Mastering TypeScript: A Comprehensive Guide

This book provides a complete guide to TypeScript, starting from the basics and advancing to more complex topics. It is ideal for both beginners and experienced developers, offering insights into TypeScript syntax, static type checking, project configuration, and advanced features. Learn how to leverage TypeScript to enhance JavaScript applications and improve code quality through detailed tutorials and case studies.

Table of Contents

Introduction to TypeScript

- TS for the New Programmer
- TypeScript for JS Programmers
- TS for Java/C# Programmers
- TS for Functional Programmers
- TypeScript Tooling in 5 minutes

Understanding JavaScript

- What is JavaScript? A Brief History

Static Type Checking

- TypeScript: A Static Type Checker

TypeScript Syntax

- A Typed Superset of JavaScript

Type Manipulation

- Type Manipulation

TypeScript in Practice

- Utility Types
- Cheat Sheets

JavaScript and TypeScript Interoperability

- Using TypeScript with JavaScript

Project Configuration in TypeScript

- What is a tsconfig.json

Advanced TypeScript Features

- Generics

Tutorials and Case Studies

- ASP.NET Core
- Gulp

TypeScript for JavaScript Programmers

Overview

TypeScript stands in an unusual relationship to JavaScript. TypeScript offers all of JavaScript's features, and an additional layer on top of these: TypeScript's type system.

For example, JavaScript provides language primitives like `string` and `number`, but it doesn't check that you've consistently assigned these. TypeScript does.

This means that your existing working JavaScript code is also TypeScript code. The main benefit of TypeScript is that it can highlight unexpected behavior in your code, lowering the chance of bugs.

This tutorial provides a brief overview of TypeScript, focusing on its type system.

Types by Inference

TypeScript knows the JavaScript language and will generate types for you in many cases. For example, when creating a variable and assigning it a value, TypeScript will infer its type.

```
let helloWorld = "Hello World"; // helloWorld is inferred as string
```

Try it here

By understanding how JavaScript works, TypeScript can build a type-system that accepts JavaScript code but has types. This offers a type-system without needing to add extra characters to make types explicit in your code. That's how TypeScript knows that `helloWorld` is a `string` in the above example.

You may have written JavaScript in Visual Studio Code, and had editor auto-completion. Visual Studio Code uses TypeScript under the hood to make it easier to work with JavaScript.

Defining Types

You can use a wide variety of design patterns in JavaScript. However, some make it difficult for types to be inferred automatically (e.g., patterns that use dynamic programming). To handle these cases, TypeScript supports an extension of JavaScript, allowing you to specify types explicitly.

For example, to create an object with an inferred type including `name: string` and `id: number`, you can write:

```
const user = {
  name: "Hayes",
  id: 0
};
```

You can explicitly define the shape with an interface:

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

Then declare that a JavaScript object conforms to this shape:

```
const user: User = {
  name: "Hayes",
  id: 0
};
```

TypeScript warns if the object doesn't match the interface.

You can also use interfaces with classes:

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

class UserAccount {
  name: string;
  id: number;

  constructor(name: string, id: number) {
    this.name = name;
    this.id = id;
  }
}

const user: User = new UserAccount("Murphy", 1);
```

Interfaces can be used to annotate function parameters and return types:

```
function deleteUser(user: User) {
  // ...
}

function getAdminUser(): User {
  // ...
}
```

JavaScript supports primitive types such as `boolean`, `bigint`, `null`, `number`, `string`, `symbol`, and `undefined`, which TypeScript extends with types like `any`, `unknown`, `never`, and `void`.

TypeScript has two syntax options for defining types: interfaces and type aliases. Preferred is `interface`; use `type` when specific features are needed.

Composing Types

Types can be combined to form complex types, mainly via unions and generics.

Unions

A union type declares that a variable could be one of several types:

```
type MyBool = true | false;
```

Additionally, to describe a value that can be specific string literals:

```
type WindowStates = "open" | "closed" | "minimized";
type LockStates = "locked" | "unlocked";
type PositiveOddNumbersUnderTen = 1 | 3 | 5 | 7 | 9;
```

Unions are useful in functions:

```
function getLength(obj: string | string[]) {
  return obj.length;
}
```

To determine the type of a variable at runtime, use `typeof`:

Type	Predicate
string	`typeof s === "string"`
number	`typeof n === "number"`
boolean	`typeof b === "boolean"`
undefined	`typeof undefined === "undefined"`
function	`typeof f === "function"`
array	`Array.isArray(a)`

Example usage:

```
function wrapInArray(obj: string | string[]) {
  if (typeof obj === "string") {
    return [obj];
  }
  return obj;
}
```

Generics

Generics add variables to types, such as in arrays:

```
type StringArray = Array<string>;
type NumberArray = Array<number>;
type ObjectWithNameArray = Array<{ name: string }>;
```

You can define your own generic types:

```
interface Backpack<Type> {
  add: (obj: Type) => void;
  get: () => Type;
}

declare const backpack: Backpack<string>;
const object = backpack.get(); // object is a string
backpack.add(23); // Error: number not assignable to string
```

Functions can also be generic:

```
function identity<Type>(arg: Type): Type {
  return arg;
}
```

Structural Type System

TypeScript checks if two objects have the same shape. If they do, they are compatible, regardless of their explicit types.

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

function logPoint(p: Point) {
  console.log(`${p.x}, ${p.y}`);
}

const point = { x: 12, y: 26 };
logPoint(point); // OK

const point3 = { x: 12, y: 26, z: 89 };
logPoint(point3); // OK, shape matches

const rect = { x: 33, y: 3, width: 30, height: 80 };
logPoint(rect); // OK if shape matches Point, but this object has extra properties

const color = { hex: "#187ABF" };
logPoint(color); // Error: missing x and y properties
```

Classes and objects also conform to shapes:

```
class VirtualPoint {
  x: number;
  y: number;

  constructor(x: number, y: number) {
    this.x = x;
    this.y = y;
  }
}

const newVPoint = new VirtualPoint(13, 56);
logPoint(newVPoint); // OK
```

Next Steps

- Read the full Handbook
- Explore Playground examples

On this page

- Types by Inference
- Defining Types
- Composing Types
  - Unions
  - Generics
- Structural Type System
- Next Steps

Additional

The TypeScript docs are an open source project. Help improve these pages by sending a Pull Request.

