
```

Both `number[]` and `Array<number>` are valid — use whichever feels more readable.
2. **Tuples**
A **tuple** is a fixed-length array with **known types at each position**.

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

- Position 0 must be a `string`
- Position 1 must be a `number`

3. **Objects**
   Objects represent **structured data** with named properties.

```
   // Inline object type:
let user: { name: string; age: number } = {
  name: "Alice",
  age: 25
};
// With optional properties:
let user: { name: string; age?: number } = {
  name: "Bob"
};
//Using a `type` or `interface`:
type User = {
  name: string;
  age: number;
  isAdmin?: boolean;
```

```
};

let user1: User = {
  name: "Charlie",
  age: 40
};
```

Arrays of objects:

```
  type Product = {
  id: number;
  title: string;
};

let products: Product[] = [
  { id: 1, title: "Book" },
  { id: 2, title: "Laptop" }
];
```

How do optional and readonly properties work?

in TypeScript, optional and readonly properties are used to make object types more flexible and safer by explicitly controlling how properties behave.

1. Optional Properties (`?`)
   Makes a property optional — meaning it may or may not be present on an object.

```
   type User = {
  name: string;
  age?: number; // optional
};

const user1: User = { name: "Alice" };            // ✅ OK
const user2: User = { name: "Bob", age: 30 };     // ✅ OK
```

2. Readonly Properties (`readonly`)
   A readonly property cannot be reassigned after it's set — it's immutable from the outside.

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

```
const p: Point = { x: 10, y: 20 };
p.x = 15; // ❌ Error: Cannot assign to 'x' because it is a read-only
property
```

It’s great for protecting data structures like coordinates, settings, etc.

You Can Combine Both

```
type Settings = {
  readonly id: string;
  theme?: "dark" | "light";
};
const config: Settings = {
  id: "abc123"
};
config.id = "xyz"; // ❌ Error
```

How do functions get typed (parameters, returns)?

you can **explicitly type functions** to ensure both the **inputs (parameters)** and **outputs (return values)** are what you expect. This helps catch bugs early and improves code clarity.

1. **Typing Function Parameters**

```
   function greet(name: string, age: number) {
  console.log(`Hello, ${name}. You are ${age} years old.`);
}
```

- Each parameter must match the expected type.
- If a parameter is **optional**, use `?`:

```
function greet(name: string, age?: number) {
  console.log(`Hi, ${name}${age ? ", age " + age : ""}`);
}
```

2. **Typing the Return Value**

```
function add(a: number, b: number): number {
  return a + b;
}
```

if the function returns nothing (only side effects), use `void`:

```
function log(message: string): void {
  console.log(message);
}
//3. **Typing Function Expressions**
const multiply = (a: number, b: number): number => {
  return a * b;
};
```

What are generics and when should you use them?

> ***Generics*** *let you write code that works with* ***any type****, but still enforces* ***type safety****.*

Instead of hardcoding a type like `number` or `string`, you use a type parameter, often called `T`.

```
function identity<T>(value: T): T {
  return value;
}

identity<number>(42);      // returns 42
identity<string>("hello"); // returns "hello"
```

Why Use Generics?

Without Generics	With Generics
`function identity(value: any): any`	`function identity<T>(value: T): T`
× Loses type info	✅ Keeps type info
× Needs manual type checks	✅ Compiler enforces correct usage

Generic Arrays

```
function firstItem<T>(items: T[]): T {
  return items[0];
}
firstItem([1, 2, 3]);           // T = number
firstItem(["a", "b", "c"]);     // T = string
```

Generic Types

You can use generics in type aliases or interfaces

```ts
type Box<T> = {
  value: T;
};
const stringBox: Box<string> = { value: "Hello" };
const numberBox: Box<number> = { value: 123 };
```

Generic Functions with Constraints

```ts
function getLength<T extends { length: number }>(item: T): number {
  return item.length;
}
getLength("hello");         // ✅ string has length
getLength([1, 2, 3]);       // ✅ array has length
getLength(123);             // ❌ number has no length
```

When Should You Use Generics?

Use generics when:

- Your function/class/type works with **many types**
- You want to **preserve type information**
- You want **type safety without duplication**

Quick Summary

Concept	Syntax	Purpose
Basic generic	`function <T>(arg: T): T`	Works with any type
Generic array	`T[]` or `Array<T>`	Arrays of any type
Type constraint	`<T extends SomeType>`	Limit generic to certain shapes

`tsconfig.json`

> *The `tsconfig.json` file tells TypeScript **how to compile your code** and **what files to include**.*

