MockAttempt

TECH ACADEMY • OFFICIAL COURSE SYLLABUS BROCHURE

MockAttempt Academy Pvt. Ltd. • Live Architect Mentorship, AI Assessments & Client Production

Admissions & Support:
support@mockattempt.com
Phone / WhatsApp: +91 95400 07917
Course Code: TSCR • **Updated:** 05 Nov 2026
https://mockattempt.com/courses/typescript-training

TSCR • FULL-STACK TYPE-SAFE JAVASCRIPT & SYSTEMS | **BEGINNER TO STAFF TYPESCRIPT ENGINEER**

100% LIVE INDUSTRY EXPERT COHORT + INTERNSHIP LETTER

TypeScript Training

Master Strict TypeScript 5 from `tsc` Compilation, Type Erasure, `unknown` vs `any` & `satisfies` Inference to `strictNullChecks`, `readonly` Arrays/Tuples, Discriminated Unions, `never` Exhaustiveness, `as const` vs Enums & Typed `Map`/`Set` Collections — with runnable TypeScript 5 programs, 244 Topic Mock Tests (10,440 MCQs), and Spoken Voice + MockAttempt AI Interviews at ₹1,799 (Incl. 18% GST).

Duration: 14–16 Weeks • Live + Hands-On Projects (205 Hrs) • **Modules:** 16 • **Live Topics:** 42 • **Subtopics:** 336+ • **Labs & Tasks:** 84 • **Mocks / AI Vivas:** 244 / 244 • **Projects:** 16 Module + 1 Live Client Prod

All-Inclusive Direct Fee (Incl. 18% GST • Zero Hidden Charges): Total Payable: ₹1,799 (Net Tuition ₹1,525 + Included 18% GST ₹274) • **Split Option:** 100% Credit to Job Track • **Effective ROI:** ~₹10/Guided Hr (~₹43/Live Topic)

Faculty: 100% Active Working IT Professionals • **Live Batch Tracks (7:00 AM – 8:00 PM IST Flexible Slots):** Weekday Live Online Batch (Mon – Fri • 1 Hr/Day • 7:00 AM – 8:00 PM) (Monday to Friday (1 Hour Live Session / Day) • 07:00 AM – 08:00 PM IST (Students Can Choose Any 1-Hour Slot Based on Trainer Availability) • 17–19 Weeks Fast-Track) | Weekend Live Online Batch (Sat – Sun • 2 Hrs/Day • 7:00 AM – 8:00 PM) (Saturday & Sunday (2 Hours Live Session / Day) • 07:00 AM – 08:00 PM IST (Students Can Choose Any 2-Hour Slot Based on Trainer Availability) • 19–21 Weeks Weekend Track)

✓ **Trainer LeetCode & HackerRank Mentorship (TypeScript 5 & JavaScript ES6+): Live trainer support to crack LeetCode & HackerRank problems, optimize complexity, and handle edge cases.**

Core Technologies & Engineering Stack: TypeScript 5.x (`strict: true`) • tsc Compiler, tsx & Node.js Native Strip • Structural Typing & Control-Flow Narrowing • `unknown`, `never` & `satisfies` Operator • Readonly Arrays, Labeled Tuples & Unions • `as const` Objects & String/Const Enums • Typed ES6 `Map<K,V>`, `Set<T>` & Iterators

1. PROGRAM MASTERY OUTCOMES & ELIGIBILITY

End-to-End Engineering Outcomes:

1. Configure rock-solid `tsconfig.json` (`strict: true`, `noUncheckedIndexedAccess`, `exactOptionalPropertyTypes`, `verbatimModuleSyntax`) across full-stack workspaces
2. Eliminate runtime `TypeError` crashes using `unknown` narrowing, `strictNullChecks`, optional chaining (`?.`), nullish coalescing (`??`), and `satisfies`
3. Model impossible-to-misuse domain states with Labeled `readonly` Tuples, Discriminated Unions, `never` exhaustiveness checks, and O(1) `Map` / `Set` structures
4. Clear 244 Timed Mock Exams (10,000 TypeScript MCQs) and 200 Spoken Voice + MockAttempt AI Full-Stack TypeScript Interviews
5. Strict compiler configuration (`strict: true`, `noUncheckedIndexedAccess`, `exactOptionalPropertyTypes`) with zero `any`.
6. Crack LeetCode and HackerRank problems in TypeScript 5 & JavaScript ES6+ with live trainer mentorship on algorithmic/query patterns, edge cases, and optimal complexity.
7. Advanced type-level modeling: Generics, Discriminated Unions, Type Narrowing, Mapped/Conditional Types, and `infer`.
8. Bridging compile-time types and untrusted I/O boundaries using Zod/Valibot runtime schema validation.

Eligibility & Prerequisites: Basic JavaScript familiarity (variables, functions, objects, arrays) — teaches static typing and TypeScript 5 compiler rules step-by-step from scratch • Any laptop/PC running Node.js 20/22 LTS, npm/pnpm, `typescript` (`tsc`), and VS Code • No prior TypeScript industry experience required — Module 1 starts from environment setup, core mental models, and plain-English fundamentals. • Ideal for B.Tech / BCA / MCA / B.Sc IT students, non-CS beginners, career switchers, and working developers upgrading to 2026 production standards. • Commitment to attend live Weekday or Weekend classes, complete daily hands-on lab checklists, and practice topic-wise Mock Tests & AI Interviews.

2. DETAILED ZERO-TO-EXPERT MODULE, TOPIC & SUBTOPIC SYLLABUS (16 MODULES • 42 LIVE TOPICS • 336+ SUBTOPICS)

Progressive Zero-to-Expert Mastery Stages: Modules 1–4: Stage 1: Day-1 Foundations & Core Mental Models (8 Topics / 64 Subtopics) • Modules 5–8: Stage 2: Applied Engineering, Data & API Workflows (8 Topics / 64 Subtopics) • Modules 9–12: Stage 3: Internals, Concurrency, Security & Performance (12 Topics / 96 Subtopics) • Modules 13–16: Stage 4: Production Architecture, Testing, CI/CD & Expert Mastery (14 Topics / 112 Subtopics)

MODULE 1 OF 16 • STAGE 1 • 10 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 1: TypeScript Foundations, Static Type Checking & Type Erasure vs JavaScript

Understand TypeScript as a typed syntactic superset of JavaScript, compile-time Type Erasure, and why static type checking catches bugs before production.

Objectives: ✓ TypeScript 5 Overview — Syntactic Superset, Compile-Time Type Erasure & Zero-Cost Abstractions — How TypeScript layers a Turing-complete structural type checker and language service over standard JavaScript while erasing all type annotations at emit time. • ✓ TypeScript vs JavaScript — Static Structural Checking vs Dynamic Coercion Traps — Preventing JavaScript's silent string-concatenation (`"999" + 1 === "9991"`), misspelled property `undefined` propagation, and refactoring regressions at compile time. • ✓ Build, verify, and ship Module Capstone: Zero-Runtime-Overhead Invoice Calculator & Boundary Type Guard.

1.1 TypeScript 5 Overview — Syntactic Superset, Compile-Time Type Erasure & Zero-Cost Abstractions

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Typed syntactic superset of JavaScript & Compile-time Type Erasure rule • Core Syntax, CLI & Production APIs: interface Invoice { readonly id: string; amount: number } + tsc --noEmit • Deep-Dive Mechanics: Zero-runtime-cost interface & type aliases & Language Server Protocol (tsserver) refactoring • Advanced Patterns: Tsc --noEmit CI typecheck gate & Runtime JSON boundary validation necessity • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Compile-Time Interface Contract + Runtime Type Predicate Guard • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript 5 Overview

Purpose: To understand and apply TypeScript 5 Overview — Syntactic Superset, Compile-Time Type Erasure & Zero-Cost Abstractions in real TypeScript systems — specifically: How Type...

What Topic Is Next → Next Topic 1.2: TypeScript vs JavaScript — Static Structural Checking vs Dynamic Coercion Traps

1.2 TypeScript vs JavaScript — Static Structural Checking vs Dynamic Coercion Traps

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Compile-time vs runtime error discovery & Preventing "999" + 1 implicit coercion bugs • Core Syntax, CLI & Production APIs: function computeNet(price: number, tax: number): number • Deep-Dive Mechanics: Arity checking on function arguments & Property typo detection via structural shapes • Advanced Patterns: Gradual allowJs / checkJs migration path & Self-documenting API signatures • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Strict Numeric Arithmetic vs Dynamic Coercion Guard • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript vs JavaScript

Purpose: To understand and apply TypeScript vs JavaScript — Static Structural Checking vs Dynamic Coercion Traps in real TypeScript systems — specifically: Preventing JavaScript's...

