

TYPESCRIPT Docs - English



Table of contents

• En - Nightly Builds	6
• Declaration files - By Example	8
• Declaration files - Consumption	13
• Declaration files - Deep Dive	14
• Declaration files - Do's and Don'ts	18
• Declaration files - Introduction	22
• Declaration files - Library Structures	24
• Declaration files - Publishing	30
• Declaration files - Templates	34
• Templates - Global modifying module.d.ts	35
• Templates - Global plugin.d.ts	37
• Templates - Global.d.ts	43
• Templates - Module class.d.ts	46
• Templates - Module function.d.ts	48
• Templates - Module plugin.d.ts	50
• Templates - Module.d.ts	52
• Get started - TS for Functional Programmers	58
• Get started - TS for JS Programmers	68
• Get started - TS for OOPers	74
• Get started - TS for the New Programmer	79
• Handbook v1 - Basic Types	83
• Handbook v1 - Classes	91
• Handbook v1 - Functions	102
• Handbook v1 - Generics	112
• Handbook v1 - Interfaces	119
• Handbook v1 - Literal Types	132
• Handbook v1 - Unions and Intersections	134
• Handbook v2 - Basics	140
• Handbook v2 - Classes	149
• Handbook v2 - Everyday Types	174
• Handbook v2 - Modules	188
• Handbook v2 - More on Functions	196
• Handbook v2 - Narrowing	212
• Handbook v2 - Object Types	226

• Handbook v2 - The Handbook	242
• Handbook v2 - Type Declarations	244
• Type Manipulation - Conditional Types	247
• Type Manipulation - Generics	252
• Type Manipulation - Indexed Access Types	259
• Type Manipulation - Keyof Type Operator	261
• Type Manipulation - Mapped Types	262
• Type Manipulation - Template Literal Types	265
• Type Manipulation - Typeof Type Operator	270
• Type Manipulation - Creating Types from Types	271
• Handbook v2 - Understanding Errors	272
• Javascript - Creating DTS files From JS	274
• Javascript - Intro to JS with TS	276
• Javascript - JSDoc Reference	278
• Javascript - Type Checking JavaScript Files	292
• Project config - Compiler Options	298
• Project config - Compiler Options in MSBuild	317
• Project config - Configuring Watch	326
• Project config - Integrating with Build Tools	329
• Project config - Project References	334
• Project config - Tsconfig.json	339
• Reference - Advanced Types	342
• Reference - Declaration Merging	362
• Reference - Decorators	368
• Reference - ESM Support for Node	380
• Reference - Enums	385
• Reference - Iterators and Generators	393
• Reference - JSX	395
• Reference - Mixins	404
• Reference - Module Resolution	409
• Reference - Modules	420
• Reference - Namespaces	436
• Reference - Namespaces and Modules	441
• Reference - Symbols	444
• Reference - Triple Slash Directives	447
• Reference - Type Compatibility	450

• Reference - Type Inference	457
• Reference - Utility Types	460
• Reference - Variable Declarations	469
• Release notes - TypeScript 1.1	481
• Release notes - TypeScript 1.3	482
• Release notes - TypeScript 1.4	483
• Release notes - TypeScript 1.5	488
• Release notes - TypeScript 1.6	496
• Release notes - TypeScript 1.7	505
• Release notes - TypeScript 1.8	509
• Release notes - TypeScript 2.0	522
• Release notes - TypeScript 2.1	540
• Release notes - TypeScript 2.2	550
• Release notes - TypeScript 2.3	555
• Release notes - TypeScript 2.4	560
• Release notes - TypeScript 2.5	563
• Release notes - TypeScript 2.6	565
• Release notes - TypeScript 2.7	571
• Release notes - TypeScript 2.8	579
• Release notes - TypeScript 2.9	587
• Release notes - TypeScript 3.0	593
• Release notes - TypeScript 3.1	600
• Release notes - TypeScript 3.2	602
• Release notes - TypeScript 3.3	607
• Release notes - TypeScript 3.4	610
• Release notes - TypeScript 3.5	618
• Release notes - TypeScript 3.6	623
• Release notes - TypeScript 3.7	630
• Release notes - TypeScript 3.8	646
• Release notes - TypeScript 3.9	655
• Release notes - TypeScript 4.0	667
• Release notes - TypeScript 4.1	683
• Release notes - TypeScript 4.2	695
• Release notes - TypeScript 4.3	709
• Release notes - TypeScript 4.4	727
• Release notes - TypeScript 4.5	742

• Release notes - TypeScript 4.6	752
• Release notes - TypeScript 4.7	761
• Release notes - TypeScript 4.8	779
• Release notes - TypeScript 4.9	789
• Tutorials - ASP.NET Core	799
• Tutorials - Angular	804
• Tutorials - Babel with TypeScript	805
• Tutorials - DOM Manipulation	807
• Tutorials - Gulp	812
• Tutorials - Migrating from JavaScript	820
• Tutorials - React	828
• Tutorials - TypeScript Tooling in 5 minutes	829

A nightly build from the [TypeScript's](#) `main` branch is published by midnight PST to npm. Here is how you can get it and use it with your tools.

Using npm

```
npm install -g typescript@next
```

Updating your IDE to use the nightly builds

You can also update your IDE to use the nightly drop. First you will need to install the package through npm. You can either install the npm package globally or to a local `node_modules` folder.

The rest of this section assumes `typescript@next` is already installed.

Visual Studio Code

Update `.vscode/settings.json` with the following:

```
"typescript.tsdk": "<path to your folder>/node_modules/typescript/lib"
```

More information is available at [VSCode documentation](#).

Sublime Text

Update the `Settings - User` file with the following:

```
"typescript_tsdk": "<path to your folder>/node_modules/typescript/lib"
```

More information is available at the [TypeScript Plugin for Sublime Text installation documentation](#).

Visual Studio 2013 and 2015

Note: Most changes do not require you to install a new version of the VS TypeScript plugin.

The nightly build currently does not include the full plugin setup, but we are working on publishing an installer on a nightly basis as well.

1. Download the [VSDevMode.ps1](#) script.

Also see our wiki page on [using a custom language service file](#).

2. From a PowerShell command window, run:

For VS 2015:

```
VSDevMode.ps1 14 -tsScript <path to your folder>/node_modules/typescript/lib
```

For VS 2013:

```
VSDevMode.ps1 12 -tsScript <path to your folder>/node_modules/typescript/lib
```

IntelliJ IDEA (Mac)

Go to Preferences > Languages & Frameworks > TypeScript :

TypeScript Version: If you installed with npm: /usr/local/lib/node_modules/typescript/lib

IntelliJ IDEA (Windows)

Go to File > Settings > Languages & Frameworks > TypeScript :

TypeScript Version: If you installed with npm: C:\Users\USERNAME\AppData\Roaming\npm\node_modules\typescript\lib

[Go to TOC](#)

The purpose of this guide is to teach you how to write a high-quality definition file. This guide is structured by showing documentation for some API, along with sample usage of that API, and explaining how to write the corresponding declaration.

These examples are ordered in approximately increasing order of complexity.

Objects with Properties

Documentation

The global variable `myLib` has a function `makeGreeting` for creating greetings, and a property `numberOfGreetings` indicating the number of greetings made so far.

Code

```
let result = myLib.makeGreeting("hello, world");
console.log("The computed greeting is:" + result);

let count = myLib.numberOfGreetings;
```

Declaration

Use `declare namespace` to describe types or values accessed by dotted notation.

```
declare namespace myLib {
  function makeGreeting(s: string): string;
  let numberOfGreetings: number;
}
```

Overloaded Functions

Documentation

The `getWidget` function accepts a number and returns a Widget, or accepts a string and returns a Widget array.

Code

```
let x: Widget = getWidget(43);
let arr: Widget[] = getWidget("all of them");
```

Declaration

```
declare function getWidget(n: number): Widget;
declare function getWidget(s: string): Widget[];
```

Reusable Types (Interfaces)

Documentation

When specifying a greeting, you must pass a `GreetingSettings` object. This object has the following properties:

- 1 - `greeting`: Mandatory string
- 2 - `duration`: Optional length of time (in milliseconds)
- 3 - `color`: Optional string, e.g. '#ff00ff'

Code

```
greet({
  greeting: "hello world",
  duration: 4000
});
```

Declaration

Use an `interface` to define a type with properties.

```
interface GreetingSettings {
  greeting: string;
  duration?: number;
  color?: string;
}

declare function greet(setting: GreetingSettings): void;
```

Reusable Types (Type Aliases)

Documentation

Anywhere a greeting is expected, you can provide a `string`, a function returning a `string`, or a `Greeter` instance.

Code

```
function getGreeting() {
  return "howdy";
}

class MyGreeter extends Greeter {}
```

```
greet("hello");
greet(getGreeting);
greet(new MyGreeter());
```

Declaration

You can use a type alias to make a shorthand for a type:

```
type GreetingLike = string | () => string | MyGreeter;
declare function greet(g: GreetingLike): void;
```

Organizing Types

Documentation

The `greeter` object can log to a file or display an alert. You can provide LogOptions to `.log(...)` and alert options to `.alert(...)`

Code

```
const g = new Greeter("Hello");
g.log({ verbose: true });
g.alert({ modal: false, title: "Current Greeting" });
```

Declaration

Use namespaces to organize types.

```
declare namespace GreetingLib {
  interface LogOptions {
    verbose?: boolean;
  }
  interface AlertOptions {
    modal: boolean;
    title?: string;
    color?: string;
  }
}
```

You can also create nested namespaces in one declaration:

```
declare namespace GreetingLib.Options {
  // Refer to via GreetingLib.Options.Log
  interface Log {
    verbose?: boolean;
  }
  interface Alert {
    modal: boolean;
    title?: string;
    color?: string;
  }
}
```

Classes

Documentation

You can create a greeter by instantiating the `Greeter` object, or create a customized greeter by extending from it.

Code

```
const myGreeter = new Greeter("hello, world");
myGreeter.greeting = "howdy";
myGreeter.showGreeting();

class SpecialGreeter extends Greeter {
  constructor() {
    super("Very special greetings");
  }
}
```

Declaration

Use `declare class` to describe a class or class-like object. Classes can have properties and methods as well as a constructor.

```
declare class Greeter {
  constructor(greeting: string);

  greeting: string;
  showGreeting(): void;
}
```

Global Variables

Documentation

The global variable `foo` contains the number of widgets present.

Code

```
console.log("Half the number of widgets is " + foo / 2);
```

Declaration

Use `declare var` to declare variables. If the variable is read-only, you can use `declare const`. You can also use `declare let` if the variable is block-scoped.

```
/** The number of widgets present */  
declare var foo: number;
```

Global Functions

Documentation

You can call the function `greet` with a string to show a greeting to the user.

Code

```
greet("hello, world");
```

Declaration

Use `declare function` to declare functions.

```
declare function greet(greeting: string): void;
```

[Go to TOC](#)

Downloading

Getting type declarations requires no tools apart from npm.

As an example, getting the declarations for a library like lodash takes nothing more than the following command

```
npm install --save-dev @types/lodash
```

It is worth noting that if the npm package already includes its declaration file as described in [Publishing](#), downloading the corresponding `@types` package is not needed.

Consuming

From there you'll be able to use lodash in your TypeScript code with no fuss. This works for both modules and global code.

For example, once you've `npm install`-ed your type declarations, you can use imports and write

```
import * as _ from "lodash";
_.padStart("Hello TypeScript!", 20, " ");
```

or if you're not using modules, you can just use the global variable `_`.

```
_.padStart("Hello TypeScript!", 20, " ");
```

Searching

For the most part, type declaration packages should always have the same name as the package name on `npm`, but prefixed with `@types/`, but if you need, you can check out [this Type Search](#) to find the package for your favorite library.

Note: if the declaration file you are searching for is not present, you can always contribute one back and help out the next developer looking for it. Please see the DefinitelyTyped [contribution guidelines page](#) for details.

[Go to TOC](#)

Declaration File Theory: A Deep Dive

Structuring modules to give the exact API shape you want can be tricky. For example, we might want a module that can be invoked with or without `new` to produce different types, has a variety of named types exposed in a hierarchy, and has some properties on the module object as well.

By reading this guide, you'll have the tools to write complex declaration files that expose a friendly API surface. This guide focuses on module (or UMD) libraries because the options here are more varied.

Key Concepts

You can fully understand how to make any shape of declaration by understanding some key concepts of how TypeScript works.

Types

If you're reading this guide, you probably already roughly know what a type in TypeScript is. To be more explicit, though, a *type* is introduced with:

- A type alias declaration (`type sn = number | string;`)
- An interface declaration (`interface I { x: number[]; }`)
- A class declaration (`class C { }`)
- An enum declaration (`enum E { A, B, C }`)
- An `import` declaration which refers to a type

Each of these declaration forms creates a new type name.

Values

As with types, you probably already understand what a value is. Values are runtime names that we can reference in expressions. For example `let x = 5;` creates a value called `x`.

Again, being explicit, the following things create values:

- `let`, `const`, and `var` declarations
- A `namespace` or `module` declaration which contains a value
- An `enum` declaration
- A `class` declaration
- An `import` declaration which refers to a value
- A `function` declaration

Namespaces

Types can exist in *namespaces*. For example, if we have the declaration `let x: A.B.C`, we say that the type `C` comes from the `A.B` namespace.

This distinction is subtle and important -- here, `A.B` is not necessarily a type or a value.

Simple Combinations: One name, multiple meanings

Given a name `A`, we might find up to three different meanings for `A`: a type, a value or a namespace. How the name is interpreted depends on the context in which it is used. For example, in the declaration `let m: A.A = A;`, `A` is used first as a namespace, then as a type name, then as a value. These meanings might end up referring to entirely different declarations!

This may seem confusing, but it's actually very convenient as long as we don't excessively overload things. Let's look at some useful aspects of this combining behavior.

Built-in Combinations

Astute readers will notice that, for example, `class` appeared in both the *type* and *value* lists. The declaration `class C { }` creates two things: a *type* `C` which refers to the instance shape of the class, and a *value* `C` which refers to the constructor function of the class. Enum declarations behave similarly.

User Combinations

Let's say we wrote a module file `foo.d.ts`:

```
export var SomeVar: { a: SomeType };
export interface SomeType {
  count: number;
}
```

Then consumed it:

```
import * as foo from "./foo";
let x: foo.SomeType = foo.SomeVar.a;
console.log(x.count);
```

This works well enough, but we might imagine that `SomeType` and `SomeVar` were very closely related such that you'd like them to have the same name. We can use combining to present these two different objects (the value and the type) under the same name `Bar`:

```
export var Bar: { a: Bar };
export interface Bar {
  count: number;
}
```

This presents a very good opportunity for destructuring in the consuming code:

```
import { Bar } from "./foo";
let x: Bar = Bar.a;
console.log(x.count);
```

Again, we've used `Bar` as both a type and a value here. Note that we didn't have to declare the `Bar` value as being of the `Bar` type -- they're independent.

Advanced Combinations

Some kinds of declarations can be combined across multiple declarations. For example, `class C {}` and `interface C {}` can co-exist and both contribute properties to the `C` types.

This is legal as long as it does not create a conflict. A general rule of thumb is that values always conflict with other values of the same name unless they are declared as `namespace`s, types will conflict if they are declared with a type alias declaration (`type s = string`), and namespaces never conflict.

Let's see how this can be used.

Adding using an `interface`

We can add additional members to an `interface` with another `interface` declaration:

```
interface Foo {
  x: number;
}
// ... elsewhere ...
interface Foo {
  y: number;
}
let a: Foo = ...;
console.log(a.x + a.y); // OK
```

This also works with classes:

```
class Foo {
  x: number;
}
// ... elsewhere ...
interface Foo {
  y: number;
}
let a: Foo = ...;
console.log(a.x + a.y); // OK
```

Note that we cannot add to type aliases (`type s = string;`) using an interface.

Adding using a `namespace`

A `namespace` declaration can be used to add new types, values, and namespaces in any way which does not create a conflict.

For example, we can add a static member to a class:

```
class C {}
// ... elsewhere ...
namespace C {
  export let x: number;
}
let y = C.x; // OK
```

Note that in this example, we added a value to the *static* side of `c` (its constructor function). This is because we added a *value*, and the container for all values is another value (types are contained by namespaces, and namespaces are contained by other namespaces).

We could also add a namespaced type to a class:

```
class C {}
// ... elsewhere ...
namespace C {
  export interface D {}
}
let y: C.D; // OK
```

In this example, there wasn't a namespace `C` until we wrote the `namespace` declaration for it. The meaning `C` as a namespace doesn't conflict with the value or type meanings of `C` created by the class.

Finally, we could perform many different merges using `namespace` declarations. This isn't a particularly realistic example, but shows all sorts of interesting behavior:

```
namespace X {
  export interface Y {}
  export class Z {}
}

// ... elsewhere ...
namespace X {
  export var Y: number;
  export namespace Z {
    export class C {}
  }
}
type X = string;
```

In this example, the first block creates the following name meanings:

- A value `X` (because the `namespace` declaration contains a value, `Z`)
- A namespace `X` (because the `namespace` declaration contains a type, `Y`)
- A type `Y` in the `X` namespace
- A type `Z` in the `X` namespace (the instance shape of the class)
- A value `Z` that is a property of the `X` value (the constructor function of the class)

The second block creates the following name meanings:

- A value `Y` (of type `number`) that is a property of the `X` value
- A namespace `Z`
- A value `Z` that is a property of the `X` value
- A type `C` in the `X.Z` namespace
- A value `C` that is a property of the `X.Z` value
- A type `X`

[Go to TOC](#)

General Types

Number, String, Boolean, Symbol and Object

✗ Don't ever use the types `Number`, `String`, `Boolean`, `Symbol`, or `Object`. These types refer to non-primitive boxed objects that are almost never used appropriately in JavaScript code.

```
/* WRONG */
function reverse(s: String): String;
```

✓ Do use the types `number`, `string`, `boolean`, and `symbol`.

```
/* OK */
function reverse(s: string): string;
```

Instead of `Object`, use the non-primitive `object` type ([added in TypeScript 2.2](#)).

Generics

✗ Don't ever have a generic type which doesn't use its type parameter. See more details in [TypeScript FAQ page](#).

any

✗ Don't use `any` as a type unless you are in the process of migrating a JavaScript project to TypeScript. The compiler *effectively* treats `any` as "please turn off type checking for this thing". It is similar to putting an `@ts-ignore` comment around every usage of the variable. This can be very helpful when you are first migrating a JavaScript project to TypeScript as you can set the type for stuff you haven't migrated yet as `any`, but in a full TypeScript project you are disabling type checking for any parts of your program that use it.

In cases where you don't know what type you want to accept, or when you want to accept anything because you will be blindly passing it through without interacting with it, you can use `unknown`.

Callback Types

Return Types of Callbacks

✗ Don't use the return type `any` for callbacks whose value will be ignored:

```
/* WRONG */
function fn(x: () => any) {
  x();
}
```

✓ Do use the return type `void` for callbacks whose value will be ignored:

```
/* OK */
function fn(x: () => void) {
  x();
}
```

⚠ **Why:** Using `void` is safer because it prevents you from accidentally using the return value of `x` in an unchecked way:

```
function fn(x: () => void) {
  var k = x(); // oops! meant to do something else
  k.doSomething(); // error, but would be OK if the return type had been 'any'
}
```

Optional Parameters in Callbacks

✗ **Don't** use optional parameters in callbacks unless you really mean it:

```
/* WRONG */
interface Fetcher {
  getObjet(done: (data: unknown, elapsedTime?: number) => void): void;
}
```

This has a very specific meaning: the `done` callback might be invoked with 1 argument or might be invoked with 2 arguments. The author probably intended to say that the callback might not care about the `elapsedTime` parameter, but there's no need to make the parameter optional to accomplish this -- it's always legal to provide a callback that accepts fewer arguments.

✓ **Do** write callback parameters as non-optional:

```
/* OK */
interface Fetcher {
  getObjet(done: (data: unknown, elapsedTime: number) => void): void;
}
```

Overloads and Callbacks

✗ **Don't** write separate overloads that differ only on callback arity:

```
/* WRONG */
declare function beforeAll(action: () => void, timeout?: number): void;
declare function beforeAll(
  action: (done: DoneFn) => void,
  timeout?: number
): void;
```

✓ **Do** write a single overload using the maximum arity:

```
/* OK */
declare function beforeAll(
  action: (done: DoneFn) => void,
  timeout?: number
): void;
```

? **Why:** It's always legal for a callback to disregard a parameter, so there's no need for the shorter overload. Providing a shorter callback first allows incorrectly-typed functions to be passed in because they match the first overload.

Function Overloads

Ordering

X **Don't** put more general overloads before more specific overloads:

```
/* WRONG */
declare function fn(x: unknown): unknown;
declare function fn(x: HTMLElement): number;
declare function fn(x: HTMLDivElement): string;

var myElem: HTMLDivElement;
var x = fn(myElem); // x: unknown, wat?
```

✓ **Do** sort overloads by putting the more general signatures after more specific signatures:

```
/* OK */
declare function fn(x: HTMLDivElement): string;
declare function fn(x: HTMLElement): number;
declare function fn(x: unknown): unknown;

var myElem: HTMLDivElement;
var x = fn(myElem); // x: string, :)
```

? **Why:** TypeScript chooses the *first matching overload* when resolving function calls. When an earlier overload is "more general" than a later one, the later one is effectively hidden and cannot be called.

Use Optional Parameters

X **Don't** write several overloads that differ only in trailing parameters:

```
/* WRONG */
interface Example {
  diff(one: string): number;
  diff(one: string, two: string): number;
  diff(one: string, two: string, three: boolean): number;
}
```

✓ **Do** use optional parameters whenever possible:

```
/* OK */
interface Example {
  diff(one: string, two?: string, three?: boolean): number;
}
```

Note that this collapsing should only occur when all overloads have the same return type.

? **Why:** This is important for two reasons.

TypeScript resolves signature compatibility by seeing if any signature of the target can be invoked with the arguments of the source, *and extraneous arguments are allowed*. This code, for example, exposes a bug only when the signature is correctly written using optional parameters:

```
function fn(x: (a: string, b: number, c: number) => void) {}
var x: Example;
// When written with overloads, OK -- used first overload
// When written with optionals, correctly an error
fn(x.diff);
```

The second reason is when a consumer uses the "strict null checking" feature of TypeScript. Because unspecified parameters appear as `undefined` in JavaScript, it's usually fine to pass an explicit `undefined` to a function with optional arguments. This code, for example, should be OK under strict nulls:

```
var x: Example;
// When written with overloads, incorrectly an error because of passing
'undefined' to 'string'
// When written with optionals, correctly OK
x.diff("something", true ? undefined : "hour");
```

Use Union Types

✗ Don't write overloads that differ by type in only one argument position:

```
/* WRONG */
interface Moment {
  utcOffset(): number;
  utcOffset(b: number): Moment;
  utcOffset(b: string): Moment;
}
```

✓ Do use union types whenever possible:

```
/* OK */
interface Moment {
  utcOffset(): number;
  utcOffset(b: number | string): Moment;
}
```

Note that we didn't make `b` optional here because the return types of the signatures differ.

? Why: This is important for people who are "passing through" a value to your function:

```
function fn(x: string): void;
function fn(x: number): void;
function fn(x: number | string) {
  // When written with separate overloads, incorrectly an error
  // When written with union types, correctly OK
  return moment().utcOffset(x);
}
```

[Go to TOC](#)

The Declaration Files section is designed to teach you how to write a high-quality TypeScript Declaration File. We need to assume basic familiarity with the TypeScript language in order to get started.

If you haven't already, you should read the [TypeScript Handbook](#) to familiarize yourself with basic concepts, especially types and modules.

The most common case for learning how .d.ts files work is that you're typing an npm package with no types. In that case, you can jump straight to [Modules .d.ts](#).

The Declaration Files section is broken down into the following sections.

Declaration Reference

We are often faced with writing a declaration file when we only have examples of the underlying library to guide us. The [Declaration Reference](#) section shows many common API patterns and how to write declarations for each of them. This guide is aimed at the TypeScript novice who may not yet be familiar with every language construct in TypeScript.

Library Structures

The [Library Structures](#) guide helps you understand common library formats and how to write a proper declaration file for each format. If you're editing an existing file, you probably don't need to read this section. Authors of new declaration files are strongly encouraged to read this section to properly understand how the format of the library influences the writing of the declaration file.

In the [Template](#) section you'll find a number of declaration files that serve as a useful starting point when writing a new file. If you already know what your structure is, see the d.ts Template section in the sidebar.

Do's and Don'ts

Many common mistakes in declaration files can be easily avoided. The [Do's and Don'ts](#) section identifies common errors, describes how to detect them, and how to fix them. Everyone should read this section to help themselves avoid common mistakes.

Deep Dive

For seasoned authors interested in the underlying mechanics of how declaration files work, the [Deep Dive](#) section explains many advanced concepts in declaration writing, and shows how to leverage these concepts to create cleaner and more intuitive declaration files.

Publish to npm

The [Publishing](#) section explains how to publish your declaration files to an npm package, and shows how to manage your dependent packages.

Find and Install Declaration Files

For JavaScript library users, the [Consumption](#) section offers a few simple steps to locate and install corresponding declaration files.

[Go to TOC](#)

Broadly speaking, the way you *structure* your declaration file depends on how the library is consumed. There are many ways of offering a library for consumption in JavaScript, and you'll need to write your declaration file to match it. This guide covers how to identify common library patterns, and how to write declaration files which correspond to that pattern.

Each type of major library structuring pattern has a corresponding file in the [Templates](#) section. You can start with these templates to help you get going faster.

Identifying Kinds of Libraries

First, we'll review the kinds of libraries TypeScript declaration files can represent. We'll briefly show how each kind of library is *used*, how it is *written*, and list some example libraries from the real world.

Identifying the structure of a library is the first step in writing its declaration file. We'll give hints on how to identify structure both based on its *usage* and its *code*. Depending on the library's documentation and organization, one might be easier than the other. We recommend using whichever is more comfortable to you.

What should you look for?

Question to ask yourself while looking at a library you are trying to type.

1. How do you obtain the library?

For example, can you *only* get it through npm or only from a CDN?

2. How would you import it?

Does it add a global object? Does it use `require` or `import / export` statements?

Smaller samples for different types of libraries

Modular Libraries

Almost every modern Node.js library falls into the module family. These type of libraries only work in a JS environment with a module loader. For example, `express` only works in Node.js and must be loaded using the CommonJS `require` function.

ECMAScript 2015 (also known as ES2015, ECMAScript 6, and ES6), CommonJS, and RequireJS have similar notions of *importing a module*. In JavaScript CommonJS (Node.js), for example, you would write

```
var fs = require("fs");
```

In TypeScript or ES6, the `import` keyword serves the same purpose:

```
import * as fs from "fs";
```

You'll typically see modular libraries include one of these lines in their documentation:

```
var someLib = require("someLib");
```

or

```
define(..., ['someLib'], function(someLib) {
});
```

As with global modules, you might see these examples in the documentation of a UMD module, so be sure to check the code or documentation.

Identifying a Module Library from Code

Modular libraries will typically have at least some of the following:

- Unconditional calls to `require` or `define`
- Declarations like `import * as a from 'b'`; or `export c;`
- Assignments to `exports` or `module.exports`

They will rarely have:

- Assignments to properties of `window` or `global`

Templates For Modules

There are four templates available for modules, `module.d.ts`, `module-class.d.ts`, `module-function.d.ts` and `module-plugin.d.ts`.

You should first read `module.d.ts` for an overview on the way they all work.

Then use the template `module-function.d.ts` if your module can be *called* like a function:

```
const x = require("foo");
// Note: calling 'x' as a function
const y = x(42);
```

Use the template `module-class.d.ts` if your module can be *constructed* using `new`:

```
const x = require("bar");
// Note: using 'new' operator on the imported variable
const y = new x("hello");
```

If you have a module which when imported, makes changes to other modules use template `module-plugin.d.ts`:

```
const jest = require("jest");
require("jest-matchers-files");
```

Global Libraries

A *global* library is one that can be accessed from the global scope (i.e. without using any form of `import`). Many libraries simply expose one or more global variables for use. For example, if you were using `jQuery`, the `$` variable can be used by simply referring to it:

```
$(() => {
  console.log("hello!");
});
```

You'll usually see guidance in the documentation of a global library of how to use the library in an HTML script tag:

```
<script src="http://a.great.cdn.for/someLib.js"></script>
```

Today, most popular globally-accessible libraries are actually written as UMD libraries (see below). UMD library documentation is hard to distinguish from global library documentation. Before writing a global declaration file, make sure the library isn't actually UMD.

Identifying a Global Library from Code

Global library code is usually extremely simple. A global "Hello, world" library might look like this:

```
function createGreeting(s) {
  return "Hello, " + s;
}
```

or like this:

```
// Web
window.createGreeting = function (s) {
  return "Hello, " + s;
};

// Node
global.createGreeting = function (s) {
  return "Hello, " + s;
};

// Potentially any runtime
globalThis.createGreeting = function (s) {
  return "Hello, " + s;
};
```

When looking at the code of a global library, you'll usually see:

- Top-level `var` statements or `function` declarations
- One or more assignments to `window.someName`
- Assumptions that DOM primitives like `document` or `window` exist

You *won't* see:

- Checks for, or usage of, module loaders like `require` or `define`

- CommonJS/Node.js-style imports of the form `var fs = require("fs");`
- Calls to `define(...)`
- Documentation describing how to `require` or import the library

Examples of Global Libraries

Because it's usually easy to turn a global library into a UMD library, very few popular libraries are still written in the global style. However, libraries that are small and require the DOM (or have *no* dependencies) may still be global.

Global Library Template

The template file `global.d.ts` defines an example library `myLib`. Be sure to read the "[Preventing Name Conflicts](#)" footnote.

UMD

A *UMD* module is one that can *either* be used as module (through an import), or as a global (when run in an environment without a module loader). Many popular libraries, such as [Moment.js](#), are written this way. For example, in Node.js or using RequireJS, you would write:

```
import moment = require("moment");
console.log(moment.format());
```

whereas in a vanilla browser environment you would write:

```
console.log(moment.format());
```

Identifying a UMD library

[UMD modules](#) check for the existence of a module loader environment. This is an easy-to-spot pattern that looks something like this:

```
(function (root, factory) {
  if (typeof define === "function" && define.amd) {
    define(["libName"], factory);
  } else if (typeof module === "object" && module.exports) {
    module.exports = factory(require("libName"));
  } else {
    root.returnExports = factory(root.libName);
  }
})(this, function (b) {
```

If you see tests for `typeof define`, `typeof window`, or `typeof module` in the code of a library, especially at the top of the file, it's almost always a UMD library.

Documentation for UMD libraries will also often demonstrate a "Using in Node.js" example showing `require`, and a "Using in the browser" example showing using a `<script>` tag to load the script.

Examples of UMD libraries

Most popular libraries are now available as UMD packages. Examples include [jQuery](#), [Moment.js](#), [lodash](#), and many more.

Template

Use the `module-plugin.d.ts` template.

Consuming Dependencies

There are several kinds of dependencies your library might have. This section shows how to import them into the declaration file.

Dependencies on Global Libraries

If your library depends on a global library, use a `/// <reference types="..." />` directive:

```
/// <reference types="someLib" />
function getThing(): someLib.thing;
```

Dependencies on Modules

If your library depends on a module, use an `import` statement:

```
import * as moment from "moment";
function getThing(): moment;
```

Dependencies on UMD libraries

From a Global Library

If your global library depends on a UMD module, use a `/// <reference types` directive:

```
/// <reference types="moment" />
function getThing(): moment;
```

From a Module or UMD Library

If your module or UMD library depends on a UMD library, use an `import` statement:

```
import * as someLib from "someLib";
```

Do not use a `/// <reference` directive to declare a dependency to a UMD library!

Footnotes

Preventing Name Conflicts

Note that it's possible to define many types in the global scope when writing a global declaration file. We strongly discourage this as it leads to possible unresolvable name conflicts when many declaration files are in a project.

A simple rule to follow is to only declare types *namespaced* by whatever global variable the library defines. For example, if the library defines the global value 'cats', you should write

```
declare namespace cats {
    interface KittySettings {}
}
```

But *not*

```
// at top-level
interface CatsKittySettings {}
```

This guidance also ensures that the library can be transitioned to UMD without breaking declaration file users.

The Impact of ES6 on Module Call Signatures

Many popular libraries, such as Express, expose themselves as a callable function when imported. For example, the typical Express usage looks like this:

```
import exp = require("express");
var app = exp();
```

In ES6-compliant module loaders, the top-level object (here imported as `exp`) can only have properties; the top-level module object can *never* be callable.

The most common solution here is to define a `default` export for a callable/constructable object; module loaders commonly detect this situation automatically and replace the top-level object with the `default` export. TypeScript can handle this for you, if you have `"esModuleInterop": true` in your `tsconfig.json`.

[Go to TOC](#)

Now that you have authored a declaration file following the steps of this guide, it is time to publish it to npm. There are two main ways you can publish your declaration files to npm:

1. bundling with your npm package
2. publishing to the [@types organization](#) on npm.

If your types are generated by your source code, publish the types with your source code. Both TypeScript and JavaScript projects can generate types via `declaration`.

Otherwise, we recommend submitting the types to DefinitelyTyped, which will publish them to the [@types](#) organization on npm.

Including declarations in your npm package

If your package has a main `.js` file, you will need to indicate the main declaration file in your `package.json` file as well. Set the `types` property to point to your bundled declaration file. For example:

```
{
  "name": "awesome",
  "author": "Vandelay Industries",
  "version": "1.0.0",
  "main": "./lib/main.js",
  "types": "./lib/main.d.ts"
}
```

Note that the `" typings "` field is synonymous with `types`, and could be used as well.

Also note that if your main declaration file is named `index.d.ts` and lives at the root of the package (next to `index.js`) you do not need to mark the `types` property, though it is advisable to do so.

Dependencies

All dependencies are managed by npm. Make sure all the declaration packages you depend on are marked appropriately in the `"dependencies"` section in your `package.json`. For example, imagine we authored a package that used Browserify and TypeScript.

```
{
  "name": "browserify-typescript-extension",
  "author": "Vandelay Industries",
  "version": "1.0.0",
  "main": "./lib/main.js",
  "types": "./lib/main.d.ts",
  "dependencies": {
    "browserify": "latest",
    "@types/browserify": "latest",
    "typescript": "next"
  }
}
```

Here, our package depends on the `browserify` and `typescript` packages. `browserify` does not bundle its declaration files with its npm packages, so we needed to depend on `@types/browserify` for its declarations. `typescript`, on the other hand, packages its declaration files, so there was no need for any additional dependencies.

Our package exposes declarations from each of those, so any user of our `browserify-typescript-extension` package needs to have these dependencies as well. For that reason, we used `"dependencies"` and not `"devDependencies"`, because otherwise our consumers would have needed to manually install those packages. If we had just written a command line application and not expected our package to be used as a library, we might have used `devDependencies`.

Red flags

```
/// <reference path="..." />
```

Don't use `/// <reference path="..." />` in your declaration files.

```
/// <reference path="../typescript/lib/typescriptServices.d.ts" />
....
```

Do use `/// <reference types="typescript" />` instead.

```
/// <reference types="typescript" />
....
```

Make sure to revisit the [Consuming dependencies](#) section for more information.

Packaging dependent declarations

If your type definitions depend on another package:

- *Don't* combine it with yours, keep each in their own file.
- *Don't* copy the declarations in your package either.
- *Do* depend on the npm type declaration package if it doesn't package its declaration files.

Version selection with `typesVersions`

When TypeScript opens a `package.json` file to figure out which files it needs to read, it first looks at a field called `typesVersions`.

Folder redirects (using *)

A `package.json` with a `typesVersions` field might look like this:

```
{
  "name": "package-name",
  "version": "1.0.0",
  "types": "./index.d.ts",
  "typesVersions": {
```

```

    ">=3.1": { "*": ["ts3.1/*"] }
}

```

This `package.json` tells TypeScript to first check the current version of TypeScript. If it's 3.1 or later, TypeScript figures out the path you've imported relative to the package, and reads from the package's `ts3.1` folder.

That's what that `{ "*": ["ts3.1/*"] }` means - if you're familiar with [path mapping](#), it works exactly like that.

In the above example, if we're importing from `"package-name"`, TypeScript will try to resolve from `[...]/node_modules/package-name/ts3.1/index.d.ts` (and other relevant paths) when running in TypeScript 3.1. If we import from `package-name/foo`, we'll try to look for `[...]/node_modules/package-name/ts3.1/foo.d.ts` and `[...]/node_modules/package-name/ts3.1/foo/index.d.ts`.

What if we're not running in TypeScript 3.1 in this example? Well, if none of the fields in `typesVersions` get matched, TypeScript falls back to the `types` field, so here TypeScript 3.0 and earlier will be redirected to `[...]/node_modules/package-name/index.d.ts`.

File redirects

When you want to only change the resolution for a single file at a time, you can tell TypeScript the file to resolve differently by passing in the exact filenames:

```

{
  "name": "package-name",
  "version": "1.0.0",
  "types": "./index.d.ts",
  "typesVersions": {
    "<4.0": { "index.d.ts": ["index.v3.d.ts"] }
  }
}

```

On TypeScript 4.0 and above, an import for `"package-name"` would resolve to `./index.d.ts` and for 3.9 and below `./index.v3.d.ts`.

Matching behavior

The way that TypeScript decides on whether a version of the compiler & language matches is by using Node's [semver ranges](#).

Multiple fields

`typesVersions` can support multiple fields where each field name is specified by the range to match on.

```

{
  "name": "package-name",
  "version": "1.0",
  "types": "./index.d.ts",
  "typesVersions": {
    ...
  }
}

```

```
    ">=3.2": { "*": ["ts3.2/*"] },
    ">=3.1": { "*": ["ts3.1/*"] }
}
```

Since ranges have the potential to overlap, determining which redirect applies is order-specific. That means in the above example, even though both the `>=3.2` and the `>=3.1` matchers support TypeScript 3.2 and above, reversing the order could have different behavior, so the above sample would not be equivalent to the following.

```
{
  "name": "package-name",
  "version": "1.0",
  "types": "./index.d.ts",
  "typesVersions": {
    // NOTE: this doesn't work!
    ">=3.1": { "*": ["ts3.1/*"] },
    ">=3.2": { "*": ["ts3.2/*"] }
}
```

Publish to [@types](#)

Packages under the [@types](#) organization are published automatically from [DefinitelyTyped](#) using the [types-publisher tool](#). To get your declarations published as an @types package, please submit a pull request to [DefinitelyTyped](#). You can find more details in the [contribution guidelines page](#).

[Go to TOC](#)

global-modifying-module.d.ts

- [global-plugin.d.ts](#)
- [global.d.ts](#)
- [module-class.d.ts](#)
- [module-function.d.ts](#)
- [module-plugin.d.ts](#)
- [module.d.ts](#)

[Go to TOC](#)

Global-modifying Modules

A *global-modifying module* alters existing values in the global scope when they are imported. For example, there might exist a library which adds new members to `String.prototype` when imported. This pattern is somewhat dangerous due to the possibility of runtime conflicts, but we can still write a declaration file for it.

Identifying global-modifying modules

Global-modifying modules are generally easy to identify from their documentation. In general, they're similar to global plugins, but need a `require` call to activate their effects.

You might see documentation like this:

```
// 'require' call that doesn't use its return value
var unused = require("magic-string-time");
/* or */
require("magic-string-time");

var x = "hello, world";
// Creates new methods on built-in types
console.log(x.startsWithHello());

var y = [1, 2, 3];
// Creates new methods on built-in types
console.log(y.reverseAndSort());
```

Here is an example

```
// Type definitions for [~THE LIBRARY NAME~] [~OPTIONAL VERSION NUMBER~]
// Project: [~THE PROJECT NAME~]
// Definitions by: [~YOUR NAME~] <[~A URL FOR YOU~]>

/*~ This is the global-modifying module template file. You should rename it to
index.d.ts
*~ and place it in a folder with the same name as the module.
*~ For example, if you were writing a file for "super-greeter", this
*~ file should be 'super-greeter/index.d.ts'
*/

/*~ Note: If your global-modifying module is callable or constructable, you'll
*~ need to combine the patterns here with those in the module-class or module-
function
*~ template files
*/
declare global {
  /*~ Here, declare things that go in the global namespace, or augment
  *~ existing declarations in the global namespace
  */
  interface String {
    fancyFormat(opts: StringFormatOptions): string;
  }
}

/*~ If your module exports types or values, write them as usual */
export interface StringFormatOptions {
  fancinessLevel: number;
```

```
}

/*~ For example, declaring a method on the module (in addition to its global side
effects) */
export function doSomething(): void;

/*~ If your module exports nothing, you'll need this line. Otherwise, delete it */
export {};
```

[Go to TOC](#)

UMD

A UMD module is one that can either be used as module (through an import), or as a global (when run in an environment without a module loader). Many popular libraries, such as [Moment.js](#), are written this way. For example, in Node.js or using RequireJS, you would write:

```
import moment = require("moment");
console.log(moment.format());
```

whereas in a vanilla browser environment you would write:

```
console.log(moment.format());
```

Identifying a UMD library

UMD modules check for the existence of a module loader environment. This is an easy-to-spot pattern that looks something like this:

```
(function (root, factory) {
  if (typeof define === "function" && define.amd) {
    define(["libName"], factory);
  } else if (typeof module === "object" && module.exports) {
    module.exports = factory(require("libName"));
  } else {
    root.returnExports = factory(root.libName);
  }
})(this, function (b) {
```

If you see tests for `typeof define`, `typeof window`, or `typeof module` in the code of a library, especially at the top of the file, it's almost always a UMD library.

Documentation for UMD libraries will also often demonstrate a "Using in Node.js" example showing `require`, and a "Using in the browser" example showing using a `<script>` tag to load the script.

Examples of UMD libraries

Most popular libraries are now available as UMD packages. Examples include [jQuery](#), [Moment.js](#), [Lodash](#), and many more.

Template

There are three templates available for modules, `module.d.ts`, `module-class.d.ts` and `module-function.d.ts`.

Use `module-function.d.ts` if your module can be *called* like a function:

```
var x = require("foo");
// Note: calling 'x' as a function
var y = x(42);
```

Be sure to read the [footnote "The Impact of ES6 on Module Call Signatures"](#)

Use `module-class.d.ts` if your module can be *constructed* using `new`:

```
var x = require("bar");
// Note: using 'new' operator on the imported variable
var y = new x("hello");
```

The same [footnote](#) applies to these modules.

If your module is not callable or constructable, use the `module.d.ts` file.

Module Plugin or UMD Plugin

A *module plugin* changes the shape of another module (either UMD or module). For example, in `Moment.js`, `moment-range` adds a new `range` method to the `moment` object.

For the purposes of writing a declaration file, you'll write the same code whether the module being changed is a plain module or UMD module.

Template

Use the `module-plugin.d.ts` template.

Global Plugin

A *global plugin* is global code that changes the shape of some global. As with *global-modifying modules*, these raise the possibility of runtime conflict.

For example, some libraries add new functions to `Array.prototype` or `String.prototype`.

Identifying global plugins

Global plugins are generally easy to identify from their documentation.

You'll see examples that look like this:

```
var x = "hello, world";
// Creates new methods on built-in types
console.log(x.startsWithHello());

var y = [1, 2, 3];
// Creates new methods on built-in types
console.log(y.reverseAndSort());
```

Template

Use the `global-plugin.d.ts` template.

Global-modifying Modules

A *global-modifying module* alters existing values in the global scope when they are imported. For example, there might exist a library which adds new members to `String.prototype` when imported. This pattern is somewhat dangerous due to the possibility of runtime conflicts, but we can still write a declaration file for it.

Identifying global-modifying modules

Global-modifying modules are generally easy to identify from their documentation. In general, they're similar to global plugins, but need a `require` call to activate their effects.

You might see documentation like this:

```
// 'require' call that doesn't use its return value
var unused = require("magic-string-time");
/* or */
require("magic-string-time");

var x = "hello, world";
// Creates new methods on built-in types
console.log(x.startsWithHello());

var y = [1, 2, 3];
// Creates new methods on built-in types
console.log(y.reverseAndSort());
```

Template

Use the `global-modifying-module.d.ts` template.

Consuming Dependencies

There are several kinds of dependencies your library might have. This section shows how to import them into the declaration file.

Dependencies on Global Libraries

If your library depends on a global library, use a `/// <reference types="..." />` directive:

```
/// <reference types="someLib" />
function getThing(): someLib.thing;
```

Dependencies on Modules

If your library depends on a module, use an `import` statement:

```
import * as moment from "moment";
function getThing(): moment;
```

Dependencies on UMD libraries

From a Global Library

If your global library depends on a UMD module, use a `/// <reference types` directive:

```
//> <reference types="moment" />
function getThing(): moment;
```

From a Module or UMD Library

If your module or UMD library depends on a UMD library, use an `import` statement:

```
import * as someLib from "someLib";
```

Do *not* use a `/// <reference` directive to declare a dependency to a UMD library!

Footnotes

Preventing Name Conflicts

Note that it's possible to define many types in the global scope when writing a global declaration file. We strongly discourage this as it leads to possible unresolvable name conflicts when many declaration files are in a project.

A simple rule to follow is to only declare types *namespaced* by whatever global variable the library defines. For example, if the library defines the global value 'cats', you should write

```
declare namespace cats {
    interface KittySettings {}
}
```

But *not*

```
// at top-level
interface CatsKittySettings {}
```

This guidance also ensures that the library can be transitioned to UMD without breaking declaration file users.

The Impact of ES6 on Module Plugins

Some plugins add or modify top-level exports on existing modules. While this is legal in CommonJS and other loaders, ES6 modules are considered immutable and this pattern will not be possible. Because TypeScript is loader-agnostic, there is no compile-time enforcement of this policy, but developers intending to transition to an ES6 module loader should be aware of this.

The Impact of ES6 on Module Call Signatures

Many popular libraries, such as Express, expose themselves as a callable function when imported. For example, the typical Express usage looks like this:

```
import exp = require("express");
var app = exp();
```

In ES6 module loaders, the top-level object (here imported as `exp`) can only have properties; the top-level module object is *never* callable. The most common solution here is to define a `default` export for a callable/constructable object; some module loader shims will automatically detect this situation and replace the top-level object with the `default` export.

Library file layout

The layout of your declaration files should mirror the layout of the library.

A library can consist of multiple modules, such as

```
myLib
+---- index.js
+---- foo.js
+---- bar
    +---- index.js
    +---- baz.js
```

These could be imported as

```
var a = require("myLib");
var b = require("myLib/foo");
var c = require("myLib/bar");
var d = require("myLib/bar/baz");
```

Your declaration files should thus be

```
@types/myLib
+---- index.d.ts
+---- foo.d.ts
+---- bar
    +---- index.d.ts
    +---- baz.d.ts

// Type definitions for [~THE LIBRARY NAME~] [~OPTIONAL VERSION NUMBER~]
// Project: [~THE PROJECT NAME~]
// Definitions by: [~YOUR NAME~] <[~A URL FOR YOU~]>

/*~ This template shows how to write a global plugin. */

/*~ Write a declaration for the original type and add new members.
 *~ For example, this adds a 'toBinaryString' method with overloads to
 *~ the built-in number type.
 */
interface Number {
    toBinaryString(opts?: MyLibrary.BinaryFormatOptions): string;
```

```
toBinaryString(  
    callback: MyLibrary.BinaryFormatCallback,  
    opts?: MyLibrary.BinaryFormatOptions  
)  
: string;  
  
/*~ If you need to declare several types, place them inside a namespace  
 *~ to avoid adding too many things to the global namespace.  
 */  
declare namespace MyLibrary {  
    type BinaryFormatCallback = (n: number) => string;  
    interface BinaryFormatOptions {  
        prefix?: string;  
        padding: number;  
    }  
}
```

[Go to TOC](#)

Global Libraries

A *global* library is one that can be accessed from the global scope (i.e. without using any form of `import`). Many libraries simply expose one or more global variables for use. For example, if you were using `jQuery`, the `$` variable can be used by simply referring to it:

```
$(() => {
  console.log("hello!");
});
```

You'll usually see guidance in the documentation of a global library of how to use the library in an HTML script tag:

```
<script src="http://a.great.cdn.for/someLib.js"></script>
```

Today, most popular globally-accessible libraries are actually written as UMD libraries (see below). UMD library documentation is hard to distinguish from global library documentation. Before writing a global declaration file, make sure the library isn't actually UMD.

Identifying a Global Library from Code

Global library code is usually extremely simple. A global "Hello, world" library might look like this:

```
function createGreeting(s) {
  return "Hello, " + s;
}
```

or like this:

```
window.createGreeting = function (s) {
  return "Hello, " + s;
};
```

When looking at the code of a global library, you'll usually see:

- Top-level `var` statements or `function` declarations
- One or more assignments to `window.someName`
- Assumptions that DOM primitives like `document` or `window` exist

You *won't* see:

- Checks for, or usage of, module loaders like `require` or `define`
- CommonJS/Node.js-style imports of the form `var fs = require("fs");`
- Calls to `define(...)`
- Documentation describing how to `require` or import the library

Examples of Global Libraries

Because it's usually easy to turn a global library into a UMD library, very few popular libraries are still written in the global style. However, libraries that are small and require the DOM (or have *no* dependencies) may still be global.

Global Library Template

You can see an example DTS below:

```
// Type definitions for [~THE LIBRARY NAME~] [~OPTIONAL VERSION NUMBER~]
// Project: [~THE PROJECT NAME~]
// Definitions by: [~YOUR NAME~] <[~A URL FOR YOU~]>

/*~ If this library is callable (e.g. can be invoked as myLib(3)),
 *~ include those call signatures here.
 *~ Otherwise, delete this section.
 */
declare function myLib(a: string): string;
declare function myLib(a: number): number;

/*~ If you want the name of this library to be a valid type name,
 *~ you can do so here.
 */
/*~ For example, this allows us to write 'var x: myLib';
 *~ Be sure this actually makes sense! If it doesn't, just
 *~ delete this declaration and add types inside the namespace below.
 */
interface myLib {
  name: string;
  length: number;
  extras?: string[];
}

/*~ If your library has properties exposed on a global variable,
 *~ place them here.
 *~ You should also place types (interfaces and type alias) here.
 */
declare namespace myLib {
  //~ We can write 'myLib.timeout = 50;'
  let timeout: number;

  //~ We can access 'myLib.version', but not change it
  const version: string;

  //~ There's some class we can create via 'let c = new myLib.Cat(42)'
  //~ Or reference e.g. 'function f(c: myLib.Cat) { ... }'
  class Cat {
    constructor(n: number);

    //~ We can read 'c.age' from a 'Cat' instance
    readonly age: number;

    //~ We can invoke 'c.purr()' from a 'Cat' instance
    purr(): void;
  }

  //~ We can declare a variable as
}
```

```
//~ 'var s: myLib.CatSettings = { weight: 5, name: "Maru" };'  
interface CatSettings {  
    weight: number;  
    name: string;  
    tailLength?: number;  
}  
  
//~ We can write 'const v: myLib.VetID = 42;'  
//~ or 'const v: myLib.VetID = "bob";'  
type VetID = string | number;  
  
//~ We can invoke 'myLib.checkCat(c)' or 'myLib.checkCat(c, v);'  
function checkCat(c: Cat, s?: VetID);  
}
```

[Go to TOC](#)

For example, when you want to work with JavaScript code which looks like:

```
const Greeter = require("super-greeter");
const greeter = new Greeter();
greeter.greet();
```

To handle both importing via UMD and modules:

```
// Type definitions for [~THE LIBRARY NAME~] [~OPTIONAL VERSION NUMBER~]
// Project: [~THE PROJECT NAME~]
// Definitions by: [~YOUR NAME~] <[~A URL FOR YOU~]>

/*~ This is the module template file for class modules.
 *~ You should rename it to index.d.ts and place it in a folder with the same name
as the module.
 *~ For example, if you were writing a file for "super-greeter", this
 *~ file should be 'super-greeter/index.d.ts'
 */

// Note that ES6 modules cannot directly export class objects.
// This file should be imported using the CommonJS-style:
// import x = require('[~THE MODULE~]');
//
// Alternatively, if --allowSyntheticDefaultImports or
// --esModuleInterop is turned on, this file can also be
// imported as a default import:
// import x from '[~THE MODULE~]';
//
// Refer to the TypeScript documentation at
// https://www.typescriptlang.org/docs/handbook/modules.html#export--and-import--
require
// to understand common workarounds for this limitation of ES6 modules.

/*~ If this module is a UMD module that exposes a global variable 'myClassLib'
when
 *~ loaded outside a module loader environment, declare that global here.
 *~ Otherwise, delete this declaration.
 */
export as namespace "super-greeter";

/*~ This declaration specifies that the class constructor function
 *~ is the exported object from the file
 */
export = Greeter;

/*~ Write your module's methods and properties in this class */
declare class Greeter {
    constructor(customGreeting?: string);

    greet: void;

    myMethod(opts: MyClass.MyClassMethodOptions): number;
}

/*~ If you want to expose types from your module as well, you can
 *~ place them in this block.
 *~
 *~ Note that if you decide to include this namespace, the module can be
 *~ incorrectly imported as a namespace object, unless
```

```
*~ --esModuleInterop is turned on:  
*~   import * as x from '[~THE MODULE~]'; // WRONG! DO NOT DO THIS!  
*/  
declare namespace MyClass {  
  export interface MyClassMethodOptions {  
    width?: number;  
    height?: number;  
  }  
}
```

[Go to TOC](#)

For example, when you want to work with JavaScript code which looks like:

```
import greeter from "super-greeter";
greeter(2);
greeter("Hello world");
```

To handle both importing via UMD and modules:

```
// Type definitions for [~THE LIBRARY NAME~] [~OPTIONAL VERSION NUMBER~]
// Project: [~THE PROJECT NAME~]
// Definitions by: [~YOUR NAME~] <[~A URL FOR YOU~]>

/*~ This is the module template file for function modules.
 *~ You should rename it to index.d.ts and place it in a folder with the same name
as the module.
 *~ For example, if you were writing a file for "super-greeter", this
 *~ file should be 'super-greeter/index.d.ts'
 */

// Note that ES6 modules cannot directly export class objects.
// This file should be imported using the CommonJS-style:
// import x = require('[~THE MODULE~]');
//
// Alternatively, if --allowSyntheticDefaultImports or
// --esModuleInterop is turned on, this file can also be
// imported as a default import:
// import x from '[~THE MODULE~]';
//
// Refer to the TypeScript documentation at
// https://www.typescriptlang.org/docs/handbook/modules.html#export--and-import--
require
// to understand common workarounds for this limitation of ES6 modules.

/*~ If this module is a UMD module that exposes a global variable 'myFuncLib' when
 *~ loaded outside a module loader environment, declare that global here.
 *~ Otherwise, delete this declaration.
 */
export as namespace myFuncLib;

/*~ This declaration specifies that the function
 *~ is the exported object from the file
 */
export = Greeter;

/*~ This example shows how to have multiple overloads for your function */
declare function Greeter(name: string): Greeter.NamedReturnType;
declare function Greeter(length: number): Greeter.LengthReturnType;

/*~ If you want to expose types from your module as well, you can
 *~ place them in this block. Often you will want to describe the
 *~ shape of the return type of the function; that type should
 *~ be declared in here, as this example shows.
 *
 *~ Note that if you decide to include this namespace, the module can be
 *~ incorrectly imported as a namespace object, unless
 *~ --esModuleInterop is turned on:
 *~ import * as x from '[~THE MODULE~]'; // WRONG! DO NOT DO THIS!
 */
declare namespace Greeter {
```

```
export interface LengthReturnType {
  width: number;
  height: number;
}
export interface NamedReturnType {
  firstName: string;
  lastName: string;
}

/*~ If the module also has properties, declare them here. For example,
 *~ this declaration says that this code is legal:
 *~ import f = require('super-greeter');
 *~ console.log(f.defaultName);
 */
export const defaultName: string;
export let defaultLength: number;
```

[Go to TOC](#)

For example, when you want to work with JavaScript code which extends another library.

```
import { greeter } from "super-greeter";

// Normal Greeter API
greeter(2);
greeter("Hello world");

// Now we extend the object with a new function at runtime
import "hyper-super-greeter";
greeter.hyperGreet();
```

The definition for "super-greeter":

```
/*~ This example shows how to have multiple overloads for your function */
export interface GreeterFunction {
  (name: string): void
  (time: number): void
}

/*~ This example shows how to export a function specified by an interface */
export const greeter: GreeterFunction;
```

We can extend the existing module like the following:

```
// Type definitions for [~THE LIBRARY NAME~] [~OPTIONAL VERSION NUMBER~]
// Project: [~THE PROJECT NAME~]
// Definitions by: [~YOUR NAME~] <[~A URL FOR YOU~]>

/*~ This is the module plugin template file. You should rename it to index.d.ts
 *~ and place it in a folder with the same name as the module.
 *~ For example, if you were writing a file for "super-greeter", this
 *~ file should be 'super-greeter/index.d.ts'
 */

/*~ On this line, import the module which this module adds to */
import { greeter } from "super-greeter";

/*~ Here, declare the same module as the one you imported above
 *~ then we expand the existing declaration of the greeter function
 */
export module "super-greeter" {
  export interface GreeterFunction {
    /** Greets even better! */
    hyperGreet(): void;
  }
}
```

This uses [declaration merging](#)

The Impact of ES6 on Module Plugins

Some plugins add or modify top-level exports on existing modules. While this is legal in CommonJS and other loaders, ES6 modules are considered immutable and this pattern will not be possible. Because TypeScript is loader-agnostic, there is no compile-time enforcement of this policy, but developers intending to transition to an ES6 module loader should be aware of this.

[Go to TOC](#)

Comparing JavaScript to an example DTS

Common CommonJS Patterns

A module using CommonJS patterns uses `module.exports` to describe the exported values. For example, here is a module which exports a function and a numerical constant:

```
const maxInterval = 12;
function getArrayLength(arr) {
  return arr.length;
}

module.exports = {
  getArrayLength,
  maxInterval,
};
```

This can be described by the following `.d.ts`:

```
export function getArrayLength(arr: any[]): number;
export const maxInterval: 12;
```

The TypeScript playground can show you the `.d.ts` equivalent for JavaScript code. You can [try it yourself here](#).

The `.d.ts` syntax intentionally looks like [ES Modules](#) syntax. ES Modules was ratified by TC39 in 2019, while it has been available via transpilers for a long time, however if you have a JavaScript codebase using ES Modules:

```
export function getArrayLength(arr) {
  return arr.length;
}
```

This would have the following `.d.ts` equivalent:

```
export function getArrayLength(arr: any[]): number;
```

Default Exports

In CommonJS you can export any value as the default export, for example here is a regular expression module:

```
module.exports = /hello( world)?/;
```

Which can be described by the following `.d.ts`:

```
declare const helloWorld: RegExp;
export default helloWorld;
```

Or a number:

```
module.exports = 3.142;

declare const pi: number;
export default pi;
```

One style of exporting in CommonJS is to export a function. Because a function is also an object, then extra fields can be added and are included in the export.

```
function getArrayLength(arr) {
  return arr.length;
}
getArrayLength.maxInterval = 12;
module.exports = getArrayLength;
```

Which can be described with:

```
export default function getArrayLength(arr: any[]): number;
export const maxInterval: 12;
```

Note that using `export default` in your .d.ts files requires `esModuleInterop: true` to work. If you can't have `esModuleInterop: true` in your project, such as when you're submitting a PR to Definitely Typed, you'll have to use the `export=` syntax instead. This older syntax is harder to use but works everywhere. Here's how the above example would have to be written using `export=`:

```
declare function getArrayLength(arr: any[]): number;
declare namespace getArrayLength {
  declare const maxInterval: 12;
}

export = getArrayLength;
```

See [Module: Functions](#) for details of how that works, and the [Modules reference](#) page.

Handling Many Consuming Import

There are many ways to import a module in modern consuming code:

```
const fastify = require("fastify");
const { fastify } = require("fastify");
import fastify = require("fastify");
import * as Fastify from "fastify";
import { fastify, FastifyInstance } from "fastify";
import fastify from "fastify";
import fastify, { FastifyInstance } from "fastify";
```

Covering all of these cases requires the JavaScript code to actually support all of these patterns. To support many of these patterns, a CommonJS module would need to look something like:

```

class FastifyInstance {}

function fastify() {
  return new FastifyInstance();
}

fastify.FastifyInstance = FastifyInstance;

// Allows for { fastify }
fastify.fastify = fastify;
// Allows for strict ES Module support
fastify.default = fastify;
// Sets the default export
module.exports = fastify;

```

Types in Modules

You may want to provide a type for JavaScript code which does not exist

```

function getArrayMetadata(arr) {
  return {
    length: getArrayLength(arr),
    firstObject: arr[0],
  };
}

module.exports = {
  getArrayMetadata,
};

```

This can be described with:

```

export type ArrayMetadata = {
  length: number;
  firstObject: any | undefined;
};

export function getArrayMetadata(arr: any[]): ArrayMetadata;

```

This example is a good case for [using generics](#) to provide richer type information:

```

export type ArrayMetadata<ArrType> = {
  length: number;
  firstObject: ArrType | undefined;
};

export function getArrayMetadata<ArrType>(
  arr: ArrType[]
): ArrayMetadata<ArrType>;

```

Now the type of the array propagates into the `ArrayMetadata` type.

The types which are exported can then be re-used by consumers of the modules using either `import` or `import type` in TypeScript code or [JSDoc imports](#).

Namespaces in Module Code

Trying to describe the runtime relationship of JavaScript code can be tricky. When the ES Module-like syntax doesn't provide enough tools to describe the exports then you can use `namespaces`.

For example, you may have complex enough types to describe that you choose to namespace them inside your `.d.ts`:

```
// This represents the JavaScript class which would be available at runtime
export class API {
    constructor(baseURL: string);
    getInfo(opts: API.InfoRequest): API.InfoResponse;
}

// This namespace is merged with the API class and allows for consumers, and this
// file
// to have types which are nested away in their own sections.
declare namespace API {
    export interface InfoRequest {
        id: string;
    }

    export interface InfoResponse {
        width: number;
        height: number;
    }
}
```

To understand how namespaces work in `.d.ts` files read the [.d.ts deep dive](#).

Optional Global Usage

You can use `export as namespace` to declare that your module will be available in the global scope in UMD contexts:

```
export as namespace moduleName;
```

Reference Example

To give you an idea of how all these pieces can come together, here is a reference `.d.ts` to start with when making a new module

```
// Type definitions for [~THE LIBRARY NAME~] [~OPTIONAL VERSION NUMBER~]
// Project: [~THE PROJECT NAME~]
// Definitions by: [~YOUR NAME~] <[~A URL FOR YOU~]>

/*~ This is the module template file. You should rename it to index.d.ts
 *~ and place it in a folder with the same name as the module.
 *~ For example, if you were writing a file for "super-greeter", this
 *~ file should be 'super-greeter/index.d.ts'
 */

/*~ If this module is a UMD module that exposes a global variable 'myLib' when
 *~ loaded outside a module loader environment, declare that global here.
 *~ Otherwise, delete this declaration.
 */
```

```

export as namespace myLib;

/*~ If this module exports functions, declare them like so.
 */
export function myFunction(a: string): string;
export function myOtherFunction(a: number): number;

/*~ You can declare types that are available via importing the module */
export interface SomeType {
  name: string;
  length: number;
  extras?: string[];
}

/*~ You can declare properties of the module using const, let, or var */
export const myField: number;

```

Library file layout

The layout of your declaration files should mirror the layout of the library.

A library can consist of multiple modules, such as

```

myLib
 +--- index.js
 +--- foo.js
 +--- bar
   +--- index.js
   +--- baz.js

```

These could be imported as

```

var a = require("myLib");
var b = require("myLib/foo");
var c = require("myLib/bar");
var d = require("myLib/bar/baz");

```

Your declaration files should thus be

```

@types/myLib
 +--- index.d.ts
 +--- foo.d.ts
 +--- bar
   +--- index.d.ts
   +--- baz.d.ts

```

Testing your types

If you are planning on submitting these changes to DefinitelyTyped for everyone to also use, then we recommend you:

1. Create a new folder in `node_modules/@types/[libname]`
2. Create an `index.d.ts` in that folder, and copy the example in
3. See where your usage of the module breaks, and start to fill out the `index.d.ts`

4. When you're happy, clone [DefinitelyTyped/DefinitelyTyped](#) and follow the instructions in the README.

Otherwise

1. Create a new file in the root of your source tree: `[libname].d.ts`
2. Add `declare module "[libname]" { }`
3. Add the template inside the braces of the declare module, and see where your usage breaks

[Go to TOC](#)

TypeScript began its life as an attempt to bring traditional object-oriented types to JavaScript so that the programmers at Microsoft could bring traditional object-oriented programs to the web. As it has developed, TypeScript's type system has evolved to model code written by native JavaScripters. The resulting system is powerful, interesting and messy.

This introduction is designed for working Haskell or ML programmers who want to learn TypeScript. It describes how the type system of TypeScript differs from Haskell's type system. It also describes unique features of TypeScript's type system that arise from its modelling of JavaScript code.

This introduction does not cover object-oriented programming. In practice, object-oriented programs in TypeScript are similar to those in other popular languages with OO features.

Prerequisites

In this introduction, I assume you know the following:

- How to program in JavaScript, the good parts.
- Type syntax of a C-descended language.

If you need to learn the good parts of JavaScript, read [JavaScript: The Good Parts](#). You may be able to skip the book if you know how to write programs in a call-by-value lexically scoped language with lots of mutability and not much else. [R⁴S Scheme](#) is a good example.

[The C++ Programming Language](#) is a good place to learn about C-style type syntax. Unlike C++, TypeScript uses postfix types, like so: `x: string` instead of `string x`.

Concepts not in Haskell

Built-in types

JavaScript defines 8 built-in types:

Type	Explanation
<code>Number</code>	a double-precision IEEE 754 floating point.
<code>String</code>	an immutable UTF-16 string.
<code>BigInt</code>	integers in the arbitrary precision format.
<code>Boolean</code>	<code>true</code> and <code>false</code> .
<code>Symbol</code>	a unique value usually used as a key.
<code>Null</code>	equivalent to the unit type.
<code>Undefined</code>	also equivalent to the unit type.
<code>Object</code>	similar to records.

[See the MDN page for more detail.](#)

TypeScript has corresponding primitive types for the built-in types:

- `number`
- `string`
- `bignum`
- `boolean`
- `symbol`
- `null`
- `undefined`
- `object`

Other important TypeScript types

Type	Explanation
<code>unknown</code>	the top type.
<code>never</code>	the bottom type.
<code>object literal</code>	eg <code>{ property: Type }</code>
<code>void</code>	a subtype of <code>undefined</code> intended for use as a return type.
<code>T[]</code>	mutable arrays, also written <code>Array<T></code>
<code>[T, T]</code>	tuples, which are fixed-length but mutable
<code>(t: T) => U</code>	functions

Notes:

1. Function syntax includes parameter names. This is pretty hard to get used to!

```
let fst: (a: any, b: any) => any = (a, b) => a;
// or more precisely:
let fst: <T, U>(a: T, b: U) => T = (a, b) => a;
```

2. Object literal type syntax closely mirrors object literal value syntax:

```
let o: { n: number; xs: object[] } = { n: 1, xs: [] };
```

3. `[T, T]` is a subtype of `T[]`. This is different than Haskell, where tuples are not related to lists.

Boxed types

JavaScript has boxed equivalents of primitive types that contain the methods that programmers associate with those types. TypeScript reflects this with, for example, the difference between the primitive type `number` and the boxed type `Number`. The boxed types are rarely needed, since their methods return primitives.

```
(1).toExponential();
// equivalent to
Number.prototype.toExponential.call(1);
```

Note that calling a method on a numeric literal requires it to be in parentheses to aid the parser.

Gradual typing

TypeScript uses the type `any` whenever it can't tell what the type of an expression should be. Compared to `Dynamic`, calling `any` a type is an overstatement. It just turns off the type checker wherever it appears. For example, you can push any value into an `any[]` without marking the value in any way:

```
// with "noImplicitAny": false in tsconfig.json, anys: any[]
const anys = [];
anys.push(1);
anys.push("oh no");
anys.push({ anything: "goes" });
```

And you can use an expression of type `any` anywhere:

```
any.map(anys[1]); // oh no, "oh no" is not a function
```

`any` is contagious, too — if you initialize a variable with an expression of type `any`, the variable has type `any` too.

```
let sepsis = anys[0] + anys[1]; // this could mean anything
```

To get an error when TypeScript produces an `any`, use `"noImplicitAny": true`, or `"strict": true` in `tsconfig.json`.

Structural typing

Structural typing is a familiar concept to most functional programmers, although Haskell and most MLs are not structurally typed. Its basic form is pretty simple:

```
// @strict: false
let o = { x: "hi", extra: 1 }; // ok
let o2: { x: string } = o; // ok
```

Here, the object literal `{ x: "hi", extra: 1 }` has a matching literal type `{ x: string, extra: number }`. That type is assignable to `{ x: string }` since it has all the required properties and those properties have assignable types. The extra property doesn't prevent assignment, it just makes it a subtype of `{ x: string }`.

Named types just give a name to a type; for assignability purposes there's no difference between the type alias `One` and the interface type `Two` below. They both have a property `p: string`. (Type aliases behave differently from interfaces with respect to recursive definitions and type parameters, however.)

```
// @errors: 2322
type One = { p: string };
interface Two {
  p: string;
}
class Three {
  p = "Hello";
}

let x: One = { p: "hi" };
let two: Two = x;
two = new Three();
```

Unions

In TypeScript, union types are untagged. In other words, they are not discriminated unions like `data` in Haskell. However, you can often discriminate types in a union using built-in tags or other properties.

```
function start(
  arg: string | string[] | () => string | { s: string }
): string {
  // this is super common in JavaScript
  if (typeof arg === "string") {
    return commonCase(arg);
  } else if (Array.isArray(arg)) {
    return arg.map(commonCase).join(",");
  } else if (typeof arg === "function") {
    return commonCase(arg());
  } else {
    return commonCase(arg.s);
  }

  function commonCase(s: string): string {
    // finally, just convert a string to another string
    return s;
  }
}
```

`string`, `Array` and `Function` have built-in type predicates, conveniently leaving the object type for the `else` branch. It is possible, however, to generate unions that are difficult to differentiate at runtime. For new code, it's best to build only discriminated unions.

The following types have built-in predicates:

Type	Predicate
<code>string</code>	<code>typeof s === "string"</code>
<code>number</code>	<code>typeof n === "number"</code>
<code>bigint</code>	<code>typeof m === "bigint"</code>
<code>boolean</code>	<code>typeof b === "boolean"</code>
<code>symbol</code>	<code>typeof g === "symbol"</code>
<code>undefined</code>	<code>typeof undefined === "undefined"</code>

Type	Predicate
function	<code>typeof f === "function"</code>
array	<code>Array.isArray(a)</code>
object	<code>typeof o === "object"</code>

Note that functions and arrays are objects at runtime, but have their own predicates.

Intersections

In addition to unions, TypeScript also has intersections:

```
type Combined = { a: number } & { b: string };
type Conflicting = { a: number } & { a: string };
```

`Combined` has two properties, `a` and `b`, just as if they had been written as one object literal type. Intersection and union are recursive in case of conflicts, so `Conflicting.a: number & string`.

Unit types

Unit types are subtypes of primitive types that contain exactly one primitive value. For example, the string `"foo"` has the type `"foo"`. Since JavaScript has no built-in enums, it is common to use a set of well-known strings instead. Unions of string literal types allow TypeScript to type this pattern:

```
declare function pad(s: string, n: number, direction: "left" | "right"): string;
pad("hi", 10, "left");
```

When needed, the compiler *widens* — converts to a supertype — the unit type to the primitive type, such as `"foo"` to `string`. This happens when using mutability, which can hamper some uses of mutable variables:

```
// @errors: 2345
declare function pad(s: string, n: number, direction: "left" | "right"): string;
// ---cut---
let s = "right";
pad("hi", 10, s); // error: 'string' is not assignable to '"left" | "right"'
```

Here's how the error happens:

- `"right": "right"`
- `s: string` because `"right"` widens to `string` on assignment to a mutable variable.
- `string` is not assignable to `"left" | "right"`

You can work around this with a type annotation for `s`, but that in turn prevents assignments to `s` of variables that are not of type `"left" | "right"`.

```
declare function pad(s: string, n: number, direction: "left" | "right"): string;
// ---cut---
let s: "left" | "right" = "right";
pad("hi", 10, s);
```

Concepts similar to Haskell

Contextual typing

TypeScript has some obvious places where it can infer types, like variable declarations:

```
let s = "I'm a string!";
```

But it also infers types in a few other places that you may not expect if you've worked with other C-syntax languages:

```
declare function map<T, U>(f: (t: T) => U, ts: T[]): U[];  
let sns = map((n) => n.toString(), [1, 2, 3]);
```

Here, `n: number` in this example also, despite the fact that `T` and `U` have not been inferred before the call. In fact, after `[1,2,3]` has been used to infer `T=number`, the return type of `n => n.toString()` is used to infer `U=string`, causing `sns` to have the type `string[]`.

Note that inference will work in any order, but intellisense will only work left-to-right, so TypeScript prefers to declare `map` with the array first:

```
declare function map<T, U>(ts: T[], f: (t: T) => U): U[];
```

Contextual typing also works recursively through object literals, and on unit types that would otherwise be inferred as `string` or `number`. And it can infer return types from context:

```
declare function run<T>(thunk: (t: T) => void): T;  
let i: { inference: string } = run((o) => {  
  o.inference = "INSERT STATE HERE";  
});
```

The type of `o` is determined to be `{ inference: string }` because

1. Declaration initializers are contextually typed by the declaration's type: `{ inference: string }`.
2. The return type of a call uses the contextual type for inferences, so the compiler infers that `T={ inference: string }`.
3. Arrow functions use the contextual type to type their parameters, so the compiler gives `o: { inference: string }`.

And it does so while you are typing, so that after typing `o.`, you get completions for the property `inference`, along with any other properties you'd have in a real program. Altogether, this feature can make TypeScript's inference look a bit like a unifying type inference engine, but it is not.

Type aliases

Type aliases are mere aliases, just like `type` in Haskell. The compiler will attempt to use the alias name wherever it was used in the source code, but does not always succeed.

```
type Size = [number, number];
let x: Size = [101.1, 999.9];
```

The closest equivalent to `newtype` is a *tagged intersection*:

```
type FString = string & { __compileTimeOnly: any };
```

An `FString` is just like a normal string, except that the compiler thinks it has a property named `__compileTimeOnly` that doesn't actually exist. This means that `FString` can still be assigned to `string`, but not the other way round.

Discriminated Unions

The closest equivalent to `data` is a union of types with discriminant properties, normally called discriminated unions in TypeScript:

```
type Shape =
| { kind: "circle"; radius: number }
| { kind: "square"; x: number }
| { kind: "triangle"; x: number; y: number };
```

Unlike Haskell, the tag, or discriminant, is just a property in each object type. Each variant has an identical property with a different unit type. This is still a normal union type; the leading `|` is an optional part of the union type syntax. You can discriminate the members of the union using normal JavaScript code:

```
type Shape =
| { kind: "circle"; radius: number }
| { kind: "square"; x: number }
| { kind: "triangle"; x: number; y: number };

function area(s: Shape) {
  if (s.kind === "circle") {
    return Math.PI * s.radius * s.radius;
  } else if (s.kind === "square") {
    return s.x * s.x;
  } else {
    return (s.x * s.y) / 2;
  }
}
```

Note that the return type of `area` is inferred to be `number` because TypeScript knows the function is total. If some variant is not covered, the return type of `area` will be `number | undefined` instead.

Also, unlike Haskell, common properties show up in any union, so you can usefully discriminate multiple members of the union:

```
type Shape =
| { kind: "circle"; radius: number }
| { kind: "square"; x: number }
| { kind: "triangle"; x: number; y: number };
// ---cut---
function height(s: Shape) {
  if (s.kind === "circle") {
    return 2 * s.radius;
  }
}
```

```

    } else {
      // s.kind: "square" | "triangle"
      return s.x;
    }
}

```

Type Parameters

Like most C-descended languages, TypeScript requires declaration of type parameters:

```

function liftArray<T>(t: T): Array<T> {
  return [t];
}

```

There is no case requirement, but type parameters are conventionally single uppercase letters. Type parameters can also be constrained to a type, which behaves a bit like type class constraints:

```

function firstish<T extends { length: number }>(t1: T, t2: T): T {
  return t1.length > t2.length ? t1 : t2;
}

```

TypeScript can usually infer type arguments from a call based on the type of the arguments, so type arguments are usually not needed.

Because TypeScript is structural, it doesn't need type parameters as much as nominal systems. Specifically, they are not needed to make a function polymorphic. Type parameters should only be used to *propagate* type information, such as constraining parameters to be the same type:

```

function length<T extends ArrayLike<unknown>>(t: T): number {}
function length(t: ArrayLike<unknown>): number {}

```

In the first `length`, `T` is not necessary; notice that it's only referenced once, so it's not being used to constrain the type of the return value or other parameters.

Higher-kinded types

TypeScript does not have higher kinded types, so the following is not legal:

```

function length<T extends ArrayLike<unknown>, U>(m: T<U>) {}

```

Point-free programming

Point-free programming — heavy use of currying and function composition — is possible in JavaScript, but can be verbose. In TypeScript, type inference often fails for point-free programs, so you'll end up specifying type parameters instead of value parameters. The result is so verbose that it's usually better to avoid point-free programming.

Module system

JavaScript's modern module syntax is a bit like Haskell's, except that any file with `import` or `export` is implicitly a module:

```
import { value, Type } from "npm-package";
import { other, Types } from "./local-package";
import * as prefix from "../lib/third-package";
```

You can also import commonjs modules — modules written using node.js' module system:

```
import f = require("single-function-package");
```

You can export with an export list:

```
export { f };
function f() {
    return g();
}
function g() {} // g is not exported
```

Or by marking each export individually:

```
export function f() { return g() }
function g() {}
```

The latter style is more common but both are allowed, even in the same file.

readonly and const

In JavaScript, mutability is the default, although it allows variable declarations with `const` to declare that the reference is immutable. The referent is still mutable:

```
const a = [1, 2, 3];
a.push(102); // OK:
a[0] = 101; // D:
```

TypeScript additionally has a `readonly` modifier for properties.

```
interface Rx {
    readonly x: number;
}
let rx: Rx = { x: 1 };
rx.x = 12; // error
```

It also ships with a mapped type `Readonly<T>` that makes all properties `readonly`:

```
interface X {
    x: number;
}
let rx: Readonly<X> = { x: 1 };
rx.x = 12; // error
```

And it has a specific `ReadonlyArray<T>` type that removes side-affecting methods and prevents writing to indices of the array, as well as special syntax for this type:

```
let a: ReadonlyArray<number> = [1, 2, 3];
let b: readonly number[] = [1, 2, 3];
a.push(102); // error
b[0] = 101; // error
```

You can also use a `const`-assertion, which operates on arrays and object literals:

```
let a = [1, 2, 3] as const;
a.push(102); // error
a[0] = 101; // error
```

However, none of these options are the default, so they are not consistently used in TypeScript code.

Next Steps

This doc is a high level overview of the syntax and types you would use in everyday code. From here you should:

- Read the full Handbook [from start to finish](#) (30m)
- Explore the [Playground examples](#)

[Go to TOC](#)

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 in creating a variable and assigning it to a particular value, TypeScript will use the value as its type.

```
let helloWorld = "Hello World";
// ^?
```

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 design patterns make it difficult for types to be inferred automatically (for example, patterns that use dynamic programming). To cover these cases, TypeScript supports an extension of the JavaScript language, which offers places for you to tell TypeScript what the types should be.

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

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

You can explicitly describe this object's shape using an `interface` declaration:

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

You can then declare that a JavaScript object conforms to the shape of your new `interface` by using syntax like `: TypeName` after a variable declaration:

```
interface User {
  name: string;
  id: number;
}
// ---cut---
const user: User = {
  name: "Hayes",
  id: 0,
};
```

If you provide an object that doesn't match the interface you have provided, TypeScript will warn you:

```
// @errors: 2322
interface User {
  name: string;
  id: number;
}

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

Since JavaScript supports classes and object-oriented programming, so does TypeScript. You can use an interface declaration 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);
```

You can use interfaces to annotate parameters and return values to functions:

```
// @noErrors
interface User {
  name: string;
  id: number;
}
// ---cut---
function getAdminUser(): User {
  //...
}
```

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

There is already a small set of primitive types available in JavaScript: `boolean`, `bignumber`, `null`, `number`, `string`, `symbol`, and `undefined`, which you can use in an interface. TypeScript extends this list with a few more, such as `any` (allow anything), `unknown` (ensure someone using this type declares what the type is), `never` (it's not possible that this type could happen), and `void` (a function which returns `undefined` or has no return value).

You'll see that there are two syntaxes for building types: [Interfaces and Types](#). You should prefer `interface`. Use `type` when you need specific features.

Composing Types

With TypeScript, you can create complex types by combining simple ones. There are two popular ways to do so: with unions, and with generics.

Unions

With a union, you can declare that a type could be one of many types. For example, you can describe a `boolean` type as being either `true` or `false`:

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

Note: If you hover over `MyBool` above, you'll see that it is classed as `boolean`. That's a property of the Structural Type System. More on this below.

A popular use-case for union types is to describe the set of `string` or `number` literals that a value is allowed to be:

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

Unions provide a way to handle different types too. For example, you may have a function that takes an `array` or a `string`:

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

To learn the type of a variable, use `typeof`:

Type	Predicate
<code>string</code>	<code>typeof s === "string"</code>
<code>number</code>	<code>typeof n === "number"</code>
<code>boolean</code>	<code>typeof b === "boolean"</code>

Type	Predicate
undefined	<code>typeof undefined === "undefined"</code>
function	<code>typeof f === "function"</code>
array	<code>Array.isArray(a)</code>

For example, you can make a function return different values depending on whether it is passed a string or an array:

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

Generics

Generics provide variables to types. A common example is an array. An array without generics could contain anything. An array with generics can describe the values that the array contains.

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

You can declare your own types that use generics:

```
// @errors: 2345
interface Backpack<Type> {
  add: (obj: Type) => void;
  get: () => Type;
}

// This line is a shortcut to tell TypeScript there is a
// constant called `backpack`, and to not worry about where it came from.
declare const backpack: Backpack<string>;

// object is a string, because we declared it above as the variable part of
Backpack.
const object = backpack.get();

// Since the backpack variable is a string, you can't pass a number to the add
function.
backpack.add(23);
```

Structural Type System

One of TypeScript's core principles is that type checking focuses on the *shape* that values have. This is sometimes called "duck typing" or "structural typing".

In a structural type system, if two objects have the same shape, they are considered to be of the same type.

```

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

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

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

```

The `point` variable is never declared to be a `Point` type. However, TypeScript compares the shape of `point` to the shape of `Point` in the type-check. They have the same shape, so the code passes.

The shape-matching only requires a subset of the object's fields to match.

```

// @errors: 2345
interface Point {
  x: number;
  y: number;
}

function logPoint(p: Point) {
  console.log(`${p.x}, ${p.y}`);
}
// ---cut---
const point3 = { x: 12, y: 26, z: 89 };
logPoint(point3); // logs "12, 26"

const rect = { x: 33, y: 3, width: 30, height: 80 };
logPoint(rect); // logs "33, 3"

const color = { hex: "#187ABF" };
logPoint(color);

```

There is no difference between how classes and objects conform to shapes:

```

// @errors: 2345
interface Point {
  x: number;
  y: number;
}

function logPoint(p: Point) {
  console.log(`${p.x}, ${p.y}`);
}
// ---cut---
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); // logs "13, 56"
```

If the object or class has all the required properties, TypeScript will say they match, regardless of the implementation details.

Next Steps

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

- Read the full Handbook [from start to finish](#)
- Explore the [Playground examples](#)

[Go to TOC](#)

TypeScript is a popular choice for programmers accustomed to other languages with static typing, such as C# and Java.

TypeScript's type system offers many of the same benefits, such as better code completion, earlier detection of errors, and clearer communication between parts of your program. While TypeScript provides many familiar features for these developers, it's worth stepping back to see how JavaScript (and therefore TypeScript) differ from traditional OOP languages. Understanding these differences will help you write better JavaScript code, and avoid common pitfalls that programmers who go straight from C#/Java to TypeScript may fall in to.

Co-learning JavaScript

If you're familiar with JavaScript already but are primarily a Java or C# programmer, this introductory page can help explain some of the common misconceptions and pitfalls you might be susceptible to. Some of the ways that TypeScript models types are quite different from Java or C#, and it's important to keep these in mind when learning TypeScript.

If you're a Java or C# programmer that is new to JavaScript in general, we recommend learning a little bit of JavaScript *without* types first to understand JavaScript's runtime behaviors. Because TypeScript doesn't change how your code *runs*, you'll still have to learn how JavaScript works in order to write code that actually does something!

It's important to remember that TypeScript uses the same *runtime* as JavaScript, so any resources about how to accomplish specific runtime behavior (converting a string to a number, displaying an alert, writing a file to disk, etc.) will always apply equally well to TypeScript programs. Don't limit yourself to TypeScript-specific resources!

Rethinking the Class

C# and Java are what we might call *mandatory OOP* languages. In these languages, the *class* is the basic unit of code organization, and also the basic container of all data *and* behavior at runtime. Forcing all functionality and data to be held in classes can be a good domain model for some problems, but not every domain *needs* to be represented this way.

Free Functions and Data

In JavaScript, functions can live anywhere, and data can be passed around freely without being inside a pre-defined `class` or `struct`. This flexibility is extremely powerful. "Free" functions (those not associated with a class) working over data without an implied OOP hierarchy tends to be the preferred model for writing programs in JavaScript.

Static Classes

Additionally, certain constructs from C# and Java such as singletons and static classes are unnecessary in TypeScript.

OOP in TypeScript

That said, you can still use classes if you like! Some problems are well-suited to being solved by a traditional OOP hierarchy, and TypeScript's support for JavaScript classes will make these models even more powerful. TypeScript supports many common patterns such as implementing interfaces, inheritance, and static methods.

We'll cover classes later in this guide.

Rethinking Types

TypeScript's understanding of a *type* is actually quite different from C# or Java's. Let's explore some differences.

Nominal Reified Type Systems

In C# or Java, any given value or object has one exact type - either `null`, a primitive, or a known class type. We can call methods like `value.GetType()` or `value.getClass()` to query the exact type at runtime. The definition of this type will reside in a class somewhere with some name, and we can't use two classes with similar shapes in lieu of each other unless there's an explicit inheritance relationship or commonly-implemented interface.

These aspects describe a *reified, nominal* type system. The types we wrote in the code are present at runtime, and the types are related via their declarations, not their structures.

Types as Sets

In C# or Java, it's meaningful to think of a one-to-one correspondence between runtime types and their compile-time declarations.

In TypeScript, it's better to think of a type as a *set of values* that share something in common. Because types are just sets, a particular value can belong to *many* sets at the same time.

Once you start thinking of types as sets, certain operations become very natural. For example, in C#, it's awkward to pass around a value that is *either* a `string` or `int`, because there isn't a single type that represents this sort of value.

In TypeScript, this becomes very natural once you realize that every type is just a set. How do you describe a value that either belongs in the `string` set or the `number` set? It simply belongs to the *union* of those sets: `string | number`.

TypeScript provides a number of mechanisms to work with types in a set-theoretic way, and you'll find them more intuitive if you think of types as sets.

Erased Structural Types

In TypeScript, objects are *not* of a single exact type. For example, if we construct an object that satisfies an interface, we can use that object where that interface is expected even though there was no declarative relationship between the two.

```
interface Pointlike {
  x: number;
  y: number;
}
interface Named {
  name: string;
}

function logPoint(point: Pointlike) {
  console.log("x = " + point.x + ", y = " + point.y);
}

function logName(x: Named) {
  console.log("Hello, " + x.name);
}

const obj = {
  x: 0,
  y: 0,
  name: "Origin",
};

logPoint(obj);
logName(obj);
```

TypeScript's type system is *structural*, not nominal: We can use `obj` as a `Pointlike` because it has `x` and `y` properties that are both numbers. The relationships between types are determined by the properties they contain, not whether they were declared with some particular relationship.

TypeScript's type system is also *not refined*: There's nothing at runtime that will tell us that `obj` is `Pointlike`. In fact, the `Pointlike` type is not present *in any form* at runtime.

Going back to the idea of *types as sets*, we can think of `obj` as being a member of both the `Pointlike` set of values and the `Named` set of values.

Consequences of Structural Typing

OOP programmers are often surprised by two particular aspects of structural typing.

Empty Types

The first is that the *empty type* seems to defy expectation:

```
class Empty {}

function fn(arg: Empty) {
  // do something?
}
```

```
// No error, but this isn't an 'Empty' ?
fn({ k: 10 });
```

TypeScript determines if the call to `fn` here is valid by seeing if the provided argument is a valid `Empty`. It does so by examining the *structure* of `{ k: 10 }` and `class Empty { }`. We can see that `{ k: 10 }` has *all* of the properties that `Empty` does, because `Empty` has no properties. Therefore, this is a valid call!

This may seem surprising, but it's ultimately a very similar relationship to one enforced in nominal OOP languages. A subclass cannot *remove* a property of its base class, because doing so would destroy the natural subtype relationship between the derived class and its base. Structural type systems simply identify this relationship implicitly by describing subtypes in terms of having properties of compatible types.

Identical Types

Another frequent source of surprise comes with identical types:

```
class Car {
  drive() {
    // hit the gas
  }
}

class Golfer {
  drive() {
    // hit the ball far
  }
}

// No error?
let w: Car = new Golfer();
```

Again, this isn't an error because the *structures* of these classes are the same. While this may seem like a potential source of confusion, in practice, identical classes that shouldn't be related are not common.

We'll learn more about how classes relate to each other in the [Classes chapter](#).

Reflection

OOP programmers are accustomed to being able to query the type of any value, even a generic one:

```
// C#
static void LogType<T>() {
  Console.WriteLine(typeof(T).Name);
}
```

Because TypeScript's type system is fully erased, information about e.g. the instantiation of a generic type parameter is not available at runtime.

JavaScript does have some limited primitives like `typeof` and `instanceof`, but remember that these operators are still working on the values as they exist in the type-erased output code. For example, `typeof (new Car())` will be `"object"`, not `Car` or `"Car"`.

Next Steps

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

- Read the full Handbook [from start to finish](#) (30m)
- Explore the [Playground examples](#)

[Go to TOC](#)

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 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 arguments, 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.height;
```

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*, it's 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:

```
// @errors: 2551
const obj = { width: 10, height: 15 };
const area = obj.width * obj.height;
```

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 `:`:

```
// @errors: 1005
let a = (4
```

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.height` 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:

```
// @errors: 2363
console.log(4 / []);
```

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](#) or
 - [JavaScript guide at the Mozilla Web Docs](#)
- Continue to [TypeScript for JavaScript Programmers](#)
- Read the full Handbook [from start to finish](#) (30m)
- Explore the [Playground examples](#)

[Go to TOC](#)

For programs to be useful, we need to be able to work with some of the simplest units of data: numbers, strings, structures, boolean values, and the like. In TypeScript, we support the same types as you would expect in JavaScript, with an extra enumeration type thrown in to help things along.

Boolean

The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a `boolean` value.

```
let isDone: boolean = false;
```

Number

As in JavaScript, all numbers in TypeScript are either floating point values or BigIntegers. These floating point numbers get the type `number`, while BigIntegers get the type `bigint`. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals introduced in ECMAScript 2015.

```
// @target: ES2020
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
let big: bigint = 100n;
```

String

Another fundamental part of creating programs in JavaScript for webpages and servers alike is working with textual data. As in other languages, we use the type `string` to refer to these textual datatypes. Just like JavaScript, TypeScript also uses double quotes (`"`) or single quotes (`'`) to surround string data.

```
let color: string = "blue";
// prettier-ignore
color = 'red';
```

You can also use *template strings*, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (```) character, and embedded expressions are of the form `${ expr }` .

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${fullName}.

I'll be ${age + 1} years old next month.`;
```

This is equivalent to declaring `sentence` like so:

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;
// ---cut---
let sentence: string =
```

```
"Hello, my name is " +
fullName +
".\n\n" +
"I'll be " +
(age + 1) +
" years old next month.";
```

Array

TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by `[]` to denote an array of that element type:

```
let list: number[] = [1, 2, 3];
```

The second way uses a generic array type, `Array<elementType>`:

```
let list: Array<number> = [1, 2, 3];
```

Tuple

Tuple types allow you to express an array with a fixed number of elements whose types are known, but need not be the same. For example, you may want to represent a value as a pair of a `string` and a `number`:

```
// @errors: 2322
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

When accessing an element with a known index, the correct type is retrieved:

```
// @errors: 2339
let x: [string, number];
x = ["hello", 10]; // OK
/// ---cut---
// OK
console.log(x[0].substring(1));
console.log(x[1].substring(1));
```

Accessing an element outside the set of known indices fails with an error:

```
// @errors: 2493 2532 2322
let x: [string, number];
x = ["hello", 10]; // OK
/// ---cut---
x[3] = "world";
console.log(x[5].toString());
```

Enum

A helpful addition to the standard set of datatypes from JavaScript is the `enum`. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

```
enum Color {
  Red,
  Green,
  Blue,
}
let c: Color = Color.Green;
```

By default, enums begin numbering their members starting at `0`. You can change this by manually setting the value of one of its members. For example, we can start the previous example at `1` instead of `0`:

```
enum Color {
  Red = 1,
  Green,
  Blue,
}
let c: Color = Color.Green;
```

Or, even manually set all the values in the enum:

```
enum Color {
  Red = 1,
  Green = 2,
  Blue = 4,
}
let c: Color = Color.Green;
```

A handy feature of enums is that you can also go from a numeric value to the name of that value in the enum. For example, if we had the value `2` but weren't sure what that mapped to in the `Color` enum above, we could look up the corresponding name:

```
enum Color {
  Red = 1,
  Green,
  Blue,
}
let colorName: string = Color[2];

// Displays 'Green'
console.log(colorName);
```

Unknown

We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content – e.g. from the user – or we may want to intentionally accept all values in our API. In these cases, we want to provide a type that tells the compiler and future readers that this variable could be anything, so we give it the `unknown` type.

```
let notSure: unknown = 4;
notSure = "maybe a string instead";

// OK, definitely a boolean
notSure = false;
```

If you have a variable with an unknown type, you can narrow it to something more specific by doing `typeof` checks, comparison checks, or more advanced type guards that will be discussed in a later chapter:

```
// @errors: 2322 2322 2322
declare const maybe: unknown;
// 'maybe' could be a string, object, boolean, undefined, or other types
const aNumber: number = maybe;

if (maybe === true) {
    // TypeScript knows that maybe is a boolean now
    const aBoolean: boolean = maybe;
    // So, it cannot be a string
    const aString: string = maybe;
}

if (typeof maybe === "string") {
    // TypeScript knows that maybe is a string
    const aString: string = maybe;
    // So, it cannot be a boolean
    const aBoolean: boolean = maybe;
}
```

Any

In some situations, not all type information is available or its declaration would take an inappropriate amount of effort. These may occur for values from code that has been written without TypeScript or a 3rd party library. In these cases, we might want to opt-out of type checking. To do so, we label these values with the `any` type:

```
declare function getValue(key: string): any;
// OK, return value of 'getValue' is not checked
const str: string = getValue("myString");
```

The `any` type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type checking during compilation.

Unlike `unknown`, variables of type `any` allow you to access arbitrary properties, even ones that don't exist. These properties include functions and TypeScript will not check their existence or type:

```
// @errors: 2571
let looselyTyped: any = 4;
// OK, ifItExists might exist at runtime
looselyTyped.ifItExists();
// OK,toFixed exists (but the compiler doesn't check)
looselyTyped.toFixed();

let strictlyTyped: unknown = 4;
strictlyTyped.toFixed();
```

The `any` will continue to propagate through your objects:

```
let looselyTyped: any = {};
let d = looselyTyped.a.b.c.d;
// ^?
```

After all, remember that all the convenience of `any` comes at the cost of losing type safety. Type safety is one of the main motivations for using TypeScript and you should try to avoid using `any` when not necessary.

Void

`void` is a little like the opposite of `any`: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

```
function warnUser(): void {
  console.log("This is my warning message");
}
```

Declaring variables of type `void` is not useful because you can only assign `null` (only if `strictNullChecks` is not specified, see next section) or `undefined` to them:

```
// @strict: false
let unusable: void = undefined;
// OK if `--strictNullChecks` is not given
unusable = null;
```

Null and Undefined

In TypeScript, both `undefined` and `null` actually have their types named `undefined` and `null` respectively. Much like `void`, they're not extremely useful on their own:

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

By default `null` and `undefined` are subtypes of all other types. That means you can assign `null` and `undefined` to something like `number`.

However, when using the `strictNullChecks` flag, `null` and `undefined` are only assignable to `unknown`, `any` and their respective types (the one exception being that `undefined` is also assignable to `void`). This helps avoid many common errors. In cases where you want to pass in either a `string` or `null` or `undefined`, you can use the union type `string | null | undefined`.

Union types are an advanced topic that we'll cover in a later chapter.

As a note: we encourage the use of `strictNullChecks` when possible, but for the purposes of this handbook, we will assume it is turned off.

Never

The `never` type represents the type of values that never occur. For instance, `never` is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns. Variables also acquire the type `never` when narrowed by any type guards that can never be true.

The `never` type is a subtype of, and assignable to, every type; however, `no` type is a subtype of, or assignable to, `never` (except `never` itself). Even `any` isn't assignable to `never`.

Some examples of functions returning `never`:

```
// Function returning never must not have a reachable end point
function error(message: string): never {
    throw new Error(message);
}

// Inferred return type is never
function fail() {
    return error("Something failed");
}

// Function returning never must not have a reachable end point
function infiniteLoop(): never {
    while (true) {}
}
```

Object

`object` is a type that represents the non-primitive type, i.e. anything that is not `number`, `string`, `boolean`, `bigint`, `symbol`, `null`, or `undefined`.

With `object` type, APIs like `Object.create` can be better represented. For example:

```
// @errors: 2345
declare function create(o: object | null): void;

// OK
create({ prop: 0 });
create(null);
create(undefined); // with `--strictNullChecks` flag enabled, undefined is not a
                   // subtype of null

create(42);
create("string");
create(false);
```

Generally, you won't need to use this.

Type assertions

Sometimes you'll end up in a situation where you'll know more about a value than TypeScript does. Usually, this will happen when you know the type of some entity could be more specific than its current type.

Type *assertions* are a way to tell the compiler "trust me, I know what I'm doing." A type assertion is like a type cast in other languages, but it performs no special checking or restructuring of data. It has no runtime impact and is used purely by the compiler. TypeScript assumes that you, the programmer, have performed any special checks that you need.

Type assertions have two forms.

One is the `as`-syntax:

```
let someValue: unknown = "this is a string";
let strLength: number = (someValue as string).length;
```

The other version is the "angle-bracket" syntax:

```
let someValue: unknown = "this is a string";
let strLength: number = (<string>someValue).length;
```

The two samples are equivalent. Using one over the other is mostly a choice of preference; however, when using TypeScript with JSX, only `as`-style assertions are allowed.

A note about `let`

You may have noticed that so far, we've been using the `let` keyword instead of JavaScript's `var` keyword which you might be more familiar with. The `let` keyword is actually a newer JavaScript construct that TypeScript makes available. You can read in the Handbook Reference on [Variable Declarations](#) more about how `let` and `const` fix a lot of the problems with `var`.

About `Number`, `String`, `Boolean`, `Symbol` and `Object`

It can be tempting to think that the types `Number`, `String`, `Boolean`, `Symbol`, or `Object` are the same as the lowercase versions recommended above. These types do not refer to the language primitives however, and almost never should be used as a type.

```
// @errors: 2339
function reverse(s: String): String {
  return s.split("").reverse().join("");
}

reverse("hello world");
```

Instead, use the types `number`, `string`, `boolean`, `object` and `symbol`.

```
function reverse(s: string): string {
    return s.split("").reverse().join("");
}

reverse("hello world");
```

[Go to TOC](#)

Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components, but this may feel a bit awkward to programmers more comfortable with an object-oriented approach, where classes inherit functionality and objects are built from these classes. Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers can build their applications using this object-oriented class-based approach. In TypeScript, we allow developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

Classes

Let's take a look at a simple class-based example:

```
class Greeter {
  greeting: string;

  constructor(message: string) {
    this.greeting = message;
  }

  greet() {
    return "Hello, " + this.greeting;
  }
}

let greeter = new Greeter("world");
```

The syntax should look familiar if you've used C# or Java before. We declare a new class `Greeter`. This class has three members: a property called `greeting`, a constructor, and a method `greet`.

You'll notice that in the class when we refer to one of the members of the class we prepend `this..`. This denotes that it's a member access.

In the last line we construct an instance of the `Greeter` class using `new`. This calls into the constructor we defined earlier, creating a new object with the `Greeter` shape, and running the constructor to initialize it.

Inheritance

In TypeScript, we can use common object-oriented patterns. One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance.

Let's take a look at an example:

```
class Animal {
  move(distanceInMeters: number = 0) {
    console.log(`Animal moved ${distanceInMeters}m.`);
  }
}

class Dog extends Animal {
  bark() {
    console.log("Woof! Woof!");
  }
}
```

```

    }
const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();

```

This example shows the most basic inheritance feature: classes inherit properties and methods from base classes. Here, `Dog` is a *derived* class that derives from the `Animal` *base* class using the `extends` keyword. Derived classes are often called *subclasses*, and base classes are often called *superclasses*.

Because `Dog` extends the functionality from `Animal`, we were able to create an instance of `Dog` that could both `bark()` and `move()`.

Let's now look at a more complex example.

```

class Animal {
  name: string;
  constructor(theName: string) {
    this.name = theName;
  }
  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Snake extends Animal {
  constructor(name: string) {
    super(name);
  }
  move(distanceInMeters = 5) {
    console.log("Slithering...");
    super.move(distanceInMeters);
  }
}

class Horse extends Animal {
  constructor(name: string) {
    super(name);
  }
  move(distanceInMeters = 45) {
    console.log("Galloping...");
    super.move(distanceInMeters);
  }
}

let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the Palomino");

sam.move();
tom.move(34);

```

This example covers a few other features we didn't previously mention. Again, we see the `extends` keywords used to create two new subclasses of `Animal`: `Horse` and `Snake`.

One difference from the prior example is that each derived class that contains a constructor function *must* call `super()` which will execute the constructor of the base class. What's more, before we ever access a property on `this` in a constructor body, we *have* to call `super()`. This is an important rule that TypeScript will enforce.

The example also shows how to override methods in the base class with methods that are specialized for the subclass. Here both `Snake` and `Horse` create a `move` method that overrides the `move` from `Animal`, giving it functionality specific to each class. Note that even though `tom` is declared as an `Animal`, since its value is a `Horse`, calling `tom.move(34)` will call the overriding method in `Horse`:

```
Slithering...
Sammy the Python moved 5m.
Galloping...
Tommy the Palomino moved 34m.
```

Public, private, and protected modifiers

Public by default

In our examples, we've been able to freely access the members that we declared throughout our programs. If you're familiar with classes in other languages, you may have noticed in the above examples we haven't had to use the word `public` to accomplish this; for instance, C# requires that each member be explicitly labeled `public` to be visible. In TypeScript, each member is `public` by default.

You may still mark a member `public` explicitly. We could have written the `Animal` class from the previous section in the following way:

```
class Animal {
  public name: string;

  public constructor(theName: string) {
    this.name = theName;
  }

  public move(distanceInMeters: number) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}
```

ECMAScript Private Fields

With TypeScript 3.8, TypeScript supports the new JavaScript syntax for private fields:

```
// @errors: 18013
class Animal {
  #name: string;
  constructor(theName: string) {
    this.#name = theName;
  }
}

new Animal("Cat").#name;
```

This syntax is built into the JavaScript runtime and can have better guarantees about the isolation of each private field. Right now, the best documentation for these private fields is in the TypeScript 3.8 [release notes](#).

Understanding TypeScript's `private`

TypeScript also has its own way to declare a member as being marked `private`, it cannot be accessed from outside of its containing class. For example:

```
// @errors: 2341
class Animal {
  private name: string;

  constructor(theName: string) {
    this.name = theName;
  }

  new Animal("Cat").name;
```

TypeScript is a structural type system. When we compare two different types, regardless of where they came from, if the types of all members are compatible, then we say the types themselves are compatible.

However, when comparing types that have `private` and `protected` members, we treat these types differently. For two types to be considered compatible, if one of them has a `private` member, then the other must have a `private` member that originated in the same declaration. The same applies to `protected` members.

Let's look at an example to better see how this plays out in practice:

```
// @errors: 2322
class Animal {
  private name: string;
  constructor(theName: string) {
    this.name = theName;
  }
}

class Rhino extends Animal {
  constructor() {
    super("Rhino");
  }
}

class Employee {
  private name: string;
  constructor(theName: string) {
    this.name = theName;
  }
}

let animal = new Animal("Goat");
let rhino = new Rhino();
let employee = new Employee("Bob");
```

```
animal = rhino;
animal = employee;
```

In this example, we have an `Animal` and a `Rhino`, with `Rhino` being a subclass of `Animal`. We also have a new class `Employee` that looks identical to `Animal` in terms of shape. We create some instances of these classes and then try to assign them to each other to see what will happen. Because `Animal` and `Rhino` share the `private` side of their shape from the same declaration of `private name: string` in `Animal`, they are compatible. However, this is not the case for `Employee`. When we try to assign from an `Employee` to `Animal` we get an error that these types are not compatible. Even though `Employee` also has a `private` member called `name`, it's not the one we declared in `Animal`.

Understanding `protected`

The `protected` modifier acts much like the `private` modifier with the exception that members declared `protected` can also be accessed within deriving classes. For example,

```
// @errors: 2445
class Person {
  protected name: string;
  constructor(name: string) {
    this.name = name;
  }
}

class Employee extends Person {
  private department: string;

  constructor(name: string, department: string) {
    super(name);
    this.department = department;
  }

  public getElevatorPitch() {
    return `Hello, my name is ${this.name} and I work in ${this.department}.`;
  }
}

let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name);
```

Notice that while we can't use `name` from outside of `Person`, we can still use it from within an instance method of `Employee` because `Employee` derives from `Person`.

A constructor may also be marked `protected`. This means that the class cannot be instantiated outside of its containing class, but can be extended. For example,

```
// @errors: 2674
class Person {
  protected name: string;
  protected constructor(theName: string) {
    this.name = theName;
  }
}
```

```
// Employee can extend Person
class Employee extends Person {
  private department: string;

  constructor(name: string, department: string) {
    super(name);
    this.department = department;
  }

  public getElevatorPitch() {
    return `Hello, my name is ${this.name} and I work in ${this.department}.`;
  }
}

let howard = new Employee("Howard", "Sales");
let john = new Person("John");
```

Readonly modifier

You can make properties readonly by using the `readonly` keyword. Readonly properties must be initialized at their declaration or in the constructor.

```
// @errors: 2540
class Octopus {
  readonly name: string;
  readonly numberOfLegs: number = 8;

  constructor(theName: string) {
    this.name = theName;
  }
}

let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit";
```

Parameter properties

In our last example, we had to declare a readonly member `name` and a constructor parameter `theName` in the `Octopus` class. This is needed in order to have the value of `theName` accessible after the `Octopus` constructor is executed. *Parameter properties* let you create and initialize a member in one place. Here's a further revision of the previous `Octopus` class using a parameter property:

```
class Octopus {
  readonly numberOfLegs: number = 8;
  constructor(readonly name: string) {}

  let dad = new Octopus("Man with the 8 strong legs");
  dad.name;
```

Notice how we dropped `theName` altogether and just use the shortened `readonly name: string` parameter on the constructor to create and initialize the `name` member. We've consolidated the declarations and assignment into one location.

Parameter properties are declared by prefixing a constructor parameter with an accessibility modifier or `readonly`, or both. Using `private` for a parameter property declares and initializes a private member; likewise, the same is done for `public`, `protected`, and `readonly`.

Accessors

TypeScript supports getters/setters as a way of intercepting accesses to a member of an object. This gives you a way of having finer-grained control over how a member is accessed on each object.

Let's convert a simple class to use `get` and `set`. First, let's start with an example without getters and setters.

```
// @strict: false
class Employee {
  fullName: string;
}

let employee = new Employee();
employee.fullName = "Bob Smith";

if (employee.fullName) {
  console.log(employee.fullName);
}
```

While allowing people to randomly set `fullName` directly is pretty handy, we may also want enforce some constraints when `fullName` is set.

In this version, we add a setter that checks the length of the `newName` to make sure it's compatible with the max-length of our backing database field. If it isn't we throw an error notifying client code that something went wrong.

To preserve existing functionality, we also add a simple getter that retrieves `fullName` unmodified.

```
// @strict: false
const fullNameMaxLength = 10;

class Employee {
  private _fullName: string = "";

  get fullName(): string {
    return this._fullName;
  }

  set fullName(newName: string) {
    if (newName && newName.length > fullNameMaxLength) {
      throw new Error("fullName has a max length of " + fullNameMaxLength);
    }

    this._fullName = newName;
  }
}

let employee = new Employee();
employee.fullName = "Bob Smith";
```

```
if (employee.fullName) {
  console.log(employee.fullName);
}
```

To prove to ourselves that our accessor is now checking the length of values, we can attempt to assign a name longer than 10 characters and verify that we get an error.

A couple of things to note about accessors:

First, accessors require you to set the compiler to output ECMAScript 5 or higher. Downleveling to ECMAScript 3 is not supported. Second, accessors with a `get` and no `set` are automatically inferred to be `readonly`. This is helpful when generating a `.d.ts` file from your code, because users of your property can see that they can't change it.

Static Properties

Up to this point, we've only talked about the `instance` members of the class, those that show up on the object when it's instantiated. We can also create `static` members of a class, those that are visible on the class itself rather than on the instances. In this example, we use `static` on the origin, as it's a general value for all grids. Each instance accesses this value through prepending the name of the class. Similarly to prepending `this.` in front of instance accesses, here we prepend `Grid.` in front of static accesses.

```
class Grid {
  static origin = { x: 0, y: 0 };

  calculateDistanceFromOrigin(point: { x: number; y: number }) {
    let xDist = point.x - Grid.origin.x;
    let yDist = point.y - Grid.origin.y;
    return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
  }

  constructor(public scale: number) {}

  let grid1 = new Grid(1.0); // 1x scale
  let grid2 = new Grid(5.0); // 5x scale

  console.log(grid1.calculateDistanceFromOrigin({ x: 10, y: 10 }));
  console.log(grid2.calculateDistanceFromOrigin({ x: 10, y: 10 }));
}
```

Abstract Classes

Abstract classes are base classes from which other classes may be derived. They may not be instantiated directly. Unlike an interface, an abstract class may contain implementation details for its members. The `abstract` keyword is used to define abstract classes as well as abstract methods within an abstract class.

```
abstract class Animal {
  abstract makeSound(): void;

  move(): void {
    console.log("roaming the earth...");
  }
}
```

Methods within an abstract class that are marked as abstract do not contain an implementation and must be implemented in derived classes. Abstract methods share a similar syntax to interface methods. Both define the signature of a method without including a method body. However, abstract methods must include the `abstract` keyword and may optionally include access modifiers.

```
// @errors: 2511 2339
abstract class Department {
    constructor(public name: string) {}

    printName(): void {
        console.log("Department name: " + this.name);
    }

    abstract printMeeting(): void; // must be implemented in derived classes
}

class AccountingDepartment extends Department {
    constructor() {
        super("Accounting and Auditing"); // constructors in derived classes must call
        super();
    }

    printMeeting(): void {
        console.log("The Accounting Department meets each Monday at 10am.");
    }

    generateReports(): void {
        console.log("Generating accounting reports...");
    }
}

let department: Department; // ok to create a reference to an abstract type
department = new Department(); // error: cannot create an instance of an abstract
class
department = new AccountingDepartment(); // ok to create and assign a non-abstract
subclass
department.printName();
department.printMeeting();
department.generateReports(); // error: department is not of type
AccountingDepartment, cannot access generateReports
```

Advanced Techniques

Constructor functions

When you declare a class in TypeScript, you are actually creating multiple declarations at the same time. The first is the type of the *instance* of the class.

```
class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

    greet() {
        return "Hello, " + this.greeting;
    }
}
```

```

    }
}

let greeter: Greeter;
greeter = new Greeter("world");
console.log(greeter.greet()); // "Hello, world"

```

Here, when we say `let greeter: Greeter`, we're using `Greeter` as the type of instances of the class `Greeter`. This is almost second nature to programmers from other object-oriented languages.

We're also creating another value that we call the *constructor function*. This is the function that is called when we `new` up instances of the class. To see what this looks like in practice, let's take a look at the JavaScript created by the above example:

```

// @strict: false
let Greeter = (function () {
  function Greeter(message) {
    this.greeting = message;
  }

  Greeter.prototype.greet = function () {
    return "Hello, " + this.greeting;
  };

  return Greeter;
})();

let greeter;
greeter = new Greeter("world");
console.log(greeter.greet()); // "Hello, world"

```

Here, `let Greeter` is going to be assigned the constructor function. When we call `new` and run this function, we get an instance of the class. The constructor function also contains all of the static members of the class. Another way to think of each class is that there is an *instance* side and a *static* side.

Let's modify the example a bit to show this difference:

```

// @strict: false
class Greeter {
  static standardGreeting = "Hello, there";
  greeting: string;
  greet() {
    if (this.greeting) {
      return "Hello, " + this.greeting;
    } else {
      return Greeter.standardGreeting;
    }
  }
}

let greeter1: Greeter;
greeter1 = new Greeter();
console.log(greeter1.greet()); // "Hello, there"

let greeterMaker: typeof Greeter = Greeter;
greeterMaker.standardGreeting = "Hey there!";

let greeter2: Greeter = new greeterMaker();

```

```
console.log(greeter2.greet()); // "Hey there!"

let greeter3: Greeter;
greeter3 = new Greeter();
console.log(greeter3.greet()); // "Hey there!"
```

In this example, `greeter1` works similarly to before. We instantiate the `Greeter` class, and use this object. This we have seen before.

Next, we then use the class directly. Here we create a new variable called `greeterMaker`. This variable will hold the class itself, or said another way its constructor function. Here we use `typeof Greeter`, that is "give me the type of the `Greeter` class itself" rather than the instance type. Or, more precisely, "give me the type of the symbol called `Greeter`," which is the type of the constructor function. This type will contain all of the static members of `Greeter` along with the constructor that creates instances of the `Greeter` class. We show this by using `new` on `greeterMaker`, creating new instances of `Greeter` and invoking them as before. It is also good to mention that changing static property is frowned upon, here `greeter3` has "Hey there!" instead of "Hello, there" on `standardGreeting`.

Using a class as an interface

As we said in the previous section, a class declaration creates two things: a type representing instances of the class and a constructor function. Because classes create types, you can use them in the same places you would be able to use interfaces.

```
// @strict: false
class Point {
  x: number;
  y: number;
}

interface Point3d extends Point {
  z: number;
}

let point3d: Point3d = { x: 1, y: 2, z: 3 };
```

[Go to TOC](#)

Functions are the fundamental building block of any application in JavaScript. They're how you build up layers of abstraction, mimicking classes, information hiding, and modules. In TypeScript, while there are classes, namespaces, and modules, functions still play the key role in describing how to *do* things. TypeScript also adds some new capabilities to the standard JavaScript functions to make them easier to work with.

Functions

To begin, just as in JavaScript, TypeScript functions can be created both as a named function or as an anonymous function. This allows you to choose the most appropriate approach for your application, whether you're building a list of functions in an API or a one-off function to hand off to another function.

To quickly recap what these two approaches look like in JavaScript:

```
// @strict: false
// Named function
function add(x, y) {
  return x + y;
}

// Anonymous function
let myAdd = function (x, y) {
  return x + y;
};
```

Just as in JavaScript, functions can refer to variables outside of the function body. When they do so, they're said to *capture* these variables. While understanding how this works (and the trade-offs when using this technique) is outside of the scope of this article, having a firm understanding how this mechanic works is an important piece of working with JavaScript and TypeScript.

```
// @strict: false
let z = 100;

function addToZ(x, y) {
  return x + y + z;
}
```

Function Types

Typing the function

Let's add types to our simple examples from earlier:

```
function add(x: number, y: number): number {
  return x + y;
}

let myAdd = function (x: number, y: number): number {
  return x + y;
};
```

We can add types to each of the parameters and then to the function itself to add a return type. TypeScript can figure the return type out by looking at the return statements, so we can also optionally leave this off in many cases.

Writing the function type

Now that we've typed the function, let's write the full type of the function out by looking at each piece of the function type.

```
let myAdd: (x: number, y: number) => number = function (
  x: number,
  y: number
): number {
  return x + y;
};
```

A function's type has the same two parts: the type of the arguments and the return type. When writing out the whole function type, both parts are required. We write out the parameter types just like a parameter list, giving each parameter a name and a type. This name is just to help with readability. We could have instead written:

```
let myAdd: (baseValue: number, increment: number) => number = function (
  x: number,
  y: number
): number {
  return x + y;
};
```

As long as the parameter types line up, it's considered a valid type for the function, regardless of the names you give the parameters in the function type.

The second part is the return type. We make it clear which is the return type by using an arrow (`=>`) between the parameters and the return type. As mentioned before, this is a required part of the function type, so if the function doesn't return a value, you would use `void` instead of leaving it off.

Of note, only the parameters and the return type make up the function type. Captured variables are not reflected in the type. In effect, captured variables are part of the "hidden state" of any function and do not make up its API.

Inferring the types

In playing with the example, you may notice that the TypeScript compiler can figure out the type even if you only have types on one side of the equation:

```
// The parameters 'x' and 'y' have the type number
let myAdd = function (x: number, y: number): number {
  return x + y;
};

// myAdd has the full function type
```

```
// @errors: 2554
let myAdd2: (baseValue: number, increment: number) => number = function (x, y) {
    return x + y;
};
```

This is called "contextual typing", a form of type inference. This helps cut down on the amount of effort to keep your program typed.

Optional and Default Parameters

In TypeScript, every parameter is assumed to be required by the function. This doesn't mean that it can't be given `null` or `undefined`, but rather, when the function is called, the compiler will check that the user has provided a value for each parameter. The compiler also assumes that these parameters are the only parameters that will be passed to the function. In short, the number of arguments given to a function has to match the number of parameters the function expects.

```
// @errors: 2554
function buildName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob"); // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result3 = buildName("Bob", "Adams"); // ah, just right
```

In JavaScript, every parameter is optional, and users may leave them off as they see fit. When they do, their value is `undefined`. We can get this functionality in TypeScript by adding a `?` to the end of parameters we want to be optional. For example, let's say we want the last name parameter from above to be optional:

```
// @errors: 2554
function buildName(firstName: string, lastName?: string) {
    if (lastName) return firstName + " " + lastName;
    else return firstName;
}

let result1 = buildName("Bob"); // works correctly now
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result3 = buildName("Bob", "Adams"); // ah, just right
```

Any optional parameters must follow required parameters. Had we wanted to make the first name optional, rather than the last name, we would need to change the order of parameters in the function, putting the first name last in the list.

In TypeScript, we can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes `undefined` in its place. These are called default-initialized parameters. Let's take the previous example and default the last name to `"Smith"`.

```
// @errors: 2554
function buildName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob"); // works correctly now, returns "Bob Smith"
```

```
let result2 = buildName("Bob", undefined); // still works, also returns "Bob Smith"
let result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result4 = buildName("Bob", "Adams"); // ah, just right
```

Default-initialized parameters that come after all required parameters are treated as optional, and just like optional parameters, can be omitted when calling their respective function. This means optional parameters and trailing default parameters will share commonality in their types, so both

```
function buildName(firstName: string, lastName?: string) {
    // ...
}
```

and

```
function buildName(firstName: string, lastName = "Smith") {
    // ...
}
```

share the same type `(firstName: string, lastName?: string) => string`. The default value of `lastName` disappears in the type, only leaving behind the fact that the parameter is optional.

Unlike plain optional parameters, default-initialized parameters don't *need* to occur after required parameters. If a default-initialized parameter comes before a required parameter, users need to explicitly pass `undefined` to get the default initialized value. For example, we could write our last example with only a default initializer on `firstName`:

```
// @errors: 2554
function buildName(firstName = "Will", lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob"); // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result3 = buildName("Bob", "Adams"); // okay and returns "Bob Adams"
let result4 = buildName(undefined, "Adams"); // okay and returns "Will Adams"
```

Rest Parameters

Required, optional, and default parameters all have one thing in common: they talk about one parameter at a time. Sometimes, you want to work with multiple parameters as a group, or you may not know how many parameters a function will ultimately take. In JavaScript, you can work with the arguments directly using the `arguments` variable that is visible inside every function body.

In TypeScript, you can gather these arguments together into a variable:

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

// employeeName will be "Joseph Samuel Lucas MacKinzie"
let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

Rest parameters are treated as a boundless number of optional parameters. When passing arguments for a rest parameter, you can use as many as you want; you can even pass none. The compiler will build an array of the arguments passed in with the name given after the ellipsis (`...`), allowing you to use it in your function.

The ellipsis is also used in the type of the function with rest parameters:

```
function buildName(firstName: string, ...restOfName: string[]) {
  return firstName + " " + restOfName.join(" ");
}

let buildNameFun: (fname: string, ...rest: string[]) => string = buildName;
```

this

Learning how to use `this` in JavaScript is something of a rite of passage. Since TypeScript is a superset of JavaScript, TypeScript developers also need to learn how to use `this` and how to spot when it's not being used correctly. Fortunately, TypeScript lets you catch incorrect uses of `this` with a couple of techniques. If you need to learn how `this` works in JavaScript, though, first read Yehuda Katz's [Understanding JavaScript Function Invocation and "this"](#). Yehuda's article explains the inner workings of `this` very well, so we'll just cover the basics here.

this and arrow functions

In JavaScript, `this` is a variable that's set when a function is called. This makes it a very powerful and flexible feature, but it comes at the cost of always having to know about the context that a function is executing in. This is notoriously confusing, especially when returning a function or passing a function as an argument.

Let's look at an example:

```
// @strict: false
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function () {
    return function () {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);

      return { suit: this.suits[pickedSuit], card: pickedCard % 13 };
    };
  },
};

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Notice that `createCardPicker` is a function that itself returns a function. If we tried to run the example, we would get an error instead of the expected alert box. This is because the `this` being used in the function created by `createCardPicker` will be set to `window` instead of our `deck` object. That's because we call `cardPicker()` on its own. A top-level non-method syntax call like this will use `window` for `this`. (Note: under strict mode, `this` will be `undefined` rather than `window`).

We can fix this by making sure the function is bound to the correct `this` before we return the function to be used later. This way, regardless of how it's later used, it will still be able to see the original `deck` object. To do this, we change the function expression to use the ECMAScript 6 arrow syntax. Arrow functions capture the `this` where the function is created rather than where it is invoked:

```
// @strict: false
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function () {
    // NOTE: the line below is now an arrow function, allowing us to capture
    'this' right here
    return () => {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);

      return { suit: this.suits[pickedSuit], card: pickedCard % 13 };
    };
  },
};

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Even better, TypeScript will warn you when you make this mistake if you pass the `noImplicitThis` flag to the compiler. It will point out that `this` in `this.suits[pickedSuit]` is of type `any`.

this parameters

Unfortunately, the type of `this.suits[pickedSuit]` is still `any`. That's because `this` comes from the function expression inside the object literal. To fix this, you can provide an explicit `this` parameter. `this` parameters are fake parameters that come first in the parameter list of a function:

```
function f(this: void) {
  // make sure `this` is unusable in this standalone function
}
```

Let's add a couple of interfaces to our example above, `Card` and `Deck`, to make the types clearer and easier to reuse:

```
interface Card {
  suit: string;
  card: number;
}
```

```

interface Deck {
  suits: string[];
  cards: number[];
  createCardPicker(this: Deck): () => Card;
}

let deck: Deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  // NOTE: The function now explicitly specifies that its callee must be of type
  createCardPicker: function (this: Deck) {
    return () => {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);

      return { suit: this.suits[pickedSuit], card: pickedCard % 13 };
    };
  },
};

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);

```

Now TypeScript knows that `createCardPicker` expects to be called on a `Deck` object. That means that `this` is of type `Deck` now, not `any`, so `noImplicitThis` will not cause any errors.

`this` parameters in callbacks

You can also run into errors with `this` in callbacks, when you pass functions to a library that will later call them. Because the library that calls your callback will call it like a normal function, `this` will be `undefined`. With some work you can use `this` parameters to prevent errors with callbacks too. First, the library author needs to annotate the callback type with `this`:

```

interface UIElement {
  addClickListener(onclick: (this: void, e: Event) => void): void;
}

```

`this: void` means that `addClickListener` expects `onclick` to be a function that does not require a `this` type. Second, annotate your calling code with `this`:

```

// @strict: false
// @errors: 2345
interface UIElement {
  addClickListener(onclick: (this: void, e: Event) => void): void;
}
interface Event {
  message: string;
}
declare const uiElement: UIElement;
// ---cut---
class Handler {
  info: string;
  onClickBad(this: Handler, e: Event) {
    // oops, used `this` here. using this callback would crash at runtime
  }
}

```

```

    this.info = e.message;
}

let h = new Handler();
uiElement.addEventListener(h.onClickBad); // error!

```

With `this` annotated, you make it explicit that `onClickBad` must be called on an instance of `Handler`. Then TypeScript will detect that `addEventListener` requires a function that has `this: void`. To fix the error, change the type of `this`:

```

// @strict: false
interface UIElement {
  addEventListener(onclick: (this: void, e: Event) => void): void;
}
interface Event {
  message: string;
}
declare const uiElement: UIElement;
// ---cut---
class Handler {
  info: string;
  onClickGood(this: void, e: Event) {
    // can't use `this` here because it's of type void!
    console.log("clicked!");
  }
}

let h = new Handler();
uiElement.addEventListener(h.onClickGood);

```

Because `onClickGood` specifies its `this` type as `void`, it is legal to pass to `addEventListener`. Of course, this also means that it can't use `this.info`. If you want both then you'll have to use an arrow function:

```

// @strict: false
interface UIElement {
  addEventListener(onclick: (this: void, e: Event) => void): void;
}
interface Event {
  message: string;
}
declare const uiElement: UIElement;
// ---cut---
class Handler {
  info: string;
  onClickGood = (e: Event) => {
    this.info = e.message;
  };
}

```

This works because arrow functions use the outer `this`, so you can always pass them to something that expects `this: void`. The downside is that one arrow function is created per object of type `Handler`. Methods, on the other hand, are only created once and attached to `Handler`'s prototype. They are shared between all objects of type `Handler`.

Overloads

JavaScript is inherently a very dynamic language. It's not uncommon for a single JavaScript function to return different types of objects based on the shape of the arguments passed in.

```
// @strict: false
let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: any): any {
  // Check to see if we're working with an object/array
  // if so, they gave us the deck and we'll pick the card
  if (typeof x == "object") {
    let pickedCard = Math.floor(Math.random() * x.length);
    return pickedCard;
  }
  // Otherwise just let them pick the card
  else if (typeof x == "number") {
    let pickedSuit = Math.floor(x / 13);
    return { suit: suits[pickedSuit], card: x % 13 };
  }
}

let myDeck = [
  { suit: "diamonds", card: 2 },
  { suit: "spades", card: 10 },
  { suit: "hearts", card: 4 },
];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);
```

Here, the `pickCard` function will return two different things based on what the user has passed in. If the user passes in an object that represents the deck, the function will pick the card. If the user picks the card, we tell them which card they've picked. But how do we describe this to the type system?

The answer is to supply multiple function types for the same function as a list of overloads. This list is what the compiler will use to resolve function calls. Let's create a list of overloads that describe what our `pickCard` accepts and what it returns.

```
let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: { suit: string; card: number }[]): number;
function pickCard(x: number): { suit: string; card: number };
function pickCard(x: any): any {
  // Check to see if we're working with an object/array
  // if so, they gave us the deck and we'll pick the card
  if (typeof x == "object") {
    let pickedCard = Math.floor(Math.random() * x.length);
    return pickedCard;
  }
  // Otherwise just let them pick the card
  else if (typeof x == "number") {
    let pickedSuit = Math.floor(x / 13);
    return { suit: suits[pickedSuit], card: x % 13 };
  }
}
```

```

    }
}

let myDeck = [
  { suit: "diamonds", card: 2 },
  { suit: "spades", card: 10 },
  { suit: "hearts", card: 4 },
];

let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);

```

With this change, the overloads now give us type checked calls to the `pickCard` function.

In order for the compiler to pick the correct type check, it follows a similar process to the underlying JavaScript. It looks at the overload list and, proceeding with the first overload, attempts to call the function with the provided parameters. If it finds a match, it picks this overload as the correct overload. For this reason, it's customary to order overloads from most specific to least specific.

Note that the `function pickCard(x): any` piece is not part of the overload list, so it only has two overloads: one that takes an object and one that takes a number. Calling `pickCard` with any other parameter types would cause an error.

[Go to TOC](#)

A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable. Components that are capable of working on the data of today as well as the data of tomorrow will give you the most flexible capabilities for building up large software systems.

In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is *generics*, that is, being able to create a component that can work over a variety of types rather than a single one. This allows users to consume these components and use their own types.

Hello World of Generics

To start off, let's do the "hello world" of generics: the identity function. The identity function is a function that will return back whatever is passed in. You can think of this in a similar way to the `echo` command.

Without generics, we would either have to give the identity function a specific type:

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

Or, we could describe the identity function using the `any` type:

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

While using `any` is certainly generic in that it will cause the function to accept any and all types for the type of `arg`, we actually are losing the information about what that type was when the function returns. If we passed in a number, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned. Here, we will use a *type variable*, a special kind of variable that works on types rather than values.

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

We've now added a type variable `T` to the identity function. This `T` allows us to capture the type the user provides (e.g. `number`), so that we can use that information later. Here, we use `T` again as the return type. On inspection, we can now see the same type is used for the argument and the return type. This allows us to traffic that type information in one side of the function and out the other.

We say that this version of the `identity` function is *generic*, as it works over a range of types. Unlike using `any`, it's also just as precise (ie, it doesn't lose any information) as the first `identity` function that used numbers for the argument and return type.

Once we've written the generic identity function, we can call it in one of two ways. The first way is to pass all of the arguments, including the type argument, to the function:

```
function identity<T>(arg: T): T {
  return arg;
}
// ---cut---
let output = identity<string>("myString");
//      ^?
```

Here we explicitly set `T` to be `string` as one of the arguments to the function call, denoted using the `<>` around the arguments rather than `()`.

The second way is also perhaps the most common. Here we use *type argument inference* -- that is, we want the compiler to set the value of `T` for us automatically based on the type of the argument we pass in:

```
function identity<T>(arg: T): T {
  return arg;
}
// ---cut---
let output = identity("myString");
//      ^?
```

Notice that we didn't have to explicitly pass the type in the angle brackets (`<>`); the compiler just looked at the value `"myString"`, and set `T` to its type. While type argument inference can be a helpful tool to keep code shorter and more readable, you may need to explicitly pass in the type arguments as we did in the previous example when the compiler fails to infer the type, as may happen in more complex examples.

Working with Generic Type Variables

When you begin to use generics, you'll notice that when you create generic functions like `identity`, the compiler will enforce that you use any generically typed parameters in the body of the function correctly. That is, that you actually treat these parameters as if they could be any and all types.

Let's take our `identity` function from earlier:

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

What if we want to also log the length of the argument `arg` to the console with each call? We might be tempted to write this:

```
// @errors: 2339
function loggingIdentity<T>(arg: T): T {
  console.log(arg.length);
  return arg;
}
```

When we do, the compiler will give us an error that we're using the `.length` member of `arg`, but nowhere have we said that `arg` has this member. Remember, we said earlier that these type variables stand in for any and all types, so someone using this function could have passed in a `number` instead, which does not have a `.length` member.

Let's say that we've actually intended this function to work on arrays of `T` rather than `T` directly. Since we're working with arrays, the `.length` member should be available. We can describe this just like we would create arrays of other types:

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

You can read the type of `loggingIdentity` as "the generic function `loggingIdentity` takes a type parameter `T`, and an argument `arg` which is an array of `T`s, and returns an array of `T`s." If we passed in an array of numbers, we'd get an array of numbers back out, as `T` would bind to `number`. This allows us to use our generic type variable `T` as part of the types we're working with, rather than the whole type, giving us greater flexibility.

We can alternatively write the sample example this way:

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {
  console.log(arg.length); // Array has a .length, so no more error
  return arg;
}
```

You may already be familiar with this style of type from other languages. In the next section, we'll cover how you can create your own generic types like `Array<T>`.

Generic Types

In previous sections, we created generic identity functions that worked over a range of types. In this section, we'll explore the type of the functions themselves and how to create generic interfaces.

The type of generic functions is just like those of non-generic functions, with the type parameters listed first, similarly to function declarations:

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

let myIdentity: <T>(arg: T) => T = identity;
```

We could also have used a different name for the generic type parameter in the type, so long as the number of type variables and how the type variables are used line up.