```ts
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "strict": true,
    "outDir": "./dist",
    "esModuleInterop": true
  },
```

```
  "include": ["src"],
  "exclude": ["node_modules"]
}
```

Key `compilerOptions`:

Option	Purpose
`target`	Output JS version (`ES5`, `ES6`, `ES2020`)
`module`	Module system (`commonjs`, `ESNext`, etc.)
`strict`	Enables all strict type-checking features
`outDir`	Where to put compiled `.js` files
`esModuleInterop`	Makes `import` work better with CommonJS

What is declaration merging?

Declaration merging is a unique feature in TypeScript where multiple declarations with the same name are automatically merged into a single definition — combining their features.

It works with certain structures like interfaces, namespaces, functions, enums, and classes.

- Lets libraries (like `@types/node` or `express`) extend or modify types without touching the original code.
- Enables you to add custom properties or methods to existing types.

1. Interface Merging
   If two interfaces share the same name, TypeScript merges their properties.

```
interface User {
name: string;
}
interface User {
age: number;
}
const u: User = {
name: "Alice",
age: 30,
};
```

2. **Namespace + Function/Class/Enum Merging**
   You can merge a `namespace` with a `function`, `class`, or `enum`.
   ```ts
   function greet(name: string) {
  return `Hello, ${name}`;
}
namespace greet {
  export const version = "1.0";
}

console.log(greet("Alice"));       // Hello, Alice
console.log(greet.version);        // 1.0
//Class + Namespace

class Car {
  drive() {
    console.log("Driving");
  }
}

namespace Car {
  export const manufacturer = "Toyota";
}

console.log(Car.manufacturer); // "Toyota"
//Enum + Namespace
enum Color {
  Red,
  Green,
  Blue,
}

namespace Color {
  export function isPrimary(c: Color) {
    return c === Color.Red || c === Color.Blue;
  }
}
console.log(Color.isPrimary(Color.Green)); // false
   ```

Explain conditional types and the infer keyword

this dives into some of the **most powerful advanced features** of TypeScript's type system: **conditional types** and the `infer` keyword. These are essential for creating flexible, intelligent, type-safe APIs.

1. **Conditional Types**
   A **conditional type** chooses one type or another based on whether a condition is true

   ```
   T extends U ? X : Y
   ```

```
“If `T` is assignable to `U`, then use type `X`, otherwise use type `Y`.”

Simple Conditional Type
```ts
type IsString<T> = T extends string ? "Yes" : "No";

type A = IsString<string>;  // "Yes"
type B = IsString<number>;  // "No"
```

2. **The `infer` Keyword**

> *`infer` is used **inside a conditional type** to **extract** or **infer** a type from another type.*

```
type UnwrapPromise<T> = T extends Promise<infer U> ? U : T;
```

- If `T` is a `Promise<U>`, infer `U`
- Otherwise return `T` as-is

   ```
   type A = UnwrapPromise<Promise<number>>;  // number
   type B = UnwrapPromise<string>;           // string
   ```

```
## What are `keyof`, `typeof`, mapped types, and utility types?
1. `keyof`
   The `keyof` operator creates a **union of all property names** (keys) of a type.
   ```ts
   type Person = {
```

```
  name: string;
  age: number;
};
type PersonKeys = keyof Person; // "name" | "age"
```

You can use this for **key-safe functions**:

```
function getValue(obj: Person, key: keyof Person) {
  return obj[key];
}
```

2. `typeof`
   The `typeof` operator gives you the **type of a value or variable**.