Last updated: Jun 30, 2025

404

File not found

The site configured at this address does not contain the requested file.

If this is your site, make sure that the filename case matches the URL as well as any file permissions.
For root URLs (like http://example.com/) you must provide an `index.html` file.

Read the full documentation for more information about using **GitHub Pages**.

GitHub Status — @githubstatus

404

File not found

The site configured at this address does not contain the requested file.

If this is your site, make sure that the filename case matches the URL as well as any file permissions. For root URLs (like `http://example.com/`) you must provide an `index.html` file.

Read the full documentation for more information about using **GitHub Pages**.

GitHub Status — @githubstatus

404

File not found

The site configured at this address does not contain the requested file.

If this is your site, make sure that the filename case matches the URL as well as any file permissions. For root URLs (like `http://example.com/`) you must provide an `index.html` file.

Read the full documentation for more information about using **GitHub Pages**.

GitHub Status — @githubstatus

404

File not found

The site configured at this address does not contain the requested file.

If this is your site, make sure that the filename case matches the URL as well as any file permissions. For root URLs (like `http://example.com/`) you must provide an `index.html` file.

Read the full documentation for more information about using **GitHub Pages**.

GitHub Status — @githubstatus

TypeScript for the New Programmer

Congratulations on choosing TypeScript as one of your first languages — you're already making good decisions!

You've probably already heard that TypeScript is a "flavor" or "variant" of JavaScript. The relationship between TypeScript (TS) and JavaScript (JS) is rather unique among modern programming languages, so learning more about this relationship will help you understand how TypeScript adds to JavaScript.

What is JavaScript? A Brief History

JavaScript (also known as ECMAScript) started its life as a simple scripting language for browsers. At the time it was invented, it was expected to be used for short snippets of code embedded in a web page — writing more than a few dozen lines of code would have been somewhat unusual. Due to this, early web browsers executed such code pretty slowly. Over time, though, JS became more and more popular, and web developers started using it to create interactive experiences.

Web browser developers responded to this increased JS usage by optimizing their execution engines (dynamic compilation) and extending what could be done with it (adding APIs), which in turn made web developers use it even more. On modern websites, your browser is frequently running applications that span hundreds of thousands of lines of code. This is the long and gradual growth of "the web," starting as a simple network of static pages, and evolving into a platform for rich *applications* of all kinds.

More than this, JS has become popular enough to be used outside the context of browsers, such as implementing JS servers using node.js. The "run anywhere" nature of JS makes it an attractive choice for cross-platform development. There are many developers these days that use *only* JavaScript to program their entire stack!

To summarize, we have a language that was designed for quick uses, and then grew to a full-fledged tool to write applications with millions of lines. Every language has its own *quirks* — oddities and surprises, and JavaScript's humble beginning makes it have *many* of these. Some examples:

- JavaScript's equality operator (`==`) *coerces* its operands, leading to unexpected behavior:

```
if ("" == 0) {
  // It is! But why??
}
if (1 < x < 3) {
  // True for *any* value of x!
}
```

- JavaScript also allows accessing properties which aren't present:

```
const obj = { width: 10, height: 15 };
// Why is this NaN? Spelling is hard!
const area = obj.width * obj.heigth;
```

Most programming languages would throw an error when these sorts of errors occur, some would do so during compilation — before any code is running. When writing small programs, such quirks are annoying but manageable; when writing applications with hundreds or thousands of lines of code, these constant surprises are a serious problem.

TypeScript: A Static Type Checker

We said earlier that some languages wouldn't allow those buggy programs to run at all. Detecting errors in code without running it is referred to as *static checking*. Determining what's an error and what's not based on the kinds of values being operated on is known as static *type* checking.

TypeScript checks a program for errors before execution, and does so based on the *kinds of values*, making it a *static type checker*. For example, the last example above has an error because of the *type* of `obj`. Here's the error TypeScript found:

Try the example:

```
const obj = { width: 10, height: 15 };
const area = obj.width * obj.heigth; // Error: Property 'heigth' does not exist on typ
```

A Typed Superset of JavaScript

How does TypeScript relate to JavaScript, though?

Syntax

TypeScript is a language that is a *superset* of JavaScript: JS syntax is therefore legal TS. Syntax refers to the way we write text to form a program. For example, this code has a *syntax* error because it's missing a `)`:

```
let a = (4;
// ')' expected. (TS1005)
// Try: [link](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygIwAYMFYBQA
```

TypeScript doesn't consider any JavaScript code to be an error because of its syntax. This means you can take any working JavaScript code and put it in a TypeScript file without worrying about exactly how it is written.

Types

However, TypeScript is a *typed* superset, meaning that it adds rules about how different kinds of values can be used. The earlier error about `obj.heigth` was not a *syntax* error: it is an error of using some kind of value (a *type*) in an incorrect way.

As another example, this is JavaScript code that you can run in your browser, and it *will* log a value:

```
console.log(4 / []);
```

This syntactically-legal program logs `Infinity`. TypeScript, though, considers division of number by an array to be a nonsensical operation, and will issue an error:

Try this example:

```
console.log(4 / []);
// Error: The right-hand side of an arithmetic operation must be of type 'any', 'numbe
```

It's possible you really *did* intend to divide a number by an array, perhaps just to see what happens, but most of the time, though, this is a programming mistake. TypeScript's type checker is designed to allow correct programs through while still catching as many common errors as possible.

