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

REPORT

Advanced Programming Project

Francesco Polperio 635406

January 2026

Contents

Chapter 1

Task 1

TypeScript and JavaScript do not represent two different execution models. TypeScript adds a static type-checking phase on top of JavaScript's inherently dynamic runtime. Understanding this distinction is crucial: TypeScript does not change JavaScript's semantics—it constrains and analyzes them before execution.

1.1 JavaScript's Dynamic Typing

A language is dynamically typed if:

- Types are associated with values, not variables;
- Type checking occurs at runtime;
- Type errors are discovered during execution.

JavaScript fits this model precisely.

1.1.1 Types Exist Only at Runtime

Variables have no fixed type. The same variable may refer to values of different types over time.

```
let x = 42;        // number
x = "hello";       // string
x = { a: 1 };      // object
```

No static restriction prevents this reassignment.

1.1.2 Type Errors Are Runtime Errors

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 therefore primarily runtime behaviors, but not all errors are runtime errors.

```
function double(x) {
    return x * 2;
}

double("hello");   // NaN (silent failure)
double(null);      // 0
double(undefined); // NaN
```

JavaScript often **coerces** values instead of throwing errors, making many bugs latent rather than explicit.

1.1.3 Dynamic Object Shape (Structural Mutation)

Objects are open and mutable.

```
const user = {};
user.name = "Alice";
user.age = 30;
delete user.age;
```

The "shape" of an object can change freely at runtime. Dynamic typing maximizes flexibility but sacrifices:

- early error detection;
- refactoring safety;
- Static Type Documentation.

In PL terms, JavaScript prioritizes *expressiveness*.

1.1.4 Type Coercion

JavaScript's implicit type coercion is not accidental, nor is it "sloppy execution." It is the result of explicit design decisions encoded in the ECMAScript spec, driven by historical constraints and a particular philosophy: "Do something reasonable instead of failing." When JavaScript evaluates an operation like `x * 2`, it does not say: "If x is not a number, throw an error."

Instead, the ECMAScript spec defines a deterministic conversion pipeline using abstract operations such as:

- `ToPrimitive`
- `ToNumber`
- `ToString`
- `ToBoolean`

These are *formal algorithms*, not heuristics. Automatic coercion exists for: Backward compatibility, Fault tolerance, Low barrier to entry and Expressiveness.

Why `null` Becomes 0

It is explicitly defined:

$$\texttt{ToNumber(null)} \rightarrow +0$$

Null represents "no object" and converting it to `0` preserved compatibility with early web scripts and numeric defaults. Thus:

$$\texttt{null * 2} \rightarrow 0$$

Silent Failures and Latent Bugs

A **silent failure** occurs when an operation is semantically invalid but produces a valid runtime value without signaling an error. Instead a **latent bug** is a bug that is introduced at one point, manifests much later and often far from its source. This is far worse than a crash, because: the program appears to work but the output is silently corrupted.

Key Takeaways: Type Coercion

JavaScript performs implicit conversions because the ECMAScript specification defines abstract conversion operations 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.

1.2 TypeScript's Static Type System

A language is statically typed if:

- Types are associated with program expressions;
- Type checking occurs before execution;
- Programs that violate type rules are rejected.

TypeScript applies static typing through compile time analysis, but it does not enforce types at runtime. TypeScript is not a statically typed runtime language. It is a statically analyzed language that erases to JavaScript.

```
let x: number = 42;
x = "hello"; // compile time error
```

At runtime, this becomes:

```
let x = 42;
x = "hello"; // perfectly valid JavaScript
```

1.2.1 Variables Have Declared or Inferred Types

Types constrain how values may flow through the program.

```
let count = 10;       // inferred as number
count = 20;           // ok
count = "twenty";     // Error
```

1.2.2 Function Contracts Are Statically Checked

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

```
function greet(name: string): string {
  return "Hello" + name;
}

greet(42); // Error
```

This introduces explicit semantic contracts, absent in JavaScript.

1.2.3 Object Shapes Are Statically Checked

TypeScript models object structure using structural typing.