```
  const person = {
  name: "Alice",
  age: 30
};

type PersonType = typeof person;
// PersonType = { name: string; age: number }
```

This is commonly used to **reference runtime values as types**.

4. **Utility Types**
   TypeScript comes with **built-in utility types** that use `keyof`, `typeof`, and mapped types under the hood.

Common Utility Types:

Utility	What it does
`Partial<T>`	All properties optional
`Required<T>`	All properties required
`Readonly<T>`	All properties readonly
`Pick<T, K>`	Pick only specified keys
`Omit<T, K>`	Exclude specified keys
`Record<K, T>`	Create an object type with keys `K` and value type `T`
`ReturnType<T>`	Get return type of function `T`
`Parameters<T>`	Get parameter types of function `T`

```ts
type Person = { name: string; age: number };

type PartialPerson = Partial<Person>;   // { name?: string; age?: number }
type NameOnly = Pick<Person, "name">;   // { name: string }
type WithoutAge = Omit<Person, "age">;  // { name: string }
```

What is the non-null assertion (`!`)?

The **non-null assertion operator** (`!`) in TypeScript is used to **tell the compiler**:

> *"I know this value is not `null` or `undefined` — trust me."*

It's a way to **override strict null checks** at a specific point in your code.

- *Syntax*

```ts
someValue! // asserts that `someValue` is NOT null or undefined
let name: string | undefined;
function greet() {
  console.log("Hello, " + name!.toUpperCase());
}
```

Describe access modifiers: public, private, protected

In TypeScript, **access modifiers** control the **visibility** of class members (properties or methods). They determine **where a member can be accessed from** — inside the class, subclasses, or externally.

1. **`public`** **(default)**

> *Accessible **anywhere** — inside the class, outside the class, or in subclasses.*

```ts
class Animal {
  public name: string;

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

  public speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

const dog = new Animal("Dog");
console.log(dog.name); // ✅ Accessible
```

```
dog.speak();           // ✅ Accessible
```

2. `private`
   Accessible **only within the class** itself.

```
class BankAccount {
  private balance: number;

  constructor(initial: number) {
    this.balance = initial;
  }

  public getBalance() {
    return this.balance;
  }
}

const account = new BankAccount(1000);
console.log(account.getBalance()); // ✅ OK
console.log(account.balance);      // ❌ Error: Property 'balance' is
private
```

3. `protected`

> *Accessible **inside the class** and in **subclasses**, but **not** from outside.*

```
class Person {
  protected ssn: string;

  constructor(ssn: string) {
    this.ssn = ssn;
  }
}

class Employee extends Person {
  revealSSN() {
    return this.ssn; // ✅ Accessible in subclass
  }
}

const emp = new Employee("123-45-6789");
emp.revealSSN(); // ✅
console.log(emp.ssn); // ❌ Error: Property 'ssn' is protected
```

Interfaces

- **Compile-time only**: They do *not* exist at runtime—no emitted JavaScript code
- Define the **shape of data**, listing property and method signatures (public only).
- Cannot contain actual method implementations or data members (though TS now allows limited default method bodies, it's better to assume no state)
- A class can **implement multiple interfaces**—promoting flexible composition

Use interfaces when:

- You want to define a contract for unrelated classes.
- You don't need shared implementation or state.
- You prefer loose coupling and type-only design.

Abstract Classes

- **Exist at runtime**—you can check `instanceof` them
- Can include:
  - **Abstract methods** (no implementation) and **concrete methods**
  - **State (properties)**, constructors, protected/private members—just like regular classes
    A class can **extend only one abstract class**, limiting inheritance

Use abstract classes when:

- You want to **share behavior and state** among a family of related classes.
- You need some **common logic implemented**, while still requiring subclasses to override specific parts.
- You want to enforce a specific inheritance structure (template pattern)

Tips from the Community

> ***Interfaces*** *should be preferred in many cases because a class can implement multiple interfaces, offering greater flexibility*
> ***Abstract classes*** *shine when you need shared logic and the inheritance structure makes sense, especially when you maintain state or provide default method implementations inside the base class.*

Advanced Types & Patterns

What Is `infer`?

- The `infer` keyword **only works within** a conditional type (`T extends U ? X : Y`)
- It lets you **capture a type** from within another type during the check.
- Think of it as a way to **"unpack"** part of a type into a fresh type variable you can re-use.
  Extract Array Element

```ts
type ArrayElementType<T> = T extends (infer E)[] ? E : T;

type A = ArrayElementType<number[]>; // number
type B = ArrayElementType<string>;   // string
```

Why use `type`?

1. Union and Intersection Types

`type` can do more than just describe object shapes — it can define unions, intersections, and primitives.

