Engineer

TypeScript

JavaScript with types checked before it runs

What it is

TypeScript is a programming language from Microsoft that adds type annotations to JavaScript, together with a compiler, `tsc`, that checks them. People shorten the name to TS, and its files end in `.ts`, or `.tsx` when they contain JSX markup. JavaScript syntax is legal TypeScript, which is why the language is called a superset of JavaScript. The compiler is open source under the Apache License 2.0.

The types last only as long as the checking. Ordinary type annotations are erased when TypeScript emits JavaScript; they do not provide runtime validation or require a type-checking runtime library. Type annotations do not change how the JavaScript code behaves when it runs. That one fact explains both its value and its limits: the checker reasons about the types described in the source; it does not validate data that arrives while the program is running.

The type system is structural. Two types are compatible when their shapes match, whatever they are called, which suits JavaScript's habit of passing plain objects around; C# and Java compare declared names instead. Much of the typing is inferred, so annotations sit mostly on function signatures and exported values. The system is also deliberately unsound in a few places. `any` switches checking off for a value, and an assertion written with `as` tells the compiler to take the programmer's word for a type. Both disappear from the output along with everything else.

There are three common ways to run it. `tsc` checks a project and emits JavaScript. Bundlers such as esbuild strip the types without checking them, and leave checking to a separate `tsc --noEmit` run. Node.js, since v23.6.0 and v22.18.0, runs `.ts` files directly by replacing the type syntax with whitespace,

again with no type checking, and accepts only syntax that can be erased that way. TypeScript 7.0, announced in July 2026, is a native Go version of the compiler, which the team reports as typically 8 to 12 times faster on full builds.

Use the concept in a decision

Put “What is TypeScript?” to work in a real decision. We can help relate the concept to the system you are building or buying. Interfaces that load quickly, work on a phone and behave the same in every browser your customers happen to use.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Directed full-stack GIS map development — full-stack mapping, from database to browser

Directed full-stack GIS map development Get in touch →

Why it matters

The question a team answers is whether a build step and a checker earn their keep by catching errors before the code runs. In a codebase that several people change every week they usually do. Renaming a field shows every place that used it, the editor completes and explains code from the types, and a module’s signature becomes a contract its callers cannot quietly break. The cost is a check in every build, a configuration file with many options, and type definitions that have to keep pace with the libraries they describe.

Strictness is the setting that matters most. The `strict` flag turns on a family of checks, among them `strictNullChecks` and `noImplicitAny`, and the handbook describes the result as a dial rather than a switch. Since TypeScript 6.0 that flag is on by default, so a new project starts strict; an older one may still set it to false, and turning it on later means working through every file it touches. Two useful checks sit outside the family and have to be enabled separately: `noUncheckedIndexedAccess` and `exactOptionalPropertyTypes`.

Erasure sets the limit. A program that reads JSON from a network request, a file, a queue or a form has only the programmer’s word for that data’s shape, because the types are gone by the time the data arrives. Boundaries therefore need validation at runtime, ideally generated from the same definition as the types, so the check and the type cannot disagree. TypeScript makes the inside of a program consistent and leaves the edges to code.

The checker also feeds other tools. A linter such as typescript-eslint can ask the compiler for type information, so that a rule can flag a promise nobody awaits or a value of type `any` spreading through a module. That is how a team turns its own conventions into errors rather than review comments.

What it looks like in practice

In the projects shown on this site, TypeScript is kept honest by generated code and by tooling rather than by convention. Three pages show how.

On one platform the frontend's TypeScript client is never written by hand. It is generated from the API's own description, which Huma emits from the Go types, which in turn follow the database schema and its functions. At the documented implementation stage, the client came to 44,076 lines, and 61 mock handlers generated alongside it let the frontend's tests run against the real contract instead of a hand-written fixture. Every hop has its own guard. A contract check in CI compares what the frontend sends with what the API expects and fails the build when they diverge, with a schema probe underneath that inspects the real shapes. It covers optional body fields, where missing and null are easily confused, and enums in query parameters. A field that changes on one side now fails a build minutes after the change, rather than showing up as a 422 in the browser after the mismatch has shipped. In exchange, a change to the contract reaches the client's types only by running the chain again, and regenerating is a chore.

Another page sets one bar for six languages, TypeScript among them. Every diagnostic ESLint reports is an error, there is no warning tier to hide in, and inline suppressions are banned, so a finding has to be fixed rather than silenced. Pre-commit and pre-push hooks run the linters and the tests, and function- and file-length limits keep modules readable.

The linters that ship with a language stop short of a project's own rules, so fifteen custom ESLint rules held that repository's house patterns in TypeScript at the documented implementation stage. They belonged to a guard engine of 274 check scripts, 268 of them registered in the commit hooks, and the guards carried 146 tests of their own, because a broken guard passes everything without anyone noticing. One rule concerns tests rather than types: a commit prefixed `fix:` must include a test that fails without the fix. Review time moved from mechanics to design, and committing became slow in exchange.

Where it goes wrong

Most TypeScript failures come from trusting the types further than the compiler can see.

A field typed as a string turns out to be undefined at runtime. The data came from outside the program and was given a type rather than checked. The DOM library types `res.json()` as a promise of `any`, so the compiler accepts whatever the code does with the result, and an assertion makes the trust explicit without making it safe:

```
const res = await fetch("/api/user");
const user = (await res.json()) as User; // trusted, never checked
user.email.toLowerCase(); // TypeError if email is missing
```

Treat external data as `unknown`, validate it at the boundary with a runtime schema, and generate the type from that schema so the two stay in step. A typed lint rule can then flag any `any` that slips past.

Code with type errors reaches production anyway. `tsc` still writes JavaScript when it reports errors, because `noEmitOnError` defaults to false, and bundlers and Node.js strip types without checking them at all. Run `tsc --noEmit` as a required step in CI and in the pre-commit hook, and fail the build on its exit code.

An index lookup returns undefined that the type ruled out. Reading an array element or a key through an index signature is typed as always present, even past the end of the array. Turn on `noUncheckedIndexedAccess`, which adds `undefined` to those reads and makes the code handle it. Pair it with `exactOptionalPropertyTypes`, which tells an absent property apart from one set to `undefined`.

A file that compiles with tsc fails when Node.js runs it. Type stripping removes only syntax with no runtime meaning. Enums and namespaces that hold values make Node.js stop with `ERR_UNSUPPORTED_TYPESCRIPT_SYNTAX`, and a type imported without the `type` keyword is treated as a value import that fails at runtime. Turn on `erasableSyntaxOnly` and `verbatimModuleSyntax` so the compiler reports both before Node.js does, write `import type` for types, and replace enums with object literals marked `as const`.

Further reading

- TypeScript Documentation — the handbook, the reference pages and the release notes, maintained by the TypeScript team.
- The Basics — the handbook chapter on static checking, emitting with errors, erased types and strictness.
- TSConfig Reference: strict — every compiler option with its default, including the checks the strict flag turns on.
- Type Compatibility — structural subtyping, and the places where the type system is unsound on purpose.
- Node.js: Modules: TypeScript — type stripping, the syntax it rejects and the compiler settings Node.js recommends.
- Linting with Type Information — how typescript-eslint gives lint rules access to the type checker.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is TypeScript?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/typescript/