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

let myIdentity: <U>(arg: U) => U = identity;
```

We can also write the generic type as a call signature of an object literal type:

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

let myIdentity: { <T>(arg: T): T } = identity;
```

Which leads us to writing our first generic interface. Let's take the object literal from the previous example and move it to an interface:

```
interface GenericIdentityFn {
  <T>(arg: T): T;
}

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

let myIdentity: GenericIdentityFn = identity;
```

In a similar example, we may want to move the generic parameter to be a parameter of the whole interface. This lets us see what type(s) we're generic over (e.g. `Dictionary<string>` rather than just `Dictionary`). This makes the type parameter visible to all the other members of the interface.

```
interface GenericIdentityFn<T> {
  (arg: T): T;
}

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

let myIdentity: GenericIdentityFn<number> = identity;
```

Notice that our example has changed to be something slightly different. Instead of describing a generic function, we now have a non-generic function signature that is a part of a generic type. When we use `GenericIdentityFn`, we now will also need to specify the corresponding type argument (here: `number`), effectively locking in what the underlying call signature will use. Understanding when to put the type parameter directly on the call signature and when to put it on the interface itself will be helpful in describing what aspects of a type are generic.

In addition to generic interfaces, we can also create generic classes. Note that it is not possible to create generic enums and namespaces.

Generic Classes

A generic class has a similar shape to a generic interface. Generic classes have a generic type parameter list in angle brackets (`<>`) following the name of the class.

```
// @strict: false
class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}
```

```
let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function (x, y) {
    return x + y;
};
```

This is a pretty literal use of the `GenericNumber` class, but you may have noticed that nothing is restricting it to only use the `number` type. We could have instead used `string` or even more complex objects.

```
// @strict: false
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}
// ---cut---
let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function (x, y) {
    return x + y;
};
console.log(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

Just as with interface, putting the type parameter on the class itself lets us make sure all of the properties of the class are working with the same type.

As we covered in [our section on classes](#), a class has two sides to its type: the static side and the instance side. Generic classes are only generic over their instance side rather than their static side, so when working with classes, static members can not use the class's type parameter.

Generic Constraints

If you remember from an earlier example, you may sometimes want to write a generic function that works on a set of types where you have some knowledge about what capabilities that set of types will have. In our `loggingIdentity` example, we wanted to be able to access the `.length` property of `arg`, but the compiler could not prove that every type had a `.length` property, so it warns us that we can't make this assumption.

```
// @errors: 2339
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);
    return arg;
}
```

Instead of working with any and all types, we'd like to constrain this function to work with any and all types that also have the `.length` property. As long as the type has this member, we'll allow it, but it's required to have at least this member. To do so, we must list our requirement as a constraint on what `T` can be.

To do so, we'll create an interface that describes our constraint. Here, we'll create an interface that has a single `.length` property and then we'll use this interface and the `extends` keyword to denote our constraint:

```
// @errors: 2345
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length); // Now we know it has a .length property, so no more
  error
  return arg;
}
```

Because the generic function is now constrained, it will no longer work over any and all types:

```
// @errors: 2345
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}
// ---cut---
loggingIdentity(3);
```

Instead, we need to pass in values whose type has all the required properties:

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}
// ---cut---
loggingIdentity({ length: 10, value: 3 });
```

Using Type Parameters in Generic Constraints

You can declare a type parameter that is constrained by another type parameter. For example, here we'd like to get a property from an object given its name. We'd like to ensure that we're not accidentally grabbing a property that does not exist on the `obj`, so we'll place a constraint between the two types:

```
// @errors: 2345
function getProperty<T, K extends keyof T>(obj: T, key: K) {
  return obj[key];
}

let x = { a: 1, b: 2, c: 3, d: 4 };

getProperty(x, "a");
getProperty(x, "m");
```

Using Class Types in Generics

When creating factories in TypeScript using generics, it is necessary to refer to class types by their constructor functions. For example,

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

A more advanced example uses the prototype property to infer and constrain relationships between the constructor function and the instance side of class types.

```
// @strict: false
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;
```

[Go to TOC](#)

One of TypeScript's core principles is that type checking focuses on the *shape* that values have. This is sometimes called "duck typing" or "structural subtyping". In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.

Our First Interface

The easiest way to see how interfaces work is to start with a simple example:

```
function printLabel(labeledObj: { label: string }) {
  console.log(labeledObj.label);
}

let myObj = { size: 10, label: "Size 10 Object" };
printLabel(myObj);
```

The type checker checks the call to `printLabel`. The `printLabel` function has a single parameter that requires that the object passed in has a property called `label` of type `string`. Notice that our object actually has more properties than this, but the compiler only checks that *at least* the ones required are present and match the types required. There are some cases where TypeScript isn't as lenient, which we'll cover in a bit.

We can write the same example again, this time using an interface to describe the requirement of having the `label` property that is a string:

```
interface LabeledValue {
  label: string;
}

function printLabel(labeledObj: LabeledValue) {
  console.log(labeledObj.label);
}

let myObj = { size: 10, label: "Size 10 Object" };
printLabel(myObj);
```

The interface `LabeledValue` is a name we can now use to describe the requirement in the previous example. It still represents having a single property called `label` that is of type `string`. Notice we didn't have to explicitly say that the object we pass to `printLabel` implements this interface like we might have to in other languages. Here, it's only the shape that matters. If the object we pass to the function meets the requirements listed, then it's allowed.

It's worth pointing out that the type checker does not require that these properties come in any sort of order, only that the properties the interface requires are present and have the required type.

Optional Properties

Not all properties of an interface may be required. Some exist under certain conditions or may not be there at all. These optional properties are popular when creating patterns like "option bags" where you pass an object to a function that only has a couple of properties filled in.

Here's an example of this pattern:

```
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  let newSquare = { color: "white", area: 100 };
  if (config.color) {
    newSquare.color = config.color;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({ color: "black" });
```

Interfaces with optional properties are written similar to other interfaces, with each optional property denoted by a `?` at the end of the property name in the declaration.

The advantage of optional properties is that you can describe these possibly available properties while still also preventing use of properties that are not part of the interface. For example, had we mistyped the name of the `color` property in `createSquare`, we would get an error message letting us know:

```
// @errors: 2551
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  let newSquare = { color: "white", area: 100 };
  if (config.clor) {
    // Error: Property 'clor' does not exist on type 'SquareConfig'
    newSquare.color = config.clor;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({ color: "black" });
```

Readonly properties

Some properties should only be modifiable when an object is first created. You can specify this by putting `readonly` before the name of the property:

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

You can construct a `Point` by assigning an object literal. After the assignment, `x` and `y` can't be changed.

```
// @errors: 2540
interface Point {
  readonly x: number;
  readonly y: number;
}
// ---cut---
let p1: Point = { x: 10, y: 20 };
p1.x = 5; // error!
```

TypeScript comes with a `ReadonlyArray<T>` type that is the same as `Array<T>` with all mutating methods removed, so you can make sure you don't change your arrays after creation:

```
// @errors: 2542 2339 2540 4104
let a: number[] = [1, 2, 3, 4];
let ro: ReadonlyArray<number> = a;

ro[0] = 12; // error!
ro.push(5); // error!
ro.length = 100; // error!
a = ro; // error!
```

On the last line of the snippet you can see that even assigning the entire `ReadonlyArray` back to a normal array is illegal. You can still override it with a type assertion, though:

```
let a: number[] = [1, 2, 3, 4];
let ro: ReadonlyArray<number> = a;

a = ro as number[];
```

readonly vs const

The easiest way to remember whether to use `readonly` or `const` is to ask whether you're using it on a variable or a property. Variables use `const` whereas properties use `readonly`.

Excess Property Checks

In our first example using interfaces, TypeScript lets us pass `{ size: number; label: string; }` to something that only expected a `{ label: string; }`. We also just learned about optional properties, and how they're useful when describing so-called "option bags".

However, combining the two naively would allow an error to sneak in. For example, taking our last example using `createSquare`:

```
// @errors: 2345 2739
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  return {
    color: config.color || "red",
    area: config.width ? config.width * config.width : 20,
  };
}

let mySquare = createSquare({ colour: "red", width: 100 });
```

Notice the given argument to `createSquare` is spelled `colour` instead of `color`. In plain JavaScript, this sort of thing fails silently.

You could argue that this program is correctly typed, since the `width` properties are compatible, there's no `color` property present, and the extra `colour` property is insignificant.

However, TypeScript takes the stance that there's probably a bug in this code. Object literals get special treatment and undergo *excess property checking* when assigning them to other variables, or passing them as arguments. If an object literal has any properties that the "target type" doesn't have, you'll get an error:

```
// @errors: 2345 2739
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  return {
    color: config.color || "red",
    area: config.width ? config.width * config.width : 20,
  };
}

// ---cut---
let mySquare = createSquare({ colour: "red", width: 100 });
```

Getting around these checks is actually really simple. The easiest method is to just use a type assertion:

```
// @errors: 2345 2739
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  return {
    color: config.color || "red",
    area: config.width ? config.width * config.width : 20,
  };
}
```

```
// ---cut---
let mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig);
```

However, a better approach might be to add a string index signature if you're sure that the object can have some extra properties that are used in some special way. If `SquareConfig` can have `color` and `width` properties with the above types, but could *also* have any number of other properties, then we could define it like so:

```
interface SquareConfig {
  color?: string;
  width?: number;
  [propName: string]: any;
}
```

We'll discuss index signatures in a bit, but here we're saying a `SquareConfig` can have any number of properties, and as long as they aren't `color` or `width`, their types don't matter.

One final way to get around these checks, which might be a bit surprising, is to assign the object to another variable: Since `squareOptions` won't undergo excess property checks, the compiler won't give you an error.

```
interface SquareConfig {
  color?: string;
  width?: number;
  [propName: string]: any;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  return {
    color: config.color || "red",
    area: config.width ? config.width * config.width : 20,
  };
}
// ---cut---
let squareOptions = { colour: "red", width: 100 };
let mySquare = createSquare(squareOptions);
```

The above workaround will work as long as you have a common property between `squareOptions` and `SquareConfig`. In this example, it was the property `width`. It will however, fail if the variable does not have any common object property. For example:

```
// @errors: 2559
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
  return {
    color: config.color || "red",
    area: config.width ? config.width * config.width : 20,
  };
}
```

```
// ---cut---
let squareOptions = { colour: "red" };
let mySquare = createSquare(squareOptions);
```

Keep in mind that for simple code like above, you probably shouldn't be trying to "get around" these checks. For more complex object literals that have methods and hold state, you might need to keep these techniques in mind, but a majority of excess property errors are actually bugs. That means if you're running into excess property checking problems for something like option bags, you might need to revise some of your type declarations. In this instance, if it's okay to pass an object with both a `color` or `colour` property to `createSquare`, you should fix up the definition of `SquareConfig` to reflect that.

Function Types

Interfaces are capable of describing the wide range of shapes that JavaScript objects can take. In addition to describing an object with properties, interfaces are also capable of describing function types.

To describe a function type with an interface, we give the interface a call signature. This is like a function declaration with only the parameter list and return type given. Each parameter in the parameter list requires both name and type.

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
```

Once defined, we can use this function type interface like we would other interfaces. Here, we show how you can create a variable of a function type and assign it a function value of the same type.

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
// ---cut---
let mySearch: SearchFunc;

mySearch = function (source: string, subString: string): boolean {
  let result = source.search(subString);
  return result > -1;
};
```

For function types to correctly type check, the names of the parameters do not need to match. We could have, for example, written the above example like this:

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
// ---cut---
let mySearch: SearchFunc;

mySearch = function (src: string, sub: string): boolean {
  let result = src.search(sub);
  return result > -1;
};
```

Function parameters are checked one at a time, with the type in each corresponding parameter position checked against each other. If you do not want to specify types at all, TypeScript's contextual typing can infer the argument types since the function value is assigned directly to a variable of type `SearchFunc`. Here, also, the return type of our function expression is implied by the values it returns (here `false` and `true`).

```
// @errors: 2322
interface SearchFunc {
  (source: string, subString: string): boolean;
}
// ---cut---
let mySearch: SearchFunc;

mySearch = function (src, sub) {
  let result = src.search(sub);
  return result > -1;
};
```

Had the function expression returned numbers or strings, the type checker would have made an error that indicates return type doesn't match the return type described in the `SearchFunc` interface.

```
// @errors: 2322
interface SearchFunc {
  (source: string, subString: string): boolean;
}
// ---cut---
let mySearch: SearchFunc;

mySearch = function (src, sub) {
  let result = src.search(sub);
  return "string";
};
```

Indexable Types

Similarly to how we can use interfaces to describe function types, we can also describe types that we can "index into" like `a[10]`, or `ageMap["daniel"]`. Indexable types have an *index signature* that describes the types we can use to index into the object, along with the corresponding return types when indexing.

Let's take an example:

```
interface StringArray {
  [index: number]: string;
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];
```

Above, we have a `StringArray` interface that has an index signature. This index signature states that when a `StringArray` is indexed with a `number`, it will return a `string`.

There are four types of supported index signatures: string, number, symbol and template strings. It is possible to support many types of indexers, but the type returned from a numeric indexer must be a subtype of the type returned from the string indexer.

This is because when indexing with a `number`, JavaScript will actually convert that to a `string` before indexing into an object. That means that indexing with `100` (a `number`) is the same thing as indexing with `"100"` (a `string`), so the two need to be consistent.

```
// @errors: 2413
// @strictPropertyInitialization: false
interface Animal {
  name: string;
}

interface Dog extends Animal {
  breed: string;
}

// Error: indexing with a numeric string might get you a completely separate type
// of Animal!
interface NotOkay {
  [x: number]: Animal;
  [x: string]: Dog;
}
```

While string index signatures are a powerful way to describe the "dictionary" pattern, they also enforce that all properties match their return type. This is because a string index declares that `obj.property` is also available as `obj["property"]`. In the following example, `name`'s type does not match the string index's type, and the type checker gives an error:

```
// @errors: 2411
interface NumberDictionary {
  [index: string]: number;

  length: number; // ok, length is a number
  name: string; // error, the type of 'name' is not a subtype of the indexer
}
```

However, properties of different types are acceptable if the index signature is a union of the property types:

```
interface NumberOrStringDictionary {
  [index: string]: number | string;

  length: number; // ok, length is a number
  name: string; // ok, name is a string
}
```

Finally, you can make index signatures `readonly` in order to prevent assignment to their indices:

```
// @errors: 2542
interface ReadonlyStringArray {
  readonly [index: number]: string;
}

let myArray: ReadonlyStringArray = ["Alice", "Bob"];
myArray[2] = "Mallory"; // error!
```

You can't set `myArray[2]` because the index signature is `readonly`.

Indexable Types with Template Strings

A template string can be used to indicate that a particular pattern is allowed, but not all. For example, a `HTTP headers` object may have a set list of known headers and support any [custom defined properties](#) which are prefixed with `x-`.

```
// @errors: 2339

interface HeadersResponse {
  "content-type": string,
  date: string,
  "content-length": string

  // Permit any property starting with 'data-'.
  [headerName: `x-${string}`]: string;
}

function handleResponse(r: HeadersResponse) {
  // Handle known, and x- prefixed
  const type = r["content-type"]
  const poweredBy = r["x-powered-by"]

  // Unknown keys without the prefix raise errors
  const origin = r.origin
}
```

Class Types

Implementing an interface

One of the most common uses of interfaces in languages like C# and Java, that of explicitly enforcing that a class meets a particular contract, is also possible in TypeScript.

```
interface ClockInterface {
  currentTime: Date;
}

class Clock implements ClockInterface {
  currentTime: Date = new Date();
  constructor(h: number, m: number) {}
}
```

You can also describe methods in an interface that are implemented in the class, as we do with `setTime` in the below example:

```
// @strictPropertyInitialization: false
interface ClockInterface {
  currentTime: Date;
  setTime(d: Date): void;
}

class Clock implements ClockInterface {
  currentTime: Date = new Date();
  setTime(d: Date) {
    this.currentTime = d;
```

```

    }
constructor(h: number, m: number) {}
}

```

Interfaces describe the public side of the class, rather than both the public and private side. This prohibits you from using them to check that a class also has particular types for the private side of the class instance.

Difference between the static and instance sides of classes

When working with classes and interfaces, it helps to keep in mind that a class has *two* types: the type of the static side and the type of the instance side. You may notice that if you create an interface with a constructor signature and try to create a class that implements this interface you get an error:

```

// @errors: 7013 2420 2564
// @strictPropertyInitialization: false
// @noImplicitAny: false
interface ClockConstructor {
  new (hour: number, minute: number);
}

class Clock implements ClockConstructor {
  currentTime: Date;
  constructor(h: number, m: number) {}
}

```

This is because when a class implements an interface, only the instance side of the class is checked. Since the constructor sits in the static side, it is not included in this check.

Instead, you would need to work with the static side of the class directly. In this example, we define two interfaces, `ClockConstructor` for the constructor and `ClockInterface` for the instance methods. Then, for convenience, we define a constructor function `createClock` that creates instances of the type that is passed to it:

```

interface ClockConstructor {
  new (hour: number, minute: number): ClockInterface;
}

interface ClockInterface {
  tick(): void;
}

function createClock(
  ctor: ClockConstructor,
  hour: number,
  minute: number
): ClockInterface {
  return new ctor(hour, minute);
}

class DigitalClock implements ClockInterface {
  constructor(h: number, m: number) {}
  tick() {
    console.log("beep beep");
  }
}

```

```

class AnalogClock implements ClockInterface {
  constructor(h: number, m: number) {}
  tick() {
    console.log("tick tock");
  }
}

let digital = createClock(DigitalClock, 12, 17);
let analog = createClock(AnalogClock, 7, 32);

```

Because `createClock`'s first parameter is of type `ClockConstructor`, in `createClock(AnalogClock, 7, 32)`, it checks that `AnalogClock` has the correct constructor signature.

Another simple way is to use class expressions:

```

// @strictPropertyInitialization: false
// @noImplicitAny: false
interface ClockConstructor {
  new (hour: number, minute: number): ClockInterface;
}

interface ClockInterface {
  tick(): void;
}

const Clock: ClockConstructor = class Clock implements ClockInterface {
  constructor(h: number, m: number) {}
  tick() {
    console.log("beep beep");
  }
};

let clock = new Clock(12, 17);
clock.tick();

```

Extending Interfaces

Like classes, interfaces can extend each other. This allows you to copy the members of one interface into another, which gives you more flexibility in how you separate your interfaces into reusable components.

```

interface Shape {
  color: string;
}

interface Square extends Shape {
  sideLength: number;
}

let square = {} as Square;
square.color = "blue";
square.sideLength = 10;

```

An interface can extend multiple interfaces, creating a combination of all of the interfaces.

```

interface Shape {
  color: string;
}

```

```

interface PenStroke {
  penWidth: number;
}

interface Square extends Shape, PenStroke {
  sideLength: number;
}

let square = {} as Square;
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;

```

Hybrid Types

As we mentioned earlier, interfaces can describe the rich types present in real world JavaScript. Because of JavaScript's dynamic and flexible nature, you may occasionally encounter an object that works as a combination of some of the types described above.

One such example is an object that acts as both a function and an object, with additional properties:

```

interface Counter {
  (start: number): string;
  interval: number;
  reset(): void;
}

function getCounter(): Counter {
  let counter = function (start: number) {} as Counter;
  counter.interval = 123;
  counter.reset = function () {};
  return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;

```

When interacting with 3rd-party JavaScript, you may need to use patterns like the above to fully describe the shape of the type.

Interfaces Extending Classes

When an interface type extends a class type it inherits the members of the class but not their implementations. It is as if the interface had declared all of the members of the class without providing an implementation. Interfaces inherit even the private and protected members of a base class. This means that when you create an interface that extends a class with private or protected members, that interface type can only be implemented by that class or a subclass of it.

This is useful when you have a large inheritance hierarchy, but want to specify that your code works with only subclasses that have certain properties. The subclasses don't have to be related besides inheriting from the base class. For example:

```
// @errors: 2300 2420 2300
class Control {
    private state: any;
}

interface SelectableControl extends Control {
    select(): void;
}

class Button extends Control implements SelectableControl {
    select() {}
}

class TextBox extends Control {
    select() {}
}

class ImageControl implements SelectableControl {
    private state: any;
    select() {}
}
```

In the above example, `SelectableControl` contains all of the members of `Control`, including the private `state` property. Since `state` is a private member it is only possible for descendants of `Control` to implement `SelectableControl`. This is because only descendants of `Control` will have a `state` private member that originates in the same declaration, which is a requirement for private members to be compatible.

Within the `Control` class it is possible to access the `state` private member through an instance of `SelectableControl`. Effectively, a `SelectableControl` acts like a `Control` that is known to have a `select` method. The `Button` and `TextBox` classes are subtypes of `SelectableControl` (because they both inherit from `Control` and have a `select` method). The `ImageControl` class has its own `state` private member rather than extending `Control`, so it cannot implement `SelectableControl`.

[Go to TOC](#)

A literal is a more concrete sub-type of a collective type. What this means is that `"Hello World"` is a `string`, but a `string` is not `"Hello World"` inside the type system.

There are three sets of literal types available in TypeScript today: strings, numbers, and booleans; by using literal types you can allow an exact value which a string, number, or boolean must have.

Literal Narrowing

When you declare a variable via `var` or `let`, you are telling the compiler that there is the chance that this variable will change its contents. In contrast, using `const` to declare a variable will inform TypeScript that this object will never change.

```
// We're making a guarantee that this variable
// helloWorld will never change, by using const.

// So, TypeScript sets the type to be "Hello World", not string
const helloWorld = "Hello World";

// On the other hand, a let can change, and so the compiler declares it a string
let hiWorld = "Hi World";
```

The process of going from an infinite number of potential cases (there is an infinite number of possible string values) to a smaller, finite number of potential case (in `helloWorld`'s case: 1) is called narrowing.

String Literal Types

In practice string literal types combine nicely with union types, type guards, and type aliases. You can use these features together to get enum-like behavior with strings.

```
// @errors: 2345
type Easing = "ease-in" | "ease-out" | "ease-in-out";

class UIElement {
  animate(dx: number, dy: number, easing: Easing) {
    if (easing === "ease-in") {
      // ...
    } else if (easing === "ease-out") {
    } else if (easing === "ease-in-out") {
    } else {
      // It's possible that someone could reach this
      // by ignoring your types though.
    }
  }
}

let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy");
```

You can pass any of the three allowed strings, but any other string will give the error

```
Argument of type '"uneasy"' is not assignable to parameter of type '"ease-in" | "ease-out" | "ease-in-out"'
```

String literal types can be used in the same way to distinguish overloads:

```
function createElement(tagName: "img"): HTMLImageElement;
function createElement(tagName: "input"): HTMLInputElement;
// ... more overloads ...
function createElement(tagName: string): Element {
  // ... code goes here ...
}
```

Numeric Literal Types

TypeScript also has numeric literal types, which act the same as the string literals above.

```
function rollDice(): 1 | 2 | 3 | 4 | 5 | 6 {
  return (Math.floor(Math.random() * 6) + 1) as 1 | 2 | 3 | 4 | 5 | 6;
}

const result = rollDice();
```

A common case for their use is for describing config values:

```
/** Creates a map centered at loc/lat */
declare function setupMap(config: MapConfig): void;
// ---cut---
interface MapConfig {
  lng: number;
  lat: number;
  tileSize: 8 | 16 | 32;
}
setupMap({ lng: -73.935242, lat: 40.73061, tileSize: 16 });
```

Boolean Literal Types

TypeScript also has boolean literal types. You might use these to constrain object values whose properties are interrelated.

```
interface ValidationSuccess {
  isValid: true;
  reason: null;
}

interface ValidationFailure {
  isValid: false;
  reason: string;
}
type ValidationResult = ValidationSuccess | ValidationFailure;
```

[Go to TOC](#)

So far, the handbook has covered types which are atomic objects. However, as you model more types you find yourself looking for tools which let you compose or combine existing types instead of creating them from scratch.

Intersection and Union types are one of the ways in which you can compose types.

Union Types

Occasionally, you'll run into a library that expects a parameter to be either a `number` or a `string`. For instance, take the following function:

```
/*
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: any) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${typeof padding}'`);
}

padLeft("Hello world", 4); // returns "    Hello world"
```

The problem with `padLeft` in the above example is that its `padding` parameter is typed as `any`. That means that we can call it with an argument that's neither a `number` nor a `string`, but TypeScript will be okay with it.

```
declare function padLeft(value: string, padding: any): string;
// ---cut---
// passes at compile time, fails at runtime.
let indentedString = padLeft("Hello world", true);
```

In traditional object-oriented code, we might abstract over the two types by creating a hierarchy of types. While this is much more explicit, it's also a little bit overkill. One of the nice things about the original version of `padLeft` was that we were able to just pass in primitives. That meant that usage was simple and concise. This new approach also wouldn't help if we were just trying to use a function that already exists elsewhere.

Instead of `any`, we can use a *union type* for the `padding` parameter:

```
// @errors: 2345
/*
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: string | number) {
  // ...
```

```
}
```

```
let indentedString = padLeft("Hello world", true);
```

A union type describes a value that can be one of several types. We use the vertical bar (`|`) to separate each type, so `number | string | boolean` is the type of a value that can be a `number`, a `string`, or a `boolean`.

Unions with Common Fields

If we have a value that is a union type, we can only access members that are common to all types in the union.

```
// @errors: 2339

interface Bird {
  fly(): void;
  layEggs(): void;
}

interface Fish {
  swim(): void;
  layEggs(): void;
}

declare function getSmallPet(): Fish | Bird;

let pet = getSmallPet();
pet.layEggs();

// Only available in one of the two possible types
pet.swim();
```

Union types can be a bit tricky here, but it just takes a bit of intuition to get used to. If a value has the type `A | B`, we only know for *certain* that it has members that both `A` and `B` have. In this example, `Bird` has a member named `fly`. We can't be sure whether a variable typed as `Bird | Fish` has a `fly` method. If the variable is really a `Fish` at runtime, then calling `pet.fly()` will fail.

Discriminating Unions

A common technique for working with unions is to have a single field which uses literal types which you can use to let TypeScript narrow down the possible current type. For example, we're going to create a union of three types which have a single shared field.

```
type NetworkLoadingState = {
  state: "loading";
};

type NetworkFailedState = {
  state: "failed";
  code: number;
};

type NetworkSuccessState = {
```

```

state: "success";
response: {
  title: string;
  duration: number;
  summary: string;
};
};

// Create a type which represents only one of the above types
// but you aren't sure which it is yet.
type NetworkState =
  | NetworkLoadingState
  | NetworkFailedState
  | NetworkSuccessState;

```

All of the above types have a field named `state`, and then they also have their own fields:

NetworkLoadingState	NetworkFailedState	NetworkSuccessState
state	state	state
	code	response

Given the `state` field is common in every type inside `NetworkState` - it is safe for your code to access without an existence check.

With `state` as a literal type, you can compare the value of `state` to the equivalent string and TypeScript will know which type is currently being used.

NetworkLoadingState	NetworkFailedState	NetworkSuccessState
"loading"	"failed"	"success"

In this case, you can use a `switch` statement to narrow down which type is represented at runtime:

```

// @errors: 2339
type NetworkLoadingState = {
  state: "loading";
};

type NetworkFailedState = {
  state: "failed";
  code: number;
};

type NetworkSuccessState = {
  state: "success";
  response: {
    title: string;
    duration: number;
    summary: string;
  };
};
// ---cut---
type NetworkState =
  | NetworkLoadingState
  | NetworkFailedState
  | NetworkSuccessState;

```

```

| NetworkFailedState
| NetworkSuccessState;

function logger(state: NetworkState): string {
  // Right now TypeScript does not know which of the three
  // potential types state could be.

  // Trying to access a property which isn't shared
  // across all types will raise an error
  state.code;

  // By switching on state, TypeScript can narrow the union
  // down in code flow analysis
  switch (state.state) {
    case "loading":
      return "Downloading...";
    case "failed":
      // The type must be NetworkFailedState here,
      // so accessing the `code` field is safe
      return `Error ${state.code} downloading`;
    case "success":
      return `Downloaded ${state.response.title} - ${state.response.summary}`;
  }
}

```

Union Exhaustiveness checking

We would like the compiler to tell us when we don't cover all variants of the discriminated union. For example, if we add `NetworkFromCachedState` to `NetworkState`, we need to update `logger` as well:

```

// @errors: 2366
type NetworkLoadingState = { state: "loading" };
type NetworkFailedState = { state: "failed"; code: number };
type NetworkSuccessState = {
  state: "success";
  response: {
    title: string;
    duration: number;
    summary: string;
  };
};
// ---cut---
type NetworkFromCachedState = {
  state: "from_cache";
  id: string;
  response: NetworkSuccessState["response"];
};

type NetworkState =
  | NetworkLoadingState
  | NetworkFailedState
  | NetworkSuccessState
  | NetworkFromCachedState;

function logger(s: NetworkState) {
  switch (s.state) {
    case "loading":
      return "loading request";
    case "failed":
      return `failed with code ${s.code}`;
  }
}

```

```

    case "success":
      return "got response";
}

```

There are two ways to do this. The first is to turn on `strictNullChecks` and specify a return type:

```

// @errors: 2366
type NetworkLoadingState = { state: "loading" };
type NetworkFailedState = { state: "failed"; code: number };
type NetworkSuccessState = { state: "success" };
type NetworkFromCachedState = { state: "from_cache" };

type NetworkState =
  | NetworkLoadingState
  | NetworkFailedState
  | NetworkSuccessState
  | NetworkFromCachedState;

// ---cut---
function logger(s: NetworkState): string {
  switch (s.state) {
    case "loading":
      return "loading request";
    case "failed":
      return `failed with code ${s.code}`;
    case "success":
      return "got response";
  }
}

```

Because the `switch` is no longer exhaustive, TypeScript is aware that the function could sometimes return `undefined`. If you have an explicit return type `string`, then you will get an error that the return type is actually `string | undefined`. However, this method is quite subtle and, besides, `strictNullChecks` does not always work with old code.

The second method uses the `never` type that the compiler uses to check for exhaustiveness:

```

// @errors: 2345
type NetworkLoadingState = { state: "loading" };
type NetworkFailedState = { state: "failed"; code: number };
type NetworkSuccessState = { state: "success" };
type NetworkFromCachedState = { state: "from_cache" };

type NetworkState =
  | NetworkLoadingState
  | NetworkFailedState
  | NetworkSuccessState
  | NetworkFromCachedState;
// ---cut---
function assertNever(x: never): never {
  throw new Error("Unexpected object: " + x);
}

function logger(s: NetworkState): string {
  switch (s.state) {
    case "loading":
      return "loading request";
    case "failed":

```

```

    return `failed with code ${s.code}`;
  case "success":
    return "got response";
  default:
    return assertNever(s);
}

```

Here, `assertNever` checks that `s` is of type `never` — the type that's left after all other cases have been removed. If you forget a case, then `s` will have a real type and you will get a type error. This method requires you to define an extra function, but it's much more obvious when you forget it because the error message includes the missing type name.

Intersection Types

Intersection types are closely related to union types, but they are used very differently. An intersection type combines multiple types into one. This allows you to add together existing types to get a single type that has all the features you need. For example, `Person & Serializable & Loggable` is a type which is all of `Person` and `Serializable` and `Loggable`. That means an object of this type will have all members of all three types.

For example, if you had networking requests with consistent error handling then you could separate out the error handling into its own type which is merged with types which correspond to a single response type.

```

interface ErrorHandling {
  success: boolean;
  error?: { message: string };
}

interface ArtworksData {
  artworks: { title: string }[];
}

interface ArtistsData {
  artists: { name: string }[];
}

// These interfaces are composed to have
// consistent error handling, and their own data.

type ArtworksResponse = ArtworksData & ErrorHandling;
type ArtistsResponse = ArtistsData & ErrorHandling;

const handleArtistsResponse = (response: ArtistsResponse) => {
  if (response.error) {
    console.error(response.error.message);
    return;
  }
  console.log(response.artists);
};

```

[Go to TOC](#)

Each and every value in JavaScript has a set of behaviors you can observe from running different operations. That sounds abstract, but as a quick example, consider some operations we might run on a variable named `message`.

```
// Accessing the property 'toLowerCase'
// on 'message' and then calling it
message.toLowerCase();

// Calling 'message'
message();
```

If we break this down, the first runnable line of code accesses a property called `toLowerCase` and then calls it. The second one tries to call `message` directly.

But assuming we don't know the value of `message` - and that's pretty common - we can't reliably say what results we'll get from trying to run any of this code. The behavior of each operation depends entirely on what value we had in the first place.

- Is `message` callable?
- Does it have a property called `toLowerCase` on it?
- If it does, is `toLowerCase` even callable?
- If both of these values are callable, what do they return?

The answers to these questions are usually things we keep in our heads when we write JavaScript, and we have to hope we got all the details right.

Let's say `message` was defined in the following way.

```
const message = "Hello World!";
```

As you can probably guess, if we try to run `message.toLowerCase()`, we'll get the same string only in lower-case.

What about that second line of code? If you're familiar with JavaScript, you'll know this fails with an exception:

```
TypeError: message is not a function
```

It'd be great if we could avoid mistakes like this.

When we run our code, the way that our JavaScript runtime chooses what to do is by figuring out the *type* of the value - what sorts of behaviors and capabilities it has. That's part of what that `TypeError` is alluding to - it's saying that the string `"Hello World!"` cannot be called as a function.

For some values, such as the primitives `string` and `number`, we can identify their type at runtime using the `typeof` operator. But for other things like functions, there's no corresponding runtime mechanism to identify their types. For example, consider this function:

```
function fn(x) {
  return x.flip();
}
```

We can observe by reading the code that this function will only work if given an object with a callable `flip` property, but JavaScript doesn't surface this information in a way that we can check while the code is running. The only way in pure JavaScript to tell what `fn` does with a particular value is to call it and see what happens. This kind of behavior makes it hard to predict what code will do before it runs, which means it's harder to know what your code is going to do while you're writing it.

Seen in this way, a *type* is the concept of describing which values can be passed to `fn` and which will crash. JavaScript only truly provides *dynamic* typing - running the code to see what happens.

The alternative is to use a *static* type system to make predictions about what code is expected *before* it runs.

Static type-checking

Think back to that `TypeError` we got earlier from trying to call a `string` as a function. *Most people* don't like to get any sorts of errors when running their code - those are considered bugs! And when we write new code, we try our best to avoid introducing new bugs.

If we add just a bit of code, save our file, re-run the code, and immediately see the error, we might be able to isolate the problem quickly; but that's not always the case. We might not have tested the feature thoroughly enough, so we might never actually run into a potential error that would be thrown! Or if we were lucky enough to witness the error, we might have ended up doing large refactorings and adding a lot of different code that we're forced to dig through.

Ideally, we could have a tool that helps us find these bugs *before* our code runs. That's what a static type-checker like TypeScript does. *Static types systems* describe the shapes and behaviors of what our values will be when we run our programs. A type-checker like TypeScript uses that information and tells us when things might be going off the rails.

```
// @errors: 2349
const message = "hello!";
message();
```

Running that last sample with TypeScript will give us an error message before we run the code in the first place.

Non-exception Failures

So far we've been discussing certain things like runtime errors - cases where the JavaScript runtime tells us that it thinks something is nonsensical. Those cases come up because [the ECMAScript specification](#) has explicit instructions on how the language should behave when it runs into something unexpected.

For example, the specification says that trying to call something that isn't callable should throw an error. Maybe that sounds like "obvious behavior", but you could imagine that accessing a property that doesn't exist on an object should throw an error too. Instead, JavaScript gives us different behavior and returns the value `undefined`:

```
const user = {
  name: "Daniel",
  age: 26,
};

user.location; // returns undefined
```

Ultimately, a static type system has to make the call over what code should be flagged as an error in its system, even if it's "valid" JavaScript that won't immediately throw an error. In TypeScript, the following code produces an error about `location` not being defined:

```
// @errors: 2339
const user = {
  name: "Daniel",
  age: 26,
};

user.location;
```

While sometimes that implies a trade-off in what you can express, the intent is to catch legitimate bugs in our programs. And TypeScript catches *a lot* of legitimate bugs.

For example: typos,

```
// @noErrors
const announcement = "Hello World!";

// How quickly can you spot the typos?
announcement.toLocalelowercase();
announcement.toLocallowerCase();

// We probably meant to write this...
announcement.toLocaleLowerCase();
```

uncalled functions,

```
// @noUnusedLocals
// @errors: 2365
function flipCoin() {
  // Meant to be Math.random()
  return Math.random < 0.5;
}
```

or basic logic errors.

```
// @errors: 2367
const value = Math.random() < 0.5 ? "a" : "b";
if (value !== "a") {
  // ...
```

```
//> else if (value === "b") {
//>   // Oops, unreachable
//> }
```

Types for Tooling

TypeScript can catch bugs when we make mistakes in our code. That's great, but TypeScript can *also* prevent us from making those mistakes in the first place.

The type-checker has information to check things like whether we're accessing the right properties on variables and other properties. Once it has that information, it can also start *suggesting* which properties you might want to use.

That means TypeScript can be leveraged for editing code too, and the core type-checker can provide error messages and code completion as you type in the editor. That's part of what people often refer to when they talk about tooling in TypeScript.

```
// @noErrors
// @esModuleInterop
import express from "express";
const app = express();

app.get("/", function (req, res) {
  res.send
  //     ^
});

app.listen(3000);
```

TypeScript takes tooling seriously, and that goes beyond completions and errors as you type. An editor that supports TypeScript can deliver "quick fixes" to automatically fix errors, refactorings to easily re-organize code, and useful navigation features for jumping to definitions of a variable, or finding all references to a given variable. All of this is built on top of the type-checker and is fully cross-platform, so it's likely that [your favorite editor has TypeScript support available](#).

tsc, the TypeScript compiler

We've been talking about type-checking, but we haven't yet used our type-*checker*. Let's get acquainted with our new friend `tsc`, the TypeScript compiler. First we'll need to grab it via npm.

```
npm install -g typescript
```

This installs the TypeScript Compiler `tsc` globally. You can use `npx` or similar tools if you'd prefer to run `tsc` from a local `node_modules` package instead.

Now let's move to an empty folder and try writing our first TypeScript program: `hello.ts`:

```
// Greets the world.
console.log("Hello world!");
```

Notice there are no frills here; this "hello world" program looks identical to what you'd write for a "hello world" program in JavaScript. And now let's type-check it by running the command `tsc` which was installed for us by the `typescript` package.

```
tsc hello.ts
```

Tada!

Wait, "tada" *what* exactly? We ran `tsc` and nothing happened! Well, there were no type errors, so we didn't get any output in our console since there was nothing to report.

But check again - we got some *file* output instead. If we look in our current directory, we'll see a `hello.js` file next to `hello.ts`. That's the output from our `hello.ts` file after `tsc` compiles or transforms it into a plain JavaScript file. And if we check the contents, we'll see what TypeScript spits out after it processes a `.ts` file:

```
// Greets the world.
console.log("Hello world!");
```

In this case, there was very little for TypeScript to transform, so it looks identical to what we wrote. The compiler tries to emit clean readable code that looks like something a person would write. While that's not always so easy, TypeScript indents consistently, is mindful of when our code spans across different lines of code, and tries to keep comments around.

What about if we *did* introduce a type-checking error? Let's rewrite `hello.ts`:

```
// @noErrors
// This is an industrial-grade general-purpose greeter function:
function greet(person, date) {
  console.log(`Hello ${person}, today is ${date}!`);
}

greet("Brendan");
```

If we run `tsc hello.ts` again, notice that we get an error on the command line!

```
Expected 2 arguments, but got 1.
```

TypeScript is telling us we forgot to pass an argument to the `greet` function, and rightfully so. So far we've only written standard JavaScript, and yet type-checking was still able to find problems with our code. Thanks TypeScript!

Emitting with Errors

One thing you might not have noticed from the last example was that our `hello.js` file changed again. If we open that file up then we'll see that the contents still basically look the same as our input file. That might be a bit surprising given the fact that `tsc` reported an error about our code, but this is based on one of TypeScript's core values: much of the time, *you will know better than TypeScript*.

To reiterate from earlier, type-checking code limits the sorts of programs you can run, and so there's a tradeoff on what sorts of things a type-checker finds acceptable. Most of the time that's okay, but there are scenarios where those checks get in the way. For example, imagine yourself migrating JavaScript code over to TypeScript and introducing type-checking errors. Eventually you'll get around to cleaning things up for the type-checker, but that original JavaScript code was already working! Why should converting it over to TypeScript stop you from running it?

So TypeScript doesn't get in your way. Of course, over time, you may want to be a bit more defensive against mistakes, and make TypeScript act a bit more strictly. In that case, you can use the `noEmitOnError` compiler option. Try changing your `hello.ts` file and running `tsc` with that flag:

```
tsc --noEmitOnError hello.ts
```

You'll notice that `hello.js` never gets updated.

Explicit Types

Up until now, we haven't told TypeScript what `person` or `date` are. Let's edit the code to tell TypeScript that `person` is a `string`, and that `date` should be a `Date` object. We'll also use the `toDateString()` method on `date`.

```
function greet(person: string, date: Date) {
  console.log(`Hello ${person}, today is ${date.toDateString()}!`);
```

What we did was add *type annotations* on `person` and `date` to describe what types of values `greet` can be called with. You can read that signature as "`greet` takes a `person` of type `string`, and a `date` of type `Date`".

With this, TypeScript can tell us about other cases where `greet` might have been called incorrectly. For example...

```
// @errors: 2345
function greet(person: string, date: Date) {
  console.log(`Hello ${person}, today is ${date.toDateString()}!`);
```

`greet("Maddison", Date());`

Huh? TypeScript reported an error on our second argument, but why?

Perhaps surprisingly, calling `Date()` in JavaScript returns a `string`. On the other hand, constructing a `Date` with `new Date()` actually gives us what we were expecting.

Anyway, we can quickly fix up the error:

```
function greet(person: string, date: Date) {
  console.log(`Hello ${person}, today is ${date.toDateString()}!`);
```

`greet("Maddison", new Date());`

Keep in mind, we don't always have to write explicit type annotations. In many cases, TypeScript can even just *infer* (or "figure out") the types for us even if we omit them.

```
let msg = "hello there!";  
// ^?
```

Even though we didn't tell TypeScript that `msg` had the type `string` it was able to figure that out. That's a feature, and it's best not to add annotations when the type system would end up inferring the same type anyway.

Note: The message bubble inside the previous code sample is what your editor would show if you had hovered over the word.

Erased Types

Let's take a look at what happens when we compile the above function `greet` with `tsc` to output JavaScript:

```
// @showEmit  
// @target: es5  
function greet(person: string, date: Date) {  
    console.log(`Hello ${person}, today is ${date.toDateString()}!`);  
}  
  
greet("Maddison", new Date());
```

Notice two things here:

1. Our `person` and `date` parameters no longer have type annotations.
2. Our "template string" - that string that used backticks (the ``` character) - was converted to plain strings with concatenations.

More on that second point later, but let's now focus on that first point. Type annotations aren't part of JavaScript (or ECMAScript to be pedantic), so there really aren't any browsers or other runtimes that can just run TypeScript unmodified. That's why TypeScript needs a compiler in the first place - it needs some way to strip out or transform any TypeScript-specific code so that you can run it. Most TypeScript-specific code gets erased away, and likewise, here our type annotations were completely erased.

Remember: Type annotations never change the runtime behavior of your program.

Downleveling

One other difference from the above was that our template string was rewritten from

```
`Hello ${person}, today is ${date.toDateString()}!`;
```

to

```
"Hello " + person + ", today is " + date.toDateString() + "!"
```

Why did this happen?

Template strings are a feature from a version of ECMAScript called ECMAScript 2015 (a.k.a. ECMAScript 6, ES2015, ES6, etc. - *don't ask*). TypeScript has the ability to rewrite code from newer versions of ECMAScript to older ones such as ECMAScript 3 or ECMAScript 5 (a.k.a. ES3 and ES5). This process of moving from a newer or "higher" version of ECMAScript down to an older or "lower" one is sometimes called *downleveling*.

By default TypeScript targets ES3, an extremely old version of ECMAScript. We could have chosen something a little bit more recent by using the `target` option. Running with `--target es2015` changes TypeScript to target ECMAScript 2015, meaning code should be able to run wherever ECMAScript 2015 is supported. So running `tsc --target es2015 hello.ts` gives us the following output:

```
function greet(person, date) {
  console.log(`Hello ${person}, today is ${date.toDateString()}!`);
}
greet("Maddison", new Date());
```

While the default target is ES3, the great majority of current browsers support ES2015. Most developers can therefore safely specify ES2015 or above as a target, unless compatibility with certain ancient browsers is important.

Strictness

Different users come to TypeScript looking for different things in a type-checker. Some people are looking for a more loose opt-in experience which can help validate only some parts of their program, and still have decent tooling. This is the default experience with TypeScript, where types are optional, inference takes the most lenient types, and there's no checking for potentially `null` / `undefined` values. Much like how `tsc` emits in the face of errors, these defaults are put in place to stay out of your way. If you're migrating existing JavaScript, that might be a desirable first step.

In contrast, a lot of users prefer to have TypeScript validate as much as it can straight away, and that's why the language provides strictness settings as well. These strictness settings turn static type-checking from a switch (either your code is checked or not) into something closer to a dial. The further you turn this dial up, the more TypeScript will check for you. This can require a little extra work, but generally speaking it pays for itself in the long run, and enables more thorough checks and more accurate tooling. When possible, a new codebase should always turn these strictness checks on.

TypeScript has several type-checking strictness flags that can be turned on or off, and all of our examples will be written with all of them enabled unless otherwise stated. The `strict` flag in the CLI, or `"strict": true` in a `tsconfig.json` toggles them all on simultaneously, but we can opt out of them individually. The two biggest ones you should know about are `noImplicitAny` and `strictNullChecks`.

noImplicitAny

Recall that in some places, TypeScript doesn't try to infer types for us and instead falls back to the most lenient type: `any`. This isn't the worst thing that can happen - after all, falling back to `any` is just the plain JavaScript experience anyway.

However, using `any` often defeats the purpose of using TypeScript in the first place. The more typed your program is, the more validation and tooling you'll get, meaning you'll run into fewer bugs as you code. Turning on the `noImplicitAny` flag will issue an error on any variables whose type is implicitly inferred as `any`.

strictNullChecks

By default, values like `null` and `undefined` are assignable to any other type. This can make writing some code easier, but forgetting to handle `null` and `undefined` is the cause of countless bugs in the world - some consider it a [billion dollar mistake!](#) The `strictNullChecks` flag makes handling `null` and `undefined` more explicit, and *spares* us from worrying about whether we *forgot* to handle `null` and `undefined`.

[Go to TOC](#)

Background Reading:

[Classes \(MDN\)](#)

TypeScript offers full support for the `class` keyword introduced in ES2015.

As with other JavaScript language features, TypeScript adds type annotations and other syntax to allow you to express relationships between classes and other types.

Class Members

Here's the most basic class - an empty one:

```
class Point {}
```

This class isn't very useful yet, so let's start adding some members.

Fields

A field declaration creates a public writeable property on a class:

```
// @strictPropertyInitialization: false
class Point {
  x: number;
  y: number;
}

const pt = new Point();
pt.x = 0;
pt.y = 0;
```

As with other locations, the type annotation is optional, but will be an implicit `any` if not specified.

Fields can also have *initializers*; these will run automatically when the class is instantiated:

```
class Point {
  x = 0;
  y = 0;
}

const pt = new Point();
// Prints 0, 0
console.log(`${pt.x}, ${pt.y}`);
```

Just like with `const`, `let`, and `var`, the initializer of a class property will be used to infer its type:

```
// @errors: 2322
class Point {
  x = 0;
  y = 0;
}
```

```
// ---cut---
const pt = new Point();
pt.x = "0";
```

--strictPropertyInitialization

The `strictPropertyInitialization` setting controls whether class fields need to be initialized in the constructor.

```
// @errors: 2564
class BadGreeter {
    name: string;
}
```

```
class GoodGreeter {
    name: string;

    constructor() {
        this.name = "hello";
    }
}
```

Note that the field needs to be initialized *in the constructor itself*. TypeScript does not analyze methods you invoke from the constructor to detect initializations, because a derived class might override those methods and fail to initialize the members.

If you intend to definitely initialize a field through means other than the constructor (for example, maybe an external library is filling in part of your class for you), you can use the *definite assignment assertion operator*, `!`:

```
class OKGreeter {
    // Not initialized, but no error
    name!: string;
}
```

readonly

Fields may be prefixed with the `readonly` modifier. This prevents assignments to the field outside of the constructor.

```
// @errors: 2540 2540
class Greeter {
    readonly name: string = "world";

    constructor(otherName?: string) {
        if (otherName !== undefined) {
            this.name = otherName;
        }
    }

    err() {
        this.name = "not ok";
    }
}
const g = new Greeter();
g.name = "also not ok";
```

Constructors

Background Reading:
[Constructor \(MDN\)](#)

Class constructors are very similar to functions. You can add parameters with type annotations, default values, and overloads:

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

  // Normal signature with defaults
  constructor(x = 0, y = 0) {
    this.x = x;
    this.y = y;
  }
}
```

```
class Point {
  // Overloads
  constructor(x: number, y: string);
  constructor(s: string);
  constructor(xs: any, y?: any) {
    // TBD
  }
}
```

There are just a few differences between class constructor signatures and function signatures:

- Constructors can't have type parameters - these belong on the outer class declaration, which we'll learn about later
- Constructors can't have return type annotations - the class instance type is always what's returned

Super Calls

Just as in JavaScript, if you have a base class, you'll need to call `super()` in your constructor body before using any `this.` members:

```
// @errors: 17009
class Base {
  k = 4;
}

class Derived extends Base {
  constructor() {
    // Prints a wrong value in ES5; throws exception in ES6
    console.log(this.k);
    super();
  }
}
```

Forgetting to call `super` is an easy mistake to make in JavaScript, but TypeScript will tell you when it's necessary.

Methods

Background Reading:

[Method definitions](#)

A function property on a class is called a *method*. Methods can use all the same type annotations as functions and constructors:

```
class Point {
  x = 10;
  y = 10;

  scale(n: number): void {
    this.x *= n;
    this.y *= n;
  }
}
```

Other than the standard type annotations, TypeScript doesn't add anything else new to methods.

Note that inside a method body, it is still mandatory to access fields and other methods via `this..`. An unqualified name in a method body will always refer to something in the enclosing scope:

```
// @errors: 2322
let x: number = 0;

class C {
  x: string = "hello";

  m() {
    // This is trying to modify 'x' from line 1, not the class property
    x = "world";
  }
}
```

Getters / Setters

Classes can also have *accessors*:

```
class C {
  _length = 0;
  get length(): {
    return this._length;
  }
  set length(value) {
    this._length = value;
  }
}
```

Note that a field-backed get/set pair with no extra logic is very rarely useful in JavaScript. It's fine to expose public fields if you don't need to add additional logic during the get/set operations.

TypeScript has some special inference rules for accessors:

- If `get` exists but no `set`, the property is automatically `readonly`
- If the type of the setter parameter is not specified, it is inferred from the return type of the getter
- Getters and setters must have the same [Member Visibility](#)

Since [TypeScript 4.3](#), it is possible to have accessors with different types for getting and setting.

```
class Thing {
  _size = 0;

  get size(): number {
    return this._size;
  }

  set size(value: string | number | boolean) {
    let num = Number(value);

    // Don't allow NaN, Infinity, etc
    if (!Number.isFinite(num)) {
      this._size = 0;
      return;
    }

    this._size = num;
  }
}
```

Index Signatures

Classes can declare index signatures; these work the same as [Index Signatures for other object types](#):

```
class MyClass {
  [s: string]: boolean | ((s: string) => boolean);

  check(s: string) {
    return this[s] as boolean;
  }
}
```

Because the index signature type needs to also capture the types of methods, it's not easy to usefully use these types. Generally it's better to store indexed data in another place instead of on the class instance itself.

Class Heritage

Like other languages with object-oriented features, classes in JavaScript can inherit from base classes.

implements Clauses

You can use an `implements` clause to check that a class satisfies a particular `interface`. An error will be issued if a class fails to correctly implement it:

```
// @errors: 2420
interface Pingable {
  ping(): void;
}

class Sonar implements Pingable {
  ping() {
    console.log("ping!");
  }
}

class Ball implements Pingable {
  pong() {
    console.log("pong!");
  }
}
```

Classes may also implement multiple interfaces, e.g. `class C implements A, B {`.

Cautions

It's important to understand that an `implements` clause is only a check that the class can be treated as the interface type. It doesn't change the type of the class or its methods *at all*. A common source of error is to assume that an `implements` clause will change the class type - it doesn't!

```
// @errors: 7006
interface Checkable {
  check(name: string): boolean;
}

class NameChecker implements Checkable {
  check(s) {
    // Notice no error here
    return s.toLowerCase() === "ok";
    //           ^
  }
}
```

In this example, we perhaps expected that `s`'s type would be influenced by the `name: string` parameter of `check`. It is not - `implements` clauses don't change how the class body is checked or its type inferred.

Similarly, implementing an interface with an optional property doesn't create that property:

```
// @errors: 2339
interface A {
  x: number;
  y?: number;
}

class C implements A {
  x = 0;
```

```
}
```

```
const c = new C();
c.y = 10;
```

extends Clauses

Background Reading:
[extends keyword \(MDN\)](#)

Classes may `extend` from a base class. A derived class has all the properties and methods of its base class, and also define additional members.

```
class Animal {
  move() {
    console.log("Moving along!");
  }
}

class Dog extends Animal {
  woof(times: number) {
    for (let i = 0; i < times; i++) {
      console.log("woof!");
    }
  }
}

const d = new Dog();
// Base class method
d.move();
// Derived class method
d.woof(3);
```

Overriding Methods

Background Reading:
[super keyword \(MDN\)](#)

A derived class can also override a base class field or property. You can use the `super.` syntax to access base class methods. Note that because JavaScript classes are a simple lookup object, there is no notion of a "super field".

TypeScript enforces that a derived class is always a subtype of its base class.

For example, here's a legal way to override a method:

```
class Base {
  greet() {
    console.log("Hello, world!");
  }
}
```

```

class Derived extends Base {
  greet(name?: string) {
    if (name === undefined) {
      super.greet();
    } else {
      console.log(`Hello, ${name.toUpperCase()}`);
    }
  }

  const d = new Derived();
  d.greet();
  d.greet("reader");
}

```

It's important that a derived class follow its base class contract. Remember that it's very common (and always legal!) to refer to a derived class instance through a base class reference:

```

class Base {
  greet() {
    console.log("Hello, world!");
  }
}

declare const d: Base;
// ---cut---
// Alias the derived instance through a base class reference
const b: Base = d;
// No problem
b.greet();

```

What if `Derived` didn't follow `Base`'s contract?

```

// @errors: 2416
class Base {
  greet() {
    console.log("Hello, world!");
  }
}

class Derived extends Base {
  // Make this parameter required
  greet(name: string) {
    console.log(`Hello, ${name.toUpperCase()}`);
  }
}

```

If we compiled this code despite the error, this sample would then crash:

```

declare class Base {
  greet(): void;
}

declare class Derived extends Base {}

// ---cut---
const b: Base = new Derived();
// Crashes because "name" will be undefined
b.greet();

```

Type-only Field Declarations

When `target >= ES2022` or `useDefineForClassFields` is `true`, class fields are initialized after the parent class constructor completes, overwriting any value set by the parent class. This can be a problem when you only want to re-declare a more accurate type for an inherited field. To handle these cases, you can write `declare` to indicate to TypeScript that there should be no runtime effect for this field declaration.

```
interface Animal {
  dateOfBirth: any;
}

interface Dog extends Animal {
  breed: any;
}

class AnimalHouse {
  resident: Animal;
  constructor(animal: Animal) {
    this.resident = animal;
  }
}

class DogHouse extends AnimalHouse {
  // Does not emit JavaScript code,
  // only ensures the types are correct
  declare resident: Dog;
  constructor(dog: Dog) {
    super(dog);
  }
}
```

Initialization Order

The order that JavaScript classes initialize can be surprising in some cases. Let's consider this code:

```
class Base {
  name = "base";
  constructor() {
    console.log("My name is " + this.name);
  }
}

class Derived extends Base {
  name = "derived";
}

// Prints "base", not "derived"
const d = new Derived();
```

What happened here?

The order of class initialization, as defined by JavaScript, is:

- The base class fields are initialized
- The base class constructor runs
- The derived class fields are initialized

- The derived class constructor runs

This means that the base class constructor saw its own value for `name` during its own constructor, because the derived class field initializations hadn't run yet.

Inheriting Built-in Types

Note: If you don't plan to inherit from built-in types like `Array`, `Error`, `Map`, etc. or your compilation target is explicitly set to `ES6 / ES2015` or above, you may skip this section

In ES2015, constructors which return an object implicitly substitute the value of `this` for any callers of `super(...)`. It is necessary for generated constructor code to capture any potential return value of `super(...)` and replace it with `this`.

As a result, subclassing `Error`, `Array`, and others may no longer work as expected. This is due to the fact that constructor functions for `Error`, `Array`, and the like use ECMAScript 6's `new.target` to adjust the prototype chain; however, there is no way to ensure a value for `new.target` when invoking a constructor in ECMAScript 5. Other downlevel compilers generally have the same limitation by default.

For a subclass like the following:

```
class MsgError extends Error {
  constructor(m: string) {
    super(m);
  }
  sayHello() {
    return "hello " + this.message;
  }
}
```

you may find that:

- methods may be `undefined` on objects returned by constructing these subclasses, so calling `sayHello` will result in an error.
- `instanceof` will be broken between instances of the subclass and their instances, so `(new MsgError()) instanceof MsgError` will return `false`.

As a recommendation, you can manually adjust the prototype immediately after any `super(...)` calls.

```
class MsgError extends Error {
  constructor(m: string) {
    super(m);

    // Set the prototype explicitly.
    Object.setPrototypeOf(this, MsgError.prototype);
  }
  sayHello() {
    return "hello " + this.message;
  }
}
```

However, any subclass of `MsgError` will have to manually set the prototype as well. For runtimes that don't support `Object.setPrototypeOf`, you may instead be able to use `__proto__`.

Unfortunately, these workarounds will not work on Internet Explorer 10 and prior. One can manually copy methods from the prototype onto the instance itself (i.e. `MsgError.prototype` onto `this`), but the prototype chain itself cannot be fixed.

Member Visibility

You can use TypeScript to control whether certain methods or properties are visible to code outside the class.

`public`

The default visibility of class members is `public`. A `public` member can be accessed anywhere:

```
class Greeter {
  public greet() {
    console.log("hi!");
  }
}
const g = new Greeter();
g.greet();
```

Because `public` is already the default visibility modifier, you don't ever *need* to write it on a class member, but might choose to do so for style/readability reasons.

`protected`

`protected` members are only visible to subclasses of the class they're declared in.

```
// @errors: 2445
class Greeter {
  public greet() {
    console.log("Hello, " + this.getName());
  }
  protected getName() {
    return "hi";
  }
}

class SpecialGreeter extends Greeter {
  public howdy() {
    // OK to access protected member here
    console.log("Howdy, " + this.getName());
    // ^^^^^^
  }
}
const g = new SpecialGreeter();
g.greet(); // OK
g.getName();
```

Exposure of `protected` members

Derived classes need to follow their base class contracts, but may choose to expose a subtype of base class with more capabilities. This includes making `protected` members `public`:

```
class Base {
  protected m = 10;
}
class Derived extends Base {
  // No modifier, so default is 'public'
  m = 15;
}
const d = new Derived();
console.log(d.m); // OK
```

Note that `Derived` was already able to freely read and write `m`, so this doesn't meaningfully alter the "security" of this situation. The main thing to note here is that in the derived class, we need to be careful to repeat the `protected` modifier if this exposure isn't intentional.

Cross-hierarchy `protected` access

Different OOP languages disagree about whether it's legal to access a `protected` member through a base class reference:

```
// @errors: 2446
class Base {
  protected x: number = 1;
}
class Derived1 extends Base {
  protected x: number = 5;
}
class Derived2 extends Base {
  f1(other: Derived2) {
    other.x = 10;
  }
  f2(other: Base) {
    other.x = 10;
  }
}
```

Java, for example, considers this to be legal. On the other hand, C# and C++ chose that this code should be illegal.

TypeScript sides with C# and C++ here, because accessing `x` in `Derived2` should only be legal from `Derived2`'s subclasses, and `Derived1` isn't one of them. Moreover, if accessing `x` through a `Derived1` reference is illegal (which it certainly should be!), then accessing it through a base class reference should never improve the situation.

See also [Why Can't I Access A Protected Member From A Derived Class?](#) which explains more of C#'s reasoning.

`private`

`private` is like `protected`, but doesn't allow access to the member even from subclasses:

```
// @errors: 2341
class Base {
    private x = 0;
}
const b = new Base();
// Can't access from outside the class
console.log(b.x);
```

```
// @errors: 2341
class Base {
    private x = 0;
}
// ---cut---
class Derived extends Base {
    showX() {
        // Can't access in subclasses
        console.log(this.x);
    }
}
```

Because `private` members aren't visible to derived classes, a derived class can't increase its visibility:

```
// @errors: 2415
class Base {
    private x = 0;
}
class Derived extends Base {
    x = 1;
}
```

Cross-instance `private` access

Different OOP languages disagree about whether different instances of the same class may access each others' `private` members. While languages like Java, C#, C++, Swift, and PHP allow this, Ruby does not.

TypeScript does allow cross-instance `private` access:

```
class A {
    private x = 10;

    public sameAs(other: A) {
        // No error
        return other.x === this.x;
    }
}
```

Caveats

Like other aspects of TypeScript's type system, `private` and `protected` are only enforced during type checking.

This means that JavaScript runtime constructs like `in` or simple property lookup can still access a `private` or `protected` member:

```
class MySafe {
    private secretKey = 12345;
```

```
// In a JavaScript file...
const s = new MySafe();
// Will print 12345
console.log(s.secretKey);
```

`private` also allows access using bracket notation during type checking. This makes `private`-declared fields potentially easier to access for things like unit tests, with the drawback that these fields are *soft private* and don't strictly enforce privacy.

```
// @errors: 2341
class MySafe {
    private secretKey = 12345;
}

const s = new MySafe();

// Not allowed during type checking
console.log(s.secretKey);

// OK
console.log(s["secretKey"]);
```

Unlike TypeScript's `private`, JavaScript's `private fields` (#) remain private after compilation and do not provide the previously mentioned escape hatches like bracket notation access, making them *hard private*.

```
class Dog {
    #barkAmount = 0;
    personality = "happy";

    constructor() {}
}
```

```
// @target: esnext
// @showEmit
class Dog {
    #barkAmount = 0;
    personality = "happy";

    constructor() {}
}
```

When compiling to ES2021 or less, TypeScript will use WeakMaps in place of #.

```
// @target: es2015
// @showEmit
class Dog {
    #barkAmount = 0;
    personality = "happy";

    constructor() {}
}
```

If you need to protect values in your class from malicious actors, you should use mechanisms that offer hard runtime privacy, such as closures, WeakMaps, or private fields. Note that these added privacy checks during runtime could affect performance.

Static Members

Background Reading:

[Static Members \(MDN\)](#)

Classes may have `static` members. These members aren't associated with a particular instance of the class. They can be accessed through the class constructor object itself:

```
class MyClass {
  static x = 0;
  static printX() {
    console.log(MyClass.x);
  }
}
console.log(MyClass.x);
MyClass.printX();
```

Static members can also use the same `public`, `protected`, and `private` visibility modifiers:

```
// @errors: 2341
class MyClass {
  private static x = 0;
}
console.log(MyClass.x);
```

Static members are also inherited:

```
class Base {
  static getGreeting() {
    return "Hello world";
  }
}
class Derived extends Base {
  myGreeting = Derived.getGreeting();
}
```

Special Static Names

It's generally not safe/possible to overwrite properties from the `Function` prototype. Because classes are themselves functions that can be invoked with `new`, certain `static` names can't be used. Function properties like `name`, `length`, and `call` aren't valid to define as `static` members:

```
// @errors: 2699
class S {
  static name = "S!";
}
```

Why No Static Classes?

TypeScript (and JavaScript) don't have a construct called `static class` the same way as, for example, C# does.

Those constructs *only* exist because those languages force all data and functions to be inside a class; because that restriction doesn't exist in TypeScript, there's no need for them. A class with only a single instance is typically just represented as a normal *object* in JavaScript/TypeScript.

For example, we don't need a "static class" syntax in TypeScript because a regular object (or even top-level function) will do the job just as well:

```
// Unnecessary "static" class
class MyStaticClass {
    static doSomething() {}
}

// Preferred (alternative 1)
function doSomething() {}

// Preferred (alternative 2)
const MyHelperObject = {
    doSomething(),
};


```

static Blocks in Classes

Static blocks allow you to write a sequence of statements with their own scope that can access private fields within the containing class. This means that we can write initialization code with all the capabilities of writing statements, no leakage of variables, and full access to our class's internals.

```
declare function loadLastInstances(): any[]
// ---cut---
class Foo {
    static #count = 0;

    get count() {
        return Foo.#count;
    }

    static {
        try {
            const lastInstances = loadLastInstances();
            Foo.#count += lastInstances.length;
        }
        catch {}
    }
}
```

Generic Classes

Classes, much like interfaces, can be generic. When a generic class is instantiated with `new`, its type parameters are inferred the same way as in a function call:

```
class Box<Type> {
    contents: Type;
    constructor(value: Type) {
        this.contents = value;
    }
}
```

```
// const b = new Box("hello!");
//   ^?
```

Classes can use generic constraints and defaults the same way as interfaces.

Type Parameters in Static Members

This code isn't legal, and it may not be obvious why:

```
// @errors: 2302
class Box<Type> {
  static defaultValue: Type;
}
```

Remember that types are always fully erased! At runtime, there's only one `Box.defaultValue` property slot. This means that setting `Box<string>.defaultValue` (if that were possible) would also change `Box<number>.defaultValue` - not good. The `static` members of a generic class can never refer to the class's type parameters.

this at Runtime in Classes

Background Reading:

[this keyword \(MDN\)](#)

It's important to remember that TypeScript doesn't change the runtime behavior of JavaScript, and that JavaScript is somewhat famous for having some peculiar runtime behaviors.

JavaScript's handling of `this` is indeed unusual:

```
class MyClass {
  name = "MyClass";
  getName() {
    return this.name;
  }
}
const c = new MyClass();
const obj = {
  name: "obj",
  getName: c.getName,
};

// Prints "obj", not "MyClass"
console.log(obj.getName());
```

Long story short, by default, the value of `this` inside a function depends on *how the function was called*. In this example, because the function was called through the `obj` reference, its value of `this` was `obj` rather than the class instance.

This is rarely what you want to happen! TypeScript provides some ways to mitigate or prevent this kind of error.

Arrow Functions

Background Reading:
[Arrow functions \(MDN\)](#)

If you have a function that will often be called in a way that loses its `this` context, it can make sense to use an arrow function property instead of a method definition:

```
class MyClass {
  name = "MyClass";
  getName = () => {
    return this.name;
  };
}
const c = new MyClass();
const g = c.getName;
// Prints "MyClass" instead of crashing
console.log(g());
```

This has some trade-offs:

- The `this` value is guaranteed to be correct at runtime, even for code not checked with TypeScript
- This will use more memory, because each class instance will have its own copy of each function defined this way
- You can't use `super.getName` in a derived class, because there's no entry in the prototype chain to fetch the base class method from

`this` parameters

In a method or function definition, an initial parameter named `this` has special meaning in TypeScript. These parameters are erased during compilation:

```
type SomeType = any;
// ---cut---
// TypeScript input with 'this' parameter
function fn(this: SomeType, x: number) {
  /* ... */
}

// JavaScript output
function fn(x) {
  /* ... */
}
```

TypeScript checks that calling a function with a `this` parameter is done so with a correct context. Instead of using an arrow function, we can add a `this` parameter to method definitions to statically enforce that the method is called correctly:

```
// @errors: 2684
class MyClass {
  name = "MyClass";
```

```

    getName(this: MyClass) {
      return this.name;
    }
}
const c = new MyClass();
// OK
c.getName();

// Error, would crash
const g = c.getName;
console.log(g());

```

This method makes the opposite trade-offs of the arrow function approach:

- JavaScript callers might still use the class method incorrectly without realizing it
- Only one function per class definition gets allocated, rather than one per class instance
- Base method definitions can still be called via `super`.

this Types

In classes, a special type called `this` refers *dynamically* to the type of the current class. Let's see how this is useful:

```

class Box {
  contents: string = "";
  set(value: string) {
    // ^
    this.contents = value;
    return this;
  }
}

```

Here, TypeScript inferred the return type of `set` to be `this`, rather than `Box`. Now let's make a subclass of `Box`:

```

class Box {
  contents: string = "";
  set(value: string) {
    this.contents = value;
    return this;
  }
}
// ---cut---
class ClearableBox extends Box {
  clear() {
    this.contents = "";
  }
}

const a = new ClearableBox();
const b = a.set("hello");
// ^

```

You can also use `this` in a parameter type annotation:

```
class Box {
  content: string = "";
  sameAs(other: this) {
    return other.content === this.content;
  }
}
```

This is different from writing `other: Box` -- if you have a derived class, its `sameAs` method will now only accept other instances of that same derived class:

```
// @errors: 2345
class Box {
  content: string = "";
  sameAs(other: this) {
    return other.content === this.content;
  }
}

class DerivedBox extends Box {
  otherContent: string = "?";
}

const base = new Box();
const derived = new DerivedBox();
derived.sameAs(base);
```

this-based type guards

You can use `this is Type` in the return position for methods in classes and interfaces. When mixed with a type narrowing (e.g. `if` statements) the type of the target object would be narrowed to the specified `Type`.

```
// @strictPropertyInitialization: false
class FileSystemObject {
  isFile(): this is FileRep {
    return this instanceof FileRep;
  }
  isDirectory(): this is Directory {
    return this instanceof Directory;
  }
  isNetworked(): this is Networked & this {
    return this.networked;
  }
  constructor(public path: string, private networked: boolean) {}

  class FileRep extends FileSystemObject {
    constructor(path: string, public content: string) {
      super(path, false);
    }
  }

  class Directory extends FileSystemObject {
    children: FileSystemObject[];
  }

  interface Networked {
    host: string;
  }
}
```

```
const fso: FileSystemObject = new FileRep("foo/bar.txt", "foo");

if (fso.isFile()) {
  fso.content;
// ^
} else if (fso.isDirectory()) {
  fso.children;
// ^
} else if (fso.isNetworked()) {
  fso.host;
// ^
}
```

A common use-case for a this-based type guard is to allow for lazy validation of a particular field. For example, this case removes an `undefined` from the value held inside box when `hasValue` has been verified to be true:

```
class Box<T> {
  value?: T;

  hasValue(): this is { value: T } {
    return this.value !== undefined;
  }
}

const box = new Box();
box.value = "Gameboy";

box.value;
// ^?

if (box.hasValue()) {
  box.value;
// ^
}
```

Parameter Properties

TypeScript offers special syntax for turning a constructor parameter into a class property with the same name and value. These are called *parameter properties* and are created by prefixing a constructor argument with one of the visibility modifiers `public`, `private`, `protected`, or `readonly`. The resulting field gets those modifier(s):

```
// @errors: 2341
class Params {
  constructor(
    public readonly x: number,
    protected y: number,
    private z: number
  ) {
    // No body necessary
  }
}
const a = new Params(1, 2, 3);
console.log(a.x);
// ^
console.log(a.z);
```

Class Expressions

Background Reading:

[Class expressions \(MDN\)](#)

Class expressions are very similar to class declarations. The only real difference is that class expressions don't need a name, though we can refer to them via whatever identifier they ended up bound to:

```
const someClass = class<Type> {
  content: Type;
  constructor(value: Type) {
    this.content = value;
  }
};

const m = new someClass("Hello, world");
//   ^?
```

abstract Classes and Members

Classes, methods, and fields in TypeScript may be *abstract*.

An *abstract method* or *abstract field* is one that hasn't had an implementation provided. These members must exist inside an *abstract class*, which cannot be directly instantiated.

The role of abstract classes is to serve as a base class for subclasses which do implement all the abstract members. When a class doesn't have any abstract members, it is said to be *concrete*.

Let's look at an example:

```
// @errors: 2511
abstract class Base {
  abstract getName(): string;

  printName() {
    console.log("Hello, " + this.getName());
  }
}

const b = new Base();
```

We can't instantiate `Base` with `new` because it's abstract. Instead, we need to make a derived class and implement the abstract members:

```
abstract class Base {
  abstract getName(): string;
  printName() {}
}

// ---cut---
class Derived extends Base {
  getName() {
    return "world";
```

```

    }
}

const d = new Derived();
d.printName();

```

Notice that if we forget to implement the base class's abstract members, we'll get an error:

```

// @errors: 2515
abstract class Base {
  abstract getName(): string;
  printName() {}
}
// ---cut---
class Derived extends Base {
  // forgot to do anything
}

```

Abstract Construct Signatures

Sometimes you want to accept some class constructor function that produces an instance of a class which derives from some abstract class.

For example, you might want to write this code:

```

// @errors: 2511
abstract class Base {
  abstract getName(): string;
  printName() {}
}
class Derived extends Base {
  getName() {
    return "";
  }
}
// ---cut---
function greet(ctor: typeof Base) {
  const instance = new ctor();
  instance.printName();
}

```

TypeScript is correctly telling you that you're trying to instantiate an abstract class. After all, given the definition of `greet`, it's perfectly legal to write this code, which would end up constructing an abstract class:

```

declare const greet: any, Base: any;
// ---cut---
// Bad!
greet(Base);

```

Instead, you want to write a function that accepts something with a construct signature:

```

// @errors: 2345
abstract class Base {
  abstract getName(): string;
  printName() {}
}
class Derived extends Base {

```

```

    getName() {
      return "";
    }
}
// ---cut---
function greet(ctor: new () => Base) {
  const instance = new ctor();
  instance.printName();
}
greet(Derived);
greet(Base);

```

Now TypeScript correctly tells you about which class constructor functions can be invoked - `Derived` can because it's concrete, but `Base` cannot.

Relationships Between Classes

In most cases, classes in TypeScript are compared structurally, the same as other types.

For example, these two classes can be used in place of each other because they're identical:

```

class Point1 {
  x = 0;
  y = 0;
}

class Point2 {
  x = 0;
  y = 0;
}

// OK
const p: Point1 = new Point2();

```

Similarly, subtype relationships between classes exist even if there's no explicit inheritance:

```

// @strict: false
class Person {
  name: string;
  age: number;
}

class Employee {
  name: string;
  age: number;
  salary: number;
}

// OK
const p: Person = new Employee();

```

This sounds straightforward, but there are a few cases that seem stranger than others.

Empty classes have no members. In a structural type system, a type with no members is generally a supertype of anything else. So if you write an empty class (don't!), anything can be used in place of it:

```
class Empty {}

function fn(x: Empty) {
  // can't do anything with 'x', so I won't
}

// All OK!
fn(window);
fn({});
fn(fn);
```

[Go to TOC](#)

In this chapter, we'll cover some of the most common types of values you'll find in JavaScript code, and explain the corresponding ways to describe those types in TypeScript. This isn't an exhaustive list, and future chapters will describe more ways to name and use other types.

Types can also appear in many more *places* than just type annotations. As we learn about the types themselves, we'll also learn about the places where we can refer to these types to form new constructs.

We'll start by reviewing the most basic and common types you might encounter when writing JavaScript or TypeScript code. These will later form the core building blocks of more complex types.

The primitives: `string`, `number`, and `boolean`

JavaScript has three very commonly used **primitives**: `string`, `number`, and `boolean`. Each has a corresponding type in TypeScript. As you might expect, these are the same names you'd see if you used the JavaScript `typeof` operator on a value of those types:

- `string` represents string values like `"Hello, world"`
- `number` is for numbers like `42`. JavaScript does not have a special runtime value for integers, so there's no equivalent to `int` or `float` - everything is simply `number`
- `boolean` is for the two values `true` and `false`

The type names `String`, `Number`, and `Boolean` (starting with capital letters) are legal, but refer to some special built-in types that will very rarely appear in your code. Always use `string`, `number`, or `boolean` for types.

Arrays

To specify the type of an array like `[1, 2, 3]`, you can use the syntax `number[]`; this syntax works for any type (e.g. `string[]` is an array of strings, and so on). You may also see this written as `Array<number>`, which means the same thing. We'll learn more about the syntax `T<U>` when we cover *generics*.

Note that `[number]` is a different thing; refer to the section on [Tuples](#).

any

TypeScript also has a special type, `any`, that you can use whenever you don't want a particular value to cause typechecking errors.

When a value is of type `any`, you can access any properties of it (which will in turn be of type `any`), call it like a function, assign it to (or from) a value of any type, or pretty much anything else that's syntactically legal:

```
let obj: any = { x: 0 };
// None of the following lines of code will throw compiler errors.
// Using 'any' disables all further type checking, and it is assumed
// you know the environment better than TypeScript.
obj.foo();
obj();
obj.bar = 100;
obj = "hello";
const n: number = obj;
```

The `any` type is useful when you don't want to write out a long type just to convince TypeScript that a particular line of code is okay.

`noImplicitAny`

When you don't specify a type, and TypeScript can't infer it from context, the compiler will typically default to `any`.

You usually want to avoid this, though, because `any` isn't type-checked. Use the compiler flag `noImplicitAny` to flag any implicit `any` as an error.

Type Annotations on Variables

When you declare a variable using `const`, `var`, or `let`, you can optionally add a type annotation to explicitly specify the type of the variable:

```
let myName: string = "Alice";
// ^^^^^^ Type annotation
```

TypeScript doesn't use "types on the left"-style declarations like `int x = 0;`. Type annotations will always go *after* the thing being typed.

In most cases, though, this isn't needed. Wherever possible, TypeScript tries to automatically *infer* the types in your code. For example, the type of a variable is inferred based on the type of its initializer:

```
// No type annotation needed -- 'myName' inferred as type 'string'
let myName = "Alice";
```

For the most part you don't need to explicitly learn the rules of inference. If you're starting out, try using fewer type annotations than you think - you might be surprised how few you need for TypeScript to fully understand what's going on.

Functions

Functions are the primary means of passing data around in JavaScript. TypeScript allows you to specify the types of both the input and output values of functions.

Parameter Type Annotations

When you declare a function, you can add type annotations after each parameter to declare what types of parameters the function accepts. Parameter type annotations go after the parameter name:

```
// Parameter type annotation
function greet(name: string) {
  // ^^^^^^
  console.log("Hello, " + name.toUpperCase() + "!!");
}
```

When a parameter has a type annotation, arguments to that function will be checked:

```
// @errors: 2345
declare function greet(name: string): void;
// ---cut---
// Would be a runtime error if executed!
greet(42);
```

Even if you don't have type annotations on your parameters, TypeScript will still check that you passed the right number of arguments.

Return Type Annotations

You can also add return type annotations. Return type annotations appear after the parameter list:

```
function getFavoriteNumber(): number {
  // ^^^^^^^^
  return 26;
}
```

Much like variable type annotations, you usually don't need a return type annotation because TypeScript will infer the function's return type based on its `return` statements. The type annotation in the above example doesn't change anything. Some codebases will explicitly specify a return type for documentation purposes, to prevent accidental changes, or just for personal preference.

Anonymous Functions

Anonymous functions are a little bit different from function declarations. When a function appears in a place where TypeScript can determine how it's going to be called, the parameters of that function are automatically given types.

Here's an example:

```
// @errors: 2551
// No type annotations here, but TypeScript can spot the bug
const names = ["Alice", "Bob", "Eve"];

// Contextual typing for function
names.forEach(function (s) {
    console.log(s.toUpperCase());
});

// Contextual typing also applies to arrow functions
names.forEach((s) => {
    console.log(s.toUpperCase());
});
```

Even though the parameter `s` didn't have a type annotation, TypeScript used the types of the `forEach` function, along with the inferred type of the array, to determine the type `s` will have.

This process is called *contextual typing* because the *context* that the function occurred within informs what type it should have.

Similar to the inference rules, you don't need to explicitly learn how this happens, but understanding that it does happen can help you notice when type annotations aren't needed. Later, we'll see more examples of how the context that a value occurs in can affect its type.

Object Types

Apart from primitives, the most common sort of type you'll encounter is an *object type*. This refers to any JavaScript value with properties, which is almost all of them! To define an object type, we simply list its properties and their types.

For example, here's a function that takes a point-like object:

```
// The parameter's type annotation is an object type
function printCoord(pt: { x: number; y: number }) {
  //
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}
printCoord({ x: 3, y: 7 });
```

Here, we annotated the parameter with a type with two properties - `x` and `y` - which are both of type `number`. You can use `,` or `;` to separate the properties, and the last separator is optional either way.

The type part of each property is also optional. If you don't specify a type, it will be assumed to be `any`.

Optional Properties

Object types can also specify that some or all of their properties are *optional*. To do this, add a `?` after the property name:

```
function printName(obj: { first: string; last?: string }) {  
    // ...  
}
```

```
// Both OK
printName({ first: "Bob" });
printName({ first: "Alice", last: "Alisson" });
```

In JavaScript, if you access a property that doesn't exist, you'll get the value `undefined` rather than a run-time error. Because of this, when you *read* from an optional property, you'll have to check for `undefined` before using it.

```
// @errors: 2532
function printName(obj: { first: string; last?: string }) {
  // Error - might crash if 'obj.last' wasn't provided!
  console.log(obj.last.toUpperCase());
  if (obj.last !== undefined) {
    // OK
    console.log(obj.last.toUpperCase());
  }
  // A safe alternative using modern JavaScript syntax:
  console.log(obj.last?.toUpperCase());
}
```

Union Types

TypeScript's type system allows you to build new types out of existing ones using a large variety of operators. Now that we know how to write a few types, it's time to start *combining* them in interesting ways.

Defining a Union Type

The first way to combine types you might see is a *union* type. A union type is a type formed from two or more other types, representing values that may be *any one* of those types. We refer to each of these types as the union's *members*.

Let's write a function that can operate on strings or numbers:

```
// @errors: 2345
function printId(id: number | string) {
  console.log("Your ID is: " + id);
}
// OK
printId(101);
// OK
printId("202");
// Error
printId({ myID: 22342 });
```

Working with Union Types

It's easy to *provide* a value matching a union type - simply provide a type matching any of the union's members. If you *have* a value of a union type, how do you work with it?

TypeScript will only allow an operation if it is valid for *every* member of the union. For example, if you have the union `string | number`, you can't use methods that are only available on `string`:

```
// @errors: 2339
function printId(id: number | string) {
  console.log(id.toUpperCase());
}
```

The solution is to *narrow* the union with code, the same as you would in JavaScript without type annotations. *Narrowing* occurs when TypeScript can deduce a more specific type for a value based on the structure of the code.

For example, TypeScript knows that only a `string` value will have a `typeof` value `"string"`:

```
function printId(id: number | string) {
  if (typeof id === "string") {
    // In this branch, id is of type 'string'
    console.log(id.toUpperCase());
  } else {
    // Here, id is of type 'number'
    console.log(id);
  }
}
```

Another example is to use a function like `Array.isArray`:

```
function welcomePeople(x: string[] | string) {
  if (Array.isArray(x)) {
    // Here: 'x' is 'string[]'
    console.log("Hello, " + x.join(" and "));
  } else {
    // Here: 'x' is 'string'
    console.log("Welcome lone traveler " + x);
  }
}
```

Notice that in the `else` branch, we don't need to do anything special - if `x` wasn't a `string[]`, then it must have been a `string`.

Sometimes you'll have a union where all the members have something in common. For example, both arrays and strings have a `slice` method. If every member in a union has a property in common, you can use that property without narrowing:

```
// Return type is inferred as number[] | string
function getFirstThree(x: number[] | string) {
  return x.slice(0, 3);
}
```

It might be confusing that a *union* of types appears to have the *intersection* of those types' properties. This is not an accident - the name *union* comes from type theory. The `union` `number | string` is composed by taking the *union of the values* from each type. Notice that given two sets with corresponding facts about each set, only the *intersection* of those facts applies to the *union* of the sets

themselves. For example, if we had a room of tall people wearing hats, and another room of Spanish speakers wearing hats, after combining those rooms, the only thing we know about every person is that they must be wearing a hat.

Type Aliases

We've been using object types and union types by writing them directly in type annotations. This is convenient, but it's common to want to use the same type more than once and refer to it by a single name.

A *type alias* is exactly that - a *name* for any *type*. The syntax for a type alias is:

```
type Point = {
  x: number;
  y: number;
};

// Exactly the same as the earlier example
function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

You can actually use a type alias to give a name to any type at all, not just an object type. For example, a type alias can name a union type:

```
type ID = number | string;
```

Note that aliases are *only* aliases - you cannot use type aliases to create different/distinct "versions" of the same type. When you use the alias, it's exactly as if you had written the aliased type. In other words, this code might *look* illegal, but is OK according to TypeScript because both types are aliases for the same type:

```
declare function getInput(): string;
declare function sanitize(str: string): string;
// ---cut---
type UserInputSanitizedString = string;

function sanitizeInput(str: string): UserInputSanitizedString {
  return sanitize(str);
}

// Create a sanitized input
let userInput = sanitizeInput(getInput());

// Can still be re-assigned with a string though
userInput = "new input";
```

Interfaces

An *interface declaration* is another way to name an object type:

```

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

function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });

```

Just like when we used a type alias above, the example works just as if we had used an anonymous object type. TypeScript is only concerned with the *structure* of the value we passed to `printCoord` - it only cares that it has the expected properties. Being concerned only with the structure and capabilities of types is why we call TypeScript a *structurally typed* type system.

Differences Between Type Aliases and Interfaces

Type aliases and interfaces are very similar, and in many cases you can choose between them freely. Almost all features of an `interface` are available in `type`, the key distinction is that a type cannot be re-opened to add new properties vs an interface which is always extendable.

Interface	Type
Extending an interface	Extending a type via intersections
<pre> interface Animal { name: string } interface Bear extends Animal { honey: boolean } const bear = getBear() bear.name bear.honey </pre>	<pre> type Animal = { name: string } type Bear = Animal & { honey: boolean } const bear = getBear(); bear.name; bear.honey; </pre>
Adding new fields to an existing interface	A type cannot be changed after being created

```

interface Window {           type Window = {
  title: string            title: string
}                           }

interface Window {           type Window = {
  ts: TypeScriptAPI        ts: TypeScriptAPI
}                           }

const src = 'const a = "Hello World"'; // Error: Duplicate identifier 'Window'.
window.ts.transpileModule(src, {});

```

You'll learn more about these concepts in later chapters, so don't worry if you don't understand all of these right away.

- Prior to TypeScript version 4.2, type alias names [may appear in error messages](#), sometimes in place of the equivalent anonymous type (which may or may not be desirable). Interfaces will always be named in error messages.
- Type aliases may not participate [in declaration merging, but interfaces can](#).
- Interfaces may only be used to [declare the shapes of objects, not rename primitives](#).
- Interface names will [always appear in their original form](#) in error messages, but *only* when they are used by name.

For the most part, you can choose based on personal preference, and TypeScript will tell you if it needs something to be the other kind of declaration. If you would like a heuristic, use `interface` until you need to use features from `type`.

Type Assertions

Sometimes you will have information about the type of a value that TypeScript can't know about.

For example, if you're using `document.getElementById`, TypeScript only knows that this will return *some* kind of `HTMLElement`, but you might know that your page will always have an `HTMLCanvasElement` with a given ID.

In this situation, you can use a *type assertion* to specify a more specific type:

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement;
```

Like a type annotation, type assertions are removed by the compiler and won't affect the runtime behavior of your code.

You can also use the angle-bracket syntax (except if the code is in a `.tsx` file), which is equivalent:

```
// @errors: 2352
const myCanvas = <HTMLCanvasElement>document.getElementById("main_canvas");
```

Reminder: Because type assertions are removed at compile-time, there is no runtime checking associated with a type assertion. There won't be an exception or `null` generated if the type assertion is wrong.

TypeScript only allows type assertions which convert to a *more specific* or *less specific* version of a type. This rule prevents "impossible" coercions like:

```
// @errors: 2352
const x = "hello" as number;
```

Sometimes this rule can be too conservative and will disallow more complex coercions that might be valid. If this happens, you can use two assertions, first to `any` (or `unknown`, which we'll introduce later), then to the desired type:

```
declare const expr: any;
type T = { a: 1; b: 2; c: 3 };
// ---cut---
const a = (expr as any) as T;
```

Literal Types

In addition to the general types `string` and `number`, we can refer to *specific* strings and numbers in type positions.

One way to think about this is to consider how JavaScript comes with different ways to declare a variable. Both `var` and `let` allow for changing what is held inside the variable, and `const` does not. This is reflected in how TypeScript creates types for literals.

```
let changingString = "Hello World";
changingString = "Olá Mundo";
// Because `changingString` can represent any possible string, that
// is how TypeScript describes it in the type system
changingString;
// ^?

const constantString = "Hello World";
// Because `constantString` can only represent 1 possible string, it
// has a literal type representation
constantString;
// ^?
```

By themselves, literal types aren't very valuable:

```
// @errors: 2322
let x: "hello" = "hello";
// OK
```

```
x = "hello";
// ...
x = "howdy";
```

It's not much use to have a variable that can only have one value!

But by *combining* literals into unions, you can express a much more useful concept - for example, functions that only accept a certain set of known values:

```
// @errors: 2345
function printText(s: string, alignment: "left" | "right" | "center") {
    // ...
}
printText("Hello, world", "left");
printText("G'day, mate", "centre");
```

Numeric literal types work the same way:

```
function compare(a: string, b: string): -1 | 0 | 1 {
    return a === b ? 0 : a > b ? 1 : -1;
}
```

Of course, you can combine these with non-literal types:

```
// @errors: 2345
interface Options {
    width: number;
}
function configure(x: Options | "auto") {
    // ...
}
configure({ width: 100 });
configure("auto");
configure("automatic");
```

There's one more kind of literal type: boolean literals. There are only two boolean literal types, and as you might guess, they are the types `true` and `false`. The type `boolean` itself is actually just an alias for the union `true | false`.

Literal Inference

When you initialize a variable with an object, TypeScript assumes that the properties of that object might change values later. For example, if you wrote code like this:

```
declare const someCondition: boolean;
// ---cut---
const obj = { counter: 0 };
if (someCondition) {
    obj.counter = 1;
}
```

TypeScript doesn't assume the assignment of `1` to a field which previously had `0` is an error. Another way of saying this is that `obj.counter` must have the type `number`, not `0`, because types are used to determine both *reading* and *writing* behavior.

The same applies to strings:

```
// @errors: 2345
declare function handleRequest(url: string, method: "GET" | "POST"): void;
// ---cut---
const req = { url: "https://example.com", method: "GET" };
handleRequest(req.url, req.method);
```

In the above example `req.method` is inferred to be `string`, not `"GET"`. Because code can be evaluated between the creation of `req` and the call of `handleRequest` which could assign a new string like `"GUESS"` to `req.method`, TypeScript considers this code to have an error.

There are two ways to work around this.

1. You can change the inference by adding a type assertion in either location:

```
declare function handleRequest(url: string, method: "GET" | "POST"): void;
// ---cut---
// Change 1:
const req = { url: "https://example.com", method: "GET" as "GET" };
// Change 2
handleRequest(req.url, req.method as "GET");
```

Change 1 means "I intend for `req.method` to always have the *literal type* `"GET"`", preventing the possible assignment of `"GUESS"` to that field after. Change 2 means "I know for other reasons that `req.method` has the value `"GET"`".

2. You can use `as const` to convert the entire object to be type literals:

```
declare function handleRequest(url: string, method: "GET" | "POST"): void;
// ---cut---
const req = { url: "https://example.com", method: "GET" } as const;
handleRequest(req.url, req.method);
```

The `as const` suffix acts like `const` but for the type system, ensuring that all properties are assigned the literal type instead of a more general version like `string` or `number`.

null and undefined

JavaScript has two primitive values used to signal absent or uninitialized value: `null` and `undefined`.

TypeScript has two corresponding types by the same names. How these types behave depends on whether you have the `strictNullChecks` option on.

`strictNullChecks` off

With `strictNullChecks` off, values that might be `null` or `undefined` can still be accessed normally, and the values `null` and `undefined` can be assigned to a property of any type. This is similar to how languages without null checks (e.g. C#, Java) behave. The lack of checking for these values tends to be a major source of bugs; we always recommend people turn `strictNullChecks` on if it's practical to do so in their codebase.

strictNullChecks on

With `strictNullChecks on`, when a value is `null` or `undefined`, you will need to test for those values before using methods or properties on that value. Just like checking for `undefined` before using an optional property, we can use *narrowing* to check for values that might be `null`:

```
function doSomething(x: string | null) {
  if (x === null) {
    // do nothing
  } else {
    console.log("Hello, " + x.toUpperCase());
  }
}
```

Non-null Assertion Operator (Postfix `!`)

TypeScript also has a special syntax for removing `null` and `undefined` from a type without doing any explicit checking. Writing `!` after any expression is effectively a type assertion that the value isn't `null` or `undefined`:

```
function liveDangerously(x?: number | null) {
  // No error
  console.log(x!.toFixed());
}
```

Just like other type assertions, this doesn't change the runtime behavior of your code, so it's important to only use `!` when you know that the value *can't* be `null` or `undefined`.

Enums

Enums are a feature added to JavaScript by TypeScript which allows for describing a value which could be one of a set of possible named constants. Unlike most TypeScript features, this is *not* a type-level addition to JavaScript but something added to the language and runtime. Because of this, it's a feature which you should know exists, but maybe hold off on using unless you are sure. You can read more about enums in the [Enum reference page](#).

Less Common Primitives

It's worth mentioning the rest of the primitives in JavaScript which are represented in the type system. Though we will not go into depth here.

`bigint`

From ES2020 onwards, there is a primitive in JavaScript used for very large integers, `BigInt`:

```
// @target: es2020
// Creating a bigint via the BigInt function
const oneHundred: bigint = BigInt(100);
```

```
// Creating a BigInt via the literal syntax
const anotherHundred: bigint = 100n;
```

You can learn more about BigInt in [the TypeScript 3.2 release notes](#).

symbol

There is a primitive in JavaScript used to create a globally unique reference via the function `Symbol()`:

```
// @errors: 2367
const firstName = Symbol("name");
const secondName = Symbol("name");

if (firstName === secondName) {
    // Can't ever happen
}
```

You can learn more about them in [Symbols reference page](#).

[Go to TOC](#)

JavaScript has a long history of different ways to handle modularizing code. TypeScript having been around since 2012, has implemented support for a lot of these formats, but over time the community and the JavaScript specification has converged on a format called ES Modules (or ES6 modules). You might know it as the `import / export` syntax.

ES Modules was added to the JavaScript spec in 2015, and by 2020 had broad support in most web browsers and JavaScript runtimes.

For focus, the handbook will cover both ES Modules and its popular pre-cursor CommonJS `module.exports =` syntax, and you can find information about the other module patterns in the reference section under [Modules](#).

How JavaScript Modules are Defined

In TypeScript, just as in ECMAScript 2015, any file containing a top-level `import` or `export` is considered a module.

Conversely, a file without any top-level import or export declarations is treated as a script whose contents are available in the global scope (and therefore to modules as well).

Modules are executed within their own scope, not in the global scope. This means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the export forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the import forms.

Non-modules

Before we start, it's important to understand what TypeScript considers a module. The JavaScript specification declares that any JavaScript files without an `export` or top-level `await` should be considered a script and not a module.

Inside a script file variables and types are declared to be in the shared global scope, and it's assumed that you'll either use the `outFile` compiler option to join multiple input files into one output file, or use multiple `<script>` tags in your HTML to load these files (in the correct order!).

If you have a file that doesn't currently have any `imports` or `exports`, but you want to be treated as a module, add the line:

```
export {};
```

which will change the file to be a module exporting nothing. This syntax works regardless of your module target.

Modules in TypeScript

Additional Reading:

[Impatient JS \(Modules\)](#)
[MDN: JavaScript Modules](#)

There are three main things to consider when writing module-based code in TypeScript:

- **Syntax:** What syntax do I want to use to import and export things?
- **Module Resolution:** What is the relationship between module names (or paths) and files on disk?
- **Module Output Target:** What should my emitted JavaScript module look like?

ES Module Syntax

A file can declare a main export via `export default`:

```
// @filename: hello.ts
export default function helloWorld() {
  console.log("Hello, world!");
}
```

This is then imported via:

```
// @filename: hello.ts
export default function helloWorld() {
  console.log("Hello, world!");
}
// @filename: index.ts
// ---cut---
import helloWorld from "./hello.js";
helloWorld();
```

In addition to the default export, you can have more than one export of variables and functions via the `export` by omitting `default`:

```
// @filename: maths.ts
export var pi = 3.14;
export let squareTwo = 1.41;
export const phi = 1.61;

export class RandomNumberGenerator {}

export function absolute(num: number) {
  if (num < 0) return num * -1;
  return num;
}
```

These can be used in another file via the `import` syntax:

```
// @filename: maths.ts
export var pi = 3.14;
export let squareTwo = 1.41;
export const phi = 1.61;
export class RandomNumberGenerator {}
export function absolute(num: number) {
  if (num < 0) return num * -1;
  return num;
}
// @filename: app.ts
// ---cut---
import { pi, phi, absolute } from "./maths.js";

console.log(pi);
const absPhi = absolute(phi);
// ^?
```

Additional Import Syntax

An import can be renamed using a format like `import {old as new}` :

```
// @filename: maths.ts
export var pi = 3.14;
// @filename: app.ts
// ---cut---
import { pi as π } from "./maths.js";

console.log(π);
// ^?
```

You can mix and match the above syntax into a single `import` :

```
// @filename: maths.ts
export const pi = 3.14;
export default class RandomNumberGenerator {}

// @filename: app.ts
import RandomNumberGenerator, { pi as π } from "./maths.js";

RandomNumberGenerator;
// ^?

console.log(π);
// ^?
```

You can take all of the exported objects and put them into a single namespace using `* as name` :

```
// @filename: maths.ts
export var pi = 3.14;
export let squareTwo = 1.41;
export const phi = 1.61;

export function absolute(num: number) {
  if (num < 0) return num * -1;
  return num;
}
// ---cut---
// @filename: app.ts
import * as math from "./maths.js";
```

```
// console.log(math.pi);
const positivePhi = math.absolute(math.phi);
// ^?
```

You can import a file and *not* include any variables into your current module via `import "./file"`:

```
// @filename: maths.ts
export var pi = 3.14;
// ---cut---
// @filename: app.ts
import "./maths.js";
console.log("3.14");
```

In this case, the `import` does nothing. However, all of the code in `maths.ts` was evaluated, which could trigger side-effects which affect other objects.

TypeScript Specific ES Module Syntax

Types can be exported and imported using the same syntax as JavaScript values:

```
// @filename: animal.ts
export type Cat = { breed: string; yearOfBirth: number };

export interface Dog {
  breeds: string[];
  yearOfBirth: number;
}

// @filename: app.ts
import { Cat, Dog } from "./animal.js";
type Animals = Cat | Dog;
```

TypeScript has extended the `import` syntax with two concepts for declaring an import of a type:

```
import type
```

Which is an import statement which can *only* import types:

```
// @filename: animal.ts
export type Cat = { breed: string; yearOfBirth: number };
export type Dog = { breeds: string[]; yearOfBirth: number };
export const createCatName = () => "fluffy";

// @filename: valid.ts
import type { Cat, Dog } from "./animal.js";
export type Animals = Cat | Dog;

// @filename: app.ts
// @errors: 1361
import type { createCatName } from "./animal.js";
const name = createCatName();
```

Inline `type` imports

TypeScript 4.5 also allows for individual imports to be prefixed with `type` to indicate that the imported reference is a type:

```
// @filename: animal.ts
export type Cat = { breed: string; yearOfBirth: number };
export type Dog = { breeds: string[]; yearOfBirth: number };
export const createCatName = () => "fluffy";
// ---cut---
// @filename: app.ts
import { createCatName, type Cat, type Dog } from "./animal.js";

export type Animals = Cat | Dog;
const name = createCatName();
```

Together these allow a non-TypeScript transpiler like Babel, swc or esbuild to know what imports can be safely removed.

ES Module Syntax with CommonJS Behavior

TypeScript has ES Module syntax which *directly* correlates to a CommonJS and AMD `require`. Imports using ES Module are *for most cases* the same as the `require` from those environments, but this syntax ensures you have a 1 to 1 match in your TypeScript file with the CommonJS output:

```
/// <reference types="node" />
// @module: commonjs
// ---cut---
import fs = require("fs");
const code = fs.readFileSync("hello.ts", "utf8");
```

You can learn more about this syntax in the [modules reference page](#).

CommonJS Syntax

CommonJS is the format which most modules on npm are delivered in. Even if you are writing using the ES Modules syntax above, having a brief understanding of how CommonJS syntax works will help you debug easier.

Exporting

Identifiers are exported via setting the `exports` property on a global called `module`.

```
/// <reference types="node" />
// ---cut---
function absolute(num: number) {
  if (num < 0) return num * -1;
  return num;
}

module.exports = {
  pi: 3.14,
  squareTwo: 1.41,
  phi: 1.61,
  absolute,
};
```

Then these files can be imported via a `require` statement:

```
// @module: commonjs
// @filename: maths.ts
/// <reference types="node" />
function absolute(num: number) {
  if (num < 0) return num * -1;
  return num;
}

module.exports = {
  pi: 3.14,
  squareTwo: 1.41,
  phi: 1.61,
  absolute,
};
// @filename: index.ts
// ---cut---
const maths = require("maths");
maths.pi;
// ^?
```

Or you can simplify a bit using the destructuring feature in JavaScript:

```
// @module: commonjs
// @filename: maths.ts
/// <reference types="node" />
function absolute(num: number) {
  if (num < 0) return num * -1;
  return num;
}

module.exports = {
  pi: 3.14,
  squareTwo: 1.41,
  phi: 1.61,
  absolute,
};
// @filename: index.ts
// ---cut---
const { squareTwo } = require("maths");
squareTwo;
// ^?
```

CommonJS and ES Modules interop

There is a mis-match in features between CommonJS and ES Modules regarding the distinction between a default import and a module namespace object import. TypeScript has a compiler flag to reduce the friction between the two different sets of constraints with `esModuleInterop`.

TypeScript's Module Resolution Options

Module resolution is the process of taking a string from the `import` or `require` statement, and determining what file that string refers to.

TypeScript includes two resolution strategies: Classic and Node. Classic, the default when the compiler option `module` is not `commonjs`, is included for backwards compatibility. The Node strategy replicates how Node.js works in CommonJS mode, with additional checks for `.ts` and `.d.ts`.

There are many TSConfig flags which influence the module strategy within TypeScript: `moduleResolution`, `baseUrl`, `paths`, `rootDirs`.

For the full details on how these strategies work, you can consult the [Module Resolution](#).

TypeScript's Module Output Options

There are two options which affect the emitted JavaScript output:

- `target` which determines which JS features are downleveled (converted to run in older JavaScript runtimes) and which are left intact
- `module` which determines what code is used for modules to interact with each other

Which `target` you use is determined by the features available in the JavaScript runtime you expect to run the TypeScript code in. That could be: the oldest web browser you support, the lowest version of Node.js you expect to run on or could come from unique constraints from your runtime - like Electron for example.

All communication between modules happens via a module loader, the compiler option `module` determines which one is used. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it.

For example, here is a TypeScript file using ES Modules syntax, showcasing a few different options for `module`:

```
// @filename: constants.ts
export const valueOfPi = 3.142;
// @filename: index.ts
// ---cut---
import { valueOfPi } from "./constants.js";

export const twoPi = valueOfPi * 2;
```

ES2020

```
// @showEmit
// @module: es2020
// @noErrors
import { valueOfPi } from "./constants.js";

export const twoPi = valueOfPi * 2;
```

CommonJS

```
// @showEmit
// @module: commonjs
// @noErrors
import { valueOfPi } from "./constants.js";

export const twoPi = valueOfPi * 2;
```

UMD

```
// @showEmit
// @module: umd
// @noErrors
import { valueOfPi } from "./constants.js";
export const twoPi = valueOfPi * 2;
```

Note that ES2020 is effectively the same as the original `index.ts`.

You can see all of the available options and what their emitted JavaScript code looks like in the [TSConfig Reference for `module`](#).

TypeScript namespaces

TypeScript has its own module format called `namespaces` which pre-dates the ES Modules standard. This syntax has a lot of useful features for creating complex definition files, and still sees active use [in DefinitelyTyped](#). While not deprecated, the majority of the features in namespaces exist in ES Modules and we recommend you use that to align with JavaScript's direction. You can learn more about namespaces in the [namespaces reference page](#).

[Go to TOC](#)

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 named `string` of type `any`"!

Of course, we can use a type alias to name a function type:

```
type GreetFunction = (a: string) => void;
function greeter(fn: GreetFunction) {
  // ...
}
```

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));
}
```

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 SomeObject = any;
// ---cut---
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 {
  new (s: string): Date;
  (n?: number): number;
}
```

Generic Functions

It's common to write a function 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 for a moment a function that returns the first element of an array:

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

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

In TypeScript, *generics* are used when we want to describe a correspondence between two values. We do this by declaring a *type parameter* in the function signature:

```
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, a more specific type comes out:

```
// declare function firstElement<Type>(arr: Type[]): Type | undefined;
// ---cut---
// s is of type 'string'
const s = firstElement(["a", "b", "c"]);
// n is of type 'number'
const n = firstElement([1, 2, 3]);
// u is of type undefined
const u = firstElement([]);
```

Inference

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

We can use multiple type parameters as well. For example, a standalone version of `map` would look like this:

```
// prettier-ignore
function map<Input, Output>(arr: Input[], func: (arg: Input) => Output): Output[] {
  return arr.map(func);
}

// Parameter 'n' is of type 'string'
// 'parsed' is of type 'number[]'
const parsed = map(["1", "2", "3"], (n) => parseInt(n));
```

Note that in this example, TypeScript could infer both the type of the `Input` type parameter (from the given `string` array), as well as the `Output` type parameter based on the return value of the function expression (`number`).

Constraints

We've written some generic functions that can work on *any* kind of value. Sometimes we want to relate two values, but can only operate on a certain subset of values. In this case, we can use a *constraint* to limit the kinds of types that a type parameter can accept.

Let's write a function that returns the longer of two values. To do this, we need a `length` property that's a number. We *constrain* the type parameter to that type by writing an `extends` clause:

```
// @errors: 2345 2322
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);
```

There are a few interesting things to note in this example. We allowed TypeScript to *infer* the return type of `longest`. Return type inference also works on generic functions.

Because we constrained `Type` to `{ length: number }`, we were allowed to access the `.length` property of the `a` and `b` parameters. Without the type constraint, we wouldn't be able to access those properties because the values might have been some other type without a `length` property.

The types of `longerArray` and `longerString` were inferred based on the arguments. Remember, generics are all about relating two or more values with the same type!

Finally, just as we'd like, the call to `longest(10, 100)` is rejected because the `number` type doesn't have a `.length` property.

Working with Constrained Values

Here's a common error when working with generic constraints:

```
// @errors: 2322
function minimumLength<Type extends { length: number }>(
  obj: Type,
  minimum: number
): Type {
  if (obj.length >= minimum) {
    return obj;
  } else {
    return { length: minimum };
  }
}
```

It might look like this function is OK - `Type` is constrained to `{ length: number }`, and the function either returns `Type` or a value matching that constraint. The problem is that the function promises to return the *same* kind of object as was passed in, not just *some* object matching the constraint. If this code were legal, you could write code that definitely wouldn't work:

```
declare function minimumLength<Type extends { length: number }>(
  obj: Type,
  minimum: number
): Type;
// ---cut---
// 'arr' gets value { length: 6 }
const arr = minimumLength([1, 2, 3], 6);
// and crashes here because arrays have
// a 'slice' method, but not the returned object!
console.log(arr.slice(0));
```

Specifying Type Arguments

TypeScript can usually infer the intended type arguments in a generic call, but not always. For example, let's say you wrote a function to combine two arrays:

```
function combine<Type>(arr1: Type[], arr2: Type[]): Type[] {
  return arr1.concat(arr2);
}
```

Normally it would be an error to call this function with mismatched arrays:

```
// @errors: 2322
declare function combine<Type>(arr1: Type[], arr2: Type[]): Type[];
// ---cut---
const arr = combine([1, 2, 3], ["hello"]);
```

If you intended to do this, however, you could manually specify `Type`:

```
declare function combine<Type>(arr1: Type[], arr2: Type[]): Type[];
// ---cut---
const arr = combine<string | number>([1, 2, 3], ["hello"]);
```

Guidelines for Writing Good Generic Functions

Writing generic functions is fun, and it can be easy to get carried away with type parameters. Having too many type parameters or using constraints where they aren't needed can make inference less successful, frustrating callers of your function.

Push Type Parameters Down

Here are two ways of writing a function that appear similar:

```
function firstElement1<Type>(arr: Type[]) {
  return arr[0];
}

function firstElement2<Type extends any[]>(arr: Type) {
  return arr[0];
}

// a: number (good)
const a = firstElement1([1, 2, 3]);
// b: any (bad)
const b = firstElement2([1, 2, 3]);
```

These might seem identical at first glance, but `firstElement1` is a much better way to write this function. Its inferred return type is `Type`, but `firstElement2`'s inferred return type is `any` because TypeScript has to resolve the `arr[0]` expression using the constraint type, rather than "waiting" to resolve the element during a call.

Rule: When possible, use the type parameter itself rather than constraining it

Use Fewer Type Parameters

Here's another pair of similar functions:

```
function filter1<Type>(arr: Type[], func: (arg: Type) => boolean): Type[] {
  return arr.filter(func);
}

function filter2<Type, Func extends (arg: Type) => boolean>(
```

```
arr: Type[],  
func: Func  
): Type[] {  
  return arr.filter(func);  
}
```

We've created a type parameter `Func` that *doesn't relate two values*. That's always a red flag, because it means callers wanting to specify type arguments have to manually specify an extra type argument for no reason. `Func` doesn't do anything but make the function harder to read and reason about!

Rule: Always use as few type parameters as possible

Type Parameters Should Appear Twice

Sometimes we forget that a function might not need to be generic:

```
function greet<Str extends string>(s: Str) {  
  console.log("Hello, " + s);  
}  
  
greet("world");
```

We could just as easily have written a simpler version:

```
function greet(s: string) {  
  console.log("Hello, " + s);  
}
```

Remember, type parameters are for *relating the types of multiple values*. If a type parameter is only used once in the function signature, it's not relating anything.

Rule: If a type parameter only appears in one location, strongly reconsider if you actually need it

Optional Parameters

Functions in JavaScript often take a variable number of arguments. For example, the `toFixed` method of `number` takes an optional digit count:

```
function f(n: number) {  
  console.log(n.toFixed()); // 0 arguments  
  console.log(n.toFixed(3)); // 1 argument  
}
```

We can model this in TypeScript by marking the parameter as *optional* with `?`:

```
function f(x?: number) {
  // ...
}
f(); // OK
f(10); // OK
```

Although the parameter is specified as type `number`, the `x` parameter will actually have the type `number | undefined` because unspecified parameters in JavaScript get the value `undefined`.

You can also provide a parameter *default*:

```
function f(x = 10) {
  // ...
}
```

Now in the body of `f`, `x` will have type `number` because any `undefined` argument will be replaced with `10`. Note that when a parameter is optional, callers can always pass `undefined`, as this simply simulates a "missing" argument:

```
declare function f(x?: number): void;
// cut
// All OK
f();
f(10);
f(undefined);
```

Optional Parameters in Callbacks

Once you've learned about optional parameters and function type expressions, it's very easy to make the following mistakes when writing functions that invoke callbacks:

```
function myForEach(arr: any[], callback: (arg: any, index?: number) => void) {
  for (let i = 0; i < arr.length; i++) {
    callback(arr[i], i);
  }
}
```

What people usually intend when writing `index?` as an optional parameter is that they want both of these calls to be legal:

```
// @errors: 2532
declare function myForEach(
  arr: any[],
  callback: (arg: any, index?: number) => void
): void;
// ---cut---
myForEach([1, 2, 3], (a) => console.log(a));
myForEach([1, 2, 3], (a, i) => console.log(a, i));
```

What this *actually* means is that `callback` *might* get invoked with one argument. In other words, the function definition says that the implementation might look like this:

```
// @errors: 2532
function myForEach(arr: any[], callback: (arg: any, index?: number) => void) {
  for (let i = 0; i < arr.length; i++) {
    // I don't feel like providing the index today
    callback(arr[i]);
  }
}
```

In turn, TypeScript will enforce this meaning and issue errors that aren't really possible:

```
// @errors: 2532
declare function myForEach(
  arr: any[],
  callback: (arg: any, index?: number) => void
): void;
// ---cut---
myForEach([1, 2, 3], (a, i) => {
  console.log(i.toFixed());
});
```

In JavaScript, if you call a function with more arguments than there are parameters, the extra arguments are simply ignored. TypeScript behaves the same way. Functions with fewer parameters (of the same types) can always take the place of functions with more parameters.

When writing a function type for a callback, *never* write an optional parameter unless you intend to *call* the function without passing that argument

Function Overloads

Some JavaScript functions can be called in a variety of argument counts and types. For example, you might write a function to produce a `Date` that takes either a timestamp (one argument) or a month/day/year specification (three arguments).

In TypeScript, we can specify a function that can be called in different ways by writing *overload signatures*. To do this, write some number of function signatures (usually two or more), followed by the body of the function:

```
// @errors: 2575
function makeDate(timestamp: number): Date;
function makeDate(m: number, d: number, y: number): Date;
function makeDate(mOrTimestamp: number, d?: number, y?: number): Date {
  if (d !== undefined && y !== undefined) {
    return new Date(y, mOrTimestamp, d);
  } else {
    return new Date(mOrTimestamp);
  }
}
const d1 = makeDate(12345678);
const d2 = makeDate(5, 5, 5);
const d3 = makeDate(1, 3);
```

In this example, we wrote two overloads: one accepting one argument, and another accepting three arguments. These first two signatures are called the *overload signatures*.

Then, we wrote a function implementation with a compatible signature. Functions have an *implementation* signature, but this signature can't be called directly. Even though we wrote a function with two optional parameters after the required one, it can't be called with two parameters!

Overload Signatures and the Implementation Signature

This is a common source of confusion. Often people will write code like this and not understand why there is an error:

```
// @errors: 2554
function fn(x: string): void;
function fn(): {
    // ...
}
// Expected to be able to call with zero arguments
fn();
```

Again, the signature used to write the function body can't be "seen" from the outside.

The signature of the *implementation* is not visible from the outside. When writing an overloaded function, you should always have two or more signatures above the implementation of the function.

The implementation signature must also be *compatible* with the overload signatures. For example, these functions have errors because the implementation signature doesn't match the overloads in a correct way:

```
// @errors: 2394
function fn(x: boolean): void;
// Argument type isn't right
function fn(x: string): void;
function fn(x: boolean) {}
```

```
// @errors: 2394
function fn(x: string): string;
// Return type isn't right
function fn(x: number): boolean;
function fn(x: string | number) {
    return "oops";
}
```

Writing Good Overloads

Like generics, there are a few guidelines you should follow when using function overloads. Following these principles will make your function easier to call, easier to understand, and easier to implement.

Let's consider a function that returns the length of a string or an array:

```
function len(s: string): number;
function len(arr: any[]): number;
function len(x: any) {
    return x.length;
}
```

This function is fine; we can invoke it with strings or arrays. However, we can't invoke it with a value that might be a string *or* an array, because TypeScript can only resolve a function call to a single overload:

```
// @errors: 2769
declare function len(s: string): number;
declare function len(arr: any[]): number;
// ---cut---
len(""); // OK
len([0]); // OK
len(Math.random() > 0.5 ? "hello" : [0]);
```

Because both overloads have the same argument count and same return type, we can instead write a non-overloaded version of the function:

```
function len(x: any[] | string) {
    return x.length;
}
```

This is much better! Callers can invoke this with either sort of value, and as an added bonus, we don't have to figure out a correct implementation signature.

Always prefer parameters with union types instead of overloads when possible

Declaring `this` in a Function

TypeScript will infer what the `this` should be in a function via code flow analysis, for example in the following:

```
const user = {
    id: 123,
    admin: false,
    becomeAdmin: function () {
        this.admin = true;
    },
};
```

TypeScript understands that the function `user.becomeAdmin` has a corresponding `this` which is the outer object `user`. `this`, *heh*, can be enough for a lot of cases, but there are a lot of cases where you need more control over what object `this` represents. The JavaScript specification states that you cannot have a parameter called `this`, and so TypeScript uses that syntax space to let you declare the type for `this` in the function body.

```

interface User {
  id: number;
  admin: boolean;
}
declare const getDB: () => DB;
// ---cut---
interface DB {
  filterUsers(filter: (this: User) => boolean): User[];
}

const db = getDB();
const admins = db.filterUsers(function (this: User) {
  return this.admin;
});

```

This pattern is common with callback-style APIs, where another object typically controls when your function is called. Note that you need to use `function` and not arrow functions to get this behavior:

```

// @errors: 7041 7017
interface User {
  id: number;
  isAdmin: boolean;
}
declare const getDB: () => DB;
// ---cut---
interface DB {
  filterUsers(filter: (this: User) => boolean): User[];
}

const db = getDB();
const admins = db.filterUsers(() => this.admin);

```

Other Types to Know About

There are some additional types you'll want to recognize that appear often when working with function types. Like all types, you can use them everywhere, but these are especially relevant in the context of functions.

void

`void` represents the return value of functions which don't return a value. It's the inferred type any time a function doesn't have any `return` statements, or doesn't return any explicit value from those return statements:

```

// The inferred return type is void
function noop() {
  return;
}

```

In JavaScript, a function that doesn't return any value will implicitly return the value `undefined`. However, `void` and `undefined` are not the same thing in TypeScript. There are further details at the end of this chapter.

```
void is not the same as undefined.
```

object

The special type `object` refers to any value that isn't a primitive (`string`, `number`, `bignum`, `boolean`, `symbol`, `null`, or `undefined`). This is different from the *empty object type* `{ }`, and also different from the global type `Object`. It's very likely you will never use `Object`.

```
object is not Object. Always use object!
```

Note that in JavaScript, function values are objects: They have properties, have `Object.prototype` in their prototype chain, are `instanceof Object`, you can call `Object.keys` on them, and so on. For this reason, function types are considered to be `object`s in TypeScript.

unknown

The `unknown` type represents *any* value. This is similar to the `any` type, but is safer because it's not legal to do anything with an `unknown` value:

```
// @errors: 2571
function f1(a: any) {
  a.b(); // OK
}
function f2(a: unknown) {
  a.b();
}
```

This is useful when describing function types because you can describe functions that accept any value without having `any` values in your function body.

Conversely, you can describe a function that returns a value of unknown type:

```
declare const someRandomString: string;
// ---cut---
function safeParse(s: string): unknown {
  return JSON.parse(s);
}

// Need to be careful with 'obj'!
const obj = safeParse(someRandomString);
```

never

Some functions *never* return a value:

```
function fail(msg: string): never {
  throw new Error(msg);
}
```

The `never` type represents values which are *never* observed. In a return type, this means that the function throws an exception or terminates execution of the program.

`never` also appears when TypeScript determines there's nothing left in a union.

```
function fn(x: string | number) {
  if (typeof x === "string") {
    // do something
  } else if (typeof x === "number") {
    // do something else
  } else {
    x; // has type 'never'!
}
```

Function

The global type `Function` describes properties like `bind`, `call`, `apply`, and others present on all function values in JavaScript. It also has the special property that values of type `Function` can always be called; these calls return `any`:

```
function doSomething(f: Function) {
  return f(1, 2, 3);
}
```

This is an *untyped function call* and is generally best avoided because of the unsafe `any` return type.

If you need to accept an arbitrary function but don't intend to call it, the type `() => void` is generally safer.

Rest Parameters and Arguments

Background Reading:

[Rest Parameters](#)

[Spread Syntax](#)

Rest Parameters

In addition to using optional parameters or overloads to make functions that can accept a variety of fixed argument counts, we can also define functions that take an *unbounded* number of arguments using *rest parameters*.

A rest parameter appears after all other parameters, and uses the `...` syntax:

```
function multiply(n: number, ...m: number[]) {
  return m.map((x) => n * x);
}
// 'a' gets value [10, 20, 30, 40]
const a = multiply(10, 1, 2, 3, 4);
```

In TypeScript, the type annotation on these parameters is implicitly `any[]` instead of `any`, and any type annotation given must be of the form `Array<T>` or `T[]`, or a tuple type (which we'll learn about later).

Rest Arguments

Conversely, we can *provide* a variable number of arguments from an array using the spread syntax. For example, the `push` method of arrays takes any number of arguments:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
arr1.push(...arr2);
```

Note that in general, TypeScript does not assume that arrays are immutable. This can lead to some surprising behavior:

```
// @errors: 2556
// Inferred type is number[] -- "an array with zero or more numbers",
// not specifically two numbers
const args = [8, 5];
const angle = Math.atan2(...args);
```

The best fix for this situation depends a bit on your code, but in general a `const` context is the most straightforward solution:

```
// Inferred as 2-length tuple
const args = [8, 5] as const;
// OK
const angle = Math.atan2(...args);
```

Using rest arguments may require turning on `downlevelIteration` when targeting older runtimes.

Parameter Destructuring

Background Reading:

[Destructuring Assignment](#)

You can use parameter destructuring to conveniently unpack objects provided as an argument into one or more local variables in the function body. In JavaScript, it looks like this:

```
function sum({ a, b, c }) {
  console.log(a + b + c);
}
sum({ a: 10, b: 3, c: 9 });
```

The type annotation for the object goes after the destructuring syntax:

```
function sum({ a, b, c }: { a: number; b: number; c: number }) {
  console.log(a + b + c);
}
```

This can look a bit verbose, but you can use a named type here as well:

```
// Same as prior example
type ABC = { a: number; b: number; c: number };
function sum({ a, b, c }: ABC) {
  console.log(a + b + c);
}
```

Assignability of Functions

Return type `void`

The `void` return type for functions can produce some unusual, but expected behavior.

Contextual typing with a return type of `void` does **not** force functions to **not** return something. Another way to say this is a contextual function type with a `void` return type (`type vf = () => void`), when implemented, can return *any* other value, but it will be ignored.

Thus, the following implementations of the type `() => void` are valid:

```
type voidFunc = () => void;
const f1: voidFunc = () => {
  return true;
};
const f2: voidFunc = () => true;
const f3: voidFunc = function () {
  return true;
};
```

And when the return value of one of these functions is assigned to another variable, it will retain the type of `void`:

```
type voidFunc = () => void;
const f1: voidFunc = () => {
  return true;
};
const f2: voidFunc = () => true;
const f3: voidFunc = function () {
  return true;
};
// ---cut---
const v1 = f1();
const v2 = f2();
const v3 = f3();
```

This behavior exists so that the following code is valid even though `Array.prototype.push` returns a number and the `Array.prototype.forEach` method expects a function with a return type of `void`.

```
const src = [1, 2, 3];
const dst = [0];

src.forEach((el) => dst.push(el));
```

There is one other special case to be aware of, when a literal function definition has a `void` return type, that function must **not** return anything.

```
function f2(): void {
  // @ts-expect-error
  return true;
}

const f3 = function (): void {
  // @ts-expect-error
  return true;
};
```

For more on `void` please refer to these other documentation entries:

- [v1 handbook](#)
- [v2 handbook](#)
- [FAQ - "Why are functions returning non-void assignable to function returning void?"](#)

[Go to TOC](#)

Imagine we have a function called `padLeft`.

```
function padLeft(padding: number | string, input: string): string {
    throw new Error("Not implemented yet!");
}
```

If `padding` is a `number`, it will treat that as the number of spaces we want to prepend to `input`. If `padding` is a `string`, it should just prepend `padding` to `input`. Let's try to implement the logic for when `padLeft` is passed a `number` for `padding`.

```
// @errors: 2345
function padLeft(padding: number | string, input: string) {
    return " ".repeat(padding) + input;
}
```

Uh-oh, we're getting an error on `padding`. TypeScript is warning us that adding a `number | string` to a `number` might not give us what we want, and it's right. In other words, we haven't explicitly checked if `padding` is a `number` first, nor are we handling the case where it's a `string`, so let's do exactly that.

```
function padLeft(padding: number | string, input: string) {
    if (typeof padding === "number") {
        return " ".repeat(padding) + input;
    }
    return padding + input;
}
```

If this mostly looks like uninteresting JavaScript code, that's sort of the point. Apart from the annotations we put in place, this TypeScript code looks like JavaScript. The idea is that TypeScript's type system aims to make it as easy as possible to write typical JavaScript code without bending over backwards to get type safety.

While it might not look like much, there's actually a lot going under the covers here. Much like how TypeScript analyzes runtime values using static types, it overlays type analysis on JavaScript's runtime control flow constructs like `if/else`, conditional ternaries, loops, truthiness checks, etc., which can all affect those types.

Within our `if` check, TypeScript sees `typeof padding === "number"` and understands that as a special form of code called a *type guard*. TypeScript follows possible paths of execution that our programs can take to analyze the most specific possible type of a value at a given position. It looks at these special checks (called *type guards*) and assignments, and the process of refining types to more specific types than declared is called *narrowing*. In many editors we can observe these types as they change, and we'll even do so in our examples.

```
function padLeft(padding: number | string, input: string) {
    if (typeof padding === "number") {
        return " ".repeat(padding) + input;
        //           ^
    }
    return padding + input;
    //           ^
}
```

There are a couple of different constructs TypeScript understands for narrowing.

typeof type guards

As we've seen, JavaScript supports a `typeof` operator which can give very basic information about the type of values we have at runtime. TypeScript expects this to return a certain set of strings:

- "string"
- "number"
- "bigint"
- "boolean"
- "symbol"
- "undefined"
- "object"
- "function"

Like we saw with `padLeft`, this operator comes up pretty often in a number of JavaScript libraries, and TypeScript can understand it to narrow types in different branches.

In TypeScript, checking against the value returned by `typeof` is a type guard. Because TypeScript encodes how `typeof` operates on different values, it knows about some of its quirks in JavaScript. For example, notice that in the list above, `typeof` doesn't return the string `null`. Check out the following example:

```
// @errors: 2531
function printAll(strs: string | string[] | null) {
  if (typeof strs === "object") {
    for (const s of strs) {
      console.log(s);
    }
  } else if (typeof strs === "string") {
    console.log(strs);
  } else {
    // do nothing
  }
}
```

In the `printAll` function, we try to check if `strs` is an object to see if it's an array type (now might be a good time to reinforce that arrays are object types in JavaScript). But it turns out that in JavaScript, `typeof null` is actually `"object"`! This is one of those unfortunate accidents of history.

Users with enough experience might not be surprised, but not everyone has run into this in JavaScript; luckily, TypeScript lets us know that `strs` was only narrowed down to `string[] | null` instead of just `string[]`.

This might be a good segue into what we'll call "truthiness" checking.

Truthiness narrowing

Truthiness might not be a word you'll find in the dictionary, but it's very much something you'll hear about in JavaScript.

In JavaScript, we can use any expression in conditionals, `&&`s, `||`s, `if` statements, Boolean negations (`!`), and more. As an example, `if` statements don't expect their condition to always have the type `boolean`.

```
function getUsersOnlineMessage(numUsersOnline: number) {
  if (numUsersOnline) {
    return `There are ${numUsersOnline} online now!`;
  }
  return "Nobody's here. :(";
```

In JavaScript, constructs like `if` first "coerce" their conditions to `boolean`s to make sense of them, and then choose their branches depending on whether the result is `true` or `false`. Values like

- `0`
- `Nan`
- `""` (the empty string)
- `0n` (the `bigint` version of zero)
- `null`
- `undefined`

all coerce to `false`, and other values get coerced `true`. You can always coerce values to `boolean`s by running them through the `Boolean` function, or by using the shorter double-Boolean negation. (The latter has the advantage that TypeScript infers a narrow literal boolean type `true`, while inferring the first as type `boolean`.)

```
// both of these result in 'true'
Boolean("hello"); // type: boolean, value: true
!!"world"; // type: true, value: true
```

It's fairly popular to leverage this behavior, especially for guarding against values like `null` or `undefined`. As an example, let's try using it for our `printAll` function.

```
function printAll(strs: string | string[] | null) {
  if (strs && typeof strs === "object") {
    for (const s of strs) {
      console.log(s);
    }
  } else if (typeof strs === "string") {
    console.log(strs);
  }
}
```

You'll notice that we've gotten rid of the error above by checking if `strs` is truthy. This at least prevents us from dreaded errors when we run our code like:

TypeError: null is not iterable

Keep in mind though that truthiness checking on primitives can often be error prone. As an example, consider a different attempt at writing `printAll`

```
function printAll(strs: string | string[] | null) {
  // !!!!!!!!
  // DON'T DO THIS!
  // KEEP READING
  // !!!!!!!!
  if (strs) {
    if (typeof strs === "object") {
      for (const s of strs) {
        console.log(s);
      }
    } else if (typeof strs === "string") {
      console.log(strs);
    }
  }
}
```

We wrapped the entire body of the function in a truthy check, but this has a subtle downside: we may no longer be handling the empty string case correctly.

TypeScript doesn't hurt us here at all, but this is behavior worth noting if you're less familiar with JavaScript. TypeScript can often help you catch bugs early on, but if you choose to do *nothing* with a value, there's only so much that it can do without being overly prescriptive. If you want, you can make sure you handle situations like these with a linter.

One last word on narrowing by truthiness is that Boolean negations with `!` filter out from negated branches.

```
function multiplyAll(
  values: number[] | undefined,
  factor: number
): number[] | undefined {
  if (!values) {
    return values;
  } else {
    return values.map((x) => x * factor);
  }
}
```

Equality narrowing

TypeScript also uses `switch` statements and equality checks like `==`, `!=`, `==`, and `!=` to narrow types. For example:

```
function example(x: string | number, y: string | boolean) {
  if (x === y) {
    // We can now call any 'string' method on 'x' or 'y'.
    x.toUpperCase();
    // ^
    y.toLowerCase();
    // ^
  }
}
```

```

} else {
  console.log(x);
  // ^
  console.log(y);
  // ^
}
}

```

When we checked that `x` and `y` are both equal in the above example, TypeScript knew their types also had to be equal. Since `string` is the only common type that both `x` and `y` could take on, TypeScript knows that `x` and `y` must be a `string` in the first branch.

Checking against specific literal values (as opposed to variables) works also. In our section about truthiness narrowing, we wrote a `printAll` function which was error-prone because it accidentally didn't handle empty strings properly. Instead we could have done a specific check to block out `null`s, and TypeScript still correctly removes `null` from the type of `strs`.

```

function printAll(strs: string | string[] | null) {
  if (strs !== null) {
    if (typeof strs === "object") {
      for (const s of strs) {
        // ^
        console.log(s);
      }
    } else if (typeof strs === "string") {
      console.log(strs);
      // ^
    }
  }
}

```

JavaScript's looser equality checks with `==` and `!=` also get narrowed correctly. If you're unfamiliar, checking whether something `== null` actually not only checks whether it is specifically the value `null` - it also checks whether it's potentially `undefined`. The same applies to `== undefined`: it checks whether a value is either `null` or `undefined`.

```

interface Container {
  value: number | null | undefined;
}

function multiplyValue(container: Container, factor: number) {
  // Remove both 'null' and 'undefined' from the type.
  if (container.value != null) {
    console.log(container.value);
    // ^
  }

  // Now we can safely multiply 'container.value'.
  container.value *= factor;
}

```

The `in` operator narrowing

JavaScript has an operator for determining if an object has a property with a name: the `in` operator. TypeScript takes this into account as a way to narrow down potential types.

For example, with the code: `"value" in x`. where `"value"` is a string literal and `x` is a union type. The "true" branch narrows `x`'s types which have either an optional or required property `value`, and the "false" branch narrows to types which have an optional or missing property `value`.

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };

function move(animal: Fish | Bird) {
  if ("swim" in animal) {
    return animal.swim();
  }

  return animal.fly();
}
```

To reiterate optional properties will exist in both sides for narrowing, for example a human could both swim and fly (with the right equipment) and thus should show up in both sides of the `in` check:

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };
type Human = { swim?: () => void; fly?: () => void };

function move(animal: Fish | Bird | Human) {
  if ("swim" in animal) {
    animal;
    // ^
  } else {
    animal;
    // ^
  }
}
```

`instanceof` narrowing

JavaScript has an operator for checking whether or not a value is an "instance" of another value. More specifically, in JavaScript `x instanceof Foo` checks whether the *prototype chain* of `x` contains `Foo.prototype`. While we won't dive deep here, and you'll see more of this when we get into classes, they can still be useful for most values that can be constructed with `new`. As you might have guessed, `instanceof` is also a type guard, and TypeScript narrows in branches guarded by `instanceof`s.

```
function logValue(x: Date | string) {
  if (x instanceof Date) {
    console.log(x.toUTCString());
    // ^
  } else {
    console.log(x.toUpperCase());
    // ^
  }
}
```

Assignments

As we mentioned earlier, when we assign to any variable, TypeScript looks at the right side of the assignment and narrows the left side appropriately.

```
let x = Math.random() < 0.5 ? 10 : "hello world!";
// ^
x = 1;

console.log(x);
// ^
x = "goodbye!";
console.log(x);
// ^?
```

Notice that each of these assignments is valid. Even though the observed type of `x` changed to `number` after our first assignment, we were still able to assign a `string` to `x`. This is because the *declared type* of `x` - the type that `x` started with - is `string | number`, and assignability is always checked against the declared type.