```ts
type Admin = { isAdmin: true };
type RegularUser = { isAdmin?: false };

type User = (Admin | RegularUser) & { name: string; age: number };
```

This kind of thing isn't possible with `interface`.

2. Aliasing Primitives and Complex Types

You can use `type` to create aliases for any type, not just object shapes:

```ts
type ID = string | number;
type Callback = () => void;
```

3. More Readable for Complex Combinations

If you're dealing with union/intersection-heavy logic, `type` is often cleaner:

```ts
type Status = "pending" | "success" | "error";
```

Why use `interface`?

Even though `type` is more flexible, `interface` has some advantages too:

1. Extending is Cleaner

You can extend interfaces in a very natural way:

```ts
interface Person {
  name: string;
```

```
}
interface User extends Person {
  age: number;
}
```

You can do similar things with `type` using intersections (`&`), but it's less intuitive:

```
type Person = { name: string };
type User = Person & { age: number };
```

2. Declaration Merging

Only `interface` allows you to merge declarations:

```
interface User {
  name: string;
}
interface User {
  age: number;
}
// User now has both `name` and `age`
```

example

```
type Admin = { isAdmin: true };
type RegularUser = { isAdmin?: false };

type User = (Admin | RegularUser) & {
  name: string;
  age: number;
};

function greet(user: User) {
  if (user.isAdmin) {
    console.log(`Welcome Admin ${user.name}`);
  } else {
    console.log(`Hello ${user.name}`);
  }
}

// Sample users
const adminUser: User = {
  name: "Alice",
  age: 35,
  isAdmin: true,
};
```

```
const regularUser: User = {
  name: "Bob",
  age: 28,
};

greet(adminUser);     // Output: Welcome Admin Alice
greet(regularUser);   // Output: Hello Bob
```

How do you install TypeScript ?

This allows you to use the `tsc` (TypeScript Compiler) command from anywhere.

```
npm install -g typescript

tsc --version
```

Install TypeScript Locally (in a project)

This keeps it as a project dependency (recommended for teams or shared projects).

```
npm init -y               # create package.json if not already
npm install --save-dev typescript
```

3. **Initialize TypeScript in your project**

```
   npx tsc --init
```

This will create a `tsconfig.json` file where you can configure TypeScript settings.
4. **Write and Compile TypeScript Code**

- Create a file: `index.ts`
- Write your TypeScript code.
- Compile it:

```
npx tsc
```

It will output `.js` files (by default, in the same directory).

What are the basic types in TypeScript?

1. **`string`**
   `let name: string = "Rahul";`
2. **`number`** `let age: number = 30;let price: number = 99.99;`
3. **`boolean`** `let isActive: boolean = true;`
4. **`null`** and **`undefined`** `let nothing: null = null;let notAssigned: undefined = undefined;`
5. **`any`** `let something: any = "hello";something = 42;`
6. **`unknown`** `let input: unknown = "text";`
7. **`void`**

```
function logMessage(): void {
console.log("Hello");
}
```

8. **`never`**
   For functions that never return (like errors or infinite loops).

```
function throwError(): never {
  throw new Error("Something went wrong");
}
```

9. **`object`**
   For non-primitive values (but not commonly used directly).

```
let user: object = { name: "Alice" };
```

10. **`array`**

```
let numbers: number[] = [1, 2, 3];
let names: string[] = ["Alice", "Bob"];
```

11. **`tuple`**

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

12. **`enum`**

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

13. **`enum`**

```
enum Direction {
  Up,
  Down,
  Left,
  Right
}
let dir: Direction = Direction.Up;
```

What is the difference between `any`, `unknown`, and `never`?

`any`, `unknown`, and `never` are all special types in TypeScript, but they serve very different purposes.

1. `any`
   Represents any value, and disables type checking.

```
let value: any = "hello";
value = 42;
value = true;
```

Behavior:

- TypeScript won't check what you do with `any`.
- Use with caution — you lose all TypeScript safety.

2. `unknown`
   Represents any value, like `any`, but safe.

