

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

VERIFICATION

Advanced Programming Project

Francesco Polperio 635406

January 2026

Chapter 1

Task 3

1.1 Validation and Verification Process

Since Generative AI models can occasionally produce hallucinations (plausible but incorrect information), a strict validation protocol was applied to all generated content before including it in this report. No AI output was accepted blindly.

Authoritative Sources Used (the “bibles”)

- **TypeScript Handbook and TSCConfig Reference (official docs).** Used to verify: structural typing, excess property checks, type assertions (no runtime impact), discriminated unions/narrowing, strictness flags (e.g., `strictNullChecks`, `noImplicitAny`), and compiler behavior (e.g., emitting JS with or without `noEmitOnError`).
 - Handbook (intro): typescriptlang.org/docs/handbook/intro.html
 - “TypeScript for the New Programmer” (runtime behavior, erased types): typescriptlang.org/docs/handbook/typescript-for-the-new-programmer.html
 - Type Compatibility (structural typing + note on soundness): typescriptlang.org/docs/handbook/type-compatibility.html
 - Object Types (Excess Property Checks): typescriptlang.org/docs/handbook/2/objects.html
 - Type Assertions (no runtime checking): typescriptlang.org/docs/handbook/2/everyday-types.html
 - TSCConfig options: `noImplicitAny`, `strictNullChecks` (TSCConfig reference), `noUncheckedIndexedAccess`, `noEmitOnError`, `erasableSyntaxOnly`.
- **ECMAScript Language Specification (ECMA-262, 16th edition).** Used to verify: coercion algorithms and numeric conversion rules (e.g., `ToNumber(null) = +0`, `ToNumber(undefined) = NaN`), and the fact that these behaviors are defined by deterministic abstract operations.

1.1.1 Textual Validation

Each explanatory paragraph in Task 1–2 was validated statement-by-statement against the authoritative sources above

Rectified Inaccuracies (Hallucinations check)

During the cross-referencing phase with the official specifications, the following statement was identified as technically imprecise and subsequently corrected.

- **Original Draft Statement:**
“Errors arise only when an invalid operation is executed.”
- **Corrected Statement:**
ECMAScript defines both (1) **syntax/early errors** that are detected before evaluation, and (2) **runtime errors** that arise from the runtime semantics of executed operations. Type-related failures are primarily runtime behaviors, but not all errors are runtime errors.

- **Verification & Source:**

Ref: ECMA-262 (16th Ed.), Sections 13.5.1.1 (Static Semantics: Early Errors) vs. 13.5.1.2 (Runtime Semantics).

The specification explicitly defines “Early Errors” (such as specific strict mode violations like `delete identifier`) which are thrown during the parsing/analysis phase. This contradicts the claim that errors *only* arise during the execution phase.

- **Original Draft Statement:**

`ToNumber(null) → 0`

- **Corrected Statement:**

`ToNumber(null) → +0`

- **Verification & Source:**

Ref: ECMA-262 (16th Ed.), Section 7.1.4 (ToNumber Abstract Operation).

The specification explicitly defines that if the argument is `null`, the return value is `+0` (specifically denoted as `+0F` in the spec). While often behaving like 0, the sign is preserved in the definition to comply with IEEE 754 floating-point standards.

- **Original Draft Statement:**

“JavaScript coerces values because its specification mandates automatic conversions to preserve backward compatibility and fault tolerance...”

- **Corrected Statement:**

JavaScript performs implicit conversions because the ECMAScript specification defines abstract conversion operations (like `ToNumber`, `ToString`) and uses them in the runtime semantics of many constructs. The spec also notes that some legacy implicit numeric conversions are maintained for backward compatibility.

- **Verification & Source:**

Ref: ECMA-262 (16th Ed.), Section 6.1.6 (Numeric Types), Note paragraph.

The specification explicitly states regarding implicit numeric conversions: “These legacy implicit conversions are maintained for **backward compatibility**.” The term “fault tolerance” is not used in the specification as a rationale for automatic type conversion.

- **Original Draft Statement:**

Heading: “Function Contracts Are Enforced” / Terminology: “semantic contracts”

- **Corrected Statement:**

Heading: **Function Contracts Are Statically Checked**

Text: “This introduces explicit type-level contracts checked by the TypeScript type checker; JavaScript has no built-in static type checking.”

- **Verification & Source:**

Ref: TypeScript Design Goals (Non-Goal #8: “Add specific runtime type checks”) & ECMA-262.

TypeScript operates via **Type Erasure**: types are removed during compilation and do not affect the runtime behavior defined by ECMA-262. “Enforced” implies runtime validation, which does not exist in standard TypeScript; the correct term is “statically checked” or “validated at compile-time.”

- **Original Draft Statement:**

“TypeScript accepts this (No compile error);”

- **Corrected Statement:**

By default, TypeScript allows this operation (intentional unsoundness for ergonomic reasons), but with the configuration flag `noUncheckedIndexedAccess` enabled, `xs[10]` is inferred as `number | undefined`. Consequently, assigning it to a variable of type `number` triggers a compile-time error.