What Topic Is Next → Module 1 Capstone (Zero-Runtime-Overhead Invoice Calculator & Boundary Type Guard) → Next Topic 2.1: TypeScript Core Features — Structural ("Duck") Typing, Excess Property Checks & Generics

Module 1 Hands-On Capstone Project (100 Marks): Zero-Runtime-Overhead Invoice Calculator & Boundary Type Guard — Write a strict TypeScript 5 billing module demonstrating compile-time interface enforcement paired with a runtime `unknown` type-predicate guard for untrusted `JSON.parse` payloads.

MODULE 2 OF 16 • STAGE 1 • 10 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 2: Structural Duck Typing, Generics Overview & Production `tsconfig.json` Setup

Master structural ("duck") typing vs nominal typing, modern TypeScript 5 capabilities, and production `tsconfig.json` compiler flags (`strict`, `noUncheckedIndexedAccess`, `moduleResolution`).

Objectives: ✓ TypeScript Core Features — Structural ("Duck") Typing, Excess Property Checks & Generics — Why TypeScript checks value shapes structurally (`{ id: string }` matches any superset object variable) while enforcing strict Excess Property Checks on fresh object literals. • ✓ Installing & Configuring TypeScript 5 — Local `devDependencies`, `tsc --init` & `tsconfig.json` Flags — Setting up deterministic Node.js + TypeScript 5 toolchains (`npm i -D typescript tsx @types/node`) and understanding `strict`, `moduleResolution: "NodeNext" | "Bundler"`, and Node native type stripping. • ✓ Build, verify, and ship Module Capstone: Structural Repository Adapter & Hardened `tsconfig.json` Validator.

2.1 TypeScript Core Features — Structural ("Duck") Typing, Excess Property Checks & Generics

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Structural typing vs nominal typing & Fresh object literal excess property check • Core Syntax, CLI & Production APIs: interface Identifiable { readonly id: string } + function logId<T extends Identifiable>(item... • Deep-Dive Mechanics: Generic <T extends Constraint> reuse & ECMAScript stage-3 standard decorators • Advanced Patterns: Declaration .d.ts ambient type files & Open interface declaration merging • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Structural Shape Compatibility & Generic Constraint Demo • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript Core Features

Purpose: To understand and apply TypeScript Core Features — Structural ("Duck") Typing, Excess Property Checks & Generics in real TypeScript systems — specifically: Why TypeScript...

What Topic Is Next → Next Topic 2.2: Installing & Configuring TypeScript 5 — Local `devDependencies`, `tsc --init` & `tsconfig.json` Flags

2.2 Installing & Configuring TypeScript 5 — Local `devDependencies`, `tsc --init` & `tsconfig.json` Flags

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Local devDependencies vs global tsc anti-pattern & Strict: true umbrella compiler flag • Core Syntax, CLI & Production APIs: npm install -D typescript tsx @types/node && npx tsc --init • Deep-Dive Mechanics: NoUncheckedIndexedAccess array/record safety & ModuleResolution Bundler vs NodeNext • Advanced Patterns: Incremental .tsbuildinfo build cache & Tsx and Node --experimental-strip-types • Production Failure Mode & Pitfall Guard • Hands-On Live Coding

Lab: Production `tsconfig.json` Strictness Score Auditor • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for Installing

Purpose: To understand and apply Installing & Configuring TypeScript 5 — Local `devDependencies`, `tsc --init` & `tsconfig.json` Flags in real TypeScript systems — specifically: S...

What Topic Is Next → Module 2 Capstone (Structural Repository Adapter & Hardened `tsconfig.json` Validator) → Next Topic 3.1: First TypeScript Program — Module Scope Isolation, `tsc` Emit Diagnostics & `noEmitOnError`

Module 2 Hands-On Capstone Project (100 Marks): Structural Repository Adapter & Hardened `tsconfig.json` Validator — Build a structurally typed logger/storage adapter demonstrating shape compatibility across independent classes alongside a `tsconfig.json` strictness auditor.

MODULE 3 OF 16 • STAGE 1 • 10 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 3: First TypeScript Program & The Type Hierarchy (`unknown`, `any`, `never`, `void`)

Compile and run clean `.ts` programs while mastering TypeScript's complete type lattice from Top Types (`unknown`, unsafe `any`) down to primitives, literals, and the Bottom Type (`never`).

Objectives: ✓ First TypeScript Program — Module Scope Isolation, `tsc` Emit Diagnostics & `noEmitOnError` — Writing, type-checking, and executing your first strict TypeScript module while preventing legacy script-mode global variable collisions and broken `.js` emit. • ✓ TypeScript Type Hierarchy — Primitives (`string` vs `String`), Top Type `unknown`, `any`, `void` & Bottom Type `never` — Mastering the mathematical set hierarchy of TypeScript: `unknown` (universal supertype), primitives (`string`, `number`, `boolean`, `bigint`, `symbol`), `object`, `void`, and `never` (empty set). • ✓ Build, verify, and ship Module Capstone: Safe `unknown` Error Normalizer & Type Lattice Inspector.

3.1 First TypeScript Program — Module Scope Isolation, `tsc` Emit Diagnostics & `noEmitOnError`

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: ES module isolation vs global script scope & NoEmitOnError: true safety flag • Core Syntax, CLI & Production APIs: export function greet(user: string): string { ... } /* noEmitOnError: true */ • Deep-Dive Mechanics: SourceMap: true debugger mapping & Strict parameter annotation requirement • Advanced Patterns: Template literal string interpolation & Type-only import type { X } syntax • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Isolated ES Module Greeting & Cohort Formatter • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for First TypeScript Program

Purpose: To understand and apply First TypeScript Program — Module Scope Isolation, `tsc` Emit Diagnostics & `noEmitOnError` in real TypeScript systems — specifically: Writing, ty...

What Topic Is Next → Next Topic 3.2: TypeScript Type Hierarchy — Primitives (`string` vs `String`), Top Type `unknown`, `any`, `void` & Bottom Type `never`

3.2 TypeScript Type Hierarchy — Primitives (`string` vs `String`), Top Type `unknown`, `any`, `void` & Bottom Type `never`

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Unknown safe top type vs unsafe any & Never empty-set bottom type • Core Syntax, CLI & Production APIs: unknown (safe top type) | string | number | boolean | bigint | void | never (bottom type) • Deep-Dive Mechanics: Lowercase primitives vs uppercase Object wrappers & Void callback return semantics • Advanced Patterns: UseUnknownInCatchVariables catch (e: unknown) & Literal types ("paid", 200, true) as subsets • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Safe `unknown` Narrowing in `catch` & Payload Parsing • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript Type Hierarchy

Purpose: To understand and apply TypeScript Type Hierarchy — Primitives (`string` vs `String`), Top Type `unknown`, `any`, `void` & Bottom Type `never` in real TypeScript systems...

What Topic Is Next → Module 3 Capstone (Safe `unknown` Error Normalizer & Type Lattice Inspector) → Next Topic 4.1: TypeScript Type Annotations — Function Signatures, Return Contracts, Object Shapes & Index Signatures

Module 3 Hands-On Capstone Project (100 Marks): Safe `unknown` Error Normalizer & Type Lattice Inspector — Implement a zero-`any` error extraction utility (`toErrorMessage(err: unknown): string`) for `try / catch` blocks alongside an unreachable `assertNever(x: never): never` helper.

MODULE 4 OF 16 • STAGE 1 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 4: Explicit Type Annotations vs Type Inference & The `satisfies` Operator

Know exactly when to write explicit `: Type` annotations (public boundaries, empty collections) versus letting TypeScript infer narrow literal types (`const`, `as const`, `satisfies`).

Objectives: ✓ TypeScript Type Annotations — Function Signatures, Return Contracts, Object Shapes & Index Signatures — Placing intentional `: Type` annotations at module boundaries, function parameters/returns, and uninitialized/empty structures so accidental return-shape changes fail immediately inside the function. • ✓ TypeScript Type Inference — `const` vs `let` Widening, Contextual Typing & `satisfies` — How TypeScript's Bidirectional Inference Engine deduces narrow literal types on `const`, types inline `.map(x => ...)` callbacks contextually, and validates schemas non-destructively with `satisfies`. • ✓ Build, verify, and ship Module Capstone: Route & Theme Config Validator Powered by `satisfies` & Explicit API Return Contracts.