```
let value: unknown = "hello";
value = 42;
value = true;
// Error ❌: Property 'toUpperCase' does not exist on type 'unknown'
// value.toUpperCase();

// ✅ Correct way:
if (typeof value === "string") {
  console.log(value.toUpperCase());
}
```

Behavior:

- TypeScript forces you to check the type before using it.
- Safer alternative to `any`.

3. `never`
   Represents values that never occur.

Used when:

- A function never returns (e.g., throws an error)
- A function runs forever
- You've handled all possible types in a `switch` (exhaustiveness check)

```
function throwError(): never {
  throw new Error("Something went wrong");
}
function infiniteLoop(): never {
  while (true) {}
}
```

What is type inference in TypeScript?

Type inference means that TypeScript automatically figures out the type of a variable, expression, or return value without you explicitly writing it.

1. Variable Initialization

```
let name = "Rahul"; // inferred as string
```

```
Here, TypeScript **infers** that `name` is a `string` because it's
initialized with a string value.
If you later try:
```ts
name = 123; // ❌ Error: Type 'number' is not assignable to type 'string'
```

2. Function Return Type

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

You didn't write the return type, but TypeScript knows it's `number`.
3. Arrays

```
let fruits = ["apple", "banana"]; // inferred as string[]
```

4. Contextual Typing
```

```
window.addEventListener("click", (event) => {
console.log(event.clientX); // event is inferred as MouseEvent
});
```

What are union and intersection types?

1. **Union Types (|)**
   A **union type** allows a value to be **one of several types**.

```
let value: string | number;
value = "hello"; // ✅ OK
value = 123;     // ✅ OK
value = true;    // ❌ Error
```

Function accepting multiple types

```
function printId(id: string | number) {
 console.log("ID:", id);
}

printId("abc123");
printId(101);
```

2. **Intersection Types (&)**
   An **intersection type** combines **multiple types into one** — the value must satisfy **all** of them.

```
   type A = { name: string };
type B = { age: number };

type Person = A & B;

const user: Person = {
  name: "Rahul",
  age: 30,
};
```

How do you define a function with typed parameters and return types?

```
function functionName(param1: Type1, param2: Type2): ReturnType {
  // function body
```

```
}
//Add two numbers
function add(a: number, b: number): number {
  return a + b;
}
function greet(name: string): string {
  return `Hello, ${name}!`;
}
//Function with no return (`void`)
function logMessage(message: string): void {
  console.log(message);
}
//Optional parameters
function greet(name: string, age?: number): string {
  return age ? `Hello ${name}, age ${age}` : `Hello ${name}`;
}
// Default parameters
function multiply(a: number, b: number = 2): number {
  return a * b;
}
 //Arrow function with types
 const divide = (x: number, y: number): number => {
  return x / y;
};
```

What is an interface in TypeScript?

An **interface** in TypeScript defines the **shape of an object** — meaning, what properties (and types) it must have. It's like a **contract** that objects must follow.

```
interface Person {
  name: string;
  age: number;
}
const user: Person = {
  name: "Rahul",
  age: 25,
};
//**Optional properties** (`?`)
interface Person {
  name: string;
  age?: number;
}
//Read-only properties
interface Person {
  readonly id: number;
  name: string;
```

```
}
//Function types
interface Greet {
  (name: string): string;
}

const greet: Greet = (name) => `Hello, ${name}`;
//Extending interfaces
interface Person {
  name: string;
}
interface Employee extends Person {
  employeeId: number;
}
//>>//
interface Product {
  id: number;
  name: string;
  price: number;
  inStock: boolean;
}

function showProduct(p: Product): void {
  console.log(`${p.name} costs ₹${p.price}`);
}
```

How do you create an array of a specific type?

you can create an **array of a specific type** using one of two main syntaxes.

1. **Type[] Syntax (Most common)**

```
let numbers: number[] = [1, 2, 3];
let names: string[] = ["Rahul", "Alice", "Bob"];
```

```
2. **Generic Array Syntax**
   ```ts
   let numbers: Array<number> = [1, 2, 3];
let names: Array<string> = ["Rahul", "Alice"];
```