If we'd assigned a `boolean` to `x`, we'd have seen an error since that wasn't part of the declared type.

```
// @errors: 2322
let x = Math.random() < 0.5 ? 10 : "hello world!";
// ^
x = 1;

console.log(x);
// ^
x = true;
console.log(x);
// ^?
```

Control flow analysis

Up until this point, we've gone through some basic examples of how TypeScript narrows within specific branches. But there's a bit more going on than just walking up from every variable and looking for type guards in `ifs`, `whiles`, conditionals, etc. For example

```
function padLeft(padding: number | string, input: string) {
  if (typeof padding === "number") {
    return " ".repeat(padding) + input;
  }
  return padding + input;
}
```

`padLeft` returns from within its first `if` block. TypeScript was able to analyze this code and see that the rest of the body (`return padding + input;`) is *unreachable* in the case where `padding` is a `number`. As a result, it was able to remove `number` from the type of `padding` (narrowing from `string | number` to `string`) for the rest of the function.

This analysis of code based on reachability is called *control flow analysis*, and TypeScript uses this flow analysis to narrow types as it encounters type guards and assignments. When a variable is analyzed, control flow can split off and re-merge over and over again, and that variable can be observed to have a different type at each point.

```
function example() {
  let x: string | number | boolean;

  x = Math.random() < 0.5;

  console.log(x);
  // ^?

  if (Math.random() < 0.5) {
    x = "Hello";
    console.log(x);
    // ^
  } else {
    x = 100;
    console.log(x);
    // ^
  }

  return x;
  // ^
}
```

Using type predicates

We've worked with existing JavaScript constructs to handle narrowing so far, however sometimes you want more direct control over how types change throughout your code.

To define a user-defined type guard, we simply need to define a function whose return type is a *type predicate*:

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };
declare function getSmallPet(): Fish | Bird;
// ---cut---
function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}
```

`pet is Fish` is our type predicate in this example. A predicate takes the form `parameterName is Type`, where `parameterName` must be the name of a parameter from the current function signature.

Any time `isFish` is called with some variable, TypeScript will *narrow* that variable to that specific type if the original type is compatible.

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };
declare function getSmallPet(): Fish | Bird;
function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}
```

```
// ---cut---
// Both calls to 'swim' and 'fly' are now okay.
let pet = getSmallPet();

if (isFish(pet)) {
  pet.swim();
} else {
  pet.fly();
}
```

Notice that TypeScript not only knows that `pet` is a `Fish` in the `if` branch; it also knows that in the `else` branch, you *don't* have a `Fish`, so you must have a `Bird`.

You may use the type guard `isFish` to filter an array of `Fish | Bird` and obtain an array of `Fish`:

```
type Fish = { swim: () => void; name: string };
type Bird = { fly: () => void; name: string };
declare function getSmallPet(): Fish | Bird;
function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}
// ---cut---
const zoo: (Fish | Bird)[] = [getSmallPet(), getSmallPet(), getSmallPet()];
const underWater1: Fish[] = zoo.filter(isFish);
// or, equivalently
const underWater2: Fish[] = zoo.filter(isFish) as Fish[];

// The predicate may need repeating for more complex examples
const underWater3: Fish[] = zoo.filter((pet): pet is Fish => {
  if (pet.name === "sharkey") return false;
  return isFish(pet);
});
```

In addition, classes can use `this is Type` to narrow their type.

Discriminated unions

Most of the examples we've looked at so far have focused around narrowing single variables with simple types like `string`, `boolean`, and `number`. While this is common, most of the time in JavaScript we'll be dealing with slightly more complex structures.

For some motivation, let's imagine we're trying to encode shapes like circles and squares. Circles keep track of their radii and squares keep track of their side lengths. We'll use a field called `kind` to tell which shape we're dealing with. Here's a first attempt at defining `Shape`.

```
interface Shape {
  kind: "circle" | "square";
  radius?: number;
  sideLength?: number;
}
```

Notice we're using a union of string literal types: `"circle"` and `"square"` to tell us whether we should treat the shape as a circle or square respectively. By using `"circle" | "square"` instead of `string`, we can avoid misspelling issues.

```
// @errors: 2367
interface Shape {
  kind: "circle" | "square";
  radius?: number;
  sideLength?: number;
}

// ---cut---
function handleShape(shape: Shape) {
  // oops!
  if (shape.kind === "rect") {
    // ...
  }
}
```

We can write a `getArea` function that applies the right logic based on if it's dealing with a circle or square. We'll first try dealing with circles.

```
// @errors: 2532
interface Shape {
  kind: "circle" | "square";
  radius?: number;
  sideLength?: number;
}

// ---cut---
function getArea(shape: Shape) {
  return Math.PI * shape.radius ** 2;
}
```

Under `strictNullChecks` that gives us an error - which is appropriate since `radius` might not be defined. But what if we perform the appropriate checks on the `kind` property?

```
// @errors: 2532
interface Shape {
  kind: "circle" | "square";
  radius?: number;
  sideLength?: number;
}

// ---cut---
function getArea(shape: Shape) {
  if (shape.kind === "circle") {
    return Math.PI * shape.radius ** 2;
  }
}
```

Hmm, TypeScript still doesn't know what to do here. We've hit a point where we know more about our values than the type checker does. We could try to use a non-null assertion (`a !` after `shape.radius`) to say that `radius` is definitely present.

```
interface Shape {
  kind: "circle" | "square";
  radius?: number;
  sideLength?: number;
}

// ---cut---
```

```
function getArea(shape: Shape) {
  if (shape.kind === "circle") {
    return Math.PI * shape.radius! ** 2;
  }
}
```

But this doesn't feel ideal. We had to shout a bit at the type-checker with those non-null assertions (`!`) to convince it that `shape.radius` was defined, but those assertions are error-prone if we start to move code around. Additionally, outside of `strictNullChecks` we're able to accidentally access any of those fields anyway (since optional properties are just assumed to always be present when reading them). We can definitely do better.

The problem with this encoding of `Shape` is that the type-checker doesn't have any way to know whether or not `radius` or `sideLength` are present based on the `kind` property. We need to communicate what we know to the type checker. With that in mind, let's take another swing at defining `Shape`.

```
interface Circle {
  kind: "circle";
  radius: number;
}

interface Square {
  kind: "square";
  sideLength: number;
}

type Shape = Circle | Square;
```

Here, we've properly separated `Shape` out into two types with different values for the `kind` property, but `radius` and `sideLength` are declared as required properties in their respective types.

Let's see what happens here when we try to access the `radius` of a `Shape`.

```
// @errors: 2339
interface Circle {
  kind: "circle";
  radius: number;
}

interface Square {
  kind: "square";
  sideLength: number;
}

type Shape = Circle | Square;

// ---cut---
function getArea(shape: Shape) {
  return Math.PI * shape.radius ** 2;
}
```

Like with our first definition of `Shape`, this is still an error. When `radius` was optional, we got an error (with `strictNullChecks` enabled) because TypeScript couldn't tell whether the property was present. Now that `Shape` is a union, TypeScript is telling us that `shape` might be a `Square`, and `Square`s don't have `radius` defined on them! Both interpretations are correct, but only the union encoding of `Shape` will cause an error regardless of how `strictNullChecks` is configured.

But what if we tried checking the `kind` property again?

```
interface Circle {
  kind: "circle";
  radius: number;
}

interface Square {
  kind: "square";
  sideLength: number;
}

type Shape = Circle | Square;

// ---cut---
function getArea(shape: Shape) {
  if (shape.kind === "circle") {
    return Math.PI * shape.radius ** 2;
    //           ^
  }
}
```

That got rid of the error! When every type in a union contains a common property with literal types, TypeScript considers that to be a *discriminated union*, and can narrow out the members of the union.

In this case, `kind` was that common property (which is what's considered a *discriminant* property of `Shape`). Checking whether the `kind` property was `"circle"` got rid of every type in `Shape` that didn't have a `kind` property with the type `"circle"`. That narrowed `shape` down to the type `Circle`.

The same checking works with `switch` statements as well. Now we can try to write our complete `getArea` without any pesky `!` non-null assertions.

```
interface Circle {
  kind: "circle";
  radius: number;
}

interface Square {
  kind: "square";
  sideLength: number;
}

type Shape = Circle | Square;

// ---cut---
function getArea(shape: Shape) {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    //           ^
  }
}
```

```

    case "square":  
      return shape.sideLength ** 2;  
      // ^?  
    }  

}

```

The important thing here was the encoding of `Shape`. Communicating the right information to TypeScript - that `Circle` and `Square` were really two separate types with specific `kind` fields - was crucial. Doing that let us write type-safe TypeScript code that looks no different than the JavaScript we would've written otherwise. From there, the type system was able to do the "right" thing and figure out the types in each branch of our `switch` statement.

As an aside, try playing around with the above example and remove some of the return keywords. You'll see that type-checking can help avoid bugs when accidentally falling through different clauses in a `switch` statement.

Discriminated unions are useful for more than just talking about circles and squares. They're good for representing any sort of messaging scheme in JavaScript, like when sending messages over the network (client/server communication), or encoding mutations in a state management framework.

The `never` type

When narrowing, you can reduce the options of a union to a point where you have removed all possibilities and have nothing left. In those cases, TypeScript will use a `never` type to represent a state which shouldn't exist.

Exhaustiveness checking

The `never` type is assignable to every type; however, no type is assignable to `never` (except `never` itself). This means you can use narrowing and rely on `never` turning up to do exhaustive checking in a `switch` statement.

For example, adding a `default` to our `getArea` function which tries to assign the shape to `never` will raise when every possible case has not been handled.

```

interface Circle {  
  kind: "circle";  
  radius: number;  
}  
  
interface Square {  
  kind: "square";  
  sideLength: number;  
}  
// ---cut---  
type Shape = Circle | Square;

```

```
function getArea(shape: Shape) {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.sideLength ** 2;
    default:
      const _exhaustiveCheck: never = shape;
      return _exhaustiveCheck;
  }
}
```

Adding a new member to the `Shape` union, will cause a TypeScript error:

```
// @errors: 2322
interface Circle {
  kind: "circle";
  radius: number;
}

interface Square {
  kind: "square";
  sideLength: number;
}
// ---cut---
interface Triangle {
  kind: "triangle";
  sideLength: number;
}

type Shape = Circle | Square | Triangle;

function getArea(shape: Shape) {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.sideLength ** 2;
    default:
      const _exhaustiveCheck: never = shape;
      return _exhaustiveCheck;
  }
}
```

[Go to TOC](#)

In JavaScript, the fundamental way that we group and pass around data is through objects. In TypeScript, we represent those through *object types*.

As we've seen, they can be anonymous:

```
function greet(person: { name: string; age: number }) {
  // ^^^^^^
  // ^^^^^^
  return "Hello " + person.name;
}
```

or they can be named by using either an interface

```
interface Person {
  // ^^^^
  name: string;
  age: number;
}

function greet(person: Person) {
  return "Hello " + person.name;
}
```

or a type alias.

```
type Person = {
  // ^^^^
  name: string;
  age: number;
};

function greet(person: Person) {
  return "Hello " + person.name;
}
```

In all three examples above, we've written functions that take objects that contain the property `name` (which must be a `string`) and `age` (which must be a `number`).

Property Modifiers

Each property in an object type can specify a couple of things: the type, whether the property is optional, and whether the property can be written to.

Optional Properties

Much of the time, we'll find ourselves dealing with objects that *might* have a property set. In those cases, we can mark those properties as *optional* by adding a question mark (?) to the end of their names.

```
interface Shape {}
declare function getShape(): Shape;

// ---cut---
interface PaintOptions {
  shape: Shape;
  xPos?: number;
```

```
// ^
yPos?: number;
// ^
}

function paintShape(opts: PaintOptions) {
// ...
}

const shape = getShape();
paintShape({ shape });
paintShape({ shape, xPos: 100 });
paintShape({ shape, yPos: 100 });
paintShape({ shape, xPos: 100, yPos: 100 });
```

In this example, both `xPos` and `yPos` are considered optional. We can choose to provide either of them, so every call above to `paintShape` is valid. All optionality really says is that if the property *is* set, it better have a specific type.

We can also read from those properties - but when we do under `strictNullChecks`, TypeScript will tell us they're potentially `undefined`.

```
interface Shape {}
declare function getShape(): Shape;

interface PaintOptions {
  shape: Shape;
  xPos?: number;
  yPos?: number;
}

// ---cut---
function paintShape(opts: PaintOptions) {
  let xPos = opts.xPos;
  // ^
  let yPos = opts.yPos;
  // ^
  // ...
}
```

In JavaScript, even if the property has never been set, we can still access it - it's just going to give us the value `undefined`. We can just handle `undefined` specially.

```
interface Shape {}
declare function getShape(): Shape;

interface PaintOptions {
  shape: Shape;
  xPos?: number;
  yPos?: number;
}

// ---cut---
function paintShape(opts: PaintOptions) {
  let xPos = opts.xPos === undefined ? 0 : opts.xPos;
  // ^
  let yPos = opts.yPos === undefined ? 0 : opts.yPos;
```

```
// ^
// ...
}
```

Note that this pattern of setting defaults for unspecified values is so common that JavaScript has syntax to support it.

```
interface Shape {}  
declare function getShape(): Shape;  
  
interface PaintOptions {  
    shape: Shape;  
    xPos?: number;  
    yPos?: number;  
}  
  
// ---cut---  
function paintShape({ shape, xPos = 0, yPos = 0 }: PaintOptions) {  
    console.log("x coordinate at", xPos);  
    // ^?  
    console.log("y coordinate at", yPos);  
    // ^?  
    // ...  
}
```

Here we used a [destructuring pattern](#) for `paintShape`'s parameter, and provided [default values](#) for `xPos` and `yPos`. Now `xPos` and `yPos` are both definitely present within the body of `paintShape`, but optional for any callers to `paintShape`.

Note that there is currently no way to place type annotations within destructuring patterns. This is because the following syntax already means something different in JavaScript.

```
// @noImplicitAny: false  
// @errors: 2552 2304  
interface Shape {}  
declare function render(x: unknown);  
// ---cut---  
function draw({ shape: Shape, xPos: number = 100 /*...*/ }) {  
    render(shape);  
    render(xPos);  
}
```

In an object destructuring pattern, `shape: Shape` means "grab the property `shape` and redefine it locally as a variable named `Shape`. Likewise `xPos: number` creates a variable named `number` whose value is based on the parameter's `xPos`.

Using [mapping modifiers](#), you can remove [optional](#) attributes.

readonly Properties

Properties can also be marked as `readonly` for TypeScript. While it won't change any behavior at runtime, a property marked as `readonly` can't be written to during type-checking.

```
// @errors: 2540
interface SomeType {
  readonly prop: string;
}

function doSomething(obj: SomeType) {
  // We can read from 'obj.prop'.
  console.log(`prop has the value '${obj.prop}'`);

  // But we can't re-assign it.
  obj.prop = "hello";
}
```

Using the `readonly` modifier doesn't necessarily imply that a value is totally immutable - or in other words, that its internal contents can't be changed. It just means the property itself can't be re-written to.

```
// @errors: 2540
interface Home {
  readonly resident: { name: string; age: number };
}

function visitForBirthday(home: Home) {
  // We can read and update properties from 'home.resident'.
  console.log(`Happy birthday ${home.resident.name}!`);
  home.resident.age++;
}

function evict(home: Home) {
  // But we can't write to the 'resident' property itself on a 'Home'.
  home.resident = {
    name: "Victor the Evictor",
    age: 42,
  };
}
```

It's important to manage expectations of what `readonly` implies. It's useful to signal intent during development time for TypeScript on how an object should be used. TypeScript doesn't factor in whether properties on two types are `readonly` when checking whether those types are compatible, so `readonly` properties can also change via aliasing.

```
interface Person {
  name: string;
  age: number;
}

interface ReadonlyPerson {
  readonly name: string;
  readonly age: number;
}

let writablePerson: Person = {
  name: "Person McPersonface",
  age: 42,
};

// works
let readonlyPerson: ReadonlyPerson = writablePerson;
```

```
console.log(readonlyPerson.age); // prints '42'
writablePerson.age++;
console.log(readonlyPerson.age); // prints '43'
```

Using [mapping modifiers](#), you can remove `readonly` attributes.

Index Signatures

Sometimes you don't know all the names of a type's properties ahead of time, but you do know the shape of the values.

In those cases you can use an index signature to describe the types of possible values, for example:

```
declare function getStringArray(): StringArray;
// ---cut---
interface StringArray {
  [index: number]: string;
}

const myArray: StringArray = getStringArray();
const secondItem = myArray[1];
// ^?
```

Above, we have a `StringArray` interface which has an index signature. This index signature states that when a `StringArray` is indexed with a `number`, it will return a `string`.

Only some types are allowed for index signature properties: `string`, `number`, `symbol`, template string patterns, and union types consisting only of these.

It is possible to support both types of indexers...

► Details

While string index signatures are a powerful way to describe the "dictionary" pattern, they also enforce that all properties match their return type. This is because a string index declares that `obj.property` is also available as `obj["property"]`. In the following example, `name`'s type does not match the string index's type, and the type checker gives an error:

```
// @errors: 2411
// @errors: 2411
interface NumberDictionary {
  [index: string]: number;
  length: number; // ok
  name: string;
}
```

However, properties of different types are acceptable if the index signature is a union of the property types:

```
interface NumberOrStringDictionary {
  [index: string]: number | string;
  length: number; // ok, length is a number
  name: string; // ok, name is a string
}
```

Finally, you can make index signatures `readonly` in order to prevent assignment to their indices:

```
declare function getReadOnlyStringArray(): ReadonlyStringArray;
// ---cut---
// @errors: 2542
interface ReadonlyStringArray {
  readonly [index: number]: string;
}

let myArray: ReadonlyStringArray = getReadOnlyStringArray();
myArray[2] = "Mallory";
```

You can't set `myArray[2]` because the index signature is `readonly`.

Extending Types

It's pretty common to have types that might be more specific versions of other types. For example, we might have a `BasicAddress` type that describes the fields necessary for sending letters and packages in the U.S.

```
interface BasicAddress {
  name?: string;
  street: string;
  city: string;
  country: string;
  postalCode: string;
}
```

In some situations that's enough, but addresses often have a unit number associated with them if the building at an address has multiple units. We can then describe an `AddressWithUnit`.

```
interface AddressWithUnit {
  name?: string;
  unit: string;
  //^^^^^
  street: string;
  city: string;
  country: string;
  postalCode: string;
}
```

This does the job, but the downside here is that we had to repeat all the other fields from `BasicAddress` when our changes were purely additive. Instead, we can extend the original `BasicAddress` type and just add the new fields that are unique to `AddressWithUnit`.

```
interface BasicAddress {
  name?: string;
  street: string;
  city: string;
  country: string;
  postalCode: string;
}

interface AddressWithUnit extends BasicAddress {
  unit: string;
}
```

The `extends` keyword on an `interface` allows us to effectively copy members from other named types, and add whatever new members we want. This can be useful for cutting down the amount of type declaration boilerplate we have to write, and for signaling intent that several different declarations of the same property might be related. For example, `AddressWithUnit` didn't need to repeat the `street` property, and because `street` originates from `BasicAddress`, a reader will know that those two types are related in some way.

`interface`s can also extend from multiple types.

```
interface Colorful {
  color: string;
}

interface Circle {
  radius: number;
}

interface ColorfulCircle extends Colorful, Circle {}
const cc: ColorfulCircle = {
  color: "red",
  radius: 42,
};
```

Intersection Types

`interface`s allowed us to build up new types from other types by extending them. TypeScript provides another construct called *intersection types* that is mainly used to combine existing object types.

An intersection type is defined using the `&` operator.

```
interface Colorful {
  color: string;
}
interface Circle {
  radius: number;
}

type ColorfulCircle = Colorful & Circle;
```

Here, we've intersected `Colorful` and `Circle` to produce a new type that has all the members of `Colorful` and `Circle`.

```
// @errors: 2345
interface Colorful {
  color: string;
}
interface Circle {
  radius: number;
}
// ---cut---
function draw(circle: Colorful & Circle) {
  console.log(`Color was ${circle.color}`);
  console.log(`Radius was ${circle.radius}`);
}

// okay
draw({ color: "blue", radius: 42 });

// oops
draw({ color: "red", raidus: 42 });
```

Interfaces vs. Intersections

We just looked at two ways to combine types which are similar, but are actually subtly different. With interfaces, we could use an `extends` clause to extend from other types, and we were able to do something similar with intersections and name the result with a type alias. The principle difference between the two is how conflicts are handled, and that difference is typically one of the main reasons why you'd pick one over the other between an interface and a type alias or an intersection type.

Generic Object Types

Let's imagine a `Box` type that can contain any value - `string`s, `number`s, `Giraffe`s, whatever.

```
interface Box {
  contents: any;
}
```

Right now, the `contents` property is typed as `any`, which works, but can lead to accidents down the line.

We could instead use `unknown`, but that would mean that in cases where we already know the type of `contents`, we'd need to do precautionary checks, or use error-prone type assertions.

```

interface Box {
  contents: unknown;
}

let x: Box = {
  contents: "hello world",
};

// we could check 'x.contents'
if (typeof x.contents === "string") {
  console.log(x.contents.toLowerCase());
}

// or we could use a type assertion
console.log((x.contents as string).toLowerCase());

```

One type safe approach would be to instead scaffold out different `Box` types for every type of `contents`.

```

// @errors: 2322
interface NumberBox {
  contents: number;
}

interface StringBox {
  contents: string;
}

interface BooleanBox {
  contents: boolean;
}

```

But that means we'll have to create different functions, or overloads of functions, to operate on these types.

```

interface NumberBox {
  contents: number;
}

interface StringBox {
  contents: string;
}

interface BooleanBox {
  contents: boolean;
}
// ---cut---
function setContents(box: StringBox, newContents: string): void;
function setContents(box: NumberBox, newContents: number): void;
function setContents(box: BooleanBox, newContents: boolean): void;
function setContents(box: { contents: any }, newContents: any) {
  box.contents = newContents;
}

```

That's a lot of boilerplate. Moreover, we might later need to introduce new types and overloads. This is frustrating, since our box types and overloads are all effectively the same.

Instead, we can make a *generic* `Box` type which declares a *type parameter*.

```
interface Box<Type> {
  contents: Type;
}
```

You might read this as "A `Box` of `Type` is something whose `contents` have type `Type`". Later on, when we refer to `Box`, we have to give a *type argument* in place of `Type`.

```
interface Box<Type> {
  contents: Type;
}
// ---cut---
let box: Box<string>;
```

Think of `Box` as a template for a real type, where `Type` is a placeholder that will get replaced with some other type. When TypeScript sees `Box<string>`, it will replace every instance of `Type` in `Box<Type>` with `string`, and end up working with something like `{ contents: string }`. In other words, `Box<string>` and our earlier `StringBox` work identically.

```
interface Box<Type> {
  contents: Type;
}
interface StringBox {
  contents: string;
}

let boxA: Box<string> = { contents: "hello" };
boxA.contents;
// ^?

let boxB: StringBox = { contents: "world" };
boxB.contents;
// ^?
```

`Box` is reusable in that `Type` can be substituted with anything. That means that when we need a box for a new type, we don't need to declare a new `Box` type at all (though we certainly could if we wanted to).

```
interface Box<Type> {
  contents: Type;
}

interface Apple {
  // ...
}

// Same as '{ contents: Apple }'.
type AppleBox = Box<Apple>;
```

This also means that we can avoid overloads entirely by instead using [generic functions](#).

```
interface Box<Type> {
  contents: Type;
}
// ---cut---
```

```
function setContents<Type>(box: Box<Type>, newContents: Type) {
  box.contents = newContents;
}
```

It is worth noting that type aliases can also be generic. We could have defined our new `Box<Type>` interface, which was:

```
interface Box<Type> {
  contents: Type;
}
```

by using a type alias instead:

```
type Box<Type> = {
  contents: Type;
};
```

Since type aliases, unlike interfaces, can describe more than just object types, we can also use them to write other kinds of generic helper types.

```
// @errors: 2575
type OrNull<Type> = Type | null;

type OneOrMany<Type> = Type | Type[];
type OneOrManyOrNull<Type> = OrNull<OneOrMany<Type>>;
//   ^
type OneOrManyOrNullStrings = OneOrManyOrNull<string>;
//   ^?
```

We'll circle back to type aliases in just a little bit.

The `Array` Type

Generic object types are often some sort of container type that work independently of the type of elements they contain. It's ideal for data structures to work this way so that they're re-usable across different data types.

It turns out we've been working with a type just like that throughout this handbook: the `Array` type. Whenever we write out types like `number[]` or `string[]`, that's really just a shorthand for `Array<number>` and `Array<string>`.

```
function doSomething(value: Array<string>) {
  // ...
}

let myArray: string[] = ["hello", "world"];
// either of these work!
doSomething(myArray);
doSomething(new Array("hello", "world"));
```

Much like the `Box` type above, `Array` itself is a generic type.

```
// @noLib: true
interface Number {}
interface String {}
interface Boolean {}
interface Symbol {}

// ---cut---
interface Array<Type> {
  /**
   * Gets or sets the length of the array.
   */
  length: number;

  /**
   * Removes the last element from an array and returns it.
   */
  pop(): Type | undefined;

  /**
   * Appends new elements to an array, and returns the new length of the array.
   */
  push(...items: Type[]): number;

  // ...
}
```

Modern JavaScript also provides other data structures which are generic, like `Map<K, V>`, `Set<T>`, and `Promise<T>`. All this really means is that because of how `Map`, `Set`, and `Promise` behave, they can work with any sets of types.

The `ReadonlyArray` Type

The `ReadonlyArray` is a special type that describes arrays that shouldn't be changed.

```
// @errors: 2339
function doStuff(values: ReadonlyArray<string>) {
  // We can read from 'values'...
  const copy = values.slice();
  console.log(`The first value is ${values[0]}`);

  // ...but we can't mutate 'values'.
  values.push("hello!");
}
```

Much like the `readonly` modifier for properties, it's mainly a tool we can use for intent. When we see a function that returns `ReadonlyArray`s, it tells us we're not meant to change the contents at all, and when we see a function that consumes `ReadonlyArray`s, it tells us that we can pass any array into that function without worrying that it will change its contents.

Unlike `Array`, there isn't a `ReadonlyArray` constructor that we can use.

```
// @errors: 2693
new ReadonlyArray("red", "green", "blue");
```

Instead, we can assign regular `Array`s to `ReadonlyArray`s.

```
const roArray: ReadonlyArray<string> = ["red", "green", "blue"];
```

Just as TypeScript provides a shorthand syntax for `Array<Type>` with `Type[]`, it also provides a shorthand syntax for `ReadonlyArray<Type>` with `readonly Type[]`.

```
// @errors: 2339
function doStuff(values: readonly string[]) {
  // ^^^^^^
  // We can read from 'values'...
  const copy = values.slice();
  console.log(`The first value is ${values[0]}`);

  // ...but we can't mutate 'values'.
  values.push("hello!");
}
```

One last thing to note is that unlike the `readonly` property modifier, assignability isn't bidirectional between regular `Array`s and `ReadonlyArray`s.

```
// @errors: 4104
let x: readonly string[] = [];
let y: string[] = [];

x = y;
y = x;
```

Tuple Types

A *tuple* type is another sort of `Array` type that knows exactly how many elements it contains, and exactly which types it contains at specific positions.

```
type StringNumberPair = [string, number];
// ^^^^^^
```

Here, `StringNumberPair` is a tuple type of `string` and `number`. Like `ReadonlyArray`, it has no representation at runtime, but is significant to TypeScript. To the type system, `StringNumberPair` describes arrays whose `0` index contains a `string` and whose `1` index contains a `number`.

```
function doSomething(pair: [string, number]) {
  const a = pair[0];
  // ^
  const b = pair[1];
  // ^
  // ...
}

doSomething(["hello", 42]);
```

If we try to index past the number of elements, we'll get an error.

```
// @errors: 2493
function doSomething(pair: [string, number]) {
  // ...

  const c = pair[2];
}
```

We can also [destructure tuples](#) using JavaScript's array destructuring.

```
function doSomething(stringHash: [string, number]) {
  const [inputString, hash] = stringHash;

  console.log(inputString);
  //           ^
  console.log(hash);
  //           ^
}
```

Tuple types are useful in heavily convention-based APIs, where each element's meaning is "obvious". This gives us flexibility in whatever we want to name our variables when we destructure them. In the above example, we were able to name elements `0` and `1` to whatever we wanted.

However, since not every user holds the same view of what's obvious, it may be worth reconsidering whether using objects with descriptive property names may be better for your API.

Other than those length checks, simple tuple types like these are equivalent to types which are versions of `Array`s that declare properties for specific indexes, and that declare `length` with a numeric literal type.

```
interface StringNumberPair {
  // specialized properties
  length: 2;
  0: string;
  1: number;

  // Other 'Array<string | number>' members...
  slice(start?: number, end?: number): Array<string | number>;
}
```

Another thing you may be interested in is that tuples can have optional properties by writing out a question mark (`?` after an element's type). Optional tuple elements can only come at the end, and also affect the type of `length`.

```
type Either2dOr3d = [number, number, number?];

function setCoordinate(coord: Either2dOr3d) {
  const [x, y, z?] = coord;
  //           ^
  console.log(`Provided coordinates had ${coord.length} dimensions`);
  //           ^
}
```

Tuples can also have rest elements, which have to be an array/tuple type.

```
type StringNumberBooleans = [string, number, ...boolean[]];
type StringBooleansNumber = [string, ...boolean[], number];
type BooleansStringNumber = [...boolean[], string, number];
```

- `StringNumberBooleans` describes a tuple whose first two elements are `string` and `number` respectively, but which may have any number of `boolean`s following.
- `StringBooleansNumber` describes a tuple whose first element is `string` and then any number of `boolean`s and ending with a `number`.
- `BooleansStringNumber` describes a tuple whose starting elements are any number of `boolean`s and ending with a `string` then a `number`.

A tuple with a rest element has no set "length" - it only has a set of well-known elements in different positions.

```
type StringNumberBooleans = [string, number, ...boolean[]];
// ---cut---
const a: StringNumberBooleans = ["hello", 1];
const b: StringNumberBooleans = ["beautiful", 2, true];
const c: StringNumberBooleans = ["world", 3, true, false, true];
```

Why might optional and rest elements be useful? Well, it allows TypeScript to correspond tuples with parameter lists. Tuples types can be used in [rest parameters and arguments](#), so that the following:

```
function readButtonInput(...args: [string, number, ...boolean[]]) {
  const [name, version, ...input] = args;
  // ...
}
```

is basically equivalent to:

```
function readButtonInput(name: string, version: number, ...input: boolean[]) {
  // ...
}
```

This is handy when you want to take a variable number of arguments with a rest parameter, and you need a minimum number of elements, but you don't want to introduce intermediate variables.

readonly Tuple Types

One final note about tuple types - tuples types have `readonly` variants, and can be specified by sticking a `readonly` modifier in front of them - just like with array shorthand syntax.

```
function doSomething(pair: readonly [string, number]) {
  // ^^^^^^
  // ...
}
```

As you might expect, writing to any property of a `readonly` tuple isn't allowed in TypeScript.

```
// @errors: 2540
function doSomething(pair: readonly [string, number]) {
  pair[0] = "hello!";
}
```

Tuples tend to be created and left un-modified in most code, so annotating types as `readonly` tuples when possible is a good default. This is also important given that array literals with `const` assertions will be inferred with `readonly` tuple types.

```
// @errors: 2345
let point = [3, 4] as const;

function distanceFromOrigin([x, y]: [number, number]) {
    return Math.sqrt(x ** 2 + y ** 2);
}

distanceFromOrigin(point);
```

Here, `distanceFromOrigin` never modifies its elements, but expects a mutable tuple. Since `point`'s type was inferred as `readonly [3, 4]`, it won't be compatible with `[number, number]` since that type can't guarantee `point`'s elements won't be mutated.

[Go to TOC](#)

About this Handbook

Over 20 years after its introduction to the programming community, JavaScript is now one of the most widespread cross-platform languages ever created. Starting as a small scripting language for adding trivial interactivity to webpages, JavaScript has grown to be a language of choice for both frontend and backend applications of every size. While the size, scope, and complexity of programs written in JavaScript has grown exponentially, the ability of the JavaScript language to express the relationships between different units of code has not. Combined with JavaScript's rather peculiar runtime semantics, this mismatch between language and program complexity has made JavaScript development a difficult task to manage at scale.

The most common kinds of errors that programmers write can be described as type errors: a certain kind of value was used where a different kind of value was expected. This could be due to simple typos, a failure to understand the API surface of a library, incorrect assumptions about runtime behavior, or other errors. The goal of TypeScript is to be a static typechecker for JavaScript programs - in other words, a tool that runs before your code runs (static) and ensures that the types of the program are correct (typechecked).

If you are coming to TypeScript without a JavaScript background, with the intention of TypeScript being your first language, we recommend you first start reading the documentation on either the [Microsoft Learn JavaScript tutorial](#) or read [JavaScript at the Mozilla Web Docs](#). If you have experience in other languages, you should be able to pick up JavaScript syntax quite quickly by reading the handbook.

How is this Handbook Structured

The handbook is split into two sections:

- **The Handbook**

The TypeScript Handbook is intended to be a comprehensive document that explains TypeScript to everyday programmers. You can read the handbook by going from top to bottom in the left-hand navigation.

You should expect each chapter or page to provide you with a strong understanding of the given concepts. The TypeScript Handbook is not a complete language specification, but it is intended to be a comprehensive guide to all of the language's features and behaviors.

A reader who completes the walkthrough should be able to:

- Read and understand commonly-used TypeScript syntax and patterns
- Explain the effects of important compiler options
- Correctly predict type system behavior in most cases

In the interests of clarity and brevity, the main content of the Handbook will not explore every edge case or minutiae of the features being covered. You can find more details on particular concepts in the reference articles.

- **Reference Files**

The reference section below the handbook in the navigation is built to provide a richer understanding of how a particular part of TypeScript works. You can read it top-to-bottom, but each section aims to provide a deeper explanation of a single concept - meaning there is no aim for continuity.

Non-Goals

The Handbook is also intended to be a concise document that can be comfortably read in a few hours. Certain topics won't be covered in order to keep things short.

Specifically, the Handbook does not fully introduce core JavaScript basics like functions, classes, and closures. Where appropriate, we'll include links to background reading that you can use to read up on those concepts.

The Handbook also isn't intended to be a replacement for a language specification. In some cases, edge cases or formal descriptions of behavior will be skipped in favor of high-level, easier-to-understand explanations. Instead, there are separate reference pages that more precisely and formally describe many aspects of TypeScript's behavior. The reference pages are not intended for readers unfamiliar with TypeScript, so they may use advanced terminology or reference topics you haven't read about yet.

Finally, the Handbook won't cover how TypeScript interacts with other tools, except where necessary. Topics like how to configure TypeScript with webpack, rollup, parcel, react, babel, closure, lerna, rush, bazel, preact, vue, angular, svelte, jquery, yarn, or npm are out of scope - you can find these resources elsewhere on the web.

Get Started

Before getting started with [The Basics](#), we recommend reading one of the following introductory pages. These introductions are intended to highlight key similarities and differences between TypeScript and your favored programming language, and clear up common misconceptions specific to those languages.

- [TypeScript for New Programmers](#)
- [TypeScript for JavaScript Programmers](#)
- [TypeScript for OOP Programmers](#)
- [TypeScript for Functional Programmers](#)

Otherwise, jump to [The Basics](#).

[Go to TOC](#)

Throughout the sections you've read so far, we've been demonstrating basic TypeScript concepts using the built-in functions present in all JavaScript runtimes. However, almost all JavaScript today includes many libraries to accomplish common tasks. Having types for the parts of your application that *aren't* your code will greatly improve your TypeScript experience. Where do these types come from?

What Do Type Declarations Look Like?

Let's say you write some code like this:

```
// @errors: 2339
const k = Math.max(5, 6);
const j = Math.mix(7, 8);
```

How did TypeScript know that `max` was present but not `mix`, even though `Math`'s implementation wasn't part of your code?

The answer is that there are *declaration files* describing these built-in objects. A declaration file provides a way to *declare* the existence of some types or values without actually providing implementations for those values.

.d.ts files

TypeScript has two main kinds of files. `.ts` files are *implementation* files that contain types and executable code. These are the files that produce `.js` outputs, and are where you'd normally write your code.

`.d.ts` files are *declaration* files that contain *only* type information. These files don't produce `.js` outputs; they are only used for typechecking. We'll learn more about how to write our own declaration files later.

Built-in Type Definitions

TypeScript includes declaration files for all of the standardized built-in APIs available in JavaScript runtimes. This includes things like methods and properties of built-in types like `string` or `function`, top-level names like `Math` and `Object`, and their associated types. By default, TypeScript also includes types for things available when running inside the browser, such as `window` and `document`; these are collectively referred to as the DOM APIs.

TypeScript names these declaration files with the pattern `lib.[something].d.ts`. If you navigate into a file with that name, you can know that you're dealing with some built-in part of the platform, not user code.

target setting

The methods, properties, and functions available to you actually vary based on the *version* of JavaScript your code is running on. For example, the `startsWith` method of strings is available only starting with the version of JavaScript referred as *ECMAScript 6*.

Being aware of what version of JavaScript your code ultimately runs on is important because you don't want to use APIs that are from a newer version than the platform you deploy to. This is one function of the `target` compiler setting.

TypeScript helps with this problem by varying which `lib` files are included by default based on your `target` setting. For example, if `target` is `ES5`, you will see an error if trying to use the `startsWith` method, because that method is only available in `ES6` or later.

`lib` setting

The `lib` setting allows more fine-grained control of which built-in declaration files are considered available in your program. See the documentation page on `lib` for more information.

External Definitions

For non-built-in APIs, there are a variety of ways you can get declaration files. How you do this depends on exactly which library you're getting types for.

Bundled Types

If a library you're using is published as an npm package, it may include type declaration files as part of its distribution already. You can read the project's documentation to find out, or simply try importing the package and see if TypeScript is able to automatically resolve the types for you.

If you're a package author considering bundling type definitions with your package, you can read our guide on [bundling type definitions](#).

DefinitelyTyped / `@types`

The [DefinitelyTyped repository](#) is a centralized repo storing declaration files for thousands of libraries. The vast majority of commonly-used libraries have declaration files available on DefinitelyTyped.

Definitions on DefinitelyTyped are also automatically published to npm under the `@types` scope. The name of the types package is always the same as the name of the underlying package itself. For example, if you installed the `react` npm package, you can install its corresponding types by running

```
npm install --save-dev @types/react
```

TypeScript automatically finds type definitions under `node_modules/@types`, so there's no other step needed to get these types available in your program.

Your Own Definitions

In the uncommon event that a library didn't bundle its own types and didn't have a definition on DefinitelyTyped, you can write a declaration file yourself. See the appendix [Writing Declaration Files](#) for a guide.

If you want to silence warnings about a particular module without writing a declaration file, you can also quickly declare the module as type `any` by putting an empty declaration for it in a `.d.ts` file in your project. For example, if you wanted to use a module named `some-untyped-module` without having definitions for it, you would write:

```
declare module "some-untyped-module";
```

[Go to TOC](#)

At the heart of most useful programs, we have to make decisions based on input. JavaScript programs are no different, but given the fact that values can be easily introspected, those decisions are also based on the types of the inputs. *Conditional types* help describe the relation between the types of inputs and outputs.

```
interface Animal {
  live(): void;
}
interface Dog extends Animal {
  woof(): void;
}

type Example1 = Dog extends Animal ? number : string;
// ^?

type Example2 = RegExp extends Animal ? number : string;
// ^?
```

Conditional types take a form that looks a little like conditional expressions (`condition ? trueExpression : falseExpression`) in JavaScript:

```
type SomeType = any;
type OtherType = any;
type TrueType = any;
type FalseType = any;
type Stuff =
  // ---cut---
  SomeType extends OtherType ? TrueType : FalseType;
```

When the type on the left of the `extends` is assignable to the one on the right, then you'll get the type in the first branch (the "true" branch); otherwise you'll get the type in the latter branch (the "false" branch).

From the examples above, conditional types might not immediately seem useful - we can tell ourselves whether or not `Dog extends Animal` and pick `number` or `string`! But the power of conditional types comes from using them with generics.

For example, let's take the following `createLabel` function:

```
interface IdLabel {
  id: number /* some fields */;
}
interface NameLabel {
  name: string /* other fields */;
}

function createLabel(id: number): IdLabel;
function createLabel(name: string): NameLabel;
function createLabel(nameOrId: string | number): IdLabel | NameLabel;
function createLabel(nameOrId: string | number): IdLabel | NameLabel {
  throw "unimplemented";
}
```

These overloads for `createLabel` describe a single JavaScript function that makes a choice based on the types of its inputs. Note a few things:

1. If a library has to make the same sort of choice over and over throughout its API, this becomes cumbersome.
2. We have to create three overloads: one for each case when we're *sure* of the type (one for `string` and one for `number`), and one for the most general case (taking a `string | number`). For every new type `createLabel` can handle, the number of overloads grows exponentially.

Instead, we can encode that logic in a conditional type:

```
interface IdLabel {
  id: number /* some fields */;
}
interface NameLabel {
  name: string /* other fields */;
}
// ---cut---
type NameOrId<T extends number | string> = T extends number
  ? IdLabel
  : NameLabel;
```

We can then use that conditional type to simplify our overloads down to a single function with no overloads.

```
interface IdLabel {
  id: number /* some fields */;
}
interface NameLabel {
  name: string /* other fields */;
}
type NameOrId<T extends number | string> = T extends number
  ? IdLabel
  : NameLabel;
// ---cut---
function createLabel<T extends number | string>(idOrName: T): NameOrId<T> {
  throw "unimplemented";
}

let a = createLabel("typescript");
// ^?

let b = createLabel(2.8);
// ^?

let c = createLabel(Math.random() ? "hello" : 42);
// ^?
```

Conditional Type Constraints

Often, the checks in a conditional type will provide us with some new information. Just like with narrowing with type guards can give us a more specific type, the true branch of a conditional type will further constrain generics by the type we check against.

For example, let's take the following:

```
// @errors: 2536
type MessageOf<T> = T["message"];
```

In this example, TypeScript errors because `T` isn't known to have a property called `message`. We could constrain `T`, and TypeScript would no longer complain:

```
type MessageOf<T extends { message: unknown }> = T["message"];

interface Email {
  message: string;
}

type EmailMessageContents = MessageOf<Email>;
// ^?
```

However, what if we wanted `MessageOf` to take any type, and default to something like `never` if a `message` property isn't available? We can do this by moving the constraint out and introducing a conditional type:

```
type MessageOf<T> = T extends { message: unknown } ? T["message"] : never;

interface Email {
  message: string;
}

interface Dog {
  bark(): void;
}

type EmailMessageContents = MessageOf<Email>;
// ^?

type DogMessageContents = MessageOf<Dog>;
// ^?
```

Within the true branch, TypeScript knows that `T` *will* have a `message` property.

As another example, we could also write a type called `Flatten` that flattens array types to their element types, but leaves them alone otherwise:

```
type Flatten<T> = T extends any[] ? T[number] : T;

// Extracts out the element type.
type Str = Flatten<string[]>;
// ^?

// Leaves the type alone.
type Num = Flatten<number>;
// ^?
```

When `Flatten` is given an array type, it uses an indexed access with `number` to fetch out `string[]`'s element type. Otherwise, it just returns the type it was given.

Inferring Within Conditional Types

We just found ourselves using conditional types to apply constraints and then extract out types. This ends up being such a common operation that conditional types make it easier.

Conditional types provide us with a way to infer from types we compare against in the true branch using the `infer` keyword. For example, we could have inferred the element type in `Flatten` instead of fetching it out "manually" with an indexed access type:

```
type Flatten<Type> = Type extends Array<infer Item> ? Item : Type;
```

Here, we used the `infer` keyword to declaratively introduce a new generic type variable named `Item` instead of specifying how to retrieve the element type of `T` within the true branch. This frees us from having to think about how to dig through and probing apart the structure of the types we're interested in.

We can write some useful helper type aliases using the `infer` keyword. For example, for simple cases, we can extract the return type out from function types:

```
type GetReturnType<Type> = Type extends (...args: never[]) => infer Return
? Return
: never;

type Num = GetReturnType<() => number>;
// ^?

type Str = GetReturnType<(x: string) => string>;
// ^?

type Bools = GetReturnType<(a: boolean, b: boolean) => boolean[]>;
// ^?
```

When inferring from a type with multiple call signatures (such as the type of an overloaded function), inferences are made from the *last* signature (which, presumably, is the most permissive catch-all case). It is not possible to perform overload resolution based on a list of argument types.

```
declare function stringOrNum(x: string): number;
declare function stringOrNum(x: number): string;
declare function stringOrNum(x: string | number): string | number;

type T1 = ReturnType<typeof stringOrNum>;
// ^?
```

Distributive Conditional Types

When conditional types act on a generic type, they become *distributive* when given a union type. For example, take the following:

```
type ToArray<Type> = Type extends any ? Type[] : never;
```

If we plug a union type into `ToArray`, then the conditional type will be applied to each member of that union.

```
type ToArray<Type> = Type extends any ? Type[] : never;

type StrArrOrNumArr = ToArray<string | number>;
// ^?
```

What happens here is that `StrArrOrNumArr` distributes on:

```
type StrArrOrNumArr =
// ---cut---
string | number;
```

and maps over each member type of the union, to what is effectively:

```
type ToArray<Type> = Type extends any ? Type[] : never;
type StrArrOrNumArr =
// ---cut---
ToArray<string> | ToArray<number>;
```

which leaves us with:

```
type StrArrOrNumArr =
// ---cut---
string[] | number[];
```

Typically, distributivity is the desired behavior. To avoid that behavior, you can surround each side of the `extends` keyword with square brackets.

```
type ToArrayNonDist<Type> = [Type] extends [any] ? Type[] : never;
// 'StrArrOrNumArr' is no longer a union.
type StrArrOrNumArr = ToArrayNonDist<string | number>;
// ^?
```

[Go to TOC](#)

A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable. Components that are capable of working on the data of today as well as the data of tomorrow will give you the most flexible capabilities for building up large software systems.

In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is *generics*, that is, being able to create a component that can work over a variety of types rather than a single one. This allows users to consume these components and use their own types.

Hello World of Generics

To start off, let's do the "hello world" of generics: the identity function. The identity function is a function that will return back whatever is passed in. You can think of this in a similar way to the `echo` command.

Without generics, we would either have to give the identity function a specific type:

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

Or, we could describe the identity function using the `any` type:

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

While using `any` is certainly generic in that it will cause the function to accept any and all types for the type of `arg`, we actually are losing the information about what that type was when the function returns. If we passed in a number, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned. Here, we will use a *type variable*, a special kind of variable that works on types rather than values.

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

We've now added a type variable `Type` to the identity function. This `Type` allows us to capture the type the user provides (e.g. `number`), so that we can use that information later. Here, we use `Type` again as the return type. On inspection, we can now see the same type is used for the argument and the return type. This allows us to traffic that type information in one side of the function and out the other.

We say that this version of the `identity` function is *generic*, as it works over a range of types. Unlike using `any`, it's also just as precise (i.e., it doesn't lose any information) as the first `identity` function that used numbers for the argument and return type.

Once we've written the generic identity function, we can call it in one of two ways. The first way is to pass all of the arguments, including the type argument, to the function:

```
function identity<Type>(arg: Type): Type {
    return arg;
}
// ---cut---
let output = identity<string>("myString");
//     ^?
```

Here we explicitly set `Type` to be `string` as one of the arguments to the function call, denoted using the `<>` around the arguments rather than `()`.

The second way is also perhaps the most common. Here we use *type argument inference* -- that is, we want the compiler to set the value of `Type` for us automatically based on the type of the argument we pass in:

```
function identity<Type>(arg: Type): Type {
    return arg;
}
// ---cut---
let output = identity("myString");
//     ^?
```

Notice that we didn't have to explicitly pass the type in the angle brackets (`<>`); the compiler just looked at the value `"myString"`, and set `Type` to its type. While type argument inference can be a helpful tool to keep code shorter and more readable, you may need to explicitly pass in the type arguments as we did in the previous example when the compiler fails to infer the type, as may happen in more complex examples.

Working with Generic Type Variables

When you begin to use generics, you'll notice that when you create generic functions like `identity`, the compiler will enforce that you use any generically typed parameters in the body of the function correctly. That is, that you actually treat these parameters as if they could be any and all types.

Let's take our `identity` function from earlier:

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

What if we want to also log the length of the argument `arg` to the console with each call? We might be tempted to write this:

```
// @errors: 2339
function loggingIdentity<Type>(arg: Type): Type {
    console.log(arg.length);
    return arg;
}
```

When we do, the compiler will give us an error that we're using the `.length` member of `arg`, but nowhere have we said that `arg` has this member. Remember, we said earlier that these type variables stand in for any and all types, so someone using this function could have passed in a `number` instead, which does not have a `.length` member.

Let's say that we've actually intended this function to work on arrays of `Type` rather than `Type` directly. Since we're working with arrays, the `.length` member should be available. We can describe this just like we would create arrays of other types:

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

You can read the type of `loggingIdentity` as "the generic function `loggingIdentity` takes a type parameter `Type`, and an argument `arg` which is an array of `Type`s, and returns an array of `Type`s." If we passed in an array of numbers, we'd get an array of numbers back out, as `Type` would bind to `number`. This allows us to use our generic type variable `Type` as part of the types we're working with, rather than the whole type, giving us greater flexibility.

We can alternatively write the sample example this way:

```
function loggingIdentity<Type>(arg: Array<Type>): Array<Type> {
  console.log(arg.length); // Array has a .length, so no more error
  return arg;
}
```

You may already be familiar with this style of type from other languages. In the next section, we'll cover how you can create your own generic types like `Array<Type>`.

Generic Types

In previous sections, we created generic identity functions that worked over a range of types. In this section, we'll explore the type of the functions themselves and how to create generic interfaces.

The type of generic functions is just like those of non-generic functions, with the type parameters listed first, similarly to function declarations:

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

let myIdentity: <Type>(arg: Type) => Type = identity;
```

We could also have used a different name for the generic type parameter in the type, so long as the number of type variables and how the type variables are used line up.

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

let myIdentity: <Input>(arg: Input) => Input = identity;
```

We can also write the generic type as a call signature of an object literal type:

```

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

let myIdentity: { <Type>(arg: Type): Type } = identity;

```

Which leads us to writing our first generic interface. Let's take the object literal from the previous example and move it to an interface:

```

interface GenericIdentityFn {
  <Type>(arg: Type): Type;
}

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

let myIdentity: GenericIdentityFn = identity;

```

In a similar example, we may want to move the generic parameter to be a parameter of the whole interface. This lets us see what type(s) we're generic over (e.g. `Dictionary<string>` rather than just `Dictionary`). This makes the type parameter visible to all the other members of the interface.

```

interface GenericIdentityFn<Type> {
  (arg: Type): Type;
}

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

let myIdentity: GenericIdentityFn<number> = identity;

```

Notice that our example has changed to be something slightly different. Instead of describing a generic function, we now have a non-generic function signature that is a part of a generic type. When we use `GenericIdentityFn`, we now will also need to specify the corresponding type argument (here: `number`), effectively locking in what the underlying call signature will use. Understanding when to put the type parameter directly on the call signature and when to put it on the interface itself will be helpful in describing what aspects of a type are generic.

In addition to generic interfaces, we can also create generic classes. Note that it is not possible to create generic enums and namespaces.

Generic Classes

A generic class has a similar shape to a generic interface. Generic classes have a generic type parameter list in angle brackets (`<>`) following the name of the class.

```

// @strict: false
class GenericNumber<NumType> {
  zeroValue: NumType;
  add: (x: NumType, y: NumType) => NumType;
}

```

```
let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function (x, y) {
    return x + y;
};
```

This is a pretty literal use of the `GenericNumber` class, but you may have noticed that nothing is restricting it to only use the `number` type. We could have instead used `string` or even more complex objects.

```
// @strict: false
class GenericNumber<NumType> {
    zeroValue: NumType;
    add: (x: NumType, y: NumType) => NumType;
}
// ---cut---
let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function (x, y) {
    return x + y;
};
console.log(stringNumeric.add(stringNumeric.zeroValue, "test"));
```

Just as with interface, putting the type parameter on the class itself lets us make sure all of the properties of the class are working with the same type.

As we cover in [our section on classes](#), a class has two sides to its type: the static side and the instance side. Generic classes are only generic over their instance side rather than their static side, so when working with classes, static members can not use the class's type parameter.

Generic Constraints

If you remember from an earlier example, you may sometimes want to write a generic function that works on a set of types where you have *some* knowledge about what capabilities that set of types will have. In our `loggingIdentity` example, we wanted to be able to access the `.length` property of `arg`, but the compiler could not prove that every type had a `.length` property, so it warns us that we can't make this assumption.

```
// @errors: 2339
function loggingIdentity<Type>(arg: Type): Type {
    console.log(arg.length);
    return arg;
}
```

Instead of working with any and all types, we'd like to constrain this function to work with any and all types that *also* have the `.length` property. As long as the type has this member, we'll allow it, but it's required to have at least this member. To do so, we must list our requirement as a constraint on what `Type` can be.

To do so, we'll create an interface that describes our constraint. Here, we'll create an interface that has a single `.length` property and then we'll use this interface and the `extends` keyword to denote our constraint:

```
// @errors: 2345
interface Lengthwise {
  length: number;
}

function loggingIdentity<Type extends Lengthwise>(arg: Type): Type {
  console.log(arg.length); // Now we know it has a .length property, so no more
  error
  return arg;
}
```

Because the generic function is now constrained, it will no longer work over any and all types:

```
// @errors: 2345
interface Lengthwise {
  length: number;
}

function loggingIdentity<Type extends Lengthwise>(arg: Type): Type {
  console.log(arg.length);
  return arg;
}
// ---cut---
loggingIdentity(3);
```

Instead, we need to pass in values whose type has all the required properties:

```
interface Lengthwise {
  length: number;
}

function loggingIdentity<Type extends Lengthwise>(arg: Type): Type {
  console.log(arg.length);
  return arg;
}
// ---cut---
loggingIdentity({ length: 10, value: 3 });
```

Using Type Parameters in Generic Constraints

You can declare a type parameter that is constrained by another type parameter. For example, here we'd like to get a property from an object given its name. We'd like to ensure that we're not accidentally grabbing a property that does not exist on the `obj`, so we'll place a constraint between the two types:

```
// @errors: 2345
function getProperty<Type, Key extends keyof Type>(obj: Type, key: Key) {
  return obj[key];
}

let x = { a: 1, b: 2, c: 3, d: 4 };

getProperty(x, "a");
getProperty(x, "m");
```

Using Class Types in Generics

When creating factories in TypeScript using generics, it is necessary to refer to class types by their constructor functions. For example,

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

A more advanced example uses the prototype property to infer and constrain relationships between the constructor function and the instance side of class types.

```
// @strict: false
class BeeKeeper {
    hasMask: boolean = true;
}

class ZooKeeper {
    nametag: string = "Mikle";
}

class Animal {
    numLegs: number = 4;
}

class Bee extends Animal {
    keeper: BeeKeeper = new BeeKeeper();
}

class Lion extends Animal {
    keeper: ZooKeeper = new ZooKeeper();
}

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

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

This pattern is used to power the `mixins` design pattern.

[Go to TOC](#)

We can use an *indexed access type* to look up a specific property on another type:

```
type Person = { age: number; name: string; alive: boolean };
type Age = Person["age"];
// ^?
```

The indexing type is itself a type, so we can use unions, `keyof`, or other types entirely:

```
type Person = { age: number; name: string; alive: boolean };
// ---cut---
type I1 = Person["age" | "name"];
// ^?

type I2 = Person[keyof Person];
// ^?

type AliveOrName = "alive" | "name";
type I3 = Person[AliveOrName];
// ^?
```

You'll even see an error if you try to index a property that doesn't exist:

```
// @errors: 2339
type Person = { age: number; name: string; alive: boolean };
// ---cut---
type I1 = Person["alve"];
```

Another example of indexing with an arbitrary type is using `number` to get the type of an array's elements. We can combine this with `typeof` to conveniently capture the element type of an array literal:

```
const MyArray = [
  { name: "Alice", age: 15 },
  { name: "Bob", age: 23 },
  { name: "Eve", age: 38 },
];

type Person = typeof MyArray[number];
// ^
type Age = typeof MyArray[number]["age"];
// ^
// Or
type Age2 = Person["age"];
// ^?
```

You can only use types when indexing, meaning you can't use a `const` to make a variable reference:

```
// @errors: 2538 2749
type Person = { age: number; name: string; alive: boolean };
// ---cut---
const key = "age";
type Age = Person[key];
```

However, you can use a type alias for a similar style of refactor:

```
type Person = { age: number; name: string; alive: boolean };
// ---cut---
type key = "age";
type Age = Person[key];
```

[Go to TOC](#)

The `keyof` type operator

The `keyof` operator takes an object type and produces a string or numeric literal union of its keys. The following type P is the same type as "x" | "y":

```
type Point = { x: number; y: number };
type P = keyof Point;
// ^?
```

If the type has a `string` or `number` index signature, `keyof` will return those types instead:

```
type Arrayish = { [n: number]: unknown };
type A = keyof Arrayish;
// ^?

type Mapish = { [k: string]: boolean };
type M = keyof Mapish;
// ^?
```

Note that in this example, `M` is `string` | `number` -- this is because JavaScript object keys are always coerced to a string, so `obj[0]` is always the same as `obj["0"]`.

`keyof` types become especially useful when combined with mapped types, which we'll learn more about later.

[Go to TOC](#)

When you don't want to repeat yourself, sometimes a type needs to be based on another type.

Mapped types build on the syntax for index signatures, which are used to declare the types of properties which have not been declared ahead of time:

```
type Horse = {};
// ---cut---
type OnlyBoolsAndHorses = {
  [key: string]: boolean | Horse;
};

const conforms: OnlyBoolsAndHorses = {
  del: true,
  rodney: false,
};
```

A mapped type is a generic type which uses a union of `PropertyKey`s (frequently created via a `keyof`) to iterate through keys to create a type:

```
type OptionsFlags<Type> = {
  [Property in keyof Type]: boolean;
};
```

In this example, `OptionsFlags` will take all the properties from the type `Type` and change their values to be a boolean.

```
type OptionsFlags<Type> = {
  [Property in keyof Type]: boolean;
};
// ---cut---
type FeatureFlags = {
  darkMode: () => void;
  newUserProfile: () => void;
};

type FeatureOptions = OptionsFlags<FeatureFlags>;
// ^?
```

Mapping Modifiers

There are two additional modifiers which can be applied during mapping: `readonly` and `?` which affect mutability and optionality respectively.

You can remove or add these modifiers by prefixing with `-` or `+`. If you don't add a prefix, then `+` is assumed.

```
// Removes 'readonly' attributes from a type's properties
type CreateMutable<Type> = {
  -readonly [Property in keyof Type]: Type[Property];
};

type LockedAccount = {
  readonly id: string;
  readonly name: string;
};
```

```

type UnlockedAccount = CreateMutable<LockedAccount>;
// ^?

// Removes 'optional' attributes from a type's properties
type Concrete<Type> = {
  [Property in keyof Type]-?: Type[Property];
};

type MaybeUser = {
  id: string;
  name?: string;
  age?: number;
};

type User = Concrete<MaybeUser>;
// ^?

```

Key Remapping via `as`

In TypeScript 4.1 and onwards, you can re-map keys in mapped types with an `as` clause in a mapped type:

```

type MappedTypeWithNewProperties<Type> = {
  [Properties in keyof Type as NewKeyType]: Type[Properties]
}

```

You can leverage features like [template literal types](#) to create new property names from prior ones:

```

type Getters<Type> = {
  [Property in keyof Type as `get${Capitalize<string & Property>}`]: () =>
    Type[Property]
};

interface Person {
  name: string;
  age: number;
  location: string;
}

type LazyPerson = Getters<Person>;
// ^?

```

You can filter out keys by producing `never` via a conditional type:

```

// Remove the 'kind' property
type RemoveKindField<Type> = {
  [Property in keyof Type as Exclude<Property, "kind">]: Type[Property]
};

interface Circle {
  kind: "circle";
  radius: number;
}

type KindlessCircle = RemoveKindField<Circle>;
// ^?

```

You can map over arbitrary unions, not just unions of `string | number | symbol`, but unions of any type:

```

type EventConfig<Events extends { kind: string }> = {
  [E in Events as E["kind"]]: (event: E) => void;
}

type SquareEvent = { kind: "square", x: number, y: number };
type CircleEvent = { kind: "circle", radius: number };

type Config = EventConfig<SquareEvent | CircleEvent>
// ^?

```

Further Exploration

Mapped types work well with other features in this type manipulation section, for example here is a [mapped type using a conditional type](#) which returns either a `true` or `false` depending on whether an object has the property `pii` set to the literal `true`:

```

type ExtractPII<Type> = {
  [Property in keyof Type]: Type[Property] extends { pii: true } ? true : false;
};

type DBFields = {
  id: { format: "incrementing" };
  name: { type: string; pii: true };
};

type ObjectsNeedingGDPRDeletion = ExtractPII<DBFields>;
// ^

```

[Go to TOC](#)

Template literal types build on [string literal types](#), and have the ability to expand into many strings via unions.

They have the same syntax as [template literal strings in JavaScript](#), but are used in type positions. When used with concrete literal types, a template literal produces a new string literal type by concatenating the contents.

```
type World = "world";
type Greeting = `hello ${World}`;
// ^?
```

When a union is used in the interpolated position, the type is the set of every possible string literal that could be represented by each union member:

```
type EmailLocaleIDs = "welcome_email" | "email_heading";
type FooterLocaleIDs = "footer_title" | "footer_sendoff";
type AllLocaleIDs = `${EmailLocaleIDs} | ${FooterLocaleIDs}_id`;
// ^?
```

For each interpolated position in the template literal, the unions are cross multiplied:

```
type EmailLocaleIDs = "welcome_email" | "email_heading";
type FooterLocaleIDs = "footer_title" | "footer_sendoff";
// ---cut---
type AllLocaleIDs = `${EmailLocaleIDs} | ${FooterLocaleIDs}_id`;
type Lang = "en" | "ja" | "pt";
type LocaleMessageIDs = `${Lang}_${AllLocaleIDs}`;
// ^?
```

We generally recommend that people use ahead-of-time generation for large string unions, but this is useful in smaller cases.

String Unions in Types

The power in template literals comes when defining a new string based on information inside a type.

Consider the case where a function (`makeWatchedObject`) adds a new function called `on()` to a passed object. In JavaScript, its call might look like: `makeWatchedObject(baseObject)`. We can imagine the base object as looking like:

```
// @noErrors
const passedObject = {
  firstName: "Saoirse",
  lastName: "Ronan",
  age: 26,
};
```

The `on` function that will be added to the base object expects two arguments, an `eventName` (a `string`) and a `callBack` (a `function`).

The `eventName` should be of the form `attributeInThePassedObject + "Changed"`; thus, `firstNameChanged` as derived from the attribute `firstName` in the base object.

The `callBack` function, when called:

- Should be passed a value of the type associated with the name `attributeInThePassedObject`; thus, since `firstName` is typed as `string`, the callback for the `firstNameChanged` event expects a `string` to be passed to it at call time. Similarly events associated with `age` should expect to be called with a `number` argument
- Should have `void` return type (for simplicity of demonstration)

The naive function signature of `on()` might thus be: `on(eventName: string, callBack: (newValue: any) => void)`. However, in the preceding description, we identified important type constraints that we'd like to document in our code. Template Literal types let us bring these constraints into our code.

```
// @noErrors
declare function makeWatchedObject(obj: any): any;
// ---cut---
const person = makeWatchedObject({
  firstName: "Saoirse",
  lastName: "Ronan",
  age: 26,
});

// makeWatchedObject has added `on` to the anonymous Object

person.on("firstNameChanged", (newValue) => {
  console.log(`firstName was changed to ${newValue}`);
});
```

Notice that `on` listens on the event `"firstNameChanged"`, not just `"firstName"`. Our naive specification of `on()` could be made more robust if we were to ensure that the set of eligible event names was constrained by the union of attribute names in the watched object with `"Changed"` added at the end. While we are comfortable with doing such a calculation in JavaScript i.e. `Object.keys(passedObject).map(x => `${x}Changed`)`, template literals *inside the type system* provide a similar approach to string manipulation:

```
type PropEventSource<Type> = {
  on(eventName: `${string & keyof Type}Changed`, callback: (newValue: any) => void): void;
};

/// Create a "watched object" with an 'on' method
/// so that you can watch for changes to properties.
declare function makeWatchedObject<Type>(obj: Type): Type & PropEventSource<Type>;
```

With this, we can build something that errors when given the wrong property:

```
// @errors: 2345
type PropEventSource<Type> = {
  on(eventName: `${string & keyof Type}Changed`, callback: (newValue: any) => void): void;
};
```

```

declare function makeWatchedObject<T>(obj: T): T & PropEventSource<T>;
// ---cut---
const person = makeWatchedObject({
  firstName: "Saoirse",
  lastName: "Ronan",
  age: 26
});

person.on("firstNameChanged", () => {});

// Prevent easy human error (using the key instead of the event name)
person.on("firstName", () => {});

// It's typo-resistant
person.on("frstNameChanged", () => {});

```

Inference with Template Literals

Notice that we did not benefit from all the information provided in the original passed object. Given change of a `firstName` (i.e. a `firstNameChanged` event), we should expect that the callback will receive an argument of type `string`. Similarly, the callback for a change to `age` should receive a `number` argument. We're naively using `any` to type the `callback`'s argument. Again, template literal types make it possible to ensure an attribute's data type will be the same type as that attribute's callback's first argument.

The key insight that makes this possible is this: we can use a function with a generic such that:

1. The literal used in the first argument is captured as a literal type
2. That literal type can be validated as being in the union of valid attributes in the generic
3. The type of the validated attribute can be looked up in the generic's structure using Indexed Access
4. This typing information can *then* be applied to ensure the argument to the callback function is of the same type

```

type PropEventSource<Type> = {
  on<Key extends string & keyof Type>(
    eventName: `${Key}Changed`, callback: (newValue: Type[Key]) => void ): void;
};

declare function makeWatchedObject<Type>(obj: Type): Type & PropEventSource<Type>;

const person = makeWatchedObject({
  firstName: "Saoirse",
  lastName: "Ronan",
  age: 26
});

person.on("firstNameChanged", newName => {
  // ^?
  console.log(`new name is ${newName.toUpperCase()}`);
});

person.on("ageChanged", newAge => {
  // ^?
  if (newAge < 0) {
    console.warn("warning! negative age");
  }
})

```

Here we made `on` into a generic method.

When a user calls with the string `"firstNameChanged"`, TypeScript will try to infer the right type for `Key`. To do that, it will match `Key` against the content prior to `"Changed"` and infer the string `"firstName"`. Once TypeScript figures that out, the `on` method can fetch the type of `firstName` on the original object, which is `string` in this case. Similarly, when called with `"ageChanged"`, TypeScript finds the type for the property `age` which is `number`.

Inference can be combined in different ways, often to deconstruct strings, and reconstruct them in different ways.

Intrinsic String Manipulation Types

To help with string manipulation, TypeScript includes a set of types which can be used in string manipulation. These types come built-in to the compiler for performance and can't be found in the `.d.ts` files included with TypeScript.

`Uppercase<StringType>`

Converts each character in the string to the uppercase version.

Example

```
type Greeting = "Hello, world"
type ShoutyGreeting = Uppercase<Greeting>
//   ^?

type ASCIICacheKey<Str extends string> = `ID-${Uppercase<Str>}`
type MainID = ASCIICacheKey<"my_app">
//   ^?
```

`Lowercase<StringType>`

Converts each character in the string to the lowercase equivalent.

Example

```
type Greeting = "Hello, world"
type QuietGreeting = Lowercase<Greeting>
//   ^?

type ASCIICacheKey<Str extends string> = `id-${Lowercase<Str>}`
type MainID = ASCIICacheKey<"MY_APP">
//   ^?
```

`Capitalize<StringType>`

Converts the first character in the string to an uppercase equivalent.

Example

```
| type LowercaseGreeting = "hello, world";
| type Greeting = Capitalize<LowercaseGreeting>;
| // ^?
```

Uncapitalize<StringType>

Converts the first character in the string to a lowercase equivalent.

Example

```
| type UppercaseGreeting = "HELLO WORLD";
| type UncomfortableGreeting = Uncapitalize<UppercaseGreeting>;
| // ^?
```

Technical details on the intrinsic string manipulation types

[Go to TOC](#)

The `typeof` type operator

JavaScript already has a `typeof` operator you can use in an *expression* context:

```
// Prints "string"
console.log(typeof "Hello world");
```

TypeScript adds a `typeof` operator you can use in a *type* context to refer to the *type* of a variable or property:

```
let s = "hello";
let n: typeof s;
// ^?
```

This isn't very useful for basic types, but combined with other type operators, you can use `typeof` to conveniently express many patterns. For an example, let's start by looking at the predefined type `ReturnType<T>`. It takes a *function type* and produces its return type:

```
type Predicate = (x: unknown) => boolean;
type K = ReturnType<Predicate>;
// ^?
```

If we try to use `ReturnType` on a function name, we see an instructive error:

```
// @errors: 2749
function f() {
    return { x: 10, y: 3 };
}
type P = ReturnType<f>;
```

Remember that *values* and *types* aren't the same thing. To refer to the *type* that the *value* `f` has, we use `typeof`:

```
function f() {
    return { x: 10, y: 3 };
}
type P = ReturnType<typeof f>;
// ^?
```

Limitations

TypeScript intentionally limits the sorts of expressions you can use `typeof` on.

Specifically, it's only legal to use `typeof` on identifiers (i.e. variable names) or their properties. This helps avoid the confusing trap of writing code you think is executing, but isn't:

```
// @errors: 1005
declare const msgbox: () => boolean;
// type msgbox = any;
// ---cut---
// Meant to use = ReturnType<typeof msgbox>
let shouldContinue: typeof msgbox("Are you sure you want to continue?");
```

TypeScript's type system is very powerful because it allows expressing types *in terms of other types*.

The simplest form of this idea is generics, we actually have a wide variety of *type operators* available to use. It's also possible to express types in terms of *values* that we already have.

By combining various type operators, we can express complex operations and values in a succinct, maintainable way. In this section we'll cover ways to express a new type in terms of an existing type or value.

- [Generics](#) - Types which take parameters
- [Keyof Type Operator](#) - Using the `keyof` operator to create new types
- [Typeof Type Operator](#) - Using the `typeof` operator to create new types
- [Indexed Access Types](#) - Using `Type['a']` syntax to access a subset of a type
- [Conditional Types](#) - Types which act like if statements in the type system
- [Mapped Types](#) - Creating types by mapping each property in an existing type
- [Template Literal Types](#) - Mapped types which change properties via template literal strings

[Go to TOC](#)

Understanding Errors

Whenever TypeScript finds an error, it tries to explain what went wrong in as much detail as possible. Because its type system is structural, this often means providing somewhat lengthy descriptions of where it found a problem.

Terminology

There is some terminology you'll frequently see in error messages that is helpful to understand.

assignable to

TypeScript considers a type *assignable to* another type if one is an acceptable substitute for the other. In other words, a `Cat` is *assignable to* an `Animal` because a `Cat` is an acceptable substitute for an `Animal`.

As its name implies, this relationship is used to check the validity of an assignment `t = s;` by examining the types of `t` and `s`. It's also used to check most other places where two types interact. For example, when calling a function, each argument's type must be *assignable to* parameter's declared type.

Informally, if you see `T is not assignable to S`, you can think of that as TypeScript saying "`T and S` are not compatible". However, note that this is a *directional* relationship: `S` being assignable to `T` does not imply that `T` is assignable to `S`.

Examples

Let's look at some example error messages and understand what's going on.

Error Elaborations

Each error starts with a leading message, sometimes followed by more sub-messages. You can think of each sub-message as answering a "why?" question about the message above it. Let's work through some examples to see how they work in practice.

Here's an example that produces an error message longer than the example itself:

```
// @errors: 2322
let a: { m: number[] };
let b = { m: ["]" };
a = b;
```

TypeScript found an error when checking the last line. Its logic for issuing an error follows from its logic for determining if the assignment is OK:

1. Is `b`'s type assignable to `a`'s? No. Why?
2. Because the type of the `m` property is incompatible. Why?
3. Because `b`'s `m` property (`string[]`) is not assignable to `a`'s `m` property (`number[]`). Why?
4. Because one array's element type (`string`) is not assignable to the other (`number`)

Extra Properties

```
// @errors: 2322
type A = { m: number };
const a: A = { m: 10, n: "" };
```

Union Assignments

```
// @errors: 2322
type Thing = "none" | { name: string };
const a: Thing = { name: 0 };
```

[Go to TOC](#)

With [TypeScript 3.7](#), TypeScript added support for generating .d.ts files from JavaScript using JSDoc syntax.

This set up means you can own the editor experience of TypeScript-powered editors without porting your project to TypeScript, or having to maintain .d.ts files in your codebase. TypeScript supports most JSDoc tags, you can find [the reference here](#).

Setting up your Project to emit .d.ts files

To add creation of .d.ts files in your project, you will need to do up-to four steps:

- Add TypeScript to your dev dependencies
- Add a `tsconfig.json` to configure TypeScript
- Run the TypeScript compiler to generate the corresponding d.ts files for JS files
- (optional) Edit your package.json to reference the types

Adding TypeScript

You can learn how to do this in our [installation page](#).

TSConfig

The TSConfig is a jsonc file which configures both your compiler flags, and declare where to find files. In this case, you will want a file like the following:

```
{
  // Change this to match your project
  "include": ["src/**/*"],

  "compilerOptions": {
    // Tells TypeScript to read JS files, as
    // normally they are ignored as source files
    "allowJs": true,
    // Generate d.ts files
    "declaration": true,
    // This compiler run should
    // only output d.ts files
    "emitDeclarationOnly": true,
    // Types should go into this directory.
    // Removing this would place the .d.ts files
    // next to the .js files
    "outDir": "dist",
    // go to js file when using IDE functions like
    // "Go to Definition" in VSCode
    "declarationMap": true
  }
}
```

You can learn more about the options in the [tsconfig reference](#). An alternative to using a TSConfig file is the CLI, this is the same behavior as a CLI command.

```
npx -p typescript tsc src/**/*.js --declaration --allowJs --emitDeclarationOnly --
outDir types
```

Run the compiler

You can learn how to do this in our [installation page](#). You want to make sure these files are included in your package if you have the files in your project's `.gitignore`.

Editing the package.json

TypeScript replicates the node resolution for modules in a `package.json`, with an additional step for finding .d.ts files. Roughly, the resolution will first check the optional `types` field, then the `"main"` field, and finally will try `index.d.ts` in the root.

Package.json	Location of default .d.ts
No "types" field	checks "main", then index.d.ts
"types": "main.d.ts"	main.d.ts
"types": "./dist/main.js"	./dist/main.d.ts

If absent, then "main" is used

Package.json	Location of default .d.ts
No "main" field	index.d.ts
"main":"index.js"	index.d.ts
"main":"./dist/index.js"	./dist/index.d.ts

Tips

If you'd like to write tests for your .d.ts files, try [tsd](#).

[Go to TOC](#)

The type system in TypeScript has different levels of strictness when working with a codebase:

- A type-system based only on inference with JavaScript code
- Incremental typing in JavaScript via JSDoc
- Using `// @ts-check` in a JavaScript file
- TypeScript code
- TypeScript with `strict` enabled

Each step represents a move towards a safer type-system, but not every project needs that level of verification.

TypeScript with JavaScript

This is when you use an editor which uses TypeScript to provide tooling like auto-complete, jump to symbol and refactoring tools like rename. The [homepage](#) has a list of editors which have TypeScript plugins.

Providing Type Hints in JS via JSDoc

In a `.js` file, types can often be inferred. When types can't be inferred, they can be specified using JSDoc syntax.

JSDoc annotations come before a declaration will be used to set the type of that declaration. For example:

```
/** @type {number} */
var x;

x = 0; // OK
x = false; // OK?!
```

You can find the full list of supported JSDoc patterns [in JSDoc Supported Types](#).

`@ts-check`

The last line of the previous code sample would raise an error in TypeScript, but it doesn't by default in a JS project. To enable errors in your JavaScript files add: `// @ts-check` to the first line in your `.js` files to have TypeScript raise it as an error.

```
// @ts-check
// @errors: 2322
/** @type {number} */
var x;

x = 0; // OK
x = false; // Not OK
```

If you have a lot of JavaScript files you want to add errors to then you can switch to using a `jsconfig.json`. You can skip checking some files by adding a `// @ts-nocheck` comment to files.

TypeScript may offer you errors which you disagree with, in those cases you can ignore errors on specific lines by adding `// @ts-ignore` or `// @ts-expect-error` on the preceding line.

```
// @ts-check
/** @type {number} */
var x;

x = 0; // OK
// @ts-expect-error
x = false; // Not OK
```

To learn more about how JavaScript is interpreted by TypeScript read [How TS Type Checks JS](#)

[Go to TOC](#)

The list below outlines which constructs are currently supported when using JSDoc annotations to provide type information in JavaScript files.

Note any tags which are not explicitly listed below (such as `@async`) are not yet supported.

Types

- `@type`
- `@param` (or `@arg` or `@argument`)
- `@returns` (or `@return`)
- `@typedef`
- `@callback`
- `@template`

Classes

- **Property Modifiers** `@public`, `@private`, `@protected`, `@readonly`
- `@override`
- `@extends` (or `@augments`)
- `@implements`
- `@class` (or `@constructor`)
- `@this`

Documentation

Documentation tags work in both TypeScript and JavaScript.

- `@deprecated`
- `@see`
- `@link`

Other

- `@enum`
- `@author`
- [Other supported patterns](#)
- [Unsupported patterns](#)
- [Unsupported tags](#)

The meaning is usually the same, or a superset, of the meaning of the tag given at [jsdoc.app](#). The code below describes the differences and gives some example usage of each tag.

Note: You can use [the playground to explore JSDoc support](#).

Types

@type

You can reference types with the "@type" tag. The type can be:

1. Primitive, like `string` or `number`.
2. Declared in a TypeScript declaration, either global or imported.
3. Declared in a JSDoc `@typedef` tag.

You can use most JSDoc type syntax and any TypeScript syntax, from the most basic like `string` to the most advanced, like conditional types.

```
/*
 * @type {string}
 */
var s;

/** @type {Window} */
var win;

/** @type {PromiseLike<string>} */
var promisedString;

// You can specify an HTML Element with DOM properties
/** @type {HTMLElement} */
var myElement = document.querySelector(selector);
element.dataset.myData = "";
```

`@type` can specify a union type — for example, something can be either a string or a boolean.

```
/*
 * @type {string | boolean}
 */
var sb;
```

You can specify array types using a variety of syntaxes:

```
/** @type {number[]} */
var ns;
/** @type {Array.<number>} */
var jsdoc;
/** @type {Array<number>} */
var nas;
```

You can also specify object literal types. For example, an object with properties 'a' (string) and 'b' (number) uses the following syntax:

```
/** @type {{ a: string, b: number }} */
var var9;
```

You can specify map-like and array-like objects using string and number index signatures, using either standard JSDoc syntax or TypeScript syntax.

```
/**  
 * A map-like object that maps arbitrary `string` properties to `number`s.  
 */  
var stringToNumber;  
  
/** @type {Object.<string, number>} */  
var arrayLike;
```

The preceding two types are equivalent to the TypeScript types `{ [x: string]: number }` and `{ [x: number]: any }`. The compiler understands both syntaxes.

You can specify function types using either TypeScript or Google Closure syntax:

```
/** @type {function(string, boolean): number} Closure syntax */  
var sbn;  
/** @type {(s: string, b: boolean) => number} TypeScript syntax */  
var sbn2;
```

Or you can just use the unspecified `Function` type:

```
/** @type {Function} */  
var fn7;  
/** @type {function} */  
var fn6;
```

Other types from Closure also work:

```
/**  
 * @type {*} - can be 'any' type  
 */  
var star;  
/**  
 * @type {?} - unknown type (same as 'any')  
 */  
var question;
```

Casts

TypeScript borrows cast syntax from Google Closure. This lets you cast types to other types by adding a `@type` tag before any parenthesized expression.

```
/**  
 * @type {number | string}  
 */  
var numberOrString = Math.random() < 0.5 ? "hello" : 100;  
var typeAssertedNumber = /** @type {number} */ (numberOrString);
```

You can even cast to `const` just like TypeScript:

```
let one = /** @type {const} */(1);
```

Import types

You can import declarations from other files using import types. This syntax is TypeScript-specific and differs from the JSDoc standard:

```
// @filename: types.d.ts
export type Pet = {
  name: string,
};

// @filename: main.js
/**
 * @param { import("./types").Pet } p
 */
function walk(p) {
  console.log(`Walking ${p.name}...`);
}
```

import types can be used in type alias declarations:

```
// @filename: types.d.ts
export type Pet = {
  name: string,
};

// @filename: main.js
// ---cut---
/**
 * @typedef { import("./types").Pet } Pet

/**
 * @type {Pet}
 */
var myPet;
myPet.name;
```

import types can be used to get the type of a value from a module if you don't know the type, or if it has a large type that is annoying to type:

```
// @filename: accounts.d.ts
export const userAccount = {
  name: "Name",
  address: "An address",
  postalCode: "",
  country: "",
  planet: "",
  system: "",
  galaxy: "",
  universe: "",
};

// @filename: main.js
// ---cut---
/**
 * @type {typeof import("./accounts").userAccount }
 */
var x = require("./accounts").userAccount;
```

@param and @returns

`@param` uses the same type syntax as `@type`, but adds a parameter name. The parameter may also be declared optional by surrounding the name with square brackets:

```
// Parameters may be declared in a variety of syntactic forms
/**
 * @param {string} p1 - A string param.
 * @param {string=} p2 - An optional param (Google Closure syntax).
 * @param {string} [p3] - Another optional param (JSDoc syntax).
 * @param {string} [p4="test"] - An optional param with a default value.
 * @returns {string} This is the result
*/
function stringsStringStrings(p1, p2, p3, p4) {
// TODO
}
```

Likewise, for the return type of a function:

```
 /**
 * @return {PromiseLike<string>}
 */
function ps() {}

 /**
 * @returns {{ a: string, b: number }} - May use '@returns' as well as '@return'
 */
function ab() {}
```

@typedef, @callback, and @param

You can define complex types with `@typedef`. Similar syntax works with `@param`.

```
 /**
 * @typedef {Object} SpecialType - creates a new type named 'SpecialType'
 * @property {string} prop1 - a string property of SpecialType
 * @property {number} prop2 - a number property of SpecialType
 * @property {number=} prop3 - an optional number property of SpecialType
 * @prop {number} [prop4] - an optional number property of SpecialType
 * @prop {number=} [prop5=42] - an optional number property of SpecialType with
default
 */

/** @type {SpecialType} */
var specialTypeObject,
specialTypeObject.prop3;
```

You can use either `object` or `Object` on the first line.

```
 /**
 * @typedef {object} SpecialType1 - creates a new type named 'SpecialType'
 * @property {string} prop1 - a string property of SpecialType
 * @property {number} prop2 - a number property of SpecialType
 * @property {number=} prop3 - an optional number property of SpecialType
 */
```

```
/** @type {SpecialType1} */
var specialTypeObject1;
```

`@param` allows a similar syntax for one-off type specifications. Note that the nested property names must be prefixed with the name of the parameter:

```
/*
 * @param {Object} options - The shape is the same as SpecialType above
 * @param {string} options.prop1
 * @param {number} options.prop2
 * @param {number=} options.prop3
 * @param {number} [options.prop4]
 * @param {number} [options.prop5=42]
*/
function special(options) {
  return (options.prop4 || 1001) + options.prop5;
}
```

`@callback` is similar to `@typedef`, but it specifies a function type instead of an object type:

```
/*
 * @callback Predicate
 * @param {string} data
 * @param {number} [index]
 * @returns {boolean}
*/
/** @type {Predicate} */
const ok = (s) => !(s.length % 2);
```

Of course, any of these types can be declared using TypeScript syntax in a single-line `@typedef`:

```
/** @typedef {{ prop1: string, prop2: string, prop3?: number }} SpecialType */
/** @typedef {(data: string, index?: number) => boolean} Predicate */
```

`@template`

You can declare type parameters with the `@template` tag. This lets you make functions, classes, or types that are generic:

```
/*
 * @template T
 * @param {T} x - A generic parameter that flows through to the return type
 * @returns {T}
*/
function id(x) {
  return x;
}

const a = id("string");
const b = id(123);
const c = id({});
```

Use comma or multiple tags to declare multiple type parameters:

```
/**  
 * @template T,U,V  
 * @template W,X  
 */
```

You can also specify a type constraint before the type parameter name. Only the first type parameter in a list is constrained:

```
/**  
 * @template {string} K - K must be a string or string literal  
 * @template {{ serious(): string }} Seriousalizable - must have a serious method  
 * @param {K} key  
 * @param {Seriousalizable} object  
 */  
function serialize(key, object) {  
 // ????
```

Finally, you can specify a default for a type parameter:

```
/** @template [T=object] */  
class Cache {  
 /** @param {T} initial */  
 constructor(T) {  
 }  
}  
let c = new Cache()
```

Classes

Classes can be declared as ES6 classes.

```
class C {  
/**  
 * @param {number} data  
 */  
constructor(data) {  
 // property types can be inferred  
 this.name = "foo";  
  
 // or set explicitly  
 /** @type {string | null} */  
 this.title = null;  
  
 // or simply annotated, if they're set elsewhere  
 /** @type {number} */  
 this.size;  
  
 this.initialize(data); // Should error, initializer expects a string  
}  
/**  
 * @param {string} s  
 */  
initialize = function (s) {  
 this.size = s.length;  
};  
}
```

```
// var c = new C();
// C should only be called with new, but
// because it is JavaScript, this is allowed and
// considered an 'any'.
var result = C(1);
```

They can also be declared as constructor functions; use `@constructor` along with `@this` for this.

Property Modifiers

`@public`, `@private`, and `@protected` work exactly like `public`, `private`, and `protected` in TypeScript:

```
// @errors: 2341
// @ts-check

class Car {
  constructor() {
    /** @private */
    this.identifier = 100;
  }

  printIdentifier() {
    console.log(this.identifier);
  }
}

const c = new Car();
console.log(c.identifier);
```

- `@public` is always implied and can be left off, but means that a property can be reached from anywhere.
- `@private` means that a property can only be used within the containing class.
- `@protected` means that a property can only be used within the containing class, and all derived subclasses, but not on dissimilar instances of the containing class.

`@public`, `@private`, and `@protected` do not work in constructor functions.

`@readonly`

The `@readonly` modifier ensures that a property is only ever written to during initialization.

```
// @errors: 2540
// @ts-check

class Car {
  constructor() {
    /** @readonly */
    this.identifier = 100;
  }

  printIdentifier() {
    console.log(this.identifier);
  }
}
```

```
const c = new Car();
console.log(c.identifier);
```

@override

`@override` works the same way as in TypeScript; use it on methods that override a method from a base class:

```
export class C {
  m() { }
}
class D extends C {
  /** @override */
  m() { }
}
```

Set `noImplicitOverride: true` in `tsconfig` to check overrides.

@extends

When JavaScript classes extend a generic base class, there is no JavaScript syntax for passing a type argument. The `@extends` tag allows this:

```
/** 
 * @template T
 * @extends {Set<T>}
 */
class SortableSet extends Set {
  // ...
}
```

Note that `@extends` only works with classes. Currently, there is no way for a constructor function to extend a class.

@implements

In the same way, there is no JavaScript syntax for implementing a TypeScript interface. The `@implements` tag works just like in TypeScript:

```
/** @implements {Print} */
class TextBook {
  print() {
    // TODO
  }
}
```

@constructor

The compiler infers constructor functions based on `this`-property assignments, but you can make checking stricter and suggestions better if you add a `@constructor` tag:

```
// @checkJs
// @errors: 2345 2348
/**
```

```

* @constructor
* @param {number} data
*/
function C(data) {
  // property types can be inferred
  this.name = "foo";

  // or set explicitly
  /** @type {string | null} */
  this.title = null;

  // or simply annotated, if they're set elsewhere
  /** @type {number} */
  this.size;

  this.initialize(data);
}
/**
 * @param {string} s
*/
C.prototype.initialize = function (s) {
  this.size = s.length;
};

var c = new C(0);
c.size;

var result = C(1);

```

Note: Error messages only show up in JS codebases with [a JSConfig](#) and `checkJS` enabled.

With `@constructor`, `this` is checked inside the constructor function `c`, so you will get suggestions for the `initialize` method and an error if you pass it a number. Your editor may also show warnings if you call `C` instead of constructing it.

Unfortunately, this means that constructor functions that are also callable cannot use `@constructor`.

`@this`

The compiler can usually figure out the type of `this` when it has some context to work with. When it doesn't, you can explicitly specify the type of `this` with `@this`:

```

/**
 * @this {HTMLElement}
 * @param {*} e
*/
function callbackForLater(e) {
  this.clientHeight = parseInt(e); // should be fine!
}

```

Documentation

`@deprecated`

When a function, method, or property is deprecated you can let users know by marking it with a `/** @deprecated */` JSDoc comment. That information is surfaced in completion lists and as a suggestion diagnostic that editors can handle specially. In an editor like VS Code, deprecated values are typically displayed in a strike-through style like this.

```
// @noErrors
/** @deprecated */
const apiV1 = {};
const apiV2 = {};

apiV;
// ^|
```

@see

`@see` lets you link to other names in your program:

```
type Box<T> = { t: T }
/** @see Box for implementation details */
type Boxify<T> = { [K in keyof T]: Box<T> };
```

Some editors will turn `Box` into a link to make it easy to jump there and back.

@link

`@link` is like `@see`, except that it can be used inside other tags:

```
type Box<T> = { t: T }
/** @returns A {@link Box} containing the parameter. */
function box<U>(u: U): Box<U> {
    return { t: u };
}
```

Other

@enum

The `@enum` tag allows you to create an object literal whose members are all of a specified type. Unlike most object literals in JavaScript, it does not allow other members. `@enum` is intended for compatibility with Google Closure's `@enum` tag.

```
/** @enum {number} */
const JSDocState = {
    BeginningOfLine: 0,
    SawAsterisk: 1,
    SavingComments: 2,
};

JSDocState.SawAsterisk;
```

Note that `@enum` is quite different from, and much simpler than, TypeScript's `enum`. However, unlike TypeScript's enums, `@enum` can have any type:

```
/** @enum {function(number): number} */
const MathFuncs = {
  add1: (n) => n + 1,
  id: (n) => -n,
  sub1: (n) => n - 1,
};

MathFuncs.add1;
```

@author

You can specify the author of an item with `@author`:

```
/** 
 * Welcome to awesome.ts
 * @author Ian Awesome <i.am.awesome@example.com>
 */
```

Remember to surround the email address with angle brackets. Otherwise, `@example` will be parsed as a new tag.

Other supported patterns

```
class Foo {}
// ---cut---
var someObj = {
  /**
   * @param {string} param1 - JSDocs on property assignments work
   */
  x: function (param1) {},
};

/**
 * As do jsdocs on variable assignments
 * @return {Window}
 */
let someFunc = function () {};

/**
 * And class methods
 * @param {string} greeting The greeting to use
 */
Foo.prototype.sayHi = (greeting) => console.log("Hi!");

/**
 * And arrow function expressions
 * @param {number} x - A multiplier
 */
let myArrow = (x) => x * x;

/**
 * Which means it works for function components in JSX too
 * @param {{a: string, b: number}} props - Some param
 */
var fc = (props) => <div>{props.a.charAt(0)}</div>;

/**
 * A parameter can be a class constructor, using Google Closure syntax.
 *
```

```
/* @param {{new(...args: any[]): object}} C - The class to register
*/
function registerClass(C) {}

/** 
 * @param {...string} p1 - A 'rest' arg (array) of strings. (treated as 'any')
*/
function fn10(p1) {}

/** 
 * @param {...string} p1 - A 'rest' arg (array) of strings. (treated as 'any')
*/
function fn9(p1) {
    return p1.join();
}
```

Unsupported patterns

Postfix equals on a property type in an object literal type doesn't specify an optional property:

```
/** 
 * @type {{ a: string, b: number= }}
*/
var wrong;
/** 
 * Use postfix question on the property name instead:
 * @type {{ a: string, b?: number }}
*/
var right;
```

Nullable types only have meaning if `strictNullChecks` is on:

```
/** 
 * @type {?number}
 * With strictNullChecks: true -- number | null
 * With strictNullChecks: false -- number
*/
var nullable;
```

The TypeScript-native syntax is a union type:

```
/** 
 * @type {number | null}
 * With strictNullChecks: true -- number | null
 * With strictNullChecks: false -- number
*/
var unionNullable;
```

Non-nullable types have no meaning and are treated just as their original type:

```
/** 
 * @type {!number}
 * Just has type number
*/
var normal;
```

Unlike JSDoc's type system, TypeScript only allows you to mark types as containing null or not. There is no explicit non-nullability -- if strictNullChecks is on, then `number` is not nullable. If it is off, then `number` is nullable.

Unsupported tags

TypeScript ignores any unsupported JSDoc tags.

The following tags have open issues to support them:

- `@const` ([issue #19672](#))
- `@inheritDoc` ([issue #23215](#))
- `@memberof` ([issue #7237](#))
- `@yields` ([issue #23857](#))

[Go to TOC](#)

Here are some notable differences on how checking works in `.js` files compared to `.ts` files.

Properties are inferred from assignments in class bodies

ES2015 does not have a means for declaring properties on classes. Properties are dynamically assigned, just like object literals.

In a `.js` file, the compiler infers properties from property assignments inside the class body. The type of a property is the type given in the constructor, unless it's not defined there, or the type in the constructor is undefined or null. In that case, the type is the union of the types of all the right-hand values in these assignments. Properties defined in the constructor are always assumed to exist, whereas ones defined just in methods, getters, or setters are considered optional.

```
// @checkJs
// @errors: 2322
class C {
  constructor() {
    this.constructorOnly = 0;
    this.constructorUnknown = undefined;
  }
  method() {
    this.constructorOnly = false;
    this.constructorUnknown = "plunkbat"; // ok, constructorUnknown is string | undefined
    this.methodOnly = "ok"; // ok, but methodOnly could also be undefined
  }
  method2() {
    this.methodOnly = true; // also, ok, methodOnly's type is string | boolean | undefined
  }
}
```

If properties are never set in the class body, they are considered unknown. If your class has properties that are only read from, add and then annotate a declaration in the constructor with JSDoc to specify the type. You don't even have to give a value if it will be initialized later:

```
// @checkJs
// @errors: 2322
class C {
  constructor() {
    /**
     * @type {number | undefined}
     */
    this.prop = undefined;
    /**
     * @type {number | undefined}
     */
    this.count;
  }
}

let c = new C();
c.prop = 0; // OK
c.count = "string";
```

Constructor functions are equivalent to classes

Before ES2015, JavaScript used constructor functions instead of classes. The compiler supports this pattern and understands constructor functions as equivalent to ES2015 classes. The property inference rules described above work exactly the same way.

```
// @checkJs
// @errors: 2683 2322
function C() {
  this.constructorOnly = 0;
  this.constructorUnknown = undefined;
}
C.prototype.method = function () {
  this.constructorOnly = false;
  this.constructorUnknown = "plunkbat"; // OK, the type is string | undefined
};
```

CommonJS modules are supported

In a `.js` file, TypeScript understands the CommonJS module format. Assignments to `exports` and `module.exports` are recognized as export declarations. Similarly, `require` function calls are recognized as module imports. For example:

```
// same as `import module "fs"`
const fs = require("fs");

// same as `export function readFile`
module.exports.readFile = function (f) {
  return fs.readFileSync(f);
};
```

The module support in JavaScript is much more syntactically forgiving than TypeScript's module support. Most combinations of assignments and declarations are supported.

Classes, functions, and object literals are namespaces

Classes are namespaces in `.js` files. This can be used to nest classes, for example:

```
class C {}
C.D = class {};
```

And, for pre-ES2015 code, it can be used to simulate static methods:

```
function Outer() {
  this.y = 2;
}

Outer.Inner = function () {
  this.yy = 2;
};

Outer.Inner();
```

It can also be used to create simple namespaces:

```
var ns = {};
ns.C = class {};
ns.func = function () {};
ns;
```

Other variants are allowed as well:

```
// IIFE
var ns = (function (n) {
    return n || {};
}());
ns.CONST = 1;

// defaulting to global
var assign =
    assign || 
    function () {
        // code goes here
    };
assign.extra = 1;
```

Object literals are open-ended

In a `.ts` file, an object literal that initializes a variable declaration gives its type to the declaration. No new members can be added that were not specified in the original literal. This rule is relaxed in a `.js` file; object literals have an open-ended type (an index signature) that allows adding and looking up properties that were not defined originally. For instance:

```
var obj = { a: 1 };
obj.b = 2; // Allowed
```

Object literals behave as if they have an index signature `[x:string]: any` that allows them to be treated as open maps instead of closed objects.

Like other special JS checking behaviors, this behavior can be changed by specifying a JSDoc type for the variable. For example:

```
// @checkJs
// @errors: 2339
/** @type {{a: number}} */
var obj = { a: 1 };
obj.b = 2;
```

null, undefined, and empty array initializers are of type any or any[]

Any variable, parameter or property that is initialized with null or undefined will have type any, even if strict null checks is turned on. Any variable, parameter or property that is initialized with [] will have type any[], even if strict null checks is turned on. The only exception is for properties that have multiple initializers as described above.

```
function Foo(i = null) {
  if (!i) i = 1;
  var j = undefined;
  j = 2;
  this.l = [];
}

var foo = new Foo();
foo.l.push(foo.i);
foo.l.push("end");
```

Function parameters are optional by default

Since there is no way to specify optionality on parameters in pre-ES2015 JavaScript, all function parameters in .js file are considered optional. Calls with fewer arguments than the declared number of parameters are allowed.

It is important to note that it is an error to call a function with too many arguments.

For instance:

```
// @checkJs
// @strict: false
// @errors: 7006 7006 2554
function bar(a, b) {
  console.log(a + " " + b);
}

bar(1); // OK, second argument considered optional
bar(1, 2);
bar(1, 2, 3); // Error, too many arguments
```

JSDoc annotated functions are excluded from this rule. Use JSDoc optional parameter syntax ([]) to express optionality. e.g.:

```
/** 
 * @param {string} [somebody] - Somebody's name.
 */
function sayHello(somebody) {
  if (!somebody) {
    somebody = "John Doe";
  }
  console.log("Hello " + somebody);
}

sayHello();
```

Var-args parameter declaration inferred from use of arguments

A function whose body has a reference to the `arguments` reference is implicitly considered to have a var-arg parameter (i.e. `(...arg: any[]) => any`). Use JSDoc var-arg syntax to specify the type of the arguments.

```
/** @param {...number} args */
function sum(/* numbers */){
  var total = 0;
  for (var i = 0; i < arguments.length; i++) {
    total += arguments[i];
  }
  return total;
}
```

Unspecified type parameters default to `any`

Since there is no natural syntax for specifying generic type parameters in JavaScript, an unspecified type parameter defaults to `any`.

In extends clause

For instance, `React.Component` is defined to have two type parameters, `Props` and `State`. In a `.js` file, there is no legal way to specify these in the extends clause. By default the type arguments will be `any`:

```
import { Component } from "react";

class MyComponent extends Component {
  render() {
    this.props.b; // Allowed, since this.props is of type any
  }
}
```

Use JSDoc `@augments` to specify the types explicitly. for instance:

```
import { Component } from "react";

/**
 * @augments {Component<{a: number}, State>}
 */
class MyComponent extends Component {
  render() {
    this.props.b; // Error: b does not exist on {a:number}
  }
}
```

In JSDoc references

An unspecified type argument in JSDoc defaults to any:

```
/** @type{Array} */
var x = [];

x.push(1); // OK
x.push("string"); // OK, x is of type Array<any>

/** @type{Array.<number>} */
var y = [];

y.push(1); // OK
y.push("string"); // Error, string is not assignable to number
```

In function calls

A call to a generic function uses the arguments to infer the type parameters. Sometimes this process fails to infer any types, mainly because of lack of inference sources; in these cases, the type parameters will default to `any`. For example:

```
var p = new Promise((resolve, reject) => {
  reject();
});

p; // Promise<any>;
```

To learn all of the features available in JSDoc, see [the reference](#).

[Go to TOC](#)

Using the CLI

Running `tsc` locally will compile the closest project defined by a `tsconfig.json`, you can compile a set of TypeScript files by passing in a glob of files you want.

```
# Run a compile based on a backwards look through the fs for a tsconfig.json
tsc

# Emit JS for just the index.ts with the compiler defaults
tsc index.ts

# Emit JS for any .ts files in the folder src, with the default settings
tsc src/*.ts

# Emit files referenced in with the compiler settings from
tsconfig.production.json
tsc --project tsconfig.production.json

# Emit d.ts files for a js file with showing compiler options which are booleans
tsc index.js --declaration --emitDeclarationOnly

# Emit a single .js file from two files via compiler options which take string
arguments
tsc app.ts util.ts --target esnext --outfile index.js
```

Compiler Options

If you're looking for more information about the compiler options in a `tsconfig`, check out the [TSCConfig Reference](#)

CLI Commands

Flag	Type
<code>--all</code>	<code>boolean</code>
Show all compiler options.	
<code>--generateTrace</code>	<code>string</code>
Generates an event trace and a list of types.	
<code>--help</code>	<code>boolean</code>
Gives local information for help on the CLI.	

Flag	Type
--init	boolean
Initializes a TypeScript project and creates a tsconfig.json file.	
--listFilesOnly	boolean
Print names of files that are part of the compilation and then stop processing.	
--locale	string
Set the language of the messaging from TypeScript. This does not affect emit.	
--project	string
Compile the project given the path to its configuration file, or to a folder with a 'tsconfig.json'.	
--showConfig	boolean
Print the final configuration instead of building.	
--version	boolean
Print the compiler's version.	

Build Options

Flag	Type
--build	boolean
Build one or more projects and their dependencies, if out of date	
--clean	boolean

Flag	Type
Delete the outputs of all projects.	
--dry	boolean
Show what would be built (or deleted, if specified with '--clean')	
--force	boolean
Build all projects, including those that appear to be up to date.	
--verbose	boolean
Enable verbose logging.	

Watch Options

Flag	Type
-- excludeDirectories	list
Remove a list of directories from the watch process.	
--excludeFiles	list
Remove a list of files from the watch mode's processing.	
--fallbackPolling	fixedinterval, priorityinterval, dynamicpriority, or fixed-chunksize
Specify what approach the watcher should use if the system runs out of native file watchers.	
-- synchronousWatchDi- rectory	boolean

Flag	Type
Synchronously call callbacks and update the state of directory watchers on platforms that don't support recursive watching natively.	
--watch	boolean
Watch input files.	
--watchDirectory	usefsevents, fixedpollinginterval, dynamicprioritypolling, or fixedchunksizepolling
Specify how directories are watched on systems that lack recursive file-watching functionality.	
--watchFile	fixedpollinginterval, prioritypollinginterval, dynamicprioritypolling, fixedchunksizepolling, usefsevents, or usefsevents on parent directory
Specify how the TypeScript watch mode works.	

Compiler Flags

Flag	Type	Default
--allowJs	boolean	false
Allow JavaScript files to be a part of your program. Use the checkJS option to get errors from these files.		
--allowSyntheticDefaultImports	boolean	<ul style="list-style-type: none"> • true if module is system, or esModuleInterop and module is not es6 / es2015 or esnext, • false otherwise.
Allow 'import x from y' when a module doesn't have a default export.		

Flag	Type	Default
<code>--allowUmdGlobalAccess</code>	boolean	false
Allow accessing UMD globals from modules.		
<code>--allowUnreachableCode</code>	boolean	
Disable error reporting for unreachable code.		
<code>--allowUnusedLabels</code>	boolean	
Disable error reporting for unused labels.		
<code>--alwaysStrict</code>	boolean	<ul style="list-style-type: none"> • true if strict, • false otherwise.
Ensure 'use strict' is always emitted.		
<code>--assumeChangesOnlyAffectDirectDependencies</code>	boolean	false
Have recompiles in projects that use <code>incremental</code> and <code>watch</code> mode assume that changes within a file will only affect files directly depending on it.		
<code>--baseUrl</code>	string	
Specify the base directory to resolve non-relative module names.		
<code>--charset</code>	string	utf8
No longer supported. In early versions, manually set the text encoding for reading files.		
<code>--checkJs</code>	boolean	false

Flag	Type	Default
Enable error reporting in type-checked JavaScript files.		
<code>--composite</code>	boolean	false
Enable constraints that allow a TypeScript project to be used with project references.		
<code>--declaration</code>	boolean	<ul style="list-style-type: none"> • true if composite, • false otherwise.
Generate .d.ts files from TypeScript and JavaScript files in your project.		
<code>--declarationDir</code>	string	
Specify the output directory for generated declaration files.		
<code>--declarationMap</code>	boolean	false
Create sourcemaps for d.ts files.		
<code>--diagnostics</code>	boolean	false
Output compiler performance information after building.		
<code>--disableReferencedProjectLoad</code>	boolean	false
Reduce the number of projects loaded automatically by TypeScript.		
<code>--disableSizeLimit</code>	boolean	false
Remove the 20mb cap on total source code size for JavaScript files in the TypeScript language server.		

Flag	Type	Default
<code>--disableSolutionSearching</code>	boolean	false
Opt a project out of multi-project reference checking when editing.		
<code>-- disableSourceOfProjectReferenceRedirect</code>	boolean	false
Disable preferring source files instead of declaration files when referencing composite projects.		
<code>--downlevelIteration</code>	boolean	false
Emit more compliant, but verbose and less performant JavaScript for iteration.		
<code>--emitBOM</code>	boolean	false
Emit a UTF-8 Byte Order Mark (BOM) in the beginning of output files.		
<code>--emitDeclarationOnly</code>	boolean	false
Only output d.ts files and not JavaScript files.		
<code>--emitDecoratorMetadata</code>	boolean	false
Emit design-type metadata for decorated declarations in source files.		
<code>--esModuleInterop</code>	boolean	false
Emit additional JavaScript to ease support for importing CommonJS modules. This enables <code>allowSyntheticDefaultImports</code> for type compatibility.		
<code>--exactOptionalPropertyTypes</code>	boolean	false

Flag	Type	Default
Interpret optional property types as written, rather than adding <code>undefined</code> .		
<code>--experimentalDecorators</code>	boolean	false
Enable experimental support for TC39 stage 2 draft decorators.		
<code>--explainFiles</code>	boolean	false
Print files read during the compilation including why it was included.		
<code>--extendedDiagnostics</code>	boolean	false
Output more detailed compiler performance information after building.		
<code>--forceConsistentCasingInFileNames</code>	boolean	false
Ensure that casing is correct in imports.		
<code>--generateCpuProfile</code>	string	profile.cpuprofile
Emit a v8 CPU profile of the compiler run for debugging.		
<code>--importHelpers</code>	boolean	false
Allow importing helper functions from tslib once per project, instead of including them per-file.		
<code>--importsNotUsedAsValues</code>	remove, pre-serve, or error	remove
Specify emit/checking behavior for imports that are only used for types.		

Flag	Type	Default
<code>--incremental</code>	boolean	<ul style="list-style-type: none"> • <code>true</code> if <code>composite</code>, • <code>false</code> otherwise.
Save .tsbuildinfo files to allow for incremental compilation of projects.		
<code>--inlineSourceMap</code>	boolean	<code>false</code>
Include sourcemap files inside the emitted JavaScript.		
<code>--inlineSources</code>	boolean	<code>false</code>
Include source code in the sourcemaps inside the emitted JavaScript.		
<code>--isolatedModules</code>	boolean	<code>false</code>
Ensure that each file can be safely transpiled without relying on other imports.		
<code>--jsx</code>		<pre>preserve, re- act, react- native, react-jsx, or react-jsxdev</pre>
Specify what JSX code is generated.		
<code>--jsxFactory</code>	string	<code>React.createElement</code>
Specify the JSX factory function used when targeting React JSX emit, e.g. 'React.createElement' or 'h'.		
<code>--jsxFragmentFactory</code>	string	<code>React.Fragment</code>

Flag	Type	Default
Specify the JSX Fragment reference used for fragments when targeting React JSX emit e.g. 'React.Fragment' or 'Fragment'.		
<code>--jsxImportSource</code>	string	react
Specify module specifier used to import the JSX factory functions when using <code>jsx: react-jsx*</code> .		
<code>--keyofStringsOnly</code>	boolean	false
Make keyof only return strings instead of string, numbers or symbols. Legacy option.		
<code>--lib</code>	list	
Specify a set of bundled library declaration files that describe the target runtime environment.		
<code>--listEmittedFiles</code>	boolean	false
Print the names of emitted files after a compilation.		
<code>--listFiles</code>	boolean	false
Print all of the files read during the compilation.		
<code>--mapRoot</code>	string	
Specify the location where debugger should locate map files instead of generated locations.		
<code>--maxNodeModuleJsDepth</code>	number	0

Flag	Type	Default
		Specify the maximum folder depth used for checking JavaScript files from <code>node_modules</code> . Only applicable with <code>allowJs</code> .
<code>--module</code>	<code>none</code> , <code>commonjs</code> , <code>amd</code> , <code>umd</code> , <code>system</code> , <code>es6 / es2015</code> , <code>es2020</code> , <code>es2022</code> , <code>esnext</code> , <code>node16</code> , or <code>nodenext</code>	<ul style="list-style-type: none"> • <code>CommonJS</code> if <code>target</code> is <code>ES3</code> or <code>ES5</code>, • <code>ES6 / ES2015</code> otherwise.
		Specify what module code is generated.
<code>--moduleDetection</code>	<code>legacy</code> , <code>auto</code> , or <code>force</code>	"auto": Treat files with imports, exports, import.meta, jsx (with jsx: react-jsx), or esm format (with module: node16+) as modules.
		Control what method is used to detect whether a JS file is a module.
<code>--moduleResolution</code>	<code>classic</code> , <code>node</code> , <code>node16</code> , or <code>nodenext</code>	<ul style="list-style-type: none"> • <code>Classic</code> if <code>module</code> is <code>AMD</code>, <code>UMD</code>, <code>System</code> or <code>ES6 / ES2015</code>, • Matches if <code>module</code> is <code>node12</code> or <code>nodenext</code>, • <code>Node</code> otherwise.
		Specify how TypeScript looks up a file from a given module specifier.
<code>--moduleSuffixes</code>	<code>list</code>	List of file name suffixes to search when resolving a module.

Flag	Type	Default
<code>--newLine</code>	<code>crlf</code> or <code>lf</code>	Platform specific.
Set the newline character for emitting files.		
<code>--noEmit</code>	<code>boolean</code>	<code>false</code>
Disable emitting files from a compilation.		
<code>--noEmitHelpers</code>	<code>boolean</code>	<code>false</code>
Disable generating custom helper functions like <code>__extends</code> in compiled output.		
<code>--noEmitOnError</code>	<code>boolean</code>	<code>false</code>
Disable emitting files if any type checking errors are reported.		
<code>--noErrorTruncation</code>	<code>boolean</code>	<code>false</code>
Disable truncating types in error messages.		
<code>--noFallthroughCasesInSwitch</code>	<code>boolean</code>	<code>false</code>
Enable error reporting for fallthrough cases in switch statements.		
<code>--noImplicitAny</code>	<code>boolean</code>	<ul style="list-style-type: none"> • <code>true</code> if <code>strict</code>, • <code>false</code> otherwise.
Enable error reporting for expressions and declarations with an implied <code>any</code> type.		
<code>--noImplicitOverride</code>	<code>boolean</code>	<code>false</code>

Flag	Type	Default
Ensure overriding members in derived classes are marked with an override modifier.		
<code>--noImplicitReturns</code>	boolean	false
Enable error reporting for codepaths that do not explicitly return in a function.		
<code>--noImplicitThis</code>	boolean	<ul style="list-style-type: none"> • true if strict, • false otherwise.
Enable error reporting when <code>this</code> is given the type <code>any</code> .		
<code>--noImplicitUseStrict</code>	boolean	false
Disable adding 'use strict' directives in emitted JavaScript files.		
<code>--noLib</code>	boolean	false
Disable including any library files, including the default lib.d.ts.		
<code>--noPropertyAccessFromIndexSignature</code>	boolean	false
Enforces using indexed accessors for keys declared using an indexed type.		
<code>--noResolve</code>	boolean	false
Disallow <code>import s</code> , <code>require s</code> or <code><reference> s</code> from expanding the number of files TypeScript should add to a project.		
<code>--noStrictGenericChecks</code>	boolean	false
Disable strict checking of generic signatures in function types.		

Flag	Type	Default
<code>--noUncheckedIndexedAccess</code>	boolean	false
Add <code>undefined</code> to a type when accessed using an index.		
<code>--noUnusedLocals</code>	boolean	false
Enable error reporting when local variables aren't read.		
<code>--noUnusedParameters</code>	boolean	false
Raise an error when a function parameter isn't read.		
<code>--out</code>	string	
Deprecated setting. Use <code> outFile</code> instead.		
<code>--outDir</code>	string	
Specify an output folder for all emitted files.		
<code>--outFile</code>	string	
Specify a file that bundles all outputs into one JavaScript file. If <code>declaration</code> is true, also designates a file that bundles all .d.ts output.		
<code>--paths</code>	object	
Specify a set of entries that re-map imports to additional lookup locations.		
<code>--plugins</code>	list	
Specify a list of language service plugins to include.		

Flag	Type	Default
<code>--preserveConstEnums</code>	boolean	<ul style="list-style-type: none"> • <code>true</code> if <code>isolatedModules</code>, • <code>false</code> otherwise.
Disable erasing <code>const enum</code> declarations in generated code.		
<code>--preserveSymlinks</code>	boolean	<code>false</code>
Disable resolving symlinks to their realpath. This correlates to the same flag in node.		
<code>--preserveValueImports</code>	boolean	<code>false</code>
Preserve unused imported values in the JavaScript output that would otherwise be removed.		
<code>--preserveWatchOutput</code>	boolean	<code>false</code>
Disable wiping the console in watch mode.		
<code>--pretty</code>	boolean	<code>true</code>
Enable color and formatting in TypeScript's output to make compiler errors easier to read.		
<code>--reactNamespace</code>	string	<code>React</code>
Specify the object invoked for <code>createElement</code> . This only applies when targeting <code>react</code> JSX emit.		
<code>--removeComments</code>	boolean	<code>false</code>
Disable emitting comments.		

Flag	Type	Default
<code>--resolveJsonModule</code>	boolean	false
Enable importing .json files.		
<code>--rootDir</code>	string	Computed from the list of input files.
Specify the root folder within your source files.		
<code>--rootDirs</code>	list	Computed from the list of input files.
Allow multiple folders to be treated as one when resolving modules.		
<code>--skipDefaultLibCheck</code>	boolean	false
Skip type checking .d.ts files that are included with TypeScript.		
<code>--skipLibCheck</code>	boolean	false
Skip type checking all .d.ts files.		
<code>--sourceMap</code>	boolean	false
Create source map files for emitted JavaScript files.		
<code>--sourceRoot</code>	string	
Specify the root path for debuggers to find the reference source code.		
<code>--strict</code>	boolean	false
Enable all strict type-checking options.		

Flag	Type	Default
<code>--strictBindCallApply</code>	boolean	<ul style="list-style-type: none"> • <code>true</code> if <code>strict</code>, • <code>false</code> otherwise.
Check that the arguments for <code>bind</code> , <code>call</code> , and <code>apply</code> methods match the original function.		
<code>--strictFunctionTypes</code>	boolean	<ul style="list-style-type: none"> • <code>true</code> if <code>strict</code>, • <code>false</code> otherwise.
When assigning functions, check to ensure parameters and the return values are subtype-compatible.		
<code>--strictNullChecks</code>	boolean	<ul style="list-style-type: none"> • <code>true</code> if <code>strict</code>, • <code>false</code> otherwise.
When type checking, take into account <code>null</code> and <code>undefined</code> .		
<code>--strictPropertyInitialization</code>	boolean	<ul style="list-style-type: none"> • <code>true</code> if <code>strict</code>, • <code>false</code> otherwise.
Check for class properties that are declared but not set in the constructor.		
<code>--stripInternal</code>	boolean	<code>false</code>
Disable emitting declarations that have <code>@internal</code> in their JSDoc comments.		
<code>--suppressExcessPropertyErrors</code>	boolean	<code>false</code>
Disable reporting of excess property errors during the creation of object literals.		

Flag	Type	Default
<code>--suppressImplicitAnyIndexErrors</code>	boolean	false
		Suppress <code>noImplicitAny</code> errors when indexing objects that lack index signatures.
<code>--target</code>		<code>es3</code> , <code>es5</code> , <code>es6 / es2015</code> , <code>es2016</code> , <code>es-</code> <code>2017</code> , <code>es2018</code> , <code>es2019</code> , <code>es-</code> <code>2020</code> , <code>es2021</code> , <code>es2022</code> , or <code>es-</code> <code>next</code>
		Set the JavaScript language version for emitted JavaScript and include compatible library declarations.
<code>--traceResolution</code>	boolean	false
		Log paths used during the <code>moduleResolution</code> process.
<code>--tsBuildInfoFile</code>	string	<code>.tsbuildinfo</code>
		Specify the folder for <code>.tsbuildinfo</code> incremental compilation files.
<code>--typeRoots</code>		list
		Specify multiple folders that act like <code>./node_modules/@types</code> .
<code>--types</code>	list	
		Specify type package names to be included without being referenced in a source file.

Flag	Type	Default
<code>--useDefineForClassFields</code>	boolean	<ul style="list-style-type: none"> • <code>true</code> if <code>target</code> is <code>ES2022</code> or higher, including <code>ESNext</code>, • <code>false</code> otherwise. <p>Emit ECMAScript-standard-compliant class fields.</p>
<code>--useUnknownInCatchVariables</code>	boolean	<ul style="list-style-type: none"> • <code>true</code> if <code>strict</code>, • <code>false</code> otherwise.
		<p>Default catch clause variables as <code>unknown</code> instead of <code>any</code>.</p>

Related

- Every option is fully explained in the [TSConfig Reference](#).
- Learn how to use a `tsconfig.json` files.
- Learn how to work in an [MSBuild project](#).

[Go to TOC](#)

Overview

When you have an MSBuild based project which utilizes TypeScript such as an ASP.NET Core project, you can configure TypeScript in two ways. Either via a `tsconfig.json` or via the project settings.

Using a `tsconfig.json`

We recommend using a `tsconfig.json` for your project when possible. To add one to an existing project, add a new item to your project which is called a "TypeScript JSON Configuration File" in modern versions of Visual Studio.

The new `tsconfig.json` will then be used as the source of truth for TypeScript-specific build information like files and configuration. You can learn [about how TSConfigs works here](#) and there is a [comprehensive reference here](#).

Using Project Settings

You can also define the configuration for TypeScript inside your project's settings. This is done by editing the XML in your `.csproj` to define `PropertyGroups` which describe how the build can work:

```
<PropertyGroup>
  <TypeScriptNoEmitOnError>true</TypeScriptNoEmitOnError>
  <TypeScriptNoImplicitReturns>true</TypeScriptNoImplicitReturns>
</PropertyGroup>
```

There is a series of mappings for common TypeScript settings, these are settings which map directly to [TypeScript cli options](#) and are used to help you write a more understandable project file. You can use the [TSCConfig reference](#) to get more information on what values and defaults are for each mapping.

CLI Mappings

MSBuild Config Name	TSC Flag
<code><TypeScriptAllowJS></code>	<code>--allowJs</code>
Allow JavaScript files to be a part of your program. Use the <code>checkJS</code> option to get errors from these files.	
<code><TypeScriptRemoveComments></code>	<code>--removeComments</code>
Disable emitting comments.	
<code><TypeScriptNoImplicitAny></code>	<code>--noImplicitAny</code>

MSBuild	TSC Flag
Config	
Name	
Enable error reporting for expressions and declarations with an implied <code>any</code> type..	
<code><TypeScriptGeneratesDeclarations></code>	<code>--declaration</code>
Generate .d.ts files from TypeScript and JavaScript files in your project.	
<code><TypeScriptModuleKind></code>	<code>--module</code>
Specify what module code is generated.	
<code><TypeScriptJSX Emit></code>	<code>--jsx</code>
Specify what JSX code is generated.	
<code><TypeScriptOutDir></code>	<code>--outDir</code>
Specify an output folder for all emitted files.	
<code><TypeScriptSourceMap></code>	<code>--sourcemap</code>
Create source map files for emitted JavaScript files.	
<code><TypeScriptTarget></code>	<code>--target</code>
Set the JavaScript language version for emitted JavaScript and include compatible library declarations.	
<code><TypeScriptNoResolve></code>	<code>--noResolve</code>
Disallow <code>import s</code> , <code>require s</code> or <code><reference> s</code> from expanding the number of files TypeScript should add to a project.	
<code><TypeScriptMapRoot></code>	<code>--mapRoot</code>
Specify the location where debugger should locate map files instead of generated locations.	
<code><TypeScriptSourceRoot></code>	<code>--sourceRoot</code>
Specify the root path for debuggers to find the reference source code.	
<code><TypeScriptCharset></code>	<code>--charset</code>

MSBuild Config Name	TSC Flag
No longer supported. In early versions, manually set the text encoding for reading files.	
<TypeScriptEmitBOM>	--emitBOM
Emit a UTF-8 Byte Order Mark (BOM) in the beginning of output files.	
<TypeScriptNoLib>	--noLib
Disable including any library files, including the default lib.d.ts.	
<TypeScriptPreserveConstEnums>	--preserveConstEnums
Disable erasing <code>const enum</code> declarations in generated code.	--
<TypeScriptSuppressImplicitAnyIndexErrors>	suppressImplicitAnyIndexEr- rors
Suppress <code>noImplicitAny</code> errors when indexing objects that lack index signatures.	
<TypeScriptNoEmitHelpers>	--noEmitHelpers
Disable generating custom helper functions like <code>__extends</code> in compiled output.	
<TypeScriptInlineSourceMap>	--inlineSourceMap
Include sourcemap files inside the emitted JavaScript.	
<TypeScriptInlineSources>	--inlineSources
Include source code in the sourcemaps inside the emitted JavaScript.	
<TypeScript.NewLine>	--newLine
Set the newline character for emitting files.	
<TypeScriptIsolatedModules>	--isolatedModules
Ensure that each file can be safely transpiled without relying on other imports.	
<TypeScriptEmitDecoratorMetadata>	--emitDecoratorMetadata

MSBuild	TSC Flag
Config	
Name	
Emit design-type metadata for decorated declarations in source files.	
<TypeScriptRootDir>	--rootDir
Specify the root folder within your source files.	
<TypeScriptExperimentalDecorators>	--experimentalDecorators
Enable experimental support for TC39 stage 2 draft decorators.	
<TypeScriptModuleResolution>	--moduleResolution
Specify how TypeScript looks up a file from a given module specifier.	
<TypeScriptSuppressExcessPropertyErrors>	-- suppressExcessPropertyErrors
Disable reporting of excess property errors during the creation of object literals.	
<TypeScriptReactNamespace>	--reactNamespace
Specify the object invoked for <code>createElement</code> . This only applies when targeting <code>react</code> JSX emit.	
<TypeScriptSkipDefaultLibCheck>	--skipDefaultLibCheck
Skip type checking .d.ts files that are included with TypeScript.	
<TypeScriptAllowUnusedLabels>	--allowUnusedLabels
Disable error reporting for unused labels.	
<TypeScriptNoImplicitReturns>	--noImplicitReturns
Enable error reporting for codepaths that do not explicitly return in a function.	
<TypeScriptNoFallthroughCasesInSwitch>	--noFallthroughCasesInSwitch
Enable error reporting for fallthrough cases in switch statements.	
<TypeScriptAllowUnreachableCode>	--allowUnreachableCode

MSBuild Config Name	TSC Flag
Disable error reporting for unreachable code.	<code>--forceConsistentCasingInFileNames</code>
Ensure that casing is correct in imports.	<code>--allowSyntheticDefaultImports</code>
Allow 'import x from y' when a module doesn't have a default export.	<code>--noImplicitUseStrict</code>
Disable adding 'use strict' directives in emitted JavaScript files.	<code>--lib</code>
Specify a set of bundled library declaration files that describe the target runtime environment.	<code>--baseUrl</code>
Specify the base directory to resolve non-relative module names.	<code>--declarationDir</code>
Specify the output directory for generated declaration files.	<code>--noImplicitThis</code>
Enable error reporting when <code>this</code> is given the type <code>any</code> .	<code>--skipLibCheck</code>
Skip type checking all .d.ts files.	<code>--strictNullChecks</code>
When type checking, take into account <code>null</code> and <code>undefined</code> .	

MSBuild	TSC Flag
Config	
Name	
<TypeScriptNoUnusedLocals>	--noUnusedLocals
Enable error reporting when a local variables aren't read.	
<TypeScriptNoUnusedParameters>	--noUnusedParameters
Raise an error when a function parameter isn't read	
<TypeScriptAlwaysStrict>	--alwaysStrict
Ensure 'use strict' is always emitted.	
<TypeScriptImportHelpers>	--importHelpers
Allow importing helper functions from tslib once per project, instead of including them per-file.	
<TypeScriptJSXFactory>	--jsxFactory
Specify the JSX factory function used when targeting React JSX emit, e.g. 'React.createElement' or 'h'	
<TypeScriptStripInternal>	--stripInternal
Disable emitting declarations that have <code>@internal</code> in their JSDoc comments.	
<TypeScriptCheckJs>	--checkJs
Enable error reporting in type-checked JavaScript files.	
<TypeScriptDownlevelIteration>	--downlevelIteration
Emit more compliant, but verbose and less performant JavaScript for iteration.	
<TypeScriptStrict>	--strict
Enable all strict type checking options.	
<TypeScriptNoStrictGenericChecks>	--noStrictGenericChecks
Disable strict checking of generic signatures in function types.	
<TypeScriptPreserveSymlinks>	--preserveSymlinks

MSBuild Config Name	TSC Flag
Disable resolving symlinks to their realpath. This correlates to the same flag in node.	<code><TypeScriptStrictFunctionTypes></code> <code>--strictFunctionTypes</code>
When assigning functions, check to ensure parameters and the return values are subtype-compatible.	<code><TypeScriptStrictPropertyInitialization></code> <code>--strictPropertyInitialization</code>
Check for class properties that are declared but not set in the constructor.	<code><TypeScriptESModuleInterop></code> <code>--esModuleInterop</code>
Emit additional JavaScript to ease support for importing CommonJS modules. This enables <code>allowSyntheticDefaultImports</code> for type compatibility.	<code><TypeScriptEmitDeclarationOnly></code> <code>--emitDeclarationOnly</code>
Only output d.ts files and not JavaScript files.	<code><TypeScriptKeyofStringsOnly></code> <code>--keyofStringsOnly</code>
Make keyof only return strings instead of string, numbers or symbols. Legacy option.	<code><TypeScriptUseDefineForClassFields></code> <code>--useDefineForClassFields</code>
Emit ECMAScript-standard-compliant class fields.	<code><TypeScriptDeclarationMap></code> <code>--declarationMap</code>
Create sourcemaps for d.ts files.	<code><TypeScriptResolveJsonModule></code> <code>--resolveJsonModule</code>
Enable importing .json files	<code><TypeScriptStrictBindCallApply></code> <code>--strictBindCallApply</code>
Check that the arguments for <code>bind</code> , <code>call</code> , and <code>apply</code> methods match the original function.	<code><TypeScriptNoEmitOnError></code> <code>--noEmitOnError</code>

MSBuild	TSC Flag
Config	
Name	
Disable emitting files if any type checking errors are reported.	

Additional Flags

Because the MSBuild system passes arguments directly to the TypeScript CLI, you can use the option `TypeScriptAdditionalFlags` to provide specific flags which don't have a mapping above.

For example, this would turn on `noPropertyAccessFromIndexSignature`:

```
<TypeScriptAdditionalFlags> ${TypeScriptAdditionalFlags} --noPropertyAccessFromIndexSignature</TypeScriptAdditionalFlags>
```

Debug and Release Builds

You can use PropertyGroup conditions to define different sets of configurations. For example, a common task is stripping comments and sourcemaps in production. In this example, we define a debug and release property group which have different TypeScript configurations:

```
<PropertyGroup Condition="\"$(Configuration)" == 'Debug'">
  <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
  <TypeScriptSourceMap>true</TypeScriptSourceMap>
</PropertyGroup>

<PropertyGroup Condition="\"$(Configuration)" == 'Release'">
  <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
  <TypeScriptSourceMap>false</TypeScriptSourceMap>
</PropertyGroup>

<Import
  Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript"
  Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\' />
```

ToolsVersion

The value of `<TypeScriptToolsVersion>1.7</TypeScriptToolsVersion>` property in the project file identifies the compiler version to use to build (1.7 in this example). This allows a project to build against the same versions of the compiler on different machines.

If `TypeScriptToolsVersion` is not specified, the latest compiler version installed on the machine will be used to build.

Users using newer versions of TS, will see a prompt to upgrade their project on first load.

TypeScriptCompileBlocked

If you are using a different build tool to build your project (e.g. gulp, grunt , etc.) and VS for the development and debugging experience, set `<TypeScriptCompileBlocked>true</TypeScriptCompileBlocked>` in your project. This should give you all the editing support, but not the build when you hit F5.

TypeScriptEnableIncrementalMSBuild (TypeScript 4.2 Beta and later)

By default, MSBuild will attempt to only run the TypeScript compiler when the project's source files have been updated since the last compilation. However, if this behavior is causing issues, such as when TypeScript's `incremental` option is enabled, set `<TypeScriptEnableIncrementalMSBuild>false</TypeScriptEnableIncrementalMSBuild>` to ensure the TypeScript compiler is invoked with every run of MSBuild.

[Go to TOC](#)

Compiler supports configuring how to watch files and directories using compiler flags in TypeScript 3.8+, and environment variables before that.

Background

The `--watch` implementation of the compiler relies on using `fs.watch` and `fs.watchFile` which are provided by node, both of these methods have pros and cons.

`fs.watch` uses file system events to notify the changes in the file/directory. But this is OS dependent and the notification is not completely reliable and does not work as expected on many OS. Also there could be limit on number of watches that can be created, e.g. linux and we could exhaust it pretty quickly with programs that include large number of files. But because this uses file system events, there is not much CPU cycle involved. Compiler typically uses `fs.watch` to watch directories (e.g. source directories included by config file, directories in which module resolution failed etc) These can handle the missing precision in notifying about the changes. But recursive watching is supported on only Windows and OSX. That means we need something to replace the recursive nature on other OS.

`fs.watchFile` uses polling and thus involves CPU cycles. However, `fs.watchFile` is the most reliable mechanism to get the update on the status of file/directory. The compiler typically uses `fs.watchFile` to watch source files, config files and missing files (missing file references). This means the CPU usage when using `fs.watchFile` depends on number of files in the program.

Configuring file watching using a `tsconfig.json`

```
{
  // Some typical compiler options
  "compilerOptions": {
    "target": "es2020",
    "moduleResolution": "node"
    // ...
  },

  // NEW: Options for file/directory watching
  "watchOptions": {
    // Use native file system events for files and directories
    "watchFile": "useFsEvents",
    "watchDirectory": "useFsEvents",

    // Poll files for updates more frequently
    // when they're updated a lot.
    "fallbackPolling": "dynamicPriority",

    // Don't coalesce watch notification
    "synchronousWatchDirectory": true,

    // Finally, two additional settings for reducing the amount of possible
    // files to track work from these directories
    "excludeDirectories": ["**/node_modules", "__build__"],
    "excludeFiles": ["build/fileWhichChangesOften.ts"]
  }
}
```

You can read more about this in [the release notes](#).

Configuring file watching using environment variable

TSC_WATCHFILE

Option	Description
PriorityPollingInterval	Use <code>fs.watchFile</code> but use different polling intervals for source files, config files and missing files
DynamicPriorityPolling	Use a dynamic queue where frequently modified files will be polled at shorter interval and the files unchanged will be polled less frequently
UseFsEvents	Use <code>fs.watch</code> which uses file system events (but might not be accurate on different OS) to get the notifications for the file changes/creation/deletion. Note that few OS e.g. linux has limit on number of watches and failing to create watcher using <code>fs.watch</code> will result in creating using <code>fs.watchFile</code>
UseFsEventsWithFallbackDynamicPolling	This option is similar to <code>UseFsEvents</code> except on failing to create watch using <code>fs.watch</code> , the fallback watching happens through dynamic polling queues (as explained in <code>DynamicPriorityPolling</code>)
UseFsEventsOnParentDirectory	This option watches parent directory of the file with <code>fs.watch</code> (using file system events) thus being low on CPU but can compromise accuracy.
default (no value specified)	If environment variable <code>TSC_NONPOLLING_WATCHER</code> is set to true, watches parent directory of files (just like <code>UseFsEventsOnParentDirectory</code>). Otherwise watch files using <code>fs.watchFile</code> with <code>250ms</code> as the timeout for any file

Configuring directory watching using environment variable

TSC_WATCHDIRECTORY

The watching of directory on platforms that don't support recursive directory watching natively in node, is supported through recursively creating directory watcher for the child directories using different options selected by `TSC_WATCHDIRECTORY`. Note that on platforms that support native recursive directory watching (e.g windows) the value of this environment variable is ignored.

Option	Description
<code>RecursiveDirectoryUsingFsWatchFile</code>	Use <code>fs.watchFile</code> to watch the directories and child directories which is a polling watch (consuming CPU cycles)
<code>RecursiveDirectoryUsingDynamicPriorityPolling</code>	Use dynamic polling queue to poll changes to the directory and child directories.
default (no value specified)	Use <code>fs.watch</code> to watch directories and child directories

[Go to TOC](#)

Babel

Install

```
npm install @babel/cli @babel/core @babel/preset-typescript --save-dev
```

.babelrc

```
{ "presets": ["@babel/preset-typescript"] }
```

Using Command Line Interface

```
./node_modules/.bin/babel --out-file bundle.js src/index.ts
```

package.json

```
{ "scripts": { "build": "babel --out-file bundle.js main.ts" } }
```

Execute Babel from the command line

```
npm run build
```

Browserify

Install

```
npm install tsify
```

Using Command Line Interface

```
browserify main.ts -p [ tsify --noImplicitAny ] > bundle.js
```

Using API

```
var browserify = require("browserify");
var tsify = require("tsify");

browserify()
  .add("main.ts")
  .plugin("tsify", { noImplicitAny: true })
  .bundle()
  .pipe(process.stdout);
```

More details: [smrq/tsify](#)

Grunt

Install

```
npm install grunt-ts
```

Basic Gruntfile.js

```
module.exports = function (grunt) {
  grunt.initConfig({
    ts: {
      default: {
        src: ["**/*.ts", "!node_modules/**/*.ts"],
      },
    },
  });
  grunt.loadNpmTasks("grunt-ts");
  grunt.registerTask("default", ["ts"]);
};
```

More details: [TypeStrong/grunt-ts](#)

Gulp

Install

```
npm install gulp-typescript
```

Basic gulpfile.js

```
var gulp = require("gulp");
var ts = require("gulp-typescript");

gulp.task("default", function () {
  var tsResult = gulp.src("src/*.ts").pipe(
    ts({
      noImplicitAny: true,
      out: "output.js",
    })
  );
  return tsResult.js.pipe(gulp.dest("built/local"));
});
```

More details: [ivogabe/gulp-typescript](#)

Jspm

Install

```
npm install -g jspm@beta
```

Note: Currently TypeScript support in jspm is in 0.16beta

More details: [TypeScriptSamples/jspm](#)

MSBuild

Update project file to include locally installed `Microsoft.TypeScript.Default.props` (at the top) and `Microsoft.TypeScript.targets` (at the bottom) files:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- Include default props at the top -->
  <Import
    Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript'
    Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\`/>

    <!-- TypeScript configurations go here -->
    <PropertyGroup Condition="$(Configuration) == 'Debug'">
      <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
      <TypeScriptSourceMap>true</TypeScriptSourceMap>
    </PropertyGroup>
    <PropertyGroup Condition="$(Configuration) == 'Release'">
      <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
      <TypeScriptSourceMap>false</TypeScriptSourceMap>
    </PropertyGroup>

    <!-- Include default targets at the bottom -->
    <Import
      Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript'
      Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\`/>
    </Project>
```

More details about defining MSBuild compiler options: [Setting Compiler Options in MSBuild projects](#)

NuGet

- Right-Click -> Manage NuGet Packages
- Search for `Microsoft.TypeScript.MSBuild`
- Hit `Install`
- When install is complete, rebuild!