4.1 TypeScript Type Annotations — Function Signatures, Return Contracts, Object Shapes & Index Signatures

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Parameter and explicit return annotations & Record<K, V> vs index signatures • Core Syntax, CLI & Production APIs: function parsePort(raw: string): number + Record<string, number> • Deep-Dive Mechanics: Empty array initialization const ids: string[] = [] & Optional property prop?: T annotations • Advanced Patterns: Function type expressions (a: number) => boolean & IsolatedDeclarations monorepo speed • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Explicit Boundary Contract & Typed `Record` Aggregation • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript Type Annotations

Purpose: To understand and apply TypeScript Type Annotations — Function Signatures, Return Contracts, Object Shapes & Index Signatures in real TypeScript systems — specifically: P...

What Topic Is Next → Next Topic 4.2: TypeScript Type Inference — `const` vs `let` Widening, Contextual Typing & `satisfies`

4.2 TypeScript Type Inference — `const` vs `let` Widening, Contextual Typing & `satisfies` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Let primitive widening vs const literal type & Object property mutability widening trap • Core Syntax, CLI & Production APIs: const theme = { primary: '#2563eb', spacing: [4, 8] } satisfies Record<string, string | numb... • Deep-Dive Mechanics: Satisfies non-widening schema validation & Contextual typing in array/event callbacks • Advanced Patterns: Best common type array inference & As const deep readonly literal inference • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: `satisfies` Non-Widening Schema Check with Narrow Property Access • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript Type Inference

Purpose: To understand and apply TypeScript Type Inference — `const` vs `let` Widening, Contextual Typing & `satisfies` in real TypeScript systems — specifically: How TypeScript's...

What Topic Is Next → Module 4 Capstone (Route & Theme Config Validator Powered by `satisfies` & Explicit API Return Contracts) → Next Topic 5.1: TypeScript Type Assertions — `as T` Unsoundness Risks, Double Assertions & Safe `as const`

Module 4 Hands-On Capstone Project (100 Marks): Route & Theme Config Validator Powered by `satisfies` & Explicit API Return Contracts — Design a configuration map validated with `satisfies Record<string, RouteConfig>` so typos fail compilation while individual route paths retain exact literal autocomplete.

MODULE 5 OF 16 • STAGE 2 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 5: Type Assertions (`as` vs `as const`) & Strict Nullability (`null` vs `undefined`)

Understand why `as Type` assertions bypass soundness (and never convert values at runtime!), and eliminate billion-dollar null crashes with `strictNullChecks`, `?.`, `??`, and `exactOptionalPropertyTypes`.

Objectives: ✓ TypeScript Type Assertions — `as T` Unsoundness Risks, Double Assertions & Safe `as const` — Distinguishing dangerous compile-time silencers (`el as HTMLInputElement`, `x as unknown as T`) from safe immutability narrowers (`as const`) and runtime `instanceof` checks. • ✓ `null` vs `undefined` in Strict TypeScript — `strictNullChecks`, `?.` & Nullish Coalescing `??` — Distinguishing `undefined` (missing property / uninitialized) from `null` (explicit empty value), JSON serialization differences, and why `??` beats `||` for numeric/boolean defaults. • ✓ Build, verify, and ship Module Capstone: Null-Safe User Preference Merger (`??` vs `||`) & `as const` Permission Matrix.

5.1 TypeScript Type Assertions — `as T` Unsoundness Risks, Double Assertions & Safe `as const` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: As T compile-time assertion (not runtime cast) & Disjoint assertion rejection protection • Core Syntax, CLI & Production APIs: const ROLES = ['student', 'trainer', 'admin'] as const; type Role = (typeof ROLES)[number] • Deep-Dive Mechanics: Danger of as unknown as T escape hatch & Instanceof HTMLInputElement safe DOM narrowing • Advanced Patterns: As const deep readonly literal freezing & Non-null assertion ! code-smell hazard • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Single-Source-of-Truth `as const` Union Derivation • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript Type Assertions

Purpose: To understand and apply TypeScript Type Assertions — `as T` Unsoundness Risks, Double Assertions & Safe `as const` in real TypeScript systems — specifically: Distinguishi...

What Topic Is Next → Next Topic 5.2: `null` vs `undefined` in Strict TypeScript — `strictNullChecks`, `?.` & Nullish Coalescing `??`

5.2 `null` vs `undefined` in Strict TypeScript — `strictNullChecks`, `?.` & Nullish Coalescing `??` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: StrictNullChecks: true soundness & Undefined (omitted) vs null (explicit empty) • Core Syntax, CLI & Production APIs: const retries = config?.retries ?? 3 /* preserves 0, unlike || */ • Deep-Dive Mechanics: Nullish coalescing ?? vs falsy || bug on 0/"" & Optional chaining ?. short-circuiting • Advanced Patterns: JSON.stringify drops undefined keys vs keeps null & ExactOptionalPropertyTypes strictness • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: `??` vs `||` Zero-Retry Preservation & JSON Nullability Test • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for `null` vs `undefined` in Strict TypeScript

Purpose: To understand and apply `null` vs `undefined` in Strict TypeScript — `strictNullChecks`, `?.` & Nullish Coalescing `??` in real TypeScript systems — specifically: Disting...

What Topic Is Next → Module 5 Capstone (Null-Safe User Preference Merger (`??` vs `||`) & `as const` Permission Matrix) → Next Topic 6.1: TypeScript Variables — `const` vs `let` vs `var`, Temporal Dead Zone (TDZ) & `Readonly<T>`

Module 5 Hands-On Capstone Project (100 Marks): Null-Safe User Preference Merger (`??` vs `||`) & `as const` Permission Matrix — Build a configuration resolver that preserves valid falsy values (`timeoutMs: 0`, `prefix: ""`) using nullish coalescing `??` instead of `||`, paired with an `as const` role tuple.

MODULE 6 OF 16 • STAGE 2 • 10 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 6: Block-Scoped Bindings (`const` / `let` / TDZ) & Immutable `readonly` Arrays

Eliminate `var` hoisting/closure bugs with Temporal Dead Zone (`const` / `let`) rules and prevent accidental in-place mutation bugs with `readonly T[]` and `noUncheckedIndexedAccess`.

Objectives: ✓ TypeScript Variables — `const` vs `let` vs `var`, Temporal Dead Zone (TDZ) & `Readonly<T>` — Why modern TypeScript forbids `var`, enforces Temporal Dead Zone checks on `let`/`const`, and pairs `const` variable bindings with `Readonly<T>` object immutability. • ✓ TypeScript Arrays — `readonly T[]`, Numeric `.sort()` Trap & `noUncheckedIndexedAccess` — Typing homogeneous and union arrays (`string[]`, `readonly number[]`), avoiding in-place

`.sort()`/`.reverse()` caller mutations, and catching `arr[i]` `undefined` reads. • ✓ Build, verify, and ship Module Capstone: Non-Mutating Score Sorter (`readonly number[]`) & Safe Bounds Reader.

6.1 TypeScript Variables — `const` vs `let` vs `var`, Temporal Dead Zone (TDZ) & `Readonly<T>` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Block-scoped const and let vs function-scoped var & Temporal Dead Zone (TDZ) before initialization • Core Syntax, CLI & Production APIs: const CONFIG: Readonly<ServerOpts> = Object.freeze({ ... }) • Deep-Dive Mechanics: Per-iteration closure capture in for (let i...) & Const binding vs Readonly<T> property immutability • Advanced Patterns: Object and array destructuring with types & Object.freeze shallow runtime immutability • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Per-Iteration `let` Closure Capture & `Readonly<T>` Immutability • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript Variables

Purpose: To understand and apply TypeScript Variables — `const` vs `let` vs `var`, Temporal Dead Zone (TDZ) & `Readonly<T>` in real TypeScript systems — specifically: Why modern T...

What Topic Is Next → Next Topic 6.2: TypeScript Arrays — `readonly T[]`, Numeric `.sort()` Trap & `noUncheckedIndexedAccess`

6.2 TypeScript Arrays — `readonly T[]`, Numeric `.sort()` Trap & `noUncheckedIndexedAccess` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Readonly T[] / ReadonlyArray<T> variance & Lexicographic .sort() bug without (a, b) => a - b • Core Syntax, CLI & Production APIs: function median(nums: readonly number[]): number | undefined • Deep-Dive Mechanics: Non-mutating [...arr].sort() / toSorted() & NoUncheckedIndexedAccess T | undefined safety • Advanced Patterns: Typed .map, .filter, and .reduce<Acc> & Type-predicate .filter((x): x is T => ...) • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Non-Mutating Numeric Sort on `readonly number[]` • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript Arrays

Purpose: To understand and apply TypeScript Arrays — `readonly T[]`, Numeric `.sort()` Trap & `noUncheckedIndexedAccess` in real TypeScript systems — specifically: Typing homogene...