(Later, we'll learn about settings you can use to configure how strictly TypeScript checks your code.)

If you move some code from a JavaScript file to a TypeScript file, you might see *type errors* depending on how the code is written. These may be legitimate problems with the code, or TypeScript being overly conservative. Throughout this guide we'll demonstrate how to add various TypeScript syntax to eliminate such errors.

Runtime Behavior

TypeScript is also a programming language that preserves the *runtime behavior* of JavaScript. For example, dividing by zero in JavaScript produces `Infinity` instead of throwing a runtime exception. As a principle, TypeScript **never** changes the runtime behavior of JavaScript code.

This means that if you move code from JavaScript to TypeScript, it is **guaranteed** to run the same way, even if TypeScript thinks that the code has type errors.

Keeping the same runtime behavior as JavaScript is a foundational promise of TypeScript because it means you can easily transition between the two languages without worrying about subtle differences that might make your program stop working.

Erased Types

Roughly speaking, once TypeScript's compiler is done with checking your code, it *erases* the types to produce the resulting "compiled" code. This means that once your code is compiled, the resulting plain JS code has no type information.

This also means that TypeScript never changes the *behavior* of your program based on the types it inferred. The bottom line is that while you might see type errors during compilation, the type system itself has no bearing on how your program works when it runs.

Finally, TypeScript doesn't provide any additional runtime libraries. Your programs will use the same standard library (or external libraries) as JavaScript programs, so there's no additional TypeScript-specific framework to learn.

Learning JavaScript and TypeScript

We frequently see the question "Should I learn JavaScript or TypeScript?".

The answer is that you can't learn TypeScript without learning JavaScript! TypeScript shares syntax and runtime behavior with JavaScript, so anything you learn about JavaScript is helping you learn TypeScript at the same time.

There are many, many resources available for programmers to learn JavaScript; you should *not* ignore these resources if you're writing TypeScript. For example, there are about 20 times more StackOverflow questions tagged `javascript` than `typescript`, but *all* of the `javascript` questions also apply to TypeScript.

If you find yourself searching for something like "how to sort a list in TypeScript", remember: **TypeScript is JavaScript's runtime with a compile-time type checker**. The way you sort a list in TypeScript is the same way you do so in JavaScript. If you find a resource that uses TypeScript directly, that's great too, but don't limit yourself to thinking you need TypeScript-specific answers for everyday questions about how to accomplish runtime tasks.

Next Steps

This was a brief overview of the syntax and tools used in everyday TypeScript. From here, you can:

- Learn some of the JavaScript fundamentals, we recommend either:
  - Microsoft's JavaScript Resources
  - JavaScript guide at the Mozilla Web Docs
- Continue to TypeScript for JavaScript Programmers
- Read the full Handbook from start to finish
- Explore the Playground examples

On this page

- What is JavaScript? A Brief History
- TypeScript: A Static Type Checker
    - A Typed Superset of JavaScript
- Learning JavaScript and TypeScript
- Next Steps

Is this page helpful?

Yes No

Help us improve these pages

The TypeScript docs are an open source project. Help us improve these pages by sending a Pull Request ❤

Contributors to this page:

OT, EB, XL, NS, AO, 8+

Last updated: Jun 30, 2025

Using TypeScript

- Get Started
- Download
- Community
- Playground
- TSConfig Reference
- Code Samples
- Why TypeScript
- Design

Community

- Get Help
- Blog
- GitHub Repo
- Community Chat
- @TypeScript
- Mastodon
- Stack Overflow
- Web Repo

TypeScript for the New Programmer

Congratulations on choosing TypeScript as one of your first languages — you're already making good decisions!

You've probably already heard that TypeScript is a "flavor" or "variant" of JavaScript.
The relationship between TypeScript (TS) and JavaScript (JS) is rather unique among modern programming languages, so learning more about this relationship will help you understand how TypeScript adds to JavaScript.

What is JavaScript? A Brief History

JavaScript (also known as ECMAScript) started its life as a simple scripting language for browsers.
At the time it was invented, it was expected to be used for short snippets of code embedded in a web page — writing more than a few dozen lines of code would have been somewhat unusual.
Due to this, early web browsers executed such code pretty slowly.
Over time, though, JS became more and more popular, and web developers started using it to create interactive experiences.

Web browser developers responded to this increased JS usage by optimizing their execution engines (dynamic compilation) and extending what could be done with it (adding APIs), which in turn made web developers use it even more.
On modern websites, your browser is frequently running applications that span hundreds of thousands of lines of code.
This is the long and gradual growth of "the web," starting as a simple network of static pages, and evolving into a platform for rich *applications* of all kinds.

More than this, JS has become popular enough to be used outside the context of browsers, such as implementing JS servers using node.js.
The "run anywhere" nature of JS makes it an attractive choice for cross-platform development.
There are many developers these days that use *only* JavaScript to program their entire stack!

To summarize, we have a language that was designed for quick uses, and then grew to a full-fledged tool to write applications with millions of lines.
Every language has its own *quirks* — oddities and surprises, and JavaScript's humble beginning makes it have *many* of these. Some examples:

- JavaScript's equality operator (`==`) *coerces* its operands, leading to unexpected behavior:

  ```
  if ("" == 0) {
    // It is! But why??
  }
  if (1 < x < 3) {
    // True for *any* value of x!
  }
  ```

- JavaScript also allows accessing properties which aren't present:

  ```
  const obj = { width: 10, height: 15 };
  // Why is this NaN? Spelling is hard!
  const area = obj.width * obj.heigth;
  ```

Most programming languages would throw an error when these sorts of errors occur, some would do so during compilation — before any code is running.
When writing small programs, such quirks are annoying but manageable; when writing applications with hundreds or thousands of lines, these constant surprises are a serious problem.