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

const u: User = { name: "Alice", age: 30 };
u.email = "x@example.com"; // Error
```

Even though JavaScript allows this at runtime, TypeScript prevents it statically.

1.2.4 Type Erasure

All types are removed during compilation.

```
type ID = number;
const id: ID = 5;
```

Compiles to:

```
const id = 5;
```

No runtime cost, no runtime enforcement.

TypeScript is a static approximation of JavaScript, a type-theoretic model layered on top of a dynamic language.

TypeScript does not aim for full **soundness**, instead, it provides a pragmatic, incomplete static model of a fundamentally dynamic language.

1.2.5 Soundness

TypeScript is a statically analyzed language that intentionally violates soundness to remain compatible with JavaScript. A type system is sound if it satisfies the property of well-typed programs cannot go wrong. More precisely if a program type-checks successfully, then during execution, it will not produce a type error. This is typically proven via two theorems:

- Progress: a well-typed program is either a value, or can take a computational step
- Preservation:if a program is well-typed, every evaluation step preserves the type.

Together, these ensure no runtime type errors occur in well-typed programs

Examples of Sound Languages: Haskell, OCaml and Rust (with caveats around unsafe).
The critical point is that TypeScript does not control runtime execution. JavaScript executes the program and TypeScript only approximates behavior ahead of time, with all types erased. The key principle in all of this is that TypeScript prioritizes **pragmatism** over theoretical correctness. TypeScript must:

- Accept existing JavaScript patterns;
- Model dynamic behavior;
- Avoid rejecting common idioms.

A sound system would reject large portions of real-world JS. Also, **gradual typing** forces instability, but that will be discussed later.

Example 1: Array Indexing (Classic Unsoundness)

```
const xs: number[] = [1, 2, 3];
const x: number = xs[10];
```

- 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;
- **Runtime value is** `undefined`.

But `x` is statically typed as `number`. This violates soundness:

"A well-typed expression evaluates to a value outside its type."

Example 2: Optional Properties and Structural Typing

Note on compiler settings: This example only compiles when strictNullChecks is disabled. With strictNullChecks: true, the type of user.age is number | undefined, so calling .toFixed() is rejected at compile time. In other words, the mismatch is not "always accepted by TypeScript," but depends on the chosen strictness level.

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

function printAge(user: User) {
  return user.age.toFixed(2);
}
```

- **Runtime error** if `age` is `undefined` (? means optional).

Why can this compile in non-strict configurations? Because when `strictNullChecks` is disabled, TypeScript does not track `undefined` in types. As a result, an optional property like `age?: number` is treated as `number` for the purpose of method calls, even though the property may still be missing at runtime. This is a pragmatic trade-off: the program type-checks, but it can still throw at runtime if `age` is `undefined`.

Example 3: Function Parameter Bivariance

This is explicitly documented unsoundness. This is a design choice that prioritizes usability over strict correctness, though modern configuration flags can mitigate it.

```
type Animal = { name: string };
type Dog = { name: string; bark(): void };
type Handler<T> = (arg: T) => void;

let handleAnimal: Handler<Animal>;
let handleDog: Handler<Dog>;

handleAnimal = handleDog; // allowed
```

```
handleAnimal({ name: "cat" }); // runtime error: bark is missing
```

Why TypeScript Allows This: Because:

- Event handlers in JS rely on this flexibility;
- Strict contravariance would break DOM APIs and callback-heavy code;
- TS chooses **bivariance** for function parameters in many cases.

This violates the *substitution principle*, but enables real-world usage.

Key Takeaways: Soundness TypeScript deliberately sacrifices type soundness—allowing well-typed programs to fail at runtime—in order to preserve JavaScript's expressiveness, support gradual typing, and remain practical for real-world development, choosing pragmatic usefulness over formal correctness.

1.2.6 Pragmatism VS Theoretical Correctness

TypeScript is best understood not as a "typed JavaScript," but as a socio-technical compromise between: the realities of the JavaScript ecosystem, and the ideals of static type theory.