What Topic Is Next → Module 6 Capstone (Non-Mutating Score Sorter (`readonly number[]`) & Safe Bounds Reader) → Next Topic 7.1: TypeScript Tuples — Fixed-Arity Positional Types, Labeled Elements, Rest Tuples & `readonly` Guard

Module 6 Hands-On Capstone Project (100 Marks): Non-Mutating Score Sorter (`readonly number[]`) & Safe Bounds Reader — Write pure array analytics functions accepting `readonly number[]` (banning `.sort()` / `.splice()` / `.push()` mutations on caller inputs) and returning safe `number | undefined` bounds res...

MODULE 7 OF 16 • STAGE 2 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 7: Labeled `readonly` Tuples & Exhaustive Discriminated Union Types

Model fixed-arity positional records with `readonly` Labeled Tuples and build ironclad `Result<T, E>` state machines using Discriminated Unions and `never` exhaustiveness.

Objectives: ✓ TypeScript Tuples — Fixed-Arity Positional Types, Labeled Elements, Rest Tuples & `readonly` Guard — Representing React-style `[value, setValue]` hooks, `[lat, lng]` coordinates, and variadic `[first: string, ...rest: number[]]` tuples with per-index static types. • ✓ TypeScript Union Types (`A | B`) — Discriminated Unions, Shared Members & `never` Exhaustiveness — Modeling algebraic sum types (`Result<T, E>`, payment states, Redux/Signal events) where a shared literal tag (`kind`) narrows the entire object shape automatically. • ✓ Build, verify, and ship Module Capstone: Type-Safe `Result<T, E>` Parser Returning `readonly` Coordinate Tuples.

7.1 TypeScript Tuples — Fixed-Arity Positional Types, Labeled Elements, Rest Tuples & `readonly` Guard [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Fixed-length per-index tuple types & Readonly [a: T, b: U] .push() protection • Core Syntax, CLI & Production APIs: type GeoPoint = readonly [lat: number, lng: number, label?: string] • Deep-Dive Mechanics: Labeled tuple elements for IDE clarity & Optional [x: number, y?: number] elements • Advanced Patterns: Variadic rest tuples [cmd: string, ...args: string[]] & Safe positional destructuring const [ok, val] = ... • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Labeled `readonly` Tuple Destructuring & Distance Formatter • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript Tuples

Purpose: To understand and apply TypeScript Tuples — Fixed-Arity Positional Types, Labeled Elements, Rest Tuples & `readonly` Guard in real TypeScript systems — specifically: Repr...

What Topic Is Next → Next Topic 7.2: TypeScript Union Types (`A | B`) — Discriminated Unions, Shared Members & `never` Exhaustiveness

7.2 TypeScript Union Types (`A | B`) — Discriminated Unions, Shared Members & `never` Exhaustiveness [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Algebraic sum types (A | B) & Discriminated (tagged) union pattern • Core Syntax, CLI & Production APIs: type Result<T> = { ok: true; data: T } | { ok: false; error: string } • Deep-Dive Mechanics: Common-member-only pre-narrowing rule & Typeof, in, and instanceof narrowing • Advanced Patterns: Never exhaustiveness check in default branch & Making impossible states unrepresentable • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Discriminated `PaymentEvent` Union with `never` Exhaustiveness • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript Union Types (`A | B`)

Purpose: To understand and apply TypeScript Union Types (`A | B`) — Discriminated Unions, Shared Members & `never` Exhaustiveness in real TypeScript systems — specifically: Modeli...

What Topic Is Next → Module 7 Capstone (Type-Safe `Result<T, E>` Parser Returning `readonly` Coordinate Tuples) → Next Topic 8.1: TypeScript Enums — Numeric Reverse-Mapping Quirks, String Enums & `as const` (`erasableSyntaxOnly`)

Module 7 Hands-On Capstone Project (100 Marks): Type-Safe `Result<T, E>` Parser Returning `readonly` Coordinate Tuples — Build a GPS coordinate parser returning a discriminated union `{ ok: true; point: readonly [lat: number, lng: number] } | { ok: false; reason: string }` with an exhaustive formatter.

MODULE 8 OF 16 • STAGE 2 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 8: Enums (`enum` vs `as const` Erasable Syntax) & Typed `Map<K, V>` Dictionaries

Understand numeric reverse-mapping quirks vs string enums vs modern `as const` objects (`erasableSyntaxOnly`), and build O(1) prototype-safe dictionaries with `Map<K, V>`.

Objectives: ✓ TypeScript Enums — Numeric Reverse-Mapping Quirks, String Enums & `as const` (`erasableSyntaxOnly`) — Why `enum` is one of the few non-erasable TypeScript constructs emitting runtime IIFE objects, how numeric enums double-keys via reverse mapping, and why `as const` is often preferred. • ✓ TypeScript `Map<K, V>` — O(1) Hash Lookups, Insertion Order, Non-String Keys & Prototype Safety — Using generic `new Map<K, V>()` for high-churn caches, object/number keys, guaranteed insertion-order iteration, O(1) `.size`, and zero `Object.prototype` (`__proto__` / `toString`) collisions. • ✓ Build, verify, and ship Module Capstone: Prototype-Safe LRU / Rate-Limit Counter Using `Map<string, Entry>` & `as const` Status Codes.

8.1 TypeScript Enums — Numeric Reverse-Mapping Quirks, String Enums & `as const` (`erasableSyntaxOnly`) [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Non-erasable runtime IIFE emit of enum & Numeric enum reverse mapping (Object.keys double size) • Core Syntax, CLI & Production APIs: enum OrderStatus { Pending = 'PENDING', Paid = 'PAID' } /* vs as const object */ • Deep-Dive Mechanics: Numeric enum 0 falsy bug (if (!status)) & Explicit String Enum debuggability • Advanced Patterns: Const enum isolatedModules caveat & As const object + (typeof OBJ)[keyof typeof OBJ] pattern • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: String Enum vs Numeric Reverse-Mapping Inspection • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript Enums

Purpose: To understand and apply TypeScript Enums — Numeric Reverse-Mapping Quirks, String Enums & `as const` (`erasableSyntaxOnly`) in real TypeScript systems — specifically: Why...

What Topic Is Next → Next Topic 8.2: TypeScript `Map<K, V>` — O(1) Hash Lookups, Insertion Order, Non-String Keys & Prototype Safety

8.2 TypeScript `Map<K, V>` — O(1) Hash Lookups, Insertion Order, Non-String Keys & Prototype Safety [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Map<K, V> O(1) get/set/has/delete/size & Immunity to __proto__ / constructor key collisions • Core Syntax, CLI & Production APIs: const cache = new Map<string, Session>(); cache.set(id, s); cache.get(id) // Session | undef... • Deep-Dive Mechanics: Non-string keys (numbers, objects, symbols) & Guaranteed insertion-order iteration • Advanced Patterns: Map.get(k) returns V | undefined & JSON.stringify(map) === "{}" gotcha • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Prototype-Safe `Map<string, number>` Counter & Safe Serialization • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript `Map<K

Purpose: To understand and apply TypeScript `Map<K, V>` — O(1) Hash Lookups, Insertion Order, Non-String Keys & Prototype Safety in real TypeScript systems — specifically: Using g...

What Topic Is Next → Module 8 Capstone (Prototype-Safe LRU / Rate-Limit Counter Using `Map<string, Entry>` & `as const` Status Codes) → Next Topic 9.1: TypeScript `Set<T>` — O(1) Membership, Deduplication, `SameValueZero` & Object Reference Caveats

Module 8 Hands-On Capstone Project (100 Marks): Prototype-Safe LRU / Rate-Limit Counter Using `Map<string, Entry>` & `as const` Status Codes — Build an O(1) sliding request counter backed by `Map<string, number>` (maintaining insertion order and immunity to `"__proto__"` keys) paired with an `as const` status map.

MODULE 9 OF 16 • STAGE 3 • 11 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 9: Unique `Set<T>` Algebra & Control-Flow Type Narrowing (`if` / `switch`)

Deduplicate & test membership in O(1) with `Set<T>` (`SameValueZero` equality) and master TypeScript's Control-Flow Analysis across `if/else`, `typeof`, `in`, `instanceof`, and `switch`.