TypeScript: A Static Type Checker

We said earlier that some languages wouldn't allow those buggy programs to run at all.
Detecting errors in code without running it is referred to as *static checking*.
Determining what's an error and what's not based on the kinds of values being operated on is known as static *type* checking.

TypeScript checks a program for errors before execution, and does so based on the *kinds of values*, making it a *static type checker*.
For example, the last example above has an error because of the *type* of `obj`.
Here's the error TypeScript found:

```
const obj = { width: 10, height: 15 };
const area = obj.width * obj.heigth;
// Property 'heigth' does not exist on type '{ width: number; height: number; }'. Did
// 2551
```

A Typed Superset of JavaScript

How does TypeScript relate to JavaScript, though?

Syntax

TypeScript is a language that is a *superset* of JavaScript: JS syntax is therefore legal TS.
Syntax refers to the way we write text to form a program.
For example, this code has a *syntax* error because it's missing a `)`:

```
let a = (4; // ')' expected.
```

TypeScript doesn't consider any JavaScript code to be an error because of its syntax.
This means you can take any working JavaScript code and put it in a TypeScript file without worrying about exactly how it is written.

Types

However, TypeScript is a *typed* superset, meaning that it adds rules about how different kinds of values can be used.
The earlier error about `obj.heigth` was not a *syntax* error: it is an error of using some kind of value (a *type*) in an incorrect way.

As another example, this is JavaScript code that you can run in your browser, and it *will* log a value:

```
console.log(4 / []);
```

This syntactically-legal program logs `Infinity`.
TypeScript, though, considers division of number by an array to be a nonsensical operation, and will issue an error:

```
console.log(4 / []);
// The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bi
// 2363
```

It's possible you really *did* intend to divide a number by an array, perhaps just to see what happens, but most of the time, though, this is a programming mistake.

TypeScript's type checker is designed to allow correct programs through while still catching as many common errors as possible.
(Later, we'll learn about settings you can use to configure how strictly TypeScript checks your code.)

If you move some code from a JavaScript file to a TypeScript file, you might see *type errors* depending on how the code is written.
These may be legitimate problems with the code, or TypeScript being overly conservative.
Throughout this guide we'll demonstrate how to add various TypeScript syntax to eliminate such errors.

Runtime Behavior

TypeScript is also a programming language that preserves the *runtime behavior* of JavaScript.
For example, dividing by zero in JavaScript produces `Infinity` instead of throwing a runtime exception.
As a principle, TypeScript **never** changes the runtime behavior of JavaScript code.

This means that if you move code from JavaScript to TypeScript, it is **guaranteed** to run the same way, even if TypeScript thinks that the code has type errors.
Keeping the same runtime behavior as JavaScript is a foundational promise of TypeScript because it means you can easily transition between the two languages without worrying about subtle differences that might make your program stop working.

Erased Types

Roughly speaking, once TypeScript's compiler is done with checking your code, it *erases* the types to produce the resulting "compiled" code.
This means that once your code is compiled, the resulting plain JS code has no type information.

This also means that TypeScript never changes the *behavior* of your program based on the types it inferred.
The bottom line is that while you might see type errors during compilation, the type system itself has no bearing on how your program works when it runs.

Finally, TypeScript doesn't provide any additional runtime libraries.
Your programs will use the same standard library (or external libraries) as JavaScript programs, so there's no additional TypeScript-specific framework to learn.

Learning JavaScript and TypeScript

We frequently see the question "Should I learn JavaScript or TypeScript?".

The answer is that you can't learn TypeScript without learning JavaScript!
TypeScript shares syntax and runtime behavior with JavaScript, so anything you learn about JavaScript is helping you learn TypeScript at the same time.

There are many, many resources available for programmers to learn JavaScript; you should *not* ignore these resources if you're writing TypeScript.
For example, there are about 20 times more StackOverflow questions tagged `javascript` than `typescript`, but *all* of the `javascript` questions also apply to TypeScript.

If you find yourself searching for something like "how to sort a list in TypeScript", remember: **TypeScript is JavaScript's runtime with a compile-time type checker**.
The way you sort a list in TypeScript is the same way you do so in JavaScript.
If you find a resource that uses TypeScript directly, that's great too, but don't limit yourself to thinking you need TypeScript-specific answers for everyday questions about how to accomplish runtime tasks.

Next Steps

This was a brief overview of the syntax and tools used in everyday TypeScript. From here, you can:

- Learn some of the JavaScript fundamentals, we recommend either:
    - Microsoft's JavaScript Resources
    - JavaScript guide at the Mozilla Web Docs
- Continue to TypeScript for JavaScript Programmers
- Read the full Handbook from start to finish
- Explore the Playground examples

On this page

- What is JavaScript? A Brief History
- TypeScript: A Static Type Checker
    - A Typed Superset of JavaScript
- Learning JavaScript and TypeScript
- Next Steps

Is this page helpful?

Yes No

The TypeScript docs are an open source project. Help us improve these pages by sending a Pull Request ❤

Contributors to this page:
OT, EB, XL, NS, AO, 8+

Last updated: Jun 30, 2025

TypeScript for the New Programmer

Congratulations on choosing TypeScript as one of your first languages — you're already making good decisions!

You've probably already heard that TypeScript is a "flavor" or "variant" of JavaScript. The relationship between TypeScript (TS) and JavaScript (JS) is rather unique among modern programming languages, so learning more about this relationship will help you understand how TypeScript adds to JavaScript.

What is JavaScript? A Brief History

JavaScript (also known as ECMAScript) started its life as a simple scripting language for browsers.
At the time it was invented, it was expected to be used for short snippets of code embedded in a web page — writing more than a few dozen lines of code would have been somewhat unusual.
Due to this, early web browsers executed such code pretty slowly.
Over time, though, JS became more and more popular, and web developers started using it to create interactive experiences.