In TypeScript, pragmatism is a deliberate design stance, not a lack of rigor, it means: optimizing for developer velocity, supporting existing JavaScript idioms, enabling incremental adoption and accepting imperfect guarantees in exchange for usability

In PL theory, soundness is often shown via progress and preservation for a specified operational semantics. The correct type system disallows the possibility to have: unchecked nulls, out-of-bounds access, unsafe variance, implicit coercions and provide formal semantics.

TypeScript intentionally sacrifices theoretical soundness in favor of pragmatism, because a fully correct type system would reject common JavaScript patterns, slow development, overwhelm developers with annotations, and ultimately fail to be adopted in the real world.

1.3 Summary: JavaScript Vs Typescript

Dimension	JavaScript	TypeScript
Typing Time	Runtime	compile time
What Has Types	Values	Expressions
Type Errors	Runtime failures	compile time rejections
Object Shape	Mutable	Statically constrained
Type Coercion	Implicit, pervasive	Flagged statically
Runtime Semantics	Dynamic	Identical to JS (with few exceptions*)

Table 1.1: Fundamental difference

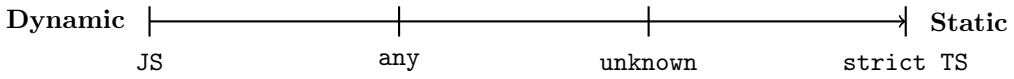
*Exceptions to Type Erasure: Although TypeScript removes types, certain features such as Enums, Namespaces, and Parameter Properties emit actual JavaScript code that affects runtime behavior. For example, Parameter Properties automate assignment within the constructor (ECMA-262 Standard, Clause 15.7), adding logic not present in the original JS source.

1.4 Gradual typing

Gradual typing is the single most important architectural decision in TypeScript. Everything else—unsoundness, pragmatism, any, control-flow analysis—exists to support it.

1.4.1 Gradual Typing as a "dial," not a switch

Gradual typing introduces: typed regions, untyped (dynamic) regions and explicit boundaries between them.
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. When disabled, the compiler permits the silent inference of any, keeping the behavior close to JavaScript's dynamism. When enabled, TypeScript enforces the explicit definition of type boundaries, eliminating uncertainties and shifting the code toward static safety:



Each variable, function, or file can sit at a different position.
How the Compiler Sees Mixed Code?

```
let x: number = 10;
let y: any = getValue();

x.toFixed();   // checked statically
y.toFixed();   // unchecked
```

The compiler does not ignore the dynamic parts. Instead typed code is checked normally, any creates a hole in the type graph and the checker trusts values flowing through that hole. This is selective blindness, not global blindness.

1.4.2 Any

In PL theory terms: any is not a true theoretical top/bottom type. Rather, it is a special "escape hatch" type that effectively turns off type checking (an opt-out from the type system). This is why any is assignable to and from almost every other type, but at the cost of losing soundness guarantees. That is any value can be assigned to any and an any can be assigned to any type.

```
let a: any = 42;
let b: string = a; // allowed
```

This breaks type preservation, subtyping guarantees and obviously soundness. When the compiler encounters any, it effectively says:

> "I will no longer reason about this value. You, the developer, promise it is safe."

This is how untyped JavaScript code remains usable.

The Contagion Effect

Any is viral, inference propagates it and static checking collapses downstream This is why senior engineers treat any as radioactive.

```
let data: any = fetchSomething();

// Infection begins
let result = data.value;    // result: any
let doubled = result * 2;   // still any
let msg = doubled.trim();   // allowed
```

No errors. No warnings. No guarantees.

1.4.3 Practical Application: Brownfield Migration

Brownfield migration is the practical strategy enabled by TypeScript's gradual typing, allowing large legacy JavaScript codebases to adopt static typing incrementally—file by file, rule by rule—without breaking existing systems or stopping development.

Step 1

Accept existing JavaScript. Configure the compiler:

```
{
  "compilerOptions": {
    "allowJs": true,
    "checkJs": false
  }
}
```

This allows:

- JS and TS files together;
- No initial errors;
- Immediate tooling benefits.

Step 2

Let inference do the Work. Even without annotations, TS infers:

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

Inferred as:

```
(a: any, b: any) => any
```

Still dynamic—but now visible.