Objectives: ✓ TypeScript `Set<T>` — O(1) Membership, Deduplication, `SameValueZero` & Object Reference Caveats — Eliminating O(N) `Array.includes` bottlenecks with `Set<T>` / `ReadonlySet<T>`, deduplicating primitives (`[...new Set(items)]`), and avoiding duplicate-object reference traps. • ✓ Decision Making & Control-Flow Narrowing — `if/else`, `typeof null` Trap, `in`, `Array.isArray` & `switch` — How TypeScript's Control-Flow Graph tracks `if`, `else`, `return` early exits, `typeof`, `in`, `instanceof`, and `switch (true)` to narrow union types automatically. • ✓ Build, verify, and ship Module Capstone: O(N) Permission Intersection Engine (`Set<string>`) & Control-Flow Narrowing Router.

9.1 TypeScript `Set<T>` — O(1) Membership, Deduplication, `SameValueZero` & Object Reference Caveats [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Set<T> and ReadonlySet<T> O(1) .has() lookup & [...new Set(arr)] O(N) insertion-order deduplication • Core Syntax, CLI & Production APIs: const allowed: ReadonlySet<string> = new Set(['read', 'write']); allowed.has(scope) • Deep-Dive Mechanics: SameValueZero equality (NaN === NaN) & Object reference identity trap in Set<Obj> • Advanced Patterns: ES2024 union/intersection/difference Set methods & JSON serialization via Array.from(set) • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: O(N + M) `ReadonlySet` Skill Matcher & Reference Identity Check • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript `Set<T>`

Purpose: To understand and apply TypeScript `Set<T>` — O(1) Membership, Deduplication, `SameValueZero` & Object Reference Caveats in real TypeScript systems — specifically: Elimin...

What Topic Is Next → Next Topic 9.2: Decision Making & Control-Flow Narrowing — `if/else`, `typeof null` Trap, `in`, `Array.isArray` & `switch`

9.2 Decision Making & Control-Flow Narrowing — `if/else`, `typeof null` Trap, `in`, `Array.isArray` & `switch` [50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: Control-Flow Analysis (CFA) graph narrowing & Early-return guard clause narrowing • Core Syntax, CLI & Production APIs: if (typeof x === 'string') { ... } else if (Array.isArray(x)) { ... } • Deep-Dive Mechanics: Typeof null === "object" JavaScript quirk & Array.isArray() vs typeof array check • Advanced Patterns: "prop" in obj structural branch narrowing & Truthiness narrowing hazard on 0 and "" • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab:

Null-Safe Control-Flow Narrowing Across Primitives, Arrays & Objects • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for Decision Making

Purpose: To understand and apply Decision Making & Control-Flow Narrowing — `if/else`, `typeof null` Trap, `in`, `Array.isArray` & `switch` in real TypeScript systems — specifical...

What Topic Is Next → Module 9 Capstone (O(N) Permission Intersection Engine (`Set<string>`) & Control-Flow Narrowing Router) → Next Topic 10.1: TypeScript Definite Loops — Classic `for`, `for...of` Iterables vs `for...in` Keys & `Object.keys` Widening

Module 9 Hands-On Capstone Project (100 Marks): O(N) Permission Intersection Engine (`Set<string>`) & Control-Flow Narrowing Router — Build an O(N + M) role-permission verifier using `Set<string>` alongside a multi-input command router narrowing `string | number | { cmd: string } | null` cleanly.

MODULE 10 OF 16 • STAGE 3 • 10 LIVE + LAB HOURS • 2 TOPICS (16 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Module 10: Definite Loops (`for`, `for...of`, `for...in`) & `.forEach()` Async Pitfalls

Choose accurately between `for`, `for...of` (values + `await` + `break`), `for...in` (`string` keys), and `.forEach()` — avoiding classic unhandled-Promise `.forEach(async ...)` production bugs!

Objectives: ✓ TypeScript Definite Loops — Classic `for`, `for...of` Iterables vs `for...in` Keys & `Object.keys` Widening — Iterating arrays, `Map`s, `Set`s, and strings cleanly with `for...of` (`entries()`) while avoiding `for...in` array string-index and prototype-chain pitfalls. • ✓ TypeScript `Array.prototype.forEach` — `void` Return Ignorance, No `break`, & The `async` Trap — Understanding `.forEach((item, index) => void)` callback typing, why `break` and `return` cannot exit the outer function, and why `.forEach(async ...)` fires unawaited floating Promises!. • ✓ Build, verify, and ship Module Capstone: Sequential Async Batch Processor (`for...of` + `await`) vs Parallel `Promise.all` Runner.

10.1 TypeScript Definite Loops — Classic `for`, `for...of` Iterables vs `for...in` Keys & `Object.keys` Widening

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: For...of values vs for...in string keys & Never use for...in on arrays ("0" + 1 === "01") • Core Syntax, CLI & Production APIs: for (const [index, value] of items.entries()) { if (done) break; } • Deep-Dive Mechanics: Items.entries() typed [number, T] tuple loop & Break, continue, return, and await support • Advanced Patterns: Why Object.keys() returns string[] structurally & Object.hasOwn() prototype guard • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Typed `for...of` `.entries()` Early-Break Search • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript Definite Loops

Purpose: To understand and apply TypeScript Definite Loops — Classic `for`, `for...of` Iterables vs `for...in` Keys & `Object.keys` Widening in real TypeScript systems — specifica...

What Topic Is Next → Next Topic 10.2: TypeScript `Array.prototype.forEach` — `void` Return Ignorance, No `break`, & The `async` Trap

10.2 TypeScript `Array.prototype.forEach` — `void` Return Ignorance, No `break`, & The `async` Trap

[50 mins • Assignment (100M) • Hands-On Lab (50M) • 10 Mock Sets • 10 AI Vivas]

Subtopics Covered: .forEach(async ...) floating-Promise anti-pattern & For...of + await sequential async alternative • Core Syntax, CLI & Production APIs: // Sequential async: for (const x of list) { await work(x); } /* NEVER list.forEach(async .... • Deep-Dive Mechanics: Promise.all(arr.map(...)) parallel async alternative & Return inside .forEach only skips current iteration • Advanced Patterns: Sparse array empty-slot skipping in .forEach & Preferring .map / .filter / .some over mutating .forEach • Production Failure Mode & Pitfall Guard • Hands-On Live Coding Lab: Sequential `for...of` `await` vs Broken `.forEach(async)` Order Proof • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for TypeScript `Array.prototype.forEach`

Purpose: To understand and apply TypeScript `Array.prototype.forEach` — `void` Return Ignorance, No `break`, & The `async` Trap in real TypeScript systems — specifically: Understa...

What Topic Is Next → Module 10 Capstone (Sequential Async Batch Processor (`for...of` + `await`) vs Parallel `Promise.all` Runner) → Next Topic 11.1: Distributive Conditional Types & the infer Keyword

Module 10 Hands-On Capstone Project (100 Marks): Sequential Async Batch Processor (`for...of` + `await`) vs Parallel `Promise.all` Runner — Replace a broken `items.forEach(async item => ...)` pipeline with deterministic sequential `for (const item of items) { await ... }` and bounded `Promise.all(items.map(...))` execution.

MODULE 11 OF 16 • STAGE 3 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Type-Level Programming: Conditional Types, infer & Template Literals

Production-grade TypeScript Training engineering module covering Type-Level Programming: Conditional Types, infer & Template Literals with hands-on labs and capstone build.

Objectives: ✓ Distributive Conditional Types & the infer Keyword — Purpose: Extract Promise payloads, function parameters, and API response shapes automatically. • ✓ Recursive Mapped Types & Template Literal Path Parsers — Purpose: Parse \"/orders/:orderId/items/:itemId\" into exact { orderId: string; itemId: string } params at compile time!. • ✓ Branded / Nominal Types (UserId vs OrderId, INRPaise) & const Type Params — Purpose: Make it a compile error to accidentally pass an OrderId where a UserId is expected. • ✓ satisfies Operator vs Type Annotations — Purpose: Validate config objects against a schema while preserving exact literal autocomplete.

11.1 Distributive Conditional Types & the infer Keyword

[65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of the infer Keyword • Integration with Recursive Mapped Types & Template Literal Path Parsers • Live Debugging, Telemetry & Technical Interview Trade • What Is Distributive Conditional Types & the infer Keyword & Why Industry Teams Use It • In TypeScript Training, Distributive Conditional Types & the infer Keyword exists for one clear real • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply Distributive Conditional Types & the infer Keyword in real TypeScript systems — specifically: Extract Promise payloads, function parameters, and A...