More details can be found at [Package Manager Dialog](#) and [using nightly builds with NuGet](#)

Rollup

Install

```
npm install @rollup/plugin-typescript --save-dev
```

Note that both `typescript` and `tslib` are peer dependencies of this plugin that need to be installed separately.

Usage

Create a `rollup.config.js` configuration file and import the plugin:

```
// rollup.config.js
import typescript from '@rollup/plugin-typescript';

export default {
  input: 'src/index.ts',
  output: {
    dir: 'output',
    format: 'cjs'
  },
  plugins: [typescript()]
};
```

Svelte Compiler

Install

```
npm install --save-dev svelte-preprocess
```

Note that `typescript` is an optional peer dependencies of this plugin and needs to be installed separately. `tslib` is not provided either.

You may also consider `svelte-check` for CLI type checking.

Usage

Create a `svelte.config.js` configuration file and import the plugin:

```
// svelte.config.js
import preprocess from 'svelte-preprocess';

const config = {
  // Consult https://github.com/sveltejs/svelte-preprocess
  // for more information about preprocessors
  preprocess: preprocess()
};

export default config;
```

You can now specify that script blocks are written in TypeScript:

```
<script lang="ts">
```

Vite

Vite supports importing `.ts` files out-of-the-box. It only performs transpilation and not type checking. It also requires that some `compilerOptions` have certain values. See the [Vite docs](#) for more details.

Webpack

Install

```
npm install ts-loader --save-dev
```

Basic webpack.config.js when using Webpack 5 or 4

```
const path = require('path');

module.exports = {
  entry: './src/index.ts',
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/,
      },
    ],
  },
  resolve: {
    extensions: ['.tsx', '.ts', '.js'],
  },
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
};
```

See [more details on ts-loader here](#).

Alternatives:

- [awesome-typescript-loader](#)

[Go to TOC](#)

Project references are a new feature in TypeScript 3.0 that allow you to structure your TypeScript programs into smaller pieces.

By doing this, you can greatly improve build times, enforce logical separation between components, and organize your code in new and better ways.

We're also introducing a new mode for `tsc`, the `--build` flag, that works hand in hand with project references to enable faster TypeScript builds.

An Example Project

Let's look at a fairly normal program and see how project references can help us better organize it. Imagine you have a project with two modules, `converter` and `units`, and a corresponding test file for each:

```
/  
  └── src/  
    |   └── converter.ts  
    |   └── units.ts  
  └── test/  
    |   └── converter-tests.ts  
    |   └── units-tests.ts  
  └── tsconfig.json
```

The test files import the implementation files and do some testing:

```
// converter-tests.ts  
import * as converter from "../converter";  
  
assert.areEqual(converter.celsiusToFahrenheit(0), 32);
```

Previously, this structure was rather awkward to work with if you used a single `tsconfig` file:

- It was possible for the implementation files to import the test files
- It wasn't possible to build `test` and `src` at the same time without having `src` appear in the output folder name, which you probably don't want
- Changing just the *internals* in the implementation files required *typechecking* the tests again, even though this wouldn't ever cause new errors
- Changing just the tests required typechecking the implementation again, even if nothing changed

You could use multiple `tsconfig` files to solve *some* of those problems, but new ones would appear:

- There's no built-in up-to-date checking, so you end up always running `tsc` twice
- Invoking `tsc` twice incurs more startup time overhead
- `tsc -w` can't run on multiple config files at once

Project references can solve all of these problems and more.

What is a Project Reference?

`tsconfig.json` files have a new top-level property, `references`. It's an array of objects that specifies projects to reference:

```
{
  "compilerOptions": {
    // The usual
  },
  "references": [
    { "path": "../src" }
  ]
}
```

The `path` property of each reference can point to a directory containing a `tsconfig.json` file, or to the config file itself (which may have any name).

When you reference a project, new things happen:

- Importing modules from a referenced project will instead load its `output` declaration file (`.d.ts`)
- If the referenced project produces an `outFile`, the output file `.d.ts` file's declarations will be visible in this project
- Build mode (see below) will automatically build the referenced project if needed

By separating into multiple projects, you can greatly improve the speed of typechecking and compiling, reduce memory usage when using an editor, and improve enforcement of the logical groupings of your program.

composite

Referenced projects must have the new `composite` setting enabled. This setting is needed to ensure TypeScript can quickly determine where to find the outputs of the referenced project. Enabling the `composite` flag changes a few things:

- The `rootDir` setting, if not explicitly set, defaults to the directory containing the `tsconfig` file
- All implementation files must be matched by an `include` pattern or listed in the `files` array. If this constraint is violated, `tsc` will inform you which files weren't specified
- `declaration` must be turned on

declarationMaps

We've also added support for [declaration source maps](#). If you enable `declarationMap`, you'll be able to use editor features like "Go to Definition" and Rename to transparently navigate and edit code across project boundaries in supported editors.

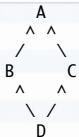
prepend with outFile

You can also enable prepending the output of a dependency using the `prepend` option in a reference:

```
"references": [
  { "path": "../utils", "prepend": true }
]
```

Prepending a project will include the project's output above the output of the current project. All output files (`.js`, `.d.ts`, `.js.map`, `.d.ts.map`) will be emitted correctly.

`tsc` will only ever use existing files on disk to do this process, so it's possible to create a project where a correct output file can't be generated because some project's output would be present more than once in the resulting file. For example:



It's important in this situation to not prepend at each reference, because you'll end up with two copies of `A` in the output of `D` - this can lead to unexpected results.

Caveats for Project References

Project references have a few trade-offs you should be aware of.

Because dependent projects make use of `.d.ts` files that are built from their dependencies, you'll either have to check in certain build outputs or build a project after cloning it before you can navigate the project in an editor without seeing spurious errors.

When using VS Code (since TS 3.7) we have a behind-the-scenes in-memory `.d.ts` generation process that should be able to mitigate this, but it has some perf implications. For very large composite projects you might want to disable this using `disableSourceOfProjectReferenceRedirect` option.

Additionally, to preserve compatibility with existing build workflows, `tsc` will *not* automatically build dependencies unless invoked with the `--build` switch. Let's learn more about `--build`.

Build Mode for TypeScript

A long-awaited feature is smart incremental builds for TypeScript projects. In 3.0 you can use the `--build` flag with `tsc`. This is effectively a new entry point for `tsc` that behaves more like a build orchestrator than a simple compiler.

Running `tsc --build` (`tsc -b` for short) will do the following:

- Find all referenced projects
- Detect if they are up-to-date
- Build out-of-date projects in the correct order

You can provide `tsc -b` with multiple config file paths (e.g. `tsc -b src test`). Just like `tsc -p`, specifying the config file name itself is unnecessary if it's named `tsconfig.json`.

tsc -b Commandline

You can specify any number of config files:

```
> tsc -b
      # Use the tsconfig.json in the current
      directory
> tsc -b src
      # Use src/tsconfig.json
> tsc -b foo/prd.tsconfig.json bar
      # Use foo/prd.tsconfig.json and
      bar/tsconfig.json
```

Don't worry about ordering the files you pass on the commandline - `tsc` will re-order them if needed so that dependencies are always built first.

There are also some flags specific to `tsc -b`:

- `--verbose` : Prints out verbose logging to explain what's going on (may be combined with any other flag)
- `--dry` : Shows what would be done but doesn't actually build anything
- `--clean` : Deletes the outputs of the specified projects (may be combined with `--dry`)
- `--force` : Act as if all projects are out of date
- `--watch` : Watch mode (may not be combined with any flag except `--verbose`)

Caveats

Normally, `tsc` will produce outputs (`.js` and `.d.ts`) in the presence of syntax or type errors, unless `noEmitOnError` is on. Doing this in an incremental build system would be very bad - if one of your out-of-date dependencies had a new error, you'd only see it *once* because a subsequent build would skip building the now up-to-date project. For this reason, `tsc -b` effectively acts as if `noEmitOnError` is enabled for all projects.

If you check in any build outputs (`.js`, `.d.ts`, `.d.ts.map`, etc.), you may need to run a `--force` build after certain source control operations depending on whether your source control tool preserves timestamps between the local copy and the remote copy.

MSBuild

If you have an msbuild project, you can enable build mode by adding

```
<TypeScriptBuildMode>true</TypeScriptBuildMode>
```

to your proj file. This will enable automatic incremental build as well as cleaning.

Note that as with `tsconfig.json` / `-p`, existing TypeScript project properties will not be respected - all settings should be managed using your `tsconfig` file.

Some teams have set up msbuild-based workflows wherein tsconfig files have the same *implicit* graph ordering as the managed projects they are paired with. If your solution is like this, you can continue to use `msbuild` with `tsc -p` along with project references; these are fully interoperable.

Guidance

Overall Structure

With more `tsconfig.json` files, you'll usually want to use [Configuration file inheritance](#) to centralize your common compiler options. This way you can change a setting in one file rather than having to edit multiple files.

Another good practice is to have a "solution" `tsconfig.json` file that simply has `references` to all of your leaf-node projects and sets `files` to an empty array (otherwise the solution file will cause double compilation of files). Note that starting with 3.0, it is no longer an error to have an empty `files` array if you have at least one `reference` in a `tsconfig.json` file.

This presents a simple entry point; e.g. in the TypeScript repo we simply run `tsc -b src` to build all endpoints because we list all the subprojects in `src/tsconfig.json`.

You can see these patterns in the TypeScript repo - see `src/tsconfig_base.json`, `src/tsconfig.json`, and `src/tsc/tsconfig.json` as key examples.

Structuring for relative modules

In general, not much is needed to transition a repo using relative modules. Simply place a `tsconfig.json` file in each subdirectory of a given parent folder, and add `reference`s to these config files to match the intended layering of the program. You will need to either set the `outDir` to an explicit subfolder of the output folder, or set the `rootDir` to the common root of all project folders.

Structuring for outFile

Layout for compilations using `outFile` is more flexible because relative paths don't matter as much. One thing to keep in mind is that you'll generally want to not use `prepend` until the "last" project - this will improve build times and reduce the amount of I/O needed in any given build. The TypeScript repo itself is a good reference here - we have some "library" projects and some "endpoint" projects; "endpoint" projects are kept as small as possible and pull in only the libraries they need.

[Go to TOC](#)

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:

Using `tsconfig.json` or `jsconfig.json`

- 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

Example `tsconfig.json` files:

- 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

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

```
{
  "extends": "@tsconfig/node12/tsconfig.json",

  "compilerOptions": {
    "preserveConstEnums": true
  },

  "include": ["src/**/*"],
  "exclude": ["node_modules", "**/*.spec.ts"]
}
```

This lets your `tsconfig.json` focus on the unique choices for your project, and not all of the runtime mechanics. There are a few tsconfig bases already, and we're hoping the community can add more for different environments.

- Recommended
- Node 10
- Node 12
- Node 14
- Node 16
- Deno
- React Native
- Svelte

Details

The `"compilerOptions"` property can be omitted, in which case the compiler's defaults are used. See our 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](#).

[Go to TOC](#)

This page lists some of the more advanced ways in which you can model types, it works in tandem with the [Utility Types](#) doc which includes types which are included in TypeScript and available globally.

Type Guards and Differentiating Types

Union types are useful for modeling situations when values can overlap in the types they can take on. What happens when we need to know specifically whether we have a `Fish`? A common idiom in JavaScript to differentiate between two possible values is to check for the presence of a member. As we mentioned, you can only access members that are guaranteed to be in all the constituents of a union type.

```
// @errors: 2339
type Fish = { swim: () => void };
type Bird = { fly: () => void };
declare function getSmallPet(): Fish | Bird;
// ---cut---
let pet = getSmallPet();

// You can use the 'in' operator to check
if ("swim" in pet) {
  pet.swim();
}
// However, you cannot use property access
if (pet.fly) {
  pet.fly();
}
```

To get the same code working via property accessors, we'll need to use a type assertion:

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };
declare function getSmallPet(): Fish | Bird;
// ---cut---
let pet = getSmallPet();
let fishPet = pet as Fish;
let birdPet = pet as Bird;

if (fishPet.swim) {
  fishPet.swim();
} else if (birdPet.fly) {
  birdPet.fly();
}
```

This isn't the sort of code you would want in your codebase however.

User-Defined Type Guards

It would be much better if once we performed the check, we could know the type of `pet` within each branch.

It just so happens that TypeScript has something called a *type guard*. A type guard is some expression that performs a runtime check that guarantees the type in some scope.

Using type predicates

To define a type guard, we simply need to define a function whose return type is a *type predicate*:

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };
declare function getSmallPet(): Fish | Bird;
// ---cut---
function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}
```

`pet is Fish` is our type predicate in this example. A predicate takes the form `parameterName is Type`, where `parameterName` must be the name of a parameter from the current function signature.

Any time `isFish` is called with some variable, TypeScript will *narrow* that variable to that specific type if the original type is compatible.

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };
declare function getSmallPet(): Fish | Bird;
function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}
// ---cut---
// Both calls to 'swim' and 'fly' are now okay.
let pet = getSmallPet();

if (isFish(pet)) {
  pet.swim();
} else {
  pet.fly();
}
```

Notice that TypeScript not only knows that `pet` is a `Fish` in the `if` branch; it also knows that in the `else` branch, you *don't* have a `Fish`, so you must have a `Bird`.

You may use the type guard `isFish` to filter an array of `Fish | Bird` and obtain an array of `Fish`:

```
// @errors: 2345
type Fish = { swim: () => void };
type Bird = { fly: () => void };
declare function getSmallPet(): Fish | Bird;
function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}
// ---cut---
const zoo: (Fish | Bird)[] = [getSmallPet(), getSmallPet(), getSmallPet()];
const underWater1: Fish[] = zoo.filter(isFish);
// or, equivalently
const underWater2: Fish[] = zoo.filter<Fish>(isFish);
const underWater3: Fish[] = zoo.filter<Fish>((pet) => isFish(pet));
```

Using the `in` operator

The `in` operator also acts as a narrowing expression for types.

For a `n in x` expression, where `n` is a string literal or string literal type and `x` is a union type, the "true" branch narrows to types which have an optional or required property `n`, and the "false" branch narrows to types which have an optional or missing property `n`.

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };
// ---cut---
function move(pet: Fish | Bird) {
  if ("swim" in pet) {
    return pet.swim();
  }
  return pet.fly();
}
```

typeof type guards

Let's go back and write the code for a version of `padLeft` which uses union types. We could write it with type predicates as follows:

```
function isNumber(x: any): x is number {
  return typeof x === "number";
}

function isString(x: any): x is string {
  return typeof x === "string";
}

function padLeft(value: string, padding: string | number) {
  if (isNumber(padding)) {
    return Array(padding + 1).join(" ") + value;
  }
  if (isString(padding)) {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${padding}'`);
}
```

However, having to define a function to figure out if a type is a primitive is kind of a pain. Luckily, you don't need to abstract `typeof x === "number"` into its own function because TypeScript will recognize it as a type guard on its own. That means we could just write these checks inline.

```
function padLeft(value: string, padding: string | number) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${padding}'`);
}
```

These `typeof` type guards are recognized in two different forms: `typeof v === "typename"` and `typeof v !== "typename"`, where "typename" can be one of `typeof` operator's return values ("`undefined`", "`number`", "`string`", "`boolean`", "`bigint`", "`symbol`", "`object`", or "`function`"). While TypeScript won't stop you from comparing to other strings, the language won't recognize those expressions as type guards.

instanceof type guards

If you've read about `typeof` type guards and are familiar with the `instanceof` operator in JavaScript, you probably have some idea of what this section is about.

`instanceof` type guards are a way of narrowing types using their constructor function. For instance, let's borrow our industrial strength string-padder example from earlier:

```
interface Padder {
  getPaddingString(): string;
}

class SpaceRepeatingPadder implements Padder {
  constructor(private numSpaces: number) {}
  getPaddingString() {
    return Array(this.numSpaces + 1).join(" ");
  }
}

class StringPadder implements Padder {
  constructor(private value: string) {}
  getPaddingString() {
    return this.value;
  }
}

function getRandomPadder() {
  return Math.random() < 0.5
    ? new SpaceRepeatingPadder(4)
    : new StringPadder(" ");
}

let padder: Padder = getRandomPadder();
//           ^?

if (padder instanceof SpaceRepeatingPadder) {
  padder;
//   ^
}
if (padder instanceof StringPadder) {
  padder;
//   ^
}
```

The right side of the `instanceof` needs to be a constructor function, and TypeScript will narrow down to:

1. the type of the function's `prototype` property if its type is not `any`
2. the union of types returned by that type's construct signatures

in that order.

Nullable types

TypeScript has two special types, `null` and `undefined`, that have the values null and undefined respectively. We mentioned these briefly in [the Basic Types section](#).

By default, the type checker considers `null` and `undefined` assignable to anything. Effectively, `null` and `undefined` are valid values of every type. That means it's not possible to *stop* them from being assigned to any type, even when you would like to prevent it. The inventor of `null`, Tony Hoare, calls this his "[billion dollar mistake](#)".

The `strictNullChecks` flag fixes this: when you declare a variable, it doesn't automatically include `null` or `undefined`. You can include them explicitly using a union type:

```
// @errors: 2322
let exampleString = "foo";
exampleString = null;

let stringOrNull: string | null = "bar";
stringOrNull = null;

stringOrNull = undefined;
```

Note that TypeScript treats `null` and `undefined` differently in order to match JavaScript semantics. `string | null` is a different type than `string | undefined` and `string | undefined | null`.

From TypeScript 3.7 and onwards, you can use [optional chaining](#) to simplify working with nullable types.

Optional parameters and properties

With `strictNullChecks`, an optional parameter automatically adds `| undefined`:

```
// @errors: 2345
function f(x: number, y?: number) {
  return x + (y ?? 0);
}

f(1, 2);
f(1);
f(1, undefined);
f(1, null);
```

The same is true for optional properties:

```
// @strict: false
// @strictNullChecks: true
// @errors: 2322
class C {
  a: number;
  b?: number;
}

let c = new C();
```

```
c.a = 12;
c.a = undefined;
c.b = 13;
c.b = undefined;
c.b = null;
```

Type guards and type assertions

Since nullable types are implemented with a union, you need to use a type guard to get rid of the `null`. Fortunately, this is the same code you'd write in JavaScript:

```
function f(stringOrNull: string | null): string {
  if (stringOrNull === null) {
    return "default";
  } else {
    return stringOrNull;
  }
}
```

The `null` elimination is pretty obvious here, but you can use terser operators too:

```
function f(stringOrNull: string | null): string {
  return stringOrNull ?? "default";
}
```

In cases where the compiler can't eliminate `null` or `undefined`, you can use the type assertion operator to manually remove them. The syntax is postfix `! : identifier!` removes `null` and `undefined` from the type of `identifier`:

```
// @errors: 2532
function getUser(id: string): UserAccount | undefined {
  return {} as any;
}
// ---cut---
interface UserAccount {
  id: number;
  email?: string;
}

const user = getUser("admin");
user.id;

if (user) {
  user.email.length;
}

// Instead if you are sure that these objects or fields exist, the
// postfix ! lets you short circuit the nullability
user!.email!.length;
```

Type Aliases

Type aliases create a new name for a type. Type aliases are sometimes similar to interfaces, but can name primitives, unions, tuples, and any other types that you'd otherwise have to write by hand.

```
type Second = number;
let timeInSeconds: number = 10;
let time: Second = 10;
```

Aliasing doesn't actually create a new type - it creates a new *name* to refer to that type. Aliasing a primitive is not terribly useful, though it can be used as a form of documentation.

Just like interfaces, type aliases can also be generic - we can just add type parameters and use them on the right side of the alias declaration:

```
type Container<T> = { value: T };
```

We can also have a type alias refer to itself in a property:

```
type Tree<T> = {
  value: T;
  left?: Tree<T>;
  right?: Tree<T>;
};
```

Together with `intersection` types, we can make some pretty mind-bending types:

```
declare function getDriversLicenseQueue(): LinkedList<Person>;
// ---cut---
type LinkedList<Type> = Type & { next: LinkedList<Type> };

interface Person {
  name: string;
}

let people = getDriversLicenseQueue();
people.name;
people.next.name;
people.next.next.name;
people.next.next.next.name;
//           ^?
```

Interfaces vs. Type Aliases

As we mentioned, type aliases can act sort of like interfaces; however, there are some subtle differences.

Almost all features of an `interface` are available in `type`, the key distinction is that a type cannot be re-opened to add new properties vs an interface which is always extendable.

Interface	Type
Extending an interface	Extending a type via intersections

```

interface Animal {
  name: string
}

interface Bear extends Animal {
  honey: boolean
}

const bear = getBear()
bear.name
bear.honey

```

```

type Animal = {
  name: string
}

type Bear = Animal & {
  honey: Boolean
}

const bear = getBear();
bear.name;
bear.honey;

```

Adding new fields to an existing interface

A type cannot be changed after being created

```

interface Window {
  title: string
}

interface Window {
  ts: import("typescript")
}

const src = 'const a = "Hello World"'; // Error: Duplicate identifier 'Window'.
window.ts.transpileModule(src, {});

```

Because an interface more closely maps how JavaScript objects work [by being open to extension](#), we recommend using an interface over a type alias when possible.

On the other hand, if you can't express some shape with an interface and you need to use a union or tuple type, type aliases are usually the way to go.

Enum Member Types

As mentioned in [our section on enums](#), enum members have types when every member is literal-initialized.

Much of the time when we talk about "singleton types", we're referring to both enum member types as well as numeric/string literal types, though many users will use "singleton types" and "literal types" interchangeably.

Polymorphic `this` types

A polymorphic `this` type represents a type that is the *subtype* of the containing class or interface. This is called *F*-bounded polymorphism, a lot of people know it as the [fluent API](#) pattern. This makes hierarchical fluent interfaces much easier to express, for example. Take a simple calculator that returns `this` after each operation:

```
class BasicCalculator {
  public constructor(protected value: number = 0) {}
  public currentValue(): number {
    return this.value;
  }
  public add(operand: number): this {
    this.value += operand;
    return this;
  }
  public multiply(operand: number): this {
    this.value *= operand;
    return this;
  }
  // ... other operations go here ...
}

let v = new BasicCalculator(2).multiply(5).add(1).currentValue();
```

Since the class uses `this` types, you can extend it and the new class can use the old methods with no changes.

```
class BasicCalculator {
  public constructor(protected value: number = 0) {}
  public currentValue(): number {
    return this.value;
  }
  public add(operand: number): this {
    this.value += operand;
    return this;
  }
  public multiply(operand: number): this {
    this.value *= operand;
    return this;
  }
  // ... other operations go here ...
}
// ---cut---
class ScientificCalculator extends BasicCalculator {
  public constructor(value = 0) {
    super(value);
  }
  public sin(): {
    this.value = Math.sin(this.value);
    return this;
  }
  // ... other operations go here ...
}
```

```
}
```

```
let v = new ScientificCalculator(2).multiply(5).sin().add(1).currentValue();
```

Without `this` types, `ScientificCalculator` would not have been able to extend `BasicCalculator` and keep the fluent interface. `multiply` would have returned `BasicCalculator`, which doesn't have the `sin` method. However, with `this` types, `multiply` returns `this`, which is `ScientificCalculator` here.

Index types

With index types, you can get the compiler to check code that uses dynamic property names. For example, a common JavaScript pattern is to pick a subset of properties from an object:

```
function pluck(o, propertyNames) {
  return propertyNames.map((n) => o[n]);
}
```

Here's how you would write and use this function in TypeScript, using the **index type query** and **indexed access** operators:

```
function pluck<T, K extends keyof T>(o: T, propertyNames: K[]): T[K][] {
  return propertyNames.map((n) => o[n]);
}

interface Car {
  manufacturer: string;
  model: string;
  year: number;
}

let taxi: Car = {
  manufacturer: "Toyota",
  model: "Camry",
  year: 2014,
};

// Manufacturer and model are both of type string,
// so we can pluck them both into a typed string array
let makeAndModel: string[] = pluck(taxi, ["manufacturer", "model"]);

// If we try to pluck model and year, we get an
// array of a union type: (string | number)[]
let modelYear = pluck(taxi, ["model", "year"]);
```

The compiler checks that `manufacturer` and `model` are actually properties on `Car`. The example introduces a couple of new type operators. First is `keyof T`, the **index type query operator**. For any type `T`, `keyof T` is the union of known, public property names of `T`. For example:

```
interface Car {
  manufacturer: string;
  model: string;
  year: number;
}
```

```
// ---cut---
let carProps: keyof Car;
// ^?
```

`keyof Car` is completely interchangeable with `"manufacturer" | "model" | "year"`. The difference is that if you add another property to `Car`, say `ownersAddress: string`, then `keyof Car` will automatically update to be `"manufacturer" | "model" | "year" | "ownersAddress"`. And you can use `keyof` in generic contexts like `pluck`, where you can't possibly know the property names ahead of time. That means the compiler will check that you pass the right set of property names to `pluck`:

```
// error, Type '"unknown"' is not assignable to type '"manufacturer" | "model" |
"year"
pluck(taxi, ["year", "unknown"]);
```

The second operator is `T[K]`, the **indexed access operator**. Here, the type syntax reflects the expression syntax. That means that `taxi["manufacturer"]` has the type `Car["manufacturer"]` — which in our example is just `string`. However, just like index type queries, you can use `T[K]` in a generic context, which is where its real power comes to life. You just have to make sure that the type variable `K` extends `keyof T`. Here's another example with a function named `getProperty`.

```
function getProperty<T, K extends keyof T>(o: T, propertyName: K): T[K] {
  return o[propertyName]; // o[propertyName] is of type T[K]
}
```

In `getProperty`, `o: T` and `propertyName: K`, so that means `o[propertyName]: T[K]`. Once you return the `T[K]` result, the compiler will instantiate the actual type of the key, so the return type of `getProperty` will vary according to which property you request.

```
// @errors: 2345
function getProperty<T, K extends keyof T>(o: T, propertyName: K): T[K] {
  return o[propertyName]; // o[propertyName] is of type T[K]
}
interface Car {
  manufacturer: string;
  model: string;
  year: number;
}
let taxi: Car = {
  manufacturer: "Toyota",
  model: "Camry",
  year: 2014,
};
// ---cut---
let manufacturer: string = getProperty(taxi, "manufacturer");
let year: number = getProperty(taxi, "year");

let unknown = getProperty(taxi, "unknown");
```

Index types and index signatures

`keyof` and `T[K]` interact with index signatures. An index signature parameter type must be 'string' or 'number'. If you have a type with a string index signature, `keyof T` will be `string | number` (and not just `string`, since in JavaScript you can access an object property either by using strings (`object["42"]`) or numbers (`object[42]`)). And `T[string]` is just the type of the index signature:

```
// @errors: 2339
interface Dictionary<T> {
  [key: string]: T;
}
let keys: keyof Dictionary<number>;
//   ^
let value: Dictionary<number>["foo"];
//   ^?
```

If you have a type with a number index signature, `keyof T` will just be `number`.

```
// @errors: 2339
interface Dictionary<T> {
  [key: number]: T;
}
let keys: keyof Dictionary<number>;
//   ^
let numberValue: Dictionary<number>[42];
//   ^
let value: Dictionary<number>["foo"];
```

Mapped types

A common task is to take an existing type and make each of its properties optional:

```
interface PersonSubset {
  name?: string;
  age?: number;
}
```

Or we might want a readonly version:

```
interface PersonReadonly {
  readonly name: string;
  readonly age: number;
}
```

This happens often enough in JavaScript that TypeScript provides a way to create new types based on old types — **mapped types**. In a mapped type, the new type transforms each property in the old type in the same way. For example, you can make all properties optional or of a type `readonly`. Here are a couple of examples:

```
type Partial<T> = {
  [P in keyof T]?: T[P];
};
```

```
// @noErrors
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};
```

And to use it:

```
type Person = {
  name: string;
  age: number;
};
// ---cut---
type PersonPartial = Partial<Person>;
// ^
type ReadonlyPerson = Readonly<Person>;
// ^?
```

Note that this syntax describes a type rather than a member. If you want to add members, you can use an intersection type:

```
// @errors: 2693 1005 1128 7061
// Use this:
type PartialWithNewMember<T> = {
  [P in keyof T]?: T[P];
} & { newMember: boolean }

// This is an error!
type WrongPartialWithNewMember<T> = {
  [P in keyof T]?: T[P];
  newMember: boolean;
}
```

Let's take a look at the simplest mapped type and its parts:

```
type Keys = "option1" | "option2";
type Flags = { [K in Keys]: boolean };
```

The syntax resembles the syntax for index signatures with a `for .. in` inside. There are three parts:

1. The type variable `K`, which gets bound to each property in turn.
2. The string literal union `Keys`, which contains the names of properties to iterate over.
3. The resulting type of the property.

In this simple example, `Keys` is a hard-coded list of property names and the property type is always `boolean`, so this mapped type is equivalent to writing:

```
type Flags = {
  option1: boolean;
  option2: boolean;
};
```

Real applications, however, look like `Readonly` or `Partial` above. They're based on some existing type, and they transform the properties in some way. That's where `keyof` and indexed access types come in:

```

type Person = {
  name: string;
  age: number;
};

// ---cut---
type NullablePerson = { [P in keyof Person]: Person[P] | null };
//   ^
type PartialPerson = { [P in keyof Person]?: Person[P] };
//   ^

```

But it's more useful to have a general version.

```

type Nullable<T> = { [P in keyof T]: T[P] | null };
type Partial<T> = { [P in keyof T]?: T[P] };

```

In these examples, the properties list is `keyof T` and the resulting type is some variant of `T[P]`. This is a good template for any general use of mapped types. That's because this kind of transformation is [homomorphic](#), which means that the mapping applies only to properties of `T` and no others. The compiler knows that it can copy all the existing property modifiers before adding any new ones. For example, if `Person.name` was readonly, `Partial<Person>.name` would be readonly and optional.

Here's one more example, in which `T[P]` is wrapped in a `Proxy<T>` class:

```

// @noErrors
type Proxy<T> = {
  get(): T;
  set(value: T): void;
};

type Proxify<T> = {
  [P in keyof T]: Proxy<T[P]>;
};

function proxify<T>(o: T): Proxify<T> {
  // ... wrap proxies ...
}

let props = { rooms: 4 };
let proxyProps = proxify(props);
// ^

```

Note that `Readonly<T>` and `Partial<T>` are so useful, they are included in TypeScript's standard library along with `Pick` and `Record`:

```

type Pick<T, K extends keyof T> = {
  [P in K]: T[P];
};

type Record<K extends keyof any, T> = {
  [P in K]: T;
};

```

`Readonly`, `Partial` and `Pick` are homomorphic whereas `Record` is not. One clue that `Record` is not homomorphic is that it doesn't take an input type to copy properties from:

```

type ThreeStringProps = Record<"prop1" | "prop2" | "prop3", string>;

```

Non-homomorphic types are essentially creating new properties, so they can't copy property modifiers from anywhere.

Note that `keyof any` represents the type of any value that can be used as an index to an object. In other words, `keyof any` is currently equal to `string | number | symbol`.

Inference from mapped types

Now that you know how to wrap the properties of a type, the next thing you'll want to do is unwrap them. Fortunately, that's pretty easy:

```
type Proxy<T> = {
  get(): T;
  set(value: T): void;
};

type Proxify<T> = {
  [P in keyof T]: Proxy<T[P]>;
};

function proxify<T>(o: T): Proxify<T> {
  return {} as any;
}

let props = { rooms: 4 };
let proxyProps = proxify(props);
// ---cut---
function unproxify<T>(t: Proxify<T>): T {
  let result = {} as T;
  for (const k in t) {
    result[k] = t[k].get();
  }
  return result;
}

let originalProps = unproxify(proxyProps);
// ^?
```

Note that this unwrapping inference only works on homomorphic mapped types. If the mapped type is not homomorphic you'll have to give an explicit type parameter to your unwrapping function.

Conditional Types

A conditional type selects one of two possible types based on a condition expressed as a type relationship test:

```
T extends U ? X : Y
```

The type above means when `T` is assignable to `U` the type is `X`, otherwise the type is `Y`.

A conditional type `T extends U ? X : Y` is either *resolved* to `X` or `Y`, or *deferred* because the condition depends on one or more type variables. When `T` or `U` contains type variables, whether to resolve to `X` or `Y`, or to defer, is determined by whether or not the type system has enough information to conclude that `T` is always assignable to `U`.

As an example of some types that are immediately resolved, we can take a look at the following example:

```
declare function f<T extends boolean>(x: T): T extends true ? string : number;
// Type is 'string | number'
let x = f(Math.random() < 0.5);
// ^?
```

Another example would be the `TypeName` type alias, which uses nested conditional types:

```
type TypeName<T> = T extends string
? "string"
: T extends number
? "number"
: T extends boolean
? "boolean"
: T extends undefined
? "undefined"
: T extends Function
? "function"
: "object";
```

```
type T0 = TypeName<string>;
// ^
type T1 = TypeName<"a">;
// ^
type T2 = TypeName<true>;
// ^
type T3 = TypeName<() => void>;
// ^
type T4 = TypeName<string[]>;
// ^?
```

But as an example of a place where conditional types are deferred - where they stick around instead of picking a branch - would be in the following:

```
interface Foo {
  propA: boolean;
  propB: boolean;
}

declare function f<T>(x: T): T extends Foo ? string : number;

function foo<U>(x: U) {
  // Has type 'U extends Foo ? string : number'
  let a = f(x);

  // This assignment is allowed though!
  let b: string | number = a;
}
```

In the above, the variable `a` has a conditional type that hasn't yet chosen a branch. When another piece of code ends up calling `foo`, it will substitute in `U` with some other type, and TypeScript will re-evaluate the conditional type, deciding whether it can actually pick a branch.

In the meantime, we can assign a conditional type to any other target type as long as each branch of the conditional is assignable to that target. So in our example above we were able to assign `U extends Foo ? string : number` to `string | number` since no matter what the conditional evaluates to, it's known to be either `string` or `number`.

Distributive conditional types

Conditional types in which the checked type is a naked type parameter are called *distributive conditional types*. Distributive conditional types are automatically distributed over union types during instantiation. For example, an instantiation of `T extends U ? X : Y` with the type argument `A | B | C` for `T` is resolved as `(A extends U ? X : Y) | (B extends U ? X : Y) | (C extends U ? X : Y)`.

Example

```
type TypeName<T> = T extends string
  ? "string"
  : T extends number
  ? "number"
  : T extends boolean
  ? "boolean"
  : T extends undefined
  ? "undefined"
  : T extends Function
  ? "function"
  : "object";
// ---cut---
type T5 = TypeName<string | (() => void)>;
// ^
type T6 = TypeName<string | string[] | undefined>;
// ^
type T7 = TypeName<string[] | number[]>;
// ^?
```

In instantiations of a distributive conditional type `T extends U ? X : Y`, references to `T` within the conditional type are resolved to individual constituents of the union type (i.e. `T` refers to the individual constituents *after* the conditional type is distributed over the union type). Furthermore, references to `T` within `X` have an additional type parameter constraint `U` (i.e. `T` is considered assignable to `U` within `X`).

Example

```
type BoxedValue<T> = { value: T };
type BoxedArray<T> = { array: T[] };
type Boxed<T> = T extends any[] ? BoxedArray<T[number]> : BoxedValue<T>;
type T1 = Boxed<string>;
// ^
type T2 = Boxed<number[]>;
```

```
// ^?
type T3 = Boxed<string | number[]>;
// ^?
```

Notice that `T` has the additional constraint `any[]` within the true branch of `Boxed<T>` and it is therefore possible to refer to the element type of the array as `T[number]`. Also, notice how the conditional type is distributed over the union type in the last example.

The distributive property of conditional types can conveniently be used to *filter* union types:

```
// @errors: 2300 2322
// Remove types from T that are assignable to U
type Diff<T, U> = T extends U ? never : T;
// Remove types from T that are not assignable to U
type Filter<T, U> = T extends U ? T : never;

type T1 = Diff<"a" | "b" | "c" | "d", "a" | "c" | "f">;
// ^
type T2 = Filter<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "a" | "c"
// ^
type T3 = Diff<string | number | ((() => void), Function>; // string | number
// ^
type T4 = Filter<string | number | ((() => void), Function>; // () => void
// ^
// Remove null and undefined from T
type NotNullable<T> = Diff<T, null | undefined>;

type T5 = NotNullable<string | number | undefined>;
// ^
type T6 = NotNullable<string | string[] | null | undefined>;
// ^

function f1<T>(x: T, y: NotNullable<T>) {
  x = y;
  y = x;
}

function f2<T extends string | undefined>(x: T, y: NotNullable<T>) {
  x = y;
  y = x;
  let s1: string = x;
  let s2: string = y;
}
```

Conditional types are particularly useful when combined with mapped types:

```
type FunctionPropertyNames<T> = {
  [K in keyof T]: T[K] extends Function ? K : never;
}[keyof T];
type FunctionProperties<T> = Pick<T, FunctionPropertyNames<T>>;

type NonFunctionPropertyNames<T> = {
  [K in keyof T]: T[K] extends Function ? never : K;
}[keyof T];
type NonFunctionProperties<T> = Pick<T, NonFunctionPropertyNames<T>>;

interface Part {
  id: number;
```

```

    name: string;
    subparts: Part[];
    updatePart(newName: string): void;
}

type T1 = FunctionPropertyNames<Part>;
// ^
type T2 = NonFunctionPropertyNames<Part>;
// ^
type T3 = FunctionProperties<Part>;
// ^
type T4 = NonFunctionProperties<Part>;
// ^

```

Note, conditional types are not permitted to reference themselves recursively. For example the following is an error.

Example

```
// @errors: 2456 2315
type ElementType<T> = T extends any[] ? ElementType<T[number]> : T; // Error
```

Type inference in conditional types

Within the `extends` clause of a conditional type, it is now possible to have `infer` declarations that introduce a type variable to be inferred. Such inferred type variables may be referenced in the true branch of the conditional type. It is possible to have multiple `infer` locations for the same type variable.

For example, the following extracts the return type of a function type:

```
// @noErrors
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;
```

Conditional types can be nested to form a sequence of pattern matches that are evaluated in order:

```

type Unpacked<T> = T extends (infer U)[]
? U
: T extends (...args: any[]) => infer U
? U
: T extends Promise<infer U>
? U
: T;

type T0 = Unpacked<string>;
// ^
type T1 = Unpacked<string[]>;
// ^
type T2 = Unpacked<() => string>;
// ^
type T3 = Unpacked<Promise<string>>;
// ^
type T4 = Unpacked<Promise<string>[]>;
// ^
type T5 = Unpacked<Unpacked<Promise<string>[]>>;
// ^

```

The following example demonstrates how multiple candidates for the same type variable in co-variant positions causes a union type to be inferred:

```
type Foo<T> = T extends { a: infer U; b: infer U } ? U : never;
type T1 = Foo<{ a: string; b: string }>;
// ^
type T2 = Foo<{ a: string; b: number }>;
// ^?
```

Likewise, multiple candidates for the same type variable in contra-variant positions causes an intersection type to be inferred:

```
type Bar<T> = T extends { a: (x: infer U) => void; b: (x: infer U) => void }
? U
: never;

type T1 = Bar<{ a: (x: string) => void; b: (x: string) => void }>;
// ^
type T2 = Bar<{ a: (x: string) => void; b: (x: number) => void }>;
// ^?
```

When inferring from a type with multiple call signatures (such as the type of an overloaded function), inferences are made from the *last* signature (which, presumably, is the most permissive catch-all case). It is not possible to perform overload resolution based on a list of argument types.

```
declare function foo(x: string): number;
declare function foo(x: number): string;
declare function foo(x: string | number): string | number;

type T1 = ReturnType<typeof foo>;
// ^?
```

It is not possible to use `infer` declarations in constraint clauses for regular type parameters:

```
// @errors: 1338 2304
type ReturnedType<T extends (...args: any[]) => infer R> = R;
```

However, much the same effect can be obtained by erasing the type variables in the constraint and instead specifying a conditional type:

```
// @noErrors
type AnyFunction = (...args: any[]) => any;
type ReturnType<T extends AnyFunction> = T extends (...args: any[]) => infer R
? R
: any;
```

Predefined conditional types

TypeScript adds several predefined conditional types, you can find the full list and examples in [Utility Types](#).

[Go to TOC](#)

Introduction

Some of the unique concepts in TypeScript describe the shape of JavaScript objects at the type level. One example that is especially unique to TypeScript is the concept of 'declaration merging'. Understanding this concept will give you an advantage when working with existing JavaScript. It also opens the door to more advanced abstraction concepts.

For the purposes of this article, "declaration merging" means that the compiler merges two separate declarations declared with the same name into a single definition. This merged definition has the features of both of the original declarations. Any number of declarations can be merged; it's not limited to just two declarations.

Basic Concepts

In TypeScript, a declaration creates entities in at least one of three groups: namespace, type, or value. Namespace-creating declarations create a namespace, which contains names that are accessed using a dotted notation. Type-creating declarations do just that: they create a type that is visible with the declared shape and bound to the given name. Lastly, value-creating declarations create values that are visible in the output JavaScript.

Declaration	Type	Namespace	Type	Value
Namespace	X		X	
Class		X	X	
Enum		X	X	
Interface		X		
Type Alias		X		
Function			X	
Variable			X	

Understanding what is created with each declaration will help you understand what is merged when you perform a declaration merge.

Merging Interfaces

The simplest, and perhaps most common, type of declaration merging is interface merging. At the most basic level, the merge mechanically joins the members of both declarations into a single interface with the same name.

```
interface Box {
  height: number;
  width: number;
}
```

```
interface Box {
  scale: number;
}

let box: Box = { height: 5, width: 6, scale: 10 };
```

Non-function members of the interfaces should be unique. If they are not unique, they must be of the same type. The compiler will issue an error if the interfaces both declare a non-function member of the same name, but of different types.

For function members, each function member of the same name is treated as describing an overload of the same function. Of note, too, is that in the case of interface `A` merging with later interface `A`, the second interface will have a higher precedence than the first.

That is, in the example:

```
interface Cloner {
  clone(animal: Animal): Animal;
}

interface Cloner {
  clone(animal: Sheep): Sheep;
}

interface Cloner {
  clone(animal: Dog): Dog;
  clone(animal: Cat): Cat;
}
```

The three interfaces will merge to create a single declaration as so:

```
interface Cloner {
  clone(animal: Dog): Dog;
  clone(animal: Cat): Cat;
  clone(animal: Sheep): Sheep;
  clone(animal: Animal): Animal;
}
```

Notice that the elements of each group maintains the same order, but the groups themselves are merged with later overload sets ordered first.

One exception to this rule is specialized signatures. If a signature has a parameter whose type is a *single* string literal type (e.g. not a union of string literals), then it will be bubbled toward the top of its merged overload list.

For instance, the following interfaces will merge together:

```
interface Document {
  createElement(tagName: any): Element;
}

interface Document {
  createElement(tagName: "div"): HTMLDivElement;
  createElement(tagName: "span"): HTMLSpanElement;
}

interface Document {
```

```
createElement(tagName: string): HTMLElement;
createElement(tagName: "canvas"): HTMLCanvasElement;
}
```

The resulting merged declaration of `Document` will be the following:

```
interface Document {
  createElement(tagName: "canvas"): HTMLCanvasElement;
  createElement(tagName: "div"): HTMLDivElement;
  createElement(tagName: "span"): HTMLSpanElement;
  createElement(tagName: string): HTMLElement;
  createElement(tagName: any): Element;
}
```

Merging Namespaces

Similarly to interfaces, namespaces of the same name will also merge their members. Since namespaces create both a namespace and a value, we need to understand how both merge.

To merge the namespaces, type definitions from exported interfaces declared in each namespace are themselves merged, forming a single namespace with merged interface definitions inside.

To merge the namespace value, at each declaration site, if a namespace already exists with the given name, it is further extended by taking the existing namespace and adding the exported members of the second namespace to the first.

The declaration merge of `Animals` in this example:

```
namespace Animals {
  export class Zebra {}
}

namespace Animals {
  export interface Legged {
    numberOfLegs: number;
  }
  export class Dog {}
}
```

is equivalent to:

```
namespace Animals {
  export interface Legged {
    numberOfLegs: number;
  }

  export class Zebra {}
  export class Dog {}
}
```

This model of namespace merging is a helpful starting place, but we also need to understand what happens with non-exported members. Non-exported members are only visible in the original (un-merged) namespace. This means that after merging, merged members that came from other declarations cannot see non-exported members.

We can see this more clearly in this example:

```
namespace Animal {
  let haveMuscles = true;

  export function animalsHaveMuscles() {
    return haveMuscles;
  }
}

namespace Animal {
  export function doAnimalsHaveMuscles() {
    return haveMuscles; // Error, because haveMuscles is not accessible here
  }
}
```

Because `haveMuscles` is not exported, only the `animalsHaveMuscles` function that shares the same unmerged namespace can see the symbol. The `doAnimalsHaveMuscles` function, even though it's part of the merged `Animal` namespace can not see this un-exported member.

Merging Namespaces with Classes, Functions, and Enums

Namespaces are flexible enough to also merge with other types of declarations. To do so, the namespace declaration must follow the declaration it will merge with. The resulting declaration has properties of both declaration types. TypeScript uses this capability to model some of the patterns in JavaScript as well as other programming languages.

Merging Namespaces with Classes

This gives the user a way of describing inner classes.

```
class Album {
  label: Album.AlbumLabel;
}
namespace Album {
  export class AlbumLabel {}
}
```

The visibility rules for merged members is the same as described in the [Merging Namespaces](#) section, so we must export the `AlbumLabel` class for the merged class to see it. The end result is a class managed inside of another class. You can also use namespaces to add more static members to an existing class.

In addition to the pattern of inner classes, you may also be familiar with the JavaScript practice of creating a function and then extending the function further by adding properties onto the function. TypeScript uses declaration merging to build up definitions like this in a type-safe way.

```
function buildLabel(name: string): string {
  return buildLabel.prefix + name + buildLabel.suffix;
}

namespace buildLabel {
```

```

    export let suffix = "";
    export let prefix = "Hello, ";
}

console.log(buildLabel("Sam Smith"));

```

Similarly, namespaces can be used to extend enums with static members:

```

enum Color {
  red = 1,
  green = 2,
  blue = 4,
}

namespace Color {
  export function mixColor(colorName: string) {
    if (colorName == "yellow") {
      return Color.red + Color.green;
    } else if (colorName == "white") {
      return Color.red + Color.green + Color.blue;
    } else if (colorName == "magenta") {
      return Color.red + Color.blue;
    } else if (colorName == "cyan") {
      return Color.green + Color.blue;
    }
  }
}

```

Disallowed Merges

Not all merges are allowed in TypeScript. Currently, classes can not merge with other classes or with variables. For information on mimicking class merging, see the [Mixins in TypeScript](#) section.

Module Augmentation

Although JavaScript modules do not support merging, you can patch existing objects by importing and then updating them. Let's look at a toy Observable example:

```

// observable.ts
export class Observable<T> {
  // ... implementation left as an exercise for the reader ...
}

// map.ts
import { Observable } from "./observable";
Observable.prototype.map = function (f) {
  // ... another exercise for the reader
};

```

This works fine in TypeScript too, but the compiler doesn't know about `Observable.prototype.map`. You can use module augmentation to tell the compiler about it:

```

// observable.ts
export class Observable<T> {
  // ... implementation left as an exercise for the reader ...
}

```

```
// map.ts
import { Observable } from "./observable";
declare module "./observable" {
  interface Observable<T> {
    map<U>(f: (x: T) => U): Observable<U>;
  }
}
Observable.prototype.map = function (f) {
  // ... another exercise for the reader
};

// consumer.ts
import { Observable } from "./observable";
import "./map";
let o: Observable<number>;
o.map((x) => x.toFixed());
```

The module name is resolved the same way as module specifiers in `import / export`. See [Modules](#) for more information. Then the declarations in an augmentation are merged as if they were declared in the same file as the original.

However, there are two limitations to keep in mind:

1. You can't declare new top-level declarations in the augmentation -- just patches to existing declarations.
2. Default exports also cannot be augmented, only named exports (since you need to augment an export by its exported name, and `default` is a reserved word - see [#14080](#) for details)

Global augmentation

You can also add declarations to the global scope from inside a module:

```
// observable.ts
export class Observable<T> {
  // ... still no implementation ...
}

declare global {
  interface Array<T> {
    toObservable(): Observable<T>;
  }
}

Array.prototype.toObservable = function () {
  // ...
};
```

Global augmentations have the same behavior and limits as module augmentations.

[Go to TOC](#)

Introduction

Further Reading:
[A Complete Guide to TypeScript Decorators](#)

With the introduction of Classes in TypeScript and ES6, there now exist certain scenarios that require additional features to support annotating or modifying classes and class members. Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members. Decorators are a [stage 2 proposal](#) for JavaScript and are available as an experimental feature of TypeScript.

NOTE Decorators are an experimental feature that may change in future releases.

To enable experimental support for decorators, you must enable the `experimentalDecorators` compiler option either on the command line or in your `tsconfig.json`:

Command Line:

```
tsc --target ES5 --experimentalDecorators
```

tsconfig.json:

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true
  }
}
```

Decorators

A *Decorator* is a special kind of declaration that can be attached to a [class declaration](#), [method](#), [accessor](#), [property](#), or [parameter](#). Decorators use the form `@expression`, where `expression` must evaluate to a function that will be called at runtime with information about the decorated declaration.

For example, given the decorator `@sealed` we might write the `sealed` function as follows:

```
function sealed(target) {
  // do something with 'target' ...
}
```

Decorator Factories

If we want to customize how a decorator is applied to a declaration, we can write a decorator factory. A *Decorator Factory* is simply a function that returns the expression that will be called by the decorator at runtime.

We can write a decorator factory in the following fashion:

```
function color(value: string) {
  // this is the decorator factory, it sets up
  // the returned decorator function
  return function (target) {
    // this is the decorator
    // do something with 'target' and 'value'...
  };
}
```

Decorator Composition

Multiple decorators can be applied to a declaration, for example on a single line:

```
// @experimentalDecorators
// @noErrors
function f() {}
function g() {}
// ---cut---
@f @g x
```

On multiple lines:

```
// @experimentalDecorators
// @noErrors
function f() {}
function g() {}
// ---cut---
@f
@g
x
```

When multiple decorators apply to a single declaration, their evaluation is similar to [function composition in mathematics](#). In this model, when composing functions f and g , the resulting composite $(f \cdot g)(x)$ is equivalent to $f(g(x))$.

As such, the following steps are performed when evaluating multiple decorators on a single declaration in TypeScript:

1. The expressions for each decorator are evaluated top-to-bottom.
2. The results are then called as functions from bottom-to-top.

If we were to use [decorator factories](#), we can observe this evaluation order with the following example:

```
// @experimentalDecorators
function first() {
    console.log("first(): factory evaluated");
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("first(): called");
    };
}

function second() {
    console.log("second(): factory evaluated");
    return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
        console.log("second(): called");
    };
}

class ExampleClass {
    @first()
    @second()
    method() {}
}
```

Which would print this output to the console:

```
first(): factory evaluated
second(): factory evaluated
second(): called
first(): called
```

Decorator Evaluation

There is a well defined order to how decorators applied to various declarations inside of a class are applied:

1. *Parameter Decorators*, followed by *Method, Accessor, or Property Decorators* are applied for each instance member.
2. *Parameter Decorators*, followed by *Method, Accessor, or Property Decorators* are applied for each static member.
3. *Parameter Decorators* are applied for the constructor.
4. *Class Decorators* are applied for the class.

Class Decorators

A *Class Decorator* is declared just before a class declaration. The class decorator is applied to the constructor of the class and can be used to observe, modify, or replace a class definition. A class decorator cannot be used in a declaration file, or in any other ambient context (such as on a `declare class`).

The expression for the class decorator will be called as a function at runtime, with the constructor of the decorated class as its only argument.

If the class decorator returns a value, it will replace the class declaration with the provided constructor function.

NOTE Should you choose to return a new constructor function, you must take care to maintain the original prototype. The logic that applies decorators at runtime will **not** do this for you.

The following is an example of a class decorator (`@sealed`) applied to a `BugReport` class:

```
// @experimentalDecorators
function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}
// ---cut---
@sealed
class BugReport {
  type = "report";
  title: string;

  constructor(t: string) {
    this.title = t;
  }
}
```

We can define the `@sealed` decorator using the following function declaration:

```
function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}
```

When `@sealed` is executed, it will seal both the constructor and its prototype, and will therefore prevent any further functionality from being added to or removed from this class during runtime by accessing `BugReport.prototype` or by defining properties on `BugReport` itself (note that ES2015 classes are really just syntactic sugar to prototype-based constructor functions). This decorator does **not** prevent classes from sub-classing `BugReport`.

Next we have an example of how to override the constructor to set new defaults.

```
// @errors: 2339
// @experimentalDecorators
function reportableClassDecorator<T extends { new (...args: any[]): {} }>(
  constructor: T) {
  return class extends constructor {
    reportingURL = "http://www...";
  };
}

@reportableClassDecorator
class BugReport {
  type = "report";
  title: string;

  constructor(t: string) {
    this.title = t;
  }
}
```

```
const bug = new BugReport("Needs dark mode");
console.log(bug.title); // Prints "Needs dark mode"
console.log(bug.type); // Prints "report"

// Note that the decorator _does not_ change the TypeScript type
// and so the new property `reportingURL` is not known
// to the type system:
bug.reportingURL;
```

Method Decorators

A *Method Decorator* is declared just before a method declaration. The decorator is applied to the *Property Descriptor* for the method, and can be used to observe, modify, or replace a method definition. A method decorator cannot be used in a declaration file, on an overload, or in any other ambient context (such as in a `declare class`).

The expression for the method decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The *Property Descriptor* for the member.

NOTE The *Property Descriptor* will be `undefined` if your script target is less than `ES5`.

If the method decorator returns a value, it will be used as the *Property Descriptor* for the method.

NOTE The return value is ignored if your script target is less than `ES5`.

The following is an example of a method decorator (`@enumerable`) applied to a method on the `Greeter` class:

```
// @experimentalDecorators
function enumerable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    descriptor.enumerable = value;
  };
}
// ---cut---
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  @enumerable(false)
  greet() {
```

```

    return "Hello, " + this.greeting;
}
}

```

We can define the `@enumerable` decorator using the following function declaration:

```

function enumerable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor:
    PropertyDescriptor) {
    descriptor.enumerable = value;
  };
}

```

The `@enumerable(false)` decorator here is a [decorator factory](#). When the `@enumerable(false)` decorator is called, it modifies the `enumerable` property of the property descriptor.

Accessor Decorators

An *Accessor Decorator* is declared just before an accessor declaration. The accessor decorator is applied to the *Property Descriptor* for the accessor and can be used to observe, modify, or replace an accessor's definitions. An accessor decorator cannot be used in a declaration file, or in any other ambient context (such as in a `declare` class).

NOTE TypeScript disallows decorating both the `get` and `set` accessor for a single member. Instead, all decorators for the member must be applied to the first accessor specified in document order. This is because decorators apply to a *Property Descriptor*, which combines both the `get` and `set` accessor, not each declaration separately.

The expression for the accessor decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The *Property Descriptor* for the member.

NOTE The *Property Descriptor* will be `undefined` if your script target is less than `ES5`.

If the accessor decorator returns a value, it will be used as the *Property Descriptor* for the member.

NOTE The return value is ignored if your script target is less than `ES5`.

The following is an example of an accessor decorator (`@configurable`) applied to a member of the `Point` class:

```
// @experimentalDecorators
function configurable(value: boolean) {
  return function (
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    descriptor.configurable = value;
  };
}
// ---cut---
class Point {
  private _x: number;
  private _y: number;
  constructor(x: number, y: number) {
    this._x = x;
    this._y = y;
  }
  @configurable(false)
  get x() {
    return this._x;
  }
  @configurable(false)
  get y() {
    return this._y;
  }
}
```

We can define the `@configurable` decorator using the following function declaration:

```
function configurable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    descriptor.configurable = value;
  };
}
```

Property Decorators

A *Property Decorator* is declared just before a property declaration. A property decorator cannot be used in a declaration file, or in any other ambient context (such as in a `declare` class).

The expression for the property decorator will be called as a function at runtime, with the following two arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.

NOTE A *Property Descriptor* is not provided as an argument to a property decorator due to how property decorators are initialized in TypeScript. This is because there is currently no mechanism to describe an instance property when defining members of a prototype, and no way to observe or modify the initializer for a property. The return value is ignored too. As such, a property decorator can only be used to observe that a property of a specific name has been declared for a class.

We can use this information to record metadata about the property, as in the following example:

```
class Greeter {
  @format("Hello, %s")
  greeting: string;

  constructor(message: string) {
    this.greeting = message;
  }

  greet() {
    let formatString = getFormat(this, "greeting");
    return formatString.replace("%s", this.greeting);
  }
}
```

We can then define the `@format` decorator and `getFormat` functions using the following function declarations:

```
import "reflect-metadata";

const formatMetadataKey = Symbol("format");

function format(formatString: string) {
  return Reflect.metadata(formatMetadataKey, formatString);
}

function getFormat(target: any, propertyKey: string) {
  return Reflect.getMetadata(formatMetadataKey, target, propertyKey);
}
```

The `@format("Hello, %s")` decorator here is a [decorator factory](#). When `@format("Hello, %s")` is called, it adds a metadata entry for the property using the `Reflect.metadata` function from the `reflect-metadata` library. When `getFormat` is called, it reads the metadata value for the format.

NOTE This example requires the `reflect-metadata` library. See [Metadata](#) for more information about the `reflect-metadata` library.

Parameter Decorators

A *Parameter Decorator* is declared just before a parameter declaration. The parameter decorator is applied to the function for a class constructor or method declaration. A parameter decorator cannot be used in a declaration file, an overload, or in any other ambient context (such as in a `declare` class).

The expression for the parameter decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The ordinal index of the parameter in the function's parameter list.

NOTE A parameter decorator can only be used to observe that a parameter has been declared on a method.

The return value of the parameter decorator is ignored.

The following is an example of a parameter decorator (`@required`) applied to parameter of a member of the `BugReport` class:

```
// @experimentalDecorators
function validate(target: any, propertyName: string, descriptor:
TypedPropertyDescriptor<any>) {}
function required(target: Object, propertyKey: string | symbol, parameterIndex:
number) {}
// ---cut---
class BugReport {
  type = "report";
  title: string;

  constructor(t: string) {
    this.title = t;
  }

  @validate
  print(@required verbose: boolean) {
    if (verbose) {
      return `type: ${this.type}\n${this.title}`;
    } else {
      return this.title;
    }
  }
}
```

We can then define the `@required` and `@validate` decorators using the following function declarations:

```
// @experimentalDecorators
// @emitDecoratorMetadata
import "reflect-metadata";
const requiredMetadataKey = Symbol("required");
```

```

function required(target: Object, propertyKey: string | symbol, parameterIndex: number) {
  let existingRequiredParameters: number[] =
    Reflect.getOwnMetadata(requiredMetadataKey, target, propertyKey) || [];
  existingRequiredParameters.push(parameterIndex);
  Reflect.defineMetadata(requiredMetadataKey, existingRequiredParameters, target,
    propertyKey);
}

function validate(target: any, propertyName: string, descriptor: TypedPropertyDescriptor<Function>) {
  let method = descriptor.value!;

  descriptor.value = function () {
    let requiredParameters: number[] = Reflect.getOwnMetadata(requiredMetadataKey,
      target, propertyName);
    if (requiredParameters) {
      for (let parameterIndex of requiredParameters) {
        if (parameterIndex >= arguments.length || arguments[parameterIndex] ===
        undefined) {
          throw new Error("Missing required argument.");
        }
      }
    }
    return method.apply(this, arguments);
  };
}

```

The `@required` decorator adds a metadata entry that marks the parameter as required. The `@validate` decorator then wraps the existing `greet` method in a function that validates the arguments before invoking the original method.

NOTE This example requires the `reflect-metadata` library. See [Metadata](#) for more information about the `reflect-metadata` library.

Metadata

Some examples use the `reflect-metadata` library which adds a polyfill for an [experimental metadata API](#). This library is not yet part of the ECMAScript (JavaScript) standard. However, once decorators are officially adopted as part of the ECMAScript standard these extensions will be proposed for adoption.

You can install this library via npm:

```
npm i reflect-metadata --save
```

TypeScript includes experimental support for emitting certain types of metadata for declarations that have decorators. To enable this experimental support, you must set the `emitDecoratorMetadata` compiler option either on the command line or in your `tsconfig.json`:

Command Line:

```
tsc --target ES5 --experimentalDecorators --emitDecoratorMetadata
```

tsconfig.json:

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

When enabled, as long as the `reflect-metadata` library has been imported, additional design-time type information will be exposed at runtime.

We can see this in action in the following example:

```
// @emitDecoratorMetadata
// @experimentalDecorators
// @strictPropertyInitialization: false
import "reflect-metadata";

class Point {
  constructor(public x: number, public y: number) {}
}

class Line {
  private _start: Point;
  private _end: Point;

  @validate
  set start(value: Point) {
    this._start = value;
  }

  get start() {
    return this._start;
  }

  @validate
  set end(value: Point) {
    this._end = value;
  }

  get end() {
    return this._end;
  }
}

function validate<T>(target: any, propertyKey: string, descriptor: TypedPropertyDescriptor<T>) {
  let set = descriptor.set!;

  descriptor.set = function (value: T) {
    let type = Reflect.getMetadata("design:type", target, propertyKey);

    if (!(value instanceof type)) {
      throw new TypeError(`Invalid type, got ${typeof value} not ${type.name}`);
    }

    set.call(this, value);
  };
}
```

```

}

const line = new Line()
line.start = new Point(0, 0)
// @ts-ignore
// line.end = {}

// Fails at runtime with:
// > Invalid type, got object not Point

```

The TypeScript compiler will inject design-time type information using the `@Reflect.metadata` decorator. You could consider it the equivalent of the following TypeScript:

```

class Line {
    private _start: Point;
    private _end: Point;

    @validate
    @Reflect.metadata("design:type", Point)
    set start(value: Point) {
        this._start = value;
    }
    get start() {
        return this._start;
    }

    @validate
    @Reflect.metadata("design:type", Point)
    set end(value: Point) {
        this._end = value;
    }
    get end() {
        return this._end;
    }
}

```

NOTE Decorator metadata is an experimental feature and may introduce breaking changes in future releases.

[Go to TOC](#)

For the last few years, Node.js has been working to support running ECMAScript modules (ESM). This has been a very difficult feature to support, since the foundation of the Node.js ecosystem is built on a different module system called CommonJS (CJS).

Interoperating between the two module systems brings large challenges, with many new features to juggle; however, support for ESM in Node.js is now implemented in Node.js, and the dust has begun to settle.

That's why TypeScript brings two new `module` and `moduleResolution` settings: `node16` and `nodenext`.

```
{
  "compilerOptions": {
    "module": "nodenext",
  }
}
```

These new modes bring a few high-level features which we'll explore here.

`type` in `package.json` and New Extensions

Node.js supports a new setting in `package.json` called `type`. `"type"` can be set to either `"module"` or `"commonjs"`.

```
{
  "name": "my-package",
  "type": "module",

  "//": "...",
  "dependencies": {
  }
}
```

This setting controls whether `.js` files are interpreted as ES modules or CommonJS modules, and defaults to CommonJS when not set. When a file is considered an ES module, a few different rules come into play compared to CommonJS:

- `import` / `export` statements and top-level `await` can be used
- relative import paths need full extensions (e.g. we have to write `import "./foo.js"` instead of `import "./foo"`)
- imports might resolve differently from dependencies in `node_modules`
- certain global-like values like `require()` and `__dirname` cannot be used directly
- CommonJS modules get imported under certain special rules

We'll come back to some of these.

To overlay the way TypeScript works in this system, `.ts` and `.tsx` files now work the same way. When TypeScript finds a `.ts`, `.tsx`, `.js`, or `.jsx` file, it will walk up looking for a `package.json` to see whether that file is an ES module, and use that to determine:

- how to find other modules which that file imports
- and how to transform that file if producing outputs

When a `.ts` file is compiled as an ES module, ECMAScript `import / export` syntax is left alone in the `.js` output; when it's compiled as a CommonJS module, it will produce the same output you get today under `module: commonjs`.

This also means paths resolve differently between `.ts` files that are ES modules and ones that are CJS modules. For example, let's say you have the following code today:

```
// ./foo.ts
export function helper() {
  // ...
}

// ./bar.ts
import { helper } from "./foo"; // only works in CJS
helper();
```

This code works in CommonJS modules, but will fail in ES modules because relative import paths need to use extensions. As a result, it will have to be rewritten to use the extension of the *output* of `foo.ts` - so `bar.ts` will instead have to import from `./foo.js`.

```
// ./bar.ts
import { helper } from "./foo.js"; // works in ESM & CJS
helper();
```

This might feel a bit cumbersome at first, but TypeScript tooling like auto-imports and path completion will typically just do this for you.

One other thing to mention is the fact that this applies to `.d.ts` files too. When TypeScript finds a `.d.ts` file in package, whether it is treated as an ESM or CommonJS file is based on the containing package.

New File Extensions

The `type` field in `package.json` is nice because it allows us to continue using the `.ts` and `.js` file extensions which can be convenient; however, you will occasionally need to write a file that differs from what `type` specifies. You might also just prefer to always be explicit.

Node.js supports two extensions to help with this: `.mjs` and `.cjs`. `.mjs` files are always ES modules, and `.cjs` files are always CommonJS modules, and there's no way to override these.

In turn, TypeScript supports two new source file extensions: `.mts` and `.cts`. When TypeScript emits these to JavaScript files, it will emit them to `.mjs` and `.cjs` respectively.

Furthermore, TypeScript also supports two new declaration file extensions: `.d.mts` and `.d.cts`. When TypeScript generates declaration files for `.mts` and `.cts`, their corresponding extensions will be `.d.mts` and `.d.cts`.

Using these extensions is entirely optional, but will often be useful even if you choose not to use them as part of your primary workflow.

CommonJS Interop

Node.js allows ES modules to import CommonJS modules as if they were ES modules with a default export.

```
// @module: nodenext
// @filename: helper.cts
export function helper() {
    console.log("hello world!");
}

// @filename: index.mts
import foo from "./helper.cjs";

// prints "hello world!"
foo.helper();
```

In some cases, Node.js also synthesizes named exports from CommonJS modules, which can be more convenient. In these cases, ES modules can use a "namespace-style" import (i.e. `import * as foo from "..."`), or named imports (i.e. `import { helper } from "..."`).

```
// @module: nodenext
// @filename: helper.cts
export function helper() {
    console.log("hello world!");
}

// @filename: index.mts
import { helper } from "./helper.cjs";

// prints "hello world!"
helper();
```

There isn't always a way for TypeScript to know whether these named imports will be synthesized, but TypeScript will err on being permissive and use some heuristics when importing from a file that is definitely a CommonJS module.

One TypeScript-specific note about interop is the following syntax:

```
import foo = require("foo");
```

In a CommonJS module, this just boils down to a `require()` call, and in an ES module, this imports `createRequire` to achieve the same thing. This will make code less portable on runtimes like the browser (which don't support `require()`), but will often be useful for interoperability. In turn, you can write the above example using this syntax as follows:

```
// @module: nodenext
// @filename: helper.cts
export function helper() {
    console.log("hello world!");
}

// @filename: index.mts
import foo = require("./foo.cjs");

foo.helper()
```

Finally, it's worth noting that the only way to import ESM files from a CJS module is using dynamic `import()` calls. This can present challenges, but is the behavior in Node.js today.

You can [read more about ESM/CommonJS interop in Node.js here](#).

`package.json` Exports, Imports, and Self-Referencing

Node.js supports a new field for defining entry points in `package.json` called "exports". This field is a more powerful alternative to defining `"main"` in `package.json`, and can control what parts of your package are exposed to consumers.

Here's an `package.json` that supports separate entry-points for CommonJS and ESM:

```
// package.json
{
  "name": "my-package",
  "type": "module",
  "exports": {
    ".": {
      // Entry-point for `import "my-package"` in ESM
      "import": "./esm/index.js",

      // Entry-point for `require("my-package")` in CJS
      "require": "./commonjs/index.cjs",
    },
  },
  // CJS fall-back for older versions of Node.js
  "main": "./commonjs/index.cjs",
}
```

There's a lot to this feature, [which you can read more about on the Node.js documentation](#). Here we'll try to focus on how TypeScript supports it.

With TypeScript's original Node support, it would look for a `"main"` field, and then look for declaration files that corresponded to that entry. For example, if `"main"` pointed to `./lib/index.js`, TypeScript would look for a file called `./lib/index.d.ts`. A package author could override this by specifying a separate field called `"types"` (e.g. `"types": "./types/index.d.ts"`).

The new support works similarly with [import conditions](#). By default, TypeScript overlays the same rules with import conditions - if you write an `import` from an ES module, it will look up the `import` field, and from a CommonJS module, it will look at the `require` field. If it finds them, it will look for a colocated declaration file. If you need to point to a different location for your type declarations, you can add a `"types"` import condition.

```
// package.json
{
  "name": "my-package",
  "type": "module",
  "exports": {
    ".": {
      // Entry-point for TypeScript resolution - must occur first!
      "types": "./types/index.d.ts",
    },
  },
}
```

```
// Entry-point for `import "my-package"` in ESM
"import": "./esm/index.js",
// Entry-point for `require("my-package")` in CJS
"require": "./commonjs/index.cjs",
},
// CJS fall-back for older versions of Node.js
"main": "./commonjs/index.cjs",
// Fall-back for older versions of TypeScript
"types": "./types/index.d.ts"
}
```

TypeScript also supports the `"imports"` field of `package.json` in a similar manner (looking for declaration files alongside corresponding files), and supports packages self-referencing themselves. These features are generally not as involved, but are supported.

[Go to TOC](#)

Enums are one of the few features TypeScript has which is not a type-level extension of JavaScript.

Enums allow a developer to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases. TypeScript provides both numeric and string-based enums.

Numeric enums

We'll first start off with numeric enums, which are probably more familiar if you're coming from other languages. An enum can be defined using the `enum` keyword.

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right,
}
```

Above, we have a numeric enum where `Up` is initialized with `1`. All of the following members are auto-incremented from that point on. In other words, `Direction.Up` has the value `1`, `Down` has `2`, `Left` has `3`, and `Right` has `4`.

If we wanted, we could leave off the initializers entirely:

```
enum Direction {
  Up,
  Down,
  Left,
  Right,
}
```

Here, `Up` would have the value `0`, `Down` would have `1`, etc. This auto-incrementing behavior is useful for cases where we might not care about the member values themselves, but do care that each value is distinct from other values in the same enum.

Using an enum is simple: just access any member as a property off of the enum itself, and declare types using the name of the enum:

```
enum UserResponse {
  No = 0,
  Yes = 1,
}

function respond(recipient: string, message: UserResponse): void {
  // ...
}

respond("Princess Caroline", UserResponse.Yes);
```

Numeric enums can be mixed in [computed and constant members \(see below\)](#). The short story is, enums without initializers either need to be first, or have to come after numeric enums initialized with numeric constants or other constant enum members. In other words, the following isn't allowed:

```
// @errors: 1061
const getValue = () => 23;
// ---cut---
enum E {
  A = getValue(),
  B,
}
```

String enums

String enums are a similar concept, but have some subtle [runtime differences](#) as documented below. In a string enum, each member has to be constant-initialized with a string literal, or with another string enum member.

```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT",
}
```

While string enums don't have auto-incrementing behavior, string enums have the benefit that they "serialize" well. In other words, if you were debugging and had to read the runtime value of a numeric enum, the value is often opaque - it doesn't convey any useful meaning on its own (though [reverse mapping](#) can often help). String enums allow you to give a meaningful and readable value when your code runs, independent of the name of the enum member itself.

Heterogeneous enums

Technically enums can be mixed with string and numeric members, but it's not clear why you would ever want to do so:

```
enum BooleanLikeHeterogeneousEnum {
  No = 0,
  Yes = "YES",
}
```

Unless you're really trying to take advantage of JavaScript's runtime behavior in a clever way, it's advised that you don't do this.

Computed and constant members

Each enum member has a value associated with it which can be either *constant* or *computed*. An enum member is considered constant if:

- It is the first member in the enum and it has no initializer, in which case it's assigned the value `0`:

```
// E.X is constant:
enum E {
  X,
}
```

- It does not have an initializer and the preceding enum member was a *numeric* constant. In this case the value of the current enum member will be the value of the preceding enum member plus one.

```
// All enum members in 'E1' and 'E2' are constant.

enum E1 {
  X,
  Y,
  Z,
}

enum E2 {
  A = 1,
  B,
  C,
}
```

- The enum member is initialized with a constant enum expression. A constant enum expression is a subset of TypeScript expressions that can be fully evaluated at compile time. An expression is a constant enum expression if it is:

1. a literal enum expression (basically a string literal or a numeric literal)
2. a reference to previously defined constant enum member (which can originate from a different enum)
3. a parenthesized constant enum expression
4. one of the `+`, `-`, `~` unary operators applied to constant enum expression
5. `+, -, *, /, %, <<, >>, >>>, &, |, ^` binary operators with constant enum expressions as operands

It is a compile time error for constant enum expressions to be evaluated to `Nan` or `Infinity`.

In all other cases enum member is considered computed.

```
enum FileAccess {
  // constant members
  None,
  Read = 1 << 1,
  Write = 1 << 2,
  ReadWrite = Read | Write,
  // computed member
  G = "123".length,
}
```

Union enums and enum member types

There is a special subset of constant enum members that aren't calculated: literal enum members. A literal enum member is a constant enum member with no initialized value, or with values that are initialized to

- any string literal (e.g. `"foo"`, `"bar"`, `"baz"`)
- any numeric literal (e.g. `1`, `100`)
- a unary minus applied to any numeric literal (e.g. `-1`, `-100`)

When all members in an enum have literal enum values, some special semantics come into play.

The first is that enum members also become types as well! For example, we can say that certain members can *only* have the value of an enum member:

```
// @errors: 2322
enum ShapeKind {
  Circle,
  Square,
}

interface Circle {
  kind: ShapeKind.Circle;
  radius: number;
}

interface Square {
  kind: ShapeKind.Square;
  sideLength: number;
}

let c: Circle = {
  kind: ShapeKind.Square,
  radius: 100,
};
```

The other change is that enum types themselves effectively become a *union* of each enum member. With union enums, the type system is able to leverage the fact that it knows the exact set of values that exist in the enum itself. Because of that, TypeScript can catch bugs where we might be comparing values incorrectly. For example:

```
// @errors: 2367
enum E {
  Foo,
  Bar,
}

function f(x: E) {
  if (x !== E.Foo || x !== E.Bar) {
    //
  }
}
```

In that example, we first checked whether `x` was *not* `E.Foo`. If that check succeeds, then our `||` will short-circuit, and the body of the 'if' will run. However, if the check didn't succeed, then `x` can *only* be `E.Foo`, so it doesn't make sense to see whether it's equal to `E.Bar`.

Enums at runtime

Enums are real objects that exist at runtime. For example, the following enum

```
enum E {
  X,
  Y,
  Z,
}
```

can actually be passed around to functions

```
enum E {
  X,
  Y,
  Z,
}

function f(obj: { X: number }) {
  return obj.X;
}

// Works, since 'E' has a property named 'X' which is a number.
f(E);
```

Enums at compile time

Even though Enums are real objects that exist at runtime, the `keyof` keyword works differently than you might expect for typical objects. Instead, use `keyof typeof` to get a Type that represents all Enum keys as strings.

```
enum LogLevel {
  ERROR,
  WARN,
  INFO,
  DEBUG,
}

/**
 * This is equivalent to:
 * type LogLevelStrings = 'ERROR' | 'WARN' | 'INFO' | 'DEBUG';
 */
type LogLevelStrings = keyof typeof LogLevel;

function printImportant(key: LogLevelStrings, message: string) {
  const num = LogLevel[key];
  if (num <= LogLevel.WARN) {
    console.log("Log level key is:", key);
    console.log("Log level value is:", num);
    console.log("Log level message is:", message);
  }
}
printImportant("ERROR", "This is a message");
```

Reverse mappings

In addition to creating an object with property names for members, numeric enum members also get a *reverse mapping* from enum values to enum names. For example, in this example:

```
enum Enum {
  A,
}

let a = Enum.A;
let nameOfA = Enum[a]; // "A"
```

TypeScript compiles this down to the following JavaScript:

```
// @showEmit
enum Enum {
  A,
}

let a = Enum.A;
let nameOfA = Enum[a]; // "A"
```

In this generated code, an enum is compiled into an object that stores both forward (`name -> value`) and reverse (`value -> name`) mappings. References to other enum members are always emitted as property accesses and never inlined.

Keep in mind that string enum members *do not* get a reverse mapping generated at all.

const enums

In most cases, enums are a perfectly valid solution. However sometimes requirements are tighter. To avoid paying the cost of extra generated code and additional indirection when accessing enum values, it's possible to use `const` enums. Const enums are defined using the `const` modifier on our enums:

```
const enum Enum {
  A = 1,
  B = A * 2,
}
```

Const enums can only use constant enum expressions and unlike regular enums they are completely removed during compilation. Const enum members are inlined at use sites. This is possible since const enums cannot have computed members.

```
const enum Direction {
  Up,
  Down,
  Left,
  Right,
}

let directions = [
  Direction.Up,
  Direction.Down,
  Direction.Left,
  Direction.Right,
];
```

in generated code will become

```
// @showEmit
const enum Direction {
  Up,
  Down,
  Left,
  Right,
}
```

```
let directions = [
  Direction.Up,
  Direction.Down,
  Direction.Left,
  Direction.Right,
];
```

Const enum pitfalls

Inlining enum values is straightforward at first, but comes with subtle implications. These pitfalls pertain to *ambient* const enums only (basically const enums in `.d.ts` files) and sharing them between projects, but if you are publishing or consuming `.d.ts` files, these pitfalls likely apply to you, because `tsc --declaration` transforms `.ts` files into `.d.ts` files.

1. For the reasons laid out in the [isolatedModules documentation](#), that mode is fundamentally incompatible with ambient const enums. This means if you publish ambient const enums, downstream consumers will not be able to use `isolatedModules` and those enum values at the same time.
2. You can easily inline values from version A of a dependency at compile time, and import version B at runtime. Version A and B's enums can have different values, if you are not very careful, resulting in [surprising bugs](#), like taking the wrong branches of `if` statements. These bugs are especially pernicious because it is common to run automated tests at roughly the same time as projects are built, with the same dependency versions, which misses these bugs completely.
3. `importsNotUsedAsValues: "preserve"` will not elide imports for const enums used as values, but ambient const enums do not guarantee that runtime `.js` files exist. The unresolvable imports cause errors at runtime. The usual way to unambiguously elide imports, [type-only imports](#), does not allow const enum values, currently.

Here are two approaches to avoiding these pitfalls:

A. Do not use const enums at all. You can easily [ban const enums](#) with the help of a linter. Obviously this avoids any issues with const enums, but prevents your project from inlining its own enums. Unlike inlining enums from other projects, inlining a project's own enums is not problematic and has performance implications. B. Do not publish ambient const enums, by deconstifying them with the help of `preserveConstEnums`. This is the approach taken internally by the [TypeScript project itself](#). `preserveConstEnums` emits the same JavaScript for const enums as plain enums. You can then safely strip the `const` modifier from `.d.ts` files in a build step.

This way downstream consumers will not inline enums from your project, avoiding the pitfalls above, but a project can still inline its own enums, unlike banning const enums entirely.

Ambient enums

Ambient enums are used to describe the shape of already existing enum types.

```
declare enum Enum {
  A = 1,
  B,
  C = 2,
}
```

One important difference between ambient and non-ambient enums is that, in regular enums, members that don't have an initializer will be considered constant if its preceding enum member is considered constant. By contrast, an ambient (and non-const) enum member that does not have an initializer is *always* considered computed.

Objects vs Enums

In modern TypeScript, you may not need an enum when an object with `as const` could suffice:

```
const enum EDirection {
  Up,
  Down,
  Left,
  Right,
}

const ODirection = {
  Up: 0,
  Down: 1,
  Left: 2,
  Right: 3,
} as const;

EDirection.Up;
//           ^?

ODirection.Up;
//           ^?

// Using the enum as a parameter
function walk(dir: EDirection) {}

// It requires an extra line to pull out the values
type Direction = typeof ODirection[keyof typeof ODirection];
function run(dir: Direction) {}

walk(EDirection.Left);
run(ODirection.Right);
```

The biggest argument in favour of this format over TypeScript's `enum` is that it keeps your codebase aligned with the state of JavaScript, and `when/if` enums are added to JavaScript then you can move to the additional syntax.

[Go to TOC](#)

Iterables

An object is deemed iterable if it has an implementation for the `Symbol.iterator` property. Some built-in types like `Array`, `Map`, `Set`, `String`, `Int32Array`, `Uint32Array`, etc. have their `Symbol.iterator` property already implemented. `Symbol.iterator` function on an object is responsible for returning the list of values to iterate on.

Iterable interface

`Iterable` is a type we can use if we want to take in types listed above which are iterable. Here is an example:

```
function toArray<X>(xs: Iterable<X>): X[] {
  return [...xs]
}
```

for..of statements

`for..of` loops over an iterable object, invoking the `Symbol.iterator` property on the object. Here is a simple `for..of` loop on an array:

```
let someArray = [1, "string", false];
for (let entry of someArray) {
  console.log(entry); // 1, "string", false
}
```

for..of vs. for..in statements

Both `for..of` and `for..in` statements iterate over lists; the values iterated on are different though, `for..in` returns a list of *keys* on the object being iterated, whereas `for..of` returns a list of *values* of the numeric properties of the object being iterated.

Here is an example that demonstrates this distinction:

```
let list = [4, 5, 6];
for (let i in list) {
  console.log(i); // "0", "1", "2",
}
for (let i of list) {
  console.log(i); // 4, 5, 6
}
```

Another distinction is that `for..in` operates on any object; it serves as a way to inspect properties on this object. `for..of` on the other hand, is mainly interested in values of iterable objects. Built-in objects like `Map` and `Set` implement `Symbol.iterator` property allowing access to stored values.

```
let pets = new Set(["Cat", "Dog", "Hamster"]);
pets["species"] = "mammals";

for (let pet in pets) {
  console.log(pet); // "species"
}

for (let pet of pets) {
  console.log(pet); // "Cat", "Dog", "Hamster"
}
```

Code generation

Targeting ES5 and ES3

When targeting an ES5 or ES3-compliant engine, iterators are only allowed on values of `Array` type. It is an error to use `for..of` loops on non-`Array` values, even if these non-`Array` values implement the `Symbol.iterator` property.

The compiler will generate a simple `for` loop for a `for..of` loop, for instance:

```
let numbers = [1, 2, 3];
for (let num of numbers) {
  console.log(num);
}
```

will be generated as:

```
var numbers = [1, 2, 3];
for (var _i = 0; _i < numbers.length; _i++) {
  var num = numbers[_i];
  console.log(num);
}
```

Targeting ECMAScript 2015 and higher

When targeting an ECMAScript 2015-compliant engine, the compiler will generate `for..of` loops to target the built-in iterator implementation in the engine.

[Go to TOC](#)

`JSX` is an embeddable XML-like syntax. It is meant to be transformed into valid JavaScript, though the semantics of that transformation are implementation-specific. JSX rose to popularity with the `React` framework, but has since seen other implementations as well. TypeScript supports embedding, type checking, and compiling JSX directly to JavaScript.

Basic usage

In order to use JSX you must do two things.

1. Name your files with a `.tsx` extension
2. Enable the `jsx` option

TypeScript ships with three JSX modes: `preserve`, `react`, and `react-native`. These modes only affect the emit stage - type checking is unaffected. The `preserve` mode will keep the JSX as part of the output to be further consumed by another transform step (e.g. `Babel`). Additionally the output will have a `.jsx` file extension. The `react` mode will emit `React.createElement`, does not need to go through a JSX transformation before use, and the output will have a `.js` file extension. The `react-native` mode is the equivalent of `preserve` in that it keeps all JSX, but the output will instead have a `.js` file extension.

Mode	Input	Output	Output File Extension
<code>preserve</code>	<code><div></div></code>	<code><div /></code>	<code>.jsx</code>
<code>react</code>	<code><div></div></code>	<code>React.createElement("div")</code>	<code>.js</code>
<code>react-native</code>	<code><div></div></code>	<code><div /></code>	<code>.js</code>
<code>react-jsx</code>	<code><div></div></code>	<code>_jsx("div", {}, void 0);</code>	<code>.js</code>
<code>react-jsxdev</code>	<code><div></div></code>	<code>_jsxDEV("div", {}, void 0, false, {...}, this);</code>	<code>.js</code>

You can specify this mode using either the `jsx` command line flag or the corresponding option `jsx` in your `tsconfig.json` file.

*Note: You can specify the JSX factory function to use when targeting react JSX emit with `jsxFactory` option (defaults to `React.createElement`)

The `as` operator

Recall how to write a type assertion:

```
const foo = <foo>bar;
```

This asserts the variable `bar` to have the type `foo`. Since TypeScript also uses angle brackets for type assertions, combining it with JSX's syntax would introduce certain parsing difficulties. As a result, TypeScript disallows angle bracket type assertions in `.tsx` files.

Since the above syntax cannot be used in `.tsx` files, an alternate type assertion operator should be used: `as`. The example can easily be rewritten with the `as` operator.

```
const foo = bar as foo;
```

The `as` operator is available in both `.ts` and `.tsx` files, and is identical in behavior to the angle-bracket type assertion style.

Type Checking

In order to understand type checking with JSX, you must first understand the difference between intrinsic elements and value-based elements. Given a JSX expression `<expr />`, `expr` may either refer to something intrinsic to the environment (e.g. a `div` or `span` in a DOM environment) or to a custom component that you've created. This is important for two reasons:

1. For React, intrinsic elements are emitted as strings (`React.createElement("div")`), whereas a component you've created is not (`React.createElement(MyComponent)`).
2. The types of the attributes being passed in the JSX element should be looked up differently. Intrinsic element attributes should be known *intrinsically* whereas components will likely want to specify their own set of attributes.

TypeScript uses the [same convention that React does](#) for distinguishing between these. An intrinsic element always begins with a lowercase letter, and a value-based element always begins with an uppercase letter.

Intrinsic elements

Intrinsic elements are looked up on the special interface `JSX.IntrinsicElements`. By default, if this interface is not specified, then anything goes and intrinsic elements will not be type checked. However, if this interface *is* present, then the name of the intrinsic element is looked up as a property on the `JSX.IntrinsicElements` interface. For example:

```
declare namespace JSX {
  interface IntrinsicElements {
    foo: any;
  }
}
```

```
<foo />; // ok
<bar />; // error
```

In the above example, `<foo />` will work fine but `<bar />` will result in an error since it has not been specified on `JSX.IntrinsicElements`.

Note: You can also specify a catch-all string indexer on `JSX.IntrinsicElements` as follows:

```
declare namespace JSX {
  interface IntrinsicElements {
    [elemName: string]: any;
  }
}
```

Value-based elements

Value-based elements are simply looked up by identifiers that are in scope.

```
import MyComponent from "./myComponent";
<MyComponent />; // ok
<SomeOtherComponent />; // error
```

There are two ways to define a value-based element:

1. Function Component (FC)
2. Class Component

Because these two types of value-based elements are indistinguishable from each other in a JSX expression, first TS tries to resolve the expression as a Function Component using overload resolution. If the process succeeds, then TS finishes resolving the expression to its declaration. If the value fails to resolve as a Function Component, TS will then try to resolve it as a class component. If that fails, TS will report an error.

Function Component

As the name suggests, the component is defined as a JavaScript function where its first argument is a `props` object. TS enforces that its return type must be assignable to `JSX.Element`.

```
interface FooProp {
  name: string;
  X: number;
  Y: number;
}

declare function AnotherComponent(prop: { name: string });
function ComponentFoo(prop: FooProp) {
  return <AnotherComponent name={prop.name} />;
}
```

```
const Button = (prop: { value: string }, context: { color: string }) => (
  <button />
);
```

Because a Function Component is simply a JavaScript function, function overloads may be used here as well:

```
// @noErrors
declare module JSX {
  interface Element {}
  interface IntrinsicElements {
    [s: string]: any;
  }
}
// ---cut---
interface ClickableProps {
  children: JSX.Element[] | JSX.Element;
}

interface HomeProps extends ClickableProps {
  home: JSX.Element;
}

interface SideProps extends ClickableProps {
  side: JSX.Element | string;
}

function MainButton(prop: HomeProps): JSX.Element;
function MainButton(prop: SideProps): JSX.Element;
function MainButton(prop: ClickableProps): JSX.Element {
  // ...
}
```

Note: Function Components were formerly known as Stateless Function Components (SFC). As Function Components can no longer be considered stateless in recent versions of react, the type `SFC` and its alias `StatelessComponent` were deprecated.

Class Component

It is possible to define the type of a class component. However, to do so it is best to understand two new terms: the *element class type* and the *element instance type*.

Given `<Expr />`, the *element class type* is the type of `Expr`. So in the example above, if `MyComponent` was an ES6 class the class type would be that class's constructor and statics. If `MyComponent` was a factory function, the class type would be that function.

Once the class type is established, the instance type is determined by the union of the return types of the class type's construct or call signatures (whichever is present). So again, in the case of an ES6 class, the instance type would be the type of an instance of that class, and in the case of a factory function, it would be the type of the value returned from the function.

```

class MyComponent {
  render() {}
}

// use a construct signature
const myComponent = new MyComponent();

// element class type => MyComponent
// element instance type => { render: () => void }

function MyFactoryFunction() {
  return {
    render: () => {},
  };
}

// use a call signature
const myComponent = MyFactoryFunction();

// element class type => MyFactoryFunction
// element instance type => { render: () => void }

```

The element instance type is interesting because it must be assignable to `JSX.ElementClass` or it will result in an error. By default `JSX.ElementClass` is `{}`, but it can be augmented to limit the use of JSX to only those types that conform to the proper interface.

```

declare namespace JSX {
  interface ElementClass {
    render: any;
  }
}

class MyComponent {
  render() {}
}
function MyFactoryFunction() {
  return { render: () => {} };
}

<MyComponent />; // ok
<MyFactoryFunction />; // ok

class NotValidComponent {}
function NotValidFactoryFunction() {
  return {};
}

<NotValidComponent />; // error
<NotValidFactoryFunction />; // error

```

Attribute type checking

The first step to type checking attributes is to determine the *element attributes type*. This is slightly different between intrinsic and value-based elements.

For intrinsic elements, it is the type of the property on `JSX.IntrinsicElements`

```

declare namespace JSX {
  interface IntrinsicElements {
    foo: { bar?: boolean };
  }
}

// element attributes type for 'foo' is '{bar?: boolean}'
<foo bar />;

```

For value-based elements, it is a bit more complex. It is determined by the type of a property on the *element instance type* that was previously determined. Which property to use is determined by `JSX.ElementAttributesProperty`. It should be declared with a single property. The name of that property is then used. As of TypeScript 2.8, if `JSX.ElementAttributesProperty` is not provided, the type of first parameter of the class element's constructor or Function Component's call will be used instead.

```

declare namespace JSX {
  interface ElementAttributesProperty {
    props; // specify the property name to use
  }
}

class MyComponent {
  // specify the property on the element instance type
  props: {
    foo?: string;
  };
}

// element attributes type for 'MyComponent' is '{foo?: string}'
<MyComponent foo="bar" />;

```

The element attribute type is used to type check the attributes in the JSX. Optional and required properties are supported.

```

declare namespace JSX {
  interface IntrinsicElements {
    foo: { requiredProp: string; optionalProp?: number };
  }
}

<foo requiredProp="bar" />; // ok
<foo requiredProp="bar" optionalProp={0} />; // ok
<foo />; // error, requiredProp is missing
<foo requiredProp={0} />; // error, requiredProp should be a string
<foo requiredProp="bar" unknownProp />; // error, unknownProp does not exist
<foo requiredProp="bar" some-unknown-prop />; // ok, because 'some-unknown-prop' is not a valid identifier

```

Note: If an attribute name is not a valid JS identifier (like a `data-*` attribute), it is not considered to be an error if it is not found in the element attributes type.

Additionally, the `JSX.IntrinsicAttributes` interface can be used to specify extra properties used by the JSX framework which are not generally used by the components' props or arguments - for instance `key` in React. Specializing further, the generic `JSX.IntrinsicClassAttributes<T>` type may also be used to specify the same kind of extra attributes just for class components (and not Function Components). In this type, the generic parameter corresponds to the class instance type. In React, this is used to allow the `ref` attribute of type `Ref<T>`. Generally speaking, all of the properties on these interfaces should be optional, unless you intend that users of your JSX framework need to provide some attribute on every tag.

The spread operator also works:

```
const props = { requiredProp: "bar" };
<foo {...props} />; // ok

const badProps = {};
<foo {...badProps} />; // error
```

Children Type Checking

In TypeScript 2.3, TS introduced type checking of `children`. `children` is a special property in an `element attributes type` where child `JSXExpressions` are taken to be inserted into the attributes. Similar to how TS uses `JSX.ElementAttributesProperty` to determine the name of `props`, TS uses `JSX.ElementChildrenAttribute` to determine the name of `children` within those `props`. `JSX.ElementChildrenAttribute` should be declared with a single property.

```
declare namespace JSX {
  interface ElementChildrenAttribute {
    children: {}; // specify children name to use
  }
}

<div>
  <h1>Hello</h1>
</div>

<div>
  <h1>Hello</h1>
  World
</div>

const CustomComp = (props) => <div>{props.children}</div>
<CustomComp>
  <div>Hello World</div>
  {"This is just a JS expression..." + 1000}
</CustomComp>
```

You can specify the type of `children` like any other attribute. This will override the default type from, e.g. the [React typings](#) if you use them.

```
interface PropsType {
  children: JSX.Element
  name: string
}
```

```

class Component extends React.Component<PropsType, {}> {
  render() {
    return (
      <h2>
        {this.props.children}
      </h2>
    )
  }
}

// OK
<Component name="foo">
  <h1>Hello World</h1>
</Component>

// Error: children is of type JSX.Element not array of JSX.Element
<Component name="bar">
  <h1>Hello World</h1>
  <h2>Hello World</h2>
</Component>

// Error: children is of type JSX.Element not array of JSX.Element or string.
<Component name="baz">
  <h1>Hello</h1>
  World
</Component>

```

The JSX result type

By default the result of a JSX expression is typed as `any`. You can customize the type by specifying the `JSX.Element` interface. However, it is not possible to retrieve type information about the element, attributes or children of the JSX from this interface. It is a black box.

Embedding Expressions

JSX allows you to embed expressions between tags by surrounding the expressions with curly braces (`{ }`).

```

const a = (
  <div>
    {"foo", "bar"].map((i) => (
      <span>{i / 2}</span>
    )));
  </div>
);

```

The above code will result in an error since you cannot divide a string by a number. The output, when using the `preserve` option, looks like:

```

const a = (
  <div>
    {"foo", "bar"].map(function (i) {
      return <span>{i / 2}</span>;
    }));
  </div>
);

```

React integration

To use JSX with React you should use the [React typings](#). These typings define the `JSX` namespace appropriately for use with React.

```
/// <reference path="react.d.ts" />

interface Props {
  foo: string;
}

class MyComponent extends React.Component<Props, {}> {
  render() {
    return <span>{this.props.foo}</span>;
  }
}

<MyComponent foo="bar" />; // ok
<MyComponent foo={0} />; // error
```

Configuring JSX

There are multiple compiler flags which can be used to customize your JSX, which work as both a compiler flag and via inline per-file pragmas. To learn more see their [tsconfig](#) reference pages:

- `jsxFactory`
- `jsxFragmentFactory`
- `jsxImportSource`

[Go to TOC](#)

Along with traditional OO hierarchies, another popular way of building up classes from reusable components is to build them by combining simpler partial classes. You may be familiar with the idea of mixins or traits for languages like Scala, and the pattern has also reached some popularity in the JavaScript community.

How Does A Mixin Work?

The pattern relies on using generics with class inheritance to extend a base class. TypeScript's best mixin support is done via the class expression pattern. You can read more about how this pattern works in JavaScript [here](#).

To get started, we'll need a class which will have the mixins applied on top of:

```
class Sprite {
  name = "";
  x = 0;
  y = 0;

  constructor(name: string) {
    this.name = name;
  }
}
```

Then you need a type and a factory function which returns a class expression extending the base class.

```
// To get started, we need a type which we'll use to extend
// other classes from. The main responsibility is to declare
// that the type being passed in is a class.

type Constructor = new (...args: any[]) => {};

// This mixin adds a scale property, with getters and setters
// for changing it with an encapsulated private property:

function Scale<TBase extends Constructor>(Base: TBase) {
  return class Scaling extends Base {
    // Mixins may not declare private/protected properties
    // however, you can use ES2020 private fields
    _scale = 1;

    setScale(scale: number) {
      this._scale = scale;
    }

    get scale(): number {
      return this._scale;
    }
  };
}
```

With these all set up, then you can create a class which represents the base class with mixins applied:

```
class Sprite {
  name = "";
  x = 0;
  y = 0;
```

```

constructor(name: string) {
  this.name = name;
}
}
type Constructor = new (...args: any[]) => {};
function Scale<TBase extends Constructor>(Base: TBase) {
  return class Scaling extends Base {
    // Mixins may not declare private/protected properties
    // however, you can use ES2020 private fields
    _scale = 1;

    setScale(scale: number) {
      this._scale = scale;
    }

    get scale(): number {
      return this._scale;
    }
  };
}

// ---cut---
// Compose a new class from the Sprite class,
// with the Mixin Scale applier:
const EightBitSprite = Scale(Sprite);

const flappySprite = new EightBitSprite("Bird");
flappySprite.setScale(0.8);
console.log(flappySprite.scale);

```

Constrained Mixins

In the above form, the mixin's have no underlying knowledge of the class which can make it hard to create the design you want.

To model this, we modify the original constructor type to accept a generic argument.

```

// This was our previous constructor:
type Constructor = new (...args: any[]) => {};
// Now we use a generic version which can apply a constraint on
// the class which this mixin is applied to
type GConstructor<T = {}> = new (...args: any[]) => T;

```

This allows for creating classes which only work with constrained base classes:

```

type GConstructor<T = {}> = new (...args: any[]) => T;
class Sprite {
  name = "";
  x = 0;
  y = 0;

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

// ---cut---
type Positionable = GConstructor<{ setPos: (x: number, y: number) => void }>;
type Spritable = GConstructor<Sprite>;
type Loggable = GConstructor<{ print: () => void }>;

```

Then you can create mixins which only work when you have a particular base to build on:

```
type GConstructor<T = {}> = new (...args: any[]) => T;
class Sprite {
  name = "";
  x = 0;
  y = 0;

  constructor(name: string) {
    this.name = name;
  }
}
type Positionable = GConstructor<{ setPos: (x: number, y: number) => void }>;
type Spritable = GConstructor<Sprite>;
type Loggable = GConstructor<{ print: () => void }>;
// ---cut--
```

```
function Jumpable<TBase extends Positionable>(Base: TBase) {
  return class Jumpable extends Base {
    jump() {
      // This mixin will only work if it is passed a base
      // class which has setPos defined because of the
      // Positionable constraint.
      this.setPos(0, 20);
    }
  };
}
```

Alternative Pattern

Previous versions of this document recommended a way to write mixins where you created both the runtime and type hierarchies separately, then merged them at the end:

```
// @strict: false
// Each mixin is a traditional ES class
class Jumpable {
  jump() {}
}

class Duckable {
  duck() {}
}

// Including the base
class Sprite {
  x = 0;
  y = 0;
}

// Then you create an interface which merges
// the expected mixins with the same name as your base
interface Sprite extends Jumpable, Duckable {}
// Apply the mixins into the base class via
// the JS at runtime
applyMixins(Sprite, [Jumpable, Duckable]);

let player = new Sprite();
player.jump();
console.log(player.x, player.y);
```

```
// This can live anywhere in your codebase:
function applyMixins(derivedCtor: any, constructors: any[]) {
  constructors.forEach((baseCtor) => {
    Object.getOwnPropertyNames(baseCtor.prototype).forEach((name) => {
      Object.defineProperty(
        derivedCtor.prototype,
        name,
        Object.getOwnPropertyDescriptor(baseCtor.prototype, name) || Object.create(null)
      );
    });
  });
}
```

This pattern relies less on the compiler, and more on your codebase to ensure both runtime and type-system are correctly kept in sync.

Constraints

The mixin pattern is supported natively inside the TypeScript compiler by code flow analysis. There are a few cases where you can hit the edges of the native support.

Decorators and Mixins #4881

You cannot use decorators to provide mixins via code flow analysis:

```
// @experimentalDecorators
// @errors: 2339
// A decorator function which replicates the mixin pattern:
const Pausable = (target: typeof Player) => {
  return class Pausable extends target {
    shouldFreeze = false;
  };
};

@Pausable
class Player {
  x = 0;
  y = 0;
}

// The Player class does not have the decorator's type merged:
const player = new Player();
player.shouldFreeze;

// The runtime aspect could be manually replicated via
// type composition or interface merging.
type FreezablePlayer = Player & { shouldFreeze: boolean };

const playerTwo = (new Player() as unknown) as FreezablePlayer;
playerTwo.shouldFreeze;
```

Static Property Mixins #17829

More of a gotcha than a constraint. The class expression pattern creates singletons, so they can't be mapped at the type system to support different variable types.

You can work around this by using functions to return your classes which differ based on a generic:

```
function base<T>() {
  class Base {
    static prop: T;
  }
  return Base;
}

function derived<T>() {
  class Derived extends base<T>() {
    static anotherProp: T;
  }
  return Derived;
}

class Spec extends derived<string>()

Spec.prop; // string
Spec.anotherProp; // string
```

[Go to TOC](#)

This section assumes some basic knowledge about modules. Please see the [Modules](#) documentation for more information.

Module resolution is the process the compiler uses to figure out what an import refers to. Consider an import statement like `import { a } from "moduleA";` in order to check any use of `a`, the compiler needs to know exactly what it represents, and will need to check its definition `moduleA`.

At this point, the compiler will ask "what's the shape of `moduleA`?" While this sounds straightforward, `moduleA` could be defined in one of your own `.ts` / `.tsx` files, or in a `.d.ts` that your code depends on.

First, the compiler will try to locate a file that represents the imported module. To do so the compiler follows one of two different strategies: [Classic](#) or [Node](#). These strategies tell the compiler *where* to look for `moduleA`.

If that didn't work and if the module name is non-relative (and in the case of `"moduleA"`, it is), then the compiler will attempt to locate an [ambient module declaration](#). We'll cover non-relative imports next.

Finally, if the compiler could not resolve the module, it will log an error. In this case, the error would be something like `error TS2307: Cannot find module 'moduleA'.`

Relative vs. Non-relative module imports

Module imports are resolved differently based on whether the module reference is relative or non-relative.

A *relative import* is one that starts with `/`, `./` or `../`. Some examples include:

- `import Entry from "./components/Entry";`
- `import { DefaultHeaders } from "../constants/http";`
- `import "/mod";`

Any other import is considered **non-relative**. Some examples include:

- `import * as $ from "jquery";`
- `import { Component } from "@angular/core";`

A relative import is resolved relative to the importing file and *cannot* resolve to an ambient module declaration. You should use relative imports for your own modules that are guaranteed to maintain their relative location at runtime.

A non-relative import can be resolved relative to `baseUrl`, or through path mapping, which we'll cover below. They can also resolve to [ambient module declarations](#). Use non-relative paths when importing any of your external dependencies.

Module Resolution Strategies

There are two possible module resolution strategies: [Node](#) and [Classic](#). You can use the `moduleResolution` option to specify the module resolution strategy. If not specified, the default is [Node](#) for `--module commonjs`, and [Classic](#) otherwise (including when `module` is set to `amd`, `system`, `umd`, `es2015`, `esnext`, etc.).

Note: `node` module resolution is the most-commonly used in the TypeScript community and is recommended for most projects. If you are having resolution problems with `import`s and `export`s in TypeScript, try setting `moduleResolution: "node"` to see if it fixes the issue.

Classic

This used to be TypeScript's default resolution strategy. Nowadays, this strategy is mainly present for backward compatibility.

A relative import will be resolved relative to the importing file. So `import { b } from "./moduleB"` in source file `/root/src/folder/A.ts` would result in the following lookups:

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`

For non-relative module imports, however, the compiler walks up the directory tree starting with the directory containing the importing file, trying to locate a matching definition file.

For example:

A non-relative import to `moduleB` such as `import { b } from "moduleB"`, in a source file `/root/src/folder/A.ts`, would result in attempting the following locations for locating `"moduleB"`:

1. `/root/src/folder/moduleB.ts`
2. `/root/src/folder/moduleB.d.ts`
3. `/root/src/moduleB.ts`
4. `/root/src/moduleB.d.ts`
5. `/root/moduleB.ts`
6. `/root/moduleB.d.ts`
7. `/moduleB.ts`
8. `/moduleB.d.ts`

Node

This resolution strategy attempts to mimic the [Node.js](#) module resolution mechanism at runtime. The full Node.js resolution algorithm is outlined in [Node.js module documentation](#).

How Node.js resolves modules

To understand what steps the TS compiler will follow, it is important to shed some light on Node.js modules. Traditionally, imports in Node.js are performed by calling a function named `require`. The behavior Node.js takes will differ depending on if `require` is given a relative path or a non-relative path.

Relative paths are fairly straightforward. As an example, let's consider a file located at `/root/src/moduleA.js`, which contains the import `var x = require("./moduleB");`. Node.js resolves that import in the following order:

1. Ask the file named `/root/src/moduleB.js`, if it exists.
2. Ask the folder `/root/src/moduleB` if it contains a file named `package.json` that specifies a `"main"` module. In our example, if Node.js found the file `/root/src/moduleB/package.json` containing `{"main": "lib/mainModule.js"}`, then Node.js will refer to `/root/src/moduleB/lib/mainModule.js`.
3. Ask the folder `/root/src/moduleB` if it contains a file named `index.js`. That file is implicitly considered that folder's "main" module.

You can read more about this in Node.js documentation on [file modules](#) and [folder modules](#).

However, resolution for a `non-relative module name` is performed differently. Node will look for your modules in special folders named `node_modules`. A `node_modules` folder can be on the same level as the current file, or higher up in the directory chain. Node will walk up the directory chain, looking through each `node_modules` until it finds the module you tried to load.

Following up our example above, consider if `/root/src/moduleA.js` instead used a non-relative path and had the import `var x = require("moduleB");`. Node would then try to resolve `moduleB` to each of the locations until one worked.

1. `/root/src/node_modules/moduleB.js`
2. `/root/src/node_modules/moduleB/package.json` (if it specifies a `"main"` property)
3. `/root/src/node_modules/moduleB/index.js`

4. `/root/node_modules/moduleB.js`
5. `/root/node_modules/moduleB/package.json` (if it specifies a `"main"` property)
6. `/root/node_modules/moduleB/index.js`

7. `/node_modules/moduleB.js`
8. `/node_modules/moduleB/package.json` (if it specifies a `"main"` property)
9. `/node_modules/moduleB/index.js`

Notice that Node.js jumped up a directory in steps (4) and (7).

You can read more about the process in Node.js documentation on [loading modules from node_modules](#).

How TypeScript resolves modules

TypeScript will mimic the Node.js run-time resolution strategy in order to locate definition files for modules at compile-time. To accomplish this, TypeScript overlays the TypeScript source file extensions (`.ts`, `.tsx`, and `.d.ts`) over Node's resolution logic. TypeScript will also use a field in `package.json` named `types` to mirror the purpose of `"main"` - the compiler will use it to find the "main" definition file to consult.

For example, an import statement like `import { b } from "./moduleB"` in `/root/src/moduleA.ts` would result in attempting the following locations for locating `"../moduleB"`:

1. `/root/src/moduleB.ts`
2. `/root/src/moduleB.tsx`
3. `/root/src/moduleB.d.ts`
4. `/root/src/moduleB/package.json` (if it specifies a `types` property)
5. `/root/src/moduleB/index.ts`
6. `/root/src/moduleB/index.tsx`
7. `/root/src/moduleB/index.d.ts`

Recall that Node.js looked for a file named `moduleB.js`, then an applicable `package.json`, and then for an `index.js`.

Similarly, a non-relative import will follow the Node.js resolution logic, first looking up a file, then looking up an applicable folder. So `import { b } from "moduleB"` in source file `/root/src/moduleA.ts` would result in the following lookups:

1. `/root/src/node_modules/moduleB.ts`
2. `/root/src/node_modules/moduleB.tsx`
3. `/root/src/node_modules/moduleB.d.ts`
4. `/root/src/node_modules/moduleB/package.json` (if it specifies a `types` property)
5. `/root/src/node_modules/@types/moduleB.d.ts`
6. `/root/src/node_modules/moduleB/index.ts`
7. `/root/src/node_modules/moduleB/index.tsx`
8. `/root/src/node_modules/moduleB/index.d.ts`

9. `/root/node_modules/moduleB.ts`
10. `/root/node_modules/moduleB.tsx`
11. `/root/node_modules/moduleB.d.ts`
12. `/root/node_modules/moduleB/package.json` (if it specifies a `types` property)
13. `/root/node_modules/@types/moduleB.d.ts`
14. `/root/node_modules/moduleB/index.ts`
15. `/root/node_modules/moduleB/index.tsx`
16. `/root/node_modules/moduleB/index.d.ts`

17. `/node_modules/moduleB.ts`
18. `/node_modules/moduleB.tsx`

```

19. /node_modules/moduleB.d.ts
20. /node_modules/moduleB/package.json (if it specifies a types property)
21. /node_modules/@types/moduleB.d.ts
22. /node_modules/moduleB/index.ts
23. /node_modules/moduleB/index.tsx
24. /node_modules/moduleB/index.d.ts

```

Don't be intimidated by the number of steps here - TypeScript is still only jumping up directories twice at steps (9) and (17). This is really no more complex than what Node.js itself is doing.

Additional module resolution flags

A project source layout sometimes does not match that of the output. Usually a set of build steps result in generating the final output. These include compiling `.ts` files into `.js`, and copying dependencies from different source locations to a single output location. The net result is that modules at runtime may have different names than the source files containing their definitions. Or module paths in the final output may not match their corresponding source file paths at compile time.

The TypeScript compiler has a set of additional flags to *inform* the compiler of transformations that are expected to happen to the sources to generate the final output.

It is important to note that the compiler will *not* perform any of these transformations; it just uses these pieces of information to guide the process of resolving a module import to its definition file.

Base URL

Using a `baseUrl` is a common practice in applications using AMD module loaders where modules are "deployed" to a single folder at run-time. The sources of these modules can live in different directories, but a build script will put them all together.

Setting `baseUrl` informs the compiler where to find modules. All module imports with non-relative names are assumed to be relative to the `baseUrl`.

Value of `baseUrl` is determined as either:

- value of `baseUrl` command line argument (if given path is relative, it is computed based on current directory)
- value of `baseUrl` property in `'tsconfig.json'` (if given path is relative, it is computed based on the location of `'tsconfig.json'`)

Note that relative module imports are not impacted by setting the `baseUrl`, as they are always resolved relative to their importing files.

You can find more documentation on `baseUrl` in [RequireJS](#) and [SystemJS](#) documentation.

Path mapping

Sometimes modules are not directly located under `baseUrl`. For instance, an import to a module `"jquery"` would be translated at runtime to `"node_modules/jquery/dist/jquery.slim.min.js"`. Loaders use a mapping configuration to map module names to files at run-time, see [RequireJs documentation](#) and [SystemJS documentation](#).

The TypeScript compiler supports the declaration of such mappings using `paths` property in `tsconfig.json` files. Here is an example for how to specify the `paths` property for `jquery`.

```
{
  "compilerOptions": {
    "baseUrl": ".", // This must be specified if "paths" is.
    "paths": {
      "jquery": ["node_modules/jquery/dist/jquery"] // This mapping is relative to
      "baseUrl"
    }
  }
}
```

Please notice that `paths` are resolved relative to `baseUrl`. When setting `baseUrl` to a value other than `".."`, i.e. the directory of `tsconfig.json`, the mappings must be changed accordingly. Say, you set `"baseUrl": "./src"` in the above example, then `jquery` should be mapped to `"../node_modules/jquery/dist/jquery"`.

Using `paths` also allows for more sophisticated mappings including multiple fall back locations. Consider a project configuration where only some modules are available in one location, and the rest are in another. A build step would put them all together in one place. The project layout may look like:

```
projectRoot
├── folder1
│   ├── file1.ts (imports 'folder1/file2' and 'folder2/file3')
│   └── file2.ts
└── generated
    ├── folder1
    │   └── folder2
    │       └── file3.ts
    └── tsconfig.json
```

The corresponding `tsconfig.json` would look like:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "*": ["*", "generated/*"]
    }
  }
}
```

This tells the compiler for any module import that matches the pattern `"*"` (i.e. all values), to look in two locations:

1. `"*"` : meaning the same name unchanged, so map `<moduleName> => <baseUrl>/<moduleName>`
2. `"generated/*"` meaning the module name with an appended prefix "generated", so map `<moduleName> => <baseUrl>/generated/<moduleName>`

Following this logic, the compiler will attempt to resolve the two imports as such:

```
import 'folder1/file2':
```

1. pattern `'*'` is matched and wildcard captures the whole module name
2. try first substitution in the list: `'*' -> folder1/file2`
3. result of substitution is non-relative name - combine it with `baseUrl` -> `projectRoot/folder1/file2.ts`.
4. File exists. Done.

```
import 'folder2/file3':
```

1. pattern `'*'` is matched and wildcard captures the whole module name
2. try first substitution in the list: `'*' -> folder2/file3`
3. result of substitution is non-relative name - combine it with `baseUrl` -> `projectRoot/folder2/file3.ts`.
4. File does not exist, move to the second substitution
5. second substitution `'generated/*'` -> `generated/folder2/file3`
6. result of substitution is non-relative name - combine it with `baseUrl` -> `projectRoot/generated/folder2/file3.ts`.
7. File exists. Done.

Virtual Directories with `rootDirs`

Sometimes the project sources from multiple directories at compile time are all combined to generate a single output directory. This can be viewed as a set of source directories create a "virtual" directory.

Using `rootDirs`, you can inform the compiler of the `roots` making up this "virtual" directory; and thus the compiler can resolve relative modules imports within these "virtual" directories *as if* they were merged together in one directory.

For example consider this project structure:

```
src
└── views
    └── view1.ts (imports './template1')
    └── view2.ts

generated
└── templates
    └── views
        └── template1.ts (imports './view2')
```

Files in `src/views` are user code for some UI controls. Files in `generated/templates` are UI template binding code auto-generated by a template generator as part of the build. A build step will copy the files in `/src/views` and `/generated/templates/views` to the same directory in the output. At run-time, a view can expect its template to exist next to it, and thus should import it using a relative name as `"./template"`.

To specify this relationship to the compiler, use `rootDirs`. `rootDirs` specify a list of *roots* whose contents are expected to merge at run-time. So following our example, the `tsconfig.json` file should look like:

```
{
  "compilerOptions": {
    "rootDirs": ["src/views", "generated/templates/views"]
  }
}
```

Every time the compiler sees a relative module import in a subfolder of one of the `rootDirs`, it will attempt to look for this import in each of the entries of `rootDirs`.

The flexibility of `rootDirs` is not limited to specifying a list of physical source directories that are logically merged. The supplied array may include any number of ad hoc, arbitrary directory names, regardless of whether they exist or not. This allows the compiler to capture sophisticated bundling and runtime features such as conditional inclusion and project specific loader plugins in a type safe way.

Consider an internationalization scenario where a build tool automatically generates locale specific bundles by interpolating a special path token, say `#{locale}`, as part of a relative module path such as `./#{locale}/messages`. In this hypothetical setup the tool enumerates supported locales, mapping the abstracted path into `./zh/messages`, `./de/messages`, and so forth.

Assume that each of these modules exports an array of strings. For example `./zh/messages` might contain:

```
export default ["您好吗", "很高兴认识你"];
```

By leveraging `rootDirs` we can inform the compiler of this mapping and thereby allow it to safely resolve `./#{locale}/messages`, even though the directory will never exist. For example, with the following `tsconfig.json`:

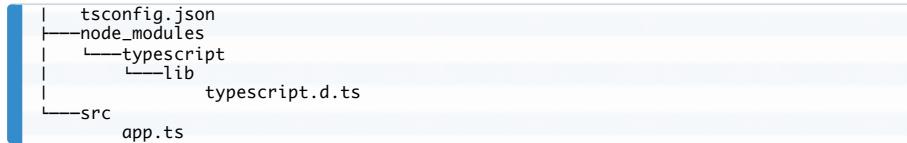
```
{
  "compilerOptions": {
    "rootDirs": ["src/zh", "src/de", "src/#{locale}"]
  }
}
```

The compiler will now resolve `import messages from './#{locale}/messages'` to `import messages from './zh/messages'` for tooling purposes, allowing development in a locale agnostic manner without compromising design time support.

Tracing module resolution

As discussed earlier, the compiler can visit files outside the current folder when resolving a module. This can be hard when diagnosing why a module is not resolved, or is resolved to an incorrect definition. Enabling the compiler module resolution tracing using `traceResolution` provides insight in what happened during the module resolution process.

Let's say we have a sample application that uses the `typescript` module. `app.ts` has an import like `import * as ts from "typescript"`.



Invoking the compiler with `traceResolution`

```
tsc --traceResolution
```

Results in an output such as:

```
===== Resolving module 'typescript' from 'src/app.ts'. =====
Module resolution kind is not specified, using 'NodeJs'.
Loading module 'typescript' from 'node_modules' folder.
File 'src/node_modules/typescript.ts' does not exist.
File 'src/node_modules/typescript.tsx' does not exist.
File 'src/node_modules/typescript.d.ts' does not exist.
File 'src/node_modules/typescript/package.json' does not exist.
File 'node_modules/typescript.ts' does not exist.
File 'node_modules/typescript.tsx' does not exist.
File 'node_modules/typescript.d.ts' does not exist.
Found 'package.json' at 'node_modules/typescript/package.json'.
'package.json' has 'types' field './lib/typescript.d.ts' that references
'node_modules/typescript/lib/typescript.d.ts'.
File 'node_modules/typescript/lib/typescript.d.ts' exist - use it as a module
resolution result.
===== Module name 'typescript' was successfully resolved to
'node_modules/typescript/lib/typescript.d.ts'. =====
```

Things to look out for

- Name and location of the import

```
===== Resolving module 'typescript' from 'src/app.ts'. =====
```

- The strategy the compiler is following

```
Module resolution kind is not specified, using 'NodeJs'.
```

- Loading of types from npm packages

```
'package.json' has 'types' field './lib/typescript.d.ts' that references  
'node_modules/typescript/lib/typescript.d.ts'.
```

- Final result

```
===== Module name 'typescript' was successfully resolved to  
'node_modules/typescript/lib/typescript.d.ts'. =====
```

Using `--noResolve`

Normally the compiler will attempt to resolve all module imports before it starts the compilation process. Every time it successfully resolves an `import` to a file, the file is added to the set of files the compiler will process later on.

The `noResolve` compiler options instructs the compiler not to "add" any files to the compilation that were not passed on the command line. It will still try to resolve the module to files, but if the file is not specified, it will not be included.

For instance:

`app.ts`

```
import * as A from "moduleA"; // OK, 'moduleA' passed on the command-line
import * as B from "moduleB"; // Error TS2307: Cannot find module 'moduleB'.
```

```
tsc app.ts moduleA.ts --noResolve
```

Compiling `app.ts` using `noResolve` should result in:

- Correctly finding `moduleA` as it was passed on the command-line.
- Error for not finding `moduleB` as it was not passed.

Common Questions

Why does a module in the exclude list still get picked up by the compiler?

`tsconfig.json` turns a folder into a "project". Without specifying any `"exclude"` or `"files"` entries, all files in the folder containing the `tsconfig.json` and all its sub-directories are included in your compilation. If you want to exclude some of the files use `"exclude"`, if you would rather specify all the files instead of letting the compiler look them up, use `"files"`.

That was `tsconfig.json` automatic inclusion. That does not embed module resolution as discussed above. If the compiler identified a file as a target of a module import, it will be included in the compilation regardless if it was excluded in the previous steps.

So to exclude a file from the compilation, you need to exclude it and **all** files that have an `import` or `///<reference path="..." />` directive to it.

[Go to TOC](#)

Starting with ECMAScript 2015, JavaScript has a concept of modules. TypeScript shares this concept.

Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the `export` forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the `import` forms.

Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.

Modules import one another using a module loader. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it. Well-known module loaders used in JavaScript are Node.js's loader for [CommonJS](#) modules and the [RequireJS](#) loader for [AMD](#) modules in Web applications.

In TypeScript, just as in ECMAScript 2015, any file containing a top-level `import` or `export` is considered a module. Conversely, a file without any top-level `import` or `export` declarations is treated as a script whose contents are available in the global scope (and therefore to modules as well).

Export

Exporting a declaration

Any declaration (such as a variable, function, class, type alias, or interface) can be exported by adding the `export` keyword.

StringValidator.ts

```
export interface StringValidator {
  isAcceptable(s: string): boolean;
}
```

ZipCodeValidator.ts

```
import { StringValidator } from "./StringValidator";

export const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
```

Export statements

Export statements are handy when exports need to be renamed for consumers, so the above example can be written as:

```
class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
export { ZipCodeValidator };
export { ZipCodeValidator as mainValidator };
```

Re-exports

Often modules extend other modules, and partially expose some of their features. A re-export does not import it locally, or introduce a local variable.

ParseIntBasedZipCodeValidator.ts

```
export class ParseIntBasedZipCodeValidator {
  isAcceptable(s: string) {
    return s.length === 5 && parseInt(s).toString() === s;
  }
}

// Export original validator but rename it
export { ZipCodeValidator as RegExpBasedZipCodeValidator } from
"./ZipCodeValidator";
```

Optionally, a module can wrap one or more modules and combine all their exports using `export * from "module"` syntax.

AllValidators.ts

```
export * from "./StringValidator"; // exports 'StringValidator' interface
export * from "./ZipCodeValidator"; // exports 'ZipCodeValidator' class and
'numberRegexp' constant value
export * from "./ParseIntBasedZipCodeValidator"; // exports the
'ParseIntBasedZipCodeValidator' class
// and re-exports 'RegExpBasedZipCodeValidator' as alias
// of the 'ZipCodeValidator' class from 'ZipCodeValidator.ts'
// module.
```

Import

Importing is just about as easy as exporting from a module. Importing an exported declaration is done through using one of the `import` forms below:

Import a single export from a module

```
import { ZipCodeValidator } from "./ZipCodeValidator";
let myValidator = new ZipCodeValidator();
```

imports can also be renamed

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

Import the entire module into a single variable, and use it to access the module exports

```
import * as validator from "./ZipCodeValidator";
let myValidator = new validator.ZipCodeValidator();
```

Import a module for side-effects only

Though not recommended practice, some modules set up some global state that can be used by other modules. These modules may not have any exports, or the consumer is not interested in any of their exports. To import these modules, use:

```
import "./my-module.js";
```

Importing Types

Prior to TypeScript 3.8, you can import a type using `import`. With TypeScript 3.8, you can import a type using the `import` statement, or using `import type`.

```
// Re-using the same import
import { APIResponseType } from "./api";

// Explicitly use import type
import type { APIResponseType } from "./api";

// Explicitly pull out a value (getResponse) and a type (APIResponseType)
import { getResponse, type APIResponseType } from "./api";
```

Any explicitly marked `type` import is guaranteed to be removed from your JavaScript, and tools like Babel can make better assumptions about your code via the `isolatedModules` compiler flag. You can read more in the [3.8 release notes](#).

Default exports

Each module can optionally export a `default` export. Default exports are marked with the keyword `default`; and there can only be one `default` export per module. `default` exports are imported using a different import form.

`default` exports are really handy. For instance, a library like jQuery might have a default export of `jQuery` or `$`, which we'd probably also import under the name `$` or `jQuery`.

[jQuery.d.ts](#)

```
declare let $: JQuery;
export default $;
```

App.ts

```
import $ from "jquery";
$("button.continue").html("Next Step...");
```

Classes and function declarations can be authored directly as default exports. Default export class and function declaration names are optional.

ZipCodeValidator.ts

```
export default class ZipCodeValidator {
  static numberRegexp = /^[0-9]+$/;
  isAcceptable(s: string) {
    return s.length === 5 && ZipCodeValidator.numberRegexp.test(s);
  }
}
```

Test.ts

```
import validator from "./ZipCodeValidator";
let myValidator = new validator();
```

or

StaticZipCodeValidator.ts

```
const numberRegexp = /^[0-9]+$/;

export default function (s: string) {
  return s.length === 5 && numberRegexp.test(s);
}
```

Test.ts

```
import validate from "./StaticZipCodeValidator";

let strings = ["Hello", "98052", "101"];

// Use function validate
strings.forEach((s) => {
  console.log(`"${s}" ${validate(s) ? "matches" : "does not match"}`);
});
```

`default` exports can also be just values:

OneTwoThree.ts

```
export default "123";
```

Log.ts

```
import num from "./OneTwoThree";
console.log(num); // "123"
```

Export all as x

With TypeScript 3.8, you can use `export * as ns` as a shorthand for re-exporting another module with a name:

```
export * as utilities from "./utilities";
```

This takes all of the dependencies from a module and makes it an exported field, you could import it like this:

```
import { utilities } from "./index";
```

export = and import = require()

Both CommonJS and AMD generally have the concept of an `exports` object which contains all exports from a module.

They also support replacing the `exports` object with a custom single object. Default exports are meant to act as a replacement for this behavior; however, the two are incompatible. TypeScript supports `export =` to model the traditional CommonJS and AMD workflow.

The `export =` syntax specifies a single object that is exported from the module. This can be a class, interface, namespace, function, or enum.

When exporting a module using `export =`, TypeScript-specific `import module = require("module")` must be used to import the module.

ZipCodeValidator.ts

```
let numberRegexp = /^[0-9]+$/;
class ZipCodeValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
export = ZipCodeValidator;
```

Test.ts

```
import zip = require("./ZipCodeValidator");

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validator = new zip();

// Show whether each string passed each validator
strings.forEach((s) => {
  console.log(`"${s}" - ${validator.isAcceptable(s) ? "matches" : "does not match"}`);
});
```

Code Generation for Modules

Depending on the module target specified during compilation, the compiler will generate appropriate code for Node.js ([CommonJS](#)), require.js ([AMD](#)), [UMD](#), [SystemJS](#), or [ECMAScript 2015 native modules](#) (ES6) module-loading systems. For more information on what the `define`, `require` and `register` calls in the generated code do, consult the documentation for each module loader.

This simple example shows how the names used during importing and exporting get translated into the module loading code.

SimpleModule.ts

```
import m = require("mod");
export let t = m.something + 1;
```

AMD / RequireJS SimpleModule.js

```
define(["require", "exports", "./mod"], function (require, exports, mod_1) {
    exports.t = mod_1.something + 1;
});
```

CommonJS / Node SimpleModule.js

```
var mod_1 = require("./mod");
exports.t = mod_1.something + 1;
```

UMD SimpleModule.js

```
(function (factory) {
    if (typeof module === "object" && typeof module.exports === "object") {
        var v = factory(require, exports);
        if (v !== undefined) module.exports = v;
    } else if (typeof define === "function" && define.amd) {
        define(["require", "exports", "./mod"], factory);
    }
})(function (require, exports) {
    var mod_1 = require("./mod");
    exports.t = mod_1.something + 1;
});
```

System SimpleModule.js

```
System.register(["./mod"], function (exports_1) {
    var mod_1;
    var t;
    return {
        setters: [
            function (mod_1_1) {
                mod_1 = mod_1_1;
            },
        ],
        execute: function () {
            exports_1("t", (t = mod_1.something + 1));
        },
    };
});
```

Native ECMAScript 2015 modules SimpleModule.js

```
import { something } from "./mod";
export var t = something + 1;
```

Simple Example

Below, we've consolidated the Validator implementations used in previous examples to only export a single named export from each module.

To compile, we must specify a module target on the command line. For Node.js, use `--module commonjs`; for require.js, use `--module amd`. For example:

```
tsc --module commonjs Test.ts
```

When compiled, each module will become a separate `.js` file. As with reference tags, the compiler will follow `import` statements to compile dependent files.

Validation.ts

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

LettersOnlyValidator.ts

```
import { StringValidator } from "./Validation";
const lettersRegexp = /^[A-Za-z]+$/;

export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}
```

ZipCodeValidator.ts

```
import { StringValidator } from "./Validation";
const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Test.ts

```
import { StringValidator } from "./Validation";
import { ZipCodeValidator } from "./ZipCodeValidator";
import { LettersOnlyValidator } from "./LettersOnlyValidator";

// Some samples to try
```

```

let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// Show whether each string passed each validator
strings.forEach((s) => {
  for (let name in validators) {
    console.log(
      `${s} - ${validators[name].isAcceptable(s) ? "matches" : "does not match"} ${name}`
    );
  }
});

```

Optional Module Loading and Other Advanced Loading Scenarios

In some cases, you may want to only load a module under some conditions. In TypeScript, we can use the pattern shown below to implement this and other advanced loading scenarios to directly invoke the module loaders without losing type safety.