3. **Array of Custom Types**

```
interface User {
name: string;
```
```

```ts
age: number;
}
let users: User[] = [
{ name: "Rahul", age: 30 },
{ name: "Alice", age: 25 },
];
```

````
4. **Array with Multiple Types (Union)**
   ```ts
   let mixed: (string | number)[] = ["Alice", 42, "Bob", 100];
````

5. Array of Tuples

```ts
let entries: [string, number][] = [
["Alice", 25],
["Bob", 30],
];
```

````
## How do you type an object in TypeScript?
you type an **object** by describing its **shape** — the properties it has and their types. You can do this in a few different ways depending on your use case.
1. **Inline Object Type**
   ```ts
   let person: { name: string; age: number } = {
  name: "Rahul",
  age: 30,
};
````

2. Using an Interface

```ts
interface Person {
  name: string;
  age: number;
}

let user: Person = {
  name: "Alice",
  age: 25,
};
```

3. **Using a Type Alias**

```
type Person = {
name: string;
age: number;
};
```

let employee: Person = {
name: "John",
age: 40,
};
type Product = {
name: string;
price: number;
manufacturer: {
name: string;
country: string;
};
};

let item: Product = {
name: "Phone",
price: 799,
manufacturer: {
name: "TechCorp",
country: "India",
},
};

```
## ### What are **Optional Properties** in TypeScript Interfaces?

**Optional properties** are properties that **may or may not be present**
in an object that implements an interface. You declare them by adding a
**`?`** after the property name.
```ts
interface Person {
  name: string;
  age?: number; // optional
}
```

What are Enums in TypeScript?
```

Enums (short for *enumerations*) are a way to define a set of **named constants**. They help you give more meaningful names to a group of related values.

```
enum Status {
  Success = 200,
  NotFound = 404,
  ServerError = 500,
}
//String Enums
enum Color {
  Red = "RED",
  Green = "GREEN",
  Blue = "BLUE",
}
```

How do you handle default parameter values in TypeScript?

```
function greet(name: string = "Guest") {
  console.log(`Hello, ${name}!`);
}
//
function greet(name: string = "Guest") {
  console.log(`Hello, ${name}!`);
}

greet();          // Output: Hello, Guest!
greet("Rahul");   // Output: Hello, Rahul!
```

What are Literal Types in TypeScript?

Literal types allow you to specify that a variable or parameter can only have **exactly one specific value** (or a set of specific values). These can be **string literals**, **number literals**, or **boolean literals**.

```
let direction: "up" | "down" | "left" | "right";

direction = "up";    // ✅ OK
direction = "down";  // ✅ OK
direction = "forward"; // ❌ Error: not assignable
//
type Status = "success" | "error" | "loading";
function setStatus(status: Status) {
```

```
  console.log("Status:", status);
}

setStatus("success"); // ✅ OK
setStatus("fail");    // ❌ Error

//
type DiceRoll = 1 | 2 | 3 | 4 | 5 | 6;
let roll: DiceRoll;
roll = 3; // ✅ OK
roll = 7; // ❌ Error
```

React + TypeScript

1. Typing Props in a Functional Component
   The most common pattern is to define a **props interface (or type alias)** and then use it as the type of the props argument.

```
import React from "react";

interface MyComponentProps {
  title: string;
  count?: number; // optional prop
}
const MyComponent = ({ title, count }: MyComponentProps) => {
  return (
    <div>
      <h1>{title}</h1>
      <p>Count: {count ?? 0}</p>
    </div>
  );
};
```

2. Typing State in Functional Components

```
const [count, setCount] = useState<number>(0);
//With an interface/type for more complex state
type User = {
name: string;
age: number;
};
const [user, setUser] = useState<User>({ name: "Alice", age: 30 });
//Union type
const [status, setStatus] = useState<"idle" | "loading" | "success" |
"error">("idle");
```

Function Component Inline Handler

```ts
<button onClick={(e: React.MouseEvent<HTMLButtonElement>) =>
handleClick(e)}>
  Click me
</button>
//
const handleClick = (e: React.MouseEvent<HTMLButtonElement>) => {
  console.log("Clicked!", e.currentTarget);
};
//Typing `onChange` for `<input>`
const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  console.log("Value:", e.target.value);
};