What Topic Is Next → Next Topic 11.2: Recursive Mapped Types & Template Literal Path Parsers

11.2 Recursive Mapped Types & Template Literal Path Parsers [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Template Literal Path Parsers • Integration with Branded / Nominal Types (UserId vs OrderId, INRPaise) & const Type Params • Live Debugging, Telemetry & Technical Interview Trade • What Is Recursive Mapped Types & Template Literal Path Parsers & Why Industry Teams Use It • In TypeScript Training, Recursive Mapped Types & Template Literal Path Parsers exists for one clear real • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply Recursive Mapped Types & Template Literal Path Parsers in real TypeScript systems — specifically: Parse \"/orders/:orderId/items/:itemId\" into ex...

What Topic Is Next → Next Topic 11.3: Branded / Nominal Types (UserId vs OrderId, INRPaise) & const Type Params

11.3 Branded / Nominal Types (UserId vs OrderId, INRPaise) & const Type Params [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of INRPaise) • Integration with satisfies Operator vs Type Annotations • Live Debugging, Telemetry & Technical Interview Trade • What Is Branded / Nominal Types (UserId vs OrderId, INRPaise) & const Type Params & Why Industry Teams Use It • In TypeScript Training, Branded / Nominal Types (UserId vs OrderId, INRPaise) & const Type Params exists for one cl... • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply Branded / Nominal Types (UserId vs OrderId, INRPaise) & const Type Params in real TypeScript systems — specifically: Make it a compile error to ac...

What Topic Is Next → Next Topic 11.4: satisfies Operator vs Type Annotations

11.4 satisfies Operator vs Type Annotations [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Edge • Integration with Distributive Conditional Types & the infer Keyword • Live Debugging, Telemetry & Technical Interview Trade • What Is satisfies Operator vs Type Annotations & Why Industry Teams Use It • In TypeScript Training, satisfies Operator vs Type Annotations exists for one clear real • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply satisfies Operator vs Type Annotations in real TypeScript systems — specifically: Validate config objects against a schema while preserving exact...

What Topic Is Next → Module 11 Capstone (Compile-Time Typed Router & Event-Bus SDK in TypeScript) → Next Topic 12.1: Zod Transforms, Refinements, Discriminated Unions & z.infer

Module 11 Hands-On Capstone Project (100 Marks): Compile-Time Typed Router & Event-Bus SDK in TypeScript — Design, implement, test, and document 'Compile-Time Typed Router & Event-Bus SDK in TypeScript' applying every production concept from Module 11 (Type-Level Programming: Conditional Types, i...

MODULE 12 OF 16 • STAGE 3 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Runtime Schema Validation (Zod), Result Error Unions & End-to-End RPC

Production-grade TypeScript Training engineering module covering Runtime Schema Validation (Zod), Result Error Unions & End-to-End RPC with hands-on labs and capstone build.

Objectives: ✓ Zod Transforms, Refinements, Discriminated Unions & z.infer — Purpose: Parse untrusted HTTP/JSON/env inputs at system boundaries into guaranteed TypeScript types. • ✓ Typed Result<T, E> Unions vs Untyped Thrown Exceptions — Purpose: Force callers to handle NotFound, QuotaExceeded, and ValidationError explicitly. • ✓ OpenAPI-to-TypeScript Codegen & End-to-End Typed API Clients — Purpose: Catch backend response field renames immediately in frontend CI typechecks. • ✓ Strict TSConfig Hardening (noUncheckedIndexedAccess, exactOptionalPropertyTypes) — Purpose: Eliminate silent undefined array-index bugs across production codebases.

12.1 Zod Transforms, Refinements, Discriminated Unions & z.infer [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Refinements • Integration with Typed Result<T, E> Unions vs Untyped Thrown Exceptions • Live Debugging, Telemetry & Technical Interview Trade • Integration with Typed Result Unions vs Untyped Thrown Exceptions • What Is Zod Transforms, Refinements, Discriminated Unions & z.infer & Why Industry Teams Use It • In TypeScript Training, Zod Transforms, Refinements, Discriminated Unions & z.infer exists for one clear real • Even if you come from a non

Purpose: To understand and apply Zod Transforms, Refinements, Discriminated Unions & z.infer in real TypeScript systems — specifically: Parse untrusted HTTP/JSON/env inputs at sys...

What Topic Is Next → Next Topic 12.2: Typed Result<T, E> Unions vs Untyped Thrown Exceptions

12.2 Typed Result<T, E> Unions vs Untyped Thrown Exceptions [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of E> Unions vs Untyped Thrown Exceptions • Integration with OpenAPI • Live Debugging, Telemetry & Technical Interview Trade • What Is Typed Result Unions vs Untyped Thrown Exceptions & Why Industry Teams Use It • In TypeScript Training, Typed Result Unions vs Untyped Thrown Exceptions exists for one clear real • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply Typed Result<T, E> Unions vs Untyped Thrown Exceptions in real TypeScript systems — specifically: Force callers to handle NotFound, QuotaExceeded,...

What Topic Is Next → Next Topic 12.3: OpenAPI-to-TypeScript Codegen & End-to-End Typed API Clients

12.3 OpenAPI-to-TypeScript Codegen & End-to-End Typed API Clients [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of End • Integration with Strict TSConfig Hardening (noUncheckedIndexedAccess, exactOptionalPropertyTypes) • Live Debugging, Telemetry & Technical Interview Trade • What Is OpenAPI • In TypeScript Training, OpenAPI • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply OpenAPI-to-TypeScript Codegen & End-to-End Typed API Clients in real TypeScript systems — specifically: Catch backend response field renames immed...

What Topic Is Next → Next Topic 12.4: Strict TSConfig Hardening (noUncheckedIndexedAccess, exactOptionalPropertyTypes)

12.4 Strict TSConfig Hardening (noUncheckedIndexedAccess, exactOptionalPropertyTypes) [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of exactOptionalPropertyTypes) • Integration with Zod Transforms, Refinements, Discriminated Unions & z.infer • Live Debugging, Telemetry & Technical Interview Trade • What Is Strict TSConfig Hardening (noUncheckedIndexedAccess, exactOptionalPropertyTypes) & Why Industry Teams Use I... • In TypeScript Training, Strict TSConfig Hardening (noUncheckedIndexedAccess, exactOptionalPropertyTypes) exists for... • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply Strict TSConfig Hardening (noUncheckedIndexedAccess, exactOptionalPropertyTypes) in real TypeScript systems — specifically: Eliminate silent undef...

What Topic Is Next → Module 12 Capstone (Zero-Drift Full-Stack Type Contract & Env Validator) → Next Topic 13.1: Polymorphic React Components (as Prop), Forwarded Refs & Discriminated Props

Module 12 Hands-On Capstone Project (100 Marks): Zero-Drift Full-Stack Type Contract & Env Validator — Design, implement, test, and document 'Zero-Drift Full-Stack Type Contract & Env Validator' applying every production concept from Module 12 (Runtime Schema Validation (Zod), Result Error Un...

MODULE 13 OF 16 • STAGE 4 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Full-Stack TypeScript Patterns Across React Props, Hooks & Node Repositories

Production-grade TypeScript Training engineering module covering Full-Stack TypeScript Patterns Across React Props, Hooks & Node Repositories with hands-on labs and capstone build.

Objectives: ✓ Polymorphic React Components (as Prop), Forwarded Refs & Discriminated Props — Purpose: Build ergonomic UI components where mutually exclusive props cannot be mixed. • ✓ Typed Express Request/Response Generics & Async Middleware Wrappers — Purpose: Flow validated params, query, body, and auth user types cleanly into controllers. • ✓ Type-Safe SQL/ORM Query Builders (Prisma / Drizzle / Kysely Concepts) — Purpose: Write relational joins where selected column names and nullability are 100% inferred. • ✓ ECMAScript Decorators, Symbols & AsyncIterators for Streaming Pagination — Purpose: Consume paginated partner APIs cleanly via for await (const page of client.list()).

13.1 Polymorphic React Components (as Prop), Forwarded Refs & Discriminated Props [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Forwarded Refs • Integration with Typed Express Request/Response Generics & Async Middleware Wrappers • Live Debugging, Telemetry & Technical Interview Trade • What Is Polymorphic React Components (as Prop), Forwarded Refs & Discriminated Props & Why Industry Teams Use It • In TypeScript Training, Polymorphic React Components (as Prop), Forwarded Refs & Discriminated Props exists for one... • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply Polymorphic React Components (as Prop), Forwarded Refs & Discriminated Props in real TypeScript systems — specifically: Build ergonomic UI compone...

