TypeScript's Type System and Its Relation to JavaScript's Dynamic Typing

PROMPTS

Advanced Programming Project

Francesco Polperio 635406

January 2026

Contents

Chapter 1

Task 3

1.1 Methodological Reflection: A Strategic Multi-Model Approach

To ensure the highest level of technical accuracy and clarity for this report, I adopted a heterogeneous multi-model approach. Rather than relying on a single AI tool or a linear interaction, I deliberately chose to utilize Artificial Intelligence as a structured supporting instrument, preceded by a brief exploratory analysis to identify the specific strengths of different Large Language Models (LLMs).

1.1.1 Superficial Study (Orientation)

Before prompting any model, I performed a quick preliminary study to orient myself:

- I reviewed the material already available to me (course notes, project requirements).
- I performed a short online exploration to identify the correct directions and vocabulary (e.g., compile-time vs runtime, type erasure, soundness trade-offs, gradual typing, structural typing).

The purpose was to avoid starting from an AI-generated outline and instead define, in advance, what the report *must* explain and which examples would best demonstrate it.

1.1.2 Online Tool Fit (Choosing the Right Model for Each Job)

After the initial orientation, I searched online to understand which models tend to perform better at which tasks (prompt refactoring vs technical synthesis). This reflection led to a deliberate separation of roles to maximize the quality of the output:

Gemini 3 Pro (Google) as the "Prompt Architect" Proved particularly effective for textual reasoning and nuance. It was selected to refine the learning path and generate optimized prompts (Prompt Engineering) aimed at eliciting deeper and more focused explanations.

ChatGPT 5.2 (OpenAI) as the "Execution Engine" Proved highly effective for technical synthesis and code generation. It was used to produce the final TypeScript examples and structured educational responses based on the strict guidelines provided by the Gemini-generated prompts.

1.1.3 Prompt Refinement Loop (Rough → Refined → Executed)

The operational loop was consistent across topics:

1. I wrote a **rough request** (goal-driven, informal, sometimes ambiguous).
2. Gemini returned a **refined prompt** (structured output, explicit constraints, "what to include" and "what to avoid").
3. I executed the refined prompt in ChatGPT and extracted only the parts relevant to the report sections.

The Human-in-the-Loop Orchestration It is crucial to emphasize that this workflow was neither automated nor uncontrolled. My role was that of the orchestrator and validator. I acted as the "semantic glue" between the two models, ensuring that: the direction suggested by Gemini aligned with the project's specific requirements and the prompts were manually reviewed and adjusted to fit my understanding of the subject.

This approach demonstrates the ability to combine multiple AI models cooperatively, leveraging their respective strengths to achieve a common goal. Rather than replacing human reasoning, this process highlights the role of the human as the coordinating element—the bridge between raw AI capabilities and meaningful, verified academic results.

1.2 Prompt Documentation (Key Prompts Only)

Below are the refined prompts generated by the "Architect" (Gemini) and fed into the "Engine" (ChatGPT) to produce the content of Task 1 and Task 2.

1.2.1 Gemini Architect

Goal

I wanted a systematic way to convert informal questions into strict prompts that force depth: edge cases, runtime boundaries, and design trade-offs (not generic definitions).

Prompt

Act as a Senior Technical Consultant and Prompt Architect. Your objective is to assist me in drafting a university-level report on TypeScript's type system. For each technical topic I provide (e.g., Structural Typing, Generics, Unsoundness), do not explain it directly to me. Instead, formulate a precise, multi-step prompt that I can feed into ChatGPT to obtain a structured explanation, contrasting examples, and specific code scenarios. The generated prompts must force the downstream AI to focus on edge cases, architectural design, and runtime behavior analysis rather than generic definitions.

1.2.2 TypeScript VS JavaScript

Goal

I needed the conceptual backbone: TypeScript adds compile-time analysis without changing JavaScript's runtime semantics.

Prompt

I am working on a project titled "TypeScript's Type System and Its Relation to JavaScript's Dynamic Typing." Please provide a comprehensive technical explanation covering the following: Dynamic vs. Static Typing: Explain the fundamental differences between JavaScript's dynamic runtime environment and TypeScript's static analysis.

1.2.3 Type coercion

Goal

I wanted coercion treated as spec-defined behavior and connected to silent failures (NaN/0) and latent bugs.

Prompt

Could you explain exactly what happens under the hood when JavaScript coerces values? Please cover: The Mechanism: Why does JavaScript convert types automatically (e.g., null becoming 0, or "hello" becoming NaN in math operations) instead of throwing an error like other languages? Silent Failures: Explain the concept of "latent bugs" caused by coercion. Why is receiving NaN or 0 often worse than an immediate program crash? Examples: Illustrate this with common scenarios (like the double(x) function) where implicit conversion hides logic errors.

1.2.4 Soundness

Goal

I wanted the theory link (progress/preservation) and then concrete unsoundness examples (array indexing, optional properties under non-strict settings, bivariance).

Prompt

I want to deepen my analysis of Type Soundness within TypeScript. I understand that while academic languages (like Haskell) aim for perfect soundness, TypeScript makes different trade-offs. Please explain the following: 1. The Definition of Soundness Define "Type Soundness" in the context of Programming Language Theory (e.g., "Well-typed programs cannot go wrong"). 2. Intentional Unsoundness in TypeScript Explain why TypeScript is intentionally unsound in certain scenarios. How does this relate to preserving JavaScript's expressiveness and usability? Discuss the concept of being "Pragmatic" vs. "Correct." 3. Concrete Examples of Unsoundness Provide code examples where TypeScript's static analysis passes (no red squiggles), but the code creates a type error at runtime. Suggestion: Please include examples related to Array Indexing (e.g., accessing an out-of-bounds index) or Function Parameter Bivariance.