<input onChange={(e: React.ChangeEvent<HTMLInputElement>) =>
handleChange(e)} />

//4. Form Submit Handler
const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {
  e.preventDefault();
  console.log("Form submitted");
};
```

General Event Type Reference

Event Type	React Type
`onClick`	`React.MouseEvent<HTMLElement>`
`onChange`	`React.ChangeEvent<HTMLInputElement>`
`onSubmit`	`React.FormEvent<HTMLFormElement>`
`onKeyDown`, etc.	`React.KeyboardEvent<HTMLElement>`
`onFocus`, `onBlur`	`React.FocusEvent<HTMLElement>`

Typing Refs to DOM Elements

Using `useRef` (Functional Components)

```
import { useRef, useEffect } from "react";
const MyComponent = () => {
  const inputRef = useRef<HTMLInputElement>(null);

  useEffect(() => {
    inputRef.current?.focus(); // Safe: inputRef.current may be null
  }, []);
  return <input ref={inputRef} />;
};
```

Typing `useState`

1. **Primitive Values**

```
   const [count, setCount] = useState<number>(0);
```

2. **Union Types**

```
const [status, setStatus] = useState<"idle" | "loading" | "error">
("idle");
```

3. Object or Custom Type

```
type User = {
  name: string;
  age: number;
};
const [user, setUser] = useState<User | null>(null);
```

Typing `useReducer`

1. Define State and Action Types

```
type CounterState = {
count: number;
};
```

type CounterAction =
| { type: "increment" }
| { type: "decrement" }
| { type: "reset" };
const reducer = (state: CounterState, action: CounterAction): CounterState ⇒ {
switch (action.type) {

```
case "increment":
return { count: state.count + 1 };
case "decrement":
return { count: state.count - 1 };
case "reset":
return { count: 0 };
default:
return state;
}
};
```

```
###### How do you type `useContext`?
Typing `useContext` in TypeScript involves **correctly typing the context
object** when you create it and then safely consuming it using
`useContext`.
######  **Define the Context Type**
```ts
type AuthContextType = {
  user: string | null;
  login: (username: string) => void;
  logout: () => void;
};
```

Create and Type the Context

```
import React, { createContext, useContext, useState, ReactNode } from
"react";
const AuthContext = createContext<AuthContextType | undefined>(undefined);
```

- `AuthContextType | undefined` allows checking whether the context is used outside a provider.
- You can also initialize with a default value, but be careful to keep it consistent.

Create the Provider Component

```
const AuthProvider = ({ children }: { children: ReactNode }) => {
  const [user, setUser] = useState<string | null>(null);
  const login = (username: string) => setUser(username);
  const logout = () => setUser(null);
  const value: AuthContextType = { user, login, logout };
  return <AuthContext.Provider value={value}>{children}
</AuthContext.Provider>;
};
```
```

Consume the Context with useContext

```
const useAuth = () => {
  const context = useContext(AuthContext);
  if (!context) {
    throw new Error("useAuth must be used within an AuthProvider");
  }
  return context;
};
```

Usage Example

```
const Dashboard = () => {
  const { user, logout } = useAuth();

  return (
    <div>
      <p>Welcome, {user}</p>
      <button onClick={logout}>Log out</button>
    </div>
  );
};
```

What is a `tsconfig.json` file?

The `tsconfig.json` file is a TypeScript configuration file that tells the TypeScript compiler (`tsc`) how to compile your project.
It defines the root files, compiler options, and other settings needed to compile your TypeScript code correctly.

Purpose of `tsconfig.json`

- Controls how TypeScript behaves during compilation.
- Specifies which files to include or exclude.
- Enables strictness, module formats, target JS version, etc.
- Essential for larger projects to maintain consistency.

```
{
  "compilerOptions": {
    "target": "ES6",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "outDir": "dist",
    "rootDir": "src"
  },
  "include": ["src"],
```

```
  "exclude": ["node_modules", "dist"]
}
```

Open your `tsconfig.json`

```
{
  "compilerOptions": {
    "strict": true
  }
}
//Optional: Customize Specific Strict Options
{
  "compilerOptions": {
    "strict": true,
    "noImplicitAny": false  // selectively override
  }
}
```