What Topic Is Next → Next Topic 13.2: Typed Express Request/Response Generics & Async Middleware Wrappers

13.2 Typed Express Request/Response Generics & Async Middleware Wrappers [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Async Middleware Wrappers • Integration with Type • Live Debugging, Telemetry & Technical Interview Trade • What Is Typed Express Request/Response Generics & Async Middleware Wrappers & Why Industry Teams Use It • In TypeScript Training, Typed Express Request/Response Generics & Async Middleware Wrappers exists for one clear re... • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply Typed Express Request/Response Generics & Async Middleware Wrappers in real TypeScript systems — specifically: Flow validated params, query, body,...

What Topic Is Next → Next Topic 13.3: Type-Safe SQL/ORM Query Builders (Prisma / Drizzle / Kysely Concepts)

13.3 Type-Safe SQL/ORM Query Builders (Prisma / Drizzle / Kysely Concepts) [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Edge • Integration with ECMAScript Decorators, Symbols & AsyncIterators for Streaming Pagination • Live Debugging, Telemetry & Technical Interview Trade • What Is Type • In TypeScript Training, Type • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply Type-Safe SQL/ORM Query Builders (Prisma / Drizzle / Kysely Concepts) in real TypeScript systems — specifically: Write relational joins where sele...

What Topic Is Next → Next Topic 13.4: ECMAScript Decorators, Symbols & AsyncIterators for Streaming Pagination

13.4 ECMAScript Decorators, Symbols & AsyncIterators for Streaming Pagination [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Symbols • Integration with Polymorphic React Components (as Prop), Forwarded Refs & Discriminated Props • Live Debugging, Telemetry & Technical Interview Trade • What Is ECMAScript Decorators, Symbols & AsyncIterators for Streaming Pagination & Why Industry Teams Use It • In TypeScript Training, ECMAScript Decorators, Symbols & AsyncIterators for Streaming Pagination exists for one cle... • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply ECMAScript Decorators, Symbols & AsyncIterators for Streaming Pagination in real TypeScript systems — specifically: Consume paginated partner APIs...

What Topic Is Next → Module 13 Capstone (Type-Safe Multi-Tenant Express + React Workspace Toolkit) → Next Topic 14.1: TS Project References (composite: true) & Turborepo/pnpm Workspaces

Module 13 Hands-On Capstone Project (100 Marks): Type-Safe Multi-Tenant Express + React Workspace Toolkit — Design, implement, test, and document 'Type-Safe Multi-Tenant Express + React Workspace Toolkit' applying every production concept from Module 13 (Full-Stack TypeScript Patterns Across React...

MODULE 14 OF 16 • STAGE 4 • 16 LIVE + LAB HOURS • 4 TOPICS (32 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Monorepo Project References, Dual ESM/CJS Library Publishing & Type Tests

Production-grade TypeScript Training engineering module covering Monorepo Project References, Dual ESM/CJS Library Publishing & Type Tests with hands-on labs and capstone build.

Objectives: ✓ TS Project References (composite: true) & Turborepo/pnpm Workspaces — Purpose: Typecheck 10-package full-stack monorepos in seconds using incremental .tsbuildinfo caches. • ✓ Modern moduleResolution: \"NodeNext\" vs \"Bundler\" & package.json exports — Purpose: Ship libraries with crisp .d.ts types that work in both Node 22 ESM and Vite bundlers. • ✓ Compile-Time Type Assertions (expectTypeOf / @ts-expect-error) — Purpose: Write unit tests for your complex generic types so refactors never widen types to any. • ✓ Type-Aware ESLint Rules (no-floating-promises, no-misused-promises) in CI — Purpose: Catch unawaited async bugs automatically before code review even begins.

14.1 TS Project References (composite: true) & Turborepo/pnpm Workspaces

[65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Turborepo/pnpm Workspaces • Integration with Modern moduleResolution • Live Debugging, Telemetry & Technical Interview Trade • What Is TS Project References (composite • In TypeScript Training, TS Project References (composite • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply TS Project References (composite: true) & Turborepo/pnpm Workspaces in real TypeScript systems — specifically: Typecheck 10-package full-stack mon...

What Topic Is Next → Next Topic 14.2: Modern moduleResolution: \"NodeNext\" vs \"Bundler\" & package.json exports

14.2 Modern moduleResolution: \"NodeNext\" vs \"Bundler\" & package.json exports

[65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of package.json exports • Integration with Compile • Live Debugging, Telemetry & Technical Interview Trade • What Is Modern moduleResolution • In TypeScript Training, Modern moduleResolution • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply Modern moduleResolution: \"NodeNext\" vs \"Bundler\" & package.json exports in real TypeScript systems — specifically: Ship libraries with crisp ....

What Topic Is Next → Next Topic 14.3: Compile-Time Type Assertions (expectTypeOf / @ts-expect-error)

14.3 Compile-Time Type Assertions (expectTypeOf / @ts-expect-error)

[65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of Edge • Integration with Type • Live Debugging, Telemetry & Technical Interview Trade • What Is Compile • In TypeScript Training, Compile • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply Compile-Time Type Assertions (expectTypeOf / @ts-expect-error) in real TypeScript systems — specifically: Write unit tests for your complex generi...

What Topic Is Next → Next Topic 14.4: Type-Aware ESLint Rules (no-floating-promises, no-misused-promises) in CI

14.4 Type-Aware ESLint Rules (no-floating-promises, no-misused-promises) in CI

[65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Plain • Production Implementation of no • Integration with TS Project References (composite • Live Debugging, Telemetry & Technical Interview Trade • What Is Type • In TypeScript Training, Type • Even if you come from a non • Runnable Code Implementation (TYPESCRIPT)

Purpose: To understand and apply Type-Aware ESLint Rules (no-floating-promises, no-misused-promises) in CI in real TypeScript systems — specifically: Catch unawaited async bugs au...

What Topic Is Next → Module 14 Capstone (Published Dual ESM+CJS TypeScript Package with Vitest Type-Tests) → Next Topic 15.1: Bridging Compile-Time & Runtime Safety: Schema Inference with Zod / Valibot

Module 14 Hands-On Capstone Project (100 Marks): Published Dual ESM+CJS TypeScript Package with Vitest Type-Tests — Design, implement, test, and document 'Published Dual ESM+CJS TypeScript Package with Vitest Type-Tests' applying every production concept from Module 14 (Monorepo Project References, Dual E...

MODULE 15 OF 16 • STAGE 4 • 14 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Runtime Boundary Validation (Zod), Branded Types & Type-Safe SDK Design

Self-explanatory list-and-bullet production mastery module covering Runtime Boundary Validation (Zod), Branded Types & Type-Safe SDK Design in TypeScript Training, paired with hands-on labs, capstone repository, Topic Mock Tests, and Spoken MockAttempt AI Viva.

Objectives: ✓ Bridging Compile-Time & Runtime Safety: Schema Inference with Zod / Valibot — Purpose: Guarantee untrusted JSON from APIs, env vars, and forms matches your static TypeScript types at runtime. • ✓ Branded / Nominal Types (UserId vs OrderId) & Exhaustive Discriminated Unions (never) — Purpose: Make mixing up entity IDs or forgetting an error state a compile-time impossibility. • ✓ Template Literal Types, satisfies Operator & Zero-any Generic Builder Patterns — Purpose: Preserve exact literal autocomplete while validating configuration contracts cleanly. • ✓ Build, verify, and document 'End-to-End Type-Safe API Client SDK with Branded IDs, Zod Schemas & Result Unions' with clean Git commits and edge-case protection. • ✓ Clear 100% of Topic Timed Mock Tests and Spoken Voice + MockAttempt AI Technical Interviews.

15.1 Bridging Compile-Time & Runtime Safety: Schema Inference with Zod / Valibot [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Bridging Compile • Performance Optimization, Complexity & Latency Impact of Bridging Compile • Live Debugging, Stack • Runnable Code Implementation (TYPESCRIPT) • Hands • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for Bridging Compile-Time • Production Hardening & Security: strictNullChecks, noUncheckedIndexedAccess & unknown boundary parsing

Purpose: To understand and apply Bridging Compile-Time & Runtime Safety: Schema Inference with Zod / Valibot in real TypeScript systems — specifically: Guarantee untrusted JSON fr...