- **Verification & Source:**

Ref: TypeScript TSCConfig Reference (Option: `noUncheckedIndexedAccess`).

The original statement was too absolute. While the default behavior (even in `strict` mode) ignores potential out-of-bounds access, the compiler explicitly supports a flag to handle this case safely. The report was updated to reflect this configurability.

- **Original Draft Statement:**

“A ‘correct’ (sound) type system aims to prove progress and preservation also enforce strict subtyping rules.”

- **Corrected Statement:**

“In formal presentations of typed languages, soundness is often shown via progress and preservation for a specified operational semantics.”

- **Verification & Source:**

Ref: Standard Type Theory Definitions (Wright & Felleisen).

The original statement conflated the definition of *soundness* with the specific features of object-oriented type systems. Soundness (defined via Progress and Preservation theorems) guarantees that well-typed programs do not enter undefined states. It does not strictly require the presence of subtyping; a language can be sound with no subtyping at all.

- **Original Draft Statement:**

“You can think of the type system as a dial, this is primarily governed by the `noImplicitAny` flag.”

- **Corrected Statement:**

“You can think of strictness as a dial controlled by the `strict` family of compiler options; `noImplicitAny` is one important knob that specifically flags implicit `any`, but it is not the only factor.”

- **Verification & Source:**

Ref: TypeScript TSConfig Reference (Option: strict vs. noImplicitAny).

The `strict` flag is a meta-option that enables a wide range of type checking rules (including `strictNullChecks`, `strictFunctionTypes`, etc.). Describing the strictness level as being “governed” primarily by `noImplicitAny` is an oversimplification that neglects other critical safety checks included in the `strict` family.

- **Original Draft Statement:**

“So, TypeScript performs duck typing checks at compile time...”

- **Corrected Statement:**

“TypeScript performs **structural type compatibility** checks at compile time (often compared informally to ‘duck typing’), while runtime behavior is plain JavaScript.”

- **Verification & Source:**

Ref: TypeScript Handbook (Section: Type Compatibility).

The Handbook explicitly defines TypeScript’s system as **Structural Typing** (static analysis based on shape). The term “Duck Typing” is historically associated with dynamic (runtime) checking in languages like Python or Ruby. Using the correct terminology distinguishes the compile-time nature of TypeScript from dynamic runtime behaviors.

- **Original Draft Statement:**

“This works because all return paths produce the same type and the type is obvious and stable.”

- **Corrected Statement:**

“In this example, the return type is inferred as `number`. In general, TypeScript infers a function’s return type from all its return statements; when different return paths produce different types, the resulting inferred type is a **union** (e.g., `string | number`).”

- **Verification & Source:**

Ref: TypeScript Handbook (Section: Type Inference, “Best common type”).

The original statement incorrectly implied that inference relies on the uniformity of return values. The Handbook clarifies that when multiple expressions are involved (like different return statements), TypeScript calculates a “best common type” or defaults to a Union Type if no single supertype encompasses all candidates.

- **Original Draft Statement:**

Code comment: `result.toFixed(2); // runtime error, but TypeScript allows it`

- **Corrected Statement:**

Code comment: `result.toFixed(2); // TypeScript allows it because result is any; at runtime it may throw (e.g., if result is a string).`

- **Verification & Source:**

*Ref: TypeScript Handbook (Section: The **any** Type).*

The original comment definitively stated a runtime error would occur. However, the **any** type simply creates an intersection of all possible types (effectively disabling type checking). The operation is statically valid; a runtime error occurs *only if* the actual runtime value does not have a `toFixed` method. The statement was adjusted to reflect this potential, rather than guaranteed, failure.

- **Original Draft Statement:**

“the array remembers this forever.”

- **Corrected Statement:**

“the array’s element type is tracked by the type system; subsequent operations are statically checked against the definition `Array<number>` during compilation.”

- **Verification & Source:**

Ref: TypeScript Handbook (Section: Erasure).

The original phrasing implied that the array retains type information indefinitely, possibly even at runtime. Due to **Type Erasure**, the runtime object is a standard JavaScript array with no “memory” of its static constraints. The correction clarifies that this is purely a compile-time tracking mechanism.

1.1.2 Code Execution (“Playground” Test)

All TypeScript code snippets included in Task 1 and Task 2 were executed in the official **TypeScript Playground** (version as shown in the Playground at the time of testing: v5.9.3).

- **Compile-time verification:** every snippet marked as “Error” was checked to confirm the compiler emits an error in the editor.
- **Runtime verification:** for mismatch examples, the compiled JavaScript was executed in the browser console to confirm the described runtime behavior (e.g., out-of-bounds array access yields `undefined` in JavaScript).

Code Snippet Validation Log (Task 1 + Task 2)

All snippets below were executed and matched the described behavior. **Legend:** **VALIDATED / CORRECT** means “compiled and/or ran exactly as described in the report”.