The compiler detects whether each module is used in the emitted JavaScript. If a module identifier is only ever used as part of a type annotations and never as an expression, then no `require` call is emitted for that module. This elision of unused references is a good performance optimization, and also allows for optional loading of those modules.

The core idea of the pattern is that the `import id = require("...")` statement gives us access to the types exposed by the module. The module loader is invoked (through `require`) dynamically, as shown in the `if` blocks below. This leverages the reference-elision optimization so that the module is only loaded when needed. For this pattern to work, it's important that the symbol defined via an `import` is only used in type positions (i.e. never in a position that would be emitted into the JavaScript).

To maintain type safety, we can use the `typeof` keyword. The `typeof` keyword, when used in a type position, produces the type of a value, in this case the type of the module.

Dynamic Module Loading in Node.js

```

declare function require(moduleName: string): any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
  let ZipCodeValidator: typeof Zip = require("./ZipCodeValidator");
  let validator = new ZipCodeValidator();
  if (validator.isAcceptable("...")) {
    /* ... */
  }
}

```

Sample: Dynamic Module Loading in require.js

```
declare function require(
  moduleNames: string[],
  onLoad: (...args: any[]) => void
): void;

import * as Zip from "./ZipCodeValidator";

if (needZipValidation) {
  require(["./ZipCodeValidator"], (ZipCodeValidator: typeof Zip) => {
    let validator = new ZipCodeValidator.ZipCodeValidator();
    if (validator.isAcceptable("...")) {
      /* ... */
    }
  });
}
```

Sample: Dynamic Module Loading in System.js

```
declare const System: any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
  System.import("./ZipCodeValidator").then((ZipCodeValidator: typeof Zip) => {
    var x = new ZipCodeValidator();
    if (x.isAcceptable("...")) {
      /* ... */
    }
  });
}
```

Working with Other JavaScript Libraries

To describe the shape of libraries not written in TypeScript, we need to declare the API that the library exposes.

We call declarations that don't define an implementation "ambient". Typically, these are defined in `.d.ts` files. If you're familiar with C/C++, you can think of these as `.h` files. Let's look at a few examples.

Ambient Modules

In Node.js, most tasks are accomplished by loading one or more modules. We could define each module in its own `.d.ts` file with top-level export declarations, but it's more convenient to write them as one larger `.d.ts` file. To do so, we use a construct similar to ambient namespaces, but we use the `module` keyword and the quoted name of the module which will be available to a later import. For example:

node.d.ts (simplified excerpt)

```
declare module "url" {
  export interface Url {
    protocol?: string;
    hostname?: string;
    pathname?: string;
```

```

}
export function parse(
  urlStr: string,
  parseQueryString?: boolean,
  slashesDenoteHost?: boolean
): Url;
}

declare module "path" {
  export function normalize(p: string): string;
  export function join(...paths: any[]): string;
  export var sep: string;
}

```

Now we can `/// <reference path="node.d.ts"` and then load the modules using `import url = require("url");` or `import * as URL from "url".`

```

/// <reference path="node.d.ts"/>
import * as URL from "url";
let myUrl = URL.parse("https://www.typescriptlang.org");

```

Shorthand ambient modules

If you don't want to take the time to write out declarations before using a new module, you can use a shorthand declaration to get started quickly.

declarations.d.ts

```
declare module "hot-new-module";
```

All imports from a shorthand module will have the `any` type.

```
import x, { y } from "hot-new-module";
x(y);
```

Wildcard module declarations

Some module loaders such as [SystemJS](#) and [AMD](#) allow non-JavaScript content to be imported. These typically use a prefix or suffix to indicate the special loading semantics. Wildcard module declarations can be used to cover these cases.

```

declare module "*!text" {
  const content: string;
  export default content;
}
// Some do it the other way around.
declare module "json!*" {
  const value: any;
  export default value;
}

```

Now you can import things that match `"*!text"` or `"json!*"`.

```
import fileContent from "./xyz.txt!text";
import data from "json!http://example.com/data.json";
console.log(data, fileContent);
```

UMD modules

Some libraries are designed to be used in many module loaders, or with no module loading (global variables). These are known as [UMD](#) modules. These libraries can be accessed through either an import or a global variable. For example:

`math-lib.d.ts`

```
export function isPrime(x: number): boolean;
export as namespace mathLib;
```

The library can then be used as an import within modules:

```
import { isPrime } from "math-lib";
isPrime(2);
mathLib.isPrime(2); // ERROR: can't use the global definition from inside a module
```

It can also be used as a global variable, but only inside of a script. (A script is a file with no imports or exports.)

```
mathLib.isPrime(2);
```

Guidance for structuring modules

Export as close to top-level as possible

Consumers of your module should have as little friction as possible when using things that you export. Adding too many levels of nesting tends to be cumbersome, so think carefully about how you want to structure things.

Exporting a namespace from your module is an example of adding too many layers of nesting. While namespaces sometimes have their uses, they add an extra level of indirection when using modules. This can quickly become a pain point for users, and is usually unnecessary.

Static methods on an exported class have a similar problem - the class itself adds a layer of nesting. Unless it increases expressivity or intent in a clearly useful way, consider simply exporting a helper function.

If you're only exporting a single `class` or `function`, use `export default`

Just as "exporting near the top-level" reduces friction on your module's consumers, so does introducing a default export. If a module's primary purpose is to house one specific export, then you should consider exporting it as a default export. This makes both importing and actually using the import a little easier. For example:

MyClass.ts

```
export default class SomeType {
  constructor() { ... }
}
```

 MyFunc.ts

```
export default function getThing() {
  return "thing";
}
```

 Consumer.ts

```
import t from "./ MyClass";
import f from "./ MyFunc";
let x = new t();
console.log(f());
```

This is optimal for consumers. They can name your type whatever they want (`t` in this case) and don't have to do any excessive dotting to find your objects.

If you're exporting multiple objects, put them all at top-level **MyThings.ts**

```
export class SomeType {
  /* ... */
}
export function someFunc() {
  /* ... */
}
```

Conversely when importing:

 Explicitly list imported names **Consumer.ts**

```
import { SomeType, someFunc } from "./ MyThings";
let x = new SomeType();
let y = someFunc();
```

 Use the namespace import pattern if you're importing a large number of things **MyLargeModule.ts**

```
export class Dog { ... }
export class Cat { ... }
export class Tree { ... }
export class Flower { ... }
```

Consumer.ts

```
import * as myLargeModule from "./MyLargeModule.ts";
let x = new myLargeModule.Dog();
```

Re-export to extend

Often you will need to extend functionality on a module. A common JS pattern is to augment the original object with *extensions*, similar to how JQuery extensions work. As we've mentioned before, modules do not merge like global namespace objects would. The recommended solution is to *not* mutate the original object, but rather export a new entity that provides the new functionality.

Consider a simple calculator implementation defined in module `Calculator.ts`. The module also exports a helper function to test the calculator functionality by passing a list of input strings and writing the result at the end.

Calculator.ts

```
export class Calculator {
  private current = 0;
  private memory = 0;
  private operator: string;

  protected processDigit(digit: string, currentValue: number) {
    if (digit >= "0" && digit <= "9") {
      return currentValue * 10 + (digit.charCodeAt(0) - "0".charCodeAt(0));
    }
  }

  protected processOperator(operator: string) {
    if (["+", "-", "*", "/"].indexOf(operator) >= 0) {
      return operator;
    }
  }

  protected evaluateOperator(
    operator: string,
    left: number,
    right: number
  ): number {
    switch (this.operator) {
      case "+":
        return left + right;
      case "-":
        return left - right;
      case "*":
        return left * right;
      case "/":
        return left / right;
    }
  }

  private evaluate() {
    if (this.operator) {
      this.memory = this.evaluateOperator(
        this.operator,
        this.memory,
        this.current
      );
    }
  }
}
```

```

        this.current
    );
} else {
    this.memory = this.current;
}
this.current = 0;
}

public handleChar(char: string) {
    if (char === "=") {
        this.evaluate();
        return;
    } else {
        let value = this.processDigit(char, this.current);
        if (value !== undefined) {
            this.current = value;
            return;
        } else {
            let value = this.processOperator(char);
            if (value !== undefined) {
                this.evaluate();
                this.operator = value;
                return;
            }
        }
    }
    throw new Error(`Unsupported input: '${char}'`);
}

public getResult() {
    return this.memory;
}
}

export function test(c: Calculator, input: string) {
    for (let i = 0; i < input.length; i++) {
        c.handleChar(input[i]);
    }
    console.log(`result of '${input}' is '${c.getResult()}'`);
}

```

Here is a simple test for the calculator using the exposed `test` function.

TestCalculator.ts

```

import { Calculator, test } from "./Calculator";
let c = new Calculator();
test(c, "1+2*33/11="); // prints 9

```

Now to extend this to add support for input with numbers in bases other than 10, let's create `ProgrammerCalculator.ts`

ProgrammerCalculator.ts

```

import { Calculator } from "./Calculator";
class ProgrammerCalculator extends Calculator {

```

```

static digits = [
  "0",
  "1",
  "2",
  "3",
  "4",
  "5",
  "6",
  "7",
  "8",
  "9",
  "A",
  "B",
  "C",
  "D",
  "E",
  "F",
];
}

constructor(public base: number) {
  super();
  const maxBase = ProgrammerCalculator.digits.length;
  if (base <= 0 || base > maxBase) {
    throw new Error(`base has to be within 0 to ${maxBase} inclusive.`);
  }
}

protected processDigit(digit: string, currentValue: number) {
  if (ProgrammerCalculator.digits.indexOf(digit) >= 0) {
    return (
      currentValue * this.base + ProgrammerCalculator.digits.indexOf(digit)
    );
  }
}

// Export the new extended calculator as Calculator
export { ProgrammerCalculator as Calculator };

// Also, export the helper function
export { test } from "./Calculator";

```

The new module `ProgrammerCalculator` exports an API shape similar to that of the original `Calculator` module, but does not augment any objects in the original module. Here is a test for our `ProgrammerCalculator` class:

TestProgrammerCalculator.ts

```

import { Calculator, test } from "./ProgrammerCalculator";

let c = new Calculator(2);
test(c, "001+010="); // prints 3

```

Do not use namespaces in modules

When first moving to a module-based organization, a common tendency is to wrap exports in an additional layer of namespaces. Modules have their own scope, and only exported declarations are visible from outside the module. With this in mind, namespaces provide very little, if any, value when working with modules.

On the organization front, namespaces are handy for grouping together logically-related objects and types in the global scope. For example, in C#, you're going to find all the collection types in System.Collections. By organizing our types into hierarchical namespaces, we provide a good "discovery" experience for users of those types. Modules, on the other hand, are already present in a file system, necessarily. We have to resolve them by path and filename, so there's a logical organization scheme for us to use. We can have a /collections/generic/ folder with a list module in it.

Namespaces are important to avoid naming collisions in the global scope. For example, you might have `My.Application.Customer.AddForm` and `My.Application.Order.AddForm` -- two types with the same name, but a different namespace. This, however, is not an issue with modules. Within a module, there's no plausible reason to have two objects with the same name. From the consumption side, the consumer of any given module gets to pick the name that they will use to refer to the module, so accidental naming conflicts are impossible.

For more discussion about modules and namespaces see [Namespaces and Modules](#).

Red Flags

All of the following are red flags for module structuring. Double-check that you're not trying to namespace your external modules if any of these apply to your files:

- A file whose only top-level declaration is `export namespace Foo { ... }` (remove `Foo` and move everything 'up' a level)
- Multiple files that have the same `export namespace Foo {` at top-level (don't think that these are going to combine into one `Foo !`)

[Go to TOC](#)

A note about terminology: It's important to note that in TypeScript 1.5, the nomenclature has changed. "Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with [ECMAScript 2015](#)'s terminology, (namely that `module X { }` is equivalent to the now-preferred `namespace X {}`).

This post outlines the various ways to organize your code using namespaces (previously "internal modules") in TypeScript. As we alluded in our note about terminology, "internal modules" are now referred to as "namespaces". Additionally, anywhere the `module` keyword was used when declaring an internal module, the `namespace` keyword can and should be used instead. This avoids confusing new users by overloading them with similarly named terms.

First steps

Let's start with the program we'll be using as our example throughout this page. We've written a small set of simplistic string validators, as you might write to check a user's input on a form in a webpage or check the format of an externally-provided data file.

Validators in a single file

```
interface StringValidator {
  isAcceptable(s: string): boolean;
}

let lettersRegexp = /^[A-Za-z]+$/;
let numberRegexp = /^[0-9]+$/;

class LettersOnlyValidator implements StringValidator {
  isAcceptable(s: string) {
    return lettersRegexp.test(s);
  }
}

class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: StringValidator } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
  for (let name in validators) {
    let isMatch = validators[name].isAcceptable(s);
```

```
    console.log(`"${s}" ${isMatch ? "matches" : "does not match"} "${name}"`);  
}
```

Namespacing

As we add more validators, we're going to want to have some kind of organization scheme so that we can keep track of our types and not worry about name collisions with other objects. Instead of putting lots of different names into the global namespace, let's wrap up our objects into a namespace.

In this example, we'll move all validator-related entities into a namespace called `Validation`. Because we want the interfaces and classes here to be visible outside the namespace, we preface them with `export`. Conversely, the variables `lettersRegexp` and `numberRegexp` are implementation details, so they are left unexported and will not be visible to code outside the namespace. In the test code at the bottom of the file, we now need to qualify the names of the types when used outside the namespace, e.g. `Validation.LettersOnlyValidator`.

Namespaced Validators

```
namespace Validation {  
  export interface StringValidator {  
    isAcceptable(s: string): boolean;  
  }  
  
  const lettersRegexp = /^[A-Za-z]+$/;  
  const numberRegexp = /^[0-9]+$/;  
  
  export class LettersOnlyValidator implements StringValidator {  
    isAcceptable(s: string) {  
      return lettersRegexp.test(s);  
    }  
  }  
  
  export class ZipCodeValidator implements StringValidator {  
    isAcceptable(s: string) {  
      return s.length === 5 && numberRegexp.test(s);  
    }  
  }  
}  
  
// Some samples to try  
let strings = ["Hello", "98052", "101"];  
  
// Validators to use  
let validators: { [s: string]: Validation.StringValidator } = {};  
validators["ZIP code"] = new Validation.ZipCodeValidator();  
validators["Letters only"] = new Validation.LettersOnlyValidator();  
  
// Show whether each string passed each validator  
for (let s of strings) {  
  for (let name in validators) {  
    console.log(`"${s}" - ${  
      validators[name].isAcceptable(s) ? "matches" : "does not match"  
    } ${name}`)  
  }  
}
```

```
    );
}
```

Splitting Across Files

As our application grows, we'll want to split the code across multiple files to make it easier to maintain.

Multi-file namespaces

Here, we'll split our `Validation` namespace across many files. Even though the files are separate, they can each contribute to the same namespace and can be consumed as if they were all defined in one place. Because there are dependencies between files, we'll add reference tags to tell the compiler about the relationships between the files. Our test code is otherwise unchanged.

Validation.ts

```
namespace Validation {
  export interface StringValidator {
    isAcceptable(s: string): boolean;
  }
}
```

LettersOnlyValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
  const lettersRegexp = /^[A-Za-z]+$/;
  export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
      return lettersRegexp.test(s);
    }
  }
}
```

ZipCodeValidator.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
  const numberRegexp = /^[0-9]+$/;
  export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
      return s.length === 5 && numberRegexp.test(s);
    }
  }
}
```

Test.ts

```
/// <reference path="Validation.ts" />
/// <reference path="LettersOnlyValidator.ts" />
/// <reference path="ZipCodeValidator.ts" />

// Some samples to try
let strings = ["Hello", "98052", "101"];
```

```
// Validators to use
let validators: { [s: string]: Validation.StringValidator } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
  for (let name in validators) {
    console.log(`"${s}" - ${validators[name].isAcceptable(s) ? "matches" : "does not match"} ${name}`);
  }
}
```

Once there are multiple files involved, we'll need to make sure all of the compiled code gets loaded. There are two ways of doing this.

First, we can use concatenated output using the `outFile` option to compile all of the input files into a single JavaScript output file:

```
tsc --outFile sample.js Test.ts
```

The compiler will automatically order the output file based on the reference tags present in the files. You can also specify each file individually:

```
tsc --outFile sample.js Validation.ts LettersOnlyValidator.ts ZipCodeValidator.ts
Test.ts
```

Alternatively, we can use per-file compilation (the default) to emit one JavaScript file for each input file. If multiple JS files get produced, we'll need to use `<script>` tags on our webpage to load each emitted file in the appropriate order, for example:

MyTestPage.html (excerpt)

```
<script src="Validation.js" type="text/javascript" />
<script src="LettersOnlyValidator.js" type="text/javascript" />
<script src="ZipCodeValidator.js" type="text/javascript" />
<script src="Test.js" type="text/javascript" />
```

Aliases

Another way that you can simplify working with namespaces is to use `import q = x.y.z` to create shorter names for commonly-used objects. Not to be confused with the `import x = require("name")` syntax used to load modules, this syntax simply creates an alias for the specified symbol. You can use these sorts of imports (commonly referred to as aliases) for any kind of identifier, including objects created from module imports.

```
namespace Shapes {
  export namespace Polygons {
    export class Triangle {}
    export class Square {}
  }
}
```

```

    }
}

import polygons = Shapes.Polygons;
let sq = new polygons.Square(); // Same as 'new Shapes.Polygons.Square()'

```

Notice that we don't use the `require` keyword; instead we assign directly from the qualified name of the symbol we're importing. This is similar to using `var`, but also works on the type and namespace meanings of the imported symbol. Importantly, for values, `import` is a distinct reference from the original symbol, so changes to an aliased `var` will not be reflected in the original variable.

Working with Other JavaScript Libraries

To describe the shape of libraries not written in TypeScript, we need to declare the API that the library exposes. Because most JavaScript libraries expose only a few top-level objects, namespaces are a good way to represent them.

We call declarations that don't define an implementation "ambient". Typically these are defined in `.d.ts` files. If you're familiar with C/C++, you can think of these as `.h` files. Let's look at a few examples.

Ambient Namespaces

The popular library D3 defines its functionality in a global object called `d3`. Because this library is loaded through a `<script>` tag (instead of a module loader), its declaration uses namespaces to define its shape. For the TypeScript compiler to see this shape, we use an ambient namespace declaration. For example, we could begin writing it as follows:

`D3.d.ts` (simplified excerpt)

```

declare namespace D3 {
  export interface Selectors {
    select: {
      (selector: string): Selection;
      (element: EventTarget): Selection;
    };
  }

  export interface Event {
    x: number;
    y: number;
  }

  export interface Base extends Selectors {
    event: Event;
  }
}

declare var d3: D3.Base;

```

[Go to TOC](#)

This post outlines the various ways to organize your code using modules and namespaces in TypeScript. We'll also go over some advanced topics of how to use namespaces and modules, and address some common pitfalls when using them in TypeScript.

See the [Modules](#) documentation for more information about ES Modules. See the [Namespaces](#) documentation for more information about TypeScript namespaces.

Note: In *very old* versions of TypeScript namespaces were called 'Internal Modules', these pre-date JavaScript module systems.

Using Modules

Modules can contain both code and declarations.

Modules also have a dependency on a module loader (such as CommonJs/Require.js) or a runtime which supports ES Modules. Modules provide for better code reuse, stronger isolation and better tooling support for bundling.

It is also worth noting that, for Node.js applications, modules are the default and **we recommended modules over namespaces in modern code**.

Starting with ECMAScript 2015, modules are native part of the language, and should be supported by all compliant engine implementations. Thus, for new projects modules would be the recommended code organization mechanism.

Using Namespaces

Namespaces are a TypeScript-specific way to organize code.

Namespaces are simply named JavaScript objects in the global namespace. This makes namespaces a very simple construct to use. Unlike modules, they can span multiple files, and can be concatenated using `out-file`. Namespaces can be a good way to structure your code in a Web Application, with all dependencies included as `<script>` tags in your HTML page.

Just like all global namespace pollution, it can be hard to identify component dependencies, especially in a large application.

Pitfalls of Namespaces and Modules

In this section we'll describe various common pitfalls in using namespaces and modules, and how to avoid them.

/// <reference>-ing a module

A common mistake is to try to use the `/// <reference ... />` syntax to refer to a module file, rather than using an `import` statement. To understand the distinction, we first need to understand how the compiler can locate the type information for a module based on the path of an `import` (e.g. the `...` in `import x from "..."; import x = require("...");`, etc.) path.

The compiler will try to find a `.ts`, `.tsx`, and then a `.d.ts` with the appropriate path. If a specific file could not be found, then the compiler will look for an *ambient module declaration*. Recall that these need to be declared in a `.d.ts` file.

- `myModules.d.ts`

```
// In a .d.ts file or .ts file that is not a module:
declare module "SomeModule" {
    export function fn(): string;
}
```

- `myOtherModule.ts`

```
/// <reference path="myModules.d.ts" />
import * as m from "SomeModule";
```

The reference tag here allows us to locate the declaration file that contains the declaration for the ambient module. This is how the `node.d.ts` file that several of the TypeScript samples use is consumed.

Needless Namespacing

If you're converting a program from namespaces to modules, it can be easy to end up with a file that looks like this:

- `shapes.ts`

```
export namespace Shapes {
    export class Triangle {
        /* ... */
    }
    export class Square {
        /* ... */
    }
}
```

The top-level namespace here `Shapes` wraps up `Triangle` and `Square` for no reason. This is confusing and annoying for consumers of your module:

- `shapeConsumer.ts`

```
import * as shapes from "./shapes";
let t = new shapes.Shapes.Triangle(); // shapes.Shapes?
```

A key feature of modules in TypeScript is that two different modules will never contribute names to the same scope. Because the consumer of a module decides what name to assign it, there's no need to proactively wrap up the exported symbols in a namespace.

To reiterate why you shouldn't try to namespace your module contents, the general idea of namespacing is to provide logical grouping of constructs and to prevent name collisions. Because the module file itself is already a logical grouping, and its top-level name is defined by the code that imports it, it's unnecessary to use an additional module layer for exported objects.

Here's a revised example:

- `shapes.ts`

```
export class Triangle {
  /* ... */
}
export class Square {
  /* ... */
}
```

- `shapeConsumer.ts`

```
import * as shapes from "./shapes";
let t = new shapes.Triangle();
```

Trade-offs of Modules

Just as there is a one-to-one correspondence between JS files and modules, TypeScript has a one-to-one correspondence between module source files and their emitted JS files. One effect of this is that it's not possible to concatenate multiple module source files depending on the module system you target. For instance, you can't use the `outFile` option while targeting `commonjs` or `umd` , but with TypeScript 1.8 and later, it's possible to use `outFile` when targeting `amd` or `system` .

[Go to TOC](#)

Starting with ECMAScript 2015, `symbol` is a primitive data type, just like `number` and `string`.

`symbol` values are created by calling the `Symbol` constructor.

```
let sym1 = Symbol();
let sym2 = Symbol("key"); // optional string key
```

Symbols are immutable, and unique.

```
let sym2 = Symbol("key");
let sym3 = Symbol("key");

sym2 === sym3; // false, symbols are unique
```

Just like strings, symbols can be used as keys for object properties.

```
const sym = Symbol();

let obj = {
  [sym]: "value",
};

console.log(obj[sym]); // "value"
```

Symbols can also be combined with computed property declarations to declare object properties and class members.

```
const getClassnameSymbol = Symbol();

class C {
  [getClassnameSymbol]() {
    return "C";
  }
}

let c = new C();
let className = c[getClassnameSymbol](); // "C"
```

unique symbol

To enable treating symbols as unique literals a special type `unique symbol` is available. `unique symbol` is a subtype of `symbol`, and are produced only from calling `Symbol()` or `Symbol.for()`, or from explicit type annotations. This type is only allowed on `const` declarations and `readonly static` properties, and in order to reference a specific unique symbol, you'll have to use the `typeof` operator. Each reference to a unique symbol implies a completely unique identity that's tied to a given declaration.

```
// @errors: 1332
declare const sym1: unique symbol;

// sym2 can only be a constant reference.
let sym2: unique symbol = Symbol();

// Works - refers to a unique symbol, but its identity is tied to 'sym1'.
```

```
let sym3: typeof sym1 = sym1;

// Also works.
class C {
  static readonly StaticSymbol: unique symbol = Symbol();
}
```

Because each `unique symbol` has a completely separate identity, no two `unique symbol` types are assignable or comparable to each other.

```
// @errors: 2367
const sym2 = Symbol();
const sym3 = Symbol();

if (sym2 === sym3) {
  // ...
}
```

Well-known Symbols

In addition to user-defined symbols, there are well-known built-in symbols. Built-in symbols are used to represent internal language behaviors.

Here is a list of well-known symbols:

`Symbol.hasInstance`

A method that determines if a constructor object recognizes an object as one of the constructor's instances. Called by the semantics of the `instanceof` operator.

`Symbol.isConcatSpreadable`

A Boolean value indicating that an object should be flattened to its array elements by `Array.prototype.concat`.

`Symbol.iterator`

A method that returns the default iterator for an object. Called by the semantics of the `for-of` statement.

`Symbol.match`

A regular expression method that matches the regular expression against a string. Called by the `String.prototype.match` method.

`Symbol.replace`

A regular expression method that replaces matched substrings of a string. Called by the `String.prototype.replace` method.

Symbol.search

A regular expression method that returns the index within a string that matches the regular expression. Called by the `String.prototype.search` method.

Symbol.species

A function valued property that is the constructor function that is used to create derived objects.

Symbol.split

A regular expression method that splits a string at the indices that match the regular expression. Called by the `String.prototype.split` method.

Symbol.toPrimitive

A method that converts an object to a corresponding primitive value. Called by the `ToPrimitive` abstract operation.

Symbol.toStringTag

A String value that is used in the creation of the default string description of an object. Called by the built-in method `Object.prototype.toString`.

Symbol.unscopables

An Object whose own property names are property names that are excluded from the 'with' environment bindings of the associated objects.

[Go to TOC](#)

Triple-slash directives are single-line comments containing a single XML tag. The contents of the comment are used as compiler directives.

Triple-slash directives are **only** valid at the top of their containing file. A triple-slash directive can only be preceded by single or multi-line comments, including other triple-slash directives. If they are encountered following a statement or a declaration they are treated as regular single-line comments, and hold no special meaning.

```
/// <reference path="..." />
```

The `/// <reference path="..." />` directive is the most common of this group. It serves as a declaration of *dependency* between files.

Triple-slash references instruct the compiler to include additional files in the compilation process.

They also serve as a method to order the output when using `out` or `outFile`. Files are emitted to the output file location in the same order as the input after preprocessing pass.

Preprocessing input files

The compiler performs a preprocessing pass on input files to resolve all triple-slash reference directives. During this process, additional files are added to the compilation.

The process starts with a set of *root files*; these are the file names specified on the command-line or in the `files` list in the `tsconfig.json` file. These root files are preprocessed in the same order they are specified. Before a file is added to the list, all triple-slash references in it are processed, and their targets included. Triple-slash references are resolved in a depth-first manner, in the order they have been seen in the file.

A triple-slash reference path is resolved relative to the containing file, if a relative path is used.

Errors

It is an error to reference a file that does not exist. It is an error for a file to have a triple-slash reference to itself.

Using `--noResolve`

If the compiler flag `noResolve` is specified, triple-slash references are ignored; they neither result in adding new files, nor change the order of the files provided.

```
/// <reference types="..." />
```

Similar to a `/// <reference path="..." />` directive, which serves as a declaration of *dependency*, a `/// <reference types="..." />` directive declares a dependency on a package.

The process of resolving these package names is similar to the process of resolving module names in an `import` statement. An easy way to think of triple-slash-reference-types directives are as an `import` for declaration packages.

For example, including `/// <reference types="node" />` in a declaration file declares that this file uses names declared in `@types/node/index.d.ts`; and thus, this package needs to be included in the compilation along with the declaration file.

Use these directives only when you're authoring a `.d.ts` file by hand.

For declaration files generated during compilation, the compiler will automatically add `/// <reference types="..." />` for you; A `/// <reference types="..." />` in a generated declaration file is added if and only if the resulting file uses any declarations from the referenced package.

For declaring a dependency on an `@types` package in a `.ts` file, use `types` on the command line or in your `tsconfig.json` instead. See [using `@types`, `typeRoots` and `types` in `tsconfig.json` files](#) for more details.

```
/// <reference lib="..." />
```

This directive allows a file to explicitly include an existing built-in `lib` file.

Built-in `lib` files are referenced in the same fashion as the `lib` compiler option in `tsconfig.json` (e.g. use `lib="es2015"` and not `lib="lib.es2015.d.ts"`, etc.).

For declaration file authors who rely on built-in types, e.g. DOM APIs or built-in JS run-time constructors like `Symbol` or `Iterable`, triple-slash-reference `lib` directives are recommended. Previously these `.d.ts` files had to add forward/duplicate declarations of such types.

For example, adding `/// <reference lib="es2017.string" />` to one of the files in a compilation is equivalent to compiling with `--lib es2017.string`.

```
/// <reference lib="es2017.string" />
"foo".padStart(4);
```

```
/// <reference no-default-lib="true" />
```

This directive marks a file as a *default library*. You will see this comment at the top of `lib.d.ts` and its different variants.

This directive instructs the compiler to *not* include the default library (i.e. `lib.d.ts`) in the compilation. The impact here is similar to passing `noLib` on the command line.

Also note that when passing `skipDefaultLibCheck`, the compiler will only skip checking files with `/// <reference no-default-lib="true" />`.

```
/// <amd-module />
```

By default AMD modules are generated anonymous. This can lead to problems when other tools are used to process the resulting modules, such as bundlers (e.g. `r.js`).

The `amd-module` directive allows passing an optional module name to the compiler:

amdModule.ts

```
///<amd-module name="NamedModule"/>
export class C {}
```

Will result in assigning the name `NamedModule` to the module as part of calling the AMD `define`:

amdModule.js

```
define("NamedModule", ["require", "exports"], function (require, exports) {
  var C = (function () {
    function C() {}
    return C;
  })();
  exports.C = C;
});
```

`/// <amd-dependency />`

Note: this directive has been deprecated. Use `import "moduleName";` statements instead.

`/// <amd-dependency path="x" />` informs the compiler about a non-TS module dependency that needs to be injected in the resulting module's require call.

The `amd-dependency` directive can also have an optional `name` property; this allows passing an optional name for an amd-dependency:

```
/// <amd-dependency path="legacy/moduleA" name="moduleA"/>
declare var moduleA: MyType;
moduleA.callStuff();
```

Generated JS code:

```
define(["require", "exports", "legacy/moduleA"], function (
  require,
  exports,
  moduleA
) {
  moduleA.callStuff();
});
```

[Go to TOC](#)

Type compatibility in TypeScript is based on structural subtyping. Structural typing is a way of relating types based solely on their members. This is in contrast with nominal typing. Consider the following code:

```
interface Pet {
  name: string;
}

class Dog {
  name: string;
}

let pet: Pet;
// OK, because of structural typing
pet = new Dog();
```

In nominally-typed languages like C# or Java, the equivalent code would be an error because the `Dog` class does not explicitly describe itself as being an implementer of the `Pet` interface.

TypeScript's structural type system was designed based on how JavaScript code is typically written. Because JavaScript widely uses anonymous objects like function expressions and object literals, it's much more natural to represent the kinds of relationships found in JavaScript libraries with a structural type system instead of a nominal one.

A Note on Soundness

TypeScript's type system allows certain operations that can't be known at compile-time to be safe. When a type system has this property, it is said to not be "sound". The places where TypeScript allows unsound behavior were carefully considered, and throughout this document we'll explain where these happen and the motivating scenarios behind them.

Starting out

The basic rule for TypeScript's structural type system is that `x` is compatible with `y` if `y` has at least the same members as `x`. For example consider the following code involving an interface named `Pet` which has a `name` property:

```
interface Pet {
  name: string;
}

let pet: Pet;
// dog's inferred type is { name: string; owner: string; }
let dog = { name: "Lassie", owner: "Rudd Weatherwax" };
pet = dog;
```

To check whether `dog` can be assigned to `pet`, the compiler checks each property of `pet` to find a corresponding compatible property in `dog`. In this case, `dog` must have a member called `name` that is a string. It does, so the assignment is allowed.

The same rule for assignment is used when checking function call arguments:

```

interface Pet {
  name: string;
}

let dog = { name: "Lassie", owner: "Rudd Weatherwax" };

function greet(pet: Pet) {
  console.log("Hello, " + pet.name);
}
greet(dog); // OK

```

Note that `dog` has an extra `owner` property, but this does not create an error. Only members of the target type (`Pet` in this case) are considered when checking for compatibility.

This comparison process proceeds recursively, exploring the type of each member and sub-member.

Comparing two functions

While comparing primitive types and object types is relatively straightforward, the question of what kinds of functions should be considered compatible is a bit more involved. Let's start with a basic example of two functions that differ only in their parameter lists:

```

let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error

```

To check if `x` is assignable to `y`, we first look at the parameter list. Each parameter in `x` must have a corresponding parameter in `y` with a compatible type. Note that the names of the parameters are not considered, only their types. In this case, every parameter of `x` has a corresponding compatible parameter in `y`, so the assignment is allowed.

The second assignment is an error, because `y` has a required second parameter that `x` does not have, so the assignment is disallowed.

You may be wondering why we allow 'discarding' parameters like in the example `y = x`. The reason for this assignment to be allowed is that ignoring extra function parameters is actually quite common in JavaScript. For example, `Array#forEach` provides three parameters to the callback function: the array element, its index, and the containing array. Nevertheless, it's very useful to provide a callback that only uses the first parameter:

```

let items = [1, 2, 3];

// Don't force these extra parameters
items.forEach((item, index, array) => console.log(item));

// Should be OK!
items.forEach((item) => console.log(item));

```

Now let's look at how return types are treated, using two functions that differ only by their return type:

```
let x = () => ({ name: "Alice" });
let y = () => ({ name: "Alice", location: "Seattle" });

x = y; // OK
y = x; // Error, because x() lacks a location property
```

The type system enforces that the source function's return type be a subtype of the target type's return type.

Function Parameter Bivariance

When comparing the types of function parameters, assignment succeeds if either the source parameter is assignable to the target parameter, or vice versa. This is unsound because a caller might end up being given a function that takes a more specialized type, but invokes the function with a less specialized type. In practice, this sort of error is rare, and allowing this enables many common JavaScript patterns. A brief example:

```
enum EventType {
  Mouse,
  Keyboard,
}

interface Event {
  timestamp: number;
}
interface MyMouseEvent extends Event {
  x: number;
  y: number;
}
interface MyKeyEvent extends Event {
  keyCode: number;
}

function listenEvent(eventType: EventType, handler: (e: Event) => void) {
  /* ... */
}

// Unsound, but useful and common
listenEvent(EventType.Mouse, (e: MyMouseEvent) => console.log(e.x + "," + e.y));

// Undesirable alternatives in presence of soundness
listenEvent(EventType.Mouse, (e: Event) =>
  console.log((e as MyMouseEvent).x + "," + (e as MyMouseEvent).y));
listenEvent(EventType.Mouse, ((e: MyMouseEvent) =>
  console.log(e.x + "," + e.y)) as (e: Event) => void);

// Still disallowed (clear error). Type safety enforced for wholly incompatible
// types
listenEvent(EventType.Mouse, (e: number) => console.log(e));
```

You can have TypeScript raise errors when this happens via the compiler flag `strictFunctionTypes`.

Optional Parameters and Rest Parameters

When comparing functions for compatibility, optional and required parameters are interchangeable. Extra optional parameters of the source type are not an error, and optional parameters of the target type without corresponding parameters in the source type are not an error.

When a function has a rest parameter, it is treated as if it were an infinite series of optional parameters.

This is unsound from a type system perspective, but from a runtime point of view the idea of an optional parameter is generally not well-enforced since passing `undefined` in that position is equivalent for most functions.

The motivating example is the common pattern of a function that takes a callback and invokes it with some predictable (to the programmer) but unknown (to the type system) number of arguments:

```
function invokeLater(args: any[], callback: (...args: any[]) => void) {
  /* ... Invoke callback with 'args' ... */
}

// Unsound - invokeLater "might" provide any number of arguments
invokeLater([1, 2], (x, y) => console.log(x + ", " + y));

// Confusing (x and y are actually required) and undiscoverable
invokeLater([1, 2], (x?, y?) => console.log(x + ", " + y));
```

Functions with overloads

When a function has overloads, each overload in the source type must be matched by a compatible signature on the target type. This ensures that the target function can be called in all the same situations as the source function.

Enums

Enums are compatible with numbers, and numbers are compatible with enums. Enum values from different enum types are considered incompatible. For example,

```
enum Status {
  Ready,
  Waiting,
}
enum Color {
  Red,
  Blue,
  Green,
}

let status = Status.Ready;
status = Color.Green; // Error
```

Classes

Classes work similarly to object literal types and interfaces with one exception: they have both a static and an instance type. When comparing two objects of a class type, only members of the instance are compared. Static members and constructors do not affect compatibility.

```
class Animal {
  feet: number;
  constructor(name: string, numFeet: number) {}
}

class Size {
  feet: number;
  constructor(numFeet: number) {}
}

let a: Animal;
let s: Size;

a = s; // OK
s = a; // OK
```

Private and protected members in classes

Private and protected members in a class affect their compatibility. When an instance of a class is checked for compatibility, if the target type contains a private member, then the source type must also contain a private member that originated from the same class. Likewise, the same applies for an instance with a protected member. This allows a class to be assignment compatible with its super class, but *not* with classes from a different inheritance hierarchy which otherwise have the same shape.

Generics

Because TypeScript is a structural type system, type parameters only affect the resulting type when consumed as part of the type of a member. For example,

```
interface Empty<T> {}
let x: Empty<number>;
let y: Empty<string>;

x = y; // OK, because y matches structure of x
```

In the above, `x` and `y` are compatible because their structures do not use the type argument in a differentiating way. Changing this example by adding a member to `Empty<T>` shows how this works:

```
interface NotEmpty<T> {
  data: T;
}
let x: NotEmpty<number>;
let y: NotEmpty<string>;

x = y; // Error, because x and y are not compatible
```

In this way, a generic type that has its type arguments specified acts just like a non-generic type.

For generic types that do not have their type arguments specified, compatibility is checked by specifying `any` in place of all unspecified type arguments. The resulting types are then checked for compatibility, just as in the non-generic case.

For example,

```
let identity = function <T>(x: T): T {
  // ...
};

let reverse = function <U>(y: U): U {
  // ...
};

identity = reverse; // OK, because (x: any) => any matches (y: any) => any
```

Advanced Topics

Subtype vs Assignment

So far, we've used "compatible", which is not a term defined in the language spec. In TypeScript, there are two kinds of compatibility: subtype and assignment. These differ only in that assignment extends subtype compatibility with rules to allow assignment to and from `any`, and to and from `enum` with corresponding numeric values.

Different places in the language use one of the two compatibility mechanisms, depending on the situation. For practical purposes, type compatibility is dictated by assignment compatibility, even in the cases of the `implements` and `extends` clauses.

Any, unknown, object, void, undefined, null, and never assignability

The following table summarizes assignability between some abstract types. Rows indicate what each is assignable to, columns indicate what is assignable to them. A "✓" indicates a combination that is compatible only when `strictNullChecks` is off.

	any	unknown	object	void	undefined	null	never
any →	✓		✓	✓	✓	✓	✗
unknown →	✓		✗	✗	✗	✗	✗
object →	✓	✓		✗	✗	✗	✗
void →	✓	✓	✗		✗	✗	✗
undefined →	✓	✓	✓	✓		✓	✗
null →	✓	✓	✓	✓	✓		✗

	<code>any</code>	<code>unknown</code>	<code>object</code>	<code>void</code>	<code>undefined</code>	<code>null</code>	<code>never</code>
<code>never</code> →	✓	✓	✓	✓	✓	✓	

Reiterating [The Basics](#):

- Everything is assignable to itself.
- `any` and `unknown` are the same in terms of what is assignable to them, different in that `unknown` is not assignable to anything except `any`.
- `unknown` and `never` are like inverses of each other. Everything is assignable to `unknown`, `never` is assignable to everything. Nothing is assignable to `never`, `unknown` is not assignable to anything (except `any`).
- `void` is not assignable to or from anything, with the following exceptions: `any`, `unknown`, `never`, `undefined`, and `null` (if `strictNullChecks` is off, see table for details).
- When `strictNullChecks` is off, `null` and `undefined` are similar to `never`: assignable to most types, most types are not assignable to them. They are assignable to each other.
- When `strictNullChecks` is on, `null` and `undefined` behave more like `void`: not assignable to or from anything, except for `any`, `unknown`, `never`, and `void` (`undefined` is always assignable to `void`).

[Go to TOC](#)

In TypeScript, there are several places where type inference is used to provide type information when there is no explicit type annotation. For example, in this code

```
let x = 3;
// ^?
```

The type of the `x` variable is inferred to be `number`. This kind of inference takes place when initializing variables and members, setting parameter default values, and determining function return types.

In most cases, type inference is straightforward. In the following sections, we'll explore some of the nuances in how types are inferred.

Best common type

When a type inference is made from several expressions, the types of those expressions are used to calculate a "best common type". For example,

```
let x = [0, 1, null];
// ^?
```

To infer the type of `x` in the example above, we must consider the type of each array element. Here we are given two choices for the type of the array: `number` and `null`. The best common type algorithm considers each candidate type, and picks the type that is compatible with all the other candidates.

Because the best common type has to be chosen from the provided candidate types, there are some cases where types share a common structure, but no one type is the super type of all candidate types. For example:

```
// @strict: false
class Animal {}
class Rhino extends Animal {
    hasHorn: true;
}
class Elephant extends Animal {
    hasTrunk: true;
}
class Snake extends Animal {
    hasLegs: false;
}
// ---cut---
let zoo = [new Rhino(), new Elephant(), new Snake()];
// ^?
```

Ideally, we may want `zoo` to be inferred as an `Animal[]`, but because there is no object that is strictly of type `Animal` in the array, we make no inference about the array element type. To correct this, instead explicitly provide the type when no one type is a super type of all other candidates:

```
// @strict: false
class Animal {}
class Rhino extends Animal {
    hasHorn: true;
}
```

```

class Elephant extends Animal {
  hasTrunk: true;
}
class Snake extends Animal {
  hasLegs: false;
}
// ---cut---
let zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
// ^?

```

When no best common type is found, the resulting inference is the union array type, `(Rhino | Elephant | Snake)[]`.

Contextual Typing

Type inference also works in "the other direction" in some cases in TypeScript. This is known as "contextual typing". Contextual typing occurs when the type of an expression is implied by its location. For example:

```

// @errors: 2339
window.onmousedown = function (MouseEvent) {
  console.log(MouseEvent.button);
  console.log(MouseEvent.kangaroo);
};

```

Here, the TypeScript type checker used the type of the `window.onmousedown` function to infer the type of the function expression on the right hand side of the assignment. When it did so, it was able to infer the type of the `MouseEvent` parameter, which does contain a `button` property, but not a `kangaroo` property.

This works because `window` already has `onmousedown` declared in its type:

```

// Declares there is a global variable called 'window'
declare var window: Window & typeof globalThis;

// Which is declared as (simplified):
interface Window extends GlobalEventHandlers {
  // ...
}

// Which defines a lot of known handler events
interface GlobalEventHandlers {
  onmousedown: ((this: GlobalEventHandlers, ev: MouseEvent) => any) | null;
  // ...
}

```

TypeScript is smart enough to infer types in other contexts as well:

```

// @errors: 2339
window.onscroll = function (uiEvent) {
  console.log(uiEvent.button);
};

```

Based on the fact that the above function is being assigned to `window.onscroll`, TypeScript knows that `uiEvent` is a `UIEvent`, and not a `MouseEvent` like the previous example. `UIEvent` objects contain no `button` property, and so TypeScript will throw an error.

If this function were not in a contextually typed position, the function's argument would implicitly have type `any`, and no error would be issued (unless you are using the `noImplicitAny` option):

```
// @noImplicitAny: false
const handler = function (uiEvent) {
  console.log(uiEvent.button); // <- OK
};
```

We can also explicitly give type information to the function's argument to override any contextual type:

```
window.onscroll = function (uiEvent: any) {
  console.log(uiEvent.button); // <- Now, no error is given
};
```

However, this code will log `undefined`, since `uiEvent` has no property called `button`.

Contextual typing applies in many cases. Common cases include arguments to function calls, right hand sides of assignments, type assertions, members of object and array literals, and return statements. The contextual type also acts as a candidate type in best common type. For example:

```
// @strict: false
class Animal {}
class Rhino extends Animal {
  hasHorn: true;
}
class Elephant extends Animal {
  hasTrunk: true;
}
class Snake extends Animal {
  hasLegs: false;
}
// ---cut---
function createZoo(): Animal[] {
  return [new Rhino(), new Elephant(), new Snake()];
}
```

In this example, best common type has a set of four candidates: `Animal`, `Rhino`, `Elephant`, and `Snake`. Of these, `Animal` can be chosen by the best common type algorithm.

[Go to TOC](#)

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 `Promises` - specifically, the way that they recursively unwrap `Promises`.

Example

```
type A = Awaited<Promise<string>>;
// ^?

type B = Awaited<Promise<Promise<number>>>;
// ^?

type C = Awaited<boolean | Promise<number>>;
// ^?
```

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",
});
```

Required<Type>

Released:

2.8

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

Example

```
// @errors: 2741
interface Props {
  a?: number;
  b?: string;
}

const obj: Props = { a: 5 };
const obj2: Required = { a: 5 };
```

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

```
// @errors: 2540
interface Todo {
  title: string;
}

const todo: Readonly<Todo> = {
  title: "Delete inactive users",
};

todo.title = "Hello";
```

This utility is useful for representing assignment expressions that will fail at runtime (i.e. when attempting to reassign properties of a [frozen object](#)).

`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`. This utility can be used to map the properties of a type to another type.

Example

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

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

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

cats.boris;
// ^?
```

Pick<Type, Keys>

Released:

2.1

Constructs a type by picking the set of properties `Keys` (string literal or union of string literals) 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,
};

todo;
// ^?
```

Omit<Type, Keys>

Released:

3.5

Constructs a type by picking all properties from `Type` and then removing `Keys` (string literal or union of string literals).

Example

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

type TodoPreview = Omit<Todo, "description">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false,
  createdAt: 161554252770,
};

todo;
// ^?

type TodoInfo = Omit<Todo, "completed" | "createdAt">;

const todoInfo: TodoInfo = {
  title: "Pick up kids",
  description: "Kindergarten closes at 5pm",
};

todoInfo;
// ^?
```

Exclude<UnionType, ExcludedMembers>

Released:

2.8

Constructs a type by excluding from `UnionType` all union members that are assignable to `ExcludedMembers`.

Example

```
type T0 = Exclude<"a" | "b" | "c", "a">;
// ^
type T1 = Exclude<"a" | "b" | "c", "a" | "b">;
// ^
type T2 = Exclude<string | number | (() => void), Function>;
// ^?
```

Extract<Type, Union>

Released:

2.8

Constructs a type by extracting from `Type` all union members that are assignable to `Union`.

Example

```
type T0 = Extract<"a" | "b" | "c", "a" | "f">;
// ^
type T1 = Extract<string | number | (() => void), Function>;
// ^?
```

NonNullable<Type>

Released:

2.8

Constructs a type by excluding `null` and `undefined` from `Type`.

Example

```
type T0 = NonNullable<string | number | undefined>;
// ^
type T1 = NonNullable<string[] | null | undefined>;
// ^?
```

Parameters<Type>

Released:

3.1

Constructs a tuple type from the types used in the parameters of a function type `Type`.

Example

```
// @errors: 2344
declare function f1(arg: { a: number; b: string }): void;

type T0 = Parameters<() => string>;
//   ^
type T1 = Parameters<(s: string) => void>;
//   ^
type T2 = Parameters<<T>(arg: T) => T>;
//   ^
type T3 = Parameters<typeof f1>;
//   ^
type T4 = Parameters<any>;
//   ^
type T5 = Parameters<never>;
//   ^
type T6 = Parameters<string>;
//   ^
type T7 = Parameters<Function>;
//   ^?
```

ConstructorParameters<Type>

Released:

3.1

Constructs a tuple or array type from the types of a constructor function type. It produces a tuple type with all the parameter types (or the type `never` if `Type` is not a function).

Example

```
// @errors: 2344
// @strict: false
type T0 = ConstructorParameters<ErrorConstructor>;
//   ^
type T1 = ConstructorParameters<FunctionConstructor>;
//   ^
type T2 = ConstructorParameters<RegExpConstructor>;
//   ^
type T3 = ConstructorParameters<any>;
//   ^
type T4 = ConstructorParameters<Function>;
//   ^?
```

ReturnType<Type>

Released:

2.8

Constructs a type consisting of the return type of function `Type`.

Example

```
// @errors: 2344 2344
declare function f1(): { a: number; b: string };

type T0 = ReturnType<() => string>;
// ^
type T1 = ReturnType<(s: string) => void>;
// ^
type T2 = ReturnType<<T extends U, U extends number[]>() => T>;
// ^
type T3 = ReturnType<<T extends U, U extends number[]>() => T>;
// ^
type T4 = ReturnType<typeof f1>;
// ^
type T5 = ReturnType<any>;
// ^
type T6 = ReturnType<never>;
// ^
type T7 = ReturnType<string>;
// ^
type T8 = ReturnType<Function>;
// ^?
```

InstanceType<Type>

Released:

2.8

Constructs a type consisting of the instance type of a constructor function in `Type`.

Example

```
// @errors: 2344 2344
// @strict: false
class C {
  x = 0;
  y = 0;
}

type T0 = InstanceType<typeof C>;
// ^
type T1 = InstanceType<any>;
// ^
type T2 = InstanceType<never>;
// ^
type T3 = InstanceType<string>;
// ^
type T4 = InstanceType<Function>;
// ^?
```

ThisParameterType<Type>

Released:

3.3

Extracts the type of the `this` parameter for a function type, or `unknown` if the function type has no `this` parameter.

Example

```
function toHex(this: Number) {
  return this.toString(16);
}

function numberToString(n: ThisParameterType<typeof toHex>) {
  return toHex.apply(n);
}
```

OmitThisParameter<Type>

Released:

3.3

Removes the `this` parameter from `Type`. If `Type` has no explicitly declared `this` parameter, the result is simply `Type`. Otherwise, a new function type with no `this` parameter is created from `Type`. Generics are erased and only the last overload signature is propagated into the new function type.

Example

```
function toHex(this: Number) {
  return this.toString(16);
}

const fiveToHex: OmitThisParameter<typeof toHex> = toHex.bind(5);
console.log(fiveToHex());
```

ThisType<Type>

Released:

2.3

This utility does not return a transformed type. Instead, it serves as a marker for a contextual `this` type. Note that the `noImplicitThis` flag must be enabled to use this utility.

Example

```
// @noImplicitThis: false
type ObjectDescriptor<D, M> = {
  data?: D;
  methods?: M & ThisType<D & M>; // Type of 'this' in methods is D & M
};

function makeObject<D, M>(desc: ObjectDescriptor<D, M>): D & M {
  let data: object = desc.data || {};
  let methods: object = desc.methods || {};
  return { ...data, ...methods } as D & M;
}

let obj = makeObject({
  data: { x: 0, y: 0 },
  methods: {
    moveBy(dx: number, dy: number) {
      this.x += dx; // Strongly typed this
      this.y += dy; // Strongly typed this
    },
  },
};

obj.x = 10;
obj.y = 20;
obj.moveBy(5, 5);
```

In the example above, the `methods` object in the argument to `makeObject` has a contextual type that includes `ThisType<D & M>` and therefore the type of `this` in methods within the `methods` object is `{ x: number, y: number } & { moveBy(dx: number, dy: number): number }`. Notice how the type of the `methods` property simultaneously is an inference target and a source for the `this` type in methods.

The `ThisType<T>` marker interface is simply an empty interface declared in `lib.d.ts`. Beyond being recognized in the contextual type of an object literal, the interface acts like any empty interface.

Intrinsic String Manipulation Types

`Uppercase<StringType>`

`Lowercase<StringType>`

`Capitalize<StringType>`

`Uncapitalize<StringType>`

To help with string manipulation around template string literals, TypeScript includes a set of types which can be used in string manipulation within the type system. You can find those in the [Template Literal Types](#) documentation.

[Go to TOC](#)

`let` and `const` are two relatively new concepts for variable declarations in JavaScript. As we mentioned earlier, `let` is similar to `var` in some respects, but allows users to avoid some of the common "gotchas" that users run into in JavaScript.

`const` is an augmentation of `let` in that it prevents re-assignment to a variable.

With TypeScript being an extension of JavaScript, the language naturally supports `let` and `const`. Here we'll elaborate more on these new declarations and why they're preferable to `var`.

If you've used JavaScript offhandedly, the next section might be a good way to refresh your memory. If you're intimately familiar with all the quirks of `var` declarations in JavaScript, you might find it easier to skip ahead.

var declarations

Declaring a variable in JavaScript has always traditionally been done with the `var` keyword.

```
var a = 10;
```

As you might've figured out, we just declared a variable named `a` with the value `10`.

We can also declare a variable inside of a function:

```
function f() {
  var message = "Hello, world!";
  return message;
}
```

and we can also access those same variables within other functions:

```
function f() {
  var a = 10;
  return function g() {
    var b = a + 1;
    return b;
  };
}

var g = f();
g(); // returns '11'
```

In this above example, `g` captured the variable `a` declared in `f`. At any point that `g` gets called, the value of `a` will be tied to the value of `a` in `f`. Even if `g` is called once `f` is done running, it will be able to access and modify `a`.

```
function f() {
  var a = 1;

  a = 2;
  var b = g();
  a = 3;
```

```

    return b;
}

function g() {
    return a;
}
}

f(); // returns '2'

```

Scoping rules

`var` declarations have some odd scoping rules for those used to other languages. Take the following example:

```

function f(shouldInitialize: boolean) {
    if (shouldInitialize) {
        var x = 10;
    }

    return x;
}

f(true); // returns '10'
f(false); // returns 'undefined'

```

Some readers might do a double-take at this example. The variable `x` was declared *within the `if` block*, and yet we were able to access it from outside that block. That's because `var` declarations are accessible anywhere within their containing function, module, namespace, or global scope - all which we'll go over later on - regardless of the containing block. Some people call this `var`-scoping or `function-scoping`. Parameters are also function scoped.

These scoping rules can cause several types of mistakes. One problem they exacerbate is the fact that it is not an error to declare the same variable multiple times:

```

function sumMatrix(matrix: number[][]) {
    var sum = 0;
    for (var i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (var i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }

    return sum;
}

```

Maybe it was easy to spot out for some experienced JavaScript developers, but the inner `for`-loop will accidentally overwrite the variable `i` because `i` refers to the same function-scoped variable. As experienced developers know by now, similar sorts of bugs slip through code reviews and can be an endless source of frustration.

Variable capturing quirks

Take a quick second to guess what the output of the following snippet is:

```
for (var i = 0; i < 10; i++) {
  setTimeout(function () {
    console.log(i);
  }, 100 * i);
}
```

For those unfamiliar, `setTimeout` will try to execute a function after a certain number of milliseconds (though waiting for anything else to stop running).

Ready? Take a look:

```
10
10
10
10
10
10
10
10
10
10
```

Many JavaScript developers are intimately familiar with this behavior, but if you're surprised, you're certainly not alone. Most people expect the output to be

```
0
1
2
3
4
5
6
7
8
9
```

Remember what we mentioned earlier about variable capturing? Every function expression we pass to `setTimeout` actually refers to the same `i` from the same scope.

Let's take a minute to consider what that means. `setTimeout` will run a function after some number of milliseconds, *but only* after the `for` loop has stopped executing; By the time the `for` loop has stopped executing, the value of `i` is `10`. So each time the given function gets called, it will print out `10`!

A common work around is to use an IIFE - an Immediately Invoked Function Expression - to capture `i` at each iteration:

```
for (var i = 0; i < 10; i++) {
  // capture the current state of 'i'
  // by invoking a function with its current value
  (function (i) {
```

```
    setTimeout(function () {
      console.log(i);
    }, 100 * i);
})(i);
}
```

This odd-looking pattern is actually pretty common. The `i` in the parameter list actually shadows the `i` declared in the `for` loop, but since we named them the same, we didn't have to modify the loop body too much.

let declarations

By now you've figured out that `var` has some problems, which is precisely why `let` statements were introduced. Apart from the keyword used, `let` statements are written the same way `var` statements are.

```
let hello = "Hello!";
```

The key difference is not in the syntax, but in the semantics, which we'll now dive into.

Block-scoping

When a variable is declared using `let`, it uses what some call *lexical-scoping* or *block-scoping*. Unlike variables declared with `var` whose scopes leak out to their containing function, block-scoped variables are not visible outside of their nearest containing block or `for`-loop.

```
function f(input: boolean) {
  let a = 100;

  if (input) {
    // Still okay to reference 'a'
    let b = a + 1;
    return b;
  }

  // Error: 'b' doesn't exist here
  return b;
}
```

Here, we have two local variables `a` and `b`. `a`'s scope is limited to the body of `f` while `b`'s scope is limited to the containing `if` statement's block.

Variables declared in a `catch` clause also have similar scoping rules.

```
try {
  throw "oh no!";
} catch (e) {
  console.log("Oh well.");
}

// Error: 'e' doesn't exist here
console.log(e);
```

Another property of block-scoped variables is that they can't be read or written to before they're actually declared. While these variables are "present" throughout their scope, all points up until their declaration are part of their *temporal dead zone*. This is just a sophisticated way of saying you can't access them before the `let` statement, and luckily TypeScript will let you know that.

```
a++; // illegal to use 'a' before it's declared;
let a;
```

Something to note is that you can still *capture* a block-scoped variable before it's declared. The only catch is that it's illegal to call that function before the declaration. If targeting ES2015, a modern runtime will throw an error; however, right now TypeScript is permissive and won't report this as an error.

```
function foo() {
  // okay to capture 'a'
  return a;
}

// illegal call 'foo' before 'a' is declared
// runtimes should throw an error here
foo();

let a;
```

For more information on temporal dead zones, see relevant content on the [Mozilla Developer Network](#).

Re-declarations and Shadowing

With `var` declarations, we mentioned that it didn't matter how many times you declared your variables; you just got one.

```
function f(x) {
  var x;
  var x;

  if (true) {
    var x;
  }
}
```

In the above example, all declarations of `x` actually refer to the *same* `x`, and this is perfectly valid. This often ends up being a source of bugs. Thankfully, `let` declarations are not as forgiving.

```
let x = 10;
let x = 20; // error: can't re-declare 'x' in the same scope
```

The variables don't necessarily need to both be block-scoped for TypeScript to tell us that there's a problem.

```
function f(x) {
  let x = 100; // error: interferes with parameter declaration
}

function g() {
```

```
let x = 100;
var x = 100; // error: can't have both declarations of 'x'
```

That's not to say that a block-scoped variable can never be declared with a function-scoped variable. The block-scoped variable just needs to be declared within a distinctly different block.

```
function f(condition, x) {
  if (condition) {
    let x = 100;
    return x;
  }
  return x;
}

f(false, 0); // returns '0'
f(true, 0); // returns '100'
```

The act of introducing a new name in a more nested scope is called *shadowing*. It is a bit of a double-edged sword in that it can introduce certain bugs on its own in the event of accidental shadowing, while also preventing certain bugs. For instance, imagine we had written our earlier `sumMatrix` function using `let` variables.

```
function sumMatrix(matrix: number[][]) {
  let sum = 0;
  for (let i = 0; i < matrix.length; i++) {
    var currentRow = matrix[i];
    for (let i = 0; i < currentRow.length; i++) {
      sum += currentRow[i];
    }
  }
  return sum;
}
```

This version of the loop will actually perform the summation correctly because the inner loop's `i` shadows `i` from the outer loop.

Shadowing should *usually* be avoided in the interest of writing clearer code. While there are some scenarios where it may be fitting to take advantage of it, you should use your best judgement.

Block-scoped variable capturing

When we first touched on the idea of variable capturing with `var` declaration, we briefly went into how variables act once captured. To give a better intuition of this, each time a scope is run, it creates an "environment" of variables. That environment and its captured variables can exist even after everything within its scope has finished executing.

```
function theCityThatAlwaysSleeps() {
  let getCity;

  if (true) {
    let city = "Seattle";
```

```

    getCity = function () {
      return city;
    };
  }
  return getCity();
}

```

Because we've captured `city` from within its environment, we're still able to access it despite the fact that the `if` block finished executing.

Recall that with our earlier `setTimeout` example, we ended up needing to use an IIFE to capture the state of a variable for every iteration of the `for` loop. In effect, what we were doing was creating a new variable environment for our captured variables. That was a bit of a pain, but luckily, you'll never have to do that again in TypeScript.

`let` declarations have drastically different behavior when declared as part of a loop. Rather than just introducing a new environment to the loop itself, these declarations sort of create a new scope *per iteration*. Since this is what we were doing anyway with our IIFE, we can change our old `setTimeout` example to just use a `let` declaration.

```

for (let i = 0; i < 10; i++) {
  setTimeout(function () {
    console.log(i);
  }, 100 * i);
}

```

and as expected, this will print out

```

0
1
2
3
4
5
6
7
8
9

```

const declarations

`const` declarations are another way of declaring variables.

```
const numLivesForCat = 9;
```

They are like `let` declarations but, as their name implies, their value cannot be changed once they are bound. In other words, they have the same scoping rules as `let`, but you can't re-assign to them.

This should not be confused with the idea that the values they refer to are *immutable*.

```

const numLivesForCat = 9;
const kitty = {
  name: "Aurora",
  numLives: numLivesForCat,
};

// Error
kitty = {
  name: "Danielle",
  numLives: numLivesForCat,
};

// all "okay"
kitty.name = "Rory";
kitty.name = "Kitty";
kitty.name = "Cat";
kitty.numLives--;

```

Unless you take specific measures to avoid it, the internal state of a `const` variable is still modifiable. Fortunately, TypeScript allows you to specify that members of an object are `readonly`. The [chapter on Interfaces](#) has the details.

let vs. const

Given that we have two types of declarations with similar scoping semantics, it's natural to find ourselves asking which one to use. Like most broad questions, the answer is: it depends.

Applying the [principle of least privilege](#), all declarations other than those you plan to modify should use `const`. The rationale is that if a variable didn't need to get written to, others working on the same code-base shouldn't automatically be able to write to the object, and will need to consider whether they really need to reassign to the variable. Using `const` also makes code more predictable when reasoning about flow of data.

Use your best judgement, and if applicable, consult the matter with the rest of your team.

The majority of this handbook uses `let` declarations.

Destructuring

Another ECMAScript 2015 feature that TypeScript has is destructuring. For a complete reference, see [the article on the Mozilla Developer Network](#). In this section, we'll give a short overview.

Array destructuring

The simplest form of destructuring is array destructuring assignment:

```

let input = [1, 2];
let [first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2

```

This creates two new variables named `first` and `second`. This is equivalent to using indexing, but is much more convenient:

```
first = input[0];
second = input[1];
```

Destructuring works with already-declared variables as well:

```
// swap variables
[first, second] = [second, first];
```

And with parameters to a function:

```
function f([first, second]: [number, number]) {
  console.log(first);
  console.log(second);
}
f([1, 2]);
```

You can create a variable for the remaining items in a list using the syntax `...`:

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(first); // outputs 1
console.log(rest); // outputs [ 2, 3, 4 ]
```

Of course, since this is JavaScript, you can just ignore trailing elements you don't care about:

```
let [first] = [1, 2, 3, 4];
console.log(first); // outputs 1
```

Or other elements:

```
let [, second, , fourth] = [1, 2, 3, 4];
console.log(second); // outputs 2
console.log(fourth); // outputs 4
```

Tuple destructuring

Tuples may be destructured like arrays; the destructuring variables get the types of the corresponding tuple elements:

```
let tuple: [number, string, boolean] = [7, "hello", true];
let [a, b, c] = tuple; // a: number, b: string, c: boolean
```

It's an error to destructure a tuple beyond the range of its elements:

```
let [a, b, c, d] = tuple; // Error, no element at index 3
```

As with arrays, you can destructure the rest of the tuple with `...`, to get a shorter tuple:

```
let [a, ...bc] = tuple; // bc: [string, boolean]
let [a, b, c, ...d] = tuple; // d: [], the empty tuple
```

Or ignore trailing elements, or other elements:

```
let [a] = tuple; // a: number
let [, b] = tuple; // b: string
```

Object destructuring

You can also destructure objects:

```
let o = {
  a: "foo",
  b: 12,
  c: "bar",
};
let { a, b } = o;
```

This creates new variables `a` and `b` from `o.a` and `o.b`. Notice that you can skip `c` if you don't need it.

Like array destructuring, you can have assignment without declaration:

```
({ a, b } = { a: "baz", b: 101 });
```

Notice that we had to surround this statement with parentheses. JavaScript normally parses a `{` as the start of block.

You can create a variable for the remaining items in an object using the syntax `...`:

```
let { a, ...passthrough } = o;
let total = passthrough.b + passthrough.c.length;
```

Property renaming

You can also give different names to properties:

```
let { a: newName1, b: newName2 } = o;
```

Here the syntax starts to get confusing. You can read `a: newName1 as "a as newName1"`. The direction is left-to-right, as if you had written:

```
let newName1 = o.a;
let newName2 = o.b;
```

Confusingly, the colon here does *not* indicate the type. The type, if you specify it, still needs to be written after the entire destructuring:

```
let { a: newName1, b: newName2 }: { a: string; b: number } = o;
```

Default values

Default values let you specify a default value in case a property is undefined:

```
function keepWholeObject(wholeObject: { a: string; b?: number }) {
  let { a, b = 1001 } = wholeObject;
}
```

In this example the `b?` indicates that `b` is optional, so it may be `undefined`. `keepWholeObject` now has a variable for `wholeObject` as well as the properties `a` and `b`, even if `b` is undefined.

Function declarations

Destructuring also works in function declarations. For simple cases this is straightforward:

```
type C = { a: string; b?: number };
function f({ a, b }: C): void {
  // ...
}
```

But specifying defaults is more common for parameters, and getting defaults right with destructuring can be tricky. First of all, you need to remember to put the pattern before the default value.

```
function f({ a = "", b = 0 } = {}): void {
  // ...
}
f();
```

The snippet above is an example of type inference, explained earlier in the handbook.

Then, you need to remember to give a default for optional properties on the destructured property instead of the main initializer. Remember that `C` was defined with `b` optional:

```
function f({ a, b = 0 } = { a: "" }): void {
  // ...
}
f({ a: "yes" }); // ok, default b = 0
f(); // ok, default to { a: "" }, which then defaults b = 0
f({}); // error, 'a' is required if you supply an argument
```

Use destructuring with care. As the previous example demonstrates, anything but the simplest destructuring expression is confusing. This is especially true with deeply nested destructuring, which gets *really* hard to understand even without piling on renaming, default values, and type annotations. Try to keep destructuring expressions small and simple. You can always write the assignments that destructuring would generate yourself.

Spread

The spread operator is the opposite of destructuring. It allows you to spread an array into another array, or an object into another object. For example:

```
let first = [1, 2];
let second = [3, 4];
let bothPlus = [0, ...first, ...second, 5];
```

This gives `bothPlus` the value `[0, 1, 2, 3, 4, 5]`. Spreading creates a shallow copy of `first` and `second`. They are not changed by the spread.

You can also spread objects:

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };
let search = { ...defaults, food: "rich" };
```

Now `search` is `{ food: "rich", price: "$$", ambiance: "noisy" }`. Object spreading is more complex than array spreading. Like array spreading, it proceeds from left-to-right, but the result is still an object. This means that properties that come later in the spread object overwrite properties that come earlier. So if we modify the previous example to spread at the end:

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };
let search = { food: "rich", ...defaults };
```

Then the `food` property in `defaults` overwrites `food: "rich"`, which is not what we want in this case.

Object spread also has a couple of other surprising limits. First, it only includes an objects' [own, enumerable properties](#). Basically, that means you lose methods when you spread instances of an object:

```
class C {
  p = 12;
  m() {}
}
let c = new C();
let clone = { ...c };
clone.p; // ok
clone.m(); // error!
```

Second, the TypeScript compiler doesn't allow spreads of type parameters from generic functions. That feature is expected in future versions of the language.

[Go to TOC](#)

Performance Improvements

The 1.1 compiler is typically around 4x faster than any previous release. See [this blog post](#) for some impressive charts.

Better Module Visibility Rules

TypeScript now only strictly enforces the visibility of types in modules if the `declaration` flag is provided. This is very useful for Angular scenarios, for example:

```
module MyControllers {
    interface ZooScope extends ng.IScope {
        animals: Animal[];
    }
    export class ZooController {
        // Used to be an error (cannot expose ZooScope), but now is only
        // an error when trying to generate .d.ts files
        constructor(public $scope: ZooScope) {}
        /* more code */
    }
}
```

[Go to TOC](#)

Protected

The new `protected` modifier in classes works like it does in familiar languages like C++, C#, and Java. A `protected` member of a class is visible only inside subclasses of the class in which it is declared:

```
class Thing {
  protected doSomething() {
    /* ... */
  }
}

class MyThing extends Thing {
  public myMethod() {
    // OK, can access protected member from subclass
    this.doSomething();
  }
}
var t = new MyThing();
t.doSomething(); // Error, cannot call protected member from outside class
```

Tuple types

Tuple types express an array where the type of certain elements is known, but need not be the same. For example, you may want to represent an array with a `string` at position 0 and a `number` at position 1:

```
// Declare a tuple type
var x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

When accessing an element with a known index, the correct type is retrieved:

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'
```

Note that in TypeScript 1.4, when accessing an element outside the set of known indices, a union type is used instead:

```
x[3] = "world"; // OK
console.log(x[5].toString()); // OK, 'string' and 'number' both have toString
x[6] = true; // Error, boolean isn't number or string
```

[Go to TOC](#)

Union types

Overview

Union types are a powerful way to express a value that can be one of several types. For example, you might have an API for running a program that takes a commandline as either a `string`, a `string[]` or a function that returns a `string`. You can now write:

```
interface RunOptions {
  program: string;
  commandline: string[] | string | ((() => string));
}
```

Assignment to union types works very intuitively -- anything you could assign to one of the union type's members is assignable to the union:

```
var opts: RunOptions = /* ... */;
opts.commandline = '-hello world'; // OK
opts.commandline = ['-hello', 'world']; // OK
opts.commandline = [42]; // Error, number is not string or string[]
```

When reading from a union type, you can see any properties that are shared by them:

```
if (opts.commandline.length === 0) {
  // OK, string and string[] both have 'length' property
  console.log("it's empty");
}
```

Using Type Guards, you can easily work with a variable of a union type:

```
function formatCommandline(c: string | string[]) {
  if (typeof c === "string") {
    return c.trim();
  } else {
    return c.join(" ");
  }
}
```

Stricter Generics

With union types able to represent a wide range of type scenarios, we've decided to improve the strictness of certain generic calls. Previously, code like this would (surprisingly) compile without error:

```
function equal<T>(lhs: T, rhs: T): boolean {
  return lhs === rhs;
}

// Previously: No error
// New behavior: Error, no best common type between 'string' and 'number'
var e = equal(42, "hello");
```

With union types, you can now specify the desired behavior at both the function declaration site and the call site:

```
// 'choose' function where types must match
function choose1<T>(a: T, b: T): T {
    return Math.random() > 0.5 ? a : b;
}
var a = choose1("hello", 42); // Error
var b = choose1<string | number>("hello", 42); // OK

// 'choose' function where types need not match
function choose2<T, U>(a: T, b: U): T | U {
    return Math.random() > 0.5 ? a : b;
}
var c = choose2("bar", "foo"); // OK, c: string
var d = choose2("hello", 42); // OK, d: string|number
```

Better Type Inference

Union types also allow for better type inference in arrays and other places where you might have multiple kinds of values in a collection:

```
var x = [1, "hello"]; // x: Array<string|number>
x[0] = "world"; // OK
x[0] = false; // Error, boolean is not string or number
```

let declarations

In JavaScript, `var` declarations are "hoisted" to the top of their enclosing scope. This can result in confusing bugs:

```
console.log(x); // meant to write 'y' here
/* later in the same block */
var x = "hello";
```

The new ES6 keyword `let`, now supported in TypeScript, declares a variable with more intuitive "block" semantics. A `let` variable can only be referred to after its declaration, and is scoped to the syntactic block where it is defined:

```
if (foo) {
    console.log(x); // Error, cannot refer to x before its declaration
    let x = "hello";
} else {
    console.log(x); // Error, x is not declared in this block
}
```

`let` is only available when targeting ECMAScript 6 (`--target ES6`).

const declarations

The other new ES6 declaration type supported in TypeScript is `const`. A `const` variable may not be assigned to, and must be initialized where it is declared. This is useful for declarations where you don't want to change the value after its initialization:

```
const halfPi = Math.PI / 2;
halfPi = 2; // Error, can't assign to a `const`
```

`const` is only available when targeting ECMAScript 6 (`--target ES6`).

Template strings

TypeScript now supports ES6 template strings. These are an easy way to embed arbitrary expressions in strings:

```
var name = "TypeScript";
var greeting = `Hello, ${name}! Your name has ${name.length} characters`;
```

When compiling to pre-ES6 targets, the string is decomposed:

```
var name = "TypeScript!";
var greeting =
  "Hello, " + name + "! Your name has " + name.length + " characters";
```

Type Guards

A common pattern in JavaScript is to use `typeof` or `instanceof` to examine the type of an expression at runtime. TypeScript now understands these conditions and will change type inference accordingly when used in an `if` block.

Using `typeof` to test a variable:

```
var x: any = /* ... */;
if(typeof x === 'string') {
  console.log(x.substr(1)); // Error, 'substr' does not exist on 'string'
}
// x is still any here
x.unknown(); // OK
```

Using `typeof` with union types and `else`:

```
var x: string | HTMLElement = /* ... */;
if(typeof x === 'string') {
  // x is string here, as shown above
}
else {
  // x is HTMLElement here
  console.log(x.innerHTML);
}
```

Using `instanceof` with classes and union types:

```
class Dog { woof() {} }
class Cat { meow() {} }
var pet: Dog|Cat = /* ... */;
if (pet instanceof Dog) {
    pet.woof(); // OK
}
else {
    pet.woof(); // Error
}
```

Type Aliases

You can now define an *alias* for a type using the `type` keyword:

```
type PrimitiveArray = Array<string | number | boolean>;
type MyNumber = number;
type NgScope = ng.IScope;
type Callback = () => void;
```

Type aliases are exactly the same as their original types; they are simply alternative names.

`const enum` (completely inlined enums)

Enums are very useful, but some programs don't actually need the generated code and would benefit from simply inlining all instances of enum members with their numeric equivalents. The new `const enum` declaration works just like a regular `enum` for type safety, but erases completely at compile time.

```
const enum Suit {
    Clubs,
    Diamonds,
    Hearts,
    Spades
}
var d = Suit.Diamonds;
```

Compiles to exactly:

```
var d = 1;
```

TypeScript will also now compute enum values when possible:

```
enum MyFlags {
    None = 0,
    Neat = 1,
    Cool = 2,
    Awesome = 4,
    Best = Neat | Cool | Awesome
}
var b = MyFlags.Best; // emits var b = 7;
```

-noEmitOnError commandline option

The default behavior for the TypeScript compiler is to still emit .js files if there were type errors (for example, an attempt to assign a `string` to a `number`). This can be undesirable on build servers or other scenarios where only output from a "clean" build is desired. The new flag `noEmitOnError` prevents the compiler from emitting .js code if there were any errors.

This is now the default for MSBuild projects; this allows MSBuild incremental build to work as expected, as outputs are only generated on clean builds.

AMD Module names

By default AMD modules are generated anonymous. This can lead to problems when other tools are used to process the resulting modules like a bundlers (e.g. `r.js`).

The new `amd-module name` tag allows passing an optional module name to the compiler:

```
//// [amdModule.ts]
///<amd-module name='NamedModule'>
export class C {}
```

Will result in assigning the name `NamedModule` to the module as part of calling the AMD `define`:

```
//// [amdModule.js]
define("NamedModule", ["require", "exports"], function(require, exports) {
  var C = (function() {
    function C() {}
    return C;
  })();
  exports.C = C;
});
```

[Go to TOC](#)

ES6 Modules

TypeScript 1.5 supports ECMAScript 6 (ES6) modules. ES6 modules are effectively TypeScript external modules with a new syntax: ES6 modules are separately loaded source files that possibly import other modules and provide a number of externally accessible exports. ES6 modules feature several new export and import declarations. It is recommended that TypeScript libraries and applications be updated to use the new syntax, but this is not a requirement. The new ES6 module syntax coexists with TypeScript's original internal and external module constructs and the constructs can be mixed and matched at will.

Export Declarations

In addition to the existing TypeScript support for decorating declarations with `export`, module members can also be exported using separate export declarations, optionally specifying different names for exports using `as` clauses.

```
interface Stream { ... }
function writeToStream(stream: Stream, data: string) { ... }
export { Stream, writeToStream as write }; // writeToStream exported as write
```

Import declarations, as well, can optionally use `as` clauses to specify different local names for the imports.

For example:

```
import { read, write, standardOutput as stdout } from "./inout";
var s = read(stdout);
write(stdout, s);
```

As an alternative to individual imports, a namespace import can be used to import an entire module:

```
import * as io from "./inout";
var s = io.read(io.standardOutput);
io.write(io.standardOutput, s);
```

Re-exporting

Using `from` clause a module can copy the exports of a given module to the current module without introducing local names.

```
export { read, write, standardOutput as stdout } from "./inout";
```

`export *` can be used to re-export all exports of another module. This is useful for creating modules that aggregate the exports of several other modules.

```
export function transform(s: string): string { ... }
export * from "./mod1";
export * from "./mod2";
```

Default Export

An export default declaration specifies an expression that becomes the default export of a module:

```
export default class Greeter {
  sayHello() {
    console.log("Greetings!");
  }
}
```

Which in turn can be imported using default imports:

```
import Greeter from "./greeter";
var g = new Greeter();
g.sayHello();
```

Bare Import

A "bare import" can be used to import a module only for its side-effects.

```
import "./polyfills";
```

For more information about module, please see the [ES6 module support spec](#).

Destructuring in declarations and assignments

TypeScript 1.5 adds support to ES6 destructuring declarations and assignments.

Declarations

A destructuring declaration introduces one or more named variables and initializes them with values extracted from properties of an object or elements of an array.

For example, the following sample declares variables `x`, `y`, and `z`, and initializes them to `getSomeObject().x`, `getSomeObject().y` and `getSomeObject().z` respectively:

```
var { x, y, z } = getSomeObject();
```

Destructuring declarations also works for extracting values from arrays:

```
var [x, y, z = 10] = getSomeArray();
```

Similarly, destructuring can be used in function parameter declarations:

```
function drawText({ text = "", location: [x, y] = [0, 0], bold = false }) {
  // Draw text
}

// Call drawText with an object literal
var item = { text: "someText", location: [1, 2, 3], style: "italics" };
drawText(item);
```

Assignments

Destructuring patterns can also be used in regular assignment expressions. For instance, swapping two variables can be written as a single destructuring assignment:

```
var x = 1;
var y = 2;
[x, y] = [y, x];
```

namespace keyword

TypeScript used the `module` keyword to define both "internal modules" and "external modules"; this has been a bit of confusion for developers new to TypeScript. "Internal modules" are closer to what most people would call a namespace; likewise, "external modules" in JS speak really just are modules now.

Note: Previous syntax defining internal modules are still supported.

Before:

```
module Math {
    export function add(x, y) { ... }
```

After:

```
namespace Math {
    export function add(x, y) { ... }
```

let and const support

ES6 `let` and `const` declarations are now supported when targeting ES3 and ES5.

Const

```
const MAX = 100;
++MAX; // Error: The operand of an increment or decrement
//          operator cannot be a constant.
```

Block scoped

```
if (true) {
    let a = 4;
    // use a
} else {
    let a = "string";
    // use a
}

alert(a); // Error: a is not defined in this scope.
```

for..of support

TypeScript 1.5 adds support to ES6 for..of loops on arrays for ES3/ES5 as well as full support for Iterator interfaces when targeting ES6.

Example

The TypeScript compiler will transpile for..of arrays to idiomatic ES3/ES5 JavaScript when targeting those versions:

```
for (var v of expr) {
```

will be emitted as:

```
for (var _i = 0, _a = expr; _i < _a.length; _i++) {
    var v = _a[_i];
}
```

Decorators

TypeScript decorators are based on the [ES7 decorator proposal](#).

A decorator is:

- an expression
- that evaluates to a function
- that takes the target, name, and property descriptor as arguments
- and optionally returns a property descriptor to install on the target object

For more information, please see the [Decorators](#) proposal.

Example

Decorators `readonly` and `enumerable(false)` will be applied to the property `method` before it is installed on class `C`. This allows the decorator to change the implementation, and in this case, augment the descriptor to be writable: false and enumerable: false.

```
class C {
    @readonly
    @enumerable(false)
    method() { ... }

    function readonly(target, key, descriptor) {
        descriptor.writable = false;
    }
}
```

```

function enumerable(value) {
  return function (target, key, descriptor) {
    descriptor.enumerable = value;
  };
}

```

Computed properties

Initializing an object with dynamic properties can be a bit of a burden. Take the following example:

```

type NeighborMap = { [name: string]: Node };
type Node = { name: string; neighbors: NeighborMap };

function makeNode(name: string, initialNeighbor: Node): Node {
  var neighbors: NeighborMap = {};
  neighbors[initialNeighbor.name] = initialNeighbor;
  return { name: name, neighbors: neighbors };
}

```

Here we need to create a variable to hold on to the neighbor-map so that we can initialize it. With TypeScript 1.5, we can let the compiler do the heavy lifting:

```

function makeNode(name: string, initialNeighbor: Node): Node {
  return {
    name: name,
    neighbors: {
      [initialNeighbor.name]: initialNeighbor,
    },
  };
}

```

Support for UMD and System module output

In addition to `AMD` and `CommonJS` module loaders, TypeScript now supports emitting modules `UMD` ([Universal Module Definition](#)) and `System` module formats.

Usage:

```
tsc --module umd
```

and

```
tsc --module system
```

Unicode codepoint escapes in strings

ES6 introduces escapes that allow users to represent a Unicode codepoint using just a single escape.

As an example, consider the need to escape a string that contains the character '𠮷'. In UTF-16/UCS2, '𠮷' is represented as a surrogate pair, meaning that it's encoded using a pair of 16-bit code units of values, specifically `0xD842` and `0xDFB7`. Previously this meant that you'd have to escape the codepoint as `"\uD842\uDFB7"`. This has the major downside that it's difficult to discern two independent characters from a surrogate pair.

With ES6's codepoint escapes, you can cleanly represent that exact character in strings and template strings with a single escape: `"\u{20bb7}"`. TypeScript will emit the string in ES3/ES5 as `"\uD842\uDFB7"`.

Tagged template strings in ES3/ES5

In TypeScript 1.4, we added support for template strings for all targets, and tagged templates for just ES6. Thanks to some considerable work done by [@ivogabe](#), we bridged the gap for tagged templates in ES3 and ES5.

When targeting ES3/ES5, the following code

```
function oddRawStrings(strs: TemplateStringsArray, n1, n2) {
    return strs.raw.filter((raw, index) => index % 2 === 1);
}

oddRawStrings`Hello \n${123} \t ${456}\n world`;
```

will be emitted as

```
function oddRawStrings(strs, n1, n2) {
    return strs.raw.filter(function (raw, index) {
        return index % 2 === 1;
    });
}
_a = ["Hello \n", " \t ", "\n world"],
_a.raw = ["Hello \\n", " \\t ", "\\n world"],
oddRawStrings(_a, 123, 456);
var _a;
```

AMD-dependency optional names

`/// <amd-dependency path="x" />` informs the compiler about a non-TS module dependency that needs to be injected in the resulting module's require call; however, there was no way to consume this module in the TS code.

The new `amd-dependency name` property allows passing an optional name for an amd-dependency:

```
/// <amd-dependency path="legacy/moduleA" name="moduleA"/>
declare var moduleA: MyType;
moduleA.callStuff();
```

Generated JS code:

```
define(["require", "exports", "legacy/moduleA"], function (
    require,
    exports,
```

```

    moduleA
  ) {
    moduleA.callStuff();
  });
}

```

Project support through `tsconfig.json`

Adding 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. 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.

Example

```

{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "sourceMap": true
  }
}

```

See the [tsconfig.json wiki page](#) for more details.

`--rootDir` command line option

Option `outDir` duplicates the input hierarchy in the output. The compiler computes the root of the input files as the longest common path of all input files; and then uses that to replicate all its substructure in the output.

Sometimes this is not desirable, for instance inputs `FolderA\FolderB\1.ts` and `FolderA\FolderB\2.ts` would result in output structure mirroring `FolderA\FolderB\`. Now if a new file `FolderA\3.ts` is added to the input, the output structure will pop out to mirror `FolderA\`.

`rootDir` specifies the input directory to be mirrored in output instead of computing it.

`--noEmitHelpers` command line option

The TypeScript compiler emits a few helpers like `_extends` when needed. The helpers are emitted in every file they are referenced in. If you want to consolidate all helpers in one place, or override the default behavior, use `noEmitHelpers` to instructs the compiler not to emit them.

--newLine command line option

By default the output new line character is `\r\n` on Windows based systems and `\n` on *nix based systems. `newLine` command line flag allows overriding this behavior and specifying the new line character to be used in generated output files.

--inlineSourceMap and inlineSources command line options

`inlineSourceMap` causes source map files to be written inline in the generated `.js` files instead of in a independent `.js.map` file. `inlineSources` allows for additionally inlining the source `.ts` file into the `.js` file.

[Go to TOC](#)

JSX support

JSX is an embeddable XML-like syntax. It is meant to be transformed into valid JavaScript, but the semantics of that transformation are implementation-specific. JSX came to popularity with the React library but has since seen other applications. TypeScript 1.6 supports embedding, type checking, and optionally compiling JSX directly into JavaScript.

New `.tsx` file extension and `as` operator

TypeScript 1.6 introduces a new `.tsx` file extension. This extension does two things: it enables JSX inside of TypeScript files, and it makes the new `as` operator the default way to cast (removing any ambiguity between JSX expressions and the TypeScript prefix cast operator). For example:

```
var x = <any>foo;
// is equivalent to:
var x = foo as any;
```

Using React

To use JSX-support with React you should use the [React typings](#). These typings define the `JSX` namespace so that TypeScript can correctly check JSX expressions for React. For example:

```
/// <reference path="react.d.ts" />

interface Props {
    name: string;
}

class MyComponent extends React.Component<Props, {}> {
    render() {
        return <span>{this.props.name}</span>;
    }
}

<MyComponent name="bar" />; // OK
<MyComponent name={0} />; // error, `name` is not a number
```

Using other JSX frameworks

JSX element names and properties are validated against the `JSX` namespace. Please see the [\[\[JSX\]\]](#) wiki page for defining the `JSX` namespace for your framework.

Output generation

TypeScript ships with two JSX modes: `preserve` and `react`.

- The `preserve` mode will keep JSX expressions as part of the output to be further consumed by another transform step. *Additionally the output will have a `.jsx` file extension.*
- The `react` mode will emit `React.createElement`, does not need to go through a JSX transformation before use, and the output will have a `.js` file extension.

See the [[JSX]] wiki page for more information on using JSX in TypeScript.

Intersection types

TypeScript 1.6 introduces intersection types, the logical complement of union types. A union type `A | B` represents an entity that is either of type `A` or type `B`, whereas an intersection type `A & B` represents an entity that is both of type `A` and type `B`.

Example

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U>{};
    for (let id in first) {
        result[id] = first[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            result[id] = second[id];
        }
    }
    return result;
}

var x = extend({ a: "hello" }, { b: 42 });
var s = x.a;
var n = x.b;
```

```
type LinkedList<T> = T & { next: LinkedList<T> };

interface Person {
    name: string;
}

var people: LinkedList<Person>;
var s = people.name;
var s = people.next.name;
var s = people.next.next.name;
var s = people.next.next.next.name;
```

```
interface A {
    a: string;
}
interface B {
    b: string;
}
interface C {
    c: string;
}

var abc: A & B & C;
abc.a = "hello";
abc.b = "hello";
abc.c = "hello";
```

See [issue #1256](#) for more information.

Local type declarations

Local class, interface, enum, and type alias declarations can now appear inside function declarations. Local types are block scoped, similar to variables declared with `let` and `const`. For example:

```
function f() {
  if (true) {
    interface T {
      x: number;
    }
    let v: T;
    v.x = 5;
  } else {
    interface T {
      x: string;
    }
    let v: T;
    v.x = "hello";
  }
}
```

The inferred return type of a function may be a type declared locally within the function. It is not possible for callers of the function to reference such a local type, but it can of course be matched structurally. For example:

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

function getPointFactory(x: number, y: number) {
  class P {
    x = x;
    y = y;
  }
  return P;
}

var PointZero = getPointFactory(0, 0);
var PointOne = getPointFactory(1, 1);
var p1 = new PointZero();
var p2 = new PointZero();
var p3 = new PointOne();
```

Local types may reference enclosing type parameters and local class and interfaces may themselves be generic. For example:

```
function f3() {
  function f<X, Y>(x: X, y: Y) {
    class C {
      public x = x;
      public y = y;
    }
    return C;
  }
  let C = f<10, "hello">;
  let v = new C();
}
```

```
let x = v.x; // number
let y = v.y; // string
}
```

Class expressions

TypeScript 1.6 adds support for ES6 class expressions. In a class expression, the class name is optional and, if specified, is only in scope in the class expression itself. This is similar to the optional name of a function expression. It is not possible to refer to the class instance type of a class expression outside the class expression, but the type can of course be matched structurally. For example:

```
let Point = class {
  constructor(public x: number, public y: number) {}
  public length() {
    return Math.sqrt(this.x * this.x + this.y * this.y);
  }
};
var p = new Point(3, 4); // p has anonymous class type
console.log(p.length());
```

Extending expressions

TypeScript 1.6 adds support for classes extending arbitrary expression that computes a constructor function. This means that built-in types can now be extended in class declarations.

The `extends` clause of a class previously required a type reference to be specified. It now accepts an expression optionally followed by a type argument list. The type of the expression must be a constructor function type with at least one construct signature that has the same number of type parameters as the number of type arguments specified in the `extends` clause. The return type of the matching construct signature(s) is the base type from which the class instance type inherits. Effectively, this allows both real classes and "class-like" expressions to be specified in the `extends` clause.

Some examples:

```
// Extend built-in types
class MyArray extends Array<number> {}
class MyError extends Error {}

// Extend computed base class
class ThingA {
  getGreeting() {
    return "Hello from A";
  }
}

class ThingB {
  getGreeting() {
    return "Hello from B";
  }
}

interface Greeter {
```

```

    getGreeting(): string;
}

interface GreeterConstructor {
    new (): Greeter;
}

function getGreeterBase(): GreeterConstructor {
    return Math.random() >= 0.5 ? ThingA : ThingB;
}

class Test extends getGreeterBase() {
    sayHello() {
        console.log(this.getGreeting());
    }
}

```

abstract classes and methods

TypeScript 1.6 adds support for `abstract` keyword for classes and their methods. An abstract class is allowed to have methods with no implementation, and cannot be constructed.

Examples

```

abstract class Base {
    abstract getThing(): string;
    getOtherThing() {
        return "hello";
    }
}

let x = new Base(); // Error, 'Base' is abstract

// Error, must either be 'abstract' or implement concrete 'getThing'
class Derived1 extends Base {}

class Derived2 extends Base {
    getThing() {
        return "hello";
    }
    foo() {
        super.getThing(); // Error: cannot invoke abstract members through 'super'
    }
}

var x = new Derived2(); // OK
var y: Base = new Derived2(); // Also OK
y.getThing(); // OK
y.getOtherThing(); // OK

```

Generic type aliases

With TypeScript 1.6, type aliases can be generic. For example:

```

type Lazy<T> = T | ((() => T));
var s: Lazy<string>;
s = "eager";

```

```
s = () => "lazy";

interface Tuple<A, B> {
  a: A;
  b: B;
}

type Pair<T> = Tuple<T, T>;
```

Stricter object literal assignment checks

TypeScript 1.6 enforces stricter object literal assignment checks for the purpose of catching excess or misspelled properties. Specifically, when a fresh object literal is assigned to a variable or passed as an argument for a non-empty target type, it is an error for the object literal to specify properties that don't exist in the target type.

Examples

```
var x: { foo: number };
x = { foo: 1, baz: 2 }; // Error, excess property `baz`

var y: { foo: number; bar?: number };
y = { foo: 1, baz: 2 }; // Error, excess or misspelled property `baz`
```

A type can include an index signature to explicitly indicate that excess properties are permitted:

```
var x: { foo: number; [x: string]: any };
x = { foo: 1, baz: 2 }; // Ok, `baz` matched by index signature
```

ES6 generators

TypeScript 1.6 adds support for generators when targeting ES6.

A generator function can have a return type annotation, just like a function. The annotation represents the type of the generator returned by the function. Here is an example:

```
function* g(): Iterable<string> {
  for (var i = 0; i < 100; i++) {
    yield ""; // string is assignable to string
  }
  yield* otherStringGenerator(); // otherStringGenerator must be iterable and
  // element type assignable to string
}
```

A generator function with no type annotation can have the type annotation inferred. So in the following case, the type will be inferred from the yield statements:

```
function* g() {
  for (var i = 0; i < 100; i++) {
    yield ""; // infer string
  }
  yield* otherStringGenerator(); // infer element type of otherStringGenerator
}
```

Experimental support for `async` functions

TypeScript 1.6 introduces experimental support of `async` functions when targeting ES6. Async functions are expected to invoke an asynchronous operation and await its result without blocking normal execution of the program. This is accomplished through the use of an ES6-compatible `Promise` implementation, and transposition of the function body into a compatible form to resume execution when the awaited asynchronous operation completes.

An *async function* is a function or method that has been prefixed with the `async` modifier. This modifier informs the compiler that function body transposition is required, and that the keyword `await` should be treated as a unary expression instead of an identifier. An *Async Function* must provide a return type annotation that points to a compatible `Promise` type. Return type inference can only be used if there is a globally defined, compatible `Promise` type.

Example

```
var p: Promise<number> = /* ... */;
async function fn(): Promise<number> {
    var i = await p; // suspend execution until 'p' is settled. 'i' has type
    "number"
    return 1 + i;
}

var a = async (): Promise<number> => 1 + await p; // suspends execution.
var a = async () => 1 + await p; // suspends execution. return type is inferred as
"Promise<number>" when compiling with --target ES6
var fe = async function(): Promise<number> {
    var i = await p; // suspend execution until 'p' is settled. 'i' has type
    "number"
    return 1 + i;
}

class C {
    async m(): Promise<number> {
        var i = await p; // suspend execution until 'p' is settled. 'i' has type
        "number"
        return 1 + i;
    }

    async get p(): Promise<number> {
        var i = await p; // suspend execution until 'p' is settled. 'i' has type
        "number"
        return 1 + i;
    }
}
```

Nightly builds

While not strictly a language change, nightly builds are now available by installing with the following command:

```
npm install -g typescript@next
```

Adjustments in module resolution logic

Starting from release 1.6 TypeScript compiler will use different set of rules to resolve module names when targeting 'commonjs'. These [rules](#) attempted to model module lookup procedure used by Node. This effectively mean that node modules can include information about its typings and TypeScript compiler will be able to find it. User however can override module resolution rules picked by the compiler by using `moduleResolution` command line option. Possible values are:

- 'classic' - module resolution rules used by pre 1.6 TypeScript compiler
- 'node' - node-like module resolution

Merging ambient class and interface declaration

The instance side of an ambient class declaration can be extended using an interface declaration. The class constructor object is unmodified. For example:

```
declare class Foo {
  public x: number;
}

interface Foo {
  y: string;
}

function bar(foo: Foo) {
  foo.x = 1; // OK, declared in the class Foo
  foo.y = "1"; // OK, declared in the interface Foo
}
```

User-defined type guard functions

TypeScript 1.6 adds a new way to narrow a variable type inside an `if` block, in addition to `typeof` and `instanceof`. A user-defined type guard function is one with a return type annotation of the form `x is T`, where `x` is a declared parameter in the signature, and `T` is any type. When a user-defined type guard function is invoked on a variable in an `if` block, the type of the variable will be narrowed to `T`.

Examples

```
function isCat(a: any): a is Cat {
  return a.name === "kitty";
}

var x: Cat | Dog;
if (isCat(x)) {
  x.meow(); // OK, x is Cat in this block
}
```

exclude property support in tsconfig.json

A tsconfig.json file that doesn't specify a files property (and therefore implicitly references all *.ts files in all subdirectories) can now contain an exclude property that specifies a list of files and/or directories to exclude from the compilation. The exclude property must be an array of strings that each specify a file or folder name relative to the location of the tsconfig.json file. For example:

```
{  
  "compilerOptions": {  
    "out": "test.js"  
  },  
  "exclude": ["node_modules", "test.ts", "utils/t2.ts"]  
}
```

The `exclude` list does not support wildcards. It must simply be a list of files and/or directories.

--init command line option

Run `tsc --init` in a directory to create an initial `tsconfig.json` in this directory with preset defaults. Optionally pass command line arguments along with `--init` to be stored in your initial `tsconfig.json` on creation.

[Go to TOC](#)

async / await support in ES6 targets (Node v4+)

TypeScript now supports asynchronous functions for engines that have native support for ES6 generators, e.g. Node v4 and above. Asynchronous functions are prefixed with the `async` keyword; `await` suspends the execution until an asynchronous function return promise is fulfilled and unwraps the value from the `Promise` returned.

Example

In the following example, each input element will be printed out one at a time with a 400ms delay:

```
"use strict";

// printDelayed is a 'Promise<void>'
async function printDelayed(elements: string[]) {
  for (const element of elements) {
    await delay(400);
    console.log(element);
  }
}

async function delay(milliseconds: number) {
  return new Promise<void>((resolve) => {
    setTimeout(resolve, milliseconds);
  });
}

printDelayed(["Hello", "beautiful", "asynchronous", "world"]).then(() => {
  console.log();
  console.log("Printed every element!");
});
```

For more information see [async function reference](#) reference.

Support for --target ES6 with --module

TypeScript 1.7 adds `ES6` to the list of options available for the `module` option and allows you to specify the module output when targeting `ES6`. This provides more flexibility to target exactly the features you want in specific runtimes.

Example

```
{
  "compilerOptions": {
    "module": "amd",
    "target": "es6"
  }
}
```

this -typing

It is a common pattern to return the current object (i.e. `this`) from a method to create fluent-style APIs. For instance, consider the following `BasicCalculator` module:

```
export default class BasicCalculator {
  public constructor(protected value: number = 0) {}

  public currentValue(): number {
    return this.value;
  }

  public add(operand: number) {
    this.value += operand;
    return this;
  }

  public subtract(operand: number) {
    this.value -= operand;
    return this;
  }

  public multiply(operand: number) {
    this.value *= operand;
    return this;
  }

  public divide(operand: number) {
    this.value /= operand;
    return this;
  }
}
```

A user could express `2 * 5 + 1` as

```
import calc from "./BasicCalculator";
let v = new calc(2).multiply(5).add(1).currentValue();
```

This often opens up very elegant ways of writing code; however, there was a problem for classes that wanted to extend from `BasicCalculator`. Imagine a user wanted to start writing a `ScientificCalculator`:

```
import BasicCalculator from "./BasicCalculator";

export default class ScientificCalculator extends BasicCalculator {
  public constructor(value = 0) {
    super(value);
  }

  public square() {
    this.value = this.value ** 2;
    return this;
  }

  public sin() {
    this.value = Math.sin(this.value);
  }
}
```

```
    return this;
}
```

Because TypeScript used to infer the type `BasicCalculator` for each method in `BasicCalculator` that returned `this`, the type system would forget that it had `ScientificCalculator` whenever using a `BasicCalculator` method.

For instance:

```
import calc from "./ScientificCalculator";

let v = new calc(0.5)
  .square()
  .divide(2)
  .sin() // Error: 'BasicCalculator' has no 'sin' method.
  .currentValue();
```

This is no longer the case - TypeScript now infers `this` to have a special type called `this` whenever inside an instance method of a class. The `this` type is written as so, and basically means "the type of the left side of the dot in a method call".

The `this` type is also useful with intersection types in describing libraries (e.g. Ember.js) that use mixin-style patterns to describe inheritance:

```
interface MyType {
  extend<T>(other: T): this & T;
}
```

ES7 exponentiation operator

TypeScript 1.7 supports upcoming [ES7/ES2016 exponentiation operators](#): `**` and `$\mathbf{**=}$` . The operators will be transformed in the output to ES3/ES5 using `Math.pow`.

Example

```
var x = 2 ** 3;
var y = 10;
y **= 2;
var z = -(4 ** 3);
```

Will generate the following JavaScript output:

```
var x = Math.pow(2, 3);
var y = 10;
y = Math.pow(y, 2);
var z = -Math.pow(4, 3);
```

Improved checking for destructuring object literal

TypeScript 1.7 makes checking of destructuring patterns with an object literal or array literal initializers less rigid and more intuitive.

When an object literal is contextually typed by the implied type of an object binding pattern:

- Properties with default values in the object binding pattern become optional in the object literal.
- Properties in the object binding pattern that have no match in the object literal are required to have a default value in the object binding pattern and are automatically added to the object literal type.
- Properties in the object literal that have no match in the object binding pattern are an error.

When an array literal is contextually typed by the implied type of an array binding pattern:

- Elements in the array binding pattern that have no match in the array literal are required to have a default value in the array binding pattern and are automatically added to the array literal type.

Example

```
// Type of f1 is (arg?: { x?: number, y?: number }) => void
function f1({ x = 0, y = 0 } = {}) {}

// And can be called as:
f1();
f1({});
f1({ x: 1 });
f1({ y: 1 });
f1({ x: 1, y: 1 });

// Type of f2 is (arg?: { x: number, y?: number }) => void
function f2({ x, y = 0 } = { x: 0 }) {}

f2();
f2({}); // Error, x not optional
f2({ x: 1 });
f2({ y: 1 }); // Error, x not optional
f2({ x: 1, y: 1 });
```

Support for decorators when targeting ES3

Decorators are now allowed when targeting ES3. TypeScript 1.7 removes the ES5-specific use of `reduceRight` from the `__decorate` helper. The changes also inline calls `Object.getOwnPropertyDescriptor` and `Object.defineProperty` in a backwards-compatible fashion that allows for a clean up the emit for ES5 and later by removing various repetitive calls to the aforementioned `Object` methods.

[Go to TOC](#)

Type parameters as constraints

With TypeScript 1.8 it becomes possible for a type parameter constraint to reference type parameters from the same type parameter list. Previously this was an error. This capability is usually referred to as [F-Bounded Polymorphism](#).

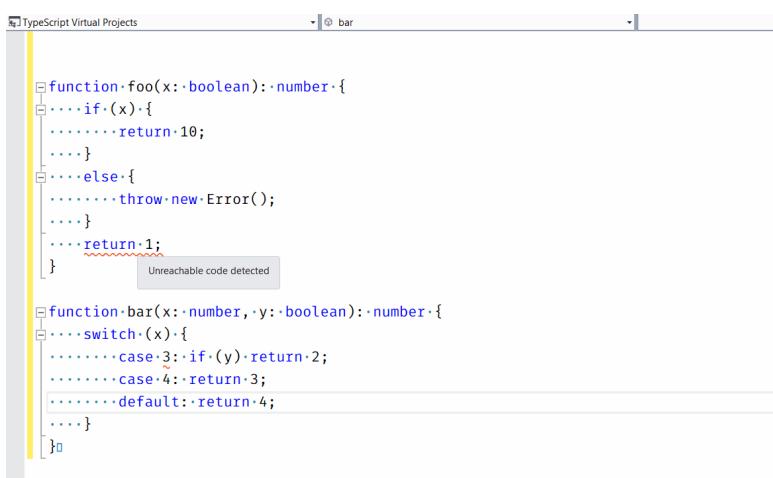
Example

```
function assign<T extends U, U>(target: T, source: U): T {
  for (let id in source) {
    target[id] = source[id];
  }
  return target;
}

let x = { a: 1, b: 2, c: 3, d: 4 };
assign(x, { b: 10, d: 20 });
assign(x, { e: 0 }); // Error
```

Control flow analysis errors

TypeScript 1.8 introduces control flow analysis to help catch common errors that users tend to run into. Read on to get more details, and check out these errors in action:



The screenshot shows a TypeScript code editor interface with the title bar "TypeScript Virtual Projects" and a tab labeled "bar". The code editor displays two functions: `function foo(x: boolean): number {` and `function bar(x: number, y: boolean): number {`. In the `foo` function, there is an `if` statement with a condition `(x)`. The body of the `if` block contains a `return 10;` statement. Below the `if` block, there is an `else` block containing a `throw new Error();` statement and a `return 1;` statement. A tooltip "Unreachable code detected" appears over the `return 1;` statement. In the `bar` function, there is a `switch` statement with a variable `(x)`. It has three cases: `case 3: if (y) return 2;`, `case 4: return 3;`, and a default case `default: return 4;`. The code editor uses color coding and underlines to highlight errors and warnings.

cfa

Unreachable code

Statements guaranteed to not be executed at run time are now correctly flagged as unreachable code errors. For instance, statements following unconditional `return`, `throw`, `break` or `continue` statements are considered unreachable. Use `allowUnreachableCode` to disable unreachable code detection and reporting.

Example

Here's a simple example of an unreachable code error:

```
function f(x) {
  if (x) {
    return true;
  } else {
    return false;
  }

  x = 0; // Error: Unreachable code detected.
}
```

A more common error that this feature catches is adding a newline after a `return` statement:

```
function f() {
  return; // Automatic Semicolon Insertion triggered at newline
  {
    x: "string"; // Error: Unreachable code detected.
}
```

Since JavaScript automatically terminates the `return` statement at the end of the line, the object literal becomes a block.

Unused labels

Unused labels are also flagged. Just like unreachable code checks, these are turned on by default; use `allowUnusedLabels` to stop reporting these errors.

Example

```
loop: while (x > 0) {
  // Error: Unused label.
  x++;
}
```

Implicit returns

Functions with code paths that do not return a value in JS implicitly return `undefined`. These can now be flagged by the compiler as implicit returns. The check is turned off by default; use `noImplicitReturns` to turn it on.

Example

```
function f(x) {
  // Error: Not all code paths return a value.
  if (x) {
    return false;
  }
  // implicitly returns `undefined`
}
```

Case clause fall-throughs

TypeScript can report errors for fall-through cases in switch statement where the case clause is non-empty. This check is turned off by default, and can be enabled using `noFallthroughCasesInSwitch`.

Example

With `noFallthroughCasesInSwitch`, this example will trigger an error:

```
switch (x % 2) {
  case 0: // Error: Fallthrough case in switch.
  console.log("even");

  case 1:
  console.log("odd");
  break;
}
```

However, in the following example, no error will be reported because the fall-through case is empty:

```
switch (x % 3) {
  case 0:
  case 1:
  console.log("Acceptable");
  break;

  case 2:
  console.log("This is *two much*!");
  break;
}
```

Function Components in React

TypeScript now supports [Function components](#). These are lightweight components that easily compose other components:

```
// Use parameter destructuring and defaults for easy definition of 'props' type
const Greeter = ({ name = "world" }) => <div>Hello, {name}!</div>;

// Properties get validated
let example = <Greeter name="TypeScript 1.8" />;
```

For this feature and simplified props, be sure to use the [latest version of react.d.ts](#).

Simplified `props` type management in React

In TypeScript 1.8 with the latest version of react.d.ts (see above), we've also greatly simplified the declaration of `props` types.

Specifically:

- You no longer need to either explicitly declare `ref` and `key` or `extend React.PropTypes`
- The `ref` and `key` properties will appear with correct types on all components
- The `ref` property is correctly disallowed on instances of Stateless Function components

Augmenting global/module scope from modules

Users can now declare any augmentations that they want to make, or that any other consumers already have made, to an existing module. Module augmentations look like plain old ambient module declarations (i.e. the `declare module "foo" { }` syntax), and are directly nested either your own modules, or in another top level ambient external module.

Furthermore, TypeScript also has the notion of *global* augmentations of the form `declare global { }`. This allows modules to augment global types such as `Array` if necessary.

The name of a module augmentation is resolved using the same set of rules as module specifiers in `import` and `export` declarations. The declarations in a module augmentation are merged with any existing declarations the same way they would if they were declared in the same file.

Neither module augmentations nor global augmentations can add new items to the top level scope - they can only "patch" existing declarations.

Example

Here `map.ts` can declare that it will internally patch the `Observable` type from `observable.ts` and add the `map` method to it.

```
// observable.ts
export class Observable<T> {
  // ...
}

// map.ts
import { Observable } from "./observable";

// Create an augmentation for "./observable"
declare module "./observable" {

  // Augment the 'Observable' class definition with interface merging
  interface Observable<T> {
    map<U>(proj: (el: T) => U): Observable<U>;
  }
}

Observable.prototype.map = /*...*/;
```

```
// consumer.ts
import { Observable } from "./observable";
import "./map";

let o: Observable<number>;
o.map((x) => x.toFixed());
```

Similarly, the global scope can be augmented from modules using a `declare global` declarations:

Example

```
// Ensure this is treated as a module.
export {};

declare global {
    interface Array<T> {
        mapToNumbers(): number[];
    }
}

Array.prototype.mapToNumbers = function () {
    /* ... */
};
```

String literal types

It's not uncommon for an API to expect a specific set of strings for certain values. For instance, consider a UI library that can move elements across the screen while controlling the "easing" of the animation.

```
declare class UIElement {
    animate(options: AnimationOptions): void;
}

interface AnimationOptions {
    deltaX: number;
    deltaY: number;
    easing: string; // Can be "ease-in", "ease-out", "ease-in-out"
}
```

However, this is error prone - there is nothing stopping a user from accidentally misspelling one of the valid easing values:

```
// No errors
new UIElement().animate({ deltaX: 100, deltaY: 100, easing: "ease-inout" });
```

With TypeScript 1.8, we've introduced string literal types. These types are written the same way string literals are, but in type positions.

Users can now ensure that the type system will catch such errors. Here's our new `AnimationOptions` using string literal types:

```
interface AnimationOptions {
    deltaX: number;
    deltaY: number;
    easing: "ease-in" | "ease-out" | "ease-in-out";
}
```

```
// Error: Type '"ease-in-out"' is not assignable to type '"ease-in" | "ease-out" | "ease-in-out"'
new UIElement().animate({ deltaX: 100, deltaY: 100, easing: "ease-in-out" });
```

Improved union/intersection type inference

TypeScript 1.8 improves type inference involving source and target sides that are both union or intersection types. For example, when inferring from `string | string[]` to `string | T`, we reduce the types to `string[]` and `T`, thus inferring `string[]` for `T`.

Example

```
type Maybe<T> = T | void;

function isDefined<T>(x: Maybe<T>): x is T {
    return x !== undefined && x !== null;
}

function isUndefined<T>(x: Maybe<T>): x is void {
    return x === undefined || x === null;
}

function getOrElse<T>(x: Maybe<T>, defaultValue: T): T {
    return isDefined(x) ? x : defaultValue;
}

function test1(x: Maybe<string>) {
    let x1 = getOrElse(x, "Undefined"); // string
    let x2 = isDefined(x) ? x : "Undefined"; // string
    let x3 = isUndefined(x) ? "Undefined" : x; // string
}

function test2(x: Maybe<number>) {
    let x1 = getOrElse(x, -1); // number
    let x2 = isDefined(x) ? x : -1; // number
    let x3 = isUndefined(x) ? -1 : x; // number
}
```

Concatenate AMD and System modules with `-- outFile`

Specifying `outFile` in conjunction with `--module amd` or `--module system` will concatenate all modules in the compilation into a single output file containing multiple module closures.

A module name will be computed for each module based on its relative location to `rootDir`.

Example

```
// file src/a.ts
import * as B from "./lib/b";
export function createA() {
    return B.createB();
}
```

```
// file src/lib/b.ts
export function createB() {
    return {};
}
```

Results in:

```
define("lib/b", ["require", "exports"], function (require, exports) {
    "use strict";
    function createB() {
        return {};
    }
    exports.createB = createB;
});
define("a", ["require", "exports", "lib/b"], function (require, exports, B) {
    "use strict";
    function createA() {
        return B.createB();
    }
    exports.createA = createA;
});
```

Support for `default import` interop with SystemJS

Module loaders like SystemJS wrap CommonJS modules and expose them as a `default` ES6 import. This makes it impossible to share the definition files between the SystemJS and CommonJS implementation of the module as the module shape looks different based on the loader.

Setting the new compiler flag `allowSyntheticDefaultImports` indicates that the module loader performs some kind of synthetic default import member creation not indicated in the imported .ts or .d.ts. The compiler will infer the existence of a `default` export that has the shape of the entire module itself.

System modules have this flag on by default.

Allow captured `let / const` in loops

Previously an error, now supported in TypeScript 1.8. `let / const` declarations within loops and captured in functions are now emitted to correctly match `let / const` freshness semantics.

Example

```
let list = [];
for (let i = 0; i < 5; i++) {
    list.push(() => i);
}

list.forEach((f) => console.log(f()));
```

is compiled to:

```
var list = [];
var _loop_1 = function (i) {
    list.push(function () {
        return i;
    });
}
for (var i = 0; i < 5; i++) {
    _loop_1(i);
}
```

```
list.forEach(function (f) {
    return console.log(f());
});
```

And results in

```
0
1
2
3
4
```

Improved checking for `for..in` statements

Previously the type of a `for..in` variable is inferred to `any`; that allowed the compiler to ignore invalid uses within the `for..in` body.

Starting with TypeScript 1.8:

- The type of a variable declared in a `for..in` statement is implicitly `string`.
- When an object with a numeric index signature of type `T` (such as an array) is indexed by a `for..in` variable of a containing `for..in` statement for an object *with* a numeric index signature and *without* a string index signature (again such as an array), the value produced is of type `T`.

Example

```
var a: MyObject[]; // Type of a is MyObject[]
for (var x in a) { // Type of x is implicitly string
    var obj = a[x]; // Type of obj is MyObject
}
```

Modules are now emitted with a `"use strict"` prologue

Modules were always parsed in strict mode as per ES6, but for non-ES6 targets this was not respected in the generated code. Starting with TypeScript 1.8, emitted modules are always in strict mode. This shouldn't have any visible changes in most code as TS considers most strict mode errors as errors at compile time, but it means that some things which used to silently fail at runtime in your TS code, like assigning to `Nan`, will now loudly fail. You can reference the [MDN Article](#) on strict mode for a detailed list of the differences between strict mode and non-strict mode.

Including `.js` files with `--allowJs`

Often there are external source files in your project that may not be authored in TypeScript. Alternatively, you might be in the middle of converting a JS code base into TS, but still want to bundle all your JS code into a single file with the output of your new TS code.

.js files are now allowed as input to tsc. The TypeScript compiler checks the input .js files for syntax errors, and emits valid output based on the target and module flags. The output can be combined with other .ts files as well. Source maps are still generated for .js files just like with .ts files.

Custom JSX factories using --reactNamespace

Passing --reactNamespace <JSX factory Name> along with --jsx react allows for using a different JSX factory from the default React.

The new factory name will be used to call createElement and __spread functions.

Example

```
import { jsxFactory } from "jsxFactory";
var div = <div>Hello JSX!</div>;
```

Compiled with:

```
tsc --jsx react --reactNamespace jsxFactory --m commonJS
```

Results in:

```
"use strict";
var jsxFactory_1 = require("jsxFactory");
var div = jsxFactory_1.jsxFactory.createElement("div", null, "Hello JSX!");
```

this-based type guards

TypeScript 1.8 extends user-defined type guard functions to class and interface methods.

this is T is now valid return type annotation for methods in classes and interfaces. When used in a type narrowing position (e.g. if statement), the type of the call expression target object would be narrowed to T.

Example

```
class FileSystemObject {
  isFile(): this is File {
    return this instanceof File;
  }
  isDirectory(): this is Directory {
    return this instanceof Directory;
  }
  isNetworked(): this is Networked & this {
    return this.networked;
  }
  constructor(public path: string, private networked: boolean) {}
}

class File extends FileSystemObject {
  constructor(path: string, public content: string) {
    super(path, false);
  }
}
```

```

    }
}

class Directory extends FileSystemObject {
    children: FileSystemObject[]; 
}
interface Networked {
    host: string;
}

let fso: FileSystemObject = new File("foo/bar.txt", "foo");
if (fso.isFile()) {
    fso.content; // fso is File
} else if (fso.isDirectory()) {
    fso.children; // fso is Directory
} else if (fso.isNetworked()) {
    fso.host; // fso is networked
}

```

Official TypeScript NuGet package

Starting with TypeScript 1.8, official NuGet packages are available for the TypeScript Compiler (`tsc.exe`) as well as the MSBuild integration (`Microsoft.TypeScript.targets` and `Microsoft.TypeScript.Tasks.dll`).

Stable packages are available here:

- [Microsoft.TypeScript.Compiler](#)
- [Microsoft.TypeScript.MSBuild](#)

Also, a nightly NuGet package to match the [nightly npm package](#) is available on [myget](#):

- [TypeScript-Preview](#)

Prettier error messages from `tsc`

We understand that a ton of monochrome output can be a little difficult on the eyes. Colors can help discern where a message starts and ends, and these visual clues are important when error output gets overwhelming.

By just passing the `pretty` command line option, TypeScript gives more colorful output with context about where things are going wrong.

```

Windows PowerShell (Admin)
PS C:\ifeelPretty> tsc --watch --pretty
    await reticulateSplines();

ohSoPretty.ts(18,5): error TS1308: 'await' expression is only allowed within an async function.

13:32:44 - Compilation complete. Watching for file changes.

node.exe[1324992]
+150513[64] 1/1 [-] NUM PR1 99x23 (1.9) 25V 15108 100%

```

Showing off pretty error messages in ConEmu

Colorization of JSX code in VS 2015

With TypeScript 1.8, JSX tags are now classified and colorized in Visual Studio 2015.

```

123ts* = X
File typeScript Virtual Projects
///<reference path="D:\sources\git\react.d.ts"/>
let React: any;
function render() {
    return <div width="5" className="some-class-name">
        some text inside
    </div>
}

```

jsx

The classification can be further customized by changing the font and color settings for the `VB XML` color and font settings through `Tools -> Options -> Environment -> Fonts and Colors` page.

The `--project (-p)` flag can now take any file path

The `--project` command line option originally could only take paths to a folder containing a `tsconfig.json`. Given the different scenarios for build configurations, it made sense to allow `--project` to point to any other compatible JSON file. For instance, a user might want to target ES2015 with CommonJS modules for Node 5, but ES5 with AMD modules for the browser. With this new work, users can easily manage two separate build targets using `tsc` alone without having to perform hacky workarounds like placing `tsconfig.json` files in separate directories.

The old behavior still remains the same if given a directory - the compiler will try to find a file in the directory named `tsconfig.json`.

Allow comments in tsconfig.json

It's always nice to be able to document your configuration! `tsconfig.json` now accepts single and multi-line comments.

```
{
  "compilerOptions": {
    "target": "ES2015", // running on node v5, yaay!
    "sourceMap": true // makes debugging easier
  },
  /* Excluded files
  */
  "exclude": ["file.d.ts"]
}
```

Support output to IPC-driven files

TypeScript 1.8 allows users to use the `outFile` argument with special file system entities like named pipes, devices, etc.

As an example, on many Unix-like systems, the standard output stream is accessible by the file `/dev/stdout`.

```
tsc foo.ts --outFile /dev/stdout
```

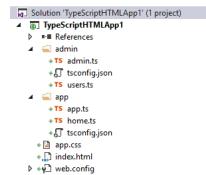
This can be used to pipe output between commands as well.

As an example, we can pipe our emitted JavaScript into a pretty printer like `pretty-js`:

```
tsc foo.ts --outFile /dev/stdout | pretty-js
```

Improved support for `tsconfig.json` in Visual Studio 2015

TypeScript 1.8 allows `tsconfig.json` files in all project types. This includes ASP.NET v4 projects, *Console Application*, and the *Html Application with TypeScript* project types. Further, you are no longer limited to a single `tsconfig.json` file but can add multiple, and each will be built as part of the project. This allows you to separate the configuration for different parts of your application without having to use multiple different projects.



Showing off `tsconfig.json` in Visual Studio

We also disable the project properties page when you add a `tsconfig.json` file. This means that all configuration changes have to be made in the `tsconfig.json` file itself.

A couple of limitations

- If you add a `tsconfig.json` file, TypeScript files that are not considered part of that context are not compiled.
- Apache Cordova Apps still have the existing limitation of a single `tsconfig.json` file, which must be in either the root or the `scripts` folder.
- There is no template for `tsconfig.json` in most project types.

[Go to TOC](#)

Null- and undefined-aware types

TypeScript has two special types, Null and Undefined, that have the values `null` and `undefined` respectively. Previously it was not possible to explicitly name these types, but `null` and `undefined` may now be used as type names regardless of type checking mode.

The type checker previously considered `null` and `undefined` assignable to anything. Effectively, `null` and `undefined` were valid values of *every* type and it wasn't possible to specifically exclude them (and therefore not possible to detect erroneous use of them).

--strictNullChecks

`strictNullChecks` switches to a new strict null checking mode.

In strict null checking mode, the `null` and `undefined` values are *not* in the domain of every type and are only assignable to themselves and `any` (the one exception being that `undefined` is also assignable to `void`). So, whereas `T` and `T | undefined` are considered synonymous in regular type checking mode (because `undefined` is considered a subtype of any `T`), they are different types in strict type checking mode, and only `T | undefined` permits `undefined` values. The same is true for the relationship of `T` to `T | null`.

Example

```
// Compiled with --strictNullChecks
let x: number;
let y: number | undefined;
let z: number | null | undefined;
x = 1; // Ok
y = 1; // Ok
z = 1; // Ok
x = undefined; // Error
y = undefined; // Ok
z = undefined; // Ok
x = null; // Error
y = null; // Error
z = null; // Ok
x = y; // Error
x = z; // Error
y = x; // Ok
y = z; // Error
z = x; // Ok
z = y; // Ok
```

Assigned-before-use checking

In strict null checking mode the compiler requires every reference to a local variable of a type that doesn't include `undefined` to be preceded by an assignment to that variable in every possible preceding code path.

Example

```
// Compiled with --strictNullChecks
let x: number;
let y: number | null;
let z: number | undefined;
x; // Error, reference not preceded by assignment
y; // Error, reference not preceded by assignment
z; // Ok
x = 1;
y = null;
x; // Ok
y; // Ok
```

The compiler checks that variables are definitely assigned by performing *control flow based type analysis*. See later for further details on this topic.

Optional parameters and properties

Optional parameters and properties automatically have `undefined` added to their types, even when their type annotations don't specifically include `undefined`. For example, the following two types are identical:

```
// Compiled with --strictNullChecks
type T1 = (x?: number) => string; // x has type number | undefined
type T2 = (x?: number | undefined) => string; // x has type number | undefined
```

Non-null and non-undefined type guards

A property access or a function call produces a compile-time error if the object or function is of a type that includes `null` or `undefined`. However, type guards are extended to support non-null and non-undefined checks.

Example

```
// Compiled with --strictNullChecks
declare function f(x: number): string;
let x: number | null | undefined;
if (x) {
  f(x); // Ok, type of x is number here
} else {
  f(x); // Error, type of x is number? here
}
let a = x != null ? f(x) : ""; // Type of a is string
let b = x && f(x); // Type of b is string | 0 | null | undefined
```

Non-null and non-undefined type guards may use the `==`, `!=`, `====`, or `!==` operator to compare to `null` or `undefined`, as in `x != null` or `x === undefined`. The effects on subject variable types accurately reflect JavaScript semantics (e.g. double-equals operators check for both values no matter which one is specified whereas triple-equals only checks for the specified value).

Dotted names in type guards

Type guards previously only supported checking local variables and parameters. Type guards now support checking "dotted names" consisting of a variable or parameter name followed one or more property accesses.

Example

```
interface Options {
  location?: {
    x?: number;
    y?: number;
  };
}

function foo(options?: Options) {
  if (options && options.location && options.location.x) {
    const x = options.location.x; // Type of x is number
  }
}
```

Type guards for dotted names also work with user defined type guard functions and the `typeof` and `instanceof` operators and do not depend on the `strictNullChecks` compiler option.

A type guard for a dotted name has no effect following an assignment to any part of the dotted name. For example, a type guard for `x.y.z` will have no effect following an assignment to `x`, `x.y`, or `x.y.z`.

Expression operators

Expression operators permit operand types to include `null` and/or `undefined` but always produce values of non-null and non-undefined types.

```
// Compiled with --strictNullChecks
function sum(a: number | null, b: number | null) {
  return a + b; // Produces value of type number
}
```

The `&&` operator adds `null` and/or `undefined` to the type of the right operand depending on which are present in the type of the left operand, and the `||` operator removes both `null` and `undefined` from the type of the left operand in the resulting union type.

```
// Compiled with --strictNullChecks
interface Entity {
  name: string;
}
let x: Entity | null;
let s = x && x.name; // s is of type string | null
let y = x || { name: "test" }; // y is of type Entity
```

Type widening

The `null` and `undefined` types are *not* widened to `any` in strict null checking mode.

```
let z = null; // Type of z is null
```

In regular type checking mode the inferred type of `z` is `any` because of widening, but in strict null checking mode the inferred type of `z` is `null` (and therefore, absent a type annotation, `null` is the only possible value for `z`).

Non-null assertion operator

A new `!` post-fix expression operator may be used to assert that its operand is non-null and non-undefined in contexts where the type checker is unable to conclude that fact. Specifically, the operation `x!` produces a value of the type of `x` with `null` and `undefined` excluded. Similar to type assertions of the forms `<T>x` and `x as T`, the `!` non-null assertion operator is simply removed in the emitted JavaScript code.

```
// Compiled with --strictNullChecks
function validateEntity(e?: Entity) {
  // Throw exception if e is null or invalid entity
}

function processEntity(e?: Entity) {
  validateEntity(e);
  let s = e!.name; // Assert that e is non-null and access name
}
```

Compatibility

The new features are designed such that they can be used in both strict null checking mode and regular type checking mode. In particular, the `null` and `undefined` types are automatically erased from union types in regular type checking mode (because they are subtypes of all other types), and the `!` non-null assertion expression operator is permitted but has no effect in regular type checking mode. Thus, declaration files that are updated to use null- and undefined-aware types can still be used in regular type checking mode for backwards compatibility.

In practical terms, strict null checking mode requires that all files in a compilation are null- and undefined-aware.

Control flow based type analysis

TypeScript 2.0 implements a control flow-based type analysis for local variables and parameters. Previously, the type analysis performed for type guards was limited to `if` statements and `?:` conditional expressions and didn't include effects of assignments and control flow constructs such as `return` and `break` statements. With TypeScript 2.0, the type checker analyses all possible flows of control in statements and expressions to produce the most specific type possible (the *narrowed type*) at any given location for a local variable or parameter that is declared to have a union type.

Example

```
function foo(x: string | number | boolean) {
  if (typeof x === "string") {
    x; // type of x is string here
  }
}
```

```

    x = 1;
    x; // type of x is number here
}
x; // type of x is number | boolean here
}

function bar(x: string | number) {
  if (typeof x === "number") {
    return;
  }
  x; // type of x is string here
}

```

Control flow based type analysis is particularly relevant in `strictNullChecks` mode because nullable types are represented using union types:

```

function test(x: string | null) {
  if (x === null) {
    return;
  }
  x; // type of x is string in remainder of function
}

```

Furthermore, in `strictNullChecks` mode, control flow based type analysis includes *definite assignment analysis* for local variables of types that don't permit the value `undefined`.

```

function mumble(check: boolean) {
  let x: number; // Type doesn't permit undefined
  x; // Error, x is undefined
  if (check) {
    x = 1;
    x; // Ok
  }
  x; // Error, x is possibly undefined
  x = 2;
  x; // Ok
}

```

Tagged union types

TypeScript 2.0 implements support for tagged (or discriminated) union types. Specifically, the TS compiler now support type guards that narrow union types based on tests of a discriminant property and furthermore extend that capability to `switch` statements.

Example

```

interface Square {
  kind: "square";
  size: number;
}

interface Rectangle {
  kind: "rectangle";
  width: number;
  height: number;
}

```

```

interface Circle {
  kind: "circle";
  radius: number;
}

type Shape = Square | Rectangle | Circle;

function area(s: Shape) {
  // In the following switch statement, the type of s is narrowed in each case
  // clause
  // according to the value of the discriminant property, thus allowing the other
  // properties
  // of that variant to be accessed without a type assertion.
  switch (s.kind) {
    case "square":
      return s.size * s.size;
    case "rectangle":
      return s.width * s.height;
    case "circle":
      return Math.PI * s.radius * s.radius;
  }
}

function test1(s: Shape) {
  if (s.kind === "square") {
    s; // Square
  } else {
    s; // Rectangle | Circle
  }
}

function test2(s: Shape) {
  if (s.kind === "square" || s.kind === "rectangle") {
    return;
  }
  s; // Circle
}

```

A *discriminant property type guard* is an expression of the form `x.p == v`, `x.p === v`, `x.p != v`, or `x.p !== v`, where `p` and `v` are a property and an expression of a string literal type or a union of string literal types. The discriminant property type guard narrows the type of `x` to those constituent types of `x` that have a discriminant property `p` with one of the possible values of `v`.

Note that we currently only support discriminant properties of string literal types. We intend to later add support for boolean and numeric literal types.

The `never` type

TypeScript 2.0 introduces a new primitive type `never`. The `never` type represents the type of values that never occur. Specifically, `never` is the return type for functions that never return and `never` is the type of variables under type guards that are never true.

The `never` type has the following characteristics:

- `never` is a subtype of and assignable to every type.
- No type is a subtype of or assignable to `never` (except `never` itself).

- In a function expression or arrow function with no return type annotation, if the function has no `return` statements, or only `return` statements with expressions of type `never`, and if the end point of the function is not reachable (as determined by control flow analysis), the inferred return type for the function is `never`.
- In a function with an explicit `never` return type annotation, all `return` statements (if any) must have expressions of type `never` and the end point of the function must not be reachable.

Because `never` is a subtype of every type, it is always omitted from union types and it is ignored in function return type inference as long as there are other types being returned.

Some examples of functions returning `never`:

```
// Function returning never must have unreachable end point
function error(message: string): never {
  throw new Error(message);
}

// Inferred return type is never
function fail() {
  return error("Something failed");
}

// Function returning never must have unreachable end point
function infiniteLoop(): never {
  while (true) {}
}
```

Some examples of use of functions returning `never`:

```
// Inferred return type is number
function move1(direction: "up" | "down") {
  switch (direction) {
    case "up":
      return 1;
    case "down":
      return -1;
  }
  return error("Should never get here");
}

// Inferred return type is number
function move2(direction: "up" | "down") {
  return direction === "up"
    ? 1
    : direction === "down"
    ? -1
    : error("Should never get here");
}

// Inferred return type is T
function check<T>(x: T | undefined) {
  return x || error("Undefined value");
}
```

Because `never` is assignable to every type, a function returning `never` can be used when a callback returning a more specific type is required:

```

function test(cb: () => string) {
  let s = cb();
  return s;
}

test(() => "hello");
test(() => fail());
test(() => {
  throw new Error();
});

```

Read-only properties and index signatures

A property or index signature can now be declared with the `readonly` modifier is considered read-only.

Read-only properties may have initializers and may be assigned to in constructors within the same class declaration, but otherwise assignments to read-only properties are disallowed.

In addition, entities are *implicitly* read-only in several situations:

- A property declared with a `get` accessor and no `set` accessor is considered read-only.
- In the type of an enum object, enum members are considered read-only properties.
- In the type of a module object, exported `const` variables are considered read-only properties.
- An entity declared in an `import` statement is considered read-only.
- An entity accessed through an ES2015 namespace import is considered read-only (e.g. `foo.x` is read-only when `foo` is declared as `import * as foo from "foo"`).

Example

```

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

var p1: Point = { x: 10, y: 20 };
p1.x = 5; // Error, p1.x is read-only

var p2 = { x: 1, y: 1 };
var p3: Point = p2; // Ok, read-only alias for p2
p3.x = 5; // Error, p3.x is read-only
p2.x = 5; // Ok, but also changes p3.x because of aliasing

```

```

class Foo {
  readonly a = 1;
  readonly b: string;
  constructor() {
    this.b = "hello"; // Assignment permitted in constructor
  }
}

```

```

let a: Array<number> = [0, 1, 2, 3, 4];
let b: ReadonlyArray<number> = a;
b[5] = 5; // Error, elements are read-only
b.push(5); // Error, no push method (because it mutates array)
b.length = 3; // Error, length is read-only
a = b; // Error, mutating methods are missing

```

Specifying the type of `this` for functions

Following up on specifying the type of `this` in a class or an interface, functions and methods can now declare the type of `this` they expect.

By default the type of `this` inside a function is `any`. Starting with TypeScript 2.0, you can provide an explicit `this` parameter. `this` parameters are fake parameters that come first in the parameter list of a function:

```
function f(this: void) {
  // make sure `this` is unusable in this standalone function
}
```

`this` parameters in callbacks

Libraries can also use `this` parameters to declare how callbacks will be invoked.

Example

```
interface UIElement {
  addClickListener(onclick: (this: void, e: Event) => void): void;
}
```

`this: void` means that `addClickListener` expects `onclick` to be a function that does not require a `this` type.

Now if you annotate calling code with `this`:

```
class Handler {
  info: string;
  onClickBad(this: Handler, e: Event) {
    // oops, used this here. using this callback would crash at runtime
    this.info = e.message;
  }
}
let h = new Handler();
uiElement.addClickListener(h.onClickBad); // error!
```

--noImplicitThis

A new flag is also added in TypeScript 2.0 to flag all uses of `this` in functions without an explicit type annotation.

Glob support in `tsconfig.json`

Glob support is here!! Glob support has been [one of the most requested features](#).

Glob-like file patterns are supported two properties `include` and `exclude`.

Example

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

The supported glob wildcards are:

- `*` matches zero or more characters (excluding directory separators)
- `?` matches any one character (excluding directory separators)
- `**/` recursively matches any subdirectory

If a segment of a glob pattern includes only `*` or `.*`, then only files with supported extensions are included (e.g. `.ts`, `.tsx`, and `.d.ts` by default with `.js` and `.jsx` if `allowJs` is set to true).

If the `files` and `include` are both left unspecified, the compiler defaults to including all TypeScript (`.ts`, `.d.ts` and `.tsx`) files in the containing directory and subdirectories except those excluded using the `exclude` property. JS files (`.js` and `.jsx`) are also included if `allowJs` is set to true.

If the `files` or `include` properties are specified, the compiler will instead include the union of the files included by those two properties. Files in the directory specified using the `outDir` compiler option are always excluded unless explicitly included via the `files` property (even when the `exclude` property is specified).

Files included using `include` can be filtered using the `exclude` property. However, files included explicitly using the `files` property are always included regardless of `exclude`. The `exclude` property defaults to excluding the `node_modules`, `bower_components`, and `jspm_packages` directories when not specified.

Module resolution enhancements: baseUrl, Path mapping, rootDirs and tracing

TypeScript 2.0 provides a set of additional module resolution knobs to *inform* the compiler where to find declarations for a given module.

See [Module Resolution](#) documentation for more details.

Base URL

Using a `baseUrl` is a common practice in applications using AMD module loaders where modules are "deployed" to a single folder at run-time. All module imports with non-relative names are assumed to be relative to the `baseUrl`.

Example

```
{
  "compilerOptions": {
    "baseUrl": "./modules"
  }
}
```

Now imports to `"moduleA"` would be looked up in `./modules/moduleA`

```
import A from "moduleA";
```

Path mapping

Sometimes modules are not directly located under `baseUrl`. Loaders use a mapping configuration to map module names to files at run-time, see [RequireJs documentation](#) and [SystemJS documentation](#).

The TypeScript compiler supports the declaration of such mappings using `paths` property in `tsconfig.json` files.

Example

For instance, an import to a module `"jquery"` would be translated at runtime to `"node_modules/jquery/dist/jquery.slim.min.js"`.