Web browser developers responded to this increased JS usage by optimizing their execution engines (dynamic compilation) and extending what could be done with it (adding APIs), which in turn made web developers use it even more.
On modern websites, your browser is frequently running applications that span hundreds of thousands of lines of code.
This is the long and gradual growth of "the web", starting as a simple network of static pages, and evolving into a platform for rich *applications* of all kinds.

More than this, JS has become popular enough to be used outside the context of browsers, such as implementing JS servers using node.js.
The "run anywhere" nature of JS makes it an attractive choice for cross-platform development.
There are many developers these days that use *only* JavaScript to program their entire stack!

To summarize, we have a language that was designed for quick uses, and then grew to a full-fledged tool to write applications with millions of lines.
Every language has its own *quirks* — oddities and surprises, and JavaScript's humble beginning makes it have *many* of these. Some examples:

- JavaScript's equality operator (`==`) *coerces* its operands, leading to unexpected behavior:

```
if ("" == 0) {
  // It is! But why??
}
if (1 < x < 3) {
  // True for *any* value of x!
}
```

- JavaScript also allows accessing properties which aren't present:

```
const obj = { width: 10, height: 15 };
// Why is this NaN? Spelling is hard!
const area = obj.width * obj.heigth;
```

Most programming languages would throw an error when these sorts of errors occur, some would do so during compilation — before any code is running.

When writing small programs, such quirks are annoying but manageable; when writing applications with hundreds or thousands of lines of code, these constant surprises are a serious problem.

TypeScript: A Static Type Checker

We said earlier that some languages wouldn't allow those buggy programs to run at all.
Detecting errors in code without running it is referred to as *static checking*.
Determining what's an error and what's not based on the kinds of values being operated on is known as static *type* checking.

TypeScript checks a program for errors before execution, and does so based on the *kinds of values*, making it a *static type checker*.
For example, the last example above has an error because of the *type* of `obj`.
Here's the error TypeScript found:

Try it on the TypeScript playground

```
const obj = { width: 10, height: 15 };
const area = obj.width * obj.heigth;
// Property 'heigth' does not exist on type '{ width: number; height: number; }'. Did
```

A Typed Superset of JavaScript

How does TypeScript relate to JavaScript, though?

Syntax

TypeScript is a language that is a *superset* of JavaScript: JS syntax is therefore legal TS.
Syntax refers to the way we write text to form a program.
For example, this code has a *syntax* error because it's missing a `)`:

```
let a = (4
// ') expected.'
```

TypeScript doesn't consider any JavaScript code to be an error because of its syntax.
This means you can take any working JavaScript code and put it in a TypeScript file without worrying about exactly how it is written.

Types

However, TypeScript is a *typed* superset, meaning that it adds rules about how different kinds of values can be used.
The earlier error about `obj.heigth` was not a *syntax* error: it is an error of using some kind of value (a *type*) in an incorrect way.

As another example, this is JavaScript code that you can run in your browser, and it *will* log a value:

```
console.log(4 / []);
```

This syntactically-legal program logs `Infinity`.
TypeScript, though, considers division of number by an array to be a nonsensical operation, and will issue an error:

Try it on the TypeScript playground

```
console.log(4 / []);
// The right-hand side of an arithmetic operation must be of type 'any', 'number', 'bi
```

It's possible you really *did* intend to divide a number by an array, perhaps just to see what happens, but most of the time, though, this is a programming mistake.
TypeScript's type checker is designed to allow correct programs through while still catching as many common errors as possible.
(Later, we'll learn about settings you can use to configure how strictly TypeScript checks your code.)

If you move some code from a JavaScript file to a TypeScript file, you might see *type errors* depending on how the code is written.
These may be legitimate problems with the code, or TypeScript being overly conservative.
Throughout this guide we'll demonstrate how to add various TypeScript syntax to eliminate such errors.

Runtime Behavior

TypeScript is also a programming language that preserves the *runtime behavior* of JavaScript.
For example, dividing by zero in JavaScript produces `Infinity` instead of throwing a runtime exception.
As a principle, TypeScript **never** changes the runtime behavior of JavaScript code.

This means that if you move code from JavaScript to TypeScript, it is **guaranteed** to run the same way, even if TypeScript thinks that the code has type errors.
Keeping the same runtime behavior as JavaScript is a foundational promise of TypeScript because it means you can easily transition between the two languages without worrying about subtle differences that might make your program stop working.

Erased Types

Roughly speaking, once TypeScript's compiler is done with checking your code, it *erases* the types to produce the resulting "compiled" code.
This means that once your code is compiled, the resulting plain JS code has no type information.

This also means that TypeScript never changes the *behavior* of your program based on the types it inferred.
The bottom line is that while you might see type errors during compilation, the type system itself has no bearing on how your program works when it runs.

Finally, TypeScript doesn't provide any additional runtime libraries.
Your programs will use the same standard library (or external libraries) as JavaScript programs, so there's no additional TypeScript-specific framework to learn.

Learning JavaScript and TypeScript

We frequently see the question "Should I learn JavaScript or TypeScript?".

The answer is that you can't learn TypeScript without learning JavaScript!
TypeScript shares syntax and runtime behavior with JavaScript, so anything you learn about JavaScript is helping you learn TypeScript at the same time.

There are many, many resources available for programmers to learn JavaScript; you should *not* ignore these resources if you're writing TypeScript.
For example, there are about 20 times more StackOverflow questions tagged `javascript` than `typescript`, but *all* of the `javascript` questions also apply to TypeScript.

If you find yourself searching for something like "how to sort a list in TypeScript", remember: **TypeScript is JavaScript's runtime with a compile-time type checker**.
The way you sort a list in TypeScript is the same way you do so in JavaScript.
If you find a resource that uses TypeScript directly, that's great too, but don't limit yourself to thinking you need TypeScript-specific answers for everyday questions about how to accomplish runtime tasks.