Step 3

Tighten the Dial. Enable flags gradually:

```
{
  "compilerOptions": {
    "noImplicitAny": true,
    "strictNullChecks": true,
    "strictFunctionTypes": true
  }
}
```

Now:

- Unannotated any becomes an error;
- Developers are forced to make boundaries explicit;
- Dynamic regions shrink over time.

Step 4

Replace broad any usage with:

- Narrow annotations;
- Runtime checks;
- Type guards.

The final goal: push any to the edges of the system.

1.4.4 Unknown

With any, the compiler assumes that every operation is valid.

```
let x: any = "hello";
x.toFixed(); // allowed, runtime crash
```

Unknown means: "I don't know what this is — prove it before using it."

```
let x: unknown = "hello";
x.toFixed(); // Error: compile time error
```

You must narrow first:

```
if (typeof x === "string") {
  x.toUpperCase(); // ok
}
```

Unknown as a Safer Boundary, this allows to keeps dynamic input, forces explicit validation and prevents contagion.

1.4.5 Final comparison

Feature	`any`	`unknown`
Assign anything to it	Yes	Yes
Assign it to other types	Yes	No
Allows unsafe operations	Yes	No
Forces narrowing	No	Yes
Preserves gradual typing	Yes	Yes
Preserves soundness locally	No	Yes

Table 1.2: Comparison: `any` vs `unknown`

Gradual typing in TypeScript works as a dial that allows developers to incrementally trade flexibility for safety; any forms an explicit boundary where static checking gives way to trust, enabling large-scale migration from JavaScript—while unknown provides a modern, safer alternative that preserves flexibility without contaminating the type system.

1.5 Structural typing

Structural typing means: types are compatible if their structure (shape) is compatible — regardless of their declared name. In TypeScript, the compiler does not care what a type is called. It only cares about which properties are present and the types of those properties. This is often summarized as: "If it looks like a duck and quacks like a duck, it is a duck." (**Duck typing**)

1.5.1 Example: Two Different Interfaces, Same Shape

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

interface Account {
  id: string;
  name: string;
}
```

Even though User and Account are different types by name, they have the same structure.

```
let user: User = { id: "u1", name: "Alice" };
let account: Account = user; // Allowed
```

This is valid because:

- User has all properties required by Account;
- types are checked structurally.

```
function printUser(u: User) {
  console.log(u.id, u.name);
}

const acc: Account = { id: "a1", name: "Bob" };

printUser(acc); // Allowed
```

The compiler checks: does acc have id:string and name:string? Yes → compatible.
Also, extra properties are allowed:

```
interface Admin {
  id: string;
  name: string;
  role: "admin";
}

const admin: Admin = {
  id: "a1",
  name: "Root",
  role: "admin",
};

let user2: User = admin; // Allowed
```

Admin has at least the properties required by User, extra properties do not invalidate compatibility. This is the structural subtyping.

Note on Excess Property Checks It is important to note that TypeScript applies stricter rules to *object literals* assigned directly to a typed variable. While `user2 = admin` is allowed due to structural typing, a direct assignment like `let u: User = { id: "a1", name: "Root", role: "admin" }` would trigger an **Excess Property Check** error. This is a safety feature designed to catch typos or unintended properties that would be "lost" or inaccessible through the narrower interface.

Contrast: Nominal Typing (Java / C#) Nominal typing means: types are compatible only if they have an explicit declared relationship. Names matter.

```
class User {
  String id;
  String name;
}

class Account {
  String id;
  String name;
}
```

This is NOT allowed:

```
User u = new Account(); // Compilation error
```

Even though the structure is identical. User and Account are different nominal types, we can not extends / implements relationship exists.

1.5.2 Final consideration

JavaScript Has No Nominal Types

JavaScript:

- is prototype-based; class is syntax over prototypes, and runtime typing remains shape-based;
- does not enforce identity-based typing;
- uses shape-based object access.

Structural typing naturally models JavaScript behavior.

Enables Easy Integration

Structural typing allows:

```
fetch("/api/user")
  .then(res => res.json())
  .then((u: User) => {
    console.log(u.name);
  });
```

The object doesn't need to be created by a `User` constructor. It just needs the right shape.

Reduces Boilerplate

No need for:

- Explicit inheritance;
- Marker interfaces;
- Adapter classes.

Downsides of Structural Typing

Structural typing can be:

- Too permissive;
- Allow accidental compatibility.

Example:

```
interface Point2D {
  x: number;
  y: number;
}

interface ScreenPosition {
  x: number;
  y: number;
}

let p: Point2D = { x: 10, y: 20 };
let s: ScreenPosition = p; // Allowed (maybe unintended)
```

To prevent this, developers sometimes use:

- Branding (TypeScript technique used to simulate nominal typing inside a structural type system);
- Nominal typing patterns.

Feature	Structural Typing (TypeScript)	Nominal Typing (Java/C#)
Type compatibility	Based on structure	Based on name
Requires inheritance	No	Yes
Works well with JS	Yes	No
Flexible	High	Lower
Boilerplate	Low	Higher

Table 1.3: Comparison of Structural vs. Nominal Typing

TypeScript uses structural typing, meaning types are compatible based on their shape rather than their name—unlike nominally typed languages such as Java or C#, where explicit inheritance or declarations are required even if two types look identical.

1.5.3 Duck typing

Duck typing (**runtime**) comes from the phrase: "If it looks like a duck and quacks like a duck, it's a duck." In programming, this means an object is accepted based on what it can do (its structure/behavior) rather than what it is named or declared as. TypeScript's structural type system is a static, compile time form of duck typing.

The Core Relationship is that Duck typing (dynamic languages) compatibility is checked at runtime. Instead Structural typing (TypeScript) compatibility is checked at **compile time**. TypeScript answers this question before the code runs: "Does this value have the required properties with the correct types?" If yes → it's accepted.

TypeScript performs structural type compatibility checks at compile time (often compared informally to "duck typing"), while runtime behavior is plain JavaScript. At runtime if the object is missing quack, it will crash. This is why runtime validation is sometimes needed.

TypeScript's structural typing is a static form of duck typing: a value is accepted by a function if it has the required shape, regardless of its name or declared type, allowing JavaScript-style flexibility with compile time safety.

1.6 Unions

A union type says: a value can be one of several types.

```
type Id = string | number;
```

Meaning: an Id is either a string or a number. it models real JS/TS situations where a value legitimately has multiple possible shapes (e.g., API responses, parsing results, optional configs, polymorphic inputs) while still keeping type safety.

```
function formatId(id: string | number) {
    if (typeof id === "string") {
        return id.toUpperCase();
    } else {
        return id.toFixed(0);
    }
}
```

Here the union forces you to narrow before using operations that only work on one member of the union.

1.6.1 Discriminated Unions (Tagged Unions)

A special pattern of union types, each variant is an object and shares the same property name (the tag).

A discriminated union is a union of object types that all share a common property (the "tag"/"discriminant") whose value is a string/number literal unique to each variant. That shared literal property is what lets TypeScript narrow safely.

Network request state example: Loading | Success | Error

```
type Loading = {
  status: "loading";
};

type Success = {
  status: "success";
  data: { userId: string; name: string };
};

type ErrorState = {
  status: "error";
  error: { message: string; code?: string };
};

type RequestState = Loading | Success | ErrorState;
```

Now TypeScript uses Control Flow Analysis to narrow the type based on the status property check.

Step-by-step narrowing

```
function render(state: RequestState) {
  if (state.status === "loading") {
    // state is Loading here
    return "Loading...";
  }

  if (state.status === "success") {
    // state is Success here
    return `Hello, ${state.data.name}`;
  }

  // by elimination, state is ErrorState here
  return `Oops: ${state.error.message}`;
}
```

Why this is better than "optional fields"

If you tried a single type like:

```
// not recommended
type BadState = {
  status: "loading" | "success" | "error";
  data?: { userId: string; name: string };
  error?: { message: string };
};
```

TypeScript can't guarantee that data exists when status === "success" (unless you add extra checks everywhere). Discriminated unions encode that relationship in the type.

1.7 Type inference

Type inference means: TypeScript automatically figures out the type for you based on how a value is used. You don't always need to write :string, :number, etc., because the compiler can see the value and infer the type safely.