```
{
  "compilerOptions": {
    "baseUrl": "./node_modules",
    "paths": {
      "jquery": ["jquery/dist/jquery.slim.min"]
    }
  }
}
```

Using `paths` also allow for more sophisticated mappings including multiple fall back locations. Consider a project configuration where only some modules are available in one location, and the rest are in another.

Virtual Directories with `rootDirs`

Using `'rootDirs'`, you can inform the compiler of the `roots` making up this "virtual" directory; and thus the compiler can resolve relative modules imports within these "virtual" directories *as if* they were merged together in one directory.

Example

Given this project structure:

```

src
  __ views
    __ view1.ts (imports './template1')
    __ view2.ts
generated
```

```

└── templates
    └── views
        └── template1.ts (imports './view2')

```

A build step will copy the files in `/src/views` and `/generated/templates/views` to the same directory in the output. At run-time, a view can expect its template to exist next to it, and thus should import it using a relative name as `"./template"`.

`rootDirs` specify a list of `roots` whose contents are expected to merge at run-time. So following our example, the `tsconfig.json` file should look like:

```
{
  "compilerOptions": {
    "rootDirs": ["src/views", "generated/templates/views"]
  }
}
```

Tracing module resolution

`traceResolution` offers a handy way to understand how modules have been resolved by the compiler.

```
tsc --traceResolution
```

Shorthand ambient module declarations

If you don't want to take the time to write out declarations before using a new module, you can now just use a shorthand declaration to get started quickly.

`declarations.d.ts`

```
declare module "hot-new-module";
```

All imports from a shorthand module will have the any type.

```
import x, { y } from "hot-new-module";
x(y);
```

Wildcard character in module names

Importing none-code resources using module loaders extension (e.g. [AMD](#) or [SystemJS](#)) has not been easy before; previously an ambient module declaration had to be defined for each resource.

TypeScript 2.0 supports the use of the wildcard character (`*`) to declare a "family" of module names; this way, a declaration is only required once for an extension, and not for every resource.

Example

```
declare module "*!text" {
  const content: string;
  export default content;
}
// Some do it the other way around.
```

```
declare module "json!*" {
  const value: any;
  export default value;
}
```

Now you can import things that match `"!*text"` or `"json!*"`.

```
import fileContent from "./xyz.txt!text";
import data from "json!http://example.com/data.json";
console.log(data, fileContent);
```

Wildcard module names can be even more useful when migrating from an un-typed code base. Combined with Shorthand ambient module declarations, a set of modules can be easily declared as `any`.

Example

```
declare module "myLibrary/*";
```

All imports to any module under `myLibrary` would be considered to have the type `any` by the compiler; thus, shutting down any checking on the shapes or types of these modules.

```
import { readFile } from "myLibrary/fileSystem/readFile`;
readFile(); // readFile is 'any'
```

Support for UMD module definitions

Some libraries are designed to be used in many module loaders, or with no module loading (global variables). These are known as [UMD](#) or [Isomorphic](#) modules. These libraries can be accessed through either an import or a global variable.

For example:

math-lib.d.ts

```
export const isPrime(x: number): boolean;
export as namespace mathLib;
```

The library can then be used as an import within modules:

```
import { isPrime } from "math-lib";
isPrime(2);
mathLib.isPrime(2); // ERROR: can't use the global definition from inside a module
```

It can also be used as a global variable, but only inside of a script. (A script is a file with no imports or exports.)

```
mathLib.isPrime(2);
```

Optional class properties

Optional properties and methods can now be declared in classes, similar to what is already permitted in interfaces.

Example

```
class Bar {
  a: number;
  b?: number;
  f(): {
    return 1;
  }
  g?(): number; // Body of optional method can be omitted
  h?(): {
    return 2;
  }
}
```

When compiled in `strictNullChecks` mode, optional properties and methods automatically have `undefined` included in their type. Thus, the `b` property above is of type `number | undefined` and the `g` method above is of type `((() => number) | undefined)`. Type guards can be used to strip away the `undefined` part of the type:

```
function test(x: Bar) {
  x.a; // number
  x.b; // number | undefined
  x.f(); // () => number
  x.g(); // ((() => number) | undefined)
  let f1 = x.f(); // number
  let g1 = x.g() && x.g(); // number | undefined
  let g2 = x.g ? x.g() : 0; // number
}
```

Private and Protected Constructors

A class constructor may be marked `private` or `protected`. A class with private constructor cannot be instantiated outside the class body, and cannot be extended. A class with protected constructor cannot be instantiated outside the class body, but can be extended.

Example

```
class Singleton {
  private static instance: Singleton;

  private constructor() {}

  static getInstance(): {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
    }
    return Singleton.instance;
  }
}
```

```
let e = new Singleton(); // Error: constructor of 'Singleton' is private.
let v = Singleton.getInstance();
```

Abstract properties and accessors

An abstract class can declare abstract properties and/or accessors. Any sub class will need to declare the abstract properties or be marked as abstract. Abstract properties cannot have an initializer. Abstract accessors cannot have bodies.

Example

```
abstract class Base {
  abstract name: string;
  abstract get value();
  abstract set value(v: number);
}

class Derived extends Base {
  name = "derived";
  value = 1;
}
```

Implicit index signatures

An object literal type is now assignable to a type with an index signature if all known properties in the object literal are assignable to that index signature. This makes it possible to pass a variable that was initialized with an object literal as parameter to a function that expects a map or dictionary:

```
function httpService(path: string, headers: { [x: string]: string }) {}

const headers = {
  "Content-Type": "application/x-www-form-urlencoded",
};

httpService("", { "Content-Type": "application/x-www-form-urlencoded" }); // Ok
httpService("", headers); // Now ok, previously wasn't
```

Including built-in type declarations with `--lib`

Getting to ES6/ES2015 built-in API declarations were only limited to `target: ES6`. Enter `lib`; with `lib` you can specify a list of built-in API declaration groups that you can chose to include in your project. For instance, if you expect your runtime to have support for `Map`, `Set` and `Promise` (e.g. most evergreen browsers today), just include `--lib es2015.collection,es2015.promise`. Similarly you can exclude declarations you do not want to include in your project, e.g. DOM if you are working on a node project using `--lib es5,es6`.

Here is a list of available API groups:

- dom
- webworker
- es5

- es6 / es2015
- es2015.core
- es2015.collection
- es2015.iterable
- es2015.promise
- es2015.proxy
- es2015.reflect
- es2015.generator
- es2015.symbol
- es2015.symbol.wellknown
- es2016
- es2016.array.include
- es2017
- es2017.object
- es2017.sharedmemory
- scripthost

Example

```
tsc --target es5 --lib es5,es2015.promise
```

```
"compilerOptions": {
  "lib": ["es5", "es2015.promise"]
}
```

Flag unused declarations with `--noUnusedParameters` and `--noUnusedLocals`

TypeScript 2.0 has two new flags to help you maintain a clean code base. `noUnusedParameters` flags any unused function or method parameters errors. `noUnusedLocals` flags any unused local (un-exported) declaration like variables, functions, classes, imports, etc... Also, unused private members of a class would be flagged as errors under `noUnusedLocals`.

Example

```
import B, { readFile } from "./b";
//          ^ Error: `B` declared but never used
readFile();

export function write(message: string, args: string[]) {
  //          ^^^^^ Error: 'arg' declared but never used.
  console.log(message);
}
```

Parameters declaration with names starting with `_` are exempt from the unused parameter checking. e.g.:

```
function returnNull(_a) {
  // OK
  return null;
}
```

Module identifiers allow for .js extension

Before TypeScript 2.0, a module identifier was always assumed to be extension-less; for instance, given an import as `import d from "./moduleA.js"`, the compiler looked up the definition of `"moduleA.js"` in `./moduleA.js.ts` or `./moduleA.js.d.ts`. This made it hard to use bundling/loading tools like [SystemJS](#) that expect URI's in their module identifier.

With TypeScript 2.0, the compiler will look up definition of `"moduleA.js"` in `./moduleA.ts` or `./moduleA.d.ts`.

Support 'target : es5' with 'module: es6'

Previously flagged as an invalid flag combination, `target: es5` and `'module: es6'` is now supported. This should facilitate using ES2015-based tree shakers like [rollup](#).

Trailing commas in function parameter and argument lists

Trailing comma in function parameter and argument lists are now allowed. This is an implementation for a [Stage-3 ECMAScript proposal](#) that emits down to valid ES3/ES5/ES6.

Example

```
function foo(
  bar: Bar,
  baz: Baz // trailing commas are OK in parameter lists
) {
  // Implementation...
}

foo(
  bar,
  baz // and in argument lists
);
```

New `--skipLibCheck`

TypeScript 2.0 adds a new `skipLibCheck` compiler option that causes type checking of declaration files (files with extension `.d.ts`) to be skipped. When a program includes large declaration files, the compiler spends a lot of time type checking declarations that are already known to not contain errors, and compile times may be significantly shortened by skipping declaration file type checks.

Since declarations in one file can affect type checking in other files, some errors may not be detected when `skipLibCheck` is specified. For example, if a non-declaration file augments a type declared in a declaration file, errors may result that are only reported when the declaration file is checked. However, in practice such situations are rare.

Allow duplicate identifiers across declarations

This has been one common source of duplicate definition errors. Multiple declaration files defining the same members on interfaces.

TypeScript 2.0 relaxes this constraint and allows duplicate identifiers across blocks, as long as they have *identical* types.

Within the same block duplicate definitions are still disallowed.

Example

```
interface Error {
  stack?: string;
}

interface Error {
  code?: string;
  path?: string;
  stack?: string; // OK
}
```

New `--declarationDir`

`declarationDir` allows for generating declaration files in a different location than JavaScript files.

[Go to TOC](#)

keyof and Lookup Types

In JavaScript it is fairly common to have APIs that expect property names as parameters, but so far it hasn't been possible to express the type relationships that occur in those APIs.

Enter Index Type Query or `keyof`; An indexed type query `keyof T` yields the type of permitted property names for `T`. A `keyof T` type is considered a subtype of `string`.

Example

```
interface Person {
  name: string;
  age: number;
  location: string;
}

type K1 = keyof Person; // "name" | "age" | "location"
type K2 = keyof Person[]; // "length" | "push" | "pop" | "concat" | ...
type K3 = keyof { [x: string]: Person }; // string
```

The dual of this is *indexed access types*, also called *lookup types*. Syntactically, they look exactly like an element access, but are written as types:

Example

```
type P1 = Person["name"]; // string
type P2 = Person["name" | "age"]; // string | number
type P3 = string["charAt"]; // (pos: number) => string
type P4 = string[]["push"]; // (...items: string[]) => number
type P5 = string[][], // string
```

You can use this pattern with other parts of the type system to get type-safe lookups.

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

function setProperty<T, K extends keyof T>(obj: T, key: K, value: T[K]) {
  obj[key] = value;
}

let x = { foo: 10, bar: "hello!" };

let foo = getProperty(x, "foo"); // number
let bar = getProperty(x, "bar"); // string

let oops = getProperty(x, "wargarbl"); // Error! "wargarbl" is not "foo" | "bar"
setProperty(x, "foo", "string"); // Error!, string expected number
```

Mapped Types

One common task is to take an existing type and make each of its properties entirely optional. Let's say we have a `Person`:

```
interface Person {
  name: string;
  age: number;
  location: string;
}
```

A partial version of it would be:

```
interface PartialPerson {
  name?: string;
  age?: number;
  location?: string;
}
```

with Mapped types, `PartialPerson` can be written as a generalized transformation on the type `Person` as:

```
type Partial<T> = {
  [P in keyof T]?: T[P];
};

type PartialPerson = Partial<Person>;
```

Mapped types are produced by taking a union of literal types, and computing a set of properties for a new object type. They're like [list comprehensions in Python](#), but instead of producing new elements in a list, they produce new properties in a type.

In addition to `Partial`, Mapped Types can express many useful transformations on types:

```
// Keep types the same, but make each property to be read-only.
type Readonly<T> = {
  readonly [P in keyof T]: T[P];
};

// Same property names, but make the value a promise instead of a concrete one
type Deferred<T> = {
  [P in keyof T]: Promise<T[P]>;
};

// Wrap proxies around properties of T
type Proxify<T> = {
  [P in keyof T]: { get(): T[P]; set(v: T[P]): void };
};
```

Partial, Readonly, Record, and Pick

`Partial` and `Readonly`, as described earlier, are very useful constructs. You can use them to describe some common JS routines like:

```
function assign<T>(obj: T, props: Partial<T>): void;
function freeze<T>(obj: T): Readonly<T>;
```

Because of that, they are now included by default in the standard library.

We're also including two other utility types as well: `Record` and `Pick`.

```
// From T pick a set of properties K
declare function pick<T, K extends keyof T>(obj: T, ...keys: K[]): Pick<T, K>;
const nameAndAgeOnly = pick(person, "name", "age"); // { name: string, age: number }

// For every properties K of type T, transform it to U
function mapObject<K extends string, T, U>(
  obj: Record<K, T>,
  f: (x: T) => U
): Record<K, U>;
const names = { foo: "hello", bar: "world", baz: "bye" };
const lengths = mapObject(names, s => s.length); // { foo: number, bar: number, baz: number }
```

Object Spread and Rest

TypeScript 2.1 brings support for [ESnext Spread and Rest](#).

Similar to array spread, spreading an object can be handy to get a shallow copy:

```
let copy = { ...original };
```

Similarly, you can merge several different objects. In the following example, `merged` will have properties from `foo`, `bar`, and `baz`.

```
let merged = { ...foo, ...bar, ...baz };
```

You can also override existing properties and add new ones:

```
let obj = { x: 1, y: "string" };
var newObj = { ...obj, z: 3, y: 4 }; // { x: number, y: number, z: number }
```

The order of specifying spread operations determines what properties end up in the resulting object; properties in later spreads "win out" over previously created properties.

Object rests are the dual of object spreads, in that they can extract any extra properties that don't get picked up when destructuring an element:

```
let obj = { x: 1, y: 1, z: 1 };
let { z, ...obj1 } = obj;
obj1; // {x: number, y: number};
```

Downlevel Async Functions

This feature was supported before TypeScript 2.1, but only when targeting ES6/ES2015. TypeScript 2.1 brings the capability to ES3 and ES5 run-times, meaning you'll be free to take advantage of it no matter what environment you're using.

Note: first, we need to make sure our run-time has an ECMAScript-compliant `Promise` available globally. That might involve grabbing a polyfill for `Promise`, or relying on one that you might have in the run-time that you're targeting. We also need to make sure that TypeScript knows `Promise` exists by setting your `lib` option to something like `"dom"`, `"es2015"` or `"dom"`, `"es2015.promise"`, `"es5"`

Example

`tsconfig.json`

```
{
  "compilerOptions": {
    "lib": ["dom", "es2015.promise", "es5"]
  }
}
```

`dramaticWelcome.ts`

```
function delay(milliseconds: number) {
  return new Promise<void>(resolve => {
    setTimeout(resolve, milliseconds);
  });
}

async function dramaticWelcome() {
  console.log("Hello");

  for (let i = 0; i < 3; i++) {
    await delay(500);
    console.log(".");
  }

  console.log("World!");
}

dramaticWelcome();
```

Compiling and running the output should result in the correct behavior on an ES3/ES5 engine.

Support for external helpers library (`tslib`)

TypeScript injects a handful of helper functions such as `__extends` for inheritance, `__assign` for spread operator in object literals and JSX elements, and `__awaiter` for async functions.

Previously there were two options:

1. inject helpers in *every* file that needs them, or
2. no helpers at all with `noEmitHelpers`.

The two options left more to be desired; bundling the helpers in *every* file was a pain point for customers trying to keep their package size small. And not including helpers, meant customers had to maintain their own helpers library.

TypeScript 2.1 allows for including these files in your project once in a separate module, and the compiler will emit imports to them as needed.

First, install the `tslib` utility library:

```
npm install tslib
```

Second, compile your files using `importHelpers`:

```
tsc --module commonjs --importHelpers a.ts
```

So given the following input, the resulting `.js` file will include an import to `tslib` and use the `_assign` helper from it instead of inlining it.

```
export const o = { a: 1, name: "o" };
export const copy = { ...o };
```

```
"use strict";
var tslib_1 = require("tslib");
exports.o = { a: 1, name: "o" };
exports.copy = tslib_1._assign({}, exports.o);
```

Untyped imports

TypeScript has traditionally been overly strict about how you can import modules. This was to avoid typos and prevent users from using modules incorrectly.

However, a lot of the time, you might just want to import an existing module that may not have its own `.d.ts` file. Previously this was an error. Starting with TypeScript 2.1 this is now much easier.

With TypeScript 2.1, you can import a JavaScript module without needing a type declaration. A type declaration (such as `declare module "foo" { ... }` or `node_modules/@types/foo`) still takes priority if it exists.

An import to a module with no declaration file will still be flagged as an error under `noImplicitAny`.

Example

```
// Succeeds if `node_modules/asdf/index.js` exists
import { x } from "asdf";
```

Support for `--target ES2016`, `--target ES2017` and `--target ESNext`

TypeScript 2.1 supports three new target values `--target ES2016`, `--target ES2017` and `--target ESNext`.

Using target `--target ES2016` will instruct the compiler not to transform ES2016-specific features, e.g. `**` operator.

Similarly, `--target ES2017` will instruct the compiler not to transform ES2017-specific features like `async / await`.

`--target ESNext` targets latest supported [ES proposed features](#).

Improved `any` Inference

Previously, if TypeScript couldn't figure out the type of a variable, it would choose the `any` type.

```
let x; // implicitly 'any'
let y = []; // implicitly 'any[]'
let z: any; // explicitly 'any'.
```

With TypeScript 2.1, instead of just choosing `any`, TypeScript will infer types based on what you end up assigning later on.

This is only enabled if `noImplicitAny` is set.

Example

```
let x;

// You can still assign anything you want to 'x'.
x = () => 42;

// After that last assignment, TypeScript 2.1 knows that 'x' has type '() => number'.
let y = x();

// Thanks to that, it will now tell you that you can't add a number to a function!
console.log(x + y);
// ~~~~~
// Error! Operator '+' cannot be applied to types '() => number' and 'number'.

// TypeScript still allows you to assign anything you want to 'x'.
x = "Hello world!";

// But now it also knows that 'x' is a 'string'!
x.toLowerCase();
```

The same sort of tracking is now also done for empty arrays.

A variable declared with no type annotation and an initial value of `[]` is considered an implicit `any[]` variable. However, each subsequent `x.push(value)`, `x.unshift(value)` or `x[n] = value` operation evolves the type of the variable in accordance with what elements are added to it.

```
function f1() {
  let x = [];
  x.push(5);
  x[1] = "hello";
  x.unshift(true);
  return x; // (string | number | boolean)[]
}
```

```
function f2() {
  let x = null;
  if (cond()) {
    x = [];
    while (cond()) {
      x.push("hello");
    }
  }
  return x; // string[] | null
}
```

Implicit any errors

One great benefit of this is that you'll see *way fewer* implicit `any` errors when running with `noImplicitAny`. Implicit `any` errors are only reported when the compiler is unable to know the type of a variable without a type annotation.

Example

```
function f3() {
  let x = []; // Error: Variable 'x' implicitly has type 'any[]' in some locations
  where its type cannot be determined.
  x.push(5);
  function g() {
    x; // Error: Variable 'x' implicitly has an 'any[]' type.
  }
}
```

Better inference for literal types

String, numeric and boolean literal types (e.g. `"abc"`, `1`, and `true`) were previously inferred only in the presence of an explicit type annotation. Starting with TypeScript 2.1, literal types are *always* inferred for `const` variables and `readonly` properties.

The type inferred for a `const` variable or `readonly` property without a type annotation is the type of the literal initializer. The type inferred for a `let` variable, `var` variable, parameter, or non-`readonly` property with an initializer and no type annotation is the widened literal type of the initializer. Where the widened type for a string literal type is `string`, `number` for numeric literal types, `boolean` for `true` or `false` and the containing enum for enum literal types.

Example

```
const c1 = 1; // Type 1
const c2 = c1; // Type 1
const c3 = "abc"; // Type "abc"
const c4 = true; // Type true
const c5 = cond ? 1 : "abc"; // Type 1 | "abc"

let v1 = 1; // Type number
let v2 = c2; // Type number
let v3 = c3; // Type string
let v4 = c4; // Type boolean
let v5 = c5; // Type number | string
```

Literal type widening can be controlled through explicit type annotations. Specifically, when an expression of a literal type is inferred for a const location without a type annotation, that `const` variable gets a widening literal type inferred. But when a `const` location has an explicit literal type annotation, the `const` variable gets a non-widening literal type.

Example

```
const c1 = "hello"; // Widening type "hello"
let v1 = c1; // Type string

const c2: "hello" = "hello"; // Type "hello"
let v2 = c2; // Type "hello"
```

Use returned values from super calls as 'this'

In ES2015, constructors which return an object implicitly substitute the value of `this` for any callers of `super()`. As a result, it is necessary to capture any potential return value of `super()` and replace it with `this`. This change enables working with [Custom Elements](#), which takes advantage of this to initialize browser-allocated elements with user-written constructors.

Example

```
class Base {
  x: number;
  constructor() {
    // return a new object other than `this`
    return {
      x: 1
    };
  }
}

class Derived extends Base {
  constructor() {
    super();
    this.x = 2;
  }
}
```

Generates:

```
var Derived = (function(_super) {
  __extends(Derived, _super);
  function Derived() {
    var _this = _super.call(this) || this;
    _this.x = 2;
    return _this;
  }
  return Derived;
})(Base);
```

This change entails a break in the behavior of extending built-in classes like `Error`, `Array`, `Map`, etc.. Please see the [extending built-ins breaking change documentation](#) for more details.

Configuration inheritance

Often a project has multiple output targets, e.g. `ES5` and `ES2015`, debug and production or `CommonJS` and `System`; Just a few configuration options change between these two targets, and maintaining multiple `tsconfig.json` files can be a hassle.

TypeScript 2.1 supports inheriting configuration using `extends`, where:

- `extends` is a new top-level property in `tsconfig.json` (alongside `compilerOptions`, `files`, `include`, and `exclude`).
- The value of `extends` must be a string containing a path to another configuration file to inherit from.
- The configuration from the base file are loaded first, then overridden by those in the inheriting config file.
- Circularity between configuration files is not allowed.
- `files`, `include` and `exclude` from the inheriting config file *overwrite* those from the base config file.
- All relative paths found in the configuration file will be resolved relative to the configuration file they originated in.

Example

`configs/base.json`:

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

`tsconfig.json`:

```
{
  "extends": "./configs/base",
  "files": ["main.ts", "supplemental.ts"]
}
```

`tsconfig.nostrictnull.json`:

```
{
  "extends": "./tsconfig",
  "compilerOptions": {
    "strictNullChecks": false
  }
}
```

New `--alwaysStrict`

Invoking the compiler with `alwaysStrict` causes:

1. Parses all the code in strict mode.
2. Writes `"use strict";` directive atop every generated file.

Modules are parsed automatically in strict mode. The new flag is recommended for non-module code.

[Go to TOC](#)

Support for Mix-in classes

TypeScript 2.2 adds support for the ECMAScript 2015 mixin class pattern (see [MDN Mixin description](#) and ["Real" Mixins with JavaScript Classes](#) for more details) as well as rules for combining mixin construct signatures with regular construct signatures in intersection types.

First some terminology

A **mixin constructor type** refers to a type that has a single construct signature with a single rest argument of type `any[]` and an object-like return type. For example, given an object-like type `X`, `new (...args: any[]) => X` is a mixin constructor type with an instance type `X`.

A **mixin class** is a class declaration or expression that `extends` an expression of a type parameter type. The following rules apply to mixin class declarations:

- The type parameter type of the `extends` expression must be constrained to a mixin constructor type.
- The constructor of a mixin class (if any) must have a single rest parameter of type `any[]` and must use the spread operator to pass those parameters as arguments in a `super(...args)` call.

Given an expression `Base` of a parametric type `T` with a constraint `X`, a mixin class `class C extends Base {...}` is processed as if `Base` had type `X` and the resulting type is the intersection `typeof C & T`.

In other words, a mixin class is represented as an intersection between the mixin class constructor type and the parametric base class constructor type.

When obtaining the construct signatures of an intersection type that contains mixin constructor types, the mixin construct signatures are discarded and their instance types are mixed into the return types of the other construct signatures in the intersection type. For example, the intersection type `{ new(...args: any[]) => A } & { new(s: string) => B }` has a single construct signature `new(s: string) => A & B`.

Putting all of the above rules together in an example

```
class Point {
  constructor(public x: number, public y: number) {}
}

class Person {
  constructor(public name: string) {}
}

type Constructor<T> = new (...args: any[]) => T;

function Tagged<T extends Constructor<{}>>(Base: T) {
  return class extends Base {
    _tag: string;
    constructor(...args: any[]) {
      super(...args);
      this._tag = "";
    }
  };
}
```

```
const TaggedPoint = Tagged(Point);

let point = new TaggedPoint(10, 20);
point._tag = "hello";

class Customer extends Tagged(Person) {
    accountBalance: number;
}

let customer = new Customer("Joe");
customer._tag = "test";
customer.accountBalance = 0;
```

Mixin classes can constrain the types of classes they can mix into by specifying a construct signature return type in the constraint for the type parameter. For example, the following `WithLocation` function implements a subclass factory that adds a `getLocation` method to any class that satisfies the `Point` interface (i.e. that has `x` and `y` properties of type `number`).

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

const WithLocation = <T extends Constructor<Point>>(Base: T) =>
    class extends Base {
        getLocation(): [number, number] {
            return [this.x, this.y];
        }
    };
};
```

object type

TypeScript did not have a type that represents the non-primitive type, i.e. any thing that is not `number`, `string`, `boolean`, `symbol`, `null`, or `undefined`. Enter the new `object` type.

With `object` type, APIs like `Object.create` can be better represented. For example:

```
declare function create(o: object | null): void;

create({ prop: 0 }); // OK
create(null); // OK

create(42); // Error
create("string"); // Error
create(false); // Error
create(undefined); // Error
```

Support for `new.target`

The `new.target` meta-property is new syntax introduced in ES2015. When an instance of a constructor is created via `new`, the value of `new.target` is set to be a reference to the constructor function initially used to allocate the instance. If a function is called rather than constructed via `new`, `new.target` is set to `undefined`.

`new.target` comes in handy when `Object.setPrototypeOf` or `__proto__` needs to be set in a class constructor. One such use case is inheriting from `Error` in NodeJS v4 and higher.

Example

```
class CustomError extends Error {
  constructor(message?: string) {
    super(message); // 'Error' breaks prototype chain here
    Object.setPrototypeOf(this, new.target.prototype); // restore prototype chain
  }
}
```

This results in the generated JS

```
var CustomError = (function(_super) {
  __extends(CustomError, _super);
  function CustomError() {
    var _newTarget = this.constructor;
    var _this = _super.apply(this, arguments); // 'Error' breaks prototype chain
    here
    _this.__proto__ = _newTarget.prototype; // restore prototype chain
    return _this;
  }
  return CustomError;
})(Error);
```

`new.target` also comes in handy for writing constructable functions, for example:

```
function f() {
  if (new.target) {
    /* called via 'new' */
  }
}
```

Which translates to:

```
function f() {
  var _newTarget = this && this instanceof f ? this.constructor : void 0;
  if (_newTarget) {
    /* called via 'new' */
  }
}
```

Better checking for `null` / `undefined` in operands of expressions

TypeScript 2.2 improves checking of nullable operands in expressions. Specifically, these are now flagged as errors:

- If either operand of a `+` operator is nullable, and neither operand is of type `any` or `string`.
- If either operand of a `-`, `*`, `**`, `/`, `%`, `<<`, `>>`, `>>>`, `&`, `|`, or `^` operator is nullable.
- If either operand of a `<`, `>`, `<=`, `>=`, or `in` operator is nullable.
- If the right operand of an `instanceof` operator is nullable.

- If the operand of a `+`, `-`, `~`, `++`, or `--` unary operator is nullable.

An operand is considered nullable if the type of the operand is `null` or `undefined` or a union type that includes `null` or `undefined`. Note that the union type case only occurs in `strictNullChecks` mode because `null` and `undefined` disappear from unions in classic type checking mode.

Dotted property for types with string index signatures

Types with a string index signature can be indexed using the `[]` notation, but were not allowed to use the `..`. Starting with TypeScript 2.2 using either should be allowed.

```
interface StringMap<T> {
  [x: string]: T;
}

const map: StringMap<number>;
map["prop1"] = 1;
map.prop2 = 2;
```

This only applies to types with an *explicit* string index signature. It is still an error to access unknown properties on a type using `..` notation.

Support for spread operator on JSX element children

TypeScript 2.2 adds support for using spread on a JSX element children. Please see [facebook/jsx#57](#) for more details.

Example

```
function Todo(prop: { key: number; todo: string }) {
  return <div>{prop.key.toString() + prop.todo}</div>;
}

function TodoList({ todos }: TodoListProps) {
  return (
    <div>{...todos.map(todo => <Todo key={todo.id} todo={todo.todo} />)}</div>
  );
}

let x: TodoListProps;
<TodoList {...x} />;
```

New `jsx: react-native`

React-native build pipeline expects all files to have a `.js` extension even if the file contains JSX syntax. The new `jsx` value `react-native` will preserve the JSX syntax in the output file, but give it a `.js` extension.

[Go to TOC](#)

Generators and Iteration for ES5/ES3

First some ES2016 terminology:

Iterators

ES2015 introduced `Iterator`, which is an object that exposes three methods, `next`, `return`, and `throw`, as per the following interface:

```
interface Iterator<T> {
  next(value?: any): IteratorResult<T>;
  return?(value?: any): IteratorResult<T>;
  throw?(e?: any): IteratorResult<T>;
}
```

This kind of iterator is useful for iterating over synchronously available values, such as the elements of an Array or the keys of a Map. An object that supports iteration is said to be "iterable" if it has a `Symbol.iterator` method that returns an `Iterator` object.

The Iterator protocol also defines the target of some of the ES2015 features like `for..of` and spread operator and the array rest in destructuring assignments.

Generators

ES2015 also introduced "Generators", which are functions that can be used to yield partial computation results via the `Iterator` interface and the `yield` keyword. Generators can also internally delegate calls to another iterable through `yield *`. For example:

```
function* f() {
  yield 1;
  yield* [2, 3];
}
```

New `--downlevelIteration`

Previously generators were only supported if the target is ES6/ES2015 or later. Moreover, constructs that operate on the Iterator protocol, e.g. `for..of` were only supported if they operate on arrays for targets below ES6/ES2015.

TypeScript 2.3 adds full support for generators and the Iterator protocol for ES3 and ES5 targets with `downlevelIteration` flag.

With `downlevelIteration`, the compiler uses new type check and emit behavior that attempts to call a `[Symbol.iterator]()` method on the iterated object if it is found, and creates a synthetic array iterator over the object if it is not.

Please note that this requires a native `Symbol.iterator` or `Symbol.iterator` shim at runtime for any non-array values.

`for..of` statements, Array Destructuring, and Spread elements in Array, Call, and New expressions support `Symbol.iterator` in ES5/E3 if available when using `__downlevelIteration`, but can be used on an Array even if it does not define `Symbol.iterator` at run time or design time.

Async Iteration

TypeScript 2.3 adds support for the async iterators and generators as described by the current [TC39 proposal](#).

Async iterators

The Async Iteration introduces an `AsyncIterator`, which is similar to `Iterator`. The difference lies in the fact that the `next`, `return`, and `throw` methods of an `AsyncIterator` return a `Promise` for the iteration result, rather than the result itself. This allows the caller to enlist in an asynchronous notification for the time at which the `AsyncIterator` has advanced to the point of yielding a value. An `AsyncIterator` has the following shape:

```
interface AsyncIterator<T> {
  next(value?: any): Promise<IteratorResult<T>>;
  return?(value?: any): Promise<IteratorResult<T>>;
  throw?(e?: any): Promise<IteratorResult<T>>;
}
```

An object that supports async iteration is said to be "iterable" if it has a `Symbol.asyncIterator` method that returns an `AsyncIterator` object.

Async Generators

The [Async Iteration proposal](#) introduces "Async Generators", which are async functions that also can be used to yield partial computation results. Async Generators can also delegate calls via `yield*` to either an iterable or async iterable:

```
async function* g() {
  yield 1;
  await sleep(100);
  yield* [2, 3];
  yield* (async function*() {
    await sleep(100);
    yield 4;
  })();
}
```

As with Generators, Async Generators can only be function declarations, function expressions, or methods of classes or object literals. Arrow functions cannot be Async Generators. Async Generators require a valid, global `Promise` implementation (either native or an ES2015-compatible polyfill), in addition to a valid `Symbol.asyncIterator` reference (either a native symbol or a shim).

The `for-await-of` Statement

Finally, ES2015 introduced the `for..of` statement as a means of iterating over an iterable. Similarly, the Async Iteration proposal introduces the `for..await..of` statement to iterate over an async iterable:

```
async function f() {
  for await (const x of g()) {
    console.log(x);
  }
}
```

The `for..await..of` statement is only legal within an Async Function or Async Generator.

Caveats

- Keep in mind that our support for `async` iterators relies on support for `Symbol.asyncIterator` to exist at runtime. You may need to polyfill `Symbol.asyncIterator`, which for simple purposes can be as simple as: `(Symbol as any).asyncIterator = Symbol.asyncIterator || Symbol.for("Symbol.asyncIterator");`
- You also need to include `esnext` in your `lib` option, to get the `AsyncIterator` declaration if you do not already have it.
- Finally, if your target is ES5 or ES3, you'll also need to set the `--downlevelIterators` flag.

Generic parameter defaults

TypeScript 2.3 adds support for declaring defaults for generic type parameters.

Example

Consider a function that creates a new `HTMLElement`, calling it with no arguments generates a `Div`; you can optionally pass a list of children as well. Previously you would have to define it as:

```
declare function create(): Container<HTMLDivElement, HTMLDivElement[]>;
declare function create<T extends HTMLElement>(element: T): Container<T, T[]>;
declare function create<T extends HTMLElement, U extends HTMLElement>(
  element: T,
  children: U[]
): Container<T, U[]>;
```

With generic parameter defaults we can reduce it to:

```
declare function create<T extends HTMLElement = HTMLDivElement, U = T[]>(
  element?: T,
  children?: U
): Container<T, U>;
```

A generic parameter default follows the following rules:

- A type parameter is deemed optional if it has a default.
- Required type parameters must not follow optional type parameters.
- Default types for a type parameter must satisfy the constraint for the type parameter, if it exists.
- When specifying type arguments, you are only required to specify type arguments for the required type parameters. Unspecified type parameters will resolve to their default types.
- If a default type is specified and inference cannot choose a candidate, the default type is inferred.
- A class or interface declaration that merges with an existing class or interface declaration may introduce a default for an existing type parameter.

- A class or interface declaration that merges with an existing class or interface declaration may introduce a new type parameter as long as it specifies a default.

New `--strict` master option

New checks added to TypeScript are often off by default to avoid breaking existing projects. While avoiding breakage is a good thing, this strategy has the drawback of making it increasingly complex to choose the highest level of type safety, and doing so requires explicit opt-in action on every TypeScript release. With the `strict` option it becomes possible to choose maximum type safety with the understanding that additional errors might be reported by newer versions of the compiler as improved type checking features are added.

The new `strict` compiler option represents the recommended setting of a number of type checking options. Specifically, specifying `strict` corresponds to specifying all of the following options (and may in the future include more options):

- `strictNullChecks`
- `noImplicitAny`
- `noImplicitThis`
- `alwaysStrict`

In exact terms, the `strict` option sets the *default* value for the compiler options listed above. This means it is still possible to individually control the options. For example,

```
--strict --noImplicitThis false
```

has the effect of turning on all strict options *except* the `noImplicitThis` option. Using this scheme it is possible to express configurations consisting of *all* strict options except some explicitly listed options. In other words, it is now possible to default to the highest level of type safety but opt out of certain checks.

Starting with TypeScript 2.3, the default `tsconfig.json` generated by `tsc --init` includes a `"strict": true` setting in the `"compilerOptions"` section. Thus, new projects started with `tsc --init` will by default have the highest level of type safety enabled.

Enhanced `--init` output

Along with setting `strict` on by default, `tsc --init` has an enhanced output. Default `tsconfig.json` files generated by `tsc --init` now include a set of the common compiler options along with their descriptions commented out. Just un-comment the configuration you like to set to get the desired behavior; we hope the new output simplifies the setting up new projects and keeps configuration files readable as projects grow.

Errors in .js files with `--checkJs`

By default the TypeScript compiler does not report any errors in `.js` files including using `allowJs`. With TypeScript 2.3 type-checking errors can also be reported in `.js` files with `checkJs`.

You can skip checking some files by adding `// @ts-nocheck` comment to them; conversely you can choose to check only a few `.js` files by adding `// @ts-check` comment to them without setting `checkJs`. You can also ignore errors on specific lines by adding `// @ts-ignore` on the preceding line.

`.js` files are still checked to ensure that they only include standard ECMAScript features; type annotations are only allowed in `.ts` files and are flagged as errors in `.js` files. JSDoc comments can be used to add some type information to your JavaScript code, see [JSDoc Support documentation](#) for more details about the supported JSDoc constructs.

See [Type checking JavaScript Files documentation](#) for more details.

[Go to TOC](#)

Dynamic Import Expressions

Dynamic `import` expressions are a new feature and part of ECMAScript that allows users to asynchronously request a module at any arbitrary point in your program.

This means that you can conditionally and lazily import other modules and libraries. For example, here's an `async` function that only imports a utility library when it's needed:

```
async function getZipFile(name: string, files: File[]): Promise<File> {
  const zipUtil = await import("./utils/create-zip-file");
  const zipContents = await zipUtil.getContentAsBlob(files);
  return new File(zipContents, name);
}
```

Many bundlers have support for automatically splitting output bundles based on these `import` expressions, so consider using this new feature with the `esnext` module target.

String Enums

TypeScript 2.4 now allows enum members to contain string initializers.

```
enum Colors {
  Red = "RED",
  Green = "GREEN",
  Blue = "BLUE"
}
```

The caveat is that string-initialized enums can't be reverse-mapped to get the original enum member name. In other words, you can't write `Colors["RED"]` to get the string `"Red"`.

Improved inference for generics

TypeScript 2.4 introduces a few wonderful changes around the way generics are inferred.

Return types as inference targets

For one, TypeScript can now make inferences for the return type of a call. This can improve your experience and catch errors. Something that now works:

```
function arrayMap<T, U>(f: (x: T) => U): (a: T[]) => U[] {
  return a => a.map(f);
}

const lengths: (a: string[]) => number[] = arrayMap(s => s.length);
```

As an example of new errors you might spot as a result:

```
let x: Promise<string> = new Promise(resolve => {
  resolve(10);
  //      ~~ Error!
});
```

Type parameter inference from contextual types

Prior to TypeScript 2.4, in the following example

```
let f: <T>(x: T) => T = y => y;
```

`y` would have the type `any`. This meant the program would type-check, but you could technically do anything with `y`, such as the following:

```
let f: <T>(x: T) => T = y => y() + y.foo.bar;
```

That last example isn't actually type-safe.

In TypeScript 2.4, the function on the right side implicitly *gains* type parameters, and `y` is inferred to have the type of that type-parameter.

If you use `y` in a way that the type parameter's constraint doesn't support, you'll correctly get an error. In this case, the constraint of `T` was (implicitly) `{}`, so the last example will appropriately fail.

Stricter checking for generic functions

TypeScript now tries to unify type parameters when comparing two single-signature types. As a result, you'll get stricter checks when relating two generic signatures, and may catch some bugs.

```
type A = <T, U>(x: T, y: U) => [T, U];
type B = <S>(x: S, y: S) => [S, S];

function f(a: A, b: B) {
  a = b; // Error
  b = a; // Ok
}
```

Strict contravariance for callback parameters

TypeScript has always compared parameters in a bivariant way. There are a number of reasons for this, but by-and-large this was not been a huge issue for our users until we saw some of the adverse effects it had with `Promise`s and `Observable`s.

TypeScript 2.4 introduces tightens this up when relating two callback types. For example:

```
interface Mappable<T> {
  map<U>(f: (x: T) => U): Mappable<U>;
}

declare let a: Mappable<number>;
declare let b: Mappable<string | number>;

a = b;
b = a;
```

Prior to TypeScript 2.4, this example would succeed. When relating the types of `map`, TypeScript would bidirectionally relate their parameters (i.e. the type of `f`). When relating each `f`, TypeScript would also bidirectionally relate the type of *those* parameters.

When relating the type of `map` in TS 2.4, the language will check whether each parameter is a callback type, and if so, it will ensure that those parameters are checked in a contravariant manner with respect to the current relation.

In other words, TypeScript now catches the above bug, which may be a breaking change for some users, but will largely be helpful.

Weak Type Detection

TypeScript 2.4 introduces the concept of "weak types". Any type that contains nothing but a set of all-optional properties is considered to be *weak*. For example, this `Options` type is a weak type:

```
interface Options {
  data?: string;
  timeout?: number;
  maxRetries?: number;
}
```

In TypeScript 2.4, it's now an error to assign anything to a weak type when there's no overlap in properties. For example:

```
function sendMessage(options: Options) {
  // ...
}

const opts = {
  payload: "hello world!",
  retryOnFail: true
};

// Error!
sendMessage(opts);
// No overlap between the type of 'opts' and 'Options' itself.
// Maybe we meant to use 'data'/'maxRetries' instead of 'payload'/'retryOnFail'.
```

You can think of this as TypeScript "toughening up" the weak guarantees of these types to catch what would otherwise be silent bugs.

Since this is a breaking change, you may need to know about the workarounds which are the same as those for strict object literal checks:

1. Declare the properties if they really do exist.
2. Add an index signature to the weak type (i.e. `[propName: string]: {}`).
3. Use a type assertion (i.e. `opts as Options`).

[Go to TOC](#)

Optional `catch` clause variables

Thanks to work done by [@tinganho](#), TypeScript 2.5 implements a new ECMAScript feature that allows users to omit the variable in `catch` clauses. For example, when using `JSON.parse` you may need to wrap calls to the function with a `try / catch`, but you may not end up using the `SyntaxError` that gets thrown when input is erroneous.

```
let input = "...";
try {
  JSON.parse(input);
} catch {
  // ^ Notice that our `catch` clause doesn't declare a variable.
  console.log("Invalid JSON given\n\n" + input);
}
```

Type assertion/cast syntax in `checkJs` / `@ts-check` mode

TypeScript 2.5 introduces the ability to [assert the type of expressions when using plain JavaScript in your projects](#). The syntax is an `/** @type {...} */` annotation comment followed by a parenthesized expression whose type needs to be re-evaluated. For example:

```
var x = /** @type {SomeType} */ AnyParenthesizedExpression;
```

Deduplicated and redirected packages

When importing using the `Node` module resolution strategy in TypeScript 2.5, the compiler will now check whether files originate from "identical" packages. If a file originates from a package with a `package.json` containing the same `name` and `version` fields as a previously encountered package, then TypeScript will redirect itself to the top-most package. This helps resolve problems where two packages might contain identical declarations of classes, but which contain `private` members that cause them to be structurally incompatible.

As a nice bonus, this can also reduce the memory and runtime footprint of the compiler and language service by avoiding loading `.d.ts` files from duplicate packages.

The `--preserveSymlinks` compiler flag

TypeScript 2.5 brings the `preserveSymlinks` flag, which parallels the behavior of the `--preserve-symlinks` flag in `Node.js`. This flag also exhibits the opposite behavior to Webpack's `resolve.symlinks` option (i.e. setting TypeScript's `preserveSymlinks` to `true` parallels setting Webpack's `resolve.symlinks` to `false`, and vice-versa).

In this mode, references to modules and packages (e.g. `imports` and `/// <reference type="..." />` directives) are all resolved relative to the location of the symbolic link file, rather than relative to the path that the symbolic link resolves to. For a more concrete example, we'll defer to [the documentation on the Node.js website](#).

[Go to TOC](#)

Strict function types

TypeScript 2.6 introduces a new strict checking flag, `strictFunctionTypes`. The `strictFunctionTypes` switch is part of the `strict` family of switches, meaning that it defaults to on in `strict` mode. You can opt-out by setting `--strictFunctionTypes false` on your command line or in your `tsconfig.json`.

Under `strictFunctionTypes` function type parameter positions are checked *contravariantly* instead of *bivariantly*. For some background on what variance means for function types check out [What are covariance and contravariance?](#)

The stricter checking applies to all function types, *except* those originating in method or constructor declarations. Methods are excluded specifically to ensure generic classes and interfaces (such as `Array<T>`) continue to mostly relate covariantly.

Consider the following example in which `Animal` is the supertype of `Dog` and `Cat`:

```
declare let f1: (x: Animal) => void;
declare let f2: (x: Dog) => void;
declare let f3: (x: Cat) => void;
f1 = f2; // Error with --strictFunctionTypes
f2 = f1; // Ok
f2 = f3; // Error
```

The first assignment is permitted in default type checking mode, but flagged as an error in strict function types mode. Intuitively, the default mode permits the assignment because it is *possibly* sound, whereas strict function types mode makes it an error because it isn't *provable* sound. In either mode the third assignment is an error because it is *never* sound.

Another way to describe the example is that the type `(x: T) => void` is *bivariant* (i.e. covariant *or* contravariant) for `T` in default type checking mode, but *contravariant* for `T` in strict function types mode.

Example

```
interface Comparer<T> {
  compare: (a: T, b: T) => number;
}

declare let animalComparer: Comparer<Animal>;
declare let dogComparer: Comparer<Dog>;

animalComparer = dogComparer; // Error
dogComparer = animalComparer; // Ok
```

The first assignment is now an error. Effectively, `T` is contravariant in `Comparer<T>` because it is used only in function type parameter positions.

By the way, note that whereas some languages (e.g. C# and Scala) require variance annotations (`out / in` or `+ / -`), variance emerges naturally from the actual use of a type parameter within a generic type due to TypeScript's structural type system.

Note

Under `strictFunctionTypes` the first assignment is still permitted if `compare` was declared as a method. Effectively, `T` is bivariant in `Comparer<T>` because it is used only in method parameter positions.

```
interface Comparer<T> {
  compare(a: T, b: T): number;
}

declare let animalComparer: Comparer<Animal>;
declare let dogComparer: Comparer<Dog>;

animalComparer = dogComparer; // Ok because of bivariance
dogComparer = animalComparer; // Ok
```

TypeScript 2.6 also improves type inference involving contravariant positions:

```
function combine<T>(...funcs: ((x: T) => void)[]): (x: T) => void {
  return x => {
    for (const f of funcs) f(x);
  };
}

function animalFunc(x: Animal) {}
function dogFunc(x: Dog) {}

let combined = combine(animalFunc, dogFunc); // (x: Dog) => void
```

Above, all inferences for `T` originate in contravariant positions, and we therefore infer the *best common subtype* for `T`. This contrasts with inferences from covariant positions, where we infer the *best common supertype*.

Cache tagged template objects in modules

TypeScript 2.6 fixes the tagged string template emit to align better with the ECMAScript spec. As per the [ECMAScript spec](#), every time a template tag is evaluated, the *same* template strings object (the same `TemplateStringsArray`) should be passed as the first argument. Before TypeScript 2.6, the generated output was a completely new template object each time. Though the string contents are the same, this affects libraries that use the identity of the string for cache invalidation purposes, e.g. [lit-html](#).

Example

```
export function id(x: TemplateStringsArray) {
  return x;
}

export function templateObjectFactory() {
  return id`hello world`;
}

let result = templateObjectFactory() === templateObjectFactory(); // true in TS
2.6
```

Results in the following generated code:

```
"use strict";
var __makeTemplateObject =
  (this && this.__makeTemplateObject) ||
  function(cooked, raw) {
    if (Object.defineProperty) {
      Object.defineProperty(cooked, "raw", { value: raw });
    } else {
      cooked.raw = raw;
    }
    return cooked;
};

function id(x) {
  return x;
}

var _a;
function templateObjectFactory() {
  return id(
    _a || (_a = __makeTemplateObject(["hello world"], ["hello world"])))
}

var result = templateObjectFactory() === templateObjectFactory();
```

Note: This change brings a new emit helper, `__makeTemplateObject`; if you are using `importHelpers` with `tslib`, an updated to version 1.8 or later.

Localized diagnostics on the command line

TypeScript 2.6 npm package ships with localized versions of diagnostic messages for 13 languages. The localized messages are available when using `--locale` flag on the command line.

Example

Error messages in Russian:

```
c:\ts>tsc --v
Version 2.6.0-dev.20171003

c:\ts>tsc --locale ru --pretty c:\test\a.ts

./test/a.ts(1,5): error TS2322: Тип ""string"" не может быть назначен для типа
"number".

1 var x: number = "string";
~
```

And help in Japanese:

```
PS C:\ts> tsc --v
Version 2.6.0-dev.20171003

PS C:\ts> tsc --locale ja-jp
バージョン 2.6.0-dev.20171003
```

構文: tsc [オプション] [ファイル ...]

例: tsc hello.ts
 tsc --outFile file.js file.ts
 tsc @args.txt

オプション:

-h, --help	このメッセージを表示します。
--all	コンパイラ オプションをすべて表示します。
-v, --version	コンパイラのバージョンを表示します。
--init	TypeScript プロジェクトを初期化して、
tsconfig.json ファイルを作成します。	
-p ファイルまたはディレクトリ, --project ファイルまたはディレクトリ 構成ファイルか、 'tsconfig.json' を含むフォルダーにパスが指定されたプロジェクトをコ ンパイルします。	構成ファイルか、 'tsconfig.json' を含むフォルダーにパスが指定されたプロジェクトをコ ンパイルします。
--pretty	色とコンテキストを使用してエラーとメッセージに
スタイルを適用します (試験的)。	スタイルを適用します (試験的)。
-w, --watch	入力ファイルを監視します。
-t バージョン, --target バージョン	ECMAScript のターゲット バージョンを
指定します: 'ES3' (既定)、'ES5'、'ES2015'、'ES2016'、'ES2017'、'ES NEXT'。	指定します: 'ES3' (既定)、'ES5'、'ES2015'、'ES2016'、'ES2017'、'ES NEXT'。
-m 種類, --module 種類	モジュール コード生成を指定します: 'none'、
'commonjs'、'amd'、'system'、'umd'、'es2015'、'ESNext'。	'commonjs'、'amd'、'system'、'umd'、'es2015'、'ESNext'。
--lib	コンパイルに含めるライブラリ ファイルを指定しま す:
	'es5' 'es6' 'es2015' 'es7' 'es2016'
	'es2017' 'esnext' 'dom' 'dom.iterable' 'webworker' 'scripthost' 'es201 5.core' 'es2015.collection' 'es2015.generator' 'es2015.iterable' 'es2015.promise' 'es2015.proxy' 'es2015.reflect' 'es2015.symbol' 'es2015.symbol.wellkno wn' 'es2016.array.include' 'es2017.object' 'es2017.sharedmemory' 'es2017.string' 'es2017.intl' 'esnext.asynciterable'
--allowJs	javascript ファイルのコンパイルを許可します。
--jsx 種類	JSX コード生成を指定します: 'preserve'、
'react-native'、'react'。	'react-native'、'react'。
-d, --declaration	対応する '.d.ts' ファイルを生成します。
--sourceMap	対応する '.map' ファイルを生成します。
-- outFile ファイル	出力を連結して 1 つのファイルを生成します。
-- outDir ディレクトリ	ディレクトリへ出力構造をリダイレクトしま す。
--removeComments	コメントを出力しないでください。
--noEmit	出力しないでください。
--strict	strict 型チェックのオプションをすべて有効にし ます。
--noImplicitAny	暗黙的な 'any' 型を含む式と宣言に関するエラー を発生させます。
--strictNullChecks	厳格な null チェックを有効にします。
--noImplicitThis	暗黙的な 'any' 型を持つ 'this' 式でエラーが 発生します。
--alwaysStrict	厳格モードで解析してソース ファイルごとに "use strict" を生成します。
strict	strict
--noUnusedLocals	使用されていないローカルに関するエラーを報告し ます。
--noUnusedParameters	使用されていないパラメーターに関するエラーを報 告します。
--noImplicitReturns	関数の一部のコード パスが値を返さない場合にエ ラーを報告します。
--noFallthroughCasesInSwitch	switch ステートメントに case のフォールスル ーがある場合にエラーを報告します。
--types	コンパイルに含む型宣言ファイル。
@<ファイル>	

Suppress errors in .ts files using '>// @ts-ignore' comments

TypeScript 2.6 support suppressing errors in .js files using `// @ts-ignore` comments placed above the offending lines.

Example

```
if (false) {
  // @ts-ignore: Unreachable code error
  console.log("hello");
}
```

A `// @ts-ignore` comment suppresses all errors that originate on the following line. It is recommended practice to have the remainder of the comment following `@ts-ignore` explain which error is being suppressed.

Please note that this comment only suppresses the error reporting, and we recommend you use this comments *very sparingly*.

Faster tsc --watch

TypeScript 2.6 brings a faster `--watch` implementation. The new version optimizes code generation and checking for code bases using ES modules. Changes detected in a module file will result in *only* regenerating the changed module, and files that depend on it, instead of the whole project. Projects with large number of files should reap the most benefit from this change.

The new implementation also brings performance enhancements to watching in tsserver. The watcher logic has been completely rewritten to respond faster to change events.

Write-only references now flagged as unused

TypeScript 2.6 adds revised implementation the `noUnusedLocals` and `noUnusedParameters` compiler options. Declarations are only written to but never read from are now flagged as unused.

Example

Bellow both `n` and `m` will be marked as unused, because their values are never *read*. Previously TypeScript would only check whether their values were *referenced*.

```
function f(n: number) {
  n = 0;
}

class C {
  private m: number;
  constructor() {
    this.m = 0;
  }
}
```

Also functions that are only called within their own bodies are considered unused.

Example

```
function f() {  
    f(); // Error: 'f' is declared but its value is never read  
}
```

[Go to TOC](#)

Constant-named properties

TypeScript 2.7 adds support for declaring const-named properties on types including ECMAScript symbols.

Example

```
// Lib
export const SERIALIZE = Symbol("serialize-method-key");

export interface Serializable {
  [SERIALIZE](obj: {}): string;
}

// consumer

import { SERIALIZE, Serializable } from "lib";

class JSONSerializableItem implements Serializable {
  [SERIALIZE](obj: {}) {
    return JSON.stringify(obj);
  }
}
```

This also applies to numeric and string literals.

Example

```
const Foo = "Foo";
const Bar = "Bar";

let x = {
  [Foo]: 100,
  [Bar]: "hello"
};

let a = x[Bar]; // has type 'number'
let b = x[Bar]; // has type 'string'
```

unique symbol

To enable treating symbols as unique literals a new type `unique symbol` is available. `unique symbol` is a subtype of `symbol`, and are produced only from calling `Symbol()` or `Symbol.for()`, or from explicit type annotations. The new type is only allowed on `const` declarations and `readonly static` properties, and in order to reference a specific unique symbol, you'll have to use the `typeof` operator. Each reference to a `unique symbol` implies a completely unique identity that's tied to a given declaration.

Example

```
// Works
declare const Foo: unique symbol;

// Error! 'Bar' isn't a constant.
let Bar: unique symbol = Symbol();

// Works - refers to a unique symbol, but its identity is tied to 'Foo'.
```

```
let Baz: typeof Foo = Foo;
// Also works.
class C {
  static readonly StaticSymbol: unique symbol = Symbol();
```

Because each `unique symbol` has a completely separate identity, no two `unique symbol` types are assignable or comparable to each other.

Example

```
const Foo = Symbol();
const Bar = Symbol();

// Error: can't compare two unique symbols.
if (Foo === Bar) {
  // ...
}
```

Strict Class Initialization

TypeScript 2.7 introduces a new flag called `strictPropertyInitialization`. This flag performs checks to ensure that each instance property of a class gets initialized in the constructor body, or by a property initializer. For example

```
class C {
  foo: number;
  bar = "hello";
  baz: boolean;
  // ~~~
  // Error! Property 'baz' has no initializer and is not definitely assigned in
  // the
  //       constructor.
  constructor() {
    this.foo = 42;
  }
}
```

In the above, if we truly meant for `baz` to potentially be `undefined`, we should have declared it with the type `boolean | undefined`.

There are certain scenarios where properties can be initialized indirectly (perhaps by a helper method or dependency injection library), in which case you can use the new *definite assignment assertion modifiers* for your properties (discussed below).

```
class C {
  foo!: number;
  // ^
  // Notice this '!' modifier.
  // This is the "definite assignment assertion"

  constructor() {
    this.initialize();
  }
}
```

```
initialize() {
  this.foo = 0;
}
```

Keep in mind that `strictPropertyInitialization` will be turned on along with other `strict` mode flags, which can impact your project. You can set the `strictPropertyInitialization` setting to `false` in your `tsconfig.json`'s `compilerOptions`, or `--strictPropertyInitialization false` on the command line to turn off this checking.

Definite Assignment Assertions

The definite assignment assertion is a feature that allows a `!` to be placed after instance property and variable declarations to relay to TypeScript that a variable is indeed assigned for all intents and purposes, even if TypeScript's analyses cannot detect so.

Example

```
let x: number;
initialize();
console.log(x + x);
// ~~~
// Error! Variable 'x' is used before being assigned.

function initialize() {
  x = 10;
}
```

With definite assignment assertions, we can assert that `x` is really assigned by appending an `!` to its declaration:

```
// Notice the '!'
let x!: number;
initialize();

// No error!
console.log(x + x);

function initialize() {
  x = 10;
}
```

In a sense, the definite assignment assertion operator is the dual of the non-null assertion operator (in which *expressions* are post-fixed with a `!`), which we could also have used in the example.

```
let x: number;
initialize();

// No error!
console.log(x! + x!);

function initialize() {
  x = 10;
}
```

In our example, we knew that all uses of `x` would be initialized so it makes more sense to use definite assignment assertions than non-null assertions.

Fixed Length Tuples

In TypeScript 2.6 and earlier, `[number, string, string]` was considered a subtype of `[number, string]`. This was motivated by TypeScript's structural nature; the first and second elements of a `[number, string, string]` are respectively subtypes of the first and second elements of `[number, string]`. However, after examining real world usage of tuples, we noticed that most situations in which this was permitted was typically undesirable.

In TypeScript 2.7, tuples of different arities are no longer assignable to each other. Thanks to a pull request from [Kiara Grouwstra](#), tuple types now encode their arity into the type of their respective `length` property. This is accomplished by leveraging numeric literal types, which now allow tuples to be distinct from tuples of different arities.

Conceptually, you might consider the type `[number, string]` to be equivalent to the following declaration of `NumStrTuple`:

```
interface NumStrTuple extends Array<number | string> {
  0: number;
  1: string;
  length: 2; // using the numeric literal type '2'
}
```

Note that this is a breaking change for some code. If you need to resort to the original behavior in which tuples only enforce a minimum length, you can use a similar declaration that does not explicitly define a `length` property, falling back to `number`.

```
interface MinimumNumStrTuple extends Array<number | string> {
  0: number;
  1: string;
}
```

Note that this does not imply tuples represent immutable arrays, but it is an implied convention.

Improved type inference for object literals

TypeScript 2.7 improves type inference for multiple object literals occurring in the same context. When multiple object literal types contribute to a union type, we now *normalize* the object literal types such that all properties are present in each constituent of the union type.

Consider:

```
const obj = test ? { text: "hello" } : {}; // { text: string } | { text?: undefined }
const s = obj.text; // string | undefined
```

Previously type `{}` was inferred for `obj` and the second line subsequently caused an error because `obj` would appear to have no properties. That obviously wasn't ideal.

Example

```
// let obj: { a: number, b: number } |
//   { a: string, b?: undefined } |
//   { a?: undefined, b?: undefined }
let obj = [{ a: 1, b: 2 }, { a: "abc" }, {}][0];
obj.a; // string | number | undefined
obj.b; // number | undefined
```

Multiple object literal type inferences for the same type parameter are similarly collapsed into a single normalized union type:

```
declare function f<T>(...items: T[]): T;
// let obj: { a: number, b: number } |
//   { a: string, b?: undefined } |
//   { a?: undefined, b?: undefined }
let obj = f({ a: 1, b: 2 }, { a: "abc" }, {});
obj.a; // string | number | undefined
obj.b; // number | undefined
```

Improved handling of structurally identical classes and `instanceof` expressions

TypeScript 2.7 improves the handling of structurally identical classes in union types and `instanceof` expressions:

- Structurally identical, but distinct, class types are now preserved in union types (instead of eliminating all but one).
- Union type subtype reduction only removes a class type if it is a subclass of *and* derives from another class type in the union.
- Type checking of the `instanceof` operator is now based on whether the type of the left operand *derives from* the type indicated by the right operand (as opposed to a structural subtype check).

This means that union types and `instanceof` properly distinguish between structurally identical classes.

Example

```
class A {}
class B extends A {}
class C extends A {}
class D extends A {
  c: string;
}
class E extends D {}

let x1 = !true ? new A() : new B(); // A
let x2 = !true ? new B() : new C(); // B | C (previously B)
let x3 = !true ? new C() : new D(); // C | D (previously C)

let a1 = [new A(), new B(), new C(), new D(), new E()]; // A[]
let a2 = [new B(), new C(), new D(), new E()]; // (B | C | D)[] (previously B[])
```

```
function f1(x: B | C | D) {
  if (x instanceof B) {
    x; // B (previously B | D)
  } else if (x instanceof C) {
    x; // C
  } else {
    x; // D (previously never)
  }
}
```

Type guards inferred from `in` operator

The `in` operator now acts as a narrowing expression for types.

For a `n in x` expression, where `n` is a string literal or string literal type and `x` is a union type, the "true" branch narrows to types which have an optional or required property `n`, and the "false" branch narrows to types which have an optional or missing property `n`.

Example

```
interface A {
  a: number;
}
interface B {
  b: string;
}

function foo(x: A | B) {
  if ("a" in x) {
    return x.a;
  }
  return x.b;
}
```

Support for `import d from "cjs"` from CommonJS modules with `--esModuleInterop`

TypeScript 2.7 updates CommonJS/AMD/UMD module emit to synthesize namespace records based on the presence of an `_esModule` indicator under `esModuleInterop`. The change brings the generated output from TypeScript closer to that generated by Babel.

Previously CommonJS/AMD/UMD modules were treated in the same way as ES6 modules, resulting in a couple of problems. Namely:

- TypeScript treats a namespace import (i.e. `import * as foo from "foo"`) for a CommonJS/AMD/UMD module as equivalent to `const foo = require("foo")`. Things are simple here, but they don't work out if the primary object being imported is a primitive or a class or a function. ECMAScript spec stipulates that a namespace record is a plain object, and that a namespace import (`foo` in the example above) is not callable, though allowed by TypeScript

- Similarly a default import (i.e. `import d from "foo"`) for a CommonJS/AMD/UMD module as equivalent to `const d = require("foo").default`. Most of the CommonJS/AMD/UMD modules available today do not have a `default` export, making this import pattern practically unusable to import non-ES modules (i.e. CommonJS/AMD/UMD). For instance `import fs from "fs"` or `import express from "express"` are not allowed.

Under the new `esModuleInterop` these two issues should be addressed:

- A namespace import (i.e. `import * as foo from "foo"`) is now correctly flagged as uncallable. Calling it will result in an error.
- Default imports to CommonJS/AMD/UMD are now allowed (e.g. `import fs from "fs"`), and should work as expected.

Note: The new behavior is added under a flag to avoid unwarranted breaks to existing code bases.

We highly recommend applying it both to new and existing projects. For existing projects, namespace imports (`import * as express from "express"; express();`) will need to be converted to default imports (`import express from "express"; express();`).

Example

With `esModuleInterop` two new helpers are generated `__importStar` and `__importDefault` for import `*` and import `default` respectively. For instance input like:

```
import * as foo from "foo";
import b from "bar";
```

Will generate:

```
"use strict";
var __importStar =
  (this && this.__importStar) ||
  function(mod) {
    if (mod && mod.__esModule) return mod;
    var result = {};
    if (mod != null)
      for (var k in mod)
        if (Object.hasOwnProperty.call(mod, k)) result[k] = mod[k];
    result["default"] = mod;
    return result;
};
var __importDefault =
  (this && this.__importDefault) ||
  function(mod) {
    return mod && mod.__esModule ? mod : { default: mod };
};
exports.__esModule = true;
var foo = __importStar(require("foo"));
var bar_1 = __importDefault(require("bar"));
```

Numeric separators

TypeScript 2.7 brings support for [ES Numeric Separators](#). Numeric literals can now be separated into segments using `_`.

Example

```
const million = 1_000_000;
const phone = 555_734_2231;
const bytes = 0xff_0c_00_ff;
const word = 0b1100_0011_1101_0001;
```

Cleaner output in `--watch` mode

TypeScript's `--watch` mode now clears the screen after a re-compilation is requested.

Prettier `--pretty` output

TypeScript's `pretty` flag can make error messages easier to read and manage. `pretty` now uses colors for file names, diagnostic codes, and line numbers. File names and positions are now also formatted to allow navigation in common terminals (e.g. Visual Studio Code terminal).

[Go to TOC](#)

Conditional Types

TypeScript 2.8 introduces *conditional types* which add the ability to express non-uniform type mappings. A conditional type selects one of two possible types based on a condition expressed as a type relationship test:

```
T extends U ? X : Y
```

The type above means when `T` is assignable to `U` the type is `X`, otherwise the type is `Y`.

A conditional type `T extends U ? X : Y` is either *resolved* to `X` or `Y`, or *deferred* because the condition depends on one or more type variables. Whether to resolve or defer is determined as follows:

- First, given types `T'` and `U'` that are instantiations of `T` and `U` where all occurrences of type parameters are replaced with `any`, if `T'` is not assignable to `U'`, the conditional type is resolved to `Y`. Intuitively, if the most permissive instantiation of `T` is not assignable to the most permissive instantiation of `U`, we know that no instantiation will be and we can just resolve to `Y`.
- Next, for each type variable introduced by an `infer` (more later) declaration within `U` collect a set of candidate types by inferring from `T` to `U` (using the same inference algorithm as type inference for generic functions). For a given `infer` type variable `V`, if any candidates were inferred from co-variant positions, the type inferred for `V` is a union of those candidates. Otherwise, if any candidates were inferred from contra-variant positions, the type inferred for `V` is an intersection of those candidates. Otherwise, the type inferred for `V` is `never`.
- Then, given a type `T''` that is an instantiation of `T` where all `infer` type variables are replaced with the types inferred in the previous step, if `T''` is *definitely assignable* to `U`, the conditional type is resolved to `X`. The definitely assignable relation is the same as the regular assignable relation, except that type variable constraints are not considered. Intuitively, when a type is definitely assignable to another type, we know that it will be assignable for *all instantiations* of those types.
- Otherwise, the condition depends on one or more type variables and the conditional type is deferred.

Example

```
type TypeName<T> = T extends string
  ? "string"
  : T extends number
  ? "number"
  : T extends boolean
  ? "boolean"
  : T extends undefined
  ? "undefined"
  : T extends Function
  ? "function"
  : "object";
```

```
type T0 = TypeName<string>; // "string"
type T1 = TypeName<"a">; // "string"
type T2 = TypeName<true>; // "boolean"
type T3 = TypeName<() => void>; // "function"
type T4 = TypeName<string[]>; // "object"
```

Distributive conditional types

Conditional types in which the checked type is a naked type parameter are called *distributive conditional types*. Distributive conditional types are automatically distributed over union types during instantiation. For example, an instantiation of `T extends U ? X : Y` with the type argument `A | B | C` for `T` is resolved as `(A extends U ? X : Y) | (B extends U ? X : Y) | (C extends U ? X : Y)`.

Example

```
type T10 = TypeName<string | (( ) => void)>; // "string" | "function"
type T12 = TypeName<string | string[] | undefined>; // "string" | "object" | "undefined"
type T11 = TypeName<string[] | number[]>; // "object"
```

In instantiations of a distributive conditional type `T extends U ? X : Y`, references to `T` within the conditional type are resolved to individual constituents of the union type (i.e. `T` refers to the individual constituents *after* the conditional type is distributed over the union type). Furthermore, references to `T` within `X` have an additional type parameter constraint `U` (i.e. `T` is considered assignable to `U` within `X`).

Example

```
type BoxedValue<T> = { value: T };
type BoxedArray<T> = { array: T[] };
type Boxed<T> = T extends any[] ? BoxedArray<T[number]> : BoxedValue<T>;
type T20 = Boxed<string>; // BoxedValue<string>;
type T21 = Boxed<number[]>; // BoxedArray<number>;
type T22 = Boxed<string | number[]>; // BoxedValue<string> | BoxedArray<number>;
```

Notice that `T` has the additional constraint `any[]` within the true branch of `Boxed<T>` and it is therefore possible to refer to the element type of the array as `T[number]`. Also, notice how the conditional type is distributed over the union type in the last example.

The distributable property of conditional types can conveniently be used to *filter* union types:

```
type Diff<T, U> = T extends U ? never : T; // Remove types from T that are assignable to U
type Filter<T, U> = T extends U ? T : never; // Remove types from T that are not assignable to U

type T30 = Diff<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "b" | "d"
type T31 = Filter<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "a" | "c"
type T32 = Diff<string | number | (( ) => void), Function>; // string | number
type T33 = Filter<string | number | (( ) => void), Function>; // () => void

type NonNullable<T> = Diff<T, null | undefined>; // Remove null and undefined from T

type T34 = NonNullable<string | number | undefined>; // string | number
type T35 = NonNullable<string | string[] | null | undefined>; // string | string[]

function f1<T>(x: T, y: NonNullable<T>) {
  x = y; // Ok
  y = x; // Error
}
```

```
function f2<T extends string | undefined>(x: T, y: NonNullable<T>) {
  x = y; // Ok
  y = x; // Error
  let s1: string = x; // Error
  let s2: string = y; // Ok
}
```

Conditional types are particularly useful when combined with mapped types:

```
type FunctionPropertyNames<T> = {
  [K in keyof T]: T[K] extends Function ? K : never;
}[keyof T];
type FunctionProperties<T> = Pick<T, FunctionPropertyNames<T>>;

type NonFunctionPropertyNames<T> = {
  [K in keyof T]: T[K] extends Function ? never : K;
}[keyof T];
type NonFunctionProperties<T> = Pick<T, NonFunctionPropertyNames<T>>;

interface Part {
  id: number;
  name: string;
  subparts: Part[];
  updatePart(newName: string): void;
}

type T40 = FunctionPropertyNames<Part>; // "updatePart"
type T41 = NonFunctionPropertyNames<Part>; // "id" | "name" | "subparts"
type T42 = FunctionProperties<Part>; // { updatePart(newName: string): void }
type T43 = NonFunctionProperties<Part>; // { id: number, name: string, subparts:
Part[] }
```

Similar to union and intersection types, conditional types are not permitted to reference themselves recursively. For example the following is an error.

Example

```
type ElementType<T> = T extends any[] ? ElementType<T[number]> : T; // Error
```

Type inference in conditional types

Within the `extends` clause of a conditional type, it is now possible to have `infer` declarations that introduce a type variable to be inferred. Such inferred type variables may be referenced in the true branch of the conditional type. It is possible to have multiple `infer` locations for the same type variable.

For example, the following extracts the return type of a function type:

```
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;
```

Conditional types can be nested to form a sequence of pattern matches that are evaluated in order:

```
type Unpacked<T> = T extends (infer U)[]?
  U
: T extends (...args: any[]) => infer U
? U
: T extends Promise<infer U>
```

```
? U
: T;

type T0 = Unpacked<string>; // string
type T1 = Unpacked<string[]>; // string
type T2 = Unpacked<() => string>; // string
type T3 = Unpacked<Promise<string>>; // string
type T4 = Unpacked<Promise<string>[]>; // Promise<string>
type T5 = Unpacked<Unpacked<Promise<string>[]>>; // string
```

The following example demonstrates how multiple candidates for the same type variable in co-variant positions causes a union type to be inferred:

```
type Foo<T> = T extends { a: infer U; b: infer U } ? U : never;
type T10 = Foo<{ a: string; b: string }>; // string
type T11 = Foo<{ a: string; b: number }>; // string | number
```

Likewise, multiple candidates for the same type variable in contra-variant positions causes an intersection type to be inferred:

```
type Bar<T> = T extends { a: (x: infer U) => void; b: (x: infer U) => void }
? U
: never;
type T20 = Bar<{ a: (x: string) => void; b: (x: string) => void }>; // string
type T21 = Bar<{ a: (x: string) => void; b: (x: number) => void }>; // string &
number
```

When inferring from a type with multiple call signatures (such as the type of an overloaded function), inferences are made from the *last* signature (which, presumably, is the most permissive catch-all case). It is not possible to perform overload resolution based on a list of argument types.

```
declare function foo(x: string): number;
declare function foo(x: number): string;
declare function foo(x: string | number): string | number;
type T30 = ReturnType<typeof foo>; // string | number
```

It is not possible to use `infer` declarations in constraint clauses for regular type parameters:

```
type ReturnType<T extends (...args: any[]) => infer R> = R; // Error, not
supported
```

However, much the same effect can be obtained by erasing the type variables in the constraint and instead specifying a conditional type:

```
type AnyFunction = (...args: any[]) => any;
type ReturnType<T extends AnyFunction> = T extends (...args: any[]) => infer R
? R
: any;
```

Predefined conditional types

TypeScript 2.8 adds several predefined conditional types to `lib.d.ts`:

- `Exclude<T, U>` -- Exclude from `T` those types that are assignable to `U`.

- `Extract<T, U>` -- Extract from `T` those types that are assignable to `U`.
- `NonNullable<T>` -- Exclude `null` and `undefined` from `T`.
- `ReturnType<T>` -- Obtain the return type of a function type.
- `InstanceType<T>` -- Obtain the instance type of a constructor function type.

Example

```

type T00 = Exclude<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "b" | "d"
type T01 = Extract<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "a" | "c"

type T02 = Exclude<string | number | () => void, Function>; // string | number
type T03 = Extract<string | number | () => void, Function>; // () => void

type T04 = NonNullable<string | number | undefined>; // string | number
type T05 = NonNullable<() => string | string[] | null | undefined>; // () => string | string[]

function f1(s: string) {
  return { a: 1, b: s };
}

class C {
  x = 0;
  y = 0;
}

type T10 = ReturnType<() => string>; // string
type T11 = ReturnType<s: string => void>; // void
type T12 = ReturnType<<T>() => T>; // {}
type T13 = ReturnType<<T extends U, U extends number[]>() => T>; // number[]
type T14 = ReturnType<typeof f1>; // { a: number, b: string }
type T15 = ReturnType<any>; // any
type T16 = ReturnType<never>; // any
type T17 = ReturnType<string>; // Error
type T18 = ReturnType<Function>; // Error

type T20 = InstanceType<typeof C>; // C
type T21 = InstanceType<any>; // any
type T22 = InstanceType<never>; // any
type T23 = InstanceType<string>; // Error
type T24 = InstanceType<Function>; // Error

```

Note: The `Exclude` type is a proper implementation of the `Diff` type suggested [here](#). We've used the name `Exclude` to avoid breaking existing code that defines a `Diff`, plus we feel that name better conveys the semantics of the type. We did not include the `Omit<T, K>` type because it is trivially written as `Pick<T, Exclude<keyof T, K>>`.

Improved control over mapped type modifiers

Mapped types support adding a `readonly` or `?` modifier to a mapped property, but they did not provide support the ability to *remove* modifiers. This matters in [homomorphic mapped types](#) which by default preserve the modifiers of the underlying type.

TypeScript 2.8 adds the ability for a mapped type to either add or remove a particular modifier. Specifically, a `readonly` or `?` property modifier in a mapped type can now be prefixed with either `+` or `-` to indicate that the modifier should be added or removed.

Example

```
type MutableRequired<T> = { -readonly [P in keyof T]-?: T[P] }; // Remove readonly and ?
type ReadonlyPartial<T> = { +readonly [P in keyof T]+?: T[P] }; // Add readonly and ?
```

A modifier with no `+` or `-` prefix is the same as a modifier with a `+` prefix. So, the `ReadonlyPartial<T>` type above corresponds to

```
type ReadonlyPartial<T> = { readonly [P in keyof T]?: T[P] }; // Add readonly and ?
```

Using this ability, `lib.d.ts` now has a new `Required<T>` type. This type strips `?` modifiers from all properties of `T`, thus making all properties required.

Example

```
type Required<T> = { [P in keyof T]-?: T[P] };
```

Note that in `strictNullChecks` mode, when a homomorphic mapped type removes a `?` modifier from a property in the underlying type it also removes `undefined` from the type of that property:

Example

```
type Foo = { a?: string }; // Same as { a?: string | undefined }
type Bar = Required<Foo>; // Same as { a: string }
```

Improved `keyof` with intersection types

With TypeScript 2.8 `keyof` applied to an intersection type is transformed to a union of `keyof` applied to each intersection constituent. In other words, types of the form `keyof (A & B)` are transformed to be `keyof A | keyof B`. This change should address inconsistencies with inference from `keyof` expressions.

Example

```
type A = { a: string };
type B = { b: string };

type T1 = keyof (A & B); // "a" | "b"
type T2<T> = keyof (T & B); // keyof T | "b"
type T3<U> = keyof (A & U); // "a" | keyof U
type T4<T, U> = keyof (T & U); // keyof T | keyof U
type T5 = T2<A>; // "a" | "b"
type T6 = T3<B>; // "a" | "b"
type T7 = T4<A, B>; // "a" | "b"
```

Better handling for namespace patterns in `.js` files

TypeScript 2.8 adds support for understanding more namespace patterns in `.js` files. Empty object literals declarations on top level, just like functions and classes, are now recognized as as namespace declarations in JavaScript.

```
var ns = {};  
ns.constant = 1;
```

Assignments at the top-level should behave the same way; in other words, a `var` or `const` declaration is not required.

```
app = {};  
app.C = class {};  
app.f = function() {};  
app.prop = 1;
```

IIFEs as namespace declarations

An IIFE returning a function, class or empty object literal, is also recognized as a namespace:

```
var C = (function() {  
    function C(n) {  
        this.p = n;  
    }  
    return C;  
})();  
C.staticProperty = 1;
```

Defaulted declarations

"Defaulted declarations" allow initializers that reference the declared name in the left side of a logical or:

```
my = window.my || {};  
my.app = my.app || {};
```

Prototype assignment

You can assign an object literal directly to the prototype property. Individual prototype assignments still work too:

```
var C = function(p) {  
    this.p = p;  
};  
C.prototype = {  
    m() {  
        console.log(this.p);  
    }  
};  
C.prototype.q = function(r) {  
    return this.p === r;  
};
```

Nested and merged declarations

Nesting works to any level now, and merges correctly across files. Previously neither was the case.

```
var app = window.app || {};
app.C = class {};
```

Per-file JSX factories

TypeScript 2.8 adds support for a per-file configurable JSX factory name using `@jsx dom` pragma. JSX factory can be configured for a compilation using `jsxFactory` (default is `React.createElement`). With TypeScript 2.8 you can override this on a per-file-basis by adding a comment to the beginning of the file.

Example

```
/** @jsx dom */
import { dom } from "./renderer";
<h></h>;
```

Generates:

```
var renderer_1 = require("./renderer");
renderer_1.dom("h", null);
```

Locally scoped JSX namespaces

JSX type checking is driven by definitions in a JSX namespace, for instance `JSX.Element` for the type of a JSX element, and `JSX.IntrinsicElements` for built-in elements. Before TypeScript 2.8 the `JSX` namespace was expected to be in the global namespace, and thus only allowing one to be defined in a project. Starting with TypeScript 2.8 the `JSX` namespace will be looked under the `jsxNamespace` (e.g. `React`) allowing for multiple jsx factories in one compilation. For backward compatibility the global `JSX` namespace is used as a fallback if none was defined on the factory function. Combined with the per-file `@jsx` pragma, each file can have a different JSX factory.

New `--emitDeclarationOnly`

`emitDeclarationOnly` allows for *only* generating declaration files; `.js/.jsx` output generation will be skipped with this flag. The flag is useful when the `.js` output generation is handled by a different transpiler like Babel.

[Go to TOC](#)

Support `number` and `symbol` named properties with `keyof` and mapped types

TypeScript 2.9 adds support for `number` and `symbol` named properties in index types and mapped types. Previously, the `keyof` operator and mapped types only supported `string` named properties.

Changes include:

- An index type `keyof T` for some type `T` is a subtype of `string | number | symbol`.
- A mapped type `{ [P in K]: XXX }` permits any `K` assignable to `string | number | symbol`.
- In a `for...in` statement for an object of a generic type `T`, the inferred type of the iteration variable was previously `keyof T` but is now `Extract<keyof T, string>`. (In other words, the subset of `keyof T` that includes only string-like values.)

Given an object type `X`, `keyof X` is resolved as follows:

- If `X` contains a string index signature, `keyof X` is a union of `string`, `number`, and the literal types representing symbol-like properties, otherwise
- If `X` contains a numeric index signature, `keyof X` is a union of `number` and the literal types representing string-like and symbol-like properties, otherwise
- `keyof X` is a union of the literal types representing string-like, number-like, and symbol-like properties.

Where:

- String-like properties of an object type are those declared using an identifier, a string literal, or a computed property name of a string literal type.
- Number-like properties of an object type are those declared using a numeric literal or computed property name of a numeric literal type.
- Symbol-like properties of an object type are those declared using a computed property name of a unique symbol type.

In a mapped type `{ [P in K]: XXX }`, each string literal type in `K` introduces a property with a string name, each numeric literal type in `K` introduces a property with a numeric name, and each unique symbol type in `K` introduces a property with a unique symbol name. Furthermore, if `K` includes type `string`, a string index signature is introduced, and if `K` includes type `number`, a numeric index signature is introduced.

Example

```
const c = "c";
const d = 10;
const e = Symbol();

const enum E1 {
  A,
  B,
  C,
}
const enum E2 {
```

```

A = "A",
B = "B",
C = "C",
}

type Foo = {
  a: string; // String-like name
  5: string; // Number-like name
  [c]: string; // String-like name
  [d]: string; // Number-like name
  [e]: string; // Symbol-like name
  [E1.A]: string; // Number-like name
  [E2.A]: string; // String-like name
};

type K1 = keyof Foo; // "a" | 5 | "c" | 10 | typeof e | E1.A | E2.A
type K2 = Extract<keyof Foo, string>; // "a" | "c" | E2.A
type K3 = Extract<keyof Foo, number>; // 5 | 10 | E1.A
type K4 = Extract<keyof Foo, symbol>; // typeof e

```

Since `keyof` now reflects the presence of a numeric index signature by including type `number` in the key type, mapped types such as `Partial<T>` and `Readonly<T>` work correctly when applied to object types with numeric index signatures:

```

type Arrayish<T> = {
  length: number;
  [x: number]: T;
};

type ReadonlyArrayish<T> = Readonly<Arrayish<T>>;

declare const map: ReadonlyArrayish<string>;
let n = map.length;
let x = map[123]; // Previously of type any (or an error with --noImplicitAny)

```

Furthermore, with the `keyof` operator's support for `number` and `symbol` named keys, it is now possible to abstract over access to properties of objects that are indexed by numeric literals (such as numeric enum types) and unique symbols.

```

const enum Enum {
  A,
  B,
  C,
}

const enumToStringMap = {
  [Enum.A]: "Name A",
  [Enum.B]: "Name B",
  [Enum.C]: "Name C",
};

const sym1 = Symbol();
const sym2 = Symbol();
const sym3 = Symbol();

const symbolToNumberMap = {
  [sym1]: 1,
  [sym2]: 2,
  [sym3]: 3,
}

```

```

};

type KE = keyof typeof enumToStringMap; // Enum (i.e. Enum.A | Enum.B | Enum.C)
type KS = keyof typeof symbolToNumberMap; // typeof sym1 | typeof sym2 | typeof
sym3

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

let x1 = getValue(enumToStringMap, Enum.C); // Returns "Name C"
let x2 = getValue(symbolToNumberMap, sym3); // Returns 3

```

This is a breaking change; previously, the `keyof` operator and mapped types only supported `string` named properties. Code that assumed values typed with `keyof T` were always `strings`, will now be flagged as error.

Example

```

function useKey<T, K extends keyof T>(o: T, k: K) {
    var name: string = k; // Error: keyof T is not assignable to string
}

```

Recommendations

- If your functions are only able to handle string named property keys, use `Extract<keyof T, string>` in the declaration:

```

function useKey<T, K extends Extract<keyof T, string>>(o: T, k: K) {
    var name: string = k; // Ok
}

```

- If your functions are open to handling all property keys, then the changes should be done down-stream:

```

function useKey<T, K extends keyof T>(o: T, k: K) {
    var name: string | number | symbol = k;
}

```

- Otherwise use `keyofStringsOnly` compiler option to disable the new behavior.

Generic type arguments in JSX elements

JSX elements now allow passing type arguments to generic components.

Example

```

class GenericComponent<P> extends React.Component<P> {
    internalProp: P;
}

type Props = { a: number; b: string };

const x = <GenericComponent<Props> a={10} b="hi" />; // OK
const y = <GenericComponent<Props> a={10} b={20} />; // Error

```

Generic type arguments in generic tagged templates

Tagged templates are a form of invocation introduced in ECMAScript 2015. Like call expressions, generic functions may be used in a tagged template and TypeScript will infer the type arguments utilized.

TypeScript 2.9 allows passing generic type arguments to tagged template strings.

Example

```
declare function styledComponent<Props>(
  strs: TemplateStringsArray
): Component<Props>;

interface MyProps {
  name: string;
  age: number;
}

styledComponent<MyProps>`  

  font-size: 1.5em;  

  text-align: center;  

  color: palevioletred;  

`;  
  

declare function tag<T>(strs: TemplateStringsArray, ...args: T[]): T;  
  

// inference fails because 'number' and 'string' are both candidates that conflict
let a = tag<string | number>`${100} ${"hello"}`;
```

import types

Modules can import types declared in other modules. But non-module global scripts cannot access types declared in modules. Enter `import` types.

Using `import("mod")` in a type annotation allows for reaching in a module and accessing its exported declaration without importing it.

Example

Given a declaration of a class `Pet` in a module file:

```
// module.d.ts  
  
export declare class Pet {  
  name: string;  
}
```

Can be used in a non-module file `global-script.ts`:

```
// global-script.ts  
  
function adopt(p: import("./module").Pet) {  
  console.log(`Adopting ${p.name}...`);  
}
```

This also works in JSDoc comments to refer to types from other modules in `.js`:

```
// a.js
/**
 * @param p { import("./module").Pet }
 */
function walk(p) {
  console.log(`Walking ${p.name}...`);
}
```

Relaxing declaration emit visibility rules

With `import` types available, many of the visibility errors reported during declaration file generation can be handled by the compiler without the need to change the input.

For instance:

```
import { createHash } from "crypto";
export const hash = createHash("sha256");
// ^^^^
// Exported variable 'hash' has or is using name 'Hash' from external module
// "crypto" but cannot be named.
```

With TypeScript 2.9, no errors are reported, and now the generated file looks like:

```
export declare const hash: import("crypto").Hash;
```

Support for `import.meta`

TypeScript 2.9 introduces support for `import.meta`, a new meta-property as described by the current [TC39 proposal](#).

The type of `import.meta` is the global `ImportMeta` type which is defined in `lib.es5.d.ts`. This interface is extremely limited. Adding well-known properties for Node or browsers requires interface merging and possibly a global augmentation depending on the context.

Example

Assuming that `__dirname` is always available on `import.meta`, the declaration would be done through re-opening `ImportMeta` interface:

```
// node.d.ts
interface ImportMeta {
  __dirname: string;
}
```

And usage would be:

```
import.meta.__dirname; // Has type 'string'
```

`import.meta` is only allowed when targeting `ESNext` modules and ECMAScript targets.

New `--resolveJsonModule`

Often in Node.js applications a `.json` is needed. With TypeScript 2.9, `resolveJsonModule` allows for importing, extracting types from and generating `.json` files.

Example

```
// settings.json
{
  "repo": "TypeScript",
  "dry": false,
  "debug": false
}

// a.ts
import settings from "./settings.json";
settings.debug === true; // OK
settings.dry === 2; // Error: Operator '===' cannot be applied boolean and number

// tsconfig.json
{
  "compilerOptions": {
    "module": "commonjs",
    "resolveJsonModule": true,
    "esModuleInterop": true
  }
}
```

`--pretty` output by default

Starting TypeScript 2.9 errors are displayed under `pretty` by default if the output device is applicable for colorful text. TypeScript will check if the output stream has `isTty` property set.

Use `--pretty false` on the command line or set `"pretty": false` in your `tsconfig.json` to disable `pretty` output.

New `--declarationMap`

Enabling `declarationMap` alongside `declaration` causes the compiler to emit `.d.ts.map` files alongside the output `.d.ts` files. Language Services can also now understand these map files, and uses them to map declaration-file based definition locations to their original source, when available.

In other words, hitting go-to-definition on a declaration from a `.d.ts` file generated with `declarationMap` will take you to the source file (`.ts`) location where that declaration was defined, and not to the `.d.ts`.

[Go to TOC](#)

Project References

TypeScript 3.0 introduces a new concept of project references. Project references allow TypeScript projects to depend on other TypeScript projects - specifically, allowing `tsconfig.json` files to reference other `tsconfig.json` files. Specifying these dependencies makes it easier to split your code into smaller projects, since it gives TypeScript (and tools around it) a way to understand build ordering and output structure.

TypeScript 3.0 also introduces a new mode for tsc, the `--build` flag, that works hand-in-hand with project references to enable faster TypeScript builds.

See [Project References handbook page](#) for more documentation.

Tuples in rest parameters and spread expressions

TypeScript 3.0 adds support to multiple new capabilities to interact with function parameter lists as tuple types. TypeScript 3.0 adds support for:

- Expansion of rest parameters with tuple types into discrete parameters.
- Expansion of spread expressions with tuple types into discrete arguments.
- Generic rest parameters and corresponding inference of tuple types.
- Optional elements in tuple types.
- Rest elements in tuple types.

With these features it becomes possible to strongly type a number of higher-order functions that transform functions and their parameter lists.

Rest parameters with tuple types

When a rest parameter has a tuple type, the tuple type is expanded into a sequence of discrete parameters. For example the following two declarations are equivalent:

```
declare function foo(...args: [number, string, boolean]): void;
declare function foo(args_0: number, args_1: string, args_2: boolean): void;
```

Spread expressions with tuple types

When a function call includes a spread expression of a tuple type as the last argument, the spread expression corresponds to a sequence of discrete arguments of the tuple element types.

Thus, the following calls are equivalent:

```
const args: [number, string, boolean] = [42, "hello", true];
foo(42, "hello", true);
foo(args[0], args[1], args[2]);
foo(...args);
```

Generic rest parameters

A rest parameter is permitted to have a generic type that is constrained to an array type, and type inference can infer tuple types for such generic rest parameters. This enables higher-order capturing and spreading of partial parameter lists:

Example

```
declare function bind<T, U extends any[], V>(
  f: (x: T, ...args: U) => V,
  x: T
): (...args: U) => V;

declare function f3(x: number, y: string, z: boolean): void;

const f2 = bind(f3, 42); // (y: string, z: boolean) => void
const f1 = bind(f2, "hello"); // (z: boolean) => void
const f0 = bind(f1, true); // () => void

f3(42, "hello", true);
f2("hello", true);
f1(true);
f0();
```

In the declaration of `f2` above, type inference infers types `number`, `[string, boolean]` and `void` for `T`, `U` and `V` respectively.

Note that when a tuple type is inferred from a sequence of parameters and later expanded into a parameter list, as is the case for `U`, the original parameter names are used in the expansion (however, the names have no semantic meaning and are not otherwise observable).

Optional elements in tuple types

Tuple types now permit a `?` postfix on element types to indicate that the element is optional:

Example

```
let t: [number, string?, boolean?];
t = [42, "hello", true];
t = [42, "hello"];
t = [42];
```

In `strictNullChecks` mode, a `?` modifier automatically includes `undefined` in the element type, similar to optional parameters.

A tuple type permits an element to be omitted if it has a postfix `?` modifier on its type and all elements to the right of it also have `?` modifiers.

When tuple types are inferred for rest parameters, optional parameters in the source become optional tuple elements in the inferred type.

The `length` property of a tuple type with optional elements is a union of numeric literal types representing the possible lengths. For example, the type of the `length` property in the tuple type `[number, string?, boolean?]` is `1 | 2 | 3`.

Rest elements in tuple types

The last element of a tuple type can be a rest element of the form `...x`, where `x` is an array type. A rest element indicates that the tuple type is open-ended and may have zero or more additional elements of the array element type. For example, `[number, ...string[]]` means tuples with a `number` element followed by any number of `string` elements.

Example

```
function tuple<T extends any[]>(...args: T): T {
    return args;
}

const numbers: number[] = getArrayOfNumbers();
const t1 = tuple("foo", 1, true); // [string, number, boolean]
const t2 = tuple("bar", ...numbers); // [string, ...number[]]
```

The type of the `length` property of a tuple type with a rest element is `number`.

New `unknown` top type

TypeScript 3.0 introduces a new top type `unknown`. `unknown` is the type-safe counterpart of `any`. Anything is assignable to `unknown`, but `unknown` isn't assignable to anything but itself and `any` without a type assertion or a control flow based narrowing. Likewise, no operations are permitted on an `unknown` without first asserting or narrowing to a more specific type.

Example

```
// In an intersection everything absorbs unknown

type T00 = unknown & null; // null
type T01 = unknown & undefined; // undefined
type T02 = unknown & null & undefined; // null & undefined (which becomes never)
type T03 = unknown & string; // string
type T04 = unknown & string[]; // string[]
type T05 = unknown & unknown; // unknown
type T06 = unknown & any; // any

// In a union an unknown absorbs everything

type T10 = unknown | null; // unknown
type T11 = unknown | undefined; // unknown
type T12 = unknown | null | undefined; // unknown
type T13 = unknown | string; // unknown
type T14 = unknown | string[]; // unknown
type T15 = unknown | unknown; // unknown
type T16 = unknown | any; // any

// Type variable and unknown in union and intersection

type T20<T> = T & {};
```

```

type T21<T> = T | {}; // T | {}
type T22<T> = T & unknown; // T
type T23<T> = T | unknown; // unknown

// unknown in conditional types

type T30<T> = unknown extends T ? true : false; // Deferred
type T31<T> = T extends unknown ? true : false; // Deferred (so it distributes)
type T32<T> = never extends T ? true : false; // true
type T33<T> = T extends never ? true : false; // Deferred

// keyof unknown

type T40 = keyof any; // string | number | symbol
type T41 = keyof unknown; // never

// Only equality operators are allowed with unknown

function f10(x: unknown) {
  x == 5;
  x != 10;
  x >= 0; // Error
  x + 1; // Error
  x * 2; // Error
  -x; // Error
  +x; // Error
}

// No property accesses, element accesses, or function calls

function f11(x: unknown) {
  x.foo; // Error
  x[5]; // Error
  x(); // Error
  new x(); // Error
}

// typeof, instanceof, and user defined type predicates

declare function isFunction(x: unknown): x is Function;

function f20(x: unknown) {
  if (typeof x === "string" || typeof x === "number") {
    x; // string | number
  }
  if (x instanceof Error) {
    x; // Error
  }
  if (isFunction(x)) {
    x; // Function
  }
}

// Homomorphic mapped type over unknown

type T50<T> = { [P in keyof T]: number };
type T51 = T50<any>; // { [x: string]: number }
type T52 = T50<unknown>; // {}

// Anything is assignable to unknown

function f21<T>(pAny: any, pNever: never, pT: T) {
}

```

```

let x: unknown;
x = 123;
x = "hello";
x = [1, 2, 3];
x = new Error();
x = x;
x = pAny;
x = pNever;
x = pT;
}

// unknown assignable only to itself and any

function f22(x: unknown) {
    let v1: any = x;
    let v2: unknown = x;
    let v3: object = x; // Error
    let v4: string = x; // Error
    let v5: string[] = x; // Error
    let v6: {} = x; // Error
    let v7: {} | null | undefined = x; // Error
}

// Type parameter 'T' extends unknown' not related to object

function f23<T extends unknown>(x: T) {
    let y: object = x; // Error
}

// Anything but primitive assignable to { [x: string]: unknown }

function f24(x: { [x: string]: unknown }) {
    x = {};
    x = { a: 5 };
    x = [1, 2, 3];
    x = 123; // Error
}

// Locals of type unknown always considered initialized

function f25() {
    let x: unknown;
    let y = x;
}

// Spread of unknown causes result to be unknown

function f26(x: {}, y: unknown, z: any) {
    let o1 = { a: 42, ...x }; // { a: number }
    let o2 = { a: 42, ...x, ...y }; // unknown
    let o3 = { a: 42, ...x, ...y, ...z }; // any
}

// Functions with unknown return type don't need return expressions

function f27(): unknown {}

// Rest type cannot be created from unknown

function f28(x: unknown) {
    let { ...a } = x; // Error
}

```

```
// Class properties of type unknown don't need definite assignment

class C1 {
  a: string; // Error
  b: unknown;
  c: any;
}
```

Support for `defaultProps` in JSX

TypeScript 2.9 and earlier didn't leverage `React.defaultProps` declarations inside JSX components. Users would often have to declare properties optional and use non-null assertions inside of `render`, or they'd use type-assertions to fix up the type of the component before exporting it.

TypeScript 3.0 adds support for a new type alias in the `JSX` namespace called `LibraryManagedAttributes`. This helper type defines a transformation on the component's `Props` type, before using to check a JSX expression targeting it; thus allowing customization like: how conflicts between provided props and inferred props are handled, how inferences are mapped, how optionality is handled, and how inferences from differing places should be combined.

In short using this general type, we can model React's specific behavior for things like `defaultProps` and, to some extent, `propTypes`.

```
export interface Props {
  name: string;
}

export class Greet extends React.Component<Props> {
  render() {
    const { name } = this.props;
    return <div>Hello {name.toUpperCase()}!</div>;
  }
  static defaultProps = { name: "world" };
}

// Type-checks! No type assertions needed!
let el = <Greet />;
```

Caveats

Explicit types on `defaultProps`

The default-ed properties are inferred from the `defaultProps` property type. If an explicit type annotation is added, e.g. `static defaultProps: Partial<Props>;` the compiler will not be able to identify which properties have defaults (since the type of `defaultProps` include all properties of `Props`).

Use `static defaultProps: Pick<Props, "name">;` as an explicit type annotation instead, or do not add a type annotation as done in the example above.

For function components (formerly known as SFCs) use ES2015 default initializers:

```
function Greet({ name = "world" }: Props) {
    return <div>Hello {name.toUpperCase()}!</div>;
}
```

Changes to `@types/React`

Corresponding changes to add `LibraryManagedAttributes` definition to the `JSX` namespace in `@types/React` are still needed. Keep in mind that there are some limitations.

`/// <reference lib="..." />` reference directives

TypeScript adds a new triple-slash-reference directive (`/// <reference lib="name" />`), allowing a file to explicitly include an existing built-in *lib* file.

Built-in *lib* files are referenced in the same fashion as the `lib` compiler option in `tsconfig.json` (e.g. use `lib="es2015"` and not `lib="lib.es2015.d.ts"`, etc.).

For declaration file authors who rely on built-in types, e.g. DOM APIs or built-in JS run-time constructors like `Symbol` or `Iterable`, triple-slash-reference lib directives are the recommended. Previously these `.d.ts` files had to add forward/duplicate declarations of such types.

Example

Using `/// <reference lib="es2017.string" />` to one of the files in a compilation is equivalent to compiling with `--lib es2017.string`.

```
/// <reference lib="es2017.string" />
"foo".padStart(4);
```

[Go to TOC](#)

Mapped types on tuples and arrays

In TypeScript 3.1, mapped object types^[1] over tuples and arrays now produce new tuples/arrays, rather than creating a new type where members like `push()`, `pop()`, and `length` are converted. For example:

```
type MapToPromise<T> = { [K in keyof T]: Promise<T[K]> };

type Coordinate = [number, number];

type PromiseCoordinate = MapToPromise<Coordinate>; // [Promise<number>, Promise<number>]
```

`MapToPromise` takes a type `T`, and when that type is a tuple like `Coordinate`, only the numeric properties are converted. In `[number, number]`, there are two numerically named properties: `0` and `1`. When given a tuple like that, `MapToPromise` will create a new tuple where the `0` and `1` properties are `Promises` of the original type. So the resulting type `PromiseCoordinate` ends up with the type `[Promise<number>, Promise<number>]`.

Properties declarations on functions

TypeScript 3.1 brings the ability to define properties on function declarations and `const`-declared functions, simply by assigning to properties on these functions in the same scope. This allows us to write canonical JavaScript code without resorting to `namespace` hacks. For example:

```
function readImage(path: string, callback: (err: any, image: Image) => void) {
  // ...
}

readImage.sync = (path: string) => {
  const contents = fs.readFileSync(path);
  return decodeImageSync(contents);
};
```

Here, we have a function `readImage` which reads an image in a non-blocking asynchronous way. In addition to `readImage`, we've provided a convenience function on `readImage` itself called `readImage.sync`.

While ECMAScript exports are often a better way of providing this functionality, this new support allows code written in this style to "just work" in TypeScript. Additionally, this approach for property declarations allows us to express common patterns like `defaultProps` and `propTypes` on React function components (formerly known as SFCs).

```
export const FooComponent = ({ name }) => <div>Hello! I am {name}</div>

FooComponent.defaultProps = {
  name: "(anonymous)",
};
```

[1] More specifically, homomorphic mapped types like in the above form.

Version selection with `typesVersions`

Feedback from our community, as well as our own experience, has shown us that leveraging the newest TypeScript features while also accommodating users on the older versions are difficult. TypeScript introduces a new feature called `typesVersions` to help accommodate these scenarios.

You can read [about it in the Publishing section of the declaration files section](#)

[Go to TOC](#)

strictBindCallApply

TypeScript 3.2 introduces a new `strictBindCallApply` compiler option (in the `strict` family of options) with which the `bind`, `call`, and `apply` methods on function objects are strongly typed and strictly checked.

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

let a = foo.apply(undefined, [10]); // error: too few arguments
let b = foo.apply(undefined, [10, 20]); // error: 2nd argument is a number
let c = foo.apply(undefined, [10, "hello", 30]); // error: too many arguments
let d = foo.apply(undefined, [10, "hello"]); // okay! returns a string
```

This is achieved by introducing two new types, `CallableFunction` and `NewableFunction`, in `lib.d.ts`. These types contain specialized generic method declarations for `bind`, `call`, and `apply` for regular functions and constructor functions, respectively. The declarations use generic rest parameters (see #24897) to capture and reflect parameter lists in a strongly typed manner. In `strictBindCallApply` mode these declarations are used in place of the (very permissive) declarations provided by type `Function`.

Caveats

Since the stricter checks may uncover previously unreported errors, this is a breaking change in `strict` mode.

Additionally, another caveat of this new functionality is that due to certain limitations, `bind`, `call`, and `apply` can't yet fully model generic functions or functions that have overloads. When using these methods on a generic function, type parameters will be substituted with the empty object type (`{}`), and when used on a function with overloads, only the last overload will ever be modeled.

Generic spread expressions in object literals

In TypeScript 3.2, object literals now allow generic spread expressions which now produce intersection types, similar to the `Object.assign` function and JSX literals. For example:

```
function taggedObject<T, U extends string>(obj: T, tag: U) {
  return { ...obj, tag };
}

let x = taggedObject({ x: 10, y: 20 }, "point"); // { x: number, y: number } & { tag: "point" }
```

Property assignments and non-generic spread expressions are merged to the greatest extent possible on either side of a generic spread expression. For example:

```
function foo1<T>(t: T, obj1: { a: string }, obj2: { b: string }) {
  return { ...obj1, x: 1, ...t, ...obj2, y: 2 }; // { a: string, x: number } & T &
}
```

Non-generic spread expressions continue to be processed as before: Call and construct signatures are stripped, only non-method properties are preserved, and for properties with the same name, the type of the rightmost property is used. This contrasts with intersection types which concatenate call and construct signatures, preserve all properties, and intersect the types of properties with the same name. Thus, spreads of the same types may produce different results when they are created through instantiation of generic types:

```
function spread<T, U>(t: T, u: U) {
  return { ...t, ...u }; // T & U
}

declare let x: { a: string; b: number };
declare let y: { b: string; c: boolean };

let s1 = { ...x, ...y }; // { a: string, b: string, c: boolean }
let s2 = spread(x, y); // { a: string, b: number } & { b: string, c: boolean }
let b1 = s1.b; // string
let b2 = s2.b; // number & string
```

Generic object rest variables and parameters

TypeScript 3.2 also allows destructuring a rest binding from a generic variable. This is achieved by using the predefined `Pick` and `Exclude` helper types from `lib.d.ts`, and using the generic type in question as well as the names of the other bindings in the destructuring pattern.

```
function excludeTag<T extends { tag: string }>(obj: T) {
  let { tag, ...rest } = obj;
  return rest; // Pick<T, Exclude<keyof T, "tag">>
}

const taggedPoint = { x: 10, y: 20, tag: "point" };
const point = excludeTag(taggedPoint); // { x: number, y: number }
```

BigInt

BigInts are part of an upcoming proposal in ECMAScript that allow us to model theoretically arbitrarily large integers. TypeScript 3.2 brings type-checking for BigInts, as well as support for emitting BigInt literals when targeting `esnext`.

BigInt support in TypeScript introduces a new primitive type called the `bigint` (all lowercase). You can get a `bigint` by calling the `BigInt()` function or by writing out a BigInt literal by adding an `n` to the end of any integer numeric literal:

```
let foo: bigint = BigInt(100); // the BigInt function
let bar: bigint = 100n; // a BigInt literal

// *Slaps roof of fibonacci function*
// This bad boy returns ints that can get *so* big!
function fibonacci(n: bigint) {
```

```
let result = 1n;
for (let last = 0n, i = 0n; i < n; i++) {
  const current = result;
  result += last;
  last = current;
}
return result;
}

fibonacci(10000n);
```

While you might imagine close interaction between `number` and `bigint`, the two are separate domains.

```
declare let foo: number;
declare let bar: bigint;

foo = bar; // error: Type 'bigint' is not assignable to type 'number'.
bar = foo; // error: Type 'number' is not assignable to type 'bigint'.
```

As specified in ECMAScript, mixing `number`s and `bigint`s in arithmetic operations is an error. You'll have to explicitly convert values to `BigInt`s.

```
console.log(3.141592 * 10000n); // error
console.log(3145 * 10n); // error
console.log(BigInt(3145) * 10n); // okay!
```

Also important to note is that `bigint`s produce a new string when using the `typeof` operator: the string `"bigint"`. Thus, TypeScript correctly narrows using `typeof` as you'd expect.

```
function whatKindOfNumberIsIt(x: number | bigint) {
  if (typeof x === "bigint") {
    console.log("'x' is a bigint!");
  } else {
    console.log("'x' is a floating-point number");
  }
}
```

We'd like to extend a huge thanks to [Caleb Sander](#) for all the work on this feature. We're grateful for the contribution, and we're sure our users are too!

Caveats

As we mentioned, `BigInt` support is only available for the `esnext` target. It may not be obvious, but because `BigInts` have different behavior for mathematical operators like `+`, `-`, `*`, etc., providing functionality for older targets where the feature doesn't exist (like `es2017` and below) would involve rewriting each of these operations. TypeScript would need to dispatch to the correct behavior depending on the type, and so every addition, string concatenation, multiplication, etc. would involve a function call.

For that reason, we have no immediate plans to provide downleveling support. On the bright side, Node 11 and newer versions of Chrome already support this feature, so you'll be able to use `BigInts` there when targeting `esnext`.

Certain targets may include a polyfill or BigInt-like runtime object. For those purposes you may want to add `esnext bigint` to the `lib` setting in your compiler options.

Non-unit types as union discriminants

TypeScript 3.2 makes narrowing easier by relaxing rules for what it considers a discriminant property. Common properties of unions are now considered discriminants as long as they contain *some* singleton type (e.g. a string literal, `null`, or `undefined`), and they contain no generics.

As a result, TypeScript 3.2 considers the `error` property in the following example to be a discriminant, whereas before it wouldn't since `Error` isn't a singleton type. Thanks to this, narrowing works correctly in the body of the `unwrap` function.

```
type Result<T> = { error: Error; data: null } | { error: null; data: T };

function unwrap<T>(result: Result<T>) {
  if (result.error) {
    // Here 'error' is non-null
    throw result.error;
  }

  // Now 'data' is non-null
  return result.data;
}
```

tsconfig.json inheritance via Node.js packages

TypeScript 3.2 now resolves `tsconfig.json`s from `node_modules`. When using a bare path for the `extends` field in `tsconfig.json`, TypeScript will dive into `node_modules` packages for us.

```
{
  "extends": "@my-team/tsconfig-base",
  "include": ["./**/*"],
  "compilerOptions": {
    // Override certain options on a project-by-project basis.
    "strictBindCallApply": false
  }
}
```

Here, TypeScript will climb up `node_modules` folders looking for a `@my-team/tsconfig-base` package. For each of those packages, TypeScript will first check whether `package.json` contains a `"tsconfig"` field, and if it does, TypeScript will try to load a configuration file from that field. If neither exists, TypeScript will try to read from a `tsconfig.json` at the root. This is similar to the lookup process for `.js` files in packages that Node uses, and the `.d.ts` lookup process that TypeScript already uses.

This feature can be extremely useful for bigger organizations, or projects with lots of distributed dependencies.

The new `--showConfig` flag

`tsc`, the TypeScript compiler, supports a new flag called `--showConfig`. When running `tsc --showConfig`, TypeScript will calculate the effective `tsconfig.json` (after calculating options inherited from the `extends` field) and print that out. This can be useful for diagnosing configuration issues in general.

`Object.defineProperty` declarations in JavaScript

When writing in JavaScript files (using `allowJs`), TypeScript now recognizes declarations that use `Object.defineProperty`. This means you'll get better completions, and stronger type-checking when enabling type-checking in JavaScript files (by turning on the `checkJs` option or adding a `// @ts-check` comment to the top of your file).

```
// @ts-check

let obj = {};
Object.defineProperty(obj, "x", { value: "hello", writable: false });

obj.x.toLowerCase();
// ~~~~~
// error:
//   Property 'toLowerCase' does not exist on type 'string'.
//   Did you mean 'toLowercase'?

obj.x = "world";
// ~
// error:
//   Cannot assign to 'x' because it is a read-only property.
```

[Go to TOC](#)

Improved behavior for calling union types

In prior versions of TypeScript, unions of callable types could *only* be invoked if they had identical parameter lists.

```
type Fruit = "apple" | "orange";
type Color = "red" | "orange";

type FruitEater = (fruit: Fruit) => number; // eats and ranks the fruit
type ColorConsumer = (color: Color) => string; // consumes and describes the
colors

declare let f: FruitEater | ColorConsumer;

// Cannot invoke an expression whose type lacks a call signature.
//   Type 'FruitEater | ColorConsumer' has no compatible call signatures.ts(2349)
f("orange");
```

However, in the above example, both `FruitEater`s and `ColorConsumer`s should be able to take the string `"orange"`, and return either a `number` or a `string`.

In TypeScript 3.3, this is no longer an error.

```
type Fruit = "apple" | "orange";
type Color = "red" | "orange";

type FruitEater = (fruit: Fruit) => number; // eats and ranks the fruit
type ColorConsumer = (color: Color) => string; // consumes and describes the
colors

declare let f: FruitEater | ColorConsumer;

f("orange"); // It works! Returns a 'number | string'.

f("apple"); // error - Argument of type '"apple"' is not assignable to parameter
of type '"orange"'.

f("red"); // error - Argument of type '"red"' is not assignable to parameter of
type '"orange"'.
```

In TypeScript 3.3, the parameters of these signatures are *intersected* together to create a new signature.

In the example above, the parameters `fruit` and `color` are intersected together to a new parameter of type `Fruit & Color`. `Fruit & Color` is really the same as `("apple" | "orange") & ("red" | "orange")` which is equivalent to `("apple" & "red") | ("apple" & "orange") | ("orange" & "red") | ("orange" & "orange")`. Each of those impossible intersections reduces to `never`, and we're left with `"orange" & "orange"` which is just `"orange"`.

Caveats

This new behavior only kicks in when at most one type in the union has multiple overloads, and at most one type in the union has a generic signature. That means methods on `number[] | string[]` like `map` (which is generic) still won't be callable.

On the other hand, methods like `forEach` will now be callable, but under `noImplicitAny` there may be some issues.

```
interface Dog {
  kind: "dog";
  dogProp: any;
}
interface Cat {
  kind: "cat";
  catProp: any;
}

const catOrDogArray: Dog[] | Cat[] = [];

catOrDogArray.forEach(animal => {
  // ~~~~~~ error!
  // Parameter 'animal' implicitly has an 'any' type.
});
```

This is still strictly more capable in TypeScript 3.3, and adding an explicit type annotation will work.

```
interface Dog {
  kind: "dog";
  dogProp: any;
}
interface Cat {
  kind: "cat";
  catProp: any;
}

const catOrDogArray: Dog[] | Cat[] = [];
catOrDogArray.forEach((animal: Dog | Cat) => {
  if (animal.kind === "dog") {
    animal.dogProp;
    // ...
  } else if (animal.kind === "cat") {
    animal.catProp;
    // ...
  }
});
```

Incremental file watching for composite projects in `--build --watch`

TypeScript 3.0 introduced a new feature for structuring builds called "composite projects". Part of the goal here was to ensure users could break up large projects into smaller parts that build quickly and preserve project structure, without compromising the existing TypeScript experience. Thanks to composite projects, TypeScript can use `--build` mode to recompile only the set of projects and dependencies. You can think of this as optimizing *inter-project* builds.

TypeScript 2.7 also introduced `--watch` mode builds via a new incremental "builder" API. In a similar vein, the entire idea is that this mode only re-checks and re-emits changed files or files whose dependencies might impact type-checking. You can think of this as optimizing *intra-project* builds.

Prior to 3.3, building composite projects using `--build --watch` actually didn't use this incremental file watching infrastructure. An update in one project under `--build --watch` mode would force a full build of that project, rather than determining which files within that project were affected.

In TypeScript 3.3, `--build` mode's `--watch` flag *does* leverage incremental file watching as well. That can mean significantly faster builds under `--build --watch`. In our testing, this functionality has resulted in a **reduction of 50% to 75% in build times** of the original `--build --watch` times. [You can read more on the original pull request for the change](#) to see specific numbers, but we believe most composite project users will see significant wins here.

[Go to TOC](#)

Faster subsequent builds with the `--incremental` flag

TypeScript 3.4 introduces a new flag called `incremental` which tells TypeScript to save information about the project graph from the last compilation. The next time TypeScript is invoked with `incremental`, it will use that information to detect the least costly way to type-check and emit changes to your project.

```
// tsconfig.json
{
  "compilerOptions": {
    "incremental": true,
    "outDir": "./lib"
  },
  "include": ["./src"]
}
```

By default with these settings, when we run `tsc`, TypeScript will look for a file called `.tsbuildinfo` in the output directory (`./lib`). If `./lib/.tsbuildinfo` doesn't exist, it'll be generated. But if it does, `tsc` will try to use that file to incrementally type-check and update our output files.

These `.tsbuildinfo` files can be safely deleted and don't have any impact on our code at runtime - they're purely used to make compilations faster. We can also name them anything that we want, and place them anywhere we want using the `tsBuildInfoFile` option.

```
// front-end.tsconfig.json
{
  "compilerOptions": {
    "incremental": true,
    "tsBuildInfoFile": "./buildcache/front-end",
    "outDir": "./lib"
  },
  "include": ["./src"]
}
```

Composite projects

Part of the intent with composite projects (`tsconfig.json`s with `composite` set to `true`) is that references between different projects can be built incrementally. As such, composite projects will **always** produce `.tsbuildinfo` files.

`outFile`

When `outFile` is used, the build information file's name will be based on the output file's name. As an example, if our output JavaScript file is `./output/foo.js`, then under the `incremental` flag, TypeScript will generate the file `./output/foo.tsbuildinfo`. As above, this can be controlled with the `tsBuildInfoFile` option.

Higher order type inference from generic functions

TypeScript 3.4 can now produce generic function types when inference from other generic functions produces free type variables for inferences. This means many function composition patterns now work better in 3.4.

To get more specific, let's build up some motivation and consider the following `compose` function:

```
function compose<A, B, C>(f: (arg: A) => B, g: (arg: B) => C): (arg: A) => C {
    return (x) => g(f(x));
}
```

`compose` takes two other functions:

- `f` which takes some argument (of type `A`) and returns a value of type `B`
- `g` which takes an argument of type `B` (the type `f` returned), and returns a value of type `C`

`compose` then returns a function which feeds its argument through `f` and then `g`.

When calling this function, TypeScript will try to figure out the types of `A`, `B`, and `C` through a process called *type argument inference*. This inference process usually works pretty well:

```
interface Person {
    name: string;
    age: number;
}

function getDisplayName(p: Person) {
    return p.name.toLowerCase();
}

function getLength(s: string) {
    return s.length;
}

// has type '(p: Person) => number'
const getDisplayNameLength = compose(getDisplayName, getLength);

// works and returns the type 'number'
getDisplayNameLength({ name: "Person McPersonface", age: 42 });
```

The inference process is fairly straightforward here because `getDisplayName` and `getLength` use types that can easily be referenced. However, in TypeScript 3.3 and earlier, generic functions like `compose` didn't work so well when passed other generic functions.

```
interface Box<T> {
    value: T;
}

function makeArray<T>(x: T): T[] {
    return [x];
}

function makeBox<U>(value: U): Box<U> {
    return { value };
}
```

```

}
// has type '<arg: {} => Box<{}[]>' 
const makeBoxedArray = compose(makeArray, makeBox);

makeBoxedArray("hello!").value[0].toUpperCase();
// ~~~~~
// error: Property 'toUpperCase' does not exist on type '{}'.

```

In older versions, TypeScript would infer the empty object type (`{}`) when inferring from other type variables like `T` and `U`.

During type argument inference in TypeScript 3.4, for a call to a generic function that returns a function type, TypeScript *will*, as appropriate, propagate type parameters from generic function arguments onto the resulting function type.

In other words, instead of producing the type

```
<arg: {} => Box<{}[]>
```

TypeScript 3.4 produces the type

```
<T>(arg: T) => Box<T[]>
```

Notice that `T` has been propagated from `makeArray` into the resulting type's type parameter list. This means that genericity from `compose`'s arguments has been preserved and our `makeBoxedArray` sample will just work!

```

interface Box<T> {
  value: T;
}

function makeArray<T>(x: T): T[] {
  return [x];
}

function makeBox<U>(value: U): Box<U> {
  return { value };
}

// has type '<T>(arg: T) => Box<T[]>'
const makeBoxedArray = compose(makeArray, makeBox);

// works with no problem!
makeBoxedArray("hello!").value[0].toUpperCase();

```

For more details, you can [read more at the original change](#).

Improvements for `ReadonlyArray` and `readonly tuples`

TypeScript 3.4 makes it a little bit easier to use read-only array-like types.

A new syntax for `ReadonlyArray`

The `ReadonlyArray` type describes `Array`s that can only be read from. Any variable with a reference to a `ReadonlyArray` can't add, remove, or replace any elements of the array.

```
function foo(arr: ReadonlyArray<string>) {
  arr.slice(); // okay
  arr.push("hello!"); // error!
}
```

While it's good practice to use `ReadonlyArray` over `Array` when no mutation is intended, it's often been a pain given that arrays have a nicer syntax. Specifically, `number[]` is a shorthand version of `Array<number>`, just as `Date[]` is a shorthand for `Array<Date>`.

TypeScript 3.4 introduces a new syntax for `ReadonlyArray` using a new `readonly` modifier for array types.

```
function foo(arr: readonly string[]) {
  arr.slice(); // okay
  arr.push("hello!"); // error!
}
```

`readonly` tuples

TypeScript 3.4 also introduces new support for `readonly` tuples. We can prefix any tuple type with the `readonly` keyword to make it a `readonly` tuple, much like we now can with array shorthand syntax. As you might expect, unlike ordinary tuples whose slots could be written to, `readonly` tuples only permit reading from those positions.

```
function foo(pair: readonly [string, string]) {
  console.log(pair[0]); // okay
  pair[1] = "hello!"; // error
}
```

The same way that ordinary tuples are types that extend from `Array` - a tuple with elements of type `T1`, `T2`, ... `Tn` extends from `Array< T1 | T2 | ... Tn >` - `readonly` tuples are types that extend from `ReadonlyArray`. So a `readonly` tuple with elements `T1`, `T2`, ... `Tn` extends from `ReadonlyArray< T1 | T2 | ... Tn >`.

`readonly` mapped type modifiers and `readonly` arrays

In earlier versions of TypeScript, we generalized mapped types to operate differently on array-like types. This meant that a mapped type like `Boxify` could work on arrays and tuples alike.

```
interface Box<T> {
  value: T;
}

type Boxify<T> = {
  [K in keyof T]: Box<T[K]>;
};
```

```
// { a: Box<string>, b: Box<number> }
type A = Boxify<{ a: string; b: number }>;
// Array<Box<number>>
type B = Boxify<number[]>;
// [Box<string>, Box<number>]
type C = Boxify<[string, boolean]>;
```

Unfortunately, mapped types like the `Readonly` utility type were effectively no-ops on array and tuple types.

```
// lib.d.ts
type Readonly<T> = {
  readonly [K in keyof T]: T[K];
};

// How code acted *before* TypeScript 3.4

// { readonly a: string, readonly b: number }
type A = Readonly<{ a: string; b: number }>;
// number[]
type B = Readonly<number[]>;
// [string, boolean]
type C = Readonly<[string, boolean]>;
```

In TypeScript 3.4, the `readonly` modifier in a mapped type will automatically convert array-like types to their corresponding `readonly` counterparts.

```
// How code acts now *with* TypeScript 3.4

// { readonly a: string, readonly b: number }
type A = Readonly<{ a: string; b: number }>;
// readonly number[]
type B = Readonly<number[]>;
// readonly [string, boolean]
type C = Readonly<[string, boolean]>;
```

Similarly, you could write a utility type like `Writable` mapped type that strips away `readonly`-ness, and that would convert `readonly` array containers back to their mutable equivalents.

```
type Writable<T> = {
  -readonly [K in keyof T]: T[K];
};

// { a: string, b: number }
type A = Writable<{
  readonly a: string;
  readonly b: number;
}>;
// number[]
type B = Writable<readonly number[]>;
```

```
// [string, boolean]
type C = Writable<readonly [string, boolean]>;
```

Caveats

Despite its appearance, the `readonly` type modifier can only be used for syntax on array types and tuple types. It is not a general-purpose type operator.

```
let err1: readonly Set<number>; // error!
let err2: readonly Array<boolean>; // error!
let okay: readonly boolean[]; // works fine
```

You can [see more details in the pull request](#).

const assertions

TypeScript 3.4 introduces a new construct for literal values called `const` assertions. Its syntax is a type assertion with `const` in place of the type name (e.g. `123 as const`). When we construct new literal expressions with `const` assertions, we can signal to the language that

- no literal types in that expression should be widened (e.g. no going from `"hello"` to `string`)
- object literals get `readonly` properties
- array literals become `readonly` tuples

```
// Type '"hello"'
let x = "hello" as const;

// Type 'readonly [10, 20]'
let y = [10, 20] as const;

// Type '{ readonly text: "hello" }'
let z = { text: "hello" } as const;
```

Outside of `.tsx` files, the angle bracket assertion syntax can also be used.

```
// Type '"hello"'
let x = <const>"hello";

// Type 'readonly [10, 20]'
let y = <const>[10, 20];

// Type '{ readonly text: "hello" }'
let z = <const>{ text: "hello" };
```

This feature means that types that would otherwise be used just to hint immutability to the compiler can often be omitted.

```
// Works with no types referenced or declared.
// We only needed a single const assertion.
function getShapes() {
  let result = [
    { kind: "circle", radius: 100 },
    { kind: "square", sideLength: 50 },
```

```

    ] as const;
}

return result;
}

for (const shape of getShapes()) {
  // Narrows perfectly!
  if (shape.kind === "circle") {
    console.log("Circle radius", shape.radius);
  } else {
    console.log("Square side length", shape.sideLength);
  }
}

```

Notice the above needed no type annotations. The `const` assertion allowed TypeScript to take the most specific type of the expression.

This can even be used to enable `enum`-like patterns in plain JavaScript code if you choose not to use TypeScript's `enum` construct.

```

export const Colors = {
  red: "RED",
  blue: "BLUE",
  green: "GREEN",
} as const;

// or use an 'export default'

export default {
  red: "RED",
  blue: "BLUE",
  green: "GREEN",
} as const;

```

Caveats

One thing to note is that `const` assertions can only be applied immediately on simple literal expressions.

```

// Error! A 'const' assertion can only be applied to a
// to a string, number, boolean, array, or object literal.
let a = (Math.random() < 0.5 ? 0 : 1) as const;
let b = (60 * 60 * 1000) as const;

// Works!
let c = Math.random() < 0.5 ? (0 as const) : (1 as const);
let d = 3_600_000 as const;

```

Another thing to keep in mind is that `const` contexts don't immediately convert an expression to be fully immutable.

```

let arr = [1, 2, 3, 4];

let foo = {
  name: "foo",
  contents: arr,
} as const;

```

```
foo.name = "bar"; // error!
foo.contents = []; // error!
foo.contents.push(5); // ...works!
```

For more details, you can [check out the respective pull request](#).

Type-checking for `globalThis`

TypeScript 3.4 introduces support for type-checking ECMAScript's new `globalThis` - a global variable that, well, refers to the global scope. Unlike the above solutions, `globalThis` provides a standard way for accessing the global scope which can be used across different environments.

```
// in a global file:
var abc = 100;
// Refers to 'abc' from above.
globalThis.abc = 200;
```

Note that global variables declared with `let` and `const` don't show up on `globalThis`.

```
let answer = 42;
// error! Property 'answer' does not exist on 'typeof globalThis'.
globalThis.answer = 333333;
```

It's also important to note that TypeScript doesn't transform references to `globalThis` when compiling to older versions of ECMAScript. As such, unless you're targeting evergreen browsers (which already support `globalThis`), you may want to [use an appropriate polyfill](#) instead.

For more details on the implementation, see [the feature's pull request](#).

[Go to TOC](#)

Speed improvements

TypeScript 3.5 introduces several optimizations around type-checking and incremental builds.

Type-checking speed-ups

TypeScript 3.5 contains certain optimizations over TypeScript 3.4 for type-checking more efficiently. These improvements are significantly more pronounced in editor scenarios where type-checking drives operations like code completion lists.

--incremental improvements

TypeScript 3.5 improves on 3.4's `incremental` build mode, by saving information about how the state of the world was calculated - compiler settings, why files were looked up, where files were found, etc. In scenarios involving hundreds of projects using TypeScript's project references in `--build` mode, we've found that the amount of time rebuilding can be reduced by as much as 68% compared to TypeScript 3.4!

For more details, you can see the pull requests to

- [cache module resolution](#)
- [cache settings calculated from `tsconfig.json`](#)

The `Omit` helper type

TypeScript 3.5 introduces the new `Omit` helper type, which creates a new type with some properties dropped from the original.

```
type Person = {
  name: string;
  age: number;
  location: string;
};

type QuantumPerson = Omit<Person, "location">;

// equivalent to
type QuantumPerson = {
  name: string;
  age: number;
};
```

Here we were able to copy over all the properties of `Person` except for `location` using the `Omit` helper.

For more details, [see the pull request on GitHub to add `Omit`](#), as well as [the change to use `Omit` for object rest](#).

Improved excess property checks in union types

In TypeScript 3.4 and earlier, certain excess properties were allowed in situations where they really shouldn't have been. For instance, TypeScript 3.4 permitted the incorrect `name` property in the object literal even though its types don't match between `Point` and `Label`.

```
type Point = {
  x: number;
  y: number;
};

type Label = {
  name: string;
};

const thing: Point | Label = {
  x: 0,
  y: 0,
  name: true // uh-oh!
};
```

Previously, a non-discriminated union wouldn't have *any* excess property checking done on its members, and as a result, the incorrectly typed `name` property slipped by.

In TypeScript 3.5, the type-checker at least verifies that all the provided properties belong to *some* union member and have the appropriate type, meaning that the sample above correctly issues an error.

Note that partial overlap is still permitted as long as the property types are valid.

```
const pl: Point | Label = {
  x: 0,
  y: 0,
  name: "origin" // okay
};
```

The `--allowUmdGlobalAccess` flag

In TypeScript 3.5, you can now reference UMD global declarations like

```
export as namespace foo;
```

from anywhere - even modules - using the new `allowUmdGlobalAccess` flag.

This mode adds flexibility for mixing and matching the way 3rd party libraries, where globals that libraries declare can always be consumed, even from within modules.

For more details, see the [pull request on GitHub](#).