Next Steps

This was a brief overview of the syntax and tools used in everyday TypeScript. From here, you can:

- Learn some of the JavaScript fundamentals, we recommend either:
  - Microsoft's JavaScript Resources
  - JavaScript guide at the Mozilla Web Docs
- Continue to TypeScript for JavaScript Programmers
- Read the full Handbook from start to finish
- Explore the Playground examples

On this page

- What is JavaScript? A Brief History
- TypeScript: A Static Type Checker
  - A Typed Superset of JavaScript
- Learning JavaScript and TypeScript
- Next Steps

Is this page helpful?

Yes | No

The TypeScript docs are an open source project. Help us improve these pages by sending a Pull Request ❤

Contributors to this page:
OT, EB, XL, NS, AO, 8+

Last updated: Jun 30, 2025

More on Functions

Functions are the basic building block of any application, whether they're local functions, imported from another module, or methods on a class. They're also values, and just like other values, TypeScript has many ways to describe how functions can be called. Let's learn about how to write types that describe functions.

Function Type Expressions

The simplest way to describe a function is with a *function type expression*. These types are syntactically similar to arrow functions:

```
function greeter(fn: (a: string) => void) {
  fn("Hello, World");
}

function printToConsole(s: string) {
  console.log(s);
}

greeter(printToConsole);
```

The syntax `(a: string) => void` means "a function with one parameter, named `a`, of type `string`, that doesn't have a return value". Just like with function declarations, if a parameter type isn't specified, it's implicitly `any`.

> Note that the parameter name is **required**. The function type `(string) => void` means "a function with a parameter of type `string`," but **without a name**, which is invalid.

Call Signatures

In JavaScript, functions can have properties in addition to being callable. However, the function type expression syntax doesn't allow for declaring properties. If we want to describe something callable with properties, we can write a *call signature* in an object type:

```
type DescribableFunction = {
  description: string;
  (someArg: number): boolean;
};

function doSomething(fn: DescribableFunction) {
  console.log(fn.description + " returned " + fn(6));
}

function myFunc(someArg: number) {
  return someArg > 3;
}

myFunc.description = "default description";

doSomething(myFunc);
```

Note that the syntax is slightly different compared to a function type expression — use `:` between the parameter list and the return type rather than `=>`.

Construct Signatures

JavaScript functions can also be invoked with the `new` operator. TypeScript refers to these as *constructors* because they usually create a new object. You can write a *construct signature* by adding the `new` keyword in front of a call signature:

```
type SomeConstructor = {
  new (s: string): SomeObject;
};

function fn(ctor: SomeConstructor) {
  return new ctor("hello");
}
```

Some objects, like JavaScript's `Date` object, can be called with or without `new`. You can combine call and construct signatures in the same type arbitrarily:

```
interface CallOrConstruct {
  (n?: number): string;
  new (s: string): Date;
}

function fn(ctor: CallOrConstruct) {
  // Passing an argument of type `number` to `ctor` matches the first definition
  // Passing an argument of type `string` to `ctor` matches the second definition
  console.log(ctor(10));
  console.log(new ctor("10"));
}

fn(Date);
```

Generic Functions

It's common to write functions where the types of the input relate to the type of the output, or where the types of two inputs are related in some way. Let's consider a function that returns the first element of an array:

```
function firstElement(arr: any[]) {
  return arr[0];
}
```

This function does its job, but has the return type `any`. It's better if the function returned the type of the array element:

```
function firstElement<Type>(arr: Type[]): Type | undefined {
  return arr[0];
}
```

By adding a type parameter `Type` to this function and using it in two places, we've created a link between the input of the function (the array) and the output (the return value). Now, when we call it:

```
const s = firstElement(["a", "b", "c"]); // s is of type 'string'
const n = firstElement([1, 2, 3]); // n is of type 'number'
const u = firstElement([]); // u is of type 'undefined'
```

TypeScript infers the types automatically, which shows how powerful generics can be.

Inference

Note that we didn't have to specify `Type` in this sample. The type was **inferred** — chosen automatically — by TypeScript.

Constraints

Sometimes, you want to relate two values but only operate on a certain subset of values. You can use a *constraint* to limit the kinds of types that a type parameter can accept. For example, a function that returns the longer of two values requires the values to have a `length` property:

```
function longest<Type extends { length: number }>(a: Type, b: Type) {
  if (a.length >= b.length) {
    return a;
  } else {
    return b;
  }
}

// longerArray is of type 'number[]'
const longerArray = longest([1, 2], [1, 2, 3]);

// longerString is of type 'alice' | 'bob'
const longerString = longest("alice", "bob");

// Error! Numbers don't have a 'length' property
const notOK = longest(10, 100);
```

Working with Constrained Values

Beware of certain errors when working with generic constraints. For example:

```
function minimumLength<Type extends { length: number }>(
  obj: Type,
  minimum: number
): Type {
  if (obj.length >= minimum) {
    return obj;
  } else {
    return { length: minimum };  // Error here
  }
}
```

This code appears correct because `Type` is constrained to `{ length: number }`, and the function either returns `Type` or a value matching that constraint. However, the function promises to return the *same* kind of object as was passed in, not just *some* object matching the constraint. Returning `{ length: minimum }` violates the type:

```
const arr = minimumLength([1, 2, 3], 6); // Error: Type '{ length: number }' is not as
console.log(arr.slice(0));
```

Common Errors when Writing Generic Functions

- Declaring unnecessary constraints
- Writing constraints that are too broad or too narrow
- Not relating input and output types properly
- Overusing overloads when union types suffice

For more details and examples, see the full documentation.

Utility Types

TypeScript provides several utility types to facilitate common type transformations. These utilities are available globally.