```
let count = 10;
```

TypeScript infers:

```
let count: number;
```

No annotation needed.

Function Return Type Inference TypeScript can infer return types from return statements.

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

Inferred as:

```
function add(a: number, b: number): number
```

In this example, the return type is inferred as number. In general, TypeScript infers a function's return type from its return statements, and when different return paths produce different types, the inferred return type can be a union.

Inferring Types from Arrays: "Best Common Type" When inferring an array type, TypeScript looks for the best common type that all elements share.

```
const nums = [1, 2, 3];
```

Inferred as:

```
number[]
```

When It's Safe to Rely on Inference

You can rely on inference when:

- local variables with clear values:

  ```
  const total = price * tax;
  ```

- obvious return types:

  ```
  function isEmpty(s: string) {
  return s.length === 0;
  }
  ```

- short, readable code

When You Should Be Explicit When you use Public APIs and library boundaries, because documents intent and prevents accidental type changes.

1.7.1 Core rule

TypeScript infers a variable's type from the value it is first assigned. While this base type remains stable to ensure that incompatible values are not assigned later, TypeScript is not rigid: it uses Control Flow Analysis to perform Narrowing. This means the compiler can "refine" its knowledge of a type within specific code blocks (e.g., after an if check), allowing operations that would otherwise be prohibited. Inference is safe because:

- the compiler sees the exact value;
- the value has an unambiguous type;
- no guesswork is involved.

TypeScript infers a variable's type by looking at its initial value, maintains that type's stability for future assignments, but constantly refines its understanding through narrowing—making explicit annotations unnecessary when the value is clear, but essential for defining public boundaries or complex logic.

1.8 Generics

Generics are a TypeScript feature that let you write reusable, type-safe code that works with many different types without losing type information. Instead of hard-coding a specific type (like string or number) or falling back to any, generics introduce type parameters (such as <T>) that act like variables for types. This allows the compiler to:

- capture the type of an input;
- preserve the relationship between inputs and outputs;
- enforce correctness at compile time.

In short Generics provide flexibility and safety at the same time.
They are fundamental for things like:

- Identity functions;
- Arrays (Array<T>)
- Promises (Promise<T>)
- Reusable data structures and APIs.

Without generics, TypeScript would be either too rigid (fixed types everywhere), or too unsafe (any everywhere).

Generics are what make TypeScript both expressive and type-safe.

1.8.1 The problem

Using `any` loses Type information, we want a function that accepts any type and returns the same value. Naive solution with `any`:

```
function identity(value: any) {
  return value;
}
```

This works, but it creates a problem.

```
const result = identity("hello");
```

What is the type of result?

```
//inferred as:
any
```

This is bad because the compiler forgets the relationship between the input type and the output type. So this is allowed:

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

Using any means:

- no type safety;
- no guarantees;
- no autocomplete;
- bugs move to runtime.

1.8.2 The Solution: Generics (<T>)

Identity function with Generics

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

<T> is like a variable that represents a type, when the function is called, T becomes the type of the argument, so the function "remembers" that type and the return value keeps the same type.

How TypeScript uses it

```
const text = identity("hello");
```

TypeScript infers:

```
T = string
```

So the function becomes:

```
(string) => string
```

Now:

```
text.toUpperCase(); //allowed
text.toFixed(2);    //compile time error
```

The input and output types are linked.

1.8.3 Real World: Generics in Arrays

Arrays use generics to lock in the element type.