Smarter union type checking

In TypeScript 3.4 and prior, the following example would fail:

```

type S = { done: boolean; value: number };
type T = { done: false; value: number } | { done: true; value: number };

declare let source: S;
declare let target: T;

target = source;

```

That's because `s` isn't assignable to `{ done: false, value: number }` nor `{ done: true, value: number }`. Why? Because the `done` property in `S` isn't specific enough - it's `boolean` whereas each constituent of `T` has a `done` property that's specifically `true` or `false`. That's what we meant by each constituent type being checked in isolation: TypeScript doesn't just union each property together and see if `s` is assignable to that. If it did, some bad code could get through like the following:

```

interface Foo {
  kind: "foo";
  value: string;
}

interface Bar {
  kind: "bar";
  value: number;
}

function doSomething(x: Foo | Bar) {
  if (x.kind === "foo") {
    x.value.toLowerCase();
  }
}

// uh-oh - luckily TypeScript errors here!
doSomething({
  kind: "foo",
  value: 123
});

```

However, this was a bit overly strict for the original example. If you figure out the precise type of any possible value of `s`, you can actually see that it matches the types in `T` exactly.

In TypeScript 3.5, when assigning to types with discriminant properties like in `T`, the language actually *will* go further and decompose types like `S` into a union of every possible inhabitant type. In this case, since `boolean` is a union of `true` and `false`, `S` will be viewed as a union of `{ done: false, value: number }` and `{ done: true, value: number }`.

For more details, you can [see the original pull request on GitHub](#).

Higher order type inference from generic constructors

In TypeScript 3.4, we improved inference for when generic functions that return functions like so:

```

function compose<T, U, V>(f: (x: T) => U, g: (y: U) => V): (x: T) => V {
  return x => g(f(x));
}

```

took other generic functions as arguments, like so:

```
function arrayify<T>(x: T): T[] {
  return [x];
}

type Box<U> = { value: U };
function boxify<U>(y: U): Box<U> {
  return { value: y };
}

let newFn = compose(arrayify, boxify);
```

Instead of a relatively useless type like `(x: {}) => Box<{}[]>`, which older versions of the language would infer, TypeScript 3.4's inference allows `newFn` to be generic. Its new type is `<T>(x: T) => Box<T[]>`.

TypeScript 3.5 generalizes this behavior to work on constructor functions as well.

```
class Box<T> {
  kind: "box";
  value: T;
  constructor(value: T) {
    this.value = value;
  }
}

class Bag<U> {
  kind: "bag";
  value: U;
  constructor(value: U) {
    this.value = value;
  }
}

function composeCtor<T, U, V>(
  F: new (x: T) => U,
  G: new (y: U) => V
): (x: T) => V {
  return x => new G(new F(x));
}

let f = composeCtor(Box, Bag); // has type '<T>(x: T) => Bag<Box<T>>'
let a = f(1024); // has type 'Bag<Box<number>>'
```

In addition to compositional patterns like the above, this new inference on generic constructors means that functions that operate on class components in certain UI libraries like React can more correctly operate on generic class components.

```
type ComponentClass<P> = new (props: P) => Component<P>;
declare class Component<P> {
  props: P;
  constructor(props: P);
}

declare function myHoc<P>(C: ComponentClass<P>): ComponentClass<P>;
type NestedProps<T> = { foo: number; stuff: T };
```

```
declare class GenericComponent<T> extends Component<NestedProps<T>> {}  
// type is 'new <T>(props: NestedProps<T>) => Component<NestedProps<T>>'  
const GenericComponent2 = myHoc(GenericComponent);
```

To learn more, check out the original pull request on [GitHub](#).

[Go to TOC](#)

Stricter Generators

TypeScript 3.6 introduces stricter checking for iterators and generator functions. In earlier versions, users of generators had no way to differentiate whether a value was yielded or returned from a generator.

```
function* foo() {
  if (Math.random() < 0.5) yield 100;
  return "Finished!";
}

let iter = foo();
let curr = iter.next();
if (curr.done) {
  // TypeScript 3.5 and prior thought this was a 'string | number'.
  // It should know it's 'string' since 'done' was 'true'!
  curr.value;
}
```

Additionally, generators just assumed the type of `yield` was always `any`.

```
function* bar() {
  let x: { hello(): void } = yield;
  x.hello();
}

let iter = bar();
iter.next();
iter.next(123); // oops! runtime error!
```

In TypeScript 3.6, the checker now knows that the correct type for `curr.value` should be `string` in our first example, and will correctly error on our call to `next()` in our last example. This is thanks to some changes in the `Iterator` and `IteratorResult` type declarations to include a few new type parameters, and to a new type that TypeScript uses to represent generators called the `Generator` type.

The `Iterator` type now allows users to specify the yielded type, the returned type, and the type that `next` can accept.

```
interface Iterator<T, TReturn = any, TNext = undefined> {
  // Takes either 0 or 1 arguments - doesn't accept 'undefined'
  next(...args: [] | [TNext]): IteratorResult<T, TReturn>;
  return?(value?: TReturn): IteratorResult<T, TReturn>;
  throw?(e?: any): IteratorResult<T, TReturn>;
}
```

Building on that work, the new `Generator` type is an `Iterator` that always has both the `return` and `throw` methods present, and is also iterable.

```
interface Generator<T = unknown, TReturn = any, TNext = unknown>
  extends Iterator<T, TReturn, TNext> {
  next(...args: [] | [TNext]): IteratorResult<T, TReturn>;
  return?(value: TReturn): IteratorResult<T, TReturn>;
  throw?(e: any): IteratorResult<T, TReturn>;
  [Symbol.iterator](): Generator<T, TReturn, TNext>;
}
```

To allow differentiation between returned values and yielded values, TypeScript 3.6 converts the `IteratorResult` type to a discriminated union type:

```
type IteratorResult<T, TReturn = any> =
  | IteratorYieldResult<T>
  | IteratorReturnResult<TReturn>;

interface IteratorYieldResult<TYield> {
  done?: false;
  value: TYield;
}

interface IteratorReturnResult<TReturn> {
  done: true;
  value: TReturn;
}
```

In short, what this means is that you'll be able to appropriately narrow down values from iterators when dealing with them directly.

To correctly represent the types that can be passed in to a generator from calls to `next()`, TypeScript 3.6 also infers certain uses of `yield` within the body of a generator function.

```
function* foo() {
  let x: string = yield;
  console.log(x.toUpperCase());
}

let x = foo();
x.next(); // first call to 'next' is always ignored
x.next(42); // error! 'number' is not assignable to 'string'
```

If you'd prefer to be explicit, you can also enforce the type of values that can be returned, yielded, and evaluated from `yield` expressions using an explicit return type. Below, `next()` can only be called with `boolean`s, and depending on the value of `done`, `value` is either a `string` or a `number`.

```
/*
 * - yields numbers
 * - returns strings
 * - can be passed in booleans
 */
function* counter(): Generator<number, string, boolean> {
  let i = 0;
  while (true) {
    if (yield i++) {
      break;
    }
  }
  return "done!";
}

var iter = counter();
var curr = iter.next();
while (!curr.done) {
  console.log(curr.value);
  curr = iter.next(curr.value === 5);
}
```

```
console.log(curr.value.toUpperCase());
// prints:
// 0
// 1
// 2
// 3
// 4
// 5
// DONE!
```

For more details on the change, [see the pull request here](#).

More Accurate Array Spread

In pre-ES2015 targets, the most faithful emit for constructs like `for / of` loops and array spreads can be a bit heavy. For this reason, TypeScript uses a simpler emit by default that only supports array types, and supports iterating on other types using the `downlevelIteration` flag. The looser default without `downlevelIteration` works fairly well; however, there were some common cases where the transformation of array spreads had observable differences. For example, the following array containing a spread

```
[...Array(5)];
```

can be rewritten as the following array literal

```
[undefined, undefined, undefined, undefined, undefined];
```

However, TypeScript would instead transform the original code into this code:

```
Array(5).slice();
```

which is slightly different. `Array(5)` produces an array with a length of 5, but with no defined property slots.

TypeScript 3.6 introduces a new `__spreadArrays` helper to accurately model what happens in ECMAScript 2015 in older targets outside of `downlevelIteration`. `__spreadArrays` is also available in `tslib`.

For more information, [see the relevant pull request](#).

Improved UX Around Promises

TypeScript 3.6 introduces some improvements for when `Promise`s are mis-handled.

For example, it's often very common to forget to `.then()` or `await` the contents of a `Promise` before passing it to another function. TypeScript's error messages are now specialized, and inform the user that perhaps they should consider using the `await` keyword.

```
interface User {
  name: string;
  age: number;
```

```

    location: string;
}

declare function getUserData(): Promise<User>;
declare function displayUser(user: User): void;

async function f() {
  displayUser(getUserData());
  // ~~~~~
  // Argument of type 'Promise<User>' is not assignable to parameter of type
  'User'.
  // ...
  // Did you forget to use 'await'?
}

```

It's also common to try to access a method before `await`-ing or `.then()`-ing a `Promise`. This is another example, among many others, where we're able to do better.

```

async function getCuteAnimals() {
  fetch("https://reddit.com/r/aww.json").json();
  // ~~~
  // Property 'json' does not exist on type 'Promise<Response>'.
  // ...
  // Did you forget to use 'await'?
}

```

For more details, [see the originating issue](#), as well as the pull requests that link back to it.

Better Unicode Support for Identifiers

TypeScript 3.6 contains better support for Unicode characters in identifiers when emitting to ES2015 and later targets.

```
const hello = "world"; // previously disallowed, now allowed in '--target es2015'
```

`import.meta` Support in SystemJS

TypeScript 3.6 supports transforming `import.meta` to `context.meta` when your `module` target is set to `system`.

```

// This module:
console.log(import.meta.url);

// gets turned into the following:

System.register([], function (exports, context) {
  return {
    setters: [],
    execute: function () {
      console.log(context.meta.url);
    },
  };
});

```

get and set Accessors Are Allowed in Ambient Contexts

In previous versions of TypeScript, the language didn't allow `get` and `set` accessors in ambient contexts (like in `declare -d` classes, or in `.d.ts` files in general). The rationale was that accessors weren't distinct from properties as far as writing and reading to these properties; however, because ECMAScript's class fields proposal may have differing behavior from in existing versions of TypeScript, we realized we needed a way to communicate this different behavior to provide appropriate errors in subclasses.

As a result, users can write getters and setters in ambient contexts in TypeScript 3.6.

```
declare class Foo {
  // Allowed in 3.6+.
  get x(): number;
  set x(val: number);
}
```

In TypeScript 3.7, the compiler itself will take advantage of this feature so that generated `.d.ts` files will also emit `get`/`set` accessors.

Ambient Classes and Functions Can Merge

In previous versions of TypeScript, it was an error to merge classes and functions under any circumstances. Now, ambient classes and functions (classes/functions with the `declare` modifier, or in `.d.ts` files) can merge. This means that now you can write the following:

```
export declare function Point2D(x: number, y: number): Point2D;
export declare class Point2D {
  x: number;
  y: number;
  constructor(x: number, y: number);
}
```

instead of needing to use

```
export interface Point2D {
  x: number;
  y: number;
}
export declare var Point2D: {
  (x: number, y: number): Point2D;
  new (x: number, y: number): Point2D;
};
```

One advantage of this is that the callable constructor pattern can be easily expressed while also allowing namespaces to merge with these declarations (since `var` declarations can't merge with `namespaces`).

In TypeScript 3.7, the compiler will take advantage of this feature so that `.d.ts` files generated from `.js` files can appropriately capture both the callability and constructability of a class-like function.

For more details, see the original PR on [GitHub](#).

APIs to Support `--build` and `--incremental`

TypeScript 3.0 introduced support for referencing other and building them incrementally using the `--build` flag. Additionally, TypeScript 3.4 introduced the `incremental` flag for saving information about previous compilations to only rebuild certain files. These flags were incredibly useful for structuring projects more flexibly and speeding builds up. Unfortunately, using these flags didn't work with 3rd party build tools like Gulp and Webpack. TypeScript 3.6 now exposes two sets of APIs to operate on project references and incremental program building.

For creating `incremental` builds, users can leverage the `createIncrementalProgram` and `createIncrementalCompilerHost` APIs. Users can also re-hydrate old program instances from `.tsbuildinfo` files generated by this API using the newly exposed `readBuilderProgram` function, which is only meant to be used as for creating new programs (i.e. you can't modify the returned instance - it's only meant to be used for the `oldProgram` parameter in other `create*Program` functions).

For leveraging project references, a new `createSolutionBuilder` function has been exposed, which returns an instance of the new type `SolutionBuilder`.

For more details on these APIs, you can [see the original pull request](#).

Semicolon-Aware Code Edits

Editors like Visual Studio and Visual Studio Code can automatically apply quick fixes, refactorings, and other transformations like automatically importing values from other modules. These transformations are powered by TypeScript, and older versions of TypeScript unconditionally added semicolons to the end of every statement; unfortunately, this disagreed with many users' style guidelines, and many users were displeased with the editor inserting semicolons.

TypeScript is now smart enough to detect whether your file uses semicolons when applying these sorts of edits. If your file generally lacks semicolons, TypeScript won't add one.

For more details, [see the corresponding pull request](#).

Smarter Auto-Import Syntax

JavaScript has a lot of different module syntaxes or conventions: the one in the ECMAScript standard, the one Node already supports (CommonJS), AMD, System.js, and more! For the most part, TypeScript would default to auto-importing using ECMAScript module syntax, which was often inappropriate in certain TypeScript projects with different compiler settings, or in Node projects with plain JavaScript and `require` calls.

TypeScript 3.6 is now a bit smarter about looking at your existing imports before deciding on how to auto-import other modules. You can [see more details in the original pull request here](#).

New TypeScript Playground

The TypeScript playground has received a much-needed refresh with handy new functionality! The new playground is largely a fork of [Artem Tyurin's TypeScript playground](#) which community members have been using more and more. We owe Artem a big thanks for helping out here!

The new playground now supports many new options including:

- The `target` option (allowing users to switch out of `es5` to `es3`, `es2015`, `esnext`, etc.)
- All the strictness flags (including just `strict`)
- Support for plain JavaScript files (using `allowJS` and optionally `checkJS`)

These options also persist when sharing links to playground samples, allowing users to more reliably share examples without having to tell the recipient "oh, don't forget to turn on the `noImplicitAny` option!".

In the near future, we're going to be refreshing the playground samples, adding JSX support, and polishing automatic type acquisition, meaning that you'll be able to see the same experience on the playground as you'd get in your personal editor.

As we improve the playground and the website, [we welcome feedback and pull requests on GitHub!](#)

[Go to TOC](#)

Optional Chaining

Playground

Optional chaining is [issue #16](#) on our issue tracker. For context, there have been over 23,000 issues on the TypeScript issue tracker since then.

At its core, optional chaining lets us write code where TypeScript can immediately stop running some expressions if we run into a `null` or `undefined`. The star of the show in optional chaining is the new `?.` operator for *optional property accesses*. When we write code like

```
let x = foo?.bar.baz();
```

this is a way of saying that when `foo` is defined, `foo.bar.baz()` will be computed; but when `foo` is `null` or `undefined`, stop what we're doing and just return `undefined`."

More plainly, that code snippet is the same as writing the following.

```
let x = foo === null || foo === undefined ? undefined : foo.bar.baz();
```

Note that if `bar` is `null` or `undefined`, our code will still hit an error accessing `baz`. Likewise, if `baz` is `null` or `undefined`, we'll hit an error at the call site. `?.` only checks for whether the value on the *left* of it is `null` or `undefined` - not any of the subsequent properties.

You might find yourself using `?.` to replace a lot of code that performs repetitive nullish checks using the `&&` operator.

```
// Before
if (foo && foo.bar && foo.bar.baz) {
  // ...
}

// After-ish
if (foo?.bar?.baz) {
  // ...
}
```

Keep in mind that `?.` acts differently than those `&&` operations since `&&` will act specially on "falsy" values (e.g. the empty string, `0`, `NaN`, and, well, `false`), but this is an intentional feature of the construct. It doesn't short-circuit on valid data like `0` or empty strings.

Optional chaining also includes two other operations. First there's the *optional element access* which acts similarly to optional property accesses, but allows us to access non-identifier properties (e.g. arbitrary strings, numbers, and symbols):

```
/*
 * Get the first element of the array if we have an array.
 * Otherwise return undefined.
 */
function tryGetFirstElement<T>(arr?: T[]) {
  return arr?.[0];
```

```
// equivalent to
//   return (arr === null || arr === undefined) ?
//     undefined :
//     arr[0];
}
```

There's also *optional call*, which allows us to conditionally call expressions if they're not `null` or `undefined`.

```
async function makeRequest(url: string, log?: (msg: string) => void) {
  log?.(`Request started at ${new Date().toISOString()}`);
  // roughly equivalent to
  // if (log != null) {
  //   log(`Request started at ${new Date().toISOString()}`);
  // }

  const result = (await fetch(url)).json();

  log?.(`Request finished at ${new Date().toISOString()}`);

  return result;
}
```

The "short-circuiting" behavior that optional chains have is limited property accesses, calls, element accesses - it doesn't expand any further out from these expressions. In other words,

```
let result = foo?.bar / someComputation();
```

doesn't stop the division or `someComputation()` call from occurring. It's equivalent to

```
let temp = foo === null || foo === undefined ? undefined : foo.bar;
let result = temp / someComputation();
```

That might result in dividing `undefined`, which is why in `strictNullChecks`, the following is an error.

```
function barPercentage(foo?: { bar: number }) {
  return foo?.bar / 100;
  // ~~~~~
  // Error: Object is possibly undefined.
}
```

More details, you can [read up on the proposal](#) and [view the original pull request](#).

Nullish Coalescing

[Playground](#)

The *nullish coalescing operator* is another upcoming ECMAScript feature that goes hand-in-hand with optional chaining, and which our team has been involved with championing in TC39.

You can think of this feature - the `??` operator - as a way to "fall back" to a default value when dealing with `null` or `undefined`. When we write code like

```
let x = foo ?? bar();
```

this is a new way to say that the value `foo` will be used when it's "present"; but when it's `null` or `undefined`, calculate `bar()` in its place.

Again, the above code is equivalent to the following.

```
let x = foo !== null && foo !== undefined ? foo : bar();
```

The `??` operator can replace uses of `||` when trying to use a default value. For example, the following code snippet tries to fetch the volume that was last saved in `localStorage` (if it ever was); however, it has a bug because it uses `||`.

```
function initializeAudio() {
  let volume = localStorage.volume || 0.5;
  // ...
}
```

When `localStorage.volume` is set to `0`, the page will set the volume to `0.5` which is unintended. `??` avoids some unintended behavior from `0`, `NaN` and `""` being treated as falsy values.

We owe a large thanks to community members [Wenlu Wang](#) and [Titian Cernicova Dragomir](#) for implementing this feature! For more details, [check out their pull request](#) and [the nullish coalescing proposal repository](#).

Assertion Functions

Playground

There's a specific set of functions that `throw` an error if something unexpected happened. They're called "assertion" functions. As an example, Node.js has a dedicated function for this called `assert`.

```
assert(someValue === 42);
```

In this example if `someValue` isn't equal to `42`, then `assert` will throw an `AssertionError`.

Assertions in JavaScript are often used to guard against improper types being passed in. For example,

```
function multiply(x, y) {
  assert(typeof x === "number");
  assert(typeof y === "number");
  return x * y;
}
```

Unfortunately in TypeScript these checks could never be properly encoded. For loosely-typed code this meant TypeScript was checking less, and for slightly conservative code it often forced users to use type assertions.

```
function yell(str) {
  assert(typeof str === "string");
  return str.toUpperCase();
  // Oops! We misspelled 'toUpperCase'.
  // Would be great if TypeScript still caught this!
}
```

The alternative was to instead rewrite the code so that the language could analyze it, but this isn't convenient.

```
function yell(str) {
  if (typeof str !== "string") {
    throw new TypeError("str should have been a string.");
  }
  // Error caught!
  return str.toUpperCase();
}
```

Ultimately the goal of TypeScript is to type existing JavaScript constructs in the least disruptive way. For that reason, TypeScript 3.7 introduces a new concept called "assertion signatures" which model these assertion functions.

The first type of assertion signature models the way that Node's `assert` function works. It ensures that whatever condition is being checked must be true for the remainder of the containing scope.

```
function assert(condition: any, msg?: string): asserts condition {
  if (!condition) {
    throw new AssertionError(msg);
  }
}
```

`asserts condition` says that whatever gets passed into the `condition` parameter must be true if the `assert` returns (because otherwise it would throw an error). That means that for the rest of the scope, that condition must be truthy. As an example, using this assertion function means we *do* catch our original `yell` example.

```
function yell(str) {
  assert(typeof str === "string");
  return str.toUpperCase();
  // ~~~~~
  // error: Property 'toUpperCase' does not exist on type 'string'.
  // Did you mean 'toUpperCase'?
}

function assert(condition: any, msg?: string): asserts condition {
  if (!condition) {
    throw new AssertionError(msg);
  }
}
```

The other type of assertion signature doesn't check for a condition, but instead tells TypeScript that a specific variable or property has a different type.

```
function assertIsString(val: any): asserts val is string {
  if (typeof val !== "string") {
    throw new AssertionError("Not a string!");
  }
}
```

Here `asserts val is string` ensures that after any call to `assertIsString`, any variable passed in will be known to be a `string`.

```
function yell(str: any) {
  assertIsString(str);

  // Now TypeScript knows that 'str' is a 'string'.

  return str.toUpperCase();
  // ~~~~~
  // error: Property 'toUpperCase' does not exist on type 'string'.
  // Did you mean 'toUpperCase'?
}
```

These assertion signatures are very similar to writing type predicate signatures:

```
function isString(val: any): val is string {
  return typeof val === "string";
}

function yell(str: any) {
  if (isString(str)) {
    return str.toUpperCase();
  }
  throw "Oops!";
}
```

And just like type predicate signatures, these assertion signatures are incredibly expressive. We can express some fairly sophisticated ideas with these.

```
function assertIsDefined<T>(val: T): asserts val is NonNullable<T> {
  if (val === undefined || val === null) {
    throw new AssertionError(
      `Expected 'val' to be defined, but received ${val}`
    );
}
```

To read up more about assertion signatures, [check out the original pull request](#).

Better Support for `never`-Returning Functions

As part of the work for assertion signatures, TypeScript needed to encode more about where and which functions were being called. This gave us the opportunity to expand support for another class of functions: functions that return `never`.

The intent of any function that returns `never` is that it never returns. It indicates that an exception was thrown, a halting error condition occurred, or that the program exited. For example, `process.exit(...)` in `@types/node` is specified to return `never`.

In order to ensure that a function never potentially returned `undefined` or effectively returned from all code paths, TypeScript needed some syntactic signal - either a `return` or `throw` at the end of a function. So users found themselves `return`-ing their failure functions.

```
function dispatch(x: string | number): SomeType {
  if (typeof x === "string") {
    return doThingWithString(x);
  } else if (typeof x === "number") {
    return doThingWithNumber(x);
  }
  return process.exit(1);
}
```

Now when these `never`-returning functions are called, TypeScript recognizes that they affect the control flow graph and accounts for them.

```
function dispatch(x: string | number): SomeType {
  if (typeof x === "string") {
    return doThingWithString(x);
  } else if (typeof x === "number") {
    return doThingWithNumber(x);
  }
  process.exit(1);
}
```

As with assertion functions, you can [read up more at the same pull request](#).

(More) Recursive Type Aliases

[Playground](#)

Type aliases have always had a limitation in how they could be "recursively" referenced. The reason is that any use of a type alias needs to be able to substitute itself with whatever it aliases. In some cases, that's not possible, so the compiler rejects certain recursive aliases like the following:

```
type Foo = Foo;
```

This is a reasonable restriction because any use of `Foo` would need to be replaced with `Foo` which would need to be replaced with `Foo` which would need to be replaced with `Foo` which... well, hopefully you get the idea! In the end, there isn't a type that makes sense in place of `Foo`.

This is fairly [consistent with how other languages treat type aliases](#), but it does give rise to some slightly surprising scenarios for how users leverage the feature. For example, in TypeScript 3.6 and prior, the following causes an error.

```
type ValueOrArray<T> = T | Array<ValueOrArray<T>>;
// ~~~~~
// error: Type alias 'ValueOrArray' circularly references itself.
```

This is strange because there is technically nothing wrong with any use users could always write what was effectively the same code by introducing an interface.

```
type ValueOrArray<T> = T | ArrayOfValueOrArray<T>;
interface ArrayOfValueOrArray<T> extends Array<ValueOrArray<T>> {}
```

Because interfaces (and other object types) introduce a level of indirection and their full structure doesn't need to be eagerly built out, TypeScript has no problem working with this structure.

But workaround of introducing the interface wasn't intuitive for users. And in principle there really wasn't anything wrong with the original version of `ValueOrArray` that used `Array` directly. If the compiler was a little bit "lazier" and only calculated the type arguments to `Array` when necessary, then TypeScript could express these correctly.

That's exactly what TypeScript 3.7 introduces. At the "top level" of a type alias, TypeScript will defer resolving type arguments to permit these patterns.

This means that code like the following that was trying to represent JSON...

```
type Json = string | number | boolean | null | JsonObject | JsonArray;
interface JsonObject {
  [property: string]: Json;
}
interface JsonArray extends Array<Json> {}
```

can finally be rewritten without helper interfaces.

```
type Json =
  | string
  | number
  | boolean
  | null
  | { [property: string]: Json }
  | Json[];
```

This new relaxation also lets us recursively reference type aliases in tuples as well. The following code which used to error is now valid TypeScript code.

```
type VirtualNode = string | [string, { [key: string]: any }, ...VirtualNode[]];
const myNode: VirtualNode = [
  "div",
  { id: "parent" },
  ["div", { id: "first-child" }, "I'm the first child"],
  ["div", { id: "second-child" }, "I'm the second child"],
];
```

For more information, you can [read up on the original pull request](#).

--declaration and --allowJs

The `declaration` flag in TypeScript allows us to generate `.d.ts` files (declaration files) from TypeScript source files (i.e. `.ts` and `.tsx` files). These `.d.ts` files are important for a couple of reasons.

First of all, they're important because they allow TypeScript to type-check against other projects without re-checking the original source code. They're also important because they allow TypeScript to interoperate with existing JavaScript libraries that weren't built with TypeScript in mind. Finally, a benefit that is often under-appreciated: both TypeScript *and* JavaScript users can benefit from these files when using editors powered by TypeScript to get things like better auto-completion.

Unfortunately, `declaration` didn't work with the `allowJs` flag which allows mixing TypeScript and JavaScript input files. This was a frustrating limitation because it meant users couldn't use the `declaration` flag when migrating codebases, even if they were JSDoc-annotated. TypeScript 3.7 changes that, and allows the two options to be used together!

The most impactful outcome of this feature might a bit subtle: with TypeScript 3.7, users can write libraries in JSDoc annotated JavaScript and support TypeScript users.

The way that this works is that when using `allowJs`, TypeScript has some best-effort analyses to understand common JavaScript patterns; however, the way that some patterns are expressed in JavaScript don't necessarily look like their equivalents in TypeScript. When `declaration` emit is turned on, TypeScript figures out the best way to transform JSDoc comments and CommonJS exports into valid type declarations and the like in the output `.d.ts` files.

As an example, the following code snippet

```
const assert = require("assert");

module.exports.blurImage = blurImage;

/**
 * Produces a blurred image from an input buffer.
 *
 * @param input {Uint8Array}
 * @param width {number}
 * @param height {number}
 */
function blurImage(input, width, height) {
  const numPixels = width * height * 4;
  assert(input.length === numPixels);
  const result = new Uint8Array(numPixels);

  // TODO

  return result;
}
```

Will produce a `.d.ts` file like

```
/** 
 * Produces a blurred image from an input buffer.
 */
```

```

/* @param input {Uint8Array}
 * @param width {number}
 * @param height {number}
 */
export function blurImage(
  input: Uint8Array,
  width: number,
  height: number
): Uint8Array;

```

This can go beyond basic functions with `@param` tags too, where the following example:

```

/**
 * @callback Job
 * @returns {void}
 */

/** Queues work */
export class Worker {
  constructor(maxDepth = 10) {
    this.started = false;
    this.depthLimit = maxDepth;
    /**
     * NOTE: queued jobs may add more items to queue
     * @type {Job[]}
     */
    this.queue = [];
  }
  /**
   * Adds a work item to the queue
   * @param {Job} work
   */
  push(work) {
    if (this.queue.length + 1 > this.depthLimit) throw new Error("Queue full!");
    this.queue.push(work);
  }
  /**
   * Starts the queue if it has not yet started
   */
  start() {
    if (this.started) return false;
    this.started = true;
    while (this.queue.length) {
      /** @type {Job} */ (this.queue.shift());
    }
    return true;
  }
}

```

will be transformed into the following `.d.ts` file:

```

/**
 * @callback Job
 * @returns {void}
 */
/** Queues work */
export class Worker {
  constructor(maxDepth?: number);
  started: boolean;
  depthLimit: number;
}

```

```
/**  
 * NOTE: queued jobs may add more items to queue  
 * @type {Job[]}  
 */  
queue: Job[];  
/**  
 * Adds a work item to the queue  
 * @param {Job} work  
 */  
push(work: Job): void;  
/**  
 * Starts the queue if it has not yet started  
 */  
start(): boolean;  
}  
export type Job = () => void;
```

Note that when using these flags together, TypeScript doesn't necessarily have to downlevel `.js` files. If you simply want TypeScript to create `.d.ts` files, you can use the `emitDeclarationOnly` compiler option.

For more details, you can [check out the original pull request](#).

The `useDefineForClassFields` Flag and The `declare` Property Modifier

Back when TypeScript implemented public class fields, we assumed to the best of our abilities that the following code

```
class C {  
  foo = 100;  
  bar: string;  
}
```

would be equivalent to a similar assignment within a constructor body.

```
class C {  
  constructor() {  
    this.foo = 100;  
  }  
}
```

Unfortunately, while this seemed to be the direction that the proposal moved towards in its earlier days, there is an extremely strong chance that public class fields will be standardized differently. Instead, the original code sample might need to de-sugar to something closer to the following:

```
class C {  
  constructor() {  
    Object.defineProperty(this, "foo", {  
      enumerable: true,  
      configurable: true,  
      writable: true,  
      value: 100,  
    });  
    Object.defineProperty(this, "bar", {  
      enumerable: true,  
    });  
  }  
}
```

```
    configurable: true,
    writable: true,
    value: void 0,
  });
}
```

While TypeScript 3.7 isn't changing any existing emit by default, we've been rolling out changes incrementally to help users mitigate potential future breakage. We've provided a new flag called `useDefineForClassFields` to enable this emit mode with some new checking logic.

The two biggest changes are the following:

- Declarations are initialized with `Object.defineProperty`.
- Declarations are *always* initialized to `undefined`, even if they have no initializer.

This can cause quite a bit of fallout for existing code that use inheritance. First of all, `set` accessors from base classes won't get triggered - they'll be completely overwritten.

```
class Base {
  set data(value: string) {
    console.log("data changed to " + value);
  }
}

class Derived extends Base {
  // No longer triggers a 'console.log'
  // when using 'useDefineForClassFields'.
  data = 10;
}
```

Secondly, using class fields to specialize properties from base classes also won't work.

```
interface Animal {
  animalStuff: any;
}
interface Dog extends Animal {
  dogStuff: any;
}

class AnimalHouse {
  resident: Animal;
  constructor(animal: Animal) {
    this.resident = animal;
  }
}

class DogHouse extends AnimalHouse {
  // Initializes 'resident' to 'undefined'
  // after the call to 'super()' when
  // using 'useDefineForClassFields'!
  resident: Dog;

  constructor(dog: Dog) {
    super(dog);
  }
}
```

What these two boil down to is that mixing properties with accessors is going to cause issues, and so will re-declaring properties with no initializers.

To detect the issue around accessors, TypeScript 3.7 will now emit `get` / `set` accessors in `.d.ts` files so that in TypeScript can check for overridden accessors.

Code that's impacted by the class fields change can get around the issue by converting field initializers to assignments in constructor bodies.

```
class Base {
  set data(value: string) {
    console.log("data changed to " + value);
  }
}

class Derived extends Base {
  constructor() {
    this.data = 10;
  }
}
```

To help mitigate the second issue, you can either add an explicit initializer or add a `declare` modifier to indicate that a property should have no emit.

```
interface Animal {
  animalStuff: any;
}
interface Dog extends Animal {
  dogStuff: any;
}

class AnimalHouse {
  resident: Animal;
  constructor(animal: Animal) {
    this.resident = animal;
  }
}

class DogHouse extends AnimalHouse {
  declare resident: Dog;
  // ^^^^^^
  // 'resident' now has a 'declare' modifier,
  // and won't produce any output code.

  constructor(dog: Dog) {
    super(dog);
  }
}
```

Currently `useDefineForClassFields` is only available when targeting ES5 and upwards, since `Object.defineProperty` doesn't exist in ES3. To achieve similar checking for issues, you can create a separate project that targets ES5 and uses `noEmit` to avoid a full build.

For more information, you can [take a look at the original pull request for these changes](#).

We strongly encourage users to try the `useDefineForClassFields` flag and report back on our issue tracker or in the comments below. This includes feedback on difficulty of adopting the flag so we can understand how we can make migration easier.

Build-Free Editing with Project References

TypeScript's project references provide us with an easy way to break codebases up to give us faster compiles. Unfortunately, editing a project whose dependencies hadn't been built (or whose output was out of date) meant that the editing experience wouldn't work well.

In TypeScript 3.7, when opening a project with dependencies, TypeScript will automatically use the source `.ts / .tsx` files instead. This means projects using project references will now see an improved editing experience where semantic operations are up-to-date and "just work". You can disable this behavior with the compiler option `disableSourceOfProjectReferenceRedirect` which may be appropriate when working in very large projects where this change may impact editing performance.

You can [read up more about this change by reading up on its pull request](#).

Uncalled Function Checks

A common and dangerous error is to forget to invoke a function, especially if the function has zero arguments or is named in a way that implies it might be a property rather than a function.

```
interface User {
  isAdministrator(): boolean;
  notify(): void;
  doNotDisturb?: boolean;
}

// later...

// Broken code, do not use!
function doAdminThing(user: User) {
  // oops!
  if (user.isAdministrator) {
    sudo();
    editTheConfiguration();
  } else {
    throw new AccessDeniedError("User is not an admin");
  }
}
```

Here, we forgot to call `isAdministrator`, and the code incorrectly allows non-administrator users to edit the configuration!

In TypeScript 3.7, this is identified as a likely error:

```
function doAdminThing(user: User) {
  if (user.isAdministrator) {
    // ~~~~~
    // error! This condition will always return true since the function is always
    // defined.
    // Did you mean to call it instead?
```

This check is a breaking change, but for that reason the checks are very conservative. This error is only issued in `if` conditions, and it is not issued on optional properties, if `strictNullChecks` is off, or if the function is later called within the body of the `if`:

```
interface User {
  isAdminAdministrator(): boolean;
  notify(): void;
  doNotDisturb?: boolean;
}

function issueNotification(user: User) {
  if (user.doNotDisturb) {
    // OK, property is optional
  }
  if (user.notify) {
    // OK, called the function
    user.notify();
  }
}
```

If you intended to test the function without calling it, you can correct the definition of it to include `undefined/null`, or use `!!` to write something like `if (!!user.isAdminAdministrator)` to indicate that the coercion is intentional.

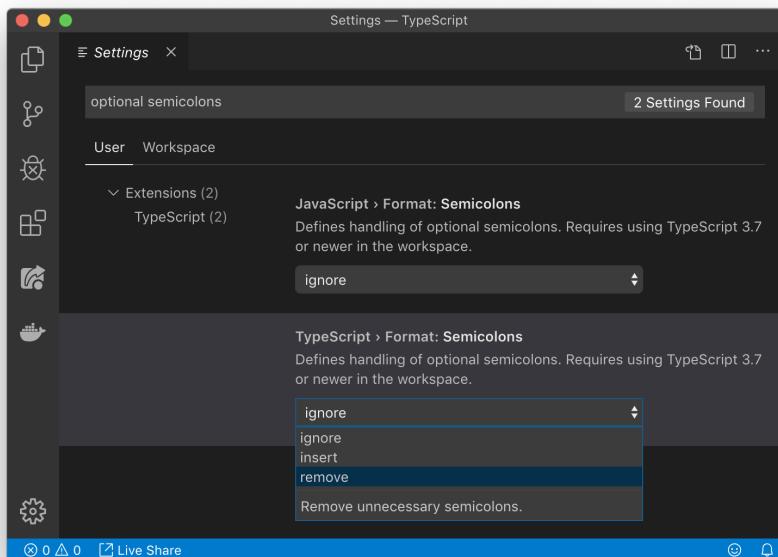
We owe a big thanks to GitHub user [@jwbay](#) who took the initiative to create a [proof-of-concept](#) and iterated to provide us with [the current version](#).

// @ts-nocheck in TypeScript Files

TypeScript 3.7 allows us to add `// @ts-nocheck` comments to the top of TypeScript files to disable semantic checks. Historically this comment was only respected in JavaScript source files in the presence of `checkJs`, but we've expanded support to TypeScript files to make migrations easier for all users.

Semicolon Formatter Option

TypeScript's built-in formatter now supports semicolon insertion and removal at locations where a trailing semicolon is optional due to JavaScript's automatic semicolon insertion (ASI) rules. The setting is available now in [Visual Studio Code Insiders](#), and will be available in Visual Studio 16.4 Preview 2 in the Tools Options menu.



Choosing a value of "insert" or "remove" also affects the format of auto-imports, extracted types, and other generated code provided by TypeScript services. Leaving the setting on its default value of "ignore" makes generated code match the semicolon preference detected in the current file.

3.7 Breaking Changes

DOM Changes

Types in `lib.dom.d.ts` have been updated. These changes are largely correctness changes related to nullability, but impact will ultimately depend on your codebase.

Class Field Mitigations

As mentioned above, TypeScript 3.7 emits `get / set` accessors in `.d.ts` files which can cause breaking changes for consumers on older versions of TypeScript like 3.5 and prior. TypeScript 3.6 users will not be impacted, since that version was future-proofed for this feature.

While not a breakage per se, opting in to the `useDefineForClassFields` flag can cause breakage when:

- overriding an accessor in a derived class with a property declaration

- re-declaring a property declaration with no initializer

To understand the full impact, read the section above on the `useDefineForClassFields` flag.

Function Truthy Checks

As mentioned above, TypeScript now errors when functions appear to be uncalled within `if` statement conditions. An error is issued when a function type is checked in `if` conditions unless any of the following apply:

- the checked value comes from an optional property
- `strictNullChecks` is disabled
- the function is later called within the body of the `if`

Local and Imported Type Declarations Now Conflict

Due to a bug, the following construct was previously allowed in TypeScript:

```
// ./someOtherModule.ts
interface SomeType {
  y: string;
}

// ./myModule.ts
import { SomeType } from "./someOtherModule";
export interface SomeType {
  x: number;
}

function fn(arg: SomeType) {
  console.log(arg.x); // Error! 'x' doesn't exist on 'SomeType'
}
```

Here, `SomeType` appears to originate in both the `import` declaration and the local `interface` declaration. Perhaps surprisingly, inside the module, `SomeType` refers exclusively to the `imported` definition, and the local declaration `SomeType` is only usable when imported from another file. This is very confusing and our review of the very small number of cases of code like this in the wild showed that developers usually thought something different was happening.

In TypeScript 3.7, this is now correctly identified as a [duplicate identifier error](#). The correct fix depends on the original intent of the author and should be addressed on a case-by-case basis. Usually, the naming conflict is unintentional and the best fix is to rename the imported type. If the intent was to augment the imported type, a proper module augmentation should be written instead.

3.7 API Changes

To enable the recursive type alias patterns described above, the `typeArguments` property has been removed from the `TypeReference` interface. Users should instead use the `getTypeArguments` function on `TypeChecker` instances.

[Go to TOC](#)

Type-Only Imports and Export

This feature is something most users may never have to think about; however, if you've hit issues under `isolatedModules`, TypeScript's `transpileModule` API, or Babel, this feature might be relevant.

TypeScript 3.8 adds a new syntax for type-only imports and exports.

```
import type { SomeThing } from "./some-module.js";
export type { SomeThing };
```

`import type` only imports declarations to be used for type annotations and declarations. It *always* gets fully erased, so there's no remnant of it at runtime. Similarly, `export type` only provides an export that can be used for type contexts, and is also erased from TypeScript's output.

It's important to note that classes have a value at runtime and a type at design-time, and the use is context-sensitive. When using `import type` to import a class, you can't do things like extend from it.

```
import type { Component } from "react";

interface ButtonProps {
  // ...
}

class Button extends Component<ButtonProps> {
  // ~~~~~
  // error! 'Component' only refers to a type, but is being used as a value here.
  // ...
}
```

If you've used Flow before, the syntax is fairly similar. One difference is that we've added a few restrictions to avoid code that might appear ambiguous.

```
// Is only 'Foo' a type? Or every declaration in the import?
// We just give an error because it's not clear.

import type Foo, { Bar, Baz } from "some-module";
// ~~~~~
// error! A type-only import can specify a default import or named bindings, but
not both.
```

In conjunction with `import type`, TypeScript 3.8 also adds a new compiler flag to control what happens with imports that won't be utilized at runtime: `importsNotUsedAsValues`. This flag takes 3 different values:

- `remove`: this is today's behavior of dropping these imports. It's going to continue to be the default, and is a non-breaking change.
- `preserve`: this *preserves* all imports whose values are never used. This can cause imports/side-effects to be preserved.

- `error`: this preserves all imports (the same as the `preserve` option), but will error when a value import is only used as a type. This might be useful if you want to ensure no values are being accidentally imported, but still make side-effect imports explicit.

For more information about the feature, you can [take a look at the pull request](#), and [relevant changes](#) around broadening where imports from an `import type` declaration can be used.

ECMAScript Private Fields

TypeScript 3.8 brings support for ECMAScript's private fields, part of the [stage-3 class fields proposal](#).

```
class Person {
  #name: string;

  constructor(name: string) {
    this.#name = name;
  }

  greet() {
    console.log(`Hello, my name is ${this.#name}!`);
  }
}

let jeremy = new Person("Jeremy Bearimy");

jeremy.#name;
// ~~~~~
// Property '#name' is not accessible outside class 'Person'
// because it has a private identifier.
```

Unlike regular properties (even ones declared with the `private` modifier), private fields have a few rules to keep in mind. Some of them are:

- Private fields start with a `#` character. Sometimes we call these *private names*.
 - Every private field name is uniquely scoped to its containing class.
 - TypeScript accessibility modifiers like `public` or `private` can't be used on private fields.
 - Private fields can't be accessed or even detected outside of the containing class - even by JS users!
- Sometimes we call this *hard privacy*.

Apart from "hard" privacy, another benefit of private fields is that uniqueness we just mentioned. For example, regular property declarations are prone to being overwritten in subclasses.

```
class C {
  foo = 10;

  cHelper() {
    return this.foo;
  }
}

class D extends C {
  foo = 20;

  dHelper() {
    return this.foo;
```

```

    }
}

let instance = new D();
// 'this.foo' refers to the same property on each instance.
console.log(instance.cHelper()); // prints '20'
console.log(instance.dHelper()); // prints '20'

```

With private fields, you'll never have to worry about this, since each field name is unique to the containing class.

```

class C {
  #foo = 10;

  cHelper() {
    return this.#foo;
  }
}

class D extends C {
  #foo = 20;

  dHelper() {
    return this.#foo;
  }
}

let instance = new D();
// 'this.#foo' refers to a different field within each class.
console.log(instance.cHelper()); // prints '10'
console.log(instance.dHelper()); // prints '20'

```

Another thing worth noting is that accessing a private field on any other type will result in a `TypeError`!

```

class Square {
  #sideLength: number;

  constructor(sideLength: number) {
    this.#sideLength = sideLength;
  }

  equals(other: any) {
    return this.#sideLength === other.#sideLength;
  }
}

const a = new Square(100);
const b = { sideLength: 100 };

// Boom!
// TypeError: attempted to get private field on non-instance
// This fails because 'b' is not an instance of 'Square'.
console.log(a.equals(b));

```

Finally, for any plain `.js` file users, private fields *always* have to be declared before they're assigned to.

```

class C {
  // No declaration for '#foo'
  // :(

```

```
constructor(foo: number) {
  // SyntaxError!
  // '#foo' needs to be declared before writing to it.
  this.#foo = foo;
}
```

JavaScript has always allowed users to access undeclared properties, whereas TypeScript has always required declarations for class properties. With private fields, declarations are always needed regardless of whether we're working in `.js` or `.ts` files.

```
class C {
  /** @type {number} */
  #foo;

  constructor(foo: number) {
    // This works.
    this.#foo = foo;
}
```

For more information about the implementation, you can [check out the original pull request](#)

Which should I use?

We've already received many questions on which type of privates you should use as a TypeScript user: most commonly, "should I use the `private` keyword, or ECMAScript's hash/pound (#) private fields?" It depends!

When it comes to properties, TypeScript's `private` modifiers are fully erased - that means that at runtime, it acts entirely like a normal property and there's no way to tell that it was declared with a `private` modifier. When using the `private` keyword, privacy is only enforced at compile-time/design-time, and for JavaScript consumers it's entirely intent-based.

```
class C {
  private foo = 10;

  // This is an error at compile time,
  // but when TypeScript outputs .js files,
  // it'll run fine and print '10'.
  console.log(new C().foo); // prints '10'
  //
  // ~~~
  // error! Property 'foo' is private and only accessible within class 'C'.

  // TypeScript allows this at compile-time
  // as a "work-around" to avoid the error.
  console.log(new C()["foo"]); // prints '10'
```

The upside is that this sort of "soft privacy" can help your consumers temporarily work around not having access to some API, and also works in any runtime.

On the other hand, ECMAScript's # privates are completely inaccessible outside of the class.

```

class C {
  #foo = 10;
}

console.log(new C().#foo); // SyntaxError
// ~~~
// TypeScript reports an error *and*
// this won't work at runtime!

console.log(new C()["#foo"]); // prints undefined
// ~~~~~
// TypeScript reports an error under 'noImplicitAny',
// and this prints 'undefined'.

```

This hard privacy is really useful for strictly ensuring that nobody can take use of any of your internals. If you're a library author, removing or renaming a private field should never cause a breaking change.

As we mentioned, another benefit is that subclassing can be easier with ECMAScript's `#` privates because they *really* are private. When using ECMAScript `#` private fields, no subclass ever has to worry about collisions in field naming. When it comes to TypeScript's `private` property declarations, users still have to be careful not to trample over properties declared in superclasses.

One more thing to think about is where you intend for your code to run. TypeScript currently can't support this feature unless targeting ECMAScript 2015 (ES6) targets or higher. This is because our downleveled implementation uses `WeakMap`s to enforce privacy, and `WeakMap`s can't be polyfilled in a way that doesn't cause memory leaks. In contrast, TypeScript's `private`-declared properties work with all targets - even ECMAScript 3!

A final consideration might be speed: `private` properties are no different from any other property, so accessing them is as fast as any other property access no matter which runtime you target. In contrast, because `#` private fields are downleveled using `WeakMap`s, they may be slower to use. While some runtimes might optimize their actual implementations of `#` private fields, and even have speedy `WeakMap` implementations, that might not be the case in all runtimes.

`export * as ns` Syntax

It's often common to have a single entry-point that exposes all the members of another module as a single member.

```

import * as utilities from "./utilities.js";
export { utilities };

```

This is so common that ECMAScript 2020 recently added a new syntax to support this pattern!

```

export * as utilities from "./utilities.js";

```

This is a nice quality-of-life improvement to JavaScript, and TypeScript 3.8 implements this syntax. When your module target is earlier than `es2020`, TypeScript will output something along the lines of the first code snippet.

Top-Level `await`

TypeScript 3.8 provides support for a handy upcoming ECMAScript feature called "top-level `await`".

JavaScript users often introduce an `async` function in order to use `await`, and then immediately called the function after defining it.

```
async function main() {
  const response = await fetch(...);
  const greeting = await response.text();
  console.log(greeting);
}

main().catch((e) => console.error(e));
```

This is because previously in JavaScript (along with most other languages with a similar feature), `await` was only allowed within the body of an `async` function. However, with top-level `await`, we can use `await` at the top level of a module.

```
const response = await fetch(...);
const greeting = await response.text();
console.log(greeting);

// Make sure we're a module
export {};
```

Note there's a subtlety: top-level `await` only works at the top level of a *module*, and files are only considered modules when TypeScript finds an `import` or an `export`. In some basic cases, you might need to write out `export {}` as some boilerplate to make sure of this.

Top level `await` may not work in all environments where you might expect at this point. Currently, you can only use top level `await` when the `target` compiler option is `es2017` or above, and `module` is `esnext` or `system`. Support within several environments and bundlers may be limited or may require enabling experimental support.

For more information on our implementation, you can [check out the original pull request](#).

es2020 for `target` and `module`

TypeScript 3.8 supports `es2020` as an option for `module` and `target`. This will preserve newer ECMAScript 2020 features like optional chaining, nullish coalescing, `export * as ns`, and dynamic `import(...)` syntax. It also means `bignum` literals now have a stable `target` below `esnext`.

JSDoc Property Modifiers

TypeScript 3.8 supports JavaScript files by turning on the `allowJs` flag, and also supports *type-checking* those JavaScript files via the `checkJs` option or by adding a `// @ts-check` comment to the top of your `.js` files.

Because JavaScript files don't have dedicated syntax for type-checking, TypeScript leverages JSDoc. TypeScript 3.8 understands a few new JSDoc tags for properties.

First are the accessibility modifiers: `@public`, `@private`, and `@protected`. These tags work exactly like `public`, `private`, and `protected` respectively work in TypeScript.

```
// @ts-check

class Foo {
  constructor() {
    /** @private */
    this.stuff = 100;
  }

  printStuff() {
    console.log(this.stuff);
  }
}

new Foo().stuff;
// ~~~~~
// error! Property 'stuff' is private and only accessible within class 'Foo'.
```

- `@public` is always implied and can be left off, but means that a property can be reached from anywhere.
- `@private` means that a property can only be used within the containing class.
- `@protected` means that a property can only be used within the containing class, and all derived subclasses, but not on dissimilar instances of the containing class.

Next, we've also added the `@readonly` modifier to ensure that a property is only ever written to during initialization.

```
// @ts-check

class Foo {
  constructor() {
    /** @readonly */
    this.stuff = 100;
  }

  writeToStuff() {
    this.stuff = 200;
    // ~~~~~
    // Cannot assign to 'stuff' because it is a read-only property.
  }
}

new Foo().stuff++;
// ~~~~~
// Cannot assign to 'stuff' because it is a read-only property.
```

Better Directory Watching on Linux and `watchOptions`

TypeScript 3.8 ships a new strategy for watching directories, which is crucial for efficiently picking up changes to `node_modules`.

For some context, on operating systems like Linux, TypeScript installs directory watchers (as opposed to file watchers) on `node_modules` and many of its subdirectories to detect changes in dependencies. This is because the number of available file watchers is often eclipsed by the of files in `node_modules`, whereas there are way fewer directories to track.

Older versions of TypeScript would *immediately* install directory watchers on folders, and at startup that would be fine; however, during an npm install, a lot of activity will take place within `node_modules` and that can overwhelm TypeScript, often slowing editor sessions to a crawl. To prevent this, TypeScript 3.8 waits slightly before installing directory watchers to give these highly volatile directories some time to stabilize.

Because every project might work better under different strategies, and this new approach might not work well for your workflows, TypeScript 3.8 introduces a new `watchOptions` field in `tsconfig.json` and `jsconfig.json` which allows users to tell the compiler/language service which watching strategies should be used to keep track of files and directories.

```
{
  // Some typical compiler options
  "compilerOptions": {
    "target": "es2020",
    "moduleResolution": "node"
    // ...
  },

  // NEW: Options for file/directory watching
  "watchOptions": {
    // Use native file system events for files and directories
    "watchFile": "useFsEvents",
    "watchDirectory": "useFsEvents",

    // Poll files for updates more frequently
    // when they're updated a lot.
    "fallbackPolling": "dynamicPriority"
  }
}
```

`watchOptions` contains 4 new options that can be configured:

- `watchFile` : the strategy for how individual files are watched. This can be set to
 - `fixedPollingInterval` : Check every file for changes several times a second at a fixed interval.
 - `priorityPollingInterval` : Check every file for changes several times a second, but use heuristics to check certain types of files less frequently than others.
 - `dynamicPriorityPolling` : Use a dynamic queue where less-frequently modified files will be checked less often.
 - `useFsEvents` (the default): Attempt to use the operating system/file system's native events for file changes.
 - `useFsEventsOnParentDirectory` : Attempt to use the operating system/file system's native events to listen for changes on a file's containing directories. This can use fewer file watchers, but might be less accurate.

- `watchDirectory`: the strategy for how entire directory trees are watched under systems that lack recursive file-watching functionality. This can be set to:
 - `fixedPollingInterval`: Check every directory for changes several times a second at a fixed interval.
 - `dynamicPriorityPolling`: Use a dynamic queue where less-frequently modified directories will be checked less often.
 - `useFsEvents` (the default): Attempt to use the operating system/file system's native events for directory changes.
- `fallbackPolling`: when using file system events, this option specifies the polling strategy that gets used when the system runs out of native file watchers and/or doesn't support native file watchers. This can be set to
 - `fixedPollingInterval`: (See above.)
 - `priorityPollingInterval`: (See above.)
 - `dynamicPriorityPolling`: (See above.)
 - `synchronousWatchDirectory`: Disable deferred watching on directories. Deferred watching is useful when lots of file changes might occur at once (e.g. a change in `node_modules` from running `npm install`), but you might want to disable it with this flag for some less-common setups.

For more information on these changes, [head over to GitHub to see the pull request](#) to read more.

"Fast and Loose" Incremental Checking

TypeScript 3.8 introduces a new compiler option called `assumeChangesOnlyAffectDirectDependencies`. When this option is enabled, TypeScript will avoid rechecking/rebuilding all truly possibly-affected files, and only recheck/rebuild files that have changed as well as files that directly import them.

For example, consider a file `fileD.ts` that imports `fileC.ts` that imports `fileB.ts` that imports `fileA.ts` as follows:

```
fileA.ts <- fileB.ts <- fileC.ts <- fileD.ts
```

In `--watch` mode, a change in `fileA.ts` would typically mean that TypeScript would need to at least re-check `fileB.ts`, `fileC.ts`, and `fileD.ts`. Under `assumeChangesOnlyAffectDirectDependencies`, a change in `fileA.ts` means that only `fileA.ts` and `fileB.ts` need to be re-checked.

In a codebase like Visual Studio Code, this reduced rebuild times for changes in certain files from about 14 seconds to about 1 second. While we don't necessarily recommend this option for all codebases, you might be interested if you have an extremely large codebase and are willing to defer full project errors until later (e.g. a dedicated build via a `tsconfig.fullbuild.json` or in CI).

For more details, you can [see the original pull request](#).

[Go to TOC](#)

Improvements in Inference and `Promise.all`

Recent versions of TypeScript (around 3.7) have had updates to the declarations of functions like `Promise.all` and `Promise.race`. Unfortunately, that introduced a few regressions, especially when mixing in values with `null` or `undefined`.

```
interface Lion {
  roar(): void;
}

interface Seal {
  singKissFromARose(): void;
}

async function visitZoo(
  lionExhibit: Promise<Lion>,
  sealExhibit: Promise<Seal | undefined>
) {
  let [lion, seal] = await Promise.all([lionExhibit, sealExhibit]);
  lion.roar(); // uh oh
  // ~~~
  // Object is possibly 'undefined'.
}
```

This is strange behavior! The fact that `sealExhibit` contained an `undefined` somehow poisoned type of `lion` to include `undefined`.

Thanks to a [pull request](#) from [Jack Bates](#), this has been fixed with improvements in our inference process in TypeScript 3.9. The above no longer errors. If you've been stuck on older versions of TypeScript due to issues around `Promise`s, we encourage you to give 3.9 a shot!

What About the `awaited` Type?

If you've been following our issue tracker and design meeting notes, you might be aware of some work around a [new type operator called `awaited`](#). This goal of this type operator is to accurately model the way that `Promise` unwrapping works in JavaScript.

We initially anticipated shipping `awaited` in TypeScript 3.9, but as we've run early TypeScript builds with existing codebases, we've realized that the feature needs more design work before we can roll it out to everyone smoothly. As a result, we've decided to pull the feature out of our main branch until we feel more confident. We'll be experimenting more with the feature, but we won't be shipping it as part of this release.

Speed Improvements

TypeScript 3.9 ships with many new speed improvements. Our team has been focusing on performance after observing extremely poor editing/compilation speed with packages like `material-ui` and `styled-components`. We've dived deep here, with a series of different pull requests that optimize certain pathological cases involving large unions, intersections, conditional types, and mapped types.

- <https://github.com/microsoft/TypeScript/pull/36576>

- <https://github.com/microsoft/TypeScript/pull/36590>
- <https://github.com/microsoft/TypeScript/pull/36607>
- <https://github.com/microsoft/TypeScript/pull/36622>
- <https://github.com/microsoft/TypeScript/pull/36754>
- <https://github.com/microsoft/TypeScript/pull/36696>

Each of these pull requests gains about a 5-10% reduction in compile times on certain codebases. In total, we believe we've achieved around a 40% reduction in material-ui's compile time!

We also have some changes to file renaming functionality in editor scenarios. We heard from the Visual Studio Code team that when renaming a file, just figuring out which import statements needed to be updated could take between 5 to 10 seconds. TypeScript 3.9 addresses this issue by [changing the internals of how the compiler and language service caches file lookups](#).

While there's still room for improvement, we hope this work translates to a snappier experience for everyone!

// @ts-expect-error Comments

Imagine that we're writing a library in TypeScript and we're exporting some function called `doStuff` as part of our public API. The function's types declare that it takes two `string`s so that other TypeScript users can get type-checking errors, but it also does a runtime check (maybe only in development builds) to give JavaScript users a helpful error.

```
function doStuff(abc: string, xyz: string) {
  assert(typeof abc === "string");
  assert(typeof xyz === "string");

  // do some stuff
}
```

So TypeScript users will get a helpful red squiggle and an error message when they misuse this function, and JavaScript users will get an assertion error. We'd like to test this behavior, so we'll write a unit test.

```
expect(() => {
  doStuff(123, 456);
}).toThrow();
```

Unfortunately if our tests are written in TypeScript, TypeScript will give us an error!

```
doStuff(123, 456);
//      ~~~
// error: Type 'number' is not assignable to type 'string'.
```

That's why TypeScript 3.9 brings a new feature: `// @ts-expect-error` comments. When a line is preceded by a `// @ts-expect-error` comment, TypeScript will suppress that error from being reported; but if there's no error, TypeScript will report that `// @ts-expect-error` wasn't necessary.

As a quick example, the following code is okay

```
// @ts-expect-error
console.log(47 * "octopus");
```

while the following code

```
// @ts-expect-error
console.log(1 + 1);
```

results in the error

```
Unused '@ts-expect-error' directive.
```

We'd like to extend a big thanks to [Josh Goldberg](#), the contributor who implemented this feature. For more information, you can take a look at the [the ts-expect-error pull request](#).

ts-ignore or ts-expect-error?

In some ways `// @ts-expect-error` can act as a suppression comment, similar to `// @ts-ignore`. The difference is that `// @ts-ignore` will do nothing if the following line is error-free.

You might be tempted to switch existing `// @ts-ignore` comments over to `// @ts-expect-error`, and you might be wondering which is appropriate for future code. While it's entirely up to you and your team, we have some ideas of which to pick in certain situations.

Pick `ts-expect-error` if:

- you're writing test code where you actually want the type system to error on an operation
- you expect a fix to be coming in fairly quickly and you just need a quick workaround
- you're in a reasonably-sized project with a proactive team that wants to remove suppression comments as soon affected code is valid again

Pick `ts-ignore` if:

- you have a larger project and new errors have appeared in code with no clear owner
- you are in the middle of an upgrade between two different versions of TypeScript, and a line of code errors in one version but not another.
- you honestly don't have the time to decide which of these options is better.

Uncalled Function Checks in Conditional Expressions

In TypeScript 3.7 we introduced *uncalled function checks* to report an error when you've forgotten to call a function.

```
function hasImportantPermissions(): boolean {
    // ...
}

// Oops!
if (hasImportantPermissions) {
    // ~~~~~
    // This condition will always return true since the function is always defined.
```

```
// Did you mean to call it instead?
deleteAllTheImportantFiles();
```

However, this error only applied to conditions in `if` statements. Thanks to a [pull request](#) from [Alexander Tarasyuk](#), this feature is also now supported in ternary conditionals (i.e. the `cond ? trueExpr : falseExpr` syntax).

```
declare function listFilesOfDirectory(dirPath: string): string[];
declare function isDirectory(): boolean;

function getAllFiles(startFileName: string) {
    const result: string[] = [];
    traverse(startFileName);
    return result;

    function traverse(currentPath: string) {
        return isDirectory
            ? // ~~~~~
            // This condition will always return true
            // since the function is always defined.
            // Did you mean to call it instead?
            listFilesOfDirectory(currentPath).forEach(traverse)
            : result.push(currentPath);
    }
}
```

<https://github.com/microsoft/TypeScript/issues/36048>

Editor Improvements

The TypeScript compiler not only powers the TypeScript editing experience in most major editors, it also powers the JavaScript experience in the Visual Studio family of editors and more. Using new TypeScript/JavaScript functionality in your editor will differ depending on your editor, but

- Visual Studio Code supports [selecting different versions of TypeScript](#). Alternatively, there's the [JavaScript/TypeScript Nightly Extension](#) to stay on the bleeding edge (which is typically very stable).
- Visual Studio 2017/2019 have [the SDK installers above] and [MSBuild installs](#).
- Sublime Text 3 supports [selecting different versions of TypeScript](#)

CommonJS Auto-Imports in JavaScript

One great new improvement is in auto-imports in JavaScript files using CommonJS modules.

In older versions, TypeScript always assumed that regardless of your file, you wanted an ECMAScript-style import like

```
import * as fs from "fs";
```

However, not everyone is targeting ECMAScript-style modules when writing JavaScript files. Plenty of users still use CommonJS-style `require(...)` imports like so

```
const fs = require("fs");
```

TypeScript now automatically detects the types of imports you're using to keep your file's style clean and consistent.



For more details on the change, see [the corresponding pull request](#).

Code Actions Preserve Newlines

TypeScript's refactorings and quick fixes often didn't do a great job of preserving newlines. As a really basic example, take the following code.

```
const maxValue = 100;

/*start*/
for (let i = 0; i <= maxValue; i++) {
    // First get the squared value.
    let square = i ** 2,
        // Now print the squared value.
        console.log(square);
}
/*end*/
```

If we highlighted the range from `/*start*/` to `/*end*/` in our editor to extract to a new function, we'd end up with code like the following.

```
const maxValue = 100;

printSquares();

function printSquares() {
    for (let i = 0; i <= maxValue; i++) {
        // First get the squared value.
        let square = i ** 2;
        // Now print the squared value.
        console.log(square);
    }
}
```

```
const maxValue = 100;

for (let i = 0; i <= maxValue; i++) {
    // First get the squared value.
    let square = i ** 2;

    // Now print the squared value.
    console.log(square);
}
```

Extracting the for loop to a function in older versions of TypeScript. A newline is not preserved.

That's not ideal - we had a blank line between each statement in our `for` loop, but the refactoring got rid of it! TypeScript 3.9 does a little more work to preserve what we write.

```
const maxValue = 100;

printSquares();

function printSquares() {
    for (let i = 0; i <= maxValue; i++) {
        // First get the squared value.
        let square = i ** 2;

        // Now print the squared value.
        console.log(square);
    }
}
```

```
const maxValue = 100;

for (let i = 0; i <= maxValue; i++) {
    // First get the squared value.
    let square = i ** 2;

    // Now print the squared value.
    console.log(square);
}
```

Extracting the for loop to a function in TypeScript 3.9. A newline is preserved.

You can see more about the implementation [in this pull request](#)

Quick Fixes for Missing Return Expressions

There are occasions where we might forget to return the value of the last statement in a function, especially when adding curly braces to arrow functions.

```
// before
let f1 = () => 42;

// oops - not the same!
let f2 = () => {
  42;
};
```

Thanks to a [pull request](#) from community member [Wenlu Wang](#), TypeScript can provide a quick-fix to add missing `return` statements, remove curly braces, or add parentheses to arrow function bodies that look suspiciously like object literals.

```
interface Person {
  name: string;
  age: number;
}

function sortByAge(people: Person[]) {
  people.sort([(a, b) => a.age - b.age]);
}
```

TypeScript fixing an error where no expression is returned by adding a 'return' statement or removing curly braces.

Support for "Solution Style" `tsconfig.json` Files

Editors need to figure out which configuration file a file belongs to so that it can apply the appropriate options and figure out which other files are included in the current "project". By default, editors powered by TypeScript's language server do this by walking up each parent directory to find a `tsconfig.json`.

One case where this slightly fell over is when a `tsconfig.json` simply existed to reference other `tsconfig.json` files.

```
// tsconfig.json
{
  "files": [],
  "references": [
    { "path": "./tsconfig.shared.json" },
    { "path": "./tsconfig.frontend.json" },
    { "path": "./tsconfig.backend.json" }
  ]
}
```

This file that really does nothing but manage other project files is often called a "solution" in some environments. Here, none of these `tsconfig.*.json` files get picked up by the server, but we'd really like the language server to understand that the current `.ts` file probably belongs to one of the mentioned projects in this root `tsconfig.json`.

TypeScript 3.9 adds support to editing scenarios for this configuration. For more details, take a look at [the pull request that added this functionality](#).

Breaking Changes

Parsing Differences in Optional Chaining and Non-Null Assertions

TypeScript recently implemented the optional chaining operator, but we've received user feedback that the behavior of optional chaining (`?.`) with the non-null assertion operator (`!`) is extremely counter-intuitive.

Specifically, in previous versions, the code

```
foo?.bar!.baz;
```

was interpreted to be equivalent to the following JavaScript.

```
(foo?.bar).baz;
```

In the above code the parentheses stop the "short-circuiting" behavior of optional chaining, so if `foo` is `undefined`, accessing `baz` will cause a runtime error.

The Babel team who pointed this behavior out, and most users who provided feedback to us, believe that this behavior is wrong. We do too! The thing we heard the most was that the `!` operator should just "disappear" since the intent was to remove `null` and `undefined` from the type of `bar`.

In other words, most people felt that the original snippet should be interpreted as

```
foo?.bar.baz;
```

which just evaluates to `undefined` when `foo` is `undefined`.

This is a breaking change, but we believe most code was written with the new interpretation in mind. Users who want to revert to the old behavior can add explicit parentheses around the left side of the `!` operator.

```
foo?.bar!.baz;
```

Stricter Checks on Intersections and Optional Properties

Type 'A & B' is not assignable to type 'C'.
 Types of property 'a' are incompatible.
 Type 'number' is not assignable to type 'boolean | undefined'.

For more information on this change, [see the corresponding pull request](#).

Intersections Reduced By Discriminant Properties

There are a few cases where you might end up with types that describe values that just don't exist. For example

```
declare function smushObjects<T, U>(x: T, y: U): T & U;

interface Circle {
  kind: "circle";
  radius: number;
}

interface Square {
  kind: "square";
  sideLength: number;
}

declare let x: Circle;
declare let y: Square;

let z = smushObjects(x, y);
console.log(z.kind);
```

This code is slightly weird because there's really no way to create an intersection of a `Circle` and a `Square` - they have two incompatible `kind` fields. In previous versions of TypeScript, this code was allowed and the type of `kind` itself was `never` because `"circle" & "square"` described a set of values that could `never` exist.

In TypeScript 3.9, the type system is more aggressive here - it notices that it's impossible to intersect `Circle` and `Square` because of their `kind` properties. So instead of collapsing the type of `z.kind` to `never`, it collapses the type of `z` itself (`Circle & Square`) to `never`. That means the above code now errors with:

Property 'kind' does not exist on type 'never'.

Most of the breaks we observed seem to correspond with slightly incorrect type declarations. For more details, [see the original pull request](#).

Getters/Setters are No Longer Enumerable

In older versions of TypeScript, `get` and `set` accessors in classes were emitted in a way that made them enumerable; however, this wasn't compliant with the ECMAScript specification which states that they must be non-enumerable. As a result, TypeScript code that targeted ES5 and ES2015 could differ in behavior.

Thanks to a [pull request](#) from GitHub user `pathurs`, TypeScript 3.9 now conforms more closely with ECMAScript in this regard.

Type Parameters That Extend `any` No Longer Act as `any`

In previous versions of TypeScript, a type parameter constrained to `any` could be treated as `any`.

```
function foo<T extends any>(arg: T) {
    arg.spfjgerijghoied; // no error!
}
```

This was an oversight, so TypeScript 3.9 takes a more conservative approach and issues an error on these questionable operations.

```
function foo<T extends any>(arg: T) {
    arg.spfjgerijghoied;
    // ~~~~~
    // Property 'spfjgerijghoied' does not exist on type 'T'.
}
```

`export *` is Always Retained

In previous TypeScript versions, declarations like `export * from "foo"` would be dropped in our JavaScript output if `foo` didn't export any values. This sort of emit is problematic because it's type-directed and can't be emulated by Babel. TypeScript 3.9 will always emit these `export *` declarations. In practice, we don't expect this to break much existing code.

More libdom.d.ts refinements

We are continuing to move more of TypeScript's built-in .d.ts library (lib.d.ts and family) to be generated from Web IDL files directly from the DOM specification. As a result some vendor-specific types related to media access have been removed.

Adding this file to an ambient `*.d.ts` to your project will bring them back:

```
interface AudioTrackList {
    [Symbol.iterator](): IterableIterator<AudioTrack>;
}

interface HTMLVideoElement {
    readonly audioTracks: AudioTrackList;

    msFrameStep(forward: boolean): void;
    msInsertVideoEffect(activatableClassId: string, effectRequired: boolean,
        config?: any): void;
    msSetVideoRectangle(left: number, top: number, right: number, bottom: number): void;
    webkitEnterFullScreen(): void;
    webkitEnterFullscreen(): void;
    webkitExitFullScreen(): void;
    webkitExitFullscreen(): void;

    msHorizontalMirror: boolean;
    readonly msIsLayoutOptimalForPlayback: boolean;
    readonly msIsStereo3D: boolean;
    msStereo3DPackingMode: string;
    msStereo3DRenderMode: string;
    msZoom: boolean;
}
```

```
onMSVideoFormatChanged: ((this: HTMLVideoElement, ev: Event) => any) | null;
onMSVideoFrameStepCompleted: ((this: HTMLVideoElement, ev: Event) => any) | null;
onMSVideoOptimalLayoutChanged: ((this: HTMLVideoElement, ev: Event) => any) | null;
webkitDisplayingFullscreen: boolean;
webkitSupportsFullscreen: boolean;
}

interface MediaError {
  readonly msExtendedCode: number;
  readonly MS_MEDIA_ERR_ENCRYPTED: number;
}
```

[Go to TOC](#)

Variadic Tuple Types

Consider a function in JavaScript called `concat` that takes two array or tuple types and concatenates them together to make a new array.

```
function concat(arr1, arr2) {
  return [...arr1, ...arr2];
}
```

Also consider `tail`, that takes an array or tuple, and returns all elements but the first.

```
function tail(arg) {
  const [_ , ...result] = arg;
  return result;
}
```

How would we type either of these in TypeScript?

For `concat`, the only valid thing we could do in older versions of the language was to try and write some overloads.

```
function concat(arr1: [], arr2: []): [];
function concat<A>(arr1: [A], arr2: []): [A];
function concat<A, B>(arr1: [A, B], arr2: []): [A, B];
function concat<A, B, C>(arr1: [A, B, C], arr2: []): [A, B, C];
function concat<A, B, C, D>(arr1: [A, B, C, D], arr2: []): [A, B, C, D];
function concat<A, B, C, D, E>(arr1: [A, B, C, D, E], arr2: []): [A, B, C, D, E];
function concat<A, B, C, D, E, F>(arr1: [A, B, C, D, E, F], arr2: []): [A, B, C, D, E, F];
```

Uh...okay, that's...seven overloads for when the second array is always empty. Let's add some for when `arr2` has one argument.

```
function concat<A2>(arr1: [], arr2: [A2]): [A2];
function concat<A1, A2>(arr1: [A1], arr2: [A2]): [A1, A2];
function concat<A1, B1, A2>(arr1: [A1, B1], arr2: [A2]): [A1, B1, A2];
function concat<A1, B1, C1, A2>(arr1: [A1, B1, C1], arr2: [A2]): [A1, B1, C1, A2];
function concat<A1, B1, C1, D1, A2>(arr1: [A1, B1, C1, D1], arr2: [A2]): [A1, B1, C1, D1, A2];
function concat<A1, B1, C1, D1, E1, A2>(arr1: [A1, B1, C1, D1, E1], arr2: [A2]): [A1, B1, C1, D1, E1, A2];
function concat<A1, B1, C1, D1, E1, F1, A2>(arr1: [A1, B1, C1, D1, E1, F1], arr2: [A2]): [A1, B1, C1, D1, E1, F1, A2];
```

We hope it's clear that this is getting unreasonable. Unfortunately, you'd also end up with the same sorts of issues typing a function like `tail`.

This is another case of what we like to call "death by a thousand overloads", and it doesn't even solve the problem generally. It only gives correct types for as many overloads as we care to write. If we wanted to make a catch-all case, we'd need an overload like the following:

```
function concat<T, U>(arr1: T[], arr2: U[]): Array<T | U>;
```

But that signature doesn't encode anything about the lengths of the input, or the order of the elements, when using tuples.

TypeScript 4.0 brings two fundamental changes, along with inference improvements, to make typing these possible.

The first change is that spreads in tuple type syntax can now be generic. This means that we can represent higher-order operations on tuples and arrays even when we don't know the actual types we're operating over. When generic spreads are instantiated (or, replaced with a real type) in these tuple types, they can produce other sets of array and tuple types.

For example, that means we can type function like `tail`, without our "death by a thousand overloads" issue.

```
function tail<T extends any[]>(arr: readonly [any, ...T]) {
  const [_ignored, ...rest] = arr;
  return rest;
}

const myTuple = [1, 2, 3, 4] as const;
const myArray = ["hello", "world"];

const r1 = tail(myTuple);
// ^?

const r2 = tail([...myTuple, ...myArray] as const);
// ^?
```

The second change is that rest elements can occur anywhere in a tuple - not just at the end!

```
type Strings = [string, string];
type Numbers = [number, number];

type StrStrNumNumBool = [...Strings, ...Numbers, boolean];
```

Previously, TypeScript would issue an error like the following:

A rest element must be last in a tuple type.

But with TypeScript 4.0, this restriction is relaxed.

Note that in cases when we spread in a type without a known length, the resulting type becomes unbounded as well, and all the following elements factor into the resulting rest element type.

```
type Strings = [string, string];
type Numbers = number[];

type Unbounded = [...Strings, ...Numbers, boolean];
```

By combining both of these behaviors together, we can write a single well-typed signature for `concat`:

```
type Arr = readonly any[];
function concat<T extends Arr, U extends Arr>(arr1: T, arr2: U): [...T, ...U] {
    return [...arr1, ...arr2];
}
```

While that one signature is still a bit lengthy, it's just one signature that doesn't have to be repeated, and it gives predictable behavior on all arrays and tuples.

This functionality on its own is great, but it shines in more sophisticated scenarios too. For example, consider a function to [partially apply arguments](#) called `partialCall`. `partialCall` takes a function - let's call it `f` - along with the initial few arguments that `f` expects. It then returns a new function that takes any other arguments that `f` still needs, and calls `f` when it receives them.

```
function partialCall(f, ...headArgs) {
    return (...tailArgs) => f(...headArgs, ...tailArgs);
}
```

TypeScript 4.0 improves the inference process for rest parameters and rest tuple elements so that we can type this and have it "just work".

```
type Arr = readonly unknown[];
function partialCall<T extends Arr, U extends Arr, R>(
    f: (...args: [...T, ...U]) => R,
    ...headArgs: T
) {
    return (...tailArgs: U) => f(...headArgs, ...tailArgs);
}
```

In this case, `partialCall` understands which parameters it can and can't initially take, and returns functions that appropriately accept and reject anything left over.

```
// @errors: 2345 2554 2554 2345
type Arr = readonly unknown[];

function partialCall<T extends Arr, U extends Arr, R>(
    f: (...args: [...T, ...U]) => R,
    ...headArgs: T
) {
    return (...tailArgs: U) => f(...headArgs, ...tailArgs);
}
// ---cut---
const foo = (x: string, y: number, z: boolean) => {};
const f1 = partialCall(foo, 100);
const f2 = partialCall(foo, "hello", 100, true, "oops");

// This works!
const f3 = partialCall(foo, "hello");
//   ^
// What can we do with f3 now?

// Works!
f3(123, true);
```

```
f3();
f3(123, "hello");
```

Variadic tuple types enable a lot of new exciting patterns, especially around function composition. We expect we may be able to leverage it to do a better job type-checking JavaScript's built-in `bind` method. A handful of other inference improvements and patterns also went into this, and if you're interested in learning more, you can take a look at [the pull request](#) for variadic tuples.

Labeled Tuple Elements

Improving the experience around tuple types and parameter lists is important because it allows us to get strongly typed validation around common JavaScript idioms - really just slicing and dicing argument lists and passing them to other functions. The idea that we can use tuple types for rest parameters is one place where this is crucial.

For example, the following function that uses a tuple type as a rest parameter...

```
function foo(...args: [string, number]): void {
  // ...
}
```

...should appear no different from the following function...

```
function foo(arg0: string, arg1: number): void {
  // ...
}
```

...for any caller of `foo`.

```
// @errors: 2554
function foo(arg0: string, arg1: number): void {
  // ...
}
// ---cut---
foo("hello", 42);
foo("hello", 42, true);
foo("hello");
```

There is one place where the differences begin to become observable though: readability. In the first example, we have no parameter names for the first and second elements. While these have no impact on type-checking, the lack of labels on tuple positions can make them harder to use - harder to communicate our intent.

That's why in TypeScript 4.0, tuples types can now provide labels.

```
type Range = [start: number, end: number];
```

To deepen the connection between parameter lists and tuple types, the syntax for rest elements and optional elements mirrors the syntax for parameter lists.

```
type Foo = [first: number, second?: string, ...rest: any[]];
```

There are a few rules when using labeled tuples. For one, when labeling a tuple element, all other elements in the tuple must also be labeled.

```
// @errors: 5084
type Bar = [first: string, number];
```

It's worth noting - labels don't require us to name our variables differently when destructuring. They're purely there for documentation and tooling.

```
function foo(x: [first: string, second: number]) {
    // ...
    // note: we didn't need to name these 'first' and 'second'
    const [a, b] = x;
    a
    // ^
    b
    // ^
}
```

Overall, labeled tuples are handy when taking advantage of patterns around tuples and argument lists, along with implementing overloads in a type-safe way. In fact, TypeScript's editor support will try to display them as overloads when possible.

```
type Name =
    | [first: string, last: string]
    | [first: string, middle: string, last: string]

function createPerson(...name: Name) {
    // ...
}

createPerson()
```

Signature help displaying a union of labeled tuples as in a parameter list as two signatures

To learn more, check out [the pull request](#) for labeled tuple elements.

Class Property Inference from Constructors

TypeScript 4.0 can now use control flow analysis to determine the types of properties in classes when `noImplicitAny` is enabled.

```
class Square {
    // Previously both of these were any
    area;
    // ^
    sideLength;
    // ^?
```

```
constructor(sideLength: number) {
  this.sideLength = sideLength;
  this.area = sideLength ** 2;
}
```

In cases where not all paths of a constructor assign to an instance member, the property is considered to potentially be `undefined`.

```
// @errors: 2532
class Square {
  sideLength;
  // ^?

  constructor(sideLength: number) {
    if (Math.random()) {
      this.sideLength = sideLength;
    }
  }

  get area() {
    return this.sideLength ** 2;
  }
}
```

In cases where you know better (e.g. you have an `initialize` method of some sort), you'll still need an explicit type annotation along with a definite assignment assertion (`!`) if you're in `strictPropertyInitialization`.

```
class Square {
  // definite assignment assertion
  //   ^
  sideLength!: number;
  //       ^^^^^^^^
  // type annotation

  constructor(sideLength: number) {
    this.initialize(sideLength);
  }

  initialize(sideLength: number) {
    this.sideLength = sideLength;
  }

  get area() {
    return this.sideLength ** 2;
  }
}
```

For more details, [see the implementing pull request](#).

Short-Circuiting Assignment Operators

JavaScript, and a lot of other languages, support a set of operators called *compound assignment* operators. Compound assignment operators apply an operator to two arguments, and then assign the result to the left side. You may have seen these before:

```
// Addition
// a = a + b
a += b;

// Subtraction
// a = a - b
a -= b;

// Multiplication
// a = a * b
a *= b;

// Division
// a = a / b
a /= b;

// Exponentiation
// a = a ** b
a **= b;

// Left Bit Shift
// a = a << b
a <<= b;
```

So many operators in JavaScript have a corresponding assignment operator! Up until recently, however, there were three notable exceptions: logical *and* (`&&`), logical *or* (`||`), and nullish coalescing (`??`).

That's why TypeScript 4.0 supports a new ECMAScript feature to add three new assignment operators: `&&=`, `||=`, and `??=`.

These operators are great for substituting any example where a user might write code like the following:

```
a = a && b;
a = a || b;
a = a ?? b;
```

Or a similar `if` block like

```
// could be 'a ||= b'
if (!a) {
  a = b;
}
```

There are even some patterns we've seen (or, uh, written ourselves) to lazily initialize values, only if they'll be needed.

```
let values: string[] =
(values ?? (values = [])).push("hello");

// After
(values ??= []).push("hello");
```

(look, we're not proud of *all* the code we write...)

On the rare case that you use getters or setters with side-effects, it's worth noting that these operators only perform assignments if necessary. In that sense, not only is the right side of the operator "short-circuited" - the assignment itself is too.

```
obj.prop ||= foo();
// roughly equivalent to either of the following
obj.prop || (obj.prop = foo());
if (!obj.prop) {
  obj.prop = foo();
}
```

Try running the following example to see how that differs from *always* performing the assignment.

```
const obj = {
  get prop() {
    console.log("getter has run");
    // Replace me!
    return Math.random() < 0.5;
  },
  set prop(_val: boolean) {
    console.log("setter has run");
  }
};

function foo() {
  console.log("right side evaluated");
  return true;
}

console.log("This one always runs the setter");
obj.prop = obj.prop || foo();
console.log("This one *sometimes* runs the setter");
obj.prop ||= foo();
```

We'd like to extend a big thanks to community member Wenlu Wang for this contribution!

For more details, you can [take a look at the pull request here](#). You can also [check out TC39's proposal repository for this feature](#).

unknown on catch Clause Bindings

Since the beginning days of TypeScript, `catch` clause variables have always been typed as `any`. This meant that TypeScript allowed you to do anything with them.

```
// @useUnknownInCatchVariables: false
try {
  // Do some work
} catch (x) {
  // x has type 'any' - have fun!
  console.log(x.message);
  console.log(x.toUpperCase());
```

```
x++;
x.yadda.yadda.yadda();
```

The above has some undesirable behavior if we're trying to prevent *more* errors from happening in our error-handling code! Because these variables have the type `any` by default, they lack any type-safety which could have errored on invalid operations.

That's why TypeScript 4.0 now lets you specify the type of `catch` clause variables as `unknown` instead. `unknown` is safer than `any` because it reminds us that we need to perform some sorts of type-checks before operating on our values.

```
// @errors: 2571
try {
  // ...
} catch (e: unknown) {
  // Can't access values on unknowns
  console.log(e.toUpperCase());

  if (typeof e === "string") {
    // We've narrowed 'e' down to the type 'string'.
    console.log(e.toUpperCase());
  }
}
```

While the types of `catch` variables won't change by default, we might consider a new `strict` mode flag in the future so that users can opt in to this behavior. In the meantime, it should be possible to write a lint rule to force `catch` variables to have an explicit annotation of either `: any` or `: unknown`.

For more details you can [peek at the changes for this feature](#).

Custom JSX Factories

When using JSX, a `fragment` is a type of JSX element that allows us to return multiple child elements. When we first implemented fragments in TypeScript, we didn't have a great idea about how other libraries would utilize them. Nowadays most other libraries that encourage using JSX and support fragments have a similar API shape.

In TypeScript 4.0, users can customize the fragment factory through the new `jsxFragmentFactory` option.

As an example, the following `tsconfig.json` file tells TypeScript to transform JSX in a way compatible with React, but switches each factory invocation to `h` instead of `React.createElement`, and uses `Fragment` instead of `React.Fragment`.

```
{
  "compilerOptions": {
    "target": "esnext",
    "module": "commonjs",
    "jsx": "react",
    "jsxFactory": "h",
    "jsxFragmentFactory": "Fragment"
  }
}
```

In cases where you need to have a different JSX factory on a per-file basis, you can take advantage of the new `/** @jsxFrag */` pragma comment. For example, the following...

```
// @noErrors
// Note: these pragma comments need to be written
// with a JSDoc-style multiline syntax to take effect.

/** @jsx h */
/** @jsxFrag Fragment */

import { h, Fragment } from "preact";

export const Header = (
  <>
    <h1>Welcome</h1>
  </>
);

```

...will get transformed to this output JavaScript...

```
// @noErrors
// @showEmit
// Note: these pragma comments need to be written
// with a JSDoc-style multiline syntax to take effect.

/** @jsx h */
/** @jsxFrag Fragment */

import { h, Fragment } from "preact";

export const Header = (
  <>
    <h1>Welcome</h1>
  </>
);

```

We'd like to extend a big thanks to community member [Noj Vek](#) for sending this pull request and patiently working with our team on it.

You can see that [the pull request](#) for more details!

Speed Improvements in `--build` mode with `--noEmitOnError`

Previously, compiling a program after a previous compile with errors under `incremental` would be extremely slow when using the `noEmitOnError` flag. This is because none of the information from the last compilation would be cached in a `.tsbuildinfo` file based on the `noEmitOnError` flag.

TypeScript 4.0 changes this which gives a great speed boost in these scenarios, and in turn improves `--build` mode scenarios (which imply both `incremental` and `noEmitOnError`).

For details, [read up more on the pull request](#).

--incremental with --noEmit

TypeScript 4.0 allows us to use the `noEmit` flag when while still leveraging `incremental` compiles. This was previously not allowed, as `incremental` needs to emit a `.tsbuildinfo` files; however, the use-case to enable faster incremental builds is important enough to enable for all users.

For more details, you can [see the implementing pull request](#).

Editor Improvements

The TypeScript compiler doesn't only power the editing experience for TypeScript itself in most major editors - it also powers the JavaScript experience in the Visual Studio family of editors and more. For that reason, much of our work focuses on improving editor scenarios - the place you spend most of your time as a developer.

Using new TypeScript/JavaScript functionality in your editor will differ depending on your editor, but

- Visual Studio Code supports [selecting different versions of TypeScript](#). Alternatively, there's the [JavaScript/TypeScript Nightly Extension](#) to stay on the bleeding edge (which is typically very stable).
- Visual Studio 2017/2019 have [the SDK installers above] and [MSBuild installs](#).
- Sublime Text 3 supports [selecting different versions of TypeScript](#)

You can check out a partial [list of editors that have support for TypeScript](#) to learn more about whether your favorite editor has support to use new versions.

Convert to Optional Chaining

Optional chaining is a recent feature that's received a lot of love. That's why TypeScript 4.0 brings a new refactoring to convert common patterns to take advantage of [optional chaining](#) and [nullish coalescing](#)!

```
function tryCallF(a) {
  return a && a.b.c && a.b.c.d.e.f();
}
```

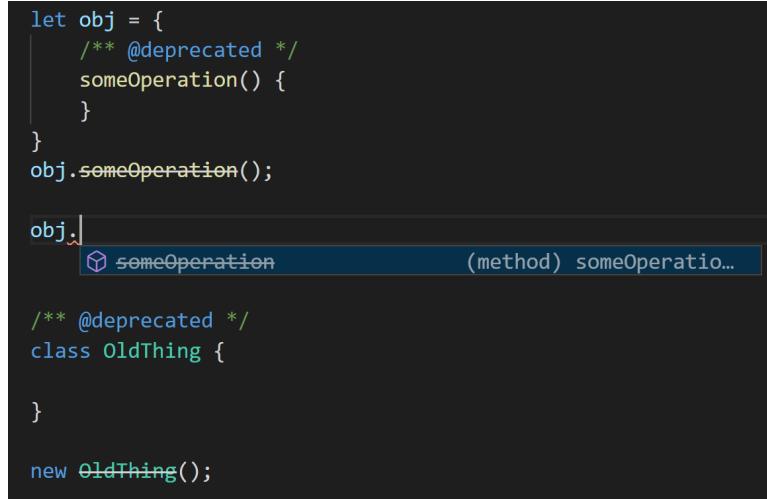
Converting `a && a.b.c && a.b.c.d.e.f()` to `a?.b.c?.d.e.f()`

Keep in mind that while this refactoring doesn't *perfectly* capture the same behavior due to subtleties with truthiness/falsiness in JavaScript, we believe it should capture the intent for most use-cases, especially when TypeScript has more precise knowledge of your types.

For more details, [check out the pull request for this feature](#).

`/** @deprecated */ Support`

TypeScript's editing support now recognizes when a declaration has been marked with a `/** @deprecated */` JSDoc comment. That information is surfaced in completion lists and as a suggestion diagnostic that editors can handle specially. In an editor like VS Code, deprecated values are typically displayed a strike-through style ~~like this~~.



```
let obj = {
  /** @deprecated */
  someOperation() {
  }
}
obj.someOperation();

obj|
someOperation (method) someOperatio...
```

```
/** @deprecated */
class OldThing {

}

new OldThing();
```

Some examples of deprecated declarations with strikethrough text in the editor

This new functionality is available thanks to [Wenlu Wang](#). See [the pull request](#) for more details.

Partial Semantic Mode at Startup

We've heard a lot from users suffering from long startup times, especially on bigger projects. The culprit is usually a process called *program construction*. This is the process of starting with an initial set of root files, parsing them, finding their dependencies, parsing those dependencies, finding those dependencies' dependencies, and so on. The bigger your project is, the longer you'll have to wait before you can get basic editor operations like go-to-definition or quick info.

That's why we've been working on a new mode for editors to provide a *partial* experience until the full language service experience has loaded up. The core idea is that editors can run a lightweight partial server that only looks at the current files that the editor has open.

It's hard to say precisely what sorts of improvements you'll see, but anecdotally, it used to take anywhere between *20 seconds to a minute* before TypeScript would become fully responsive on the Visual Studio Code codebase. In contrast, **our new partial semantic mode seems to bring that delay down to just a few seconds**. As an example, in the following video, you can see two side-by-side editors with TypeScript 3.9 running on the left and TypeScript 4.0 running on the right.



When restarting both editors on a particularly large codebase, the one with TypeScript 3.9 can't provide completions or quick info at all. On the other hand, the editor with TypeScript 4.0 can *immediately* give us a rich experience in the current file we're editing, despite loading the full project in the background.

Currently the only editor that supports this mode is [Visual Studio Code](#) which has some UX improvements coming up in [Visual Studio Code Insiders](#). We recognize that this experience may still have room for polish in UX and functionality, and we have a [list of improvements](#) in mind. We're looking for more feedback on what you think might be useful.

For more information, you can [see the original proposal](#), [the implementing pull request](#), along with [the follow-up meta issue](#).

Smarter Auto-Imports

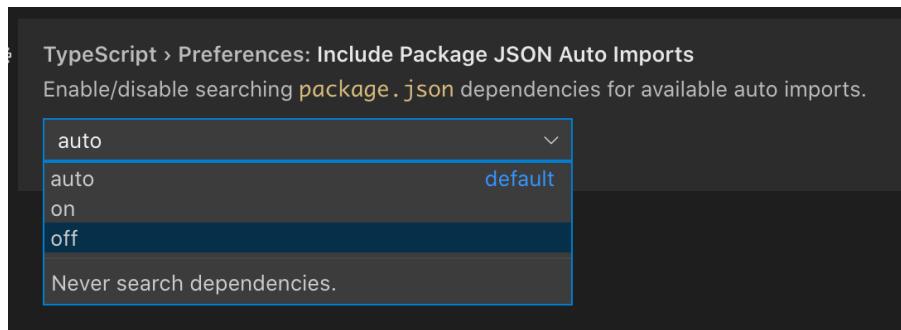
Auto-import is a fantastic feature that makes coding a lot easier; however, every time auto-import doesn't seem to work, it can throw users off a lot. One specific issue that we heard from users was that auto-imports didn't work on dependencies that were written in TypeScript - that is, until they wrote at least one explicit import somewhere else in their project.

Why would auto-imports work for `@types` packages, but not for packages that ship their own types? It turns out that auto-imports only work on packages your project *already* includes. Because TypeScript has some quirky defaults that automatically add packages in `node_modules/@types` to your project, *those* packages would be auto-imported. On the other hand, other packages were excluded because crawling through all your `node_modules` packages can be *really* expensive.

All of this leads to a pretty lousy getting started experience for when you're trying to auto-import something that you've just installed but haven't used yet.

TypeScript 4.0 now does a little extra work in editor scenarios to include the packages you've listed in your `package.json`'s `dependencies` (and `peerDependencies`) fields. The information from these packages is only used to improve auto-imports, and doesn't change anything else like type-checking. This allows us to provide auto-imports for all of your dependencies that have types, without incurring the cost of a complete `node_modules` search.

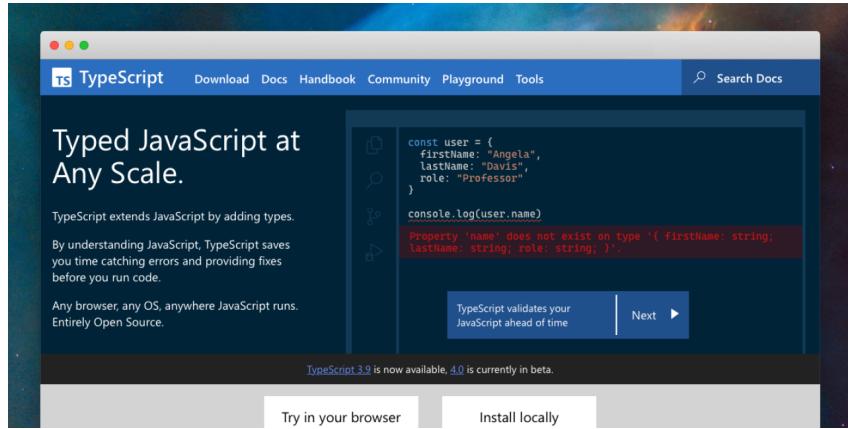
In the rare cases when your `package.json` lists more than ten typed dependencies that haven't been imported yet, this feature automatically disables itself to prevent slow project loading. To force the feature to work, or to disable it entirely, you should be able to configure your editor. For Visual Studio Code, this is the "Include Package JSON Auto Imports" (or `typescript.preferences.includePackageJsonAutoImports`) setting.



For more details, you can see the [proposal issue](#) along with [the implementing pull request](#).

Our New Website!

The [TypeScript website](#) has recently been rewritten from the ground up and rolled out!



A screenshot of the new TypeScript website

We already wrote a bit about our new site, so you can read up more there; but it's worth mentioning that we're still looking to hear what you think! If you have questions, comments, or suggestions, you can [file them over on the website's issue tracker](#).

Breaking Changes

`lib.d.ts` Changes

Our `lib.d.ts` declarations have changed - most specifically, types for the DOM have changed. The most notable change may be the removal of `document.origin` which only worked in old versions of IE and Safari MDN recommends moving to `self.origin`.

Properties Overriding Accessors (and vice versa) is an Error

Previously, it was only an error for properties to override accessors, or accessors to override properties, when using `useDefineForClassFields`; however, TypeScript now always issues an error when declaring a property in a derived class that would override a getter or setter in the base class.

```
// @errors: 1049 2610
class Base {
  get foo() {
    return 100;
  }
  set foo(value) {
    // ...
  }
}

class Derived extends Base {
  foo = 10;
}
```

```
// @errors: 2611
class Base {
  prop = 10;
}

class Derived extends Base {
  get prop() {
    return 100;
  }
}
```

See more details on [the implementing pull request](#).

Operands for `delete` must be optional.

When using the `delete` operator in `strictNullChecks`, the operand must now be `any`, `unknown`, `never`, or be optional (in that it contains `undefined` in the type). Otherwise, use of the `delete` operator is an error.

```
// @errors: 2790
interface Thing {
  prop: string;
}

function f(x: Thing) {
  delete x.prop;
}
```

See more details on [the implementing pull request](#).

Usage of TypeScript's Node Factory is Deprecated

Today TypeScript provides a set of "factory" functions for producing AST Nodes; however, TypeScript 4.0 provides a new node factory API. As a result, for TypeScript 4.0 we've made the decision to deprecate these older functions in favor of the new ones.

For more details, [read up on the relevant pull request for this change](#).

[Go to TOC](#)

Template Literal Types

String literal types in TypeScript allow us to model functions and APIs that expect a set of specific strings.

```
// @errors: 2345
function setVerticalAlignment(location: "top" | "middle" | "bottom") {
  // ...
}

setVerticalAlignment("middel");
```

This is pretty nice because string literal types can basically spell-check our string values.

We also like that string literals can be used as property names in mapped types. In this sense, they're also usable as building blocks:

```
type Options = {
  [K in "noImplicitAny" | "strictNullChecks" | "strictFunctionTypes"]?: boolean;
};

// same as
// type Options = {
//   noImplicitAny?: boolean,
//   strictNullChecks?: boolean,
//   strictFunctionTypes?: boolean
// };
```

But there's another place that string literal types could be used as building blocks: building other string literal types.

That's why TypeScript 4.1 brings the template literal string type. It has the same syntax as [template literal strings in JavaScript](#), but is used in type positions. When you use it with concrete literal types, it produces a new string literal type by concatenating the contents.

```
type World = "world";
type Greeting = `hello ${World}`;
```

What happens when you have unions in substitution positions? It produces the set of every possible string literal that could be represented by each union member.

```
type Color = "red" | "blue";
type Quantity = "one" | "two";

type SeussFish = `${Quantity | Color} fish`;
```

This can be used beyond cute examples in release notes. For example, several libraries for UI components have a way to specify both vertical and horizontal alignment in their APIs, often with both at once using a single string like `"bottom-right"`. Between vertically aligning with `"top"`, `"middle"`, and `"bottom"`, and horizontally aligning with `"left"`, `"center"`, and `"right"`, there are 9 possible strings where each of the former strings is connected with each of the latter strings using a dash.

```
// @errors: 2345
type VerticalAlignment = "top" | "middle" | "bottom";
type HorizontalAlignment = "left" | "center" | "right";

// Takes
//   | "top-left"   | "top-center"   | "top-right"
//   | "middle-left" | "middle-center" | "middle-right"
//   | "bottom-left" | "bottom-center" | "bottom-right"

declare function setAlignment(value:
` ${VerticalAlignment}`-${HorizontalAlignment}`): void;

setAlignment("top-left"); // works!
setAlignment("top-middle"); // error!
setAlignment("top-pot"); // error! but good doughnuts if you're ever in Seattle
```

While there are **lots** of examples of this sort of API in the wild, this is still a bit of a toy example since we could write these out manually. In fact, for 9 strings, this is likely fine; but when you need a ton of strings, you should consider automatically generating them ahead of time to save work on every type-check (or just use `string`, which will be much simpler to comprehend).

Some of the real value comes from dynamically creating new string literals. For example, imagine a `makeWatchedObject` API that takes an object and produces a mostly identical object, but with a new `on` method to detect for changes to the properties.

```
let person = makeWatchedObject({
  firstName: "Homer",
  age: 42, // give-or-take
  location: "Springfield",
});

person.on("firstNameChanged", () => {
  console.log(`firstName was changed!`);
});
```

Notice that `on` listens on the event `"firstNameChanged"`, not just `"firstName"`. How would we type this?

```
type PropEventSource<T> = {
  on(eventName: `${string & keyof T}Changed`, callback: () => void): void;
};

/// Create a "watched object" with an 'on' method
/// so that you can watch for changes to properties.
declare function makeWatchedObject<T>(obj: T): T & PropEventSource<T>;
```

With this, we can build something that errors when we give the wrong property!

```
// @errors: 2345
type PropEventSource<T> = {
  on(eventName: `${string & keyof T}Changed`, callback: () => void): void;
};
declare function makeWatchedObject<T>(obj: T): T & PropEventSource<T>;
let person = makeWatchedObject({
  firstName: "Homer",
  age: 42, // give-or-take
  location: "Springfield",
```

```
});  
// ---cut---  
// error!  
person.on("firstName", () => {});  
// error!  
person.on("frstNameChanged", () => {});
```

We can also do something special in template literal types: we can *infer* from substitution positions. We can make our last example generic to infer from parts of the `eventName` string to figure out the associated property.

```
type PropEventSource<T> = {  
    on<K extends string & keyof T>  
        (eventName: `${K}Changed`, callback: (newValue: T[K]) => void): void;  
};  
  
declare function makeWatchedObject<T>(obj: T): T & PropEventSource<T>;  
  
let person = makeWatchedObject({  
    firstName: "Homer",  
    age: 42,  
    location: "Springfield",  
});  
  
// works! 'newName' is typed as 'string'  
person.on("firstNameChanged", newName => {  
    // 'newName' has the type of 'firstName'  
    console.log(`new name is ${newName.toUpperCase()}`);  
});  
  
// works! 'newAge' is typed as 'number'  
person.on("ageChanged", newAge => {  
    if (newAge < 0) {  
        console.log("warning! negative age");  
    }  
})
```

Here we made `on` into a generic method. When a user calls with the string `"firstNameChanged"`, TypeScript will try to infer the right type for `K`. To do that, it will match `K` against the content prior to `"Changed"` and infer the string `"firstName"`. Once TypeScript figures that out, the `on` method can fetch the type of `firstName` on the original object, which is `string` in this case. Similarly, when we call with `"ageChanged"`, it finds the type for the property `age` which is `number`.

Inference can be combined in different ways, often to deconstruct strings, and reconstruct them in different ways. In fact, to help with modifying these string literal types, we've added a few new utility type aliases for modifying casing in letters (i.e. converting to lowercase and uppercase characters).

```
type EnthusiasticGreeting<T extends string> = `${Uppercase<T>}`  
type HELLO = EnthusiasticGreeting<"hello">;  
// ^?
```

The new type aliases are `Uppercase`, `Lowercase`, `Capitalize` and `Uncapitalize`. The first two transform every character in a string, and the latter two transform only the first character in a string.

For more details, see the [original pull request](#) and the [in-progress pull request to switch to type alias helpers](#).

Key Remapping in Mapped Types

Just as a refresher, a mapped type can create new object types based on arbitrary keys

```
type Options = {
  [K in "noImplicitAny" | "strictNullChecks" | "strictFunctionTypes"]?: boolean;
};

// same as
// type Options = {
//   noImplicitAny?: boolean,
//   strictNullChecks?: boolean,
//   strictFunctionTypes?: boolean
// };
```

or new object types based on other object types.

```
/// 'Partial<T>' is the same as 'T', but with each property marked optional.
type Partial<T> = {
  [K in keyof T]?: T[K];
};
```

Until now, mapped types could only produce new object types with keys that you provided them; however, lots of the time you want to be able to create new keys, or filter out keys, based on the inputs.

That's why TypeScript 4.1 allows you to re-map keys in mapped types with a new `as` clause.

```
type MappedTypeWithNewKeys<T> = {
  [K in keyof T as NewKeyType]: T[K]
};

// This is the new syntax!
```

With this new `as` clause, you can leverage features like template literal types to easily create property names based off of old ones.

```
type Getters<T> = {
  [K in keyof T as `get${Capitalize<string & K>}`]: () => T[K]
};

interface Person {
  name: string;
  age: number;
  location: string;
}

type LazyPerson = Getters<Person>;
// ^?
```

and you can even filter out keys by producing `never`. That means you don't have to use an extra `Omit` helper type in some cases.

```
// Remove the 'kind' property
type RemoveKindField<T> = {
  [K in keyof T as Exclude<K, "kind">]: T[K]
};

interface Circle {
  kind: "circle";
  radius: number;
}

type KindlessCircle = RemoveKindField<Circle>;
// ^?
```

For more information, take a look at [the original pull request over on GitHub](#).

Recursive Conditional Types

In JavaScript it's fairly common to see functions that can flatten and build up container types at arbitrary levels. For example, consider the `.then()` method on instances of `Promise`. `.then(...)` unwraps each promise until it finds a value that's not "promise-like", and passes that value to a callback. There's also a relatively new `flat` method on `Array`s that can take a depth of how deep to flatten.

Expressing this in TypeScript's type system was, for all practical intents and purposes, not possible. While there were hacks to achieve this, the types ended up looking very unreasonable.

That's why TypeScript 4.1 eases some restrictions on conditional types - so that they can model these patterns. In TypeScript 4.1, conditional types can now immediately reference themselves within their branches, making it easier to write recursive type aliases.

For example, if we wanted to write a type to get the element types of nested arrays, we could write the following `deepFlatten` type.

```
type ElementType<T> = T extends ReadonlyArray<infer U> ? ElementType<U> : T;

function deepFlatten<T extends readonly unknown[]>(x: T): ElementType<T>[] {
  throw "not implemented";
}

// All of these return the type 'number[]':
deepFlatten([1, 2, 3]);
deepFlatten([[1], [2, 3]]);
deepFlatten([[1], [[2]], [[[3]]]]);
```

Similarly, in TypeScript 4.1 we can write an `Awaited` type to deeply unwrap `Promise`s.

```
type Awaited<T> = T extends PromiseLike<infer U> ? Awaited<U> : T;
/// Like `promise.then(...)` , but more accurate in types.
declare function customThen<T, U>(
```

```
p: Promise<T>,
onFulfilled: (value: Awaited<T>) => U
): Promise<Awaited<U>>;
```

Keep in mind that while these recursive types are powerful, but they should be used responsibly and sparingly.

First off, these types can do a lot of work which means that they can increase type-checking time. Trying to model numbers in the Collatz conjecture or Fibonacci sequence might be fun, but don't ship that in `.d.ts` files on npm.

But apart from being computationally intensive, these types can hit an internal recursion depth limit on sufficiently-complex inputs. When that recursion limit is hit, that results in a compile-time error. In general, it's better not to use these types at all than to write something that fails on more realistic examples.

See more [at the implementation](#).

Checked Indexed Accesses (-- noUncheckedIndexedAccess)

TypeScript has a feature called *index signatures*. These signatures are a way to signal to the type system that users can access arbitrarily-named properties.

```
interface Options {
  path: string;
  permissions: number;

  // Extra properties are caught by this index signature.
  [propName: string]: string | number;
}

function checkOptions(opts: Options) {
  opts.path; // string
  opts.permissions; // number

  // These are all allowed too!
  // They have the type 'string | number'.
  opts.yadda.toString();
  opts["foo bar baz"].toString();
  opts[Math.random()].toString();
}
```

In the above example, `Options` has an index signature that says any accessed property that's not already listed should have the type `string | number`. This is often convenient for optimistic code that assumes you know what you're doing, but the truth is that most values in JavaScript do not support every potential property name. Most types will not, for example, have a value for a property key created by `Math.random()` like in the previous example. For many users, this behavior was undesirable, and felt like it wasn't leveraging the full strict-checking of `strictNullChecks`.

That's why TypeScript 4.1 ships with a new flag called `noUncheckedIndexedAccess`. Under this new mode, every property access (like `foo.bar`) or indexed access (like `foo["bar"]`) is considered potentially undefined. That means that in our last example, `opts.yadda` will have the type `string | number | undefined` as opposed to just `string | number`. If you need to access that property, you'll either have to check for its existence first or use a non-null assertion operator (the postfix `!` character).

```
// @errors: 2532
// @noUncheckedIndexedAccess
interface Options {
    path: string;
    permissions: number;

    // Extra properties are caught by this index signature.
    [propName: string]: string | number;
}
// ---cut---
function checkOptions(opts: Options) {
    opts.path; // string
    opts.permissions; // number

    // These are not allowed with noUncheckedIndexedAccess
    opts.yadda.toString();
    opts["foo bar baz"].toString();
    opts[Math.random()].toString();

    // Checking if it's really there first.
    if (opts.yadda) {
        console.log(opts.yadda.toString());
    }

    // Basically saying "trust me I know what I'm doing"
    // with the '!' non-null assertion operator.
    opts.yadda!.toString();
}
```

One consequence of using `noUncheckedIndexedAccess` is that indexing into an array is also more strictly checked, even in a bounds-checked loop.

```
// @errors: 2532
// @noUncheckedIndexedAccess
function screamLines(strs: string[]) {
    // This will have issues
    for (let i = 0; i < strs.length; i++) {
        console.log(strs[i].toUpperCase());
    }
}
```

If you don't need the indexes, you can iterate over individual elements by using a `for - of` loop or a `forEach` call.

```
// @noUncheckedIndexedAccess
function screamLines(strs: string[]) {
    // This works fine
    for (const str of strs) {
        console.log(str.toUpperCase());
    }
}
```

```
// This works fine
strs.forEach((str) => {
  console.log(str.toUpperCase());
});
}
```

This flag can be handy for catching out-of-bounds errors, but it might be noisy for a lot of code, so it is not automatically enabled by the `strict` flag; however, if this feature is interesting to you, you should feel free to try it and determine whether it makes sense for your team's codebase!

You can learn more at [the implementing pull request](#).

paths without baseUrl

Using path-mapping is fairly common - often it's to have nicer imports, often it's to simulate monorepo linking behavior.

Unfortunately, specifying `paths` to enable path-mapping required also specifying an option called `baseUrl`, which allows bare specifier paths to be resolved relative to the `baseUrl` too. This also often caused poor paths to be used by auto-imports.

In TypeScript 4.1, the `paths` option can be used without `baseUrl`. This helps avoid some of these issues.

checkJs Implies allowJs

Previously if you were starting a checked JavaScript project, you had to set both `allowJs` and `checkJs`. This was a slightly annoying bit of friction in the experience, so `checkJs` now implies `allowJs` by default.

[See more details at the pull request](#).

React 17 JSX Factories

TypeScript 4.1 supports React 17's upcoming `jsx` and `jsxs` factory functions through two new options for the `jsx` compiler option:

- `react-jsx`
- `react-jsxdev`

These options are intended for production and development compiles respectively. Often, the options from one can extend from the other. For example, a `tsconfig.json` for production builds might look like the following:

```
// ./src/tsconfig.json
{
  "compilerOptions": {
    "module": "esnext",
    "target": "es2015",
    "jsx": "react-jsx",
    "strict": true
  }
}
```

```
    },
  "include": ["./**/*"]
}
```

and one for development builds might look like the following:

```
// ./src/tsconfig.dev.json
{
  "extends": "./tsconfig.json",
  "compilerOptions": {
    "jsx": "react-jsxdev"
  }
}
```

For more information, [check out the corresponding PR](#).

Editor Support for the JSDoc `@see` Tag

The JSDoc tag `@see` tag now has better support in editors for TypeScript and JavaScript. This allows you to use functionality like go-to-definition in a dotted name following the tag. For example, going to definition on `first` or `C` in the JSDoc comment just works in the following example:

```
// @filename: first.ts
export class C {}

// @filename: main.ts
import * as first from "./first";

/**
 * @see first.C
 */
function related() {}
```

Thanks to frequent contributor [Wenlu Wang](#) for implementing this!

Breaking Changes

`lib.d.ts` Changes

`lib.d.ts` may have a set of changed APIs, potentially in part due to how the DOM types are automatically generated. One specific change is that `Reflect.enumerate` has been removed, as it was removed from ES2016.

`abstract` Members Can't Be Marked `async`

Members marked as `abstract` can no longer be marked as `async`. The fix here is to remove the `async` keyword, since callers are only concerned with the return type.

`any` / `unknown` Are Propagated in Falsy Positions

Previously, for an expression like `foo && somethingElse`, the type of `foo` was `any` or `unknown`, the type of the whole that expression would be the type of `somethingElse`.

For example, previously the type for `x` here was `{ someProp: string }`.

```
declare let foo: unknown;
declare let somethingElse: { someProp: string };

let x = foo && somethingElse;
```

However, in TypeScript 4.1, we are more careful about how we determine this type. Since nothing is known about the type on the left side of the `&&`, we propagate `any` and `unknown` outward instead of the type on the right side.

The most common pattern we saw of this tended to be when checking compatibility with `booleans`, especially in predicate functions.

```
function isThing(x: any): boolean {
  return x && typeof x === "object" && x.blah === "foo";
}
```

Often the appropriate fix is to switch from `foo && someExpression` to `!foo && someExpression`.

resolve's Parameters Are No Longer Optional in Promises

When writing code like the following

```
new Promise((resolve) => {
  doSomethingAsync() => {
    doSomething();
    resolve();
  };
});
```

You may get an error like the following:

```
resolve()
~~~~~
error TS2554: Expected 1 arguments, but got 0.
An argument for 'value' was not provided.
```

This is because `resolve` no longer has an optional parameter, so by default, it must now be passed a value. Often this catches legitimate bugs with using `Promise`s. The typical fix is to pass it the correct argument, and sometimes to add an explicit type argument.

```
new Promise<number>((resolve) => {
  // ^^^^^^
  doSomethingAsync((value) => {
    doSomething();
    resolve(value);
    // ^^^^
  });
});
```

However, sometimes `resolve()` really does need to be called without an argument. In these cases, we can give `Promise` an explicit `void` generic type argument (i.e. write it out as `Promise<void>`). This leverages new functionality in TypeScript 4.1 where a potentially-`void` trailing parameter can become optional.

```
new Promise<void>((resolve) => {
  // ^^^^^^
  doSomethingAsync() => {
    doSomething();
    resolve();
  };
});
```

TypeScript 4.1 ships with a quick fix to help fix this break.

Conditional Spreads Create Optional Properties

In JavaScript, object spreads (like `{ ...foo }`) don't operate over falsy values. So in code like `{ ...foo }`, `foo` will be skipped over if it's `null` or `undefined`.

Many users take advantage of this to spread in properties "conditionally".

```
interface Person {
  name: string;
  age: number;
  location: string;
}

interface Animal {
  name: string;
  owner: Person;
}

function copyOwner(pet?: Animal) {
  return {
    ...(pet && pet.owner),
    otherStuff: 123,
  };
}

// We could also use optional chaining here:

function copyOwner(pet?: Animal) {
  return {
    ...pet?.owner,
    otherStuff: 123,
  };
}
```

Here, if `pet` is defined, the properties of `pet.owner` will be spread in - otherwise, no properties will be spread into the returned object.

The return type of `copyOwner` was previously a union type based on each spread:

```
{ x: number } | { x: number, name: string, age: number, location: string }
```

This modeled exactly how the operation would occur: if `pet` was defined, all the properties from `Person` would be present; otherwise, none of them would be defined on the result. It was an all-or-nothing operation.

However, we've seen this pattern taken to the extreme, with hundreds of spreads in a single object, each spread potentially adding in hundreds or thousands of properties. It turns out that for various reasons, this ends up being extremely expensive, and usually for not much benefit.

In TypeScript 4.1, the returned type sometimes uses all-optional properties.

```
{  
  x: number;  
  name?: string;  
  age?: number;  
  location?: string;  
}
```

This ends up performing better and generally displaying better too.

For more details, [see the original change](#). While this behavior is not entirely consistent right now, we expect a future release will produce cleaner and more predictable results.

Unmatched parameters are no longer related

TypeScript would previously relate parameters that didn't correspond to each other by relating them to the type `any`. With [changes in TypeScript 4.1](#), the language now skips this process entirely. This means that some cases of assignability will now fail, but it also means that some cases of overload resolution can fail as well. For example, overload resolution on `util.promisify` in Node.js may select a different overload in TypeScript 4.1, sometimes causing new or different errors downstream.

As a workaround, you may be best using a type assertion to squelch errors.

[Go to TOC](#)

Smarter Type Alias Preservation

TypeScript has a way to declare new names for types called type aliases. If you're writing a set of functions that all work on `string | number | boolean`, you can write a type alias to avoid repeating yourself over and over again.

```
type BasicPrimitive = number | string | boolean;
```

TypeScript has always used a set of rules and guesses for when to reuse type aliases when printing out types. For example, take the following code snippet.

```
export type BasicPrimitive = number | string | boolean;

export function doStuff(value: BasicPrimitive) {
    let x = value;
    return x;
}
```

If we hover our mouse over `x` in an editor like Visual Studio, Visual Studio Code, or [the TypeScript Playground](#), we'll get a quick info panel that shows the type `BasicPrimitive`. Likewise, if we get the declaration file output (`.d.ts` output) for this file, TypeScript will say that `doStuff` returns `BasicPrimitive`.

However, what happens if we return a `BasicPrimitive` or `undefined`?

```
export type BasicPrimitive = number | string | boolean;

export function doStuff(value: BasicPrimitive) {
    if (Math.random() < 0.5) {
        return undefined;
    }

    return value;
}
```

We can see what happens [in the TypeScript 4.1 playground](#). While we might want TypeScript to display the return type of `doStuff` as `BasicPrimitive | undefined`, it instead displays `string | number | boolean | undefined`! What gives?

Well this has to do with how TypeScript represents types internally. When creating a union type out of one or more union types, it will always *normalize* those types into a new flattened union type - but doing that loses information. The type-checker would have to find every combination of types from `string | number | boolean | undefined` to see what type aliases could have been used, and even then, there might be multiple type aliases to `string | number | boolean`.

In TypeScript 4.2, our internals are a little smarter. We keep track of how types were constructed by keeping around parts of how they were originally written and constructed over time. We also keep track of, and differentiate, type aliases to instances of other aliases!

Being able to print back the types based on how you used them in your code means that as a TypeScript user, you can avoid some unfortunately humongous types getting displayed, and that often translates to getting better `.d.ts` file output, error messages, and in-editor type displays in quick info and signature help. This can help TypeScript feel a little bit more approachable for newcomers.

For more information, check out [the first pull request that improves various cases around preserving union type aliases](#), along with [a second pull request that preserves indirect aliases](#).

Leading/Middle Rest Elements in Tuple Types

In TypeScript, tuple types are meant to model arrays with specific lengths and element types.

```
// A tuple that stores a pair of numbers
let a: [number, number] = [1, 2];

// A tuple that stores a string, a number, and a boolean
let b: [string, number, boolean] = ["hello", 42, true];
```

Over time, TypeScript's tuple types have become more and more sophisticated, since they're also used to model things like parameter lists in JavaScript. As a result, they can have optional elements and rest elements, and can even have labels for tooling and readability.

```
// A tuple that has either one or two strings.
let c: [string, string?] = ["hello"];
c = ["hello", "world"];

// A labeled tuple that has either one or two strings.
let d: [first: string, second?: string] = ["hello"];
d = ["hello", "world"];

// A tuple with a *rest element* - holds at least 2 strings at the front,
// and any number of booleans at the back.
let e: [string, string, ...boolean[]];

e = ["hello", "world"];
e = ["hello", "world", false];
e = ["hello", "world", true, false, true];
```

In TypeScript 4.2, rest elements specifically been expanded in how they can be used. In prior versions, TypeScript only allowed `...rest` elements at the very last position of a tuple type.

However, now rest elements can occur *anywhere* within a tuple - with only a few restrictions.

```
let foo: [...string[], number];
foo = [123];
foo = ["hello", 123];
foo = ["hello!", "hello!", "hello!", 123];

let bar: [boolean, ...string[], boolean];
bar = [true, false];
bar = [true, "some text", false];
bar = [true, "some", "separated", "text", false];
```

The only restriction is that a rest element can be placed anywhere in a tuple, so long as it's not followed by another optional element or rest element. In other words, only one rest element per tuple, and no optional elements after rest elements.

```
// @errors: 1265 1266
interface Clown {
  /*...*/
}
interface Joker {
  /*...*/
}

let StealersWheel: [...Clown[], "me", ...Joker[]];
let StringsAndMaybeBoolean: [...string[], boolean?];
```

These non-trailing rest elements can be used to model functions that take any number of leading arguments, followed by a few fixed ones.

```
declare function doStuff(...args: [...names: string[], shouldCapitalize: boolean]): void;
doStuff/*shouldCapitalize:*/ false
doStuff("fee", "fi", "fo", "fum", /*shouldCapitalize:*/ true);
```

Even though JavaScript doesn't have any syntax to model leading rest parameters, we were still able to declare `doStuff` as a function that takes leading arguments by declaring the `...args` rest parameter with a *tuple type that uses a leading rest element*. This can help model lots of existing JavaScript out there!

For more details, [see the original pull request](#).

Stricter Checks For The `in` Operator

In JavaScript, it is a runtime error to use a non-object type on the right side of the `in` operator. TypeScript 4.2 ensures this can be caught at design-time.

```
// @errors: 2361
"foo" in 42;
```

This check is fairly conservative for the most part, so if you have received an error about this, it is likely an issue in the code.

A big thanks to our external contributor Jonas Hübotter for [their pull request](#)!

--noPropertyAccessFromIndexSignature

Back when TypeScript first introduced index signatures, you could only get properties declared by them with "bracketed" element access syntax like `person["name"]`.

```
interface SomeType {
  /** This is an index signature. */
  [propName: string]: any;
```

```

    }

    function doStuff(value: SomeType) {
        let x = value["someProperty"];
    }
}

```

This ended up being cumbersome in situations where we need to work with objects that have arbitrary properties. For example, imagine an API where it's common to misspell a property name by adding an extra `s` character at the end.

```

interface Options {
    /** File patterns to be excluded. */
    exclude?: string[];
}

/**
 * It handles any extra properties that we haven't declared as type 'any'.
 */
[x: string]: any;
}

function processOptions(opts: Options) {
    // Notice we're *intentionally* accessing `excludes`, not `exclude`.
    if (opts.excludes) {
        console.error(
            "The option `excludes` is not valid. Did you mean `exclude`?"
        );
    }
}

```

To make these types of situations easier, a while back, TypeScript made it possible to use "dotted" property access syntax like `person.name` when a type had a string index signature. This also made it easier to transition existing JavaScript code over to TypeScript.

However, loosening the restriction also meant that misspelling an explicitly declared property became much easier.

```

interface Options {
    /** File patterns to be excluded. */
    exclude?: string[];
}

/**
 * It handles any extra properties that we haven't declared as type 'any'.
 */
[x: string]: any;
}

// ---cut---
function processOptions(opts: Options) {
    // ...

    // Notice we're *accidentally* accessing `excludes` this time.
    // Oops! Totally valid.
    for (const excludePattern of opts.excludes) {
        // ...
    }
}

```

In some cases, users would prefer to explicitly opt into the index signature - they would prefer to get an error message when a dotted property access doesn't correspond to a specific property declaration.

That's why TypeScript introduces a new flag called `noPropertyAccessFromIndexSignature`. Under this mode, you'll be opted in to TypeScript's older behavior that issues an error. This new setting is not under the `strict` family of flags, since we believe users will find it more useful on certain codebases than others.

You can understand this feature in more detail by reading up on the corresponding [pull request](#). We'd also like to extend a big thanks to [Wenlu Wang](#) who sent us this pull request!

abstract Construct Signatures

TypeScript allows us to mark a class as *abstract*. This tells TypeScript that the class is only meant to be extended from, and that certain members need to be filled in by any subclass to actually create an instance.

```
// @errors: 2511
abstract class Shape {
    abstract getArea(): number;
}

new Shape();

class Square extends Shape {
    #sideLength: number;

    constructor(sideLength: number) {
        super();
        this.#sideLength = sideLength;
    }

    getArea() {
        return this.#sideLength ** 2;
    }
}

// Works fine.
new Square(42);
```

To make sure this restriction in `new`-ing up `abstract` classes is consistently applied, you can't assign an `abstract` class to anything that expects a construct signature.

```
// @errors: 2322
abstract class Shape {
    abstract getArea(): number;
}
// ---cut---
interface HasArea {
    getArea(): number;
}

let Ctor: new () => HasArea = Shape;
```

This does the right thing in case we intend to run code like `new Ctor`, but it's overly-restrictive in case we want to write a subclass of `Ctor`.

```
// @errors: 2345
abstract class Shape {
  abstract getArea(): number;
}

interface HasArea {
  getArea(): number;
}

function makeSubclassWithArea(Ctor: new () => HasArea) {
  return class extends Ctor {
    getArea() {
      return 42
    }
  };
}

let MyShape = makeSubclassWithArea(Shape);
```

It also doesn't work well with built-in helper types like `InstanceType`.

```
// @errors: 2344
abstract class Shape {
  abstract getArea(): number;
}
// ---cut---
type MyInstance = InstanceType<typeof Shape>;
```

That's why TypeScript 4.2 allows you to specify an `abstract` modifier on constructor signatures.

```
abstract class Shape {
  abstract getArea(): number;
}
// ---cut---
interface HasArea {
  getArea(): number;
}

// Works!
let Ctor: abstract new () => HasArea = Shape;
```

Adding the `abstract` modifier to a construct signature signals that you can pass in `abstract` constructors. It doesn't stop you from passing in other classes/constructor functions that are "concrete" - it really just signals that there's no intent to run the constructor directly, so it's safe to pass in either class type.

This feature allows us to write *mixin factories* in a way that supports abstract classes. For example, in the following code snippet, we're able to use the mixin function `withStyles` with the `abstract` class `SuperClass`.

```
abstract class SuperClass {
  abstract someMethod(): void;
  badda() {}
}

type AbstractConstructor<T> = abstract new (...args: any[]) => T
```

```

function withStyles<T extends AbstractConstructor<object>>(Ctor: T) {
    abstract class StyledClass extends Ctor {
        getStyles() {
            // ...
        }
    }
    return StyledClass;
}

class SubClass extends withStyles(SuperClass) {
    someMethod() {
        this.someMethod()
    }
}

```

Note that `withStyles` is demonstrating a specific rule, where a class (like `StyledClass`) that extends a value that's generic and bounded by an abstract constructor (like `Ctor`) has to also be declared `abstract`. This is because there's no way to know if a class with *more* abstract members was passed in, and so it's impossible to know whether the subclass implements all the abstract members.

You can read up more on abstract construct signatures [on its pull request](#).

Understanding Your Project Structure With `--explainFiles`

A surprisingly common scenario for TypeScript users is to ask "why is TypeScript including this file?". Inferring the files of your program turns out to be a complicated process, and so there are lots of reasons why a specific combination of `lib.d.ts` got used, why certain files in `node_modules` are getting included, and why certain files are being included even though we thought specifying `exclude` would keep them out.

That's why TypeScript now provides an `explainFiles` flag.

```
tsc --explainFiles
```

When using this option, the TypeScript compiler will give some very verbose output about why a file ended up in your program. To read it more easily, you can forward the output to a file, or pipe it to a program that can easily view it.

```

# Forward output to a text file
tsc --explainFiles > explanation.txt

# Pipe output to a utility program like `less`, or an editor like VS Code
tsc --explainFiles | less

tsc --explainFiles | code -

```

Typically, the output will start out by listing out reasons for including `lib.d.ts` files, then for local files, and then `node_modules` files.

```

TS_Compiler_Directory/4.2.2/lib/lib.es5.d.ts
  Library referenced via 'es5' from file
  'TS_Compiler_Directory/4.2.2/lib/lib.es2015.d.ts'
  TS_Compiler_Directory/4.2.2/lib/lib.es2015.d.ts

```

```

Library referenced via 'es2015' from file
'TS_Compiler_Directory/4.2.2/lib/lib.es2016.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.es2016.d.ts
  Library referenced via 'es2016' from file
'TS_Compiler_Directory/4.2.2/lib/lib.es2017.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.es2017.d.ts
  Library referenced via 'es2017' from file
'TS_Compiler_Directory/4.2.2/lib/lib.es2018.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.es2018.d.ts
  Library referenced via 'es2018' from file
'TS_Compiler_Directory/4.2.2/lib/lib.es2019.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.es2019.d.ts
  Library referenced via 'es2019' from file
'TS_Compiler_Directory/4.2.2/lib/lib.es2020.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.es2020.d.ts
  Library referenced via 'es2020' from file
'TS_Compiler_Directory/4.2.2/lib/lib.esnext.d.ts'
TS_Compiler_Directory/4.2.2/lib/lib.esnext.d.ts
  Library 'lib.esnext.d.ts' specified in compilerOptions

... More Library References...

foo.ts
  Matched by include pattern '**/*' in 'tsconfig.json'

```

Right now, we make no guarantees about the output format - it might change over time. On that note, we're interested in improving this format if you have any suggestions!

For more information, [check out the original pull request!](#)

Improved Uncalled Function Checks in Logical Expressions

Thanks to further improvements from [Alex Tarasyuk](#), TypeScript's uncalled function checks now apply within `&&` and `||` expressions.

Under `strictNullChecks`, the following code will now error.

```

function shouldDisplayElement(element: Element) {
  // ...
  return true;
}

function getVisibleItems(elements: Element[]) {
  return elements.filter((e) => shouldDisplayElement && e.children.length);
//           ~~~~~
// This condition will always return true since the function is always defined.
// Did you mean to call it instead.
}

```

For more details, [check out the pull request here.](#)

Destructured Variables Can Be Explicitly Marked as Unused

Thanks to another pull request from [Alex Tarasyuk](#), you can now mark destructured variables as unused by prefixing them with an underscore (the `_` character).

```
let [_first, second] = getValues();
```

Previously, if `_first` was never used later on, TypeScript would issue an error under `noUnusedLocals`. Now, TypeScript will recognize that `_first` was intentionally named with an underscore because there was no intent to use it.

For more details, take a look at [the full change](#).

Relaxed Rules Between Optional Properties and String Index Signatures

String index signatures are a way of typing dictionary-like objects, where you want to allow access with arbitrary keys:

```
const movieWatchCount: { [key: string]: number } = {};
function watchMovie(title: string) {
  movieWatchCount[title] = (movieWatchCount[title] ?? 0) + 1;
}
```

Of course, for any movie title not yet in the dictionary, `movieWatchCount[title]` will be `undefined` (TypeScript 4.1 added the option `noUncheckedIndexedAccess` to include `undefined` when reading from an index signature like this). Even though it's clear that there must be some strings not present in `movieWatchCount`, previous versions of TypeScript treated optional object properties as unassignable to otherwise compatible index signatures, due to the presence of `undefined`.

```
type WesAndersonWatchCount = {
  "Fantastic Mr. Fox?": number;
  "The Royal Tenenbaums?": number;
  "Moonrise Kingdom?": number;
  "The Grand Budapest Hotel?": number;
};

declare const wesAndersonWatchCount: WesAndersonWatchCount;
const movieWatchCount: { [key: string]: number } = wesAndersonWatchCount;
// ~~~~~ error!
// Type 'WesAndersonWatchCount' is not assignable to type '{ [key: string]: number; }'.
//   Property '"Fantastic Mr. Fox"' is incompatible with index signature.
//     Type 'number | undefined' is not assignable to type 'number'.
//       Type 'undefined' is not assignable to type 'number'. (2322)
```

TypeScript 4.2 allows this assignment. However, it does *not* allow the assignment of non-optional properties with `undefined` in their types, nor does it allow writing `undefined` to a specific key:

```
// @errors: 2322
type BatmanWatchCount = {
  "Batman Begins": number | undefined;
  "The Dark Knight": number | undefined;
  "The Dark Knight Rises": number | undefined;
};

declare const batmanWatchCount: BatmanWatchCount;

// Still an error in TypeScript 4.2.
const movieWatchCount: { [key: string]: number } = batmanWatchCount;

// Still an error in TypeScript 4.2.
// Index signatures don't implicitly allow explicit `undefined`.
movieWatchCount["It's the Great Pumpkin, Charlie Brown"] = undefined;
```

The new rule also does not apply to number index signatures, since they are assumed to be array-like and dense:

```
// @errors: 2322
declare let sortOfArrayish: { [key: number]: string };
declare let numberKeys: { 42?: string };

sortOfArrayish = numberKeys;
```

You can get a better sense of this change by [reading up on the original PR](#).

Declare Missing Helper Function

Thanks to a [community pull request](#) from [Alexander Tarasyuk](#), we now have a quick fix for declaring new functions and methods based on the call-site!

```

1
2
3   foo(1, 2, 3)

```

Cannot find name 'foo'. ts(2304)

An un-declared function `foo` being called, with a quick fix scaffolding out the new contents of the file

Breaking Changes

We always strive to minimize breaking changes in a release. TypeScript 4.2 contains some breaking changes, but we believe they should be manageable in an upgrade.

`lib.d.ts` Updates

As with every TypeScript version, declarations for `lib.d.ts` (especially the declarations generated for web contexts), have changed. There are various changes, though `Intl` and `ResizeObserver`'s may end up being the most disruptive.

`noImplicitAny` Errors Apply to Loose `yield` Expressions

When the value of a `yield` expression is captured, but TypeScript can't immediately figure out what type you intend for it to receive (i.e. the `yield` expression isn't contextually typed), TypeScript will now issue an implicit `any` error.