Awaited<Type>

Released: 4.5

This type is meant to model operations like `await` in `async` functions, or the `.then()` method on `Promise` s - specifically, the way that they recursively unwrap `Promise` s.

Example

```
type A = Awaited<Promise<string>>; // A = string
type B = Awaited<Promise<Promise<number>>>; // B = number
type C = Awaited<boolean | Promise<number>>; // C = number | boolean
```

Try

Partial<Type>

Released: 2.1

Constructs a type with all properties of `Type` set to optional. This utility will return a type that represents all subsets of a given type.

Example

```
interface Todo {
  title: string;
  description: string;
}

function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
  return { ...todo, ...fieldsToUpdate };
}

const todo1 = {
  title: "organize desk",
  description: "clear clutter",
};

const todo2 = updateTodo(todo1, {
  description: "throw out trash",
});
```

Try

Required<Type>

Released: 2.8

Constructs a type consisting of all properties of `Type` set to required. The opposite of `Partial`.

Example

```
interface Props {
  a?: number;
  b?: string;
}

const obj: Props = { a: 5 };
const obj2: Required<Props> = { a: 5 }; // Error: Property 'b' is missing in type '{ a
```

Try

Readonly<Type>

Released: 2.1

Constructs a type with all properties of `Type` set to `readonly`, meaning the properties of the constructed type cannot be reassigned.

Example

```
interface Todo {
  title: string;
}

const todo: Readonly<Todo> = {
  title: "Delete inactive users",
};
todo.title = "Hello"; // Error: Cannot assign to 'title' because it is a read-only pro
```

In this utility, the object's properties become immutable, useful for representing frozen objects.

Example with `Object.freeze`

```
function freeze<Type>(obj: Type): Readonly<Type>;
```

Record<Keys, Type>

Released: 2.1

Constructs an object type whose property keys are `Keys` and whose property values are `Type`. It can be used to map the properties of a type to another type.

Example

```
type CatName = "miffy" | "boris" | "mordred";

interface CatInfo {
  age: number;
  breed: string;
}
```

```
const cats: Record<CatName, CatInfo> = {
  miffy: { age: 10, breed: "Persian" },
  boris: { age: 5, breed: "Maine Coon" },
  mordred: { age: 16, breed: "British Shorthair" },
};
```

Try

Pick<Type, Keys>

Released: 2.1

Constructs a type by selecting a set of properties `Keys` from `Type`.

Example

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

type TodoPreview = Pick<Todo, "title" | "completed">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false,
};
```

Try

404

File not found

The site configured at this address does not contain the requested file.

If this is your site, make sure that the filename case matches the URL as well as any file permissions.
For root URLs (like `http://example.com/`) you must provide an `index.html` file.

Read the full documentation for more information about using **GitHub Pages**.

GitHub Status — @githubstatus

What is a tsconfig.json

Overview

The presence of a `tsconfig.json` file in a directory indicates that the directory is the root of a TypeScript project.
The `tsconfig.json` file specifies the root files and the compiler options required to compile the project.

JavaScript projects can use a `jsconfig.json` file instead, which acts almost the same but has some JavaScript-related compiler flags enabled by default.

A project is compiled in one of the following ways:

- By invoking `tsc` with no input files, in which case the compiler searches for the `tsconfig.json` file starting in the current directory and continuing up the parent directory chain.
- By invoking `tsc` with no input files and a `--project` (or just `-p`) command line option that specifies the path of a directory containing a `tsconfig.json` file, or a path to a valid `.json` file containing the configurations.

When input files are specified on the command line, `tsconfig.json` files are ignored.

Examples

Using the `files` property

```
{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "sourceMap": true
  },
  "files": [
    "core.ts",
    "sys.ts",
    "types.ts",
    "scanner.ts",
    "parser.ts",
    "utilities.ts",
    "binder.ts",
    "checker.ts",
    "emitter.ts",
    "program.ts",
    "commandLineParser.ts",
    "tsc.ts",
    "diagnosticInformationMap.generated.ts"
  ]
}
```

Using the `include` and `exclude` properties

```json
{
  "compilerOptions": {
    "module": "system",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "outFile": "../../built/local/tsc.js",
    "sourceMap": true
  },
  "include": ["src/**/*"],
  "exclude": ["**/*.spec.ts"]
}
```

TSConfig Bases

Depending on the JavaScript runtime environment which you intend to run your code in, there may be a base configuration which you can use at github.com/tsconfig/bases.
These are `tsconfig.json` files which your project extends from, simplifying your `tsconfig.json` by handling runtime support.

For example, if you are writing a project which uses Node.js version 12 and above, you could use the npm module `@tsconfig/node12`:

```json
{
  "extends": "@tsconfig/node12/tsconfig.json",
  "compilerOptions": {
    "preserveConstEnums": true
  },
  "include": ["src/**/*"],
  "exclude": ["**/*.spec.ts"]
}
```

This allows your `tsconfig.json` to focus on your specific choices, not on runtime mechanics. Several bases are available, and the community can add more.

Details

The `compilerOptions` property can be omitted, in which case the compiler's defaults are used. See the full list of supported Compiler Options.

TSConfig Reference

To learn more about the hundreds of configuration options in the TSConfig Reference.

Schema

The `tsconfig.json` Schema can be found at JSON Schema Store.

Help us improve these pages by sending a Pull Request.

Contributors to this page:
OT, LG, JB, L☺, AG, 4+

Last updated: Jun 30, 2025

What is a tsconfig.json

Overview

The presence of a `tsconfig.json` file in a directory indicates that the directory is the root of a TypeScript project. The `tsconfig.json` file specifies the root files and the compiler options required to compile the project.

JavaScript projects can use a `jsconfig.json` file instead, which acts almost the same but has some JavaScript-related compiler flags enabled by default.

A project is compiled in one of the following ways:

- By invoking `tsc` with no input files, in which case the compiler searches for the `tsconfig.json` file starting in the current directory and continuing up the parent directory chain.
- By invoking `tsc` with no input files and a `--project` (or just `-p`) command line option that specifies the path of a directory containing a `tsconfig.json` file, or a path to a valid `.json` file containing the configurations.

When input files are specified on the command line, `tsconfig.json` files are ignored.

Examples

Using the `files` property