```
const numbers: Array<number> = [1, 2, 3];
```

This means that every element must be a number and the array's element type is tracked by the type system; subsequent operations are statically checked against the definition `Array<number>` during compilation. So:

```
numbers.push(4);       // OK
numbers.push("five"); // error
```

Without generics, arrays would behave like any[].

Generics allow TypeScript to write flexible functions and data structures while preserving precise type information, acting as "type variables" that capture and return the same type—solving the safety problems caused by any.

Chapter 2

Task 2

Provide short TypeScript examples illustrating (a) errors caught by the type checker and (b) mismatches between static types and JavaScript runtime behaviour.

2.1 Errors caught by the type checker

One of TypeScript's main goals is to detect common programming errors before the code runs. By performing static type checking at compile time, TypeScript can identify invalid function calls, missing properties, and incorrect assignments early—long before they become runtime bugs in JavaScript.

These errors are caught by analyzing the types of values and expressions, without executing the program. This allows developers to fix mistakes immediately, improving reliability, refactorability, and overall code quality.

These examples illustrate how TypeScript's static type checker catches invalid function calls and missing object properties at compile time, enforcing declared type contracts and preventing common JavaScript runtime errors.

2.1.1 Example 1: Incorrect Function Argument Types (add)

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

add("1", 2);
//Error: Argument of type 'string' is not assignable to parameter of type '
    number'.
```

The function add explicitly declares both parameters a and b must be of type number. The return value is also a number and when the function is called with "1" (a string) as the first argument, TypeScript's type checker detects a type mismatch. Because TypeScript performs static analysis at compile time, it can determine that: "1" is not assignable to number, calling the function with these arguments violates its declared type contract. As a result, the compiler reports an error before the code is executed, preventing a potential runtime bug caused by JavaScript's implicit type coercion.

2.1.2 Example 2: Missing Required Properties (User Type)

```
type User = { id: string };

function printId(u: User) {
  console.log(u.id);
}

printId({ });
//Error: Property 'id' is missing in type '{}' but required in type 'User'.
```

The type User defines an object structure that must contain a property id of type string. The function printId relies on this guarantee and accesses u.id directly. When printId is called with an empty object , TypeScript detects that the provided argument does not satisfy the User type and the required property id is missing. This results in a compile time error, because allowing such a call would lead to u.id being undefined at runtime.

2.2 Mismatches between static types and JavaScript runtime behaviour

Although TypeScript adds a static type system on top of JavaScript, it does not change JavaScript's runtime semantics. All TypeScript types are erased during compilation, meaning that the program ultimately runs as plain JavaScript.

As a consequence, there are situations where code that is considered type-correct by the TypeScript compiler can still produce unexpected results or runtime errors when executed. These cases represent mismatches between static type assumptions and actual JavaScript behavior, and they are a direct result of TypeScript's pragmatic design choices.

The following examples illustrate two common sources of such mismatches. In both examples, TypeScript's static guarantees are weaker than JavaScript's dynamic reality. These mismatches exist because TypeScript preserves JavaScript's runtime behavior and erases types after type-checking; type information has no runtime effect.

2.2.1 Example 1: Array Indexing (Out-of-Bounds Access)

```
const xs: number[] = [1, 2, 3];

const n: number = xs[10]; // Allowed by default, but can be rejected with
    noUncheckedIndexedAccess
console.log(n); //undefined at runtime
```

Statically, TypeScript infers that xs is an array of number and accessing any element of xs returns a number. For this reason, the assignment to n is accepted by the compiler.

At runtime, however, JavaScript arrays do not perform bounds checking. Accessing an index that does not exist returns undefined. This creates a mismatch:

- Static type: number;
- Runtime value: undefined.

TypeScript allows this behavior because tracking array bounds would significantly reduce usability and compatibility with JavaScript.

2.2.2 Example 2: Type Assertions (Bypassing the Type Checker)

```
type User = { id: string };

const u = {} as User;  //compiles
console.log(u.id.toUpperCase()); //Error: Cannot read properties of undefined (
    reading 'toUpperCase')
```

The expression **{} as User** is a type assertion, which tells the compiler to treat the value as a User without performing any verification. Statically, TypeScript assumes:

- u.id exists;
- u.id is a string;

As a result, no compile time error is reported. At runtime, however, the object is still an empty JavaScript object: u.id is undefined and calling toUpperCase() on undefined causes a runtime error.

This mismatch occurs because type assertions do not introduce runtime checks; they simply override the compiler's analysis.