1.2.5 Pragmatism

Goal

I wanted an engineering justification for TypeScript's design choices: why it prioritizes usability, ecosystem compatibility, and incremental adoption over a perfectly sound (theoretically correct) type system.

Prompt

I would like to discuss the design philosophy of TypeScript, specifically the concept of "Pragmatism." Please explain the tension between Pragmatism and Theoretical Correctness in TypeScript's design. 1. The Philosophy: Explain why the TypeScript team prioritizes developer productivity and usability over achieving a mathematically perfect (100% sound) type system. Why would a fully "correct" type system be annoying or unusable for a JavaScript developer?

1.2.6 Type erasure

Goal

I wanted to make explicit the core boundary of the whole report: TypeScript types exist only at compile time and are erased, so the runtime is still pure JavaScript. This explains why TypeScript cannot enforce contracts at runtime and why certain mismatches are possible.

Prompt

I want to understand the concept of Type Erasure in TypeScript. I know that types are removed during compilation, but I want to understand the implications of this mechanism. Please explain: 1. The Mechanism (Before & After): Show me a code snippet with a complex TypeScript Interface and a Function using generics. 2. Show me the exact JavaScript output after compilation. Key Question: Demonstrate visually how all the Type Syntax disappears, leaving only the Value Syntax.

1.2.7 Gradual typing

Goal

Describe gradual typing as a dial (not on/off), showing how any weakens checks and spreads unsafety, why unknown is safer, and how this supports step-by-step migration from JavaScript to TypeScript.

Prompt

I want to focus specifically on "Gradual Typing" in TypeScript. I understand this is the feature that allows static and dynamic code to coexist, but I need a deeper technical breakdown. Please explain: 1. The Theory: The "Dial" Metaphor, explain how Gradual Typing acts as a "dial" or slider rather than

a simple on/off switch. How does the compiler handle a file that contains both fully typed variables (x: number) and dynamic ones (y: any)? Does it simply ignore the dynamic parts? 2. The Mechanism: any as the Bridge, analyze the technical role of the any type. Explain how any creates a boundary where the static type checker "gives up" control. Code Example: Show how a variable typed as any can "infect" other parts of the code, propagating unsafe behavior (the contagion effect). 3. Practical Application: Brownfield Migration, how does Gradual Typing enable the migration of a massive legacy JavaScript codebase to TypeScript? Explain the strategy of starting with allowJs: true and loose settings, then incrementally enabling stricter flags (like noImplicitAny). 4. any vs. unknown (Modern Gradual Typing) A Senior Engineer would warn against overusing any. Explain why unknown is the safer, modern alternative for gradual typing. Show a snippet contrasting an unsafe operation with any vs. the same operation blocked by unknown.

1.2.8 Structural typing

Goal

Explain that TypeScript checks type compatibility by an object's shape (its properties) rather than by its name, show how different types with the same structure are accepted, and briefly contrast this with nominal typing (Java/C#).

Prompt

Can you explain the concept of Structural Typing in TypeScript? Please contrast it with Nominal Typing (like in Java or C#) and provide a code example showing how two different interfaces with the same shape are considered compatible by the compiler.

1.2.9 Duck typing

Goal

Explain what "duck typing" means in practice, and how TypeScript relates to it: TypeScript checks the "duck-like shape" at compile time (structural typing), while at runtime it's still JavaScript and missing properties can still crash.

Prompt

How does TypeScript's structural type system relate to the concept of "Duck Typing"? Please illustrate with a code example where a function accepts an object simply because it matches the required structure, even if the object doesn't explicitly implement the requested interface.

1.2.10 Union type

Goal

Explain how union types let one value have multiple possible types, and show how TypeScript forces you to narrow the type before using it—then introduce discriminated unions as the clean way to model real states (like loading/success/error) safely.

Prompt

I would like to learn about Union Types in TypeScript using a step-by-step approach: 1. First, give me a general definition of what a Union Type is (|) and why it is useful when a value can be more than one shape. 2. Then, drill down into the specific pattern of "Discriminated Unions" (or "Tagged Unions"). Please provide a code example of a Network Request state (Loading, Success, Error) to demonstrate how we use a shared literal property (like status) to safely narrow down the specific type.

1.2.11 Type inference

Goal

Explain how TypeScript automatically figures out types for variables and return values, what "best common type" means for arrays, and when it's fine to rely on inference versus when I should write explicit types (especially at public boundaries).

Prompt

Explain Type Inference in TypeScript. I've noticed I don't always have to write generic annotations like : string or : number. How does TypeScript automatically infer types for variables and function return values? What is the concept of "Best Common Type" when inferring from an array of mixed values? Provide examples of when it's safe to rely on inference versus when it's better to be explicit.

1.2.12 Generics

Goal

Explain why using any loses type information, and how generics (<T>) let a function keep the relationship between input and output types.

Prompt

I need to finalize my understanding of TypeScript for a report. Please explain Generics using the classic "Identity Function" example. 1. The Problem: Show me a function that accepts any type of argument and returns it. Show why using any is bad here (because we lose the type information of the return value). 2. The Solution: Refactor that function using Generics (<T>). Explain simply how (<T>) acts like a "variable for types" to capture and preserve the input type. 3. Real World: Briefly explain how standard Arrays use generics (e.g., Array<number>) to prevent us from mixing types accidentally.

1.2.13 Task 2

Goal

Provide two minimal examples: one caught at compile time and one that compiles but fails/mismatches at runtime.

Prompt

I am moving to Task 2. I need to provide short, clear TypeScript code examples for two specific scenarios. 1) errors caught by the type checker. 2) mismatches between static types and JavaScript runtime behaviour.