```
{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "sourceMap": true
  },
  "files": [
    "core.ts",
    "sys.ts",
    "types.ts",
    "scanner.ts",
    "parser.ts",
    "utilities.ts",
    "binder.ts",
    "checker.ts",
    "emitter.ts",
    "program.ts",
    "commandLineParser.ts",
    "tsc.ts",
    "diagnosticInformationMap.generated.ts"
  ]
}
```

Using the `include` and `exclude` properties

```json
{
  "compilerOptions": {
    "module": "system",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "outFile": "../../built/local/tsc.js",
    "sourceMap": true
  },
  "include": ["src/**/*"],
  "exclude": ["**/*.spec.ts"]
}
```

TSConfig Bases

Depending on the JavaScript runtime environment which you intend to run your code in, there may be a base configuration which you can use at github.com/tsconfig/bases. These are `tsconfig.json` files which your project extends from which simplifies your `tsconfig.json` by handling the runtime support.

For example, if you were writing a project which uses Node.js version 12 and above, then you could use the npm module @tsconfig/node12:

```json
{
  "extends": "@tsconfig/node12/tsconfig.json",
  "compilerOptions": {
    "preserveConstEnums": true
  },
  "include": ["src/**/*"],
  "exclude": ["**/*.spec.ts"]
}
```

This lets your `tsconfig.json` focus on the unique choices for your project, and not all of the runtime mechanics.

Details

The `"compilerOptions"` property can be omitted, in which case the compiler's defaults are used. See the full list of supported Compiler Options.

TSConfig Reference

To learn more about the hundreds of configuration options in the TSConfig Reference.

Schema

The `tsconfig.json` Schema can be found at the JSON Schema Store.

Help improve these pages by sending a Pull Request on GitHub.

Contributors to this page: OT, LG, JB, L☺, AG, 4+
Last updated: Jun 30, 2025

TypeScript: Handbook - Generics

This page has been deprecated

This handbook page has been replaced, go to the new page
Go to new page

Generics

> A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable.
> Components capable of working on current and future data provide the most flexible capabilities for building large systems.

In languages like C# and Java, **generics** are a key tool for creating reusable components—allowing components to operate over a variety of types rather than a single one.
This enables users to consume these components and use their own types.

Hello World of Generics

The simplest example is the identity function, which returns whatever is passed in—similar to the `echo` command.

Without generics, you might define the identity function with a specific type:

```
function identity(arg: number): number {
  return arg;
}
```

Or, using `any` for complete generality:

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

However, `any` loses type information after the function returns.
To preserve type information, use a **type variable:**

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

This `<T>` captures the type of the argument, enabling type-safe operations and preserving precise type info throughout.

Calling the generic identity function

Explicitly specifying the type:

```
let output = identity<string>("myString");
```

Or, allowing TypeScript to infer the type:

```
let output = identity("myString");
```

Working with Generic Type Variables

When creating generic functions like `identity`, TypeScript enforces correct usage of parameters:

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

If you try to access properties not guaranteed by the type variable, TypeScript warns.
For example, adding `.length`:

```
function loggingIdentity<T>(arg: T): T {
  console.log(arg.length); // Error: 'length' does not exist on type 'T'
  return arg;
}
```

To fix this, constrain `T` to types that have `.length`, e.g., arrays:

```
function loggingIdentity<T>(arg: T[]): T[] {
  console.log(arg.length);
  return arg;
}
```

Or, by creating an interface:

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}
```

Using Type Parameters in Generic Constraints

Suppose you want to get a property `K` from object `T`, ensuring `K` is a key of `T`:

```
function getProperty<T, K extends keyof T>(obj: T, key: K) {
  return obj[key];
}
```

This enforces that `key` is one of the valid keys of `obj`.

Using Class Types in Generics

To create instances of classes dynamically:

```
function create<T>(c: { new (): T }): T {
  return new c();
}
```

For example:

```
class BeeKeeper {
  hasMask: boolean;
}
class ZooKeeper {
  nametag: string;
}
class Animal {
  numLegs: number;
}
class Bee extends Animal {
  keeper: BeeKeeper;
}
class Lion extends Animal {
  keeper: ZooKeeper;
}

function createInstance<A extends Animal>(c: { new (): A }): A {
  return new c();
}

createInstance(Lion).keeper.nametag;
createInstance(Bee).keeper.hasMask;
```

Remarks

- **Generics** enhance code reusability and type safety.
- Constraints (`extends`) enable defining bounds for type parameters.
- Class types can be used with generics, but static side members are not generic.
- Properly using type parameters helps prevent runtime errors due to incorrect assumptions about types.

Feedback

The TypeScript docs are open source. Help improve these pages by sending a Pull Request.

Last updated: Jun 30, 2025

404

File not found

The site configured at this address does not contain the requested file.

If this is your site, make sure that the filename case matches the URL as well as any file permissions. For root URLs (like `http://example.com/`) you must provide an `index.html` file.

Read the full documentation for more information about using **GitHub Pages**.

GitHub Status — @githubstatus

Hello

Hello

Hello

Hello

Hello

Hello

Hello

`console.log("Hello")`

Hello

Hello

- Hello
- World

1. Hello
2. World

Hello World

`console.log("Hello")`

TypeScript: Documentation - Gulp

Note: The content primarily consists of a comprehensive tutorial and documentation on using Gulp with TypeScript, including setup steps, build pipelines, module handling, Browserify, Babel, Terser, and Watchify. The above is a converted Markdown translation of that content.