What Topic Is Next → Next Topic 15.2: Branded / Nominal Types (UserId vs OrderId) & Exhaustive Discriminated Unions (never)

15.2 Branded / Nominal Types (UserId vs OrderId) & Exhaustive Discriminated Unions (never) [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Branded / Nominal Types (UserId vs OrderId) & Exhaustive Discriminated Uni... • Performance Optimization, Complexity & Latency Impact of Branded / Nominal Types (UserId vs OrderId) & Exhaustive D... • Live Debugging, Stack • Runnable Code Implementation (TYPESCRIPT) • Hands • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for Branded / Nominal Types (UserId vs OrderId) • Production Hardening & Security: strictNullChecks, noUncheckedIndexedAccess & unknown boundary parsing

Purpose: To understand and apply Branded / Nominal Types (UserId vs OrderId) & Exhaustive Discriminated Unions (never) in real TypeScript systems — specifically: Make mixing up en...

What Topic Is Next → Next Topic 15.3: Template Literal Types, satisfies Operator & Zero-any Generic Builder Patterns

15.3 Template Literal Types, satisfies Operator & Zero-any Generic Builder Patterns [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Template Literal Types, satisfies Operator & Zero • Performance Optimization, Complexity & Latency Impact of Template Literal Types, satisfies Operator & Zero • Live Debugging, Stack • Runnable Code Implementation (TYPESCRIPT) • Hands • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for Template Literal Types • Production Hardening & Security: strictNullChecks, noUncheckedIndexedAccess & unknown boundary parsing

Purpose: To understand and apply Template Literal Types, satisfies Operator & Zero-any Generic Builder Patterns in real TypeScript systems — specifically: Preserve exact literal a...

What Topic Is Next → Module 15 Capstone (End-to-End Type-Safe API Client SDK with Branded IDs, Zod Schemas & Result Unions) → Next Topic 16.1: Production tsconfig Hardening: strict, noUncheckedIndexedAccess & exactOptionalPropertyTypes

Module 15 Hands-On Capstone Project (100 Marks): End-to-End Type-Safe API Client SDK with Branded IDs, Zod Schemas & Result Unions — Architect, implement, test, and document 'End-to-End Type-Safe API Client SDK with Branded IDs, Zod Schemas & Result Unions' applying every bullet-wise engineering concept from Module 15 (Ru...

MODULE 16 OF 16 • STAGE 4 • 14 LIVE + LAB HOURS • 3 TOPICS (24 SUBTOPICS) • 1 CAPSTONE PROJECT (100 MARKS)

Monorepo tsconfig Project References, Type Performance & TS Interviews

Self-explanatory list-and-bullet production mastery module covering Monorepo tsconfig Project References, Type Performance & TS Interviews in TypeScript Training, paired with hands-on labs, capstone repository, Topic Mock Tests, and Spoken MockAttempt AI Viva.

Objectives: ✓ Production tsconfig Hardening: strict, noUncheckedIndexedAccess & exactOptionalPropertyTypes — Purpose: Eliminate undefined array-index crashes and subtle optional-property bugs across teams. • ✓ ESM/CJS Module Resolution, Declaration Maps & Project References Build Speed — Purpose: Publish dual-compatible libraries and keep IDE type-checking instant on large repositories. • ✓ Advanced Type-Level Challenges (Infer, Mapped Modifiers) & Senior TS Interviews — Purpose: Solve real-world generic typing puzzles and architecture questions without type assertions. • ✓ Build, verify, and document 'Strict Multi-Package TypeScript Workspace (`noUncheckedIndexedAccess` + Vitest Type Tests)' with clean Git commits and edge-case protection. • ✓ Clear 100% of Topic Timed Mock Tests and Spoken Voice + MockAttempt AI Technical Interviews.

16.1 Production tsconfig Hardening: strict, noUncheckedIndexedAccess & exactOptionalPropertyTypes [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Production tsconfig Hardening • Performance Optimization, Complexity & Latency Impact of Production tsconfig Hardening • Live Debugging, Stack • Runnable Code Implementation (TYPESCRIPT) • Hands • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for Production tsconfig Hardening • Production Hardening & Security: strictNullChecks, noUncheckedIndexedAccess & unknown boundary parsing

Purpose: To understand and apply Production tsconfig Hardening: strict, noUncheckedIndexedAccess & exactOptionalPropertyTypes in real TypeScript systems — specifically: Eliminate...

What Topic Is Next → Next Topic 16.2: ESM/CJS Module Resolution, Declaration Maps & Project References Build Speed

16.2 ESM/CJS Module Resolution, Declaration Maps & Project References Build Speed [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for ESM/CJS Module Resolution, Declaration Maps & Project References Build Spe... • Performance Optimization, Complexity & Latency Impact of ESM/CJS Module Resolution, Declaration Maps & Project Refe... • Live Debugging, Stack • Runnable Code Implementation (TYPESCRIPT) • Hands • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for ESM/CJS Module Resolution • Production Hardening & Security: strictNullChecks, noUncheckedIndexedAccess & unknown boundary parsing

Purpose: To understand and apply ESM/CJS Module Resolution, Declaration Maps & Project References Build Speed in real TypeScript systems — specifically: Publish dual-compatible li...

What Topic Is Next → Next Topic 16.3: Advanced Type-Level Challenges (Infer, Mapped Modifiers) & Senior TS Interviews

16.3 Advanced Type-Level Challenges (Infer, Mapped Modifiers) & Senior TS Interviews [65 mins • Assignment (100M) • Hands-On Lab (50M) • 2 Mock Sets • 2 AI Vivas]

Subtopics Covered: Security Hygiene & Data Safety Rules for Advanced Type • Performance Optimization, Complexity & Latency Impact of Advanced Type • Live Debugging, Stack • Runnable Code Implementation (TYPESCRIPT) • Hands • Idiomatic TSCR Implementation: discriminated unions, generic constraints, mapped types & Zod runtime schemas • TypeScript 5 Type Checker & ECMAScript Runtime Narrowing Execution & Memory Lifecycle for Advanced Type-Level Challenges (Infer, Mapped Modifi... • Production Hardening & Security: strictNullChecks, noUncheckedIndexedAccess & unknown boundary parsing

Purpose: To understand and apply Advanced Type-Level Challenges (Infer, Mapped Modifiers) & Senior TS Interviews in real TypeScript systems — specifically: Solve real-world generi...

What Topic Is Next → Module 16 Capstone → End-of-Course Live Client Project: TypeSafe SDK, End-to-End Typed API Contract Library & Event Bus Package

Module 16 Hands-On Capstone Project (100 Marks): Strict Multi-Package TypeScript Workspace (`noUncheckedIndexedAccess` + Vitest Type Tests) — Architect, implement, test, and document 'Strict Multi-Package TypeScript Workspace (`noUncheckedIndexedAccess` + Vitest Type Tests)' applying every bullet-wise engineering concept from Modu...

3. LIVE CLIENT PRODUCTION PROJECT & CAREER SCOPE

End-of-Course Live Client Production Project + Official Internship Confirmation Letter

Project: TypeSafe SDK, End-to-End Typed API Contract Library & Event Bus Package • **Confirmed Role:** TypeScript 5 Systems & Full-Stack Type Architecture Intern

Domain & Production Infra: Developer Platform & Multi-Team SaaS Engineering Client (TypeScript 5 Strict Mode + Zod Runtime Schemas +ESM/CJS Build + Vitest CI Pipeline)

- Engineered zero-any TypeScript 5 modules using generics, discriminated unions, mapped types, and template literals.
- Paired compile-time TypeScript inference with Zod runtime validation schemas for bulletproof API boundaries.
- Built type-safe event emitters, result/error monads, and dual ESM/CJS library bundles with declaration maps.
- Enforced strict compiler flags (noUncheckedIndexedAccess, exactOptionalPropertyTypes) with 100% Vitest type tests.

Target Job Roles: TypeScript Software Engineer, Frontend Engineer (React/Angular + TS), Full-Stack TypeScript Developer, SDK & Platform UI Engineer

Realistic Annual CTC Scope: **Fresher / Entry-Level TS Developer (0–1 Yr):** ₹5.2 LPA – ₹9.8 LPA **Mid-Level Full-Stack TypeScript Engineer (1–3 Yrs):** ₹10.5 LPA – ₹21.0 LPA **Staff Frontend / Full-Stack TypeScript Architect (3–6+ Yrs):** ₹22.0 LPA – ₹42.0+ LPA

MockAttempt Academy Pvt. Ltd. • Email: support@mockattempt.com • Phone/WhatsApp: +91 95400 07917 • Web: https://mockattempt.com/courses/typescript-training

Try 2 Free Topic-Wise Timed Topic Mock Tests & Spoken MockAttempt AI Interviews daily on every topic above before enrolling.