Task 1 — JavaScript snippets

1. **JS-01 (dynamic reassignment)** — `let x = 42; x = "hello"; x = {a:1};` **VALIDATED / CORRECT**
2. **JS-02 (coercion in multiplication)** — `double("hello"); double(null); double(undefined);` **VALIDATED / CORRECT**
3. **JS-03 (object shape mutation)** — `user.name; user.age; delete user.age;` **VALIDATED / CORRECT**

Task 1 — TypeScript snippets (static typing, erasure, soundness, gradual typing, etc.)

1. **TS-01 (type annotation prevents reassignment)** — `let x: number = 42; x = "hello";` **VALIDATED / CORRECT**
2. **TS-02 (inferred type stability)** — `let count = 10; count = "twenty";` **VALIDATED / CORRECT**
3. **TS-03 (function contract)** — `function greet(name: string): string; greet(42);` **VALIDATED / CORRECT**
4. **TS-04 (object shape check)** — `type User={name:string;age:number}; u.email = ...` **VALIDATED / CORRECT**

5. **TS-05 (type erasure example)** — `type ID=number; const id:ID=5;` **VALIDATED / CORRECT**
6. **TS-06 (unsound array indexing)** — `const xs:number[]=[1,2,3]; const x:number = xs[10];` **VALIDATED / CORRECT**
7. **TS-07 (optional property mismatch; depends on flags)** — `age?:number; user.age.toFixed()` **VALIDATED / CORRECT**
8. **TS-08 (function parameter variance example)** — `Handler<Animal>, Handler<Dog>` **VALIDATED / CORRECT** (as tested configuration)
9. **TS-09 (mixed typed/any)** — `let x:number=10; let y:any=getValue();` **VALIDATED / CORRECT**
10. **TS-10 (any escape hatch)** — `let a:any=42; let b:string=a;` **VALIDATED / CORRECT**
11. **TS-11 (any contagion)** — `let data:any=...; let result=data.value; ...` **VALIDATED / CORRECT**
12. **TS-12 (allowJs/checkJs config snippet)** — `{ "allowJs": true, "checkJs": false }` **VALIDATED / CORRECT**
13. **TS-13 (inference to any in untyped JS-style function)** — `function add(a,b){return a+b}` **VALIDATED / CORRECT**
14. **TS-14 (tighten strict flags snippet)** — `noImplicitAny/strictNullChecks/strictFunctionTypes` **VALIDATED / CORRECT**
15. **TS-15 (unknown blocks unsafe ops)** — `let x:unknown="hello"; x.toFixed();` **VALIDATED / CORRECT**
16. **TS-16 (unknown narrowing)** — `if(typeof x==="string") x.toUpperCase();` **VALIDATED / CORRECT**
17. **TS-17 (structural typing User/Account)** — `assignment + function call` **VALIDATED / CORRECT**
18. **TS-18 (Admin subtype to User + excess property note)** **VALIDATED / CORRECT**
19. **TS-19 (JS integration fetch example typed as User)** **VALIDATED / CORRECT**
20. **TS-20 (accidental compatibility Point2D/ScreenPosition)** **VALIDATED / CORRECT**
21. **TS-21 (union + narrowing via typeof)** — `formatId(id: string|number)` **VALIDATED / CORRECT**
22. **TS-22 (discriminated union definitions)** — `Loading/Success/ErrorState` **VALIDATED / CORRECT**
23. **TS-23 (render() narrowing on status)** **VALIDATED / CORRECT**
24. **TS-24 (BadState optional fields anti-pattern)** **VALIDATED / CORRECT**
25. **TS-25 (inference basic examples)** — `let count=10; const nums=[1,2,3];` **VALIDATED / CORRECT**
26. **TS-26 (return type inference)** — `function add(a:number,b:number){return a+b}` **VALIDATED / CORRECT**
27. **TS-27 (explicitness guidelines mini-snippets)** — `const total=...; function isEmpty(...)` **VALIDATED / CORRECT**
28. **TS-28 (generics problem with any)** — `function identity(value:any){...}` **VALIDATED / CORRECT**
29. **TS-29 (generic identity<T>preserves types)** — `function identity<T>(value:T):T` **VALIDATED / CORRECT**
30. **TS-30 (Array<number>generic)** — `push OK/error` **VALIDATED / CORRECT**

Task 2 — Examples required by the assignment

1. **T2-A (type checker error: wrong arg types)** — `add("1",2)` **VALIDATED / CORRECT**
2. **T2-B (type checker error: missing property)** — `printId({})` **VALIDATED / CORRECT**
3. **T2-C (mismatch: out-of-bounds index)** — `xs[10] -> undefined` **VALIDATED / CORRECT**
4. **T2-D (mismatch: type assertion bypass)** — `{}` as `User` **VALIDATED / CORRECT**

This verification process ensures the report reflects actual language behavior and official documentation.