```
// @errors: 7057
function* g1() {
  const value = yield 1;
}

function* g2() {
  // No error.
}
```

```
// The result of `yield 1` is unused.
yield 1;
}

function* g3() {
// No error.
// `yield 1` is contextually typed by 'string'.
const value: string = yield 1;
}

function* g4(): Generator<number, void, string> {
// No error.
// TypeScript can figure out the type of `yield 1`
// from the explicit return type of `g4`.
const value = yield 1;
}
```

See more details in [the corresponding changes](#).

Expanded Uncalled Function Checks

As described above, uncalled function checks will now operate consistently within `&&` and `||` expressions when using `strictNullChecks`. This can be a source of new breaks, but is typically an indication of a logic error in existing code.

Type Arguments in JavaScript Are Not Parsed as Type Arguments

Type arguments were already not allowed in JavaScript, but in TypeScript 4.2, the parser will parse them in a more spec-compliant way. So when writing the following code in a JavaScript file:

```
f<T>(100);
```

TypeScript will parse it as the following JavaScript:

```
f < T > 100;
```

This may impact you if you were leveraging TypeScript's API to parse type constructs in JavaScript files, which may have occurred when trying to parse Flow files.

See [the pull request](#) for more details on what's checked.

Tuple size limits for spreads

Tuple types can be made by using any sort of spread syntax (`...`) in TypeScript.

```
// Tuple types with spread elements
type NumStr = [number, string];
type NumStrNumStr = [...NumStr, ...NumStr];

// Array spread expressions
const numStr = [123, "hello"] as const;
const numStrNumStr = [...numStr, ...numStr] as const;
```

Sometimes these tuple types can accidentally grow to be huge, and that can make type-checking take a long time. Instead of letting the type-checking process hang (which is especially bad in editor scenarios), TypeScript has a limiter in place to avoid doing all that work.

You can [see this pull request](#) for more details.

.d.ts Extensions Cannot Be Used In Import Paths

In TypeScript 4.2, it is now an error for your import paths to contain `.d.ts` in the extension.

```
// must be changed something like
// - "./foo"
// - "./foo.js"
import { Foo } from "./foo.d.ts";
```

Instead, your import paths should reflect whatever your loader will do at runtime. Any of the following imports might be usable instead.

```
import { Foo } from "./foo";
import { Foo } from "./foo.js";
import { Foo } from "./foo/index.js";
```

Reverting Template Literal Inference

This change removed a feature from TypeScript 4.2 beta. If you haven't yet upgraded past our last stable release, you won't be affected, but you may still be interested in the change.

The beta version of TypeScript 4.2 included a change in inference to template strings. In this change, template string literals would either be given template string types or simplify to multiple string literal types. These types would then *widen* to `string` when assigning to mutable variables.

```
declare const yourName: string;

// 'bar' is constant.
// It has type `hello ${string}`.
const bar = `hello ${yourName}`;

// 'baz' is mutable.
// It has type 'string'.
let baz = `hello ${yourName}`;
```

This is similar to how string literal inference works.

```
// 'bar' has type '"hello"'.
const bar = "hello";

// 'baz' has type 'string'.
let baz = "hello";
```

For that reason, we believed that making template string expressions have template string types would be "consistent"; however, from what we've seen and heard, that isn't always desirable.

In response, we've reverted this feature (and potential breaking change). If you *do* want a template string expression to be given a literal-like type, you can always add `as const` to the end of it.

```
declare const yourName: string;  
// 'bar' has type `hello ${string}`.  
const bar = `hello ${yourName}`  
// ^^^^^^ as const;  
  
// 'baz' has type 'string'.  
const baz = `hello ${yourName}`;
```

TypeScript's `lift` Callback in `visitNode` Uses a Different Type

TypeScript has a `visitNode` function that takes a `lift` function. `lift` now expects a `readonly Node[]` instead of a `NodeArray<Node>`. This is technically an API breaking change which you can read more on [here](#).

[Go to TOC](#)

Separate Write Types on Properties

In JavaScript, it's pretty common for APIs to convert values that are passed in before storing them. This often happens with getters and setters too. For example, let's imagine we've got a class with a setter that always converts a value into a `number` before saving it in a private field.

```
class Thing {
  #size = 0;

  get size() {
    return this.#size;
  }

  set size(value) {
    let num = Number(value);

    // Don't allow NaN and stuff.
    if (!Number.isFinite(num)) {
      this.#size = 0;
      return;
    }

    this.#size = num;
  }
}
```

How would we type this JavaScript code in TypeScript? Well, technically we don't have to do anything special here - TypeScript can look at this with no explicit types and can figure out that `size` is a number.

The problem is that `size` allows you to assign more than just `number`s to it. We could get around this by saying that `size` has the type `unknown` or `any` like in this snippet:

```
class Thing {
  // ...
  get size(): unknown {
    return this.#size;
  }
}
```

But that's no good - `unknown` forces people reading `size` to do a type assertion, and `any` won't catch any mistakes. If we really want to model APIs that convert values, previous versions of TypeScript forced us to pick between being precise (which makes reading values easier, and writing harder) and being permissive (which makes writing values easier, and reading harder).

That's why TypeScript 4.3 allows you to specify types for reading and writing to properties.

```
class Thing {
  #size = 0;

  get size(): number {
    return this.#size;
  }

  set size(value: string | number | boolean) {
    let num = Number(value);
  }
}
```

```
// Don't allow NaN and stuff.
if (!Number.isFinite(num)) {
  this.#size = 0;
  return;
}

this.#size = num;
```

In the above example, our `set` accessor takes a broader set of types (`string`s, `boolean`s, and `number`s), but our `get` accessor always guarantees it will be a `number`. Now we can finally assign other types to these properties with no errors!

```
class Thing {
  #size = 0;

  get size(): number {
    return this.#size;
  }

  set size(value: string | number | boolean) {
    let num = Number(value);

    // Don't allow NaN and stuff.
    if (!Number.isFinite(num)) {
      this.#size = 0;
      return;
    }

    this.#size = num;
  }
}

// ---cut---
let thing = new Thing();

// Assigning other types to `thing.size` works!
thing.size = "hello";
thing.size = true;
thing.size = 42;

// Reading `thing.size` always produces a number!
let mySize: number = thing.size;
```

When considering how two properties with the same name relate to each other, TypeScript will only use the "reading" type (e.g. the type on the `get` accessor above). "Writing" types are only considered when directly writing to a property.

Keep in mind, this isn't a pattern that's limited to classes. You can write getters and setters with different types in object literals.

```
function makeThing(): Thing {
  let size = 0;
  return {
    get size(): number {
      return size;
    },
    set size(value: string | number | boolean) {
```

```

let num = Number(value);

// Don't allow NaN and stuff.
if (!Number.isFinite(num)) {
    size = 0;
    return;
}

size = num;
};
}

```

In fact, we've added syntax to interfaces/object types to support different reading/writing types on properties.

```

// Now valid!
interface Thing {
    get size(): number
    set size(value: number | string | boolean);
}

```

One limitation of using different types for reading and writing properties is that the type for reading a property has to be assignable to the type that you're writing. In other words, the getter type has to be assignable to the setter. This ensures some level of consistency, so that a property is always assignable to itself.

For more information on this feature, take a look at [the implementing pull request](#).

override and the --noImplicitOverride Flag

When extending classes in JavaScript, the language makes it super easy (pun intended) to override methods - but unfortunately, there are some mistakes that you can run into.

One big one is missing renames. For example, take the following classes:

```

class SomeComponent {
    show() {
        // ...
    }
    hide() {
        // ...
    }
}

class SpecializedComponent extends SomeComponent {
    show() {
        // ...
    }
    hide() {
        // ...
    }
}

```

`SpecializedComponent` subclasses `SomeComponent`, and overrides the `show` and `hide` methods. What happens if someone decides to rip out `show` and `hide` and replace them with a single method?

```
class SomeComponent {
-   show() {
-     // ...
-   }
-   hide() {
-     // ...
-   }
+   setVisible(value: boolean) {
+     // ...
+   }
}
class SpecializedComponent extends SomeComponent {
  show() {
    // ...
  }
  hide() {
    // ...
  }
}
```

Oh no! Our `SpecializedComponent` didn't get updated. Now it's just adding these two useless `show` and `hide` methods that probably won't get called.

Part of the issue here is that a user can't make it clear whether they meant to add a new method, or to override an existing one. That's why TypeScript 4.3 adds the `override` keyword.

```
class SpecializedComponent extends SomeComponent {
  override show() {
    // ...
  }
  override hide() {
    // ...
  }
}
```

When a method is marked with `override`, TypeScript will always make sure that a method with the same name exists in the base class.

```
// @noImplicitOverride
// @errors: 4113
class SomeComponent {
  setVisible(value: boolean) {
    // ...
  }
}
class SpecializedComponent extends SomeComponent {
  override show() {
    // ...
  }
}
```

This is a big improvement, but it doesn't help if you *forget* to write `override` on a method - and that's a big mistake users can run into also.

For example, you might accidentally "trample over" a method that exists in a base class without realizing it.

```
class Base {
    someHelperMethod() {
        // ...
    }
}

class Derived extends Base {
    // Oops! We weren't trying to override here,
    // we just needed to write a local helper method.
    someHelperMethod() {
        // ...
    }
}
```

That's why TypeScript 4.3 *also* provides a new `noImplicitOverride` flag. When this option is turned on, it becomes an error to override any method from a superclass unless you explicitly use an `override` keyword. In that last example, TypeScript would error under `noImplicitOverride`, and give us a clue that we probably need to rename our method inside of `Derived`.

We'd like to extend our thanks to our community for the implementation here. The work for these items was implemented in a pull request by [Wenlu Wang](#), though an earlier pull request implementing only the `override` keyword by [Paul Cody Johnston](#) served as a basis for direction and discussion. We extend our gratitude for putting in the time for these features.

Template String Type Improvements

In recent versions, TypeScript introduced a new type construct: template string types. These are types that either construct new string-like types by concatenating...

```
type Color = "red" | "blue";
type Quantity = "one" | "two";

type SeussFish = `${Quantity} ${Color} fish`;
// same as
//   type SeussFish = "one fish" | "two fish"
//                   | "red fish" | "blue fish";
```

...or match patterns of other string-like types.

```
declare let s1: `${number}-${number}-${number}`;
declare let s2: `1-2-3`;

// Works!
s1 = s2;
```

The first change we made is just in when TypeScript will infer a template string type. When a template string is *contextually typed* by a string-literal-like type (i.e. when TypeScript sees we're passing a template string to something that takes a literal type) it will try to give that expression a template type.

```
function bar(s: string): `hello ${string}` {
  // Previously an error, now works!
  return `hello ${s}`;
}
```

This also kicks in when inferring types, and the type parameter `extends string`

```
declare let s: string;
declare function f<T extends string>(x: T): T;

// Previously: string
// Now           : `hello ${string}`
let x2 = f(`hello ${s}`);
```

The second major change here is that TypeScript can now better-relate, and *infer between*, different template string types.

To see this, take the following example code:

```
declare let s1: `${number}-${number}-${number}`;
declare let s2: `1-2-3`;
declare let s3: `${number}-2-3`;

s1 = s2;
s1 = s3;
```

When checking against a string literal type like on `s2`, TypeScript could match against the string contents and figure out that `s2` was compatible with `s1` in the first assignment; however, as soon as it saw another template string, it just gave up. As a result, assignments like `s3` to `s1` just didn't work.

TypeScript now actually does the work to prove whether or not each part of a template string can successfully match. You can now mix and match template strings with different substitutions and TypeScript will do a good job to figure out whether they're really compatible.

```
declare let s1: `${number}-${number}-${number}`;
declare let s2: `1-2-3`;
declare let s3: `${number}-2-3`;
declare let s4: `1-${number}-3`;
declare let s5: `1-2-${number}`;
declare let s6: `${number}-2-${number}`;

// Now *all of these* work!
s1 = s2;
s1 = s3;
s1 = s4;
s1 = s5;
s1 = s6;
```

In doing this work, we were also sure to add better inference capabilities. You can see an example of these in action:

```
declare function foo<V extends string>(arg: `*${V}*`): V;
function test<T extends string>(s: string, n: number, b: boolean, t: T) {
  let x1 = foo("hello");           // "hello"
```

```

let x2 = foo(`**hello**`);           // "*hello*"
let x3 = foo(` ${s}` as const);     // string
let x4 = foo(` ${n}` as const);     // ${number}
let x5 = foo(` ${b}` as const);     // "true" | "false"
let x6 = foo(` ${t}` as const);     // ${T}
let x7 = foo(` **${s}**` as const);  // *${string}*
}

```

For more information, see [the original pull request on leveraging contextual types](#), along with [the pull request that improved inference and checking between template types](#).

ECMAScript `#private` Class Elements

TypeScript 4.3 expands which elements in a class can be given `#private` `#names` to make them truly private at run-time. In addition to properties, methods and accessors can also be given private names.

```

class Foo {
  #someMethod() {
    //...
  }

  get #someValue() {
    return 100;
  }

  publicMethod() {
    // These work.
    // We can access private-named members inside this class.
    this.#someMethod();
    return this.#someValue;
  }

  new Foo().#someMethod();
  // ~~~~~
  // error!
  // Property '#someMethod' is not accessible
  // outside class 'Foo' because it has a private identifier.

  new Foo().#someValue;
  // ~~~~~
  // error!
  // Property '#someValue' is not accessible
  // outside class 'Foo' because it has a private identifier.
}

```

Even more broadly, static members can now also have private names.

```

class Foo {
  static #someMethod() {
    // ...
  }

  Foo.#someMethod();
  // ~~~~~
  // error!
  // Property '#someMethod' is not accessible
  // outside class 'Foo' because it has a private identifier.
}

```

This feature was authored in a pull request from our friends at Bloomberg - written by [Titian Cernicova-Dragomir](#) and [Kubilay Kahveci](#), with support and expertise from [Joey Watts](#), [Rob Palmer](#), and [Tim McClure](#). We'd like to extend our thanks to all of them!

ConstructorParameters Works on Abstract Classes

In TypeScript 4.3, the `ConstructorParameters` type helper now works on `abstract` classes.

```
abstract class C {
  constructor(a: string, b: number) {
    // ...
  }
}

// Has the type '[a: string, b: number]'.
type CParams = ConstructorParameters<typeof C>;
```

This is thanks to work done in TypeScript 4.2, where construct signatures can be marked as abstract:

```
type MyConstructorOf<T> = {
  abstract new(...args: any[]): T;
}

// or using the shorthand syntax:
type MyConstructorOf<T> = abstract new (...args: any[]) => T;
```

You can [see the change in more detail on GitHub](#).

Contextual Narrowing for Generics

TypeScript 4.3 now includes some slightly smarter type-narrowing logic on generic values. This allows TypeScript to accept more patterns, and sometimes even catch mistakes.

For some motivation, let's say we're trying to write a function called `makeUnique`. It'll take a `Set` or an `Array` of elements, and if it's given an `Array`, it'll sort that `Array` remove duplicates according to some comparison function. After all that, it will return the original collection.

```
function makeUnique<T>(
  collection: Set<T> | T[],
  comparer: (x: T, y: T) => number
): Set<T> | T[] {
  // Early bail-out if we have a Set.
  // We assume the elements are already unique.
  if (collection instanceof Set) {
    return collection;
  }

  // Sort the array, then remove consecutive duplicates.
  collection.sort(comparer);
  for (let i = 0; i < collection.length; i++) {
    let j = i;
    while (
      j < collection.length &&
```

```

    comparer(collection[i], collection[j + 1]) === 0
  ) {
    j++;
  }
  collection.splice(i + 1, j - i);
}
return collection;
}

```

Let's leave questions about this function's implementation aside, and assume it arose from the requirements of a broader application. Something that you might notice is that the signature doesn't capture the original type of `collection`. We can do that by adding a type parameter called `C` in place of where we've written `Set<T> | T[]`.

```

- function makeUnique<T>(collection: Set<T> | T[], comparer: (x: T, y: T) =>
number): Set<T> | T[]
+ function makeUnique<T, C extends Set<T> | T[]>(collection: C, comparer: (x: T,
y: T) => number): C

```

In TypeScript 4.2 and earlier, you'd end up with a bunch of errors as soon as you tried this.

```

function makeUnique<T, C extends Set<T> | T[]>(
  collection: C,
  comparer: (x: T, y: T) => number
): C {
  // Early bail-out if we have a Set.
  // We assume the elements are already unique.
  if (collection instanceof Set) {
    return collection;
  }

  // Sort the array, then remove consecutive duplicates.
  collection.sort(comparer);
  //
  // error: Property 'sort' does not exist on type 'C'.
  for (let i = 0; i < collection.length; i++) {
    //
    // error: Property 'length' does not exist on type 'C'.
    let j = i;
    while (
      j < collection.length &&
      comparer(collection[i], collection[j + 1]) === 0
    ) {
      //
      // error: Property 'length' does not exist on type 'C'.
      //
      // error: Element implicitly has an 'any' type because expression of type
      'number'
      //       can't be used to index type 'Set<T> | T[]'.
      j++;
    }
    collection.splice(i + 1, j - i);
    //
    // error: Property 'splice' does not exist on type 'C'.
  }
  return collection;
}

```

Ew, errors! Why is TypeScript being so mean to us?

The issue is that when we perform our `collection instanceof Set` check, we're expecting that to act as a type guard that narrows the type from `Set<T> | T[]` to `Set<T>` and `T[]` depending on the branch we're in; however, we're not dealing with a `Set<T> | T[]`, we're trying to narrow the generic value `collection`, whose type is `C`.

It's a very subtle distinction, but it makes a difference. TypeScript can't just grab the constraint of `C` (which is `Set<T> | T[]`) and narrow that. If TypeScript *did* try to narrow from `Set<T> | T[]`, it would forget that `collection` is also a `C` in each branch because there's no easy way to preserve that information. If hypothetically TypeScript tried that approach, it would break the above example in a different way. At the return positions, where the function expects values with the type `C`, we would instead get a `Set<T>` and a `T[]` in each branch, which TypeScript would reject.

```
function makeUnique<T>(
  collection: Set<T> | T[],
  comparer: (x: T, y: T) => number
): Set<T> | T[] {
  // Early bail-out if we have a Set.
  // We assume the elements are already unique.
  if (collection instanceof Set) {
    return collection;
    // ~~~~~
    // error: Type 'Set<T>' is not assignable to type 'C'.
    // 'Set<T>' is assignable to the constraint of type 'C', but
    // 'C' could be instantiated with a different subtype of constraint
    'Set<T> | T[]'.
  }
  // ...
  return collection;
  // ~~~~~
  // error: Type 'T[]' is not assignable to type 'C'.
  // 'T[]' is assignable to the constraint of type 'C', but
  // 'C' could be instantiated with a different subtype of constraint
  'Set<T> | T[]'.
}
```

So how does TypeScript 4.3 change things? Well, basically in a few key places when writing code, all the type system really cares about is the constraint of a type. For example, when we write `collection.length`, TypeScript doesn't care about the fact that `collection` has the type `C`, it only cares about the properties available, which are determined by the constraint `T[] | Set<T>`.

In cases like this, TypeScript will grab the narrowed type of the constraint because that will give you the data you care about; however, in any other case, we'll just try to narrow the original generic type (and often end up with the original generic type).

In other words, based on how you use a generic value, TypeScript will narrow it a little differently. The end result is that the entire above example compiles with no type-checking errors.

For more details, you can [look at the original pull request on GitHub](#).

Always-Truthy Promise Checks

Under `strictNullChecks`, checking whether a `Promise` is "truthy" in a conditional will trigger an error.

```
async function foo(): Promise<boolean> {
    return false;
}

async function bar(): Promise<string> {
    if (foo()) {
        // ~~~~~
        // Error!
        // This condition will always return true since
        // this 'Promise<boolean>' appears to always be defined.
        // Did you forget to use 'await'?
        return "true";
    }
    return "false";
}
```

This change was contributed by [Jack Works](#), and we extend our thanks to them!

static Index Signatures

Index signatures allow us set more properties on a value than a type explicitly declares.

```
class Foo {
    hello = "hello";
    world = 1234;

    // This is an index signature:
    [propName: string]: string | number | undefined;
}

let instance = new Foo();

// Valid assignment
instance["whatever"] = 42;

// Has type 'string | number | undefined'.
let x = instance["something"];
```

Up until now, an index signature could only be declared on the instance side of a class. Thanks to a [pull request](#) from [Wenlu Wang](#), index signatures can now be declared as `static`.

```
class Foo {
    static hello = "hello";
    static world = 1234;

    static [propName: string]: string | number | undefined;
}

// Valid.
Foo["whatever"] = 42;

// Has type 'string | number | undefined'.
let x = Foo["something"];
```

The same sorts of rules apply for index signatures on the static side of a class as they do for the instance side - namely, that every other static property has to be compatible with the index signature.

```
class Foo {
  static prop = true;
  // ~~~
  // Error! Property 'prop' of type 'boolean'
  // is not assignable to string index type
  // 'string | number | undefined'.
  static [propName: string]: string | number | undefined;
}
```

.tsbuildinfo Size Improvements

In TypeScript 4.3, `.tsbuildinfo` files that are generated as part of `incremental` builds should be significantly smaller. This is thanks to several optimizations in the internal format, creating tables with numeric identifiers to be used throughout the file instead of repeating full paths and similar information. This work was spear-headed by [Tobias Koppers](#) in [their pull request](#), serving as inspiration for [the ensuing pull request](#) and [further optimizations](#).

We have seen significant reductions of `.tsbuildinfo` file sizes including

- 1MB to 411 KB
- 14.9MB to 1MB
- 1345MB to 467MB

Needless to say, these sorts of savings in size translate to slightly faster build times as well.

Lazier Calculations in `--incremental` and `--watch` Compilations

One of the issues with `incremental` and `--watch` modes are that while they make later compilations go faster, the initial compilation can be a bit slower - in some cases, significantly slower. This is because these modes have to perform a bunch of book-keeping, computing information about the current project, and sometimes saving that data in a `.tsbuildinfo` file for later builds.

That's why on top of `.tsbuildinfo` size improvements, TypeScript 4.3 also ships some changes to `incremental` and `--watch` modes that make the first build of a project with these flags just as fast as an ordinary build! To do this, much of the information that would ordinarily be computed up-front is instead done on an on-demand basis for later builds. While this can add some overhead to a subsequent build, TypeScript's `incremental` and `--watch` functionality will still typically operate on a much smaller set of files, and any needed information will be saved afterwards. In a sense, `incremental` and `--watch` builds will "warm up" and get faster at compiling files once you've updated them a few times.

In a repository with 3000 files, **this reduced initial build times to almost a third!**

This work was started by [Tobias Koppers](#), whose work ensued in [the resulting final change](#) for this functionality. We'd like to extend a great thanks to Tobias for helping us find these opportunities for improvements!

Import Statement Completions

One of the biggest pain-points users run into with import and export statements in JavaScript is the order - specifically that imports are written as

```
import { func } from "./module.js";
```

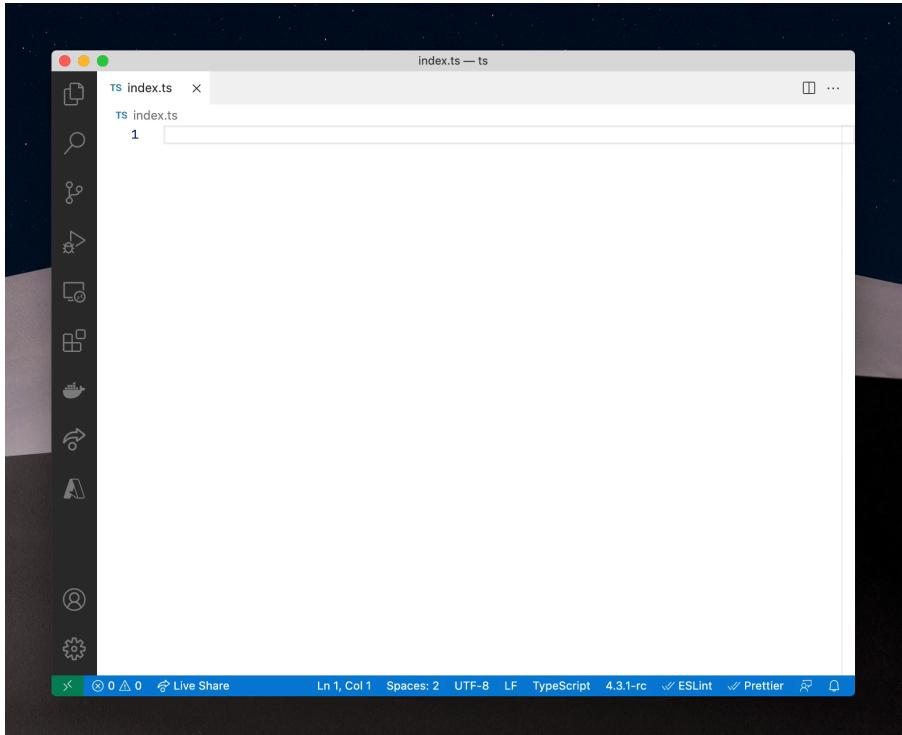
instead of

```
from "./module.js" import { func };
```

This causes some pain when writing out a full import statement from scratch because auto-complete wasn't able to work correctly. For example, if you start writing something like `import {`, TypeScript has no idea what module you're planning on importing from, so it couldn't provide any scoped-down completions.

To alleviate this, we've leveraged the power of auto-imports! Auto-imports already deal with the issue of not being able to narrow down completions from a specific module - their whole point is to provide every possible export and automatically insert an import statement at the top of your file.

So when you now start writing an `import` statement that doesn't have a path, we'll provide you with a list of possible imports. When you commit a completion, we'll complete the full import statement, including the path that you were going to write.



Import statement completions

This work requires editors that specifically support the feature. You'll be able to try this out by using the latest [Insiders versions of Visual Studio Code](#).

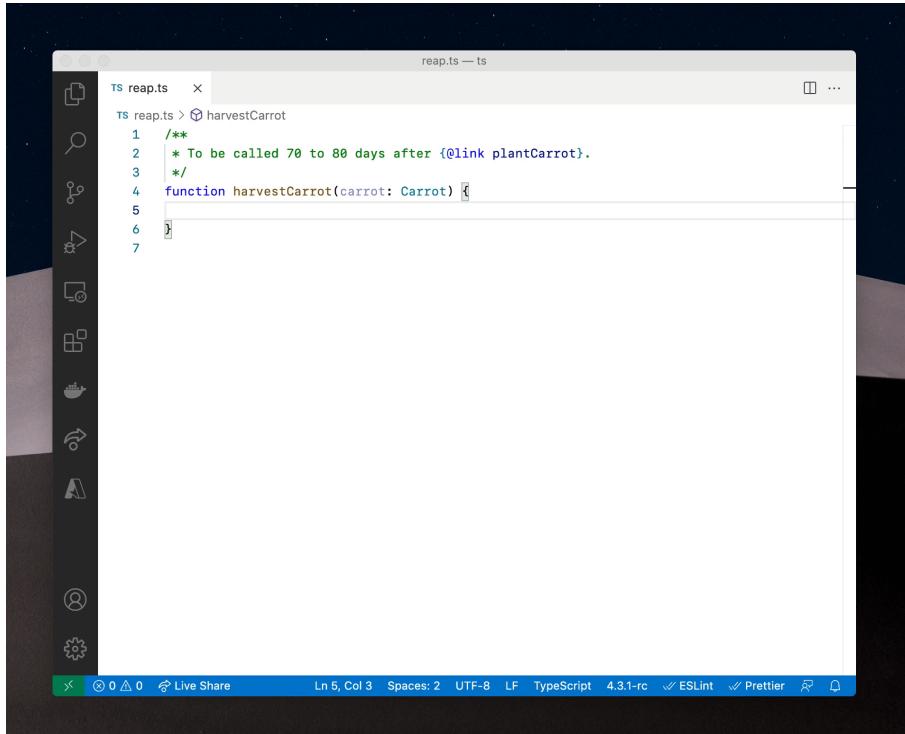
For more information, take a look at [the implementing pull request](#)!

Editor Support for `@link` Tags

TypeScript can now understand `@link` tags, and will try to resolve declarations that they link to. What this means is that you'll be able to hover over names within `@link` tags and get quick information, or use commands like go-to-definition or find-all-references.

For example, you'll be able to go-to-definition on `bar` in `@link bar` in the example below and a TypeScript-supported editor will jump to `bar`'s function declaration.

```
/**  
 * To be called 70 to 80 days after {@link plantCarrot}.  
 */  
function harvestCarrot(carrot: Carrot) {}  
  
/**  
 * Call early in spring for best results. Added in v2.1.0.  
 * @param seed Make sure it's a carrot seed!  
 */  
function plantCarrot(seed: Seed) {  
    // TODO: some gardening  
}
```



Jumping to definition and requesting quick info on a `@link` tag for

For more information, see [the pull request on GitHub!](#)

Go-to-Definition on Non-JavaScript File Paths

Many loaders allow users to include assets in their applications using JavaScript imports. They'll typically be written as something like `import "./styles.css"` or the like.

Up until now, TypeScript's editor functionality wouldn't even attempt to read this file, so go-to-definition would typically fail. At best, go-to-definition would jump to a declaration like `declare module "*.css"` if it could find something along those lines.

TypeScript's language service now tries to jump to the correct file when you perform a go-to-definition on relative file paths, even if they're not JavaScript or TypeScript files! Try it out with imports to CSS, SVGs, PNGs, font files, Vue files, and more.

For more information, you can check out [the implementing pull request](#).

Breaking Changes

`lib.d.ts` Changes

As with every TypeScript version, declarations for `lib.d.ts` (especially the declarations generated for web contexts), have changed. In this release, we leveraged Mozilla's [browser-compat-data](#) to remove APIs that no browser implements. While it is unlike that you are using them, APIs such as `Account`, `AssertionOptions`, `RTCStatsEventInit`, `MSGestureEvent`, `DeviceLightEvent`, `MSPointerEvent`, `ServiceWorkerMessageEvent`, and `WebAuthentication` have all been removed from `lib.d.ts`. This is discussed [in some detail here](#).

<https://github.com/microsoft/TypeScript-DOM-lib-generator/issues/991>

`useDefineForClassFields` now defaults to true on `esnext` and eventually on `es2022`

In 2021 the class fields feature was added into the JavaScript specification with behavior which differed from how TypeScript had implemented it. In preparation for this, in TypeScript 3.7, a flag was added (`useDefineForClassFields`) to migrate to emitted JavaScript to match the JavaScript standard behavior.

Now that the feature is in JavaScript we are changing the default to `true` for ES2022 and above, including ESNext.

Errors on Always-Truthy Promise Checks

Under `strictNullChecks`, using a `Promise` that always appears to be defined within a condition check is now considered an error.

```
declare var p: Promise<number>;
if (p) {
  // ...
  // Error!
  // This condition will always return true since
```

```
// this 'Promise<number>' appears to always be defined.
//
// Did you forget to use 'await'?
}
```

For more details, see [the original change](#).

Union Enums Cannot Be Compared to Arbitrary Numbers

Certain `enum`s are considered *union enum*s when their members are either automatically filled in, or trivially written. In those cases, an enum can recall each value that it potentially represents.

In TypeScript 4.3, if a value with a union `enum` type is compared with a numeric literal that it could never be equal to, then the type-checker will issue an error.

```
enum E {
  A = 0,
  B = 1,
}

function doSomething(x: E) {
  // Error! This condition will always return 'false' since the types 'E' and '-1'
  // have no overlap.
  if (x === -1) {
    // ...
  }
}
```

As a workaround, you can re-write an annotation to include the appropriate literal type.

```
enum E {
  A = 0,
  B = 1,
}

// Include -1 in the type, if we're really certain that -1 can come through.
function doSomething(x: E | -1) {
  if (x === -1) {
    // ...
  }
}
```

You can also use a type-assertion on the value.

```
enum E {
  A = 0,
  B = 1,
}

function doSomething(x: E) {
  // Use a type assertion on 'x' because we know we're not actually just dealing
  // with values from 'E'.
  if ((x as number) === -1) {
    // ...
  }
}
```

Alternatively, you can re-declare your enum to have a non-trivial initializer so that any number is both assignable and comparable to that enum. This may be useful if the intent is for the enum to specify a few well-known values.

```
enum E {  
    // the leading + on 0 opts TypeScript out of inferring a union enum.  
    A = +0,  
    B = 1,  
}
```

For more details, [see the original change](#)

[Go to TOC](#)

Control Flow Analysis of Aliased Conditions and Discriminants

In JavaScript, we often have to probe a value in different ways, and do something different once we know more about its type. TypeScript understands these checks and calls them *type guards*. Instead of having to convince TypeScript of a variable's type whenever we use it, the type-checker leverages something called *control flow analysis* to see if we've used a type guard before a given piece of code.

For example, we can write something like

```
function foo(arg: unknown) {
  if (typeof arg === "string") {
    // ^?
    console.log(arg.toUpperCase());
  }
}
```

In this example, we checked whether `arg` was a `string`. TypeScript recognized the `typeof arg === "string"` check, which it considered a type guard, and knew that `arg` was a `string` inside the body of the `if` block. That let us access `string` methods like `toUpperCase()` without getting an error.

However, what would happen if we moved the condition out to a constant called `argIsString`?

```
// In TS 4.3 and below

function foo(arg: unknown) {
  const argIsString = typeof arg === "string";
  if (argIsString) {
    console.log(arg.toUpperCase());
    // ~~~~~
    // Error! Property 'toUpperCase' does not exist on type 'unknown'.
  }
}
```

In previous versions of TypeScript, this would be an error - even though `argIsString` was assigned the value of a type guard, TypeScript simply lost that information. That's unfortunate since we might want to re-use the same check in several places. To get around that, users often have to repeat themselves or use type assertions (a.k.a. casts).

In TypeScript 4.4, that is no longer the case. The above example works with no errors! When TypeScript sees that we are testing a constant value, it will do a little bit of extra work to see if it contains a type guard. If that type guard operates on a `const`, a `readonly` property, or an un-modified parameter, then TypeScript is able to narrow that value appropriately.

Different sorts of type guard conditions are preserved - not just `typeof` checks. For example, checks on discriminated unions work like a charm.

```
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "square"; sideLength: number };
```

```
function area(shape: Shape): number {
  const isCircle = shape.kind === "circle";
  if (isCircle) {
    // We know we have a circle here!
    return Math.PI * shape.radius ** 2;
  } else {
    // We know we're left with a square here!
    return shape.sideLength ** 2;
  }
}
```

Analysis on discriminants in 4.4 also goes a little bit deeper - we can now extract out discriminants and TypeScript can narrow the original object.

```
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "square"; sideLength: number };

function area(shape: Shape): number {
  // Extract out the 'kind' field first.
  const { kind } = shape;

  if (kind === "circle") {
    // We know we have a circle here!
    return Math.PI * shape.radius ** 2;
  } else {
    // We know we're left with a square here!
    return shape.sideLength ** 2;
  }
}
```

As another example, here's a function that checks whether two of its inputs have contents.

```
function doSomeChecks(
  inputA: string | undefined,
  inputB: string | undefined,
  shouldDoExtraWork: boolean
) {
  const mustDoWork = inputA && inputB && shouldDoExtraWork;
  if (mustDoWork) {
    // We can access 'string' properties on both 'inputA' and 'inputB'!
    const upperA = inputA.toUpperCase();
    const upperB = inputB.toUpperCase();
    // ...
  }
}
```

TypeScript can understand that both `inputA` and `inputB` are both present if `mustDoWork` is `true`. That means we don't have to write a non-null assertion like `inputA!` to convince TypeScript that `inputA` isn't `undefined`.

One neat feature here is that this analysis works transitively. TypeScript will hop through constants to understand what sorts of checks you've already performed.

```
function f(x: string | number | boolean) {
  const isString = typeof x === "string";
  const isNumber = typeof x === "number";
```

```
const isStringOrNumber = isString || isNumber;
if (isStringOrNumber) {
    x;
    // ^
} else {
    x;
    // ^
}
```

Note that there's a cutoff - TypeScript doesn't go arbitrarily deep when checking these conditions, but its analysis is deep enough for most checks.

This feature should make a lot of intuitive JavaScript code "just work" in TypeScript without it getting in your way. For more details, [check out the implementation on GitHub!](#)

Symbol and Template String Pattern Index Signatures

TypeScript lets us describe objects where every property has to have a certain type using *index signatures*. This allows us to use these objects as dictionary-like types, where we can use string keys to index into them with square brackets.

For example, we can write a type with an index signature that takes `string` keys and maps to `boolean` values. If we try to assign anything other than a `boolean` value, we'll get an error.

```
// @errors: 2322 2375
interface BooleanDictionary {
    [key: string]: boolean;
}

declare let myDict: BooleanDictionary;

// Valid to assign boolean values
myDict["foo"] = true;
myDict["bar"] = false;

// Error, "oops" isn't a boolean
myDict["baz"] = "oops";
```

While a `Map` might be a better data structure here (specifically, a `Map<string, boolean>`), JavaScript objects are often more convenient to use or just happen to be what we're given to work with.

Similarly, `Array<T>` already defines a `number` index signature that lets us insert/retrieve values of type `T`.

```
// @errors: 2322 2375
// This is part of TypeScript's definition of the built-in Array type.
interface Array<T> {
    [index: number]: T;
    // ...
}
```

```
let arr = new Array<string>();
// Valid
arr[0] = "hello!";
// Error, expecting a 'string' value here
arr[1] = 123;
```

Index signatures are very useful to express lots of code out in the wild; however, until now they've been limited to `string` and `number` keys (and `string` index signatures have an intentional quirk where they can accept `number` keys since they'll be coerced to strings anyway). That means that TypeScript didn't allow indexing objects with `symbol` keys. TypeScript also couldn't model an index signature of some *subset* of `string` keys - for example, an index signature which describes just properties whose names start with the text `data-`.

TypeScript 4.4 addresses these limitations, and allows index signatures for `symbol`s and template string patterns.

For example, TypeScript now allows us to declare a type that can be keyed on arbitrary `symbol`s.

```
// @errors: 2322 2375
interface Colors {
  [sym: symbol]: number;
}

const red = Symbol("red");
const green = Symbol("green");
const blue = Symbol("blue");

let colors: Colors = {};
// Assignment of a number is allowed
colors[red] = 255;
let redVal = colors[red];
// ^
colors[blue] = "da ba dee";
```

Similarly, we can write an index signature with template string pattern type. One use of this might be to exempt properties starting with `data-` from TypeScript's excess property checking. When we pass an object literal to something with an expected type, TypeScript will look for excess properties that weren't declared in the expected type.

```
// @errors: 2322 2375
interface Options {
  width?: number;
  height?: number;
}

let a: Options = {
  width: 100,
  height: 100,
  "data-blah": true,
};
```

```

interface OptionsWithDataProps extends Options {
  // Permit any property starting with 'data-'
  [optName: `data-${string}`]: unknown;
}

let b: OptionsWithDataProps = {
  width: 100,
  height: 100,
  "data-blah": true,

  // Fails for a property which is not known, nor
  // starts with 'data-'
  "unknown-property": true,
};

```

A final note on index signatures is that they now permit union types, as long as they're a union of infinite-domain primitive types - specifically:

- `string`
- `number`
- `symbol`
- template string patterns (e.g. ``hello-${string}``)

An index signature whose argument is a union of these types will de-sugar into several different index signatures.

```

interface Data {
  [optName: string | symbol]: any;
}

// Equivalent to

interface Data {
  [optName: string]: any;
  [optName: symbol]: any;
}

```

For more details, [read up on the pull request](#)

Defaulting to the `unknown` Type in Catch Variables (--useUnknownInCatchVariables)

In JavaScript, any type of value can be thrown with `throw` and caught in a `catch` clause. Because of this, TypeScript historically typed catch clause variables as `any`, and would not allow any other type annotation:

```

try {
  // Who knows what this might throw...
  executeSomeThirdPartyCode();
} catch (err) {
  // err: any
  console.error(err.message); // Allowed, because 'any'
  err.thisWillProbablyFail(); // Allowed, because 'any' :(
}

```

Once TypeScript added the `unknown` type, it became clear that `unknown` was a better choice than `any` in `catch` clause variables for users who want the highest degree of correctness and type-safety, since it narrows better and forces us to test against arbitrary values. Eventually TypeScript 4.0 allowed users to specify an explicit type annotation of `unknown` (or `any`) on each `catch` clause variable so that we could opt into stricter types on a case-by-case basis; however, for some, manually specifying `: unknown` on every `catch` clause was a chore.

That's why TypeScript 4.4 introduces a new flag called `useUnknownInCatchVariables`. This flag changes the default type of `catch` clause variables from `any` to `unknown`.

```
// @errors: 2571
declare function executeSomeThirdPartyCode(): void;
// ---cut---
try {
  executeSomeThirdPartyCode();
} catch (err) {
  // err: unknown

  // Error! Property 'message' does not exist on type 'unknown'.
  console.error(err.message);

  // Works! We can narrow 'err' from 'unknown' to 'Error'.
  if (err instanceof Error) {
    console.error(err.message);
  }
}
```

This flag is enabled under the `strict` family of options. That means that if you check your code using `strict`, this option will automatically be turned on. You may end up with errors in TypeScript 4.4 such as

```
Property 'message' does not exist on type 'unknown'.
Property 'name' does not exist on type 'unknown'.
Property 'stack' does not exist on type 'unknown'.
```

In cases where we don't want to deal with an `unknown` variable in a `catch` clause, we can always add an explicit `: any` annotation so that we can opt *out* of stricter types.

```
declare function executeSomeThirdPartyCode(): void;
// ---cut---
try {
  executeSomeThirdPartyCode();
} catch (err: any) {
  console.error(err.message); // Works again!
}
```

For more information, take a look at [the implementing pull request](#).

Exact Optional Property Types (--exactOptionalPropertyTypes)

In JavaScript, reading a *missing* property on an object produces the value `undefined`. It's also possible to have an actual property with the value `undefined`. A lot of code in JavaScript tends to treat these situations the same way, and so initially TypeScript just interpreted every optional property as if a user had written `undefined` in the type. For example,

```
interface Person {
  name: string;
  age?: number;
}
```

was considered equivalent to

```
interface Person {
  name: string;
  age?: number | undefined;
}
```

What this meant is that a user could explicitly write `undefined` in place of `age`.

```
const p: Person = {
  name: "Daniel",
  age: undefined, // This is okay by default.
};
```

So by default, TypeScript doesn't distinguish between a present property with the value `undefined` and a missing property. While this works most of the time, not all code in JavaScript makes the same assumptions. Functions and operators like `Object.assign`, `Object.keys`, object spread (`{ ...obj }`), and `for-in` loops behave differently depending on whether or not a property actually exists on an object. In the case of our `Person` example, this could potentially lead to runtime errors if the `age` property was observed in a context where its presence was important.

In TypeScript 4.4, the new flag `exactOptionalPropertyTypes` specifies that optional property types should be interpreted exactly as written, meaning that `| undefined` is not added to the type:

```
// @exactOptionalPropertyTypes
// @errors: 2322 2375
interface Person {
  name: string;
  age?: number;
}
// ---cut---
// With 'exactOptionalPropertyTypes' on:
const p: Person = {
  name: "Daniel",
  age: undefined, // Error! undefined isn't a number
};
```

This flag is **not** part of the `strict` family and needs to be turned on explicitly if you'd like this behavior. It also requires `strictNullChecks` to be enabled as well. We've been making updates to DefinitelyTyped and other definitions to try to make the transition as straightforward as possible, but you may encounter some friction with this depending on how your code is structured.

For more information, you can [take a look at the implementing pull request here](#).

static Blocks in Classes

TypeScript 4.4 brings support for `static blocks in classes`, an upcoming ECMAScript feature that can help you write more-complex initialization code for static members.

```
declare function someCondition(): boolean
// ---cut---
class Foo {
  static count = 0;

  // This is a static block:
  static {
    if (someCondition()) {
      Foo.count++;
    }
  }
}
```

These static blocks allow you to write a sequence of statements with their own scope that can access private fields within the containing class. That means that we can write initialization code with all the capabilities of writing statements, no leakage of variables, and full access to our class's internals.

```
declare function loadLastInstances(): any[]
// ---cut---
class Foo {
  static #count = 0;

  get count() {
    return Foo.#count;
  }

  static {
    try {
      const lastInstances = loadLastInstances();
      Foo.#count += lastInstances.length;
    }
    catch {}
  }
}
```

Without `static` blocks, writing the code above was possible, but often involved several different types of hacks that had to compromise in some way.

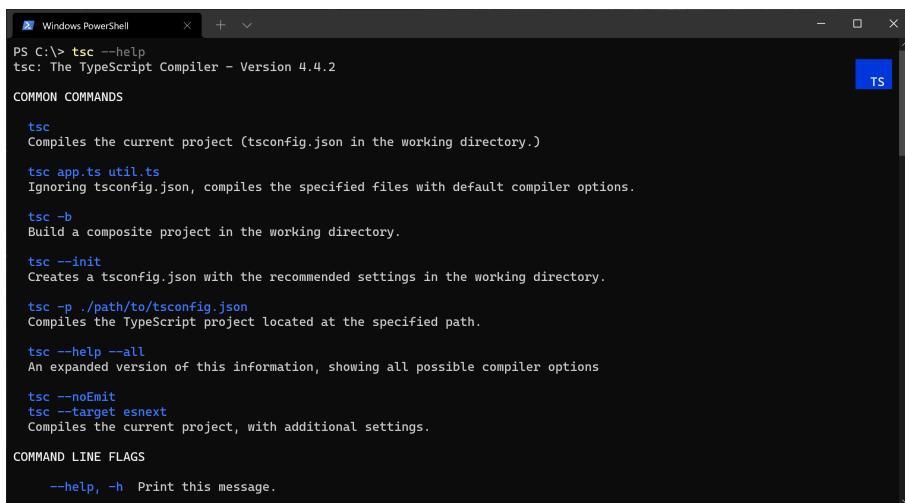
Note that a class can have multiple `static` blocks, and they're run in the same order in which they're written.

```
// Prints:
// 1
// 2
// 3
class Foo {
  static prop = 1
  static {
    console.log(Foo.prop++);
  }
  static {
    console.log(Foo.prop++);
  }
  static {
    console.log(Foo.prop++);
  }
}
```

We'd like to extend our thanks to [Wenlu Wang](#) for TypeScript's implementation of this feature. For more details, you can [see that pull request here](#).

tsc --help Updates and Improvements

TypeScript's `--help` option has gotten a refresh! Thanks to work in part by [Song Gao](#), we've brought in changes to [update the descriptions of our compiler options](#) and [restyle the `--help` menu](#) with colors and other visual separation.



```
Windows PowerShell
ps C:\> tsc --help
tsc: The TypeScript Compiler - Version 4.4.2
TS

COMMON COMMANDS

tsc
Compiles the current project (tsconfig.json in the working directory.)

tsc app.ts util.ts
Ignoring tsconfig.json, compiles the specified files with default compiler options.

tsc -b
Build a composite project in the working directory.

tsc --init
Creates a tsconfig.json with the recommended settings in the working directory.

tsc -p ./path/to/tsconfig.json
Compiles the TypeScript project located at the specified path.

tsc --help --all
An expanded version of this information, showing all possible compiler options

tsc --noEmit
tsc --target esnext
Compiles the current project, with additional settings.

COMMAND LINE FLAGS

--help, -h Print this message.
```

The new TypeScript `--help` menu where the output is bucketed into several different areas

You can read more on [the original proposal thread](#).

Performance Improvements

Faster Declaration Emit

TypeScript now caches whether internal symbols are accessible in different contexts, along with how specific types should be printed. These changes can improve TypeScript's general performance in code with fairly complex types, and is especially observed when emitting `.d.ts` files under the `declaration` flag.

[See more details here.](#)

Faster Path Normalization

TypeScript often has to do several types of "normalization" on file paths to get them into a consistent format that the compiler can use everywhere. This involves things like replacing backslashes with slashes, or removing intermediate `./` and `../` segments of paths. When TypeScript has to operate over millions of these paths, these operations end up being a bit slow. In TypeScript 4.4, paths first undergo quick checks to see whether they need any normalization in the first place. These improvements together reduce project load time by 5-10% on bigger projects, and significantly more in massive projects that we've tested internally.

For more details, you can [view the PR for path segment normalization](#) along with [the PR for slash normalization](#).

Faster Path Mapping

TypeScript now caches the way it constructs path-mappings (using the `paths` option in `tsconfig.json`). For projects with several hundred mappings, the reduction is significant. You can see more [on the change itself](#).

Faster Incremental Builds with `--strict`

In what was effectively a bug, TypeScript would end up redoing type-checking work under `incremental` compilations if `strict` was on. This led to many builds being just as slow as if `incremental` was turned off. TypeScript 4.4 fixes this, though the change has also been back-ported to TypeScript 4.3.

[See more here.](#)

Faster Source Map Generation for Big Outputs

TypeScript 4.4 adds an optimization for source map generation on extremely large output files. When building an older version of the TypeScript compiler, this results in around an 8% reduction in emit time.

We'd like to extend our thanks to [David Michon](#) who provided a [simple and clean change](#) to enable this performance win.

Faster `--force` Builds

When using `--build` mode on project references, TypeScript has to perform up-to-date checks to determine which files need to be rebuilt. When performing a `--force` build, however, that information is irrelevant since every project dependency will be rebuilt from scratch. In TypeScript 4.4, `--force` builds avoid those unnecessary steps and start a full build. See more about the change [here](#).

Spelling Suggestions for JavaScript

TypeScript powers the JavaScript editing experience in editors like Visual Studio and Visual Studio Code. Most of the time, TypeScript tries to stay out of the way in JavaScript files; however, TypeScript often has a lot of information to make confident suggestions, and ways of surfacing suggestions that aren't too invasive.

That's why TypeScript now issues spelling suggestions in plain JavaScript files - ones without `// @ts-check` or in a project with `checkJs` turned off. These are the same "Did you mean...?" suggestions that TypeScript files already have, and now they're available in *all* JavaScript files in some form.

These spelling suggestions can provide a subtle clue that your code is wrong. We managed to find a few bugs in existing code while testing this feature!

For more details on this new feature, [take a look at the pull request!](#)

Inlay Hints

TypeScript 4.4 provides support for *inlay hints* which can help display useful information like parameter names and return types in your code. You can think of it as a sort of friendly "ghost text".

```
export function getFavoriteColor(dayOfWeek: string) : Color | undefined {
    if (dayOfWeek === "Tuesday") {
        return new Color(
            kind: "rgb",
            red: 0x64,
            green: 0x95,
            blue: 0xED
        );
    }
}
```

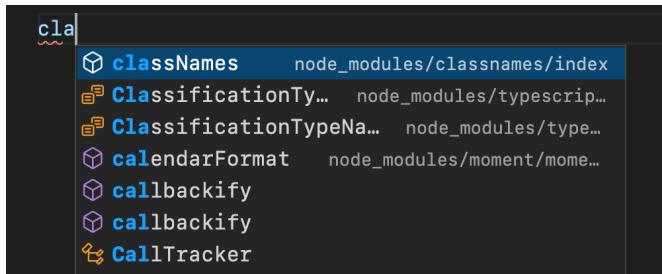
A preview of inlay hints in Visual Studio Code

This feature was built by [Wenlu Wang](#) whose [pull request](#) has more details.

Wenlu also contributed the integration for inlay hints in Visual Studio Code which has shipped as part of the July 2021 (1.59) release. If you'd like to try inlay hints out, make sure you're using a recent stable or insiders version of the editor. You can also modify when and where inlay hints get displayed in Visual Studio Code's settings.

Auto-Imports Show True Paths in Completion Lists

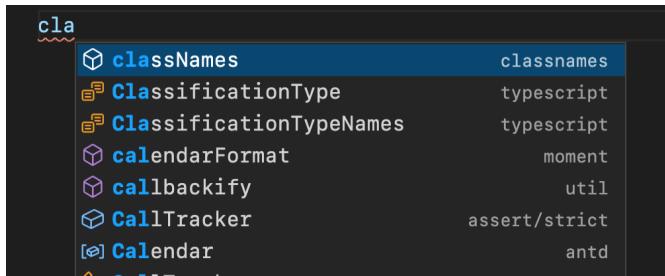
When editors like Visual Studio Code show a completion list, completions which include auto-imports are displayed with a path to the given module; however, this path usually isn't what TypeScript ends up placing in a module specifier. The path is usually something relative to the *workspace*, meaning that if you're importing from a package like `moment`, you'll often see a path like `node_modules/moment`.



A completion list containing unwieldy paths containing 'node_modules'. For example, the label for 'calendarFormat' is 'node_modules/moment/moment' instead of 'moment'.

These paths end up being unwieldy and often misleading, especially given that the path that actually gets inserted into your file needs to consider Node's `node_modules` resolution, path mappings, symlinks, and re-exports.

That's why with TypeScript 4.4, the completion item label now shows the *actual* module path that will be used for the import!



A completion list containing clean paths with no intermediate 'node_modules'. For example, the label for 'calendarFormat' is 'moment' instead of 'node_modules/moment/moment'.

Since this calculation can be expensive, completion lists containing many auto-imports may fill in the final module specifiers in batches as you type more characters. It's possible that you'll still sometimes see the old workspace-relative path labels; however, as your editing experience "warms up", they should get replaced with the actual path after another keystroke or two.

Breaking Changes

`lib.d.ts` Changes for TypeScript 4.4

As with every TypeScript version, declarations for `lib.d.ts` (especially the declarations generated for web contexts), have changed. You can consult our [list of known `lib.dom.d.ts` changes](#) to understand what is impacted.

More-Compliant Indirect Calls for Imported Functions

In earlier versions of TypeScript, calling an import from CommonJS, AMD, and other non-ES module systems would set the `this` value of the called function. Specifically, in the following example, when calling `fooModule.foo()`, the `foo()` method will have `fooModule` set as the value of `this`.

```
// Imagine this is our imported module, and it has an export named 'foo'.
let fooModule = {
  foo() {
    console.log(this);
  },
};
fooModule.foo();
```

This is not the way exported functions in ECMAScript are supposed to work when we call them. That's why TypeScript 4.4 intentionally discards the `this` value when calling imported functions, by using the following emit.

```
// Imagine this is our imported module, and it has an export named 'foo'.
let fooModule = {
  foo() {
```

```

        console.log(this);
    },
}

// Notice we're actually calling '(0, fooModule.foo)' now, which is subtly
// different.
(0, fooModule.foo)();

```

You can [read up more about the changes here](#).

Using `unknown` in Catch Variables

Users running with the `strict` flag may see new errors around `catch` variables being `unknown`, especially if the existing code assumes only `Error` values have been caught. This often results in error messages such as:

```

Property 'message' does not exist on type 'unknown'.
Property 'name' does not exist on type 'unknown'.
Property 'stack' does not exist on type 'unknown'.

```

To get around this, you can specifically add runtime checks to ensure that the thrown type matches your expected type. Otherwise, you can just use a type assertion, add an explicit `: any` to your catch variable, or turn off `useUnknownInCatchVariables`.

Broader Always-Truthy Promise Checks

In prior versions, TypeScript introduced "Always Truthy Promise checks" to catch code where an `await` may have been forgotten; however, the checks only applied to named declarations. That meant that while this code would correctly receive an error...

```

async function foo(): Promise<boolean> {
    return false;
}

async function bar(): Promise<string> {
    const fooResult = foo();
    if (fooResult) {
        // <- error! :D
        return "true";
    }
    return "false";
}

```

...the following code would not.

```

async function foo(): Promise<boolean> {
    return false;
}

async function bar(): Promise<string> {
    if (foo()) {
        // <- no error :( 
        return "true";
    }
}

```

```
    }
    return "false";
}
```

TypeScript 4.4 now flags both. For more information, [read up on the original change](#).

Abstract Properties Do Not Allow Initializers

The following code is now an error because abstract properties may not have initializers:

```
abstract class C {
    abstract prop = 1;
    // ~~~~
    // Property 'prop' cannot have an initializer because it is marked abstract.
}
```

Instead, you may only specify a type for the property:

```
abstract class C {
    abstract prop: number;
}
```

[Go to TOC](#)

Supporting `lib` from `node_modules`

To ensure that TypeScript and JavaScript support works well out of the box, TypeScript bundles a series of declaration files (`.d.ts` files). These declaration files represent the available APIs in the JavaScript language, and the standard browser DOM APIs. While there are some reasonable defaults based on your `target`, you can pick and choose which declaration files your program uses by configuring the `lib` setting in the `tsconfig.json`.

There are two occasional downsides to including these declaration files with TypeScript though:

- When you upgrade TypeScript, you're also forced to handle changes to TypeScript's built-in declaration files, and this can be a challenge when the DOM APIs change as frequently as they do.
- It is hard to customize these files to match your needs with the needs of your project's dependencies (e.g. if your dependencies declare that they use the DOM APIs, you might also be forced into using the DOM APIs).

TypeScript 4.5 introduces a way to override a specific built-in `lib` in a manner similar to how `@types/` support works. When deciding which `lib` files TypeScript should include, it will first look for a scoped `@typescript/lib-*` package in `node_modules`. For example, when including `dom` as an option in `lib`, TypeScript will use the types in `node_modules/@typescript/lib-dom` if available.

You can then use your package manager to install a specific package to take over for a given `lib`. For example, today TypeScript publishes versions of the DOM APIs on `@types/web`. If you wanted to lock your project to a specific version of the DOM APIs, you could add this to your `package.json`:

```
{
  "dependencies": {
    "@typescript/lib-dom": "npm:@types/web"
  }
}
```

Then from 4.5 onwards, you can update TypeScript and your dependency manager's lockfile will ensure that it uses the exact same version of the DOM types. That means you get to update your types on your own terms.

We'd like to give a shout-out to [saschanaz](#) who has been extremely helpful and patient as we've been building out and experimenting with this feature.

For more information, you can [see the implementation of this change](#).

The `Awaited` Type and `Promise` Improvements

TypeScript 4.5 introduces a new utility type called the `Awaited` type. This type is meant to model operations like `await` in `async` functions, or the `.then()` method on `Promises` - specifically, the way that they recursively unwrap `Promises`.

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

The `Awaited` type can be helpful for modeling existing APIs, including JavaScript built-ins like `Promise.all`, `Promise.race`, etc. In fact, some of the problems around inference with `Promise.all` served as motivations for `Awaited`. Here's an example that fails in TypeScript 4.4 and earlier.

```
declare function MaybePromise<T>(value: T): T | Promise<T> | PromisLike<T>;
async function doSomething(): Promise<[number, number]> {
  const result = await Promise.all([MaybePromise(100), MaybePromise(200)]);
  // Error!
  // [number | Promise<100>, number | Promise<200>]
  // is not assignable to type
  // [number, number]
  return result;
}
```

Now `Promise.all` leverages the combination of certain features with `Awaited` to give much better inference results, and the above example works.

For more information, you [can read about this change on GitHub](#).

Template String Types as Discriminants

TypeScript 4.5 now can narrow values that have template string types, and also recognizes template string types as discriminants.

As an example, the following used to fail, but now successfully type-checks in TypeScript 4.5.

```
export interface Success {
  type: `${string}Success`;
  body: string;
}

export interface Error {
  type: `${string}Error`;
  message: string;
}

export function handler(r: Success | Error) {
  if (r.type === "HttpSuccess") {
    const token = r.body;
    // ^
  }
}
```

For more information, [see the change that enables this feature.](#)

module es2022

Thanks to [Kagami S. Roslight](#), TypeScript now supports a new `module` setting: `es2022`. The main feature in `module es2022` is top-level `await`, meaning you can use `await` outside of `async` functions. This was already supported in `--module esnext` (and now `--module nodenext`), but `es2022` is the first stable target for this feature.

You can [read up more on this change here](#).

Tail-Recursion Elimination on Conditional Types

TypeScript often needs to gracefully fail when it detects possibly infinite recursion, or any type expansions that can take a long time and affect your editor experience. As a result, TypeScript has heuristics to make sure it doesn't go off the rails when trying to pick apart an infinitely-deep type, or working with types that generate a lot of intermediate results.

```
type InfiniteBox<T> = { item: InfiniteBox<T> };
type Unpack<T> = T extends { item: infer U } ? Unpack<U> : T;
// error: Type instantiation is excessively deep and possibly infinite.
type Test = Unpack<InfiniteBox<number>>;
```

The above example is intentionally simple and useless, but there are plenty of types that are actually useful, and unfortunately trigger our heuristics. As an example, the following `TrimLeft` type removes spaces from the beginning of a string-like type. If given a string type that has a space at the beginning, it immediately feeds the remainder of the string back into `TrimLeft`.

```
type TrimLeft<T extends string> =
  T extends ` ${infer Rest}` ? TrimLeft<Rest> : T;
// Test = "hello" | "world"
type Test = TrimLeft<" hello" | " world">;
```

This type can be useful, but if a string has 50 leading spaces, you'll get an error.

```
type TrimLeft<T extends string> =
  T extends ` ${infer Rest}` ? TrimLeft<Rest> : T;
// error: Type instantiation is excessively deep and possibly infinite.
type Test = TrimLeft<" oops">;
```

That's unfortunate, because these kinds of types tend to be extremely useful in modeling operations on strings - for example, parsers for URL routers. To make matters worse, a more useful type typically creates more type instantiations, and in turn has even more limitations on input length.

But there's a saving grace: `TrimLeft` is written in a way that is *tail-recursive* in one branch. When it calls itself again, it immediately returns the result and doesn't do anything with it. Because these types don't need to create any intermediate results, they can be implemented more quickly and in a way that avoids triggering many of type recursion heuristics that are built into TypeScript.

That's why TypeScript 4.5 performs some tail-recursion elimination on conditional types. As long as one branch of a conditional type is simply another conditional type, TypeScript can avoid intermediate instantiations. There are still heuristics to ensure that these types don't go off the rails, but they are much more generous.

Keep in mind, the following type *won't* be optimized, since it uses the result of a conditional type by adding it to a union.

```
type GetChars<S> =
  S extends `${infer Char}${infer Rest}` ? Char | GetChars<Rest> : never;
```

If you would like to make it tail-recursive, you can introduce a helper that takes an "accumulator" type parameter, just like with tail-recursive functions.

```
type GetChars<S> = GetCharsHelper<S, never>;
type GetCharsHelper<S, Acc> =
  S extends `${infer Char}${infer Rest}` ? GetCharsHelper<Rest, Char | Acc> :
  Acc;
```

You can read up more on the implementation [here](#).

Disabling Import Elision

There are some cases where TypeScript can't detect that you're using an import. For example, take the following code:

```
import { Animal } from "./animal.js";
eval("console.log(new Animal().isDangerous())");
```

By default, TypeScript always removes this import because it appears to be unused. In TypeScript 4.5, you can enable a new flag called `preserveValueImports` to prevent TypeScript from stripping out any imported values from your JavaScript outputs. Good reasons to use `eval` are few and far between, but something very similar to this happens in Svelte:

```
<!-- A .svelte File -->
<script>
  import { someFunc } from "./some-module.js";
</script>

<button on:click="{someFunc}">Click me!</button>
```

along with in Vue.js, using its `<script setup>` feature:

```
<!-- A .vue File -->
<script setup>
  import { someFunc } from "./some-module.js";
```

```
</script>
<button @click="someFunc">Click me!</button>
```

These frameworks generate some code based on markup outside of their `<script>` tags, but TypeScript only sees code within the `<script>` tags. That means TypeScript will automatically drop the import of `someFunc`, and the above code won't be runnable! With TypeScript 4.5, you can use `preserveValueImports` to avoid these situations.

Note that this flag has a special requirement when combined with `--isolatedModules`: imported types *must* be marked as type-only because compilers that process single files at a time have no way of knowing whether imports are values that appear unused, or a type that must be removed in order to avoid a runtime crash.

```
// Which of these is a value that should be preserved? tsc knows, but
`ts.transpileModule`,
// ts-loader, esbuild, etc. don't, so `isolatedModules` gives an error.
import { someFunc, BaseType } from "./some-module.js";
//           ^^^^^^^^
// Error: 'BaseType' is a type and must be imported using a type-only import
// when 'preserveValueImports' and 'isolatedModules' are both enabled.
```

That makes another TypeScript 4.5 feature, `type` modifiers on import names, especially important.

For more information, [see the pull request here](#).

`type` Modifiers on Import Names

As mentioned above, `preserveValueImports` and `isolatedModules` have special requirements so that there's no ambiguity for build tools whether it's safe to drop type imports.

```
// Which of these is a value that should be preserved? tsc knows, but
`ts.transpileModule`,
// ts-loader, esbuild, etc. don't, so `isolatedModules` issues an error.
import { someFunc, BaseType } from "./some-module.js";
//           ^^^^^^^^
// Error: 'BaseType' is a type and must be imported using a type-only import
// when 'preserveValueImports' and 'isolatedModules' are both enabled.
```

When these options are combined, we need a way to signal when an import can be legitimately dropped. TypeScript already has something for this with `import type`:

```
import type { BaseType } from "./some-module.js";
import { someFunc } from "./some-module.js";

export class Thing implements BaseType {
  // ...
}
```

This works, but it would be nice to avoid two import statements for the same module. That's part of why TypeScript 4.5 allows a `type` modifier on individual named imports, so that you can mix and match as needed.

```
import { someFunc, type BaseType } from "./some-module.js";

export class Thing implements BaseType {
    someMethod() {
        someFunc();
    }
}
```

In the above example, `BaseType` is always guaranteed to be erased and `someFunc` will be preserved under `preserveValueImports`, leaving us with the following code:

```
import { someFunc } from "./some-module.js";

export class Thing {
    someMethod() {
        someFunc();
    }
}
```

For more information, see [the changes on GitHub](#).

Private Field Presence Checks

TypeScript 4.5 supports an ECMAScript proposal for checking whether an object has a private field on it. You can now write a class with a `#private` field member and see whether another object has the same field by using the `in` operator.

```
class Person {
    #name: string;
    constructor(name: string) {
        this.#name = name;
    }

    equals(other: unknown) {
        return other &&
            typeof other === "object" &&
            #name in other && // <- this is new!
            this.#name === other.#name;
    }
}
```

One interesting aspect of this feature is that the check `#name in other` implies that `other` must have been constructed as a `Person`, since there's no other way that field could be present. This is actually one of the key features of the proposal, and it's why the proposal is named "ergonomic brand checks" - because private fields often act as a "brand" to guard against objects that aren't instances of their class. As such, TypeScript is able to appropriately narrow the type of `other` on each check, until it ends up with the type `Person`.

We'd like to extend a big thanks to our friends at Bloomberg who contributed this pull request: [Ashley Claymore](#), [Titian Cernicova-Dragomir](#), [Kubilay Kahveci](#), and [Rob Palmer](#)!

Import Assertions

TypeScript 4.5 supports an ECMAScript proposal for *import assertions*. This is a syntax used by runtimes to make sure that an import has an expected format.

```
import obj from "./something.json" assert { type: "json" };
```

The contents of these assertions are not checked by TypeScript since they're host-specific, and are simply left alone so that browsers and runtimes can handle them (and possibly error).

```
// TypeScript is fine with this.
// But your browser? Probably not.
import obj from "./something.json" assert {
  type: "fluffy bunny"
};
```

Dynamic `import()` calls can also use import assertions through a second argument.

```
const obj = await import("./something.json", {
  assert: { type: "json" },
});
```

The expected type of that second argument is defined by a new type called `ImportCallOptions`, and currently only accepts an `assert` property.

We'd like to thank [Wenlu Wang](#) for [implementing this feature!](#)

Faster Load Time with `realPathSync.native`

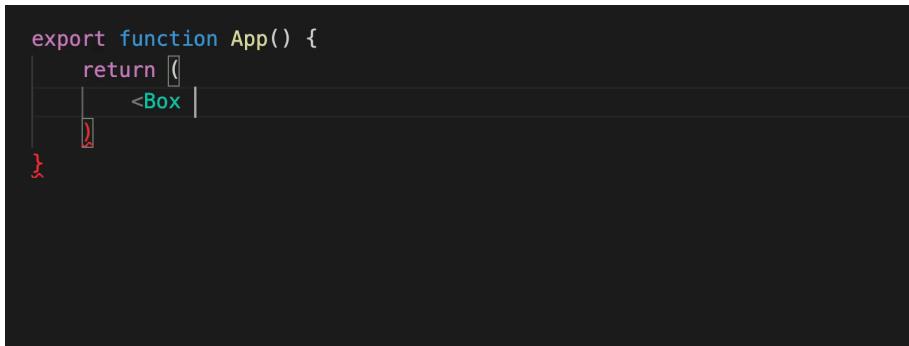
TypeScript now leverages a system-native implementation of the Node.js `realPathSync` function on all operating systems.

Previously this function was only used on Linux, but in TypeScript 4.5 it has been adopted to operating systems that are typically case-insensitive, like Windows and MacOS. On certain codebases, this change sped up project loading by 5-13% (depending on the host operating system).

For more information, see [the original change here](#), along with [the 4.5-specific changes here](#).

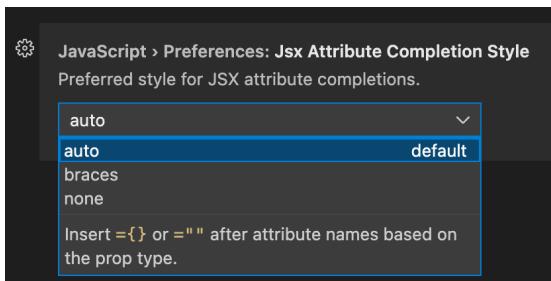
Snippet Completions for JSX Attributes

TypeScript 4.5 brings *snippet completions* for JSX attributes. When writing out an attribute in a JSX tag, TypeScript will already provide suggestions for those attributes; but with snippet completions, they can remove a little bit of extra typing by adding an initializer and putting your cursor in the right place.



Snippet completions for JSX attributes. For a string property, quotes are automatically added. For a numeric properties, braces are added.

TypeScript will typically use the type of an attribute to figure out what kind of initializer to insert, but you can customize this behavior in Visual Studio Code.



Settings in VS Code for JSX attribute completions

Keep in mind, this feature will only work in newer versions of Visual Studio Code, so you might have to use an Insiders build to get this working. For more information, [read up on the original pull request](#)

Better Editor Support for Unresolved Types

In some cases, editors will leverage a lightweight "partial" semantic mode - either while the editor is waiting for the full project to load, or in contexts like [GitHub's web-based editor](#).

In older versions of TypeScript, if the language service couldn't find a type, it would just print `any`.

```
function writeOutput(buf: any): void
function writeOutput(buf: Buffer) {
}
```

Hovering over a signature where `Buffer` isn't found, TypeScript replaces it with `any`.

In the above example, `Buffer` wasn't found, so TypeScript replaced it with `any` in *quick info*. In TypeScript 4.5, TypeScript will try its best to preserve what you wrote.

```
function writeOutput(buf: Buffer): void
function writeOutput(buf: Buffer) {
}
```

Hovering over a signature where `Buffer` isn't found, it continues to use the name `Buffer`.

However, if you hover over `Buffer` itself, you'll get a hint that TypeScript couldn't find `Buffer`.

```
type Buffer = /*unresolved*/ any
function writeOutput(buf: Buffer) {
}
```

TypeScript displays `type Buffer = /* unresolved */ any;`

Altogether, this provides a smoother experience when TypeScript doesn't have the full program available. Keep in mind, you'll always get an error in regular scenarios to tell you when a type isn't found.

For more information, [see the implementation here](#).

Breaking Changes

`lib.d.ts` Changes

TypeScript 4.5 contains changes to its built-in declaration files which may affect your compilation; however, [these changes were fairly minimal](#), and we expect most code will be unaffected.

Inference Changes from `Awaited`

Because `Awaited` is now used in `lib.d.ts` and as a result of `await`, you may see certain generic types change that might cause incompatibilities; however, given many intentional design decisions around `Awaited` to avoid breakage, we expect most code will be unaffected.

Compiler Options Checking at the Root of `tsconfig.json`

It's an easy mistake to accidentally forget about the `compilerOptions` section in a `tsconfig.json`. To help catch this mistake, in TypeScript 4.5, it is an error to add a top-level field which matches any of the available options in `compilerOptions` *without* having also defined `compilerOptions` in that `tsconfig.json`.

[Go to TOC](#)

Allowing Code in Constructors Before `super()`

In JavaScript classes it's mandatory to call `super()` before referring to `this`. TypeScript enforces this as well, though it was a bit too strict in *how* it ensured this. In TypeScript, it was previously an error to contain *any* code at the beginning of a constructor if its containing class had any property initializers.

```
class Base {
  // ...
}

class Derived extends Base {
  someProperty = true;

  constructor() {
    // error!
    // have to call 'super()' first because it needs to initialize 'someProperty'.
    doSomeStuff();
    super();
  }
}
```

This made it cheap to check that `super()` gets called before `this` is referenced, but it ended up rejecting a lot of valid code. TypeScript 4.6 is now much more lenient in that check and permits other code to run before `super()`, all while still ensuring that `super()` occurs at the top-level before any references to `this`.

We'd like to extend our thanks to [Joshua Goldberg](#) for [patiently working with us to land this change!](#)

Control Flow Analysis for Destructured Discriminated Unions

TypeScript is able to narrow types based on what's called a discriminant property. For example, in the following code snippet, TypeScript is able to narrow the type of `action` based on every time we check against the value of `kind`.

```
type Action =
  | { kind: "NumberContents"; payload: number }
  | { kind: "StringContents"; payload: string };

function processAction(action: Action) {
  if (action.kind === "NumberContents") {
    // `action.payload` is a number here.
    let num = action.payload * 2;
    // ...
  } else if (action.kind === "StringContents") {
    // `action.payload` is a string here.
    const str = action.payload.trim();
    // ...
  }
}
```

This lets us work with objects that can hold different data, but a common field tells us *which* data those objects have.

This is very common in TypeScript; however, depending on your preferences, you might have wanted to destructure `kind` and `payload` in the example above. Perhaps something like the following:

```
type Action =
| { kind: "NumberContents"; payload: number }
| { kind: "StringContents"; payload: string };

function processAction(action: Action) {
  const { kind, payload } = action;
  if (kind === "NumberContents") {
    let num = payload * 2;
    // ...
  } else if (kind === "StringContents") {
    const str = payload.trim();
    // ...
  }
}
```

Previously TypeScript would error on these - once `kind` and `payload` were extracted from the same object into variables, they were considered totally independent.

In TypeScript 4.6, this just works!

When destructuring individual properties into a `const` declaration, or when destructuring a parameter into variables that are never assigned to, TypeScript will check for if the destructured type is a discriminated union. If it is, TypeScript can now narrow the types of variables depending on checks of other variables. So in our example, a check on `kind` narrows the type of `payload`.

For more information, [see the pull request that implemented this analysis](#).

Improved Recursion Depth Checks

TypeScript has some interesting challenges due to the fact that it's built on a structural type system that also provides generics.

In a structural type system, object types are compatible based on the members they have.

```
interface Source {
  prop: string;
}

interface Target {
  prop: number;
}

function check(source: Source, target: Target) {
  target = source;
  // error!
  // Type 'Source' is not assignable to type 'Target'.
  //   Types of property 'prop' are incompatible.
  //     Type 'string' is not assignable to type 'number'.
}
```

Notice that whether or not `Source` is compatible with `Target` has to do with whether their *properties* are assignable. In this case, that's just `prop`.

When you introduce generics into this, there are some harder questions to answer. For instance, is a `Source<string>` assignable to a `Target<number>` in the following case?

```
interface Source<T> {
  prop: Source<Source<T>>;
}

interface Target<T> {
  prop: Target<Target<T>>;
}

function check(source: Source<string>, target: Target<number>) {
  target = source;
}
```

In order to answer that, TypeScript needs to check whether the types of `prop` are compatible. That leads to the another question: is a `Source<Source<string>>` assignable to a `Target<Target<number>>`? To answer that, TypeScript checks whether `prop` is compatible for *those* types, and ends up checking whether `Source<Source<Source<string>>>` is assignable to `Target<Target<Target<number>>>`. Keep going for a bit, and you might notice that the type infinitely expands the more you dig in.

TypeScript has a few heuristics here - if a type *appears* to be infinitely expanding after encountering a certain depth check, then it considers that the types *could* be compatible. This is usually enough, but embarrassingly there were some false-negatives that this wouldn't catch.

```
interface Foo<T> {
  prop: T;
}

declare let x: Foo<Foo<Foo<Foo<Foo<string>>>>>;
declare let y: Foo<Foo<Foo<Foo<string>>>>;
```

A human reader can see that `x` and `y` should be incompatible in the above example. While the types are deeply nested, that's just a consequence of how they were declared. The heuristic was meant to capture cases where deeply-nested types were generated through exploring the types, not from when a developer wrote that type out themselves.

TypeScript 4.6 is now able to distinguish these cases, and correctly errors on the last example. Additionally, because the language is no longer concerned with false-positives from explicitly-written types, TypeScript can conclude that a type is infinitely expanding much earlier, and save a bunch of work in checking for type compatibility. As a result, libraries on DefinitelyTyped like `redux-immutable`, `react-lazylog`, and `yup` saw a 50% reduction in check-time.

You may already have this change because it was cherry-picked into TypeScript 4.5.3, but it is a notable feature of TypeScript 4.6 which you can read up more about [here](#).

Indexed Access Inference Improvements

TypeScript now can correctly infer to indexed access types which immediately index into a mapped object type.

```
interface TypeMap {
  number: number;
  string: string;
  boolean: boolean;
}

type UnionRecord<P extends keyof TypeMap> = {
  [K in P]: {
    kind: K;
    v: TypeMap[K];
    f: (p: TypeMap[K]) => void;
  };
}[P];

function processRecord<K extends keyof TypeMap>(record: UnionRecord<K>) {
  record.f(record.v);
}

// This call used to have issues - now works!
processRecord({
  kind: "string",
  v: "hello!",

  // 'val' used to implicitly have the type 'string | number | boolean',
  // but now is correctly inferred to just 'string'.
  f: (val) => {
    console.log(val.toUpperCase());
  },
});
```

This pattern was already supported and allowed TypeScript to understand that the call to `record.f(record.v)` is valid, but previously the call to `processRecord` would give poor inference results for `val`.

TypeScript 4.6 improves this so that no type assertions are necessary within the call to `processRecord`.

For more information, you can [read up on the pull request](#).

Control Flow Analysis for Dependent Parameters

A signature can be declared with a rest parameter whose type is a discriminated union of tuples.

```
function func(...args: ["str", string] | ["num", number]) {
  // ...
}
```

What this says is that the arguments to `func` depends entirely on the first argument. When the first argument is the string `"str"`, then its second argument has to be a `string`. When its first argument is the string `"num"`, its second argument has to be a `number`.

In cases where TypeScript infers the type of a function from a signature like this, TypeScript can now narrow parameters that depend on each other.

```
type Func = (...args: ["a", number] | ["b", string]) => void;

const f1: Func = (kind, payload) => {
  if (kind === "a") {
    payload.toFixed(); // 'payload' narrowed to 'number'
  }
  if (kind === "b") {
    payload.toUpperCase(); // 'payload' narrowed to 'string'
  }
};

f1("a", 42);
f1("b", "hello");
```

For more information, [see the change on GitHub](#).

--target es2022

TypeScript's `--target` option now supports `es2022`. This means features like class fields now have a stable output target where they can be preserved. It also means that new built-in functionality like the `at()` method on `Array`, `Object.hasOwn`, or the `cause` option on `new Error` can be used either with this new `--target` setting, or with `--lib es2022`.

This functionality was [implemented](#) by [Kagami Sascha Rosylight \(saschanaz\)](#) over several PRs, and we're grateful for that contribution!

Removed Unnecessary Arguments in `react-jsx`

Previously, when compiling code like the following in `--jsx react-jsx`

```
export const el = <div>foo</div>;
```

TypeScript would produce the following JavaScript code:

```
import { jsx as _jsx } from "react/jsx-runtime";
export const el = _jsx("div", { children: "foo" }, void 0);
```

That last `void 0` argument is unnecessary in this emit mode, and removing it can improve bundle sizes.

```
- export const el = _jsx("div", { children: "foo" }, void 0);
+ export const el = _jsx("div", { children: "foo" });
```

Thanks to a [pull request](#) from [Alexander Tarasyuk](#), TypeScript 4.6 now drops the `void 0` argument.

JSDoc Name Suggestions

In JSDoc, you can document parameters using an `@param` tag.

```
/**  
 * @param x The first operand  
 * @param y The second operand  
 */  
function add(x, y) {  
    return x + y;  
}
```

But what happens when these comments fall out of date? What if we rename `x` and `y` to `a` and `b`?

```
/**  
 * @param x {number} The first operand  
 * @param y {number} The second operand  
 */  
function add(a, b) {  
    return a + b;  
}
```

Previously TypeScript would only tell you about this when performing type-checking on JavaScript files - when using either the `checkJS` option, or adding a `// @ts-check` comment to the top of your file.

You can now get similar information for TypeScript files in your editor! TypeScript now provides suggestions for when parameter names don't match between your function and its JSDoc comment.

```
/**  
 * Returns an object from a path string - the opposite of format().  
 *  
 * any  
 *  
 * JSDoc '@param' tag has name 'pathString', but there is no  
 * parameter with that name. ts(8024)  
 *  
 * Quick Fix... (Ctrl+.)  
 * @param pathString path to evaluate.  
 */  
parse(p: string): ParsedPath;  
/**  
 * Returns a path string from an object - the opposite of parse().  
 *  
 * @param pathString path to evaluate.  
 */  
format(pP: FormatInputPathObject): string;
```

Suggestion diagnostics being shown in the editor for parameter names in JSDoc comments that don't match an actual parameter name.

This change was provided courtesy of [Alexander Tarasyuk](#)!

More Syntax and Binding Errors in JavaScript

TypeScript has expanded its set of syntax and binding errors in JavaScript files. You'll see these new errors if you open JavaScript files in an editor like Visual Studio or Visual Studio Code, or if you run JavaScript code through the TypeScript compiler - even if you don't turn on `checkJs` or add a `// @ts-check` comment to the top of your files.

As one example, if you have two declarations of a `const` in the same scope of a JavaScript file, TypeScript will now issue an error on those declarations.

```
const foo = 1234;
// ~~~
// error: Cannot redeclare block-scoped variable 'foo'.
//
// ...
const foo = 5678;
// ~~~
// error: Cannot redeclare block-scoped variable 'foo'.
```

As another example, TypeScript will let you know if a modifier is being incorrectly used.

```
function container() {
    export function foo() {
        // ~~~~~
        // error: Modifiers cannot appear here.
    }
}
```

These errors can be disabled by adding a `// @ts-nocheck` at the top of your file, but we're interested in hearing some early feedback about how it works for your JavaScript workflow. You can easily try it out for Visual Studio Code by installing the [TypeScript and JavaScript Nightly Extension](#), and read up more on the [first](#) and [second](#) pull requests.

TypeScript Trace Analyzer

Occasionally, teams may encounter types that are computationally expensive to create and compare against other types. TypeScript has a `--generateTrace` flag to help identify some of those expensive types, or sometimes help diagnose issues in the TypeScript compiler. While the information generated by `--generateTrace` can be useful (especially with some information added in TypeScript 4.6), it can often be hard to read in existing trace visualizers.

We recently published a tool called [@typescript/analyze-trace](#) to get a more digestible view of this information. While we don't expect everyone to need `analyze-trace`, we think it can come in handy for any team that is running into [build performance issues with TypeScript](#).

For more information, [see the `analyze-trace` tool's repo](#).

Breaking Changes

Object Rests Drop Unspreadable Members from Generic Objects

Object rest expressions now drop members that appear to be unspreadable on generic objects. In the following example...

```
class Thing {
    someProperty = 42;

    someMethod() {
        // ...
    }
}

function foo<T extends Thing>(x: T) {
    let { someProperty, ...rest } = x;

    // Used to work, is now an error!
    // Property 'someMethod' does not exist on type 'Omit<T, "someProperty" | "someMethod">'.
    rest.someMethod();
}
```

the variable `rest` used to have the type `Omit<T, "someProperty">` because TypeScript would strictly analyze which other properties were destructured. This doesn't model how `...rest` would work in a destructuring from a non-generic type because `someMethod` would typically be dropped as well. In TypeScript 4.6, the type of `rest` is `Omit<T, "someProperty" | "someMethod">`.

This can also come up in cases when destructuring from `this`. When destructuring `this` using a `...rest` element, unspreadable and non-public members are now dropped, which is consistent with destructuring instances of a class in other places.

```
class Thing {
    someProperty = 42;

    someMethod() {
        // ...
    }

    someOtherMethod() {
        let { someProperty, ...rest } = this;

        // Used to work, is now an error!
        // Property 'someMethod' does not exist on type 'Omit<T, "someProperty" | "someMethod">'.
        rest.someMethod();
    }
}
```

For more details, [see the corresponding change here](#).

JavaScript Files Always Receive Grammar and Binding Errors

Previously, TypeScript would ignore most grammar errors in JavaScript apart from accidentally using TypeScript syntax in a JavaScript file. TypeScript now shows JavaScript syntax and binding errors in your file, such as using incorrect modifiers, duplicate declarations, and more. These will typically be most apparent in Visual Studio Code or Visual Studio, but can also occur when running JavaScript code through the TypeScript compiler.

You can explicitly turn these errors off by inserting a `// @ts-nocheck` comment at the top of your file.

For more information, see the [first](#) and [second](#) implementing pull requests for these features.

[Go to TOC](#)

ECMAScript Module Support in Node.js

For the last few years, Node.js has been working to support ECMAScript modules (ESM). This has been a very difficult feature, since the Node.js ecosystem is built on a different module system called CommonJS (CJS). Interoperating between the two brings large challenges, with many new features to juggle; however, support for ESM in Node.js was largely implemented in Node.js 12 and later. Around TypeScript 4.5 we rolled out nightly-only support for ESM in Node.js to get some feedback from users and let library authors ready themselves for broader support.

TypeScript 4.7 adds this functionality with two new `module` settings: `node16` and `nodenext`.

```
{
  "compilerOptions": {
    "module": "node16",
  }
}
```

These new modes bring a few high-level features which we'll explore here.

`type` in `package.json` and New Extensions

Node.js supports a new setting in `package.json` called `type`. `"type"` can be set to either `"module"` or `"commonjs"`.

```
{
  "name": "my-package",
  "type": "module",
  "//": "...",
  "dependencies": {
  }
}
```

This setting controls whether `.js` files are interpreted as ES modules or CommonJS modules, and defaults to CommonJS when not set. When a file is considered an ES module, a few different rules come into play compared to CommonJS:

- `import / export` statements can be used.
- Top-level `await` can be used
- Relative import paths need full extensions (we have to write `import "./foo.js"` instead of `import "./foo"`).
- Imports might resolve differently from dependencies in `node_modules`.
- Certain global-like values like `require` and `module` cannot be used directly.
- CommonJS modules get imported under certain special rules.

We'll come back to some of these.

To overlay the way TypeScript works in this system, `.ts` and `.tsx` files now work the same way. When TypeScript finds a `.ts`, `.tsx`, `.js`, or `.jsx` file, it will walk up looking for a `package.json` to see whether that file is an ES module, and use that to determine:

- how to find other modules which that file imports
- and how to transform that file if producing outputs

When a `.ts` file is compiled as an ES module, ECMAScript `import / export` statements are left alone in the `.js` output; when it's compiled as a CommonJS module, it will produce the same output you get today under `--module commonjs`.

This also means paths resolve differently between `.ts` files that are ES modules and ones that are CJS modules. For example, let's say you have the following code today:

```
// ./foo.ts
export function helper() {
  // ...
}

// ./bar.ts
import { helper } from "./foo"; // only works in CJS
helper();
```

This code works in CommonJS modules, but will fail in ES modules because relative import paths need to use extensions. As a result, it will have to be rewritten to use the extension of the *output* of `foo.ts` - so `bar.ts` will instead have to import from `./foo.js`.

```
// ./bar.ts
import { helper } from "./foo.js"; // works in ESM & CJS
helper();
```

This might feel a bit cumbersome at first, but TypeScript tooling like auto-imports and path completion will typically just do this for you.

One other thing to mention is the fact that this applies to `.d.ts` files too. When TypeScript finds a `.d.ts` file in package, it is interpreted based on the containing package.

New File Extensions

The `type` field in `package.json` is nice because it allows us to continue using the `.ts` and `.js` file extensions which can be convenient; however, you will occasionally need to write a file that differs from what `type` specifies. You might also just prefer to always be explicit.

Node.js supports two extensions to help with this: `.mjs` and `.cjs`. `.mjs` files are always ES modules, and `.cjs` files are always CommonJS modules, and there's no way to override these.

In turn, TypeScript supports two new source file extensions: `.mts` and `.cts`. When TypeScript emits these to JavaScript files, it will emit them to `.mjs` and `.cjs` respectively.

Furthermore, TypeScript also supports two new declaration file extensions: `.d.mts` and `.d.cts`. When TypeScript generates declaration files for `.mts` and `.cts`, their corresponding extensions will be `.d.mts` and `.d.cts`.

Using these extensions is entirely optional, but will often be useful even if you choose not to use them as part of your primary workflow.

CommonJS Interoperability

Node.js allows ES modules to import CommonJS modules as if they were ES modules with a default export.

```
// ./foo.cts
export function helper() {
    console.log("hello world!");
}

// ./bar.mts
import foo from "./foo.cjs";

// prints "hello world!"
foo.helper();
```

In some cases, Node.js also synthesizes named exports from CommonJS modules, which can be more convenient. In these cases, ES modules can use a "namespace-style" import (i.e. `import * as foo from "..."`), or named imports (i.e. `import { helper } from "..."`).

```
// ./foo.cts
export function helper() {
    console.log("hello world!");
}

// ./bar.mts
import { helper } from "./foo.cjs";

// prints "hello world!"
helper();
```

There isn't always a way for TypeScript to know whether these named imports will be synthesized, but TypeScript will err on being permissive and use some heuristics when importing from a file that is definitely a CommonJS module.

One TypeScript-specific note about interop is the following syntax:

```
import foo = require("foo");
```

In a CommonJS module, this just boils down to a `require()` call, and in an ES module, this imports `createRequire` to achieve the same thing. This will make code less portable on runtimes like the browser (which don't support `require()`), but will often be useful for interoperability. In turn, you can write the above example using this syntax as follows:

```
// ./foo.cts
export function helper() {
    console.log("hello world!");
```

```
{
  // ./bar.mts
  import foo = require("./foo.cjs");
  foo.helper()
```

Finally, it's worth noting that the only way to import ESM files from a CJS module is using dynamic `import()` calls. This can present challenges, but is the behavior in Node.js today.

You can [read more about ESM/CommonJS interop in Node.js here](#).

`package.json` Exports, Imports, and Self-Referencing

Node.js supports a new field for defining entry points in `package.json` called "exports". This field is a more powerful alternative to defining `"main"` in `package.json`, and can control what parts of your package are exposed to consumers.

Here's an `package.json` that supports separate entry-points for CommonJS and ESM:

```
// package.json
{
  "name": "my-package",
  "type": "module",
  "exports": {
    ".": {
      // Entry-point for `import "my-package"` in ESM
      "import": "./esm/index.js",

      // Entry-point for `require("my-package")` in CJS
      "require": "./commonjs/index.cjs",
    },
  },
  // CJS fall-back for older versions of Node.js
  "main": "./commonjs/index.cjs",
}
```

There's a lot to this feature, [which you can read more about on the Node.js documentation](#). Here we'll try to focus on how TypeScript supports it.

With TypeScript's original Node support, it would look for a `"main"` field, and then look for declaration files that corresponded to that entry. For example, if `"main"` pointed to `./lib/index.js`, TypeScript would look for a file called `./lib/index.d.ts`. A package author could override this by specifying a separate field called `"types"` (e.g. `"types": "./types/index.d.ts"`).

The new support works similarly with [import conditions](#). By default, TypeScript overlays the same rules with import conditions - if you write an `import` from an ES module, it will look up the `import` field, and from a CommonJS module, it will look at the `require` field. If it finds them, it will look for a corresponding declaration file. If you need to point to a different location for your type declarations, you can add a `"types"` import condition.

```
// package.json
{
  "name": "my-package",
  "type": "module",
  "exports": {
    ".": {
      // Entry-point for `import "my-package"` in ESM
      "import": {
        // Where TypeScript will look.
        "types": "./types/esm/index.d.ts",
        // Where Node.js will look.
        "default": "./esm/index.js"
      },
      // Entry-point for `require("my-package")` in CJS
      "require": {
        // Where TypeScript will look.
        "types": "./types/commonjs/index.d.cts",
        // Where Node.js will look.
        "default": "./commonjs/index.cjs"
      },
    },
    // Fall-back for older versions of TypeScript
    "types": "./types/index.d.ts",
    // CJS fall-back for older versions of Node.js
    "main": "./commonjs/index.cjs"
  }
}
```

Note that the `"types"` condition should always come first in `"exports"`.

TypeScript also supports the `"imports"` field of `package.json` in a similar manner by looking for declaration files alongside corresponding files, and supports [packages self-referencing themselves](#). These features are generally not as involved to set up, but are supported.

Your Feedback Wanted!

As we continue working on TypeScript 4.7, we expect to see more documentation and polish go into this functionality. Supporting these new features has been an ambitious under-taking, and that's why we're looking for early feedback on it! Please try it out and let us know how it works for you.

For more information, [you can see the implementing PR here](#).

Control over Module Detection

One issue with the introduction of modules to JavaScript was the ambiguity between existing "script" code and the new module code. JavaScript code in a module runs slightly differently, and has different scoping rules, so tools have to make decisions as to how each file runs. For example, Node.js requires module entry-points to be written in a `.mjs`, or have a nearby `package.json` with `"type": "module"`. TypeScript treats a file as a module whenever it finds any `import` or `export` statement in a file, but otherwise, will assume a `.ts` or `.js` file is a script file acting on the global scope.

This doesn't quite match up with the behavior of Node.js where the `package.json` can change the format of a file, or the `--jsx` setting `react-jsx`, where any JSX file contains an implicit import to a JSX factory. It also doesn't match modern expectations where most new TypeScript code is written with modules in mind.

That's why TypeScript 4.7 introduces a new option called `moduleDetection`. `moduleDetection` can take on 3 values: `"auto"` (the default), `"legacy"` (the same behavior as 4.6 and prior), and `"force"`.

Under the mode `"auto"`, TypeScript will not only look for `import` and `export` statements, but it will also check whether

- the `"type"` field in `package.json` is set to `"module"` when running under `--module nodenext` / `--module node16`, and
- check whether the current file is a JSX file when running under `--jsx react-jsx`

In cases where you want every file to be treated as a module, the `"force"` setting ensures that every non-declaration file is treated as a module. This will be true regardless of how `module`, `moduleResolution`, and `jsx` are configured.

Meanwhile, the `"legacy"` option simply goes back to the old behavior of only seeking out `import` and `export` statements to determine whether a file is a module.

You can [read up more about this change on the pull request](#).

Control-Flow Analysis for Bracketed Element Access

TypeScript 4.7 now narrows the types of element accesses when the indexed keys are literal types and unique symbols. For example, take the following code:

```
const key = Symbol();
const numberOrString = Math.random() < 0.5 ? 42 : "hello";
const obj = {
  [key]: numberOrString,
};
if (typeof obj[key] === "string") {
  let str = obj[key].toUpperCase();
}
```

Previously, TypeScript would not consider any type guards on `obj[key]`, and would have no idea that `obj[key]` was really a `string`. Instead, it would think that `obj[key]` was still a `string | number` and accessing `toUpperCase()` would trigger an error.

TypeScript 4.7 now knows that `obj[key]` is a string.

This also means that under `--strictPropertyInitialization`, TypeScript can correctly check that computed properties are initialized by the end of a constructor body.

```
// 'key' has type 'unique symbol'
const key = Symbol();

class C {
  [key]: string;

  constructor(str: string) {
    // oops, forgot to set 'this[key]'
  }

  screamString() {
    return this[key].toUpperCase();
  }
}
```

Under TypeScript 4.7, `--strictPropertyInitialization` reports an error telling us that the `[key]` property wasn't definitely assigned by the end of the constructor.

We'd like to extend our gratitude to [Oleksandr Tarasiuk](#) who provided [this change!](#)

Improved Function Inference in Objects and Methods

TypeScript 4.7 can now perform more granular inferences from functions within objects and arrays. This allows the types of these functions to consistently flow in a left-to-right manner just like for plain arguments.

```
declare function f<T>(arg: {
  produce: (n: string) => T,
  consume: (x: T) => void
}): void;

// Works
f{}{
  produce: () => "hello",
  consume: x => x.toLowerCase()
};

// Works
f{}{
  produce: (n: string) => n,
  consume: x => x.toLowerCase(),
};

// Was an error, now works.
f{}{
  produce: n => n,
  consume: x => x.toLowerCase(),
};

// Was an error, now works.
f{}{
  produce: function () { return "hello"; },
  consume: x => x.toLowerCase(),
};

// Was an error, now works.
f{}{
  produce() { return "hello" },
  consume: x => x.toLowerCase(),
};
```

Inference failed in some of these examples because knowing the type of their `produce` functions would indirectly request the type of `arg` before finding a good type for `T`. TypeScript now gathers functions that could contribute to the inferred type of `T` and infers from them lazily.

For more information, you can [take a look at the specific modifications to our inference process](#).

Instantiation Expressions

Occasionally functions can be a bit more general than we want. For example, let's say we had a `makeBox` function.

```
interface Box<T> {
  value: T;
}

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

Maybe we want to create a more specialized set of functions for making `Box`s of `Wrench`s and `Hammer`s. To do that today, we'd have to wrap `makeBox` in other functions, or use an explicit type for an alias of `makeBox`.

```
function makeHammerBox(hammer: Hammer) {
  return makeBox(hammer);
}

// or...

const makeWrenchBox: (wrench: Wrench) => Box<Wrench> = makeBox;
```

These work, but wrapping a call to `makeBox` is a bit wasteful, and writing the full signature of `makeWrenchBox` could get unwieldy. Ideally, we would be able to say that we just want to alias `makeBox` while replacing all of the generics in its signature.

TypeScript 4.7 allows exactly that! We can now take functions and constructors and feed them type arguments directly.

```
const makeHammerBox = makeBox<Hammer>;
const makeWrenchBox = makeBox<Wrench>;
```

So with this, we can specialize `makeBox` to accept more specific types and reject anything else.

```
const makeStringBox = makeBox<string>;

// TypeScript correctly rejects this.
makeStringBox(42);
```

This logic also works for constructor functions such as `Array`, `Map`, and `Set`.

```
// Has type `new () => Map<string, Error>`  
const ErrorMap = Map<string, Error>;  
  
// Has type `// Map<string, Error>`  
const errorMap = new ErrorMap();
```

When a function or constructor is given type arguments, it will produce a new type that keeps all signatures with compatible type parameter lists, and replaces the corresponding type parameters with the given type arguments. Any other signatures are dropped, as TypeScript will assume that they aren't meant to be used.

For more information on this feature, [check out the pull request](#).

extends Constraints on infer Type Variables

Conditional types are a bit of a power-user feature. They allow us to match and infer against the shape of types, and make decisions based on them. For example, we can write a conditional type that returns the first element of a tuple type if it's a `string`-like type.

```
type FirstIfString<T> =  
  T extends [infer S, ...unknown[]]  
    ? S extends string ? S : never  
    : never;  
  
// string  
type A = FirstIfString<[string, number, number]>;  
  
// "hello"  
type B = FirstIfString<["hello", number, number]>;  
  
// "hello" | "world"  
type C = FirstIfString<["hello" | "world", boolean]>;  
  
// never  
type D = FirstIfString<[boolean, number, string]>;
```

`FirstIfString` matches against any tuple with at least one element and grabs the type of the first element as `S`. Then it checks if `S` is compatible with `string` and returns that type if it is.

Note that we had to use two conditional types to write this. We could have written `FirstIfString` as follows:

```
type FirstIfString<T> =  
  T extends [string, ...unknown[]]  
    // Grab the first type out of `T`  
    ? T[0]  
    : never;
```

This works, but it's slightly more "manual" and less declarative. Instead of just pattern-matching on the type and giving the first element a name, we have to fetch out the `0`th element of `T` with `T[0]`. If we were dealing with types more complex than tuples, this could get a lot trickier, so `infer` can simplify things.

Using nested conditionals to infer a type and then match against that inferred type is pretty common. To avoid that second level of nesting, TypeScript 4.7 now allows you to place a constraint on any `infer` type.

```
type FirstIfString<T> =
  T extends [infer S extends string, ...unknown[]]
    ? S
    : never;
```

This way, when TypeScript matches against `S`, it also ensures that `S` has to be a `string`. If `S` isn't a `string`, it takes the false path, which in these cases is `never`.

For more details, you can [read up on the change on GitHub](#).

Optional Variance Annotations for Type Parameters

Let's take the following types.

```
interface Animal {
  animalStuff: any;
}

interface Dog extends Animal {
  dogStuff: any;
}

// ...

type Getter<T> = () => T;
type Setter<T> = (value: T) => void;
```

Imagine we had two different instances of `Getter`s. Figuring out whether any two different `Getter`s are substitutable for one another depends entirely on `T`. In the case of whether an assignment of `Getter<Dog> → Getter<Animal>` is valid, we have to check whether `Dog → Animal` is valid. Because each type for `T` just gets related in the same "direction", we say that the `Getter` type is *covariant* on `T`. On the other hand, checking whether `Setter<Dog> → Setter<Animal>` is valid involves checking whether `Animal → Dog` is valid. That "flip" in direction is kind of like how in math, checking whether $-x < -y$ is the same as checking whether $y < x$. When we have to flip directions like this to compare `T`, we say that `Setter` is *contravariant* on `T`.

With TypeScript 4.7, we're now able to *explicitly* specify variance on type parameters.

So now, if we want to make it explicit that `Getter` is covariant on `T`, we can now give it an `out` modifier.

```
type Getter<out T> = () => T;
```

And similarly, if we also want to make it explicit that `Setter` is contravariant on `T`, we can give it an `in` modifier.

```
type Setter<in T> = (value: T) => void;
```

`out` and `in` are used here because a type parameter's variance depends on whether it's used in an *output* or an *input*. Instead of thinking about variance, you can just think about if `T` is used in output and input positions.

There are also cases for using both `in` and `out`.

```
interface State<in out T> {
  get: () => T;
  set: (value: T) => void;
}
```

When a `T` is used in both an output and input position, it becomes *invariant*. Two different `State<T>`s can't be interchanged unless their `T`s are the same. In other words, `State<Dog>` and `State<Animal>` aren't substitutable for the other.

Now technically speaking, in a purely structural type system, type parameters and their variance don't really matter - you can just plug in types in place of each type parameter and check whether each matching member is structurally compatible. So if TypeScript uses a structural type system, why are we interested in the variance of type parameters? And why might we ever want to annotate them?

One reason is that it can be a useful for a reader to explicitly see how a type parameter is used at a glance. For much more complex types, it can be difficult to tell whether a type is meant to be read, written, or both. TypeScript will also help us out if we forget to mention how that type parameter is used. As an example, if we forgot to specify both `in` and `out` on `State`, we'd get an error.

```
interface State<out T> {
  // ~~~~~
  // error!
  // Type 'State<sub-T>' is not assignable to type 'State<super-T>' as implied
  by variance annotation.
  //   Types of property 'set' are incompatible.
  //     Type '(value: sub-T) => void' is not assignable to type '(value: super-
  T) => void'.
  //     Types of parameters 'value' and 'value' are incompatible.
  //       Type 'super-T' is not assignable to type 'sub-T'.
  get: () => T;
  set: (value: T) => void;
}
```

Another reason is precision and speed! TypeScript already tries to infer the variance of type parameters as an optimization. By doing this, it can type-check larger structural types in a reasonable amount of time. Calculating variance ahead of time allows the type-checker to skip deeper comparisons and just compare type arguments which can be *much* faster than comparing the full structure of a type over and over again. But often there are cases where this calculation is still fairly expensive, and the calculation may find circularities that can't be accurately resolved, meaning there's no clear answer for the variance of a type.

```
type Foo<T> = {
  x: T;
  f: Bar<T>;
}

type Bar<U> = (x: Baz<U[]>) => void;
```

```

type Baz<V> = {
  value: Foo<V[]>;
}

declare let foo1: Foo<unknown>;
declare let foo2: Foo<string>;

foo1 = foo2; // Should be an error but isn't ✗
foo2 = foo1; // Error - correct ✓

```

Providing an explicit annotation can speed up type-checking at these circularities and provide better accuracy. For instance, marking `T` as invariant in the above example can help stop the problematic assignment.

```

- type Foo<T> = {
+ type Foo<in out T> = {
  x: T;
  f: Bar<T>;
}

```

We don't necessarily recommend annotating every type parameter with its variance; For example, it's possible (but not recommended) to make variance a little stricter than is necessary, so TypeScript won't stop you from marking something as invariant if it's really just covariant, contravariant, or even independent. So if you do choose to add explicit variance markers, we would encourage thoughtful and precise use of them.

But if you're working with deeply recursive types, especially if you're a library author, you may be interested in using these annotations to the benefit of your users. Those annotations can provide wins in both accuracy and type-checking speed, which can even affect their code editing experience. Determining when variance calculation is a bottleneck on type-checking time can be done experimentally, and determined using tooling like our `analyze-trace` utility.

For more details on this feature, you can [read up on the pull request](#).

Resolution Customization with `moduleSuffixes`

TypeScript 4.7 now supports a `moduleSuffixes` option to customize how module specifiers are looked up.

```
{
  "compilerOptions": {
    "moduleSuffixes": [".ios", ".native", ""]
  }
}
```

Given the above configuration, an import like the following...

```
import * as foo from "./foo";
```

will try to look at the relative files `./foo.ios.ts`, `./foo.native.ts`, and finally `./foo.ts`.

Note that the empty string `""` in `moduleSuffixes` is necessary for TypeScript to also look-up `./foo.ts`. In a sense, the default value for `moduleSuffixes` is `[("")]`.

This feature can be useful for React Native projects where each target platform can use a separate `tsconfig.json` with differing `moduleSuffixes`.

The `moduleSuffixes` option was contributed thanks to Adam Foxman!

resolution-mode

With Node's ECMAScript resolution, the mode of the containing file and the syntax you use determines how imports are resolved; however it would be useful to reference the types of a CommonJS module from an ECMAScript module, or vice-versa.

TypeScript now allows `/// <reference types="..." />` directives.

```
/// <reference types="pkg" resolution-mode="require" />
// or
/// <reference types="pkg" resolution-mode="import" />
```

Additionally, in nightly versions of TypeScript, `import type` can specify an import assertion to achieve something similar.

```
// Resolve `pkg` as if we were importing with a `require()`
import type { TypeFromRequire } from "pkg" assert {
  "resolution-mode": "require"
};

// Resolve `pkg` as if we were importing with an `import`
import type { TypeFromImport } from "pkg" assert {
  "resolution-mode": "import"
};

export interface MergedType extends TypeFromRequire, TypeFromImport {}
```

These import assertions can also be used on `import()` types.

```
export type TypeFromRequire =
  import("pkg", { assert: { "resolution-mode": "require" } }).TypeFromRequire;
export type TypeFromImport =
  import("pkg", { assert: { "resolution-mode": "import" } }).TypeFromImport;
export interface MergedType extends TypeFromRequire, TypeFromImport {}
```

The `import type` and `import()` syntaxes only support `resolution-mode` in [nightly builds of TypeScript](#). You'll likely get an error like

`Resolution mode assertions are unstable. Use nightly TypeScript to silence this error. Try updating with 'npm install -D typescript@next'.`

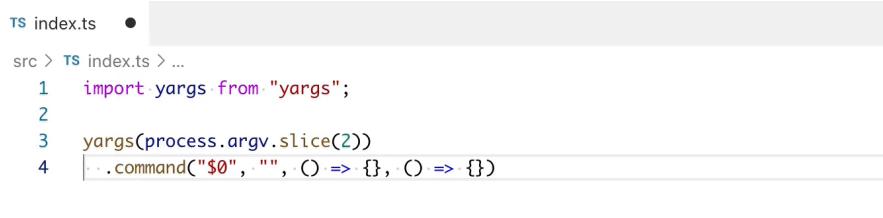
If you do find yourself using this feature in nightly versions of TypeScript, [consider providing feedback on this issue](#).

You can see the respective changes [for reference directives](#) and [for type import assertions](#).

Go to Source Definition

TypeScript 4.7 contains support for a new experimental editor command called *Go To Source Definition*. It's similar to *Go To Definition*, but it never returns results inside declaration files. Instead, it tries to find corresponding *implementation* files (like `.js` or `.ts` files), and find definitions there — even if those files are normally shadowed by `.d.ts` files.

This comes in handy most often when you need to peek at the implementation of a function you're importing from a library instead of its type declaration in a `.d.ts` file.



The screenshot shows a code editor window with a dark theme. The title bar says "index.ts". The code editor displays the following TypeScript code:

```
src > index.ts > ...
1 import {yargs} from "yargs";
2
3 yargs(process.argv.slice(2))
4 .command("$0", "", () => {}, () => {})
```

A cursor is positioned on the line ".command("\$0", "", () => {}, () => {})". A tooltip or status bar at the bottom of the editor window indicates "Go to Source Definition".

The "Go to Source Definition" command on a use of the `yargs` package jumps the editor to an `index.cjs` file in `yargs`.

You can try this new command in the latest versions of Visual Studio Code. Note, though, that this functionality is still in preview, and there are some known limitations. In some cases TypeScript uses heuristics to guess which `.js` file corresponds to the given result of a definition, so these results might be inaccurate. Visual Studio Code also doesn't yet indicate whether a result was a guess, but it's something we're collaborating on.

You can leave feedback about the feature, read about known limitations, or learn more at [our dedicated feedback issue](#).

Group-Aware Organize Imports

TypeScript has an *Organize Imports* editor feature for both JavaScript and TypeScript. Unfortunately, it could be a bit of a blunt instrument, and would often naively sort your import statements.

For instance, if you ran Organize Imports on the following file...

```
// local code
import * as bbb from "./bbb";
import * as ccc from "./ccc";
import * as aaa from "./aaa";

// built-ins
import * as path from "path";
import * as child_process from "child_process"
import * as fs from "fs";

// some code...
```

You would get something like the following

```
// local code
import * as child_process from "child_process";
import * as fs from "fs";
// built-ins
import * as path from "path";
import * as aaa from "./aaa";
import * as bbb from "./bbb";
import * as ccc from "./ccc";

// some code...
```

This is... not ideal. Sure, our imports are sorted by their paths, and our comments and newlines are preserved, but not in a way we expected. Much of the time, if we have our imports grouped in a specific way, then we want to keep them that way.

TypeScript 4.7 performs Organize Imports in a group-aware manner. Running it on the above code looks a little bit more like what you'd expect:

```
// local code
import * as aaa from "./aaa";
import * as bbb from "./bbb";
import * as ccc from "./ccc";

// built-ins
import * as child_process from "child_process";
import * as fs from "fs";
import * as path from "path";

// some code...
```

We'd like to extend our thanks to [Minh Quy](#) who provided [this feature](#).

Object Method Snippet Completions

TypeScript now provides snippet completions for object literal methods. When completing members in an object, TypeScript will provide a typical completion entry for just the name of a method, along with a separate completion entry for the full method definition!

```
interface Options {
  processChange?(key: string, oldState: string, newState: string): void;
}

function getWatchedMap(m: Map<string, string>, options: Options) {
  // ...
}

getWatchedMap(myMap, [
  // ...
]);
```

Completion a full method signature from an object

For more details, [see the implementing pull request](#).

Breaking Changes

`lib.d.ts` Updates

While TypeScript strives to avoid major breaks, even small changes in the built-in libraries can cause issues. We don't expect major breaks as a result of DOM and `lib.d.ts` updates, but there may be some small ones.

Stricter Spread Checks in JSX

When writing a `...spread` in JSX, TypeScript now enforces stricter checks that the given type is actually an object. As a result, values with the types `unknown` and `never` (and more rarely, just bare `null` and `undefined`) can no longer be spread into JSX elements.

So for the following example:

```
import * as React from "react";

interface Props {
  stuff?: string;
}
```

```
function MyComponent(props: unknown) {
    return <div {...props} />;
}
```

you'll now receive an error like the following:

Spread types may only be created from object types.

This makes this behavior more consistent with spreads in object literals.

For more details, see the [change on GitHub](#).

Stricter Checks with Template String Expressions

When a `symbol` value is used in a template string, it will trigger a runtime error in JavaScript.

```
let str = `hello ${Symbol()}`;
// TypeError: Cannot convert a Symbol value to a string
```

As a result, TypeScript will issue an error as well; however, TypeScript now also checks if a generic value that is constrained to a symbol in some way is used in a template string.

```
function logKey<S extends string | symbol>(key: S): S {
    // Now an error.
    console.log(` ${key} is the key`);
    return key;
}

function get<T, K extends keyof T>(obj: T, key: K) {
    // Now an error.
    console.log(`Grabbing property '${key}'`);
    return obj[key];
}
```

TypeScript will now issue the following error:

Implicit conversion of a 'symbol' to a 'string' will fail at runtime. Consider wrapping this expression in 'String(...)'.

In some cases, you can get around this by wrapping the expression in a call to `String`, just like the error message suggests.

```
function logKey<S extends string | symbol>(key: S): S {
    // Now an error.
    console.log(` ${String(key)} is the key`);
    return key;
}
```

In others, this error is too pedantic, and you might not ever care to even allow `symbol` keys when using `keyof`. In such cases, you can switch to `string & keyof ...`:

```
function get<T, K extends string & keyof T>(obj: T, key: K) {
  // Now an error.
  console.log(`Grabbing property '${key}'.`);
  return obj[key];
}
```

For more information, you can [see the implementing pull request](#).

readFile Method is No Longer Optional on LanguageServiceHost

If you're creating `LanguageService` instances, then provided `LanguageServiceHost`s will need to provide a `readFile` method. This change was necessary to support the new `moduleDetection` compiler option.

You can [read more on the change here](#).

readonly Tuples Have a readonly length Property

A `readonly` tuple will now treat its `length` property as `readonly`. This was almost never witnessable for fixed-length tuples, but was an oversight which could be observed for tuples with trailing optional and rest element types.

As a result, the following code will now fail:

```
function overwriteLength(tuple: readonly [string, string, string]) {
  // Now errors.
  tuple.length = 7;
}
```

You can [read more on this change here](#).

[Go to TOC](#)

Improved Intersection Reduction, Union Compatibility, and Narrowing

TypeScript 4.8 brings a series of correctness and consistency improvements under `--strictNullChecks`. These changes affect how intersection and union types work, and are leveraged in how TypeScript narrows types.

For example, `unknown` is close in spirit to the union type `{}` | `null` | `undefined` because it accepts `null`, `undefined`, and any other type. TypeScript now recognizes this, and allows assignments from `unknown` to `{}` | `null` | `undefined`.

```
function f(x: unknown, y: {} | null | undefined) {
    x = y; // always worked
    y = x; // used to error, now works
}
```

Another change is that `{}` intersected with any other object type simplifies right down to that object type. That meant that we were able to rewrite `NonNullable` to just use an intersection with `{}`, because `{}` & `null` and `{}` & `undefined` just get tossed away.

```
- type NonNullable<T> = T extends null | undefined ? never : T;
+ type NonNullable<T> = T & {};
```

This is an improvement because intersection types like this can be reduced and assigned to, while conditional types currently cannot. So `NonNullable<NonNullable<T>>` now simplifies at least to `NonNullable<T>`, whereas it didn't before.

```
function foo<T>(x: NonNullable<T>, y: NonNullable<NonNullable<T>>) {
    x = y; // always worked
    y = x; // used to error, now works
}
```

These changes also allowed us to bring in sensible improvements in control flow analysis and type narrowing. For example, `unknown` is now narrowed just like `{}` | `null` | `undefined` in truthy branches.

```
function narrowUnknownishUnion(x: {} | null | undefined) {
    if (x) {
        x; // {}
    }
    else {
        x; // {} | null | undefined
    }
}

function narrowUnknown(x: unknown) {
    if (x) {
        x; // used to be 'unknown', now '{}'
    }
    else {
        x; // unknown
    }
}
```

Generic values also get narrowed similarly. When checking that a value isn't `null` or `undefined`, TypeScript now just intersects it with `{}` - which again, is the same as saying it's `NonNullable`. Putting many of the changes here together, we can now define the following function without any type assertions.

```
function throwIfNullable<T>(value: T): NonNullable<T> {
  if (value === undefined || value === null) {
    throw Error("Nullable value!");
  }

  // Used to fail because 'T' was not assignable to 'NonNullable<T>'.
  // Now narrows to 'T & {}' and succeeds because that's just 'NonNullable<T>'.
  return value;
}
```

`value` now gets narrowed to `T & {}`, and is now identical with `NonNullable<T>` - so the body of the function just works with no TypeScript-specific syntax.

On their own, these changes may appear small - but they represent fixes for many many paper cuts that have been reported over several years.

For more specifics on these improvements, you can [read more here](#).

Improved Inference for `infer` Types in Template String Types

TypeScript recently introduced a way to add `extends` constraints to `infer` type variables in conditional types.

```
// Grabs the first element of a tuple if it's assignable to 'number',
// and returns 'never' if it can't find one.
type TryGetNumberIfFirst<T> =
  T extends [infer U extends number, ...unknown[]} ? U : never;
```

If these `infer` types appear in a template string type and are constrained to a primitive type, TypeScript will now try to parse out a literal type.

```
// SomeNum used to be 'number'; now it's '100'.
type SomeNum = "100" extends `${infer U extends number}` ? U : never;

// SomeBigInt used to be 'bigint'; now it's '100n'.
type SomeBigInt = "100" extends `${infer U extends bigint}` ? U : never;

// SomeBool used to be 'boolean'; now it's 'true'.
type SomeBool = "true" extends `${infer U extends boolean}` ? U : never;
```

This can now better convey what a library will do at runtime, and give more precise types.

One note on this is that when TypeScript parses these literal types out it will greedily try to parse out as much of what looks like of the appropriate primitive type; however it then checks to see if the print-back of that primitive matches up with the string contents. In other words, TypeScript checks whether the going from the string, to the primitive, and back matches. If it doesn't see that the string can be "round-tripped", then it will fall back to the base primitive type.

```
// JustNumber is `number` here because TypeScript parses out `1.0`, but
`String(Number("1.0"))` is `"1"` and doesn't match.
type JustNumber = "1.0" extends `${infer T extends number}` ? T : never;
```

You can [see more about this feature here](#).

--build, --watch, and --incremental Performance Improvements

TypeScript 4.8 introduces several optimizations that should speed up scenarios around `--watch` and `--incremental`, along with project references builds using `--build`. For example, TypeScript is now able to avoid spending time updating timestamps during no-op changes in `--watch` mode, which makes rebuilds faster and avoids messing with other build tools that might be watching for TypeScript's output. Many other optimizations where we're able to reuse information across `--build`, `--watch`, and `--incremental` have been introduced as well.

How big are these improvements? Well, on a fairly large internal codebase, we've seen time reductions on the order of 10%-25% on many simple common operations, with around 40% time reductions in no-change scenarios. We've seen similar results on the TypeScript codebase as well.

You can see [the changes, along with the performance results on GitHub](#).

Errors When Comparing Object and Array Literals

In many languages, operators like `==` perform what's called "value" equality on objects. For example, in Python it's valid to check whether a list is empty by checking whether a value is equal to the empty list using `==`.

```
if people_at_home == []:
    print("here's where I lie, broken inside. </3")
    adopt_animals()
```

This is not the case in JavaScript, where `==` and `===` between objects (and therefore, arrays) check whether both references point to the same value. We believe that similar code in JavaScript is at best an early foot-gun for JavaScript developers, and at worst a bug in production code. That's why TypeScript now disallows code like the following.

```
if (peopleAtHome === []) {
    // ~~~~~
    // This condition will always return 'false' since JavaScript compares objects by
    // reference, not value.
    console.log("here's where I lie, broken inside. </3")
    adoptAnimals();
}
```

We'd like to extend our gratitude to [Jack Works](#) who contributed this check. You can [view the changes involved here](#).

Improved Inference from Binding Patterns

In some cases, TypeScript will pick up a type from a binding pattern to make better inferences.

```
declare function chooseRandomly<T>(x: T, y: T): T;

let [a, b, c] = chooseRandomly([42, true, "hi!"], [0, false, "bye!"]);
//   ^  ^  ^
//   |  |  |
//   |  |  string
//   |  |
//   |  boolean
//   |
// number
```

When `chooseRandomly` needs to figure out a type for `T`, it will primarily look at `[42, true, "hi!"]` and `[0, false, "bye!"]`; but TypeScript needs to figure out whether those two types should be `Array<number | boolean | string>` or the tuple type `[number, boolean, string]`. To do that, it will look for existing candidates as a hint to see whether there are any tuple types. When TypeScript sees the binding pattern `[a, b, c]`, it creates the type `[any, any, any]`, and that type gets picked up as a low-priority candidate for `T` which also gets used as a hint for the types of `[42, true, "hi!"]` and `[0, false, "bye!"]`.

You can see how this was good for `chooseRandomly`, but it fell short in other cases. For example, take the following code

```
declare function f<T>(x?: T): T;

let [x, y, z] = f();
```

The binding pattern `[x, y, z]` hinted that `f` should produce an `[any, any, any]` tuple; but `f` really shouldn't change its type argument based on a binding pattern. It can't suddenly conjure up a new array-like value based on what it's being assigned to, so the binding pattern type has way too much influence on the produced type. On top of that, because the binding pattern type is full of `any`s, we're left with `x`, `y`, and `z` being typed as `any`.

In TypeScript 4.8, these binding patterns are never used as candidates for type arguments. Instead, they're just consulted in case a parameter needs a more specific type like in our `chooseRandomly` example. If you need to revert to the old behavior, you can always provide explicit type arguments.

You can [look at the change on GitHub](#) if you're curious to learn more.

File-Watching Fixes (Especially Across git checkouts)

We've had a long-standing bug where TypeScript has a very hard time with certain file changes in `--watch` mode and editor scenarios. Sometimes the symptoms are stale or inaccurate errors that might show up that require restarting `tsc` or VS Code. Frequently these occur on Unix systems, and you might have seen these after saving a file with vim or swapping branches in git.

This was caused by assumptions of how Node.js handles rename events across file systems. File systems used by Linux and macOS utilize [inodes](#), and [Node.js will attach file watchers to inodes rather than file paths](#). So when Node.js returns a [watcher object](#), it might be watching a path or an inode depending on the platform and file system.

To be a bit more efficient, TypeScript tries to reuse the same watcher objects if it detects a path still exists on disk. This is where things went wrong, because even if a file still exists at that path, a distinct file might have been created, and that file will have a different inode. So TypeScript would end up reusing the watcher object instead of installing a new watcher at the original location, and watch for changes at what might be a totally irrelevant file. So TypeScript 4.8 now handles these cases on inode systems and properly installs a new watcher and fixes this.

We'd like to extend our thanks to [Marc Celani](#) and his team at Airtable who invested lots of time in investigating the issues they were experiencing and pointing out the root cause. You can view [the specific fixes around file-watching here](#).

Find-All-References Performance Improvements

When running find-all-references in your editor, TypeScript is now able to act a little smarter as it aggregates references. This reduced the amount of time TypeScript took to search a widely-used identifier in its own codebase by about 20%.

[You can read up more on the improvement here.](#)

Exclude Specific Files from Auto-Imports

TypeScript 4.8 introduces an editor preference for excluding files from auto-imports. In Visual Studio Code, file names or globs can be added under "Auto Import File Exclude Patterns" in the Settings UI, or in a `.vscode/settings.json` file:

```
{
  // Note that `javascript.preferences.autoImportFileExcludePatterns` can be
  // specified for JavaScript too.
  "typescript.preferences.autoImportFileExcludePatterns": [
    "**/node_modules/@types/node"
  ]
}
```

This can be useful in cases where you can't avoid having certain modules or libraries in your compilation but you rarely want to import from them. These modules might have lots of exports that can pollute the auto-imports list and make it harder to navigate, and this option can help in those situations.

You can [see more specifics about the implementation here](#).

Correctness Fixes and Breaking Changes

Due to the nature of type system changes, there are very few changes that can be made that don't affect *some* code; however, there are a few changes that are more likely to require adapting existing code.

lib.d.ts Updates

While TypeScript strives to avoid major breaks, even small changes in the built-in libraries can cause issues. We don't expect major breaks as a result of DOM and `lib.d.ts` updates, but one notable change is that the `cause` property on `Error`s now has the type `unknown` instead of `Error`.

Unconstrained Generics No Longer Assignable to `{}`

In TypeScript 4.8, for projects with `strictNullChecks` enabled, TypeScript will now correctly issue an error when an unconstrained type parameter is used in a position where `null` or `undefined` are not legal values. That will include any type that expects `{}`, `object`, or an object type with all-optional properties.

A simple example can be seen in the following.

```
// Accepts any non-null non-undefined value
function bar(value: {}) {
  Object.keys(value); // This call throws on null/undefined at runtime.
}

// Unconstrained type parameter T...
function foo<T>(x: T) {
  bar(x); // Used to be allowed, now is an error in 4.8.
  // ~
  // error: Argument of type 'T' is not assignable to parameter of type '{}'.
}
foo(undefined);
```

As demonstrated above, code like this has a potential bug - the values `null` and `undefined` can be indirectly passed through these unconstrained type parameters to code that is not supposed to observe those values.

This behavior will also be visible in type positions. One example would be:

```
interface Foo<T> {
  x: Bar<T>;
}

interface Bar<T extends {}> { }
```

Existing code that didn't want to handle `null` and `undefined` can be fixed by propagating the appropriate constraints through.

```
- function foo<T>(x: T) {
+ function foo<T extends {}>(x: T) {
```

Another work-around would be to check for `null` and `undefined` at runtime.

```
+   if (x !== null && x !== undefined) {
+     bar(x);
+   }
```

And if you know that for some reason, your generic value can't be `null` or `undefined`, you can just use a non-null assertion.

```
function foo<T>(x: T) {
-   bar(x);
+   bar(x!);
}
```

When it comes to types, you'll often either need to propagate constraints, or intersect your types with `{}`.

For more information, you can [see the change that introduced this](#) along with [the specific discussion issue regarding how unconstrained generics now work](#).

Decorators are placed on `modifiers` on TypeScript's Syntax Trees

The current direction of decorators in TC39 means that TypeScript will have to handle a break in terms of placement of decorators. Previously, TypeScript assumed decorators would always be placed prior to all keywords/modifiers. For example

```
@decorator
export class Foo {
  // ...
}
```

Decorators as currently proposed do not support this syntax. Instead, the `export` keyword must precede the decorator.

```
export @decorator class Foo {
  // ...
}
```

Unfortunately, TypeScript's trees are *concrete* rather than *abstract*, and our architecture expects syntax tree node fields to be entirely ordered before or after each other. To support both legacy decorators and decorators as proposed, TypeScript will have to gracefully parse, and intersperse, modifiers and decorators.

To do this, it exposes a new type alias called `ModifierLike` which is a `Modifier` or a `Decorator`.

```
export type ModifierLike = Modifier | Decorator;
```

Decorators are now placed in the same field as `modifiers` which is now a `NodeArray<ModifierLike>` when set, and the entire field is deprecated.

```
- readonly modifiers?: NodeArray<Modifier> | undefined;
+ /**
+ * @deprecated ...
+ * Use `ts.canHaveModifiers()` to test whether a `Node` can have modifiers.
+ * Use `ts.getModifiers()` to get the modifiers of a `Node`.
+ */
+ readonly modifiers?: NodeArray<ModifierLike> | undefined;
```

All existing `decorators` properties have been marked as deprecated and will always be `undefined` if read. The type has also been changed to `undefined` so that existing tools know to handle them correctly.

```
- readonly decorators?: NodeArray<Decorator> | undefined;
+ /**
+ * @deprecated ...
+ * Use `ts.canHaveDecorators()` to test whether a `Node` can have decorators.
+ * Use `ts.getDecorators()` to get the decorators of a `Node`.
+ *
+ */
+ readonly decorators?: undefined;
```

To avoid new deprecation warnings and other issues, TypeScript now exposes four new functions to use in place of the `decorators` and `modifiers` properties. There are individual predicates for testing whether a node has support modifiers and decorators, along with respective accessor functions for grabbing them.

```
function canHaveModifiers(node: Node): node is HasModifiers;
function getModifiers(node: HasModifiers): readonly Modifier[] | undefined;

function canHaveDecorators(node: Node): node is HasDecorators;
function getDecorators(node: HasDecorators): readonly Decorator[] | undefined;
```

As an example of how to access modifiers off of a node, you can write

```
const modifiers = canHaveModifiers(myNode) ? getModifiers(myNode) : undefined;
```

With the note that each call to `getModifiers` and `getDecorators` may allocate a new array.

For more information, see changes around

- the restructuring of our tree nodes
- the deprecations
- exposing the predicate functions

Types Cannot Be Imported/Exported in JavaScript Files

TypeScript previously allowed JavaScript files to import and export entities declared with a type, but no value, in `import` and `export` statements. This behavior was incorrect, because named imports and exports for values that don't exist will cause a runtime error under ECMAScript modules. When a JavaScript file is type-checked under `--checkJs` or through a `// @ts-check` comment, TypeScript will now issue an error.

```
// @ts-check

// Will fail at runtime because 'SomeType' is not a value.
import { someValue, SomeType } from "some-module";

/**
 * @type {SomeType}
 */
export const myValue = someValue;

/**
 * @typedef {string | number} MyType
 */
```

```
// Will fail at runtime because 'MyType' is not a value.
export { MyType as MyExportedType };
```

To reference a type from another module, you can instead directly qualify the import.

```
- import { someValue, SomeType } from "some-module";
+ import { someValue } from "some-module";

/**
- * @type {SomeType}
+ * @type {import("some-module").SomeType}
 */
export const myValue = someValue;
```

To export a type, you can just use a `/** @typedef */` comment in JSDoc. `@typedef` comments already automatically export types from their containing modules.

```
/**
 * @typedef {string | number} MyType
 */

+ /**
+ * @typedef {MyType} MyExportedType
+ */
- export { MyType as MyExportedType };
```

You can [read more about the change here](#).

Binding Patterns Do Not Directly Contribute to Inference Candidates

As mentioned above, binding patterns no longer change the type of inference results in function calls. You can [read more about the original change here](#).

Unused Renames in Binding Patterns are Now Errors in Type Signatures

TypeScript's type annotation syntax often looks like it can be used when destructuring values. For example, take the following function.

```
declare function makePerson({ name: string, age: number }): Person;
```

You might read this signature and think that `makePerson` obviously takes an object with a `name` property with the type `string` and an `age` property with the type `number`; however, JavaScript's destructuring syntax is actually taking precedence here. `makePerson` does say that it's going to take an object with a `name` and an `age` property, but instead of specifying a type for them, it's just saying that it renames `name` and `age` to `string` and `number` respectively.

In a pure type construct, writing code like this is useless, and typically a mistake since developers usually assume they're writing a type annotation.

TypeScript 4.8 makes these an error unless they're referenced later in the signature. The correct way to write the above signature would be as follows:

```
declare function makePerson(options: { name: string, age: number }): Person;  
// or  
declare function makePerson({ name, age }: { name: string, age: number }): Person;
```

This change can catch bugs in declarations, and has been helpful for improving existing code. We'd like to extend our thanks to [GitHub user uhyo](#) for providing this check. [You can read up on the change here.](#)

[Go to TOC](#)

The `satisfies` Operator

TypeScript developers are often faced with a dilemma: we want to ensure that some expression *matches* some type, but also want to keep the *most specific* type of that expression for inference purposes.

For example:

```
// Each property can be a string or an RGB tuple.
const palette = {
  red: [255, 0, 0],
  green: "#00ff00",
  bleu: [0, 0, 255]
// ^^^ sacrebleu - we've made a typo!
};

// We want to be able to use array methods on 'red'...
const redComponent = palette.red.at(0);

// or string methods on 'green'...
const greenNormalized = palette.green.toUpperCase();
```

Notice that we've written `bleu`, whereas we probably should have written `blue`. We could try to catch that `bleu` typo by using a type annotation on `palette`, but we'd lose the information about each property.

```
type Colors = "red" | "green" | "blue";

type RGB = [red: number, green: number, blue: number];

const palette: Record<Colors, string | RGB> = {
  red: [255, 0, 0],
  green: "#00ff00",
  bleu: [0, 0, 255]
// ~~~~ The typo is now correctly detected
};

// But we now have an undesirable error here - 'palette.red' "could" be a string.
const redComponent = palette.red.at(0);
```

The new `satisfies` operator lets us validate that the type of an expression matches some type, without changing the resulting type of that expression. As an example, we could use `satisfies` to validate that all the properties of `palette` are compatible with `string | number[]`:

```
type Colors = "red" | "green" | "blue";

type RGB = [red: number, green: number, blue: number];

const palette = {
  red: [255, 0, 0],
  green: "#00ff00",
  bleu: [0, 0, 255]
// ~~~~ The typo is now caught!
} satisfies Record<Colors, string | RGB>;

// Both of these methods are still accessible!
const redComponent = palette.red.at(0);
const greenNormalized = palette.green.toUpperCase();
```

`satisfies` can be used to catch lots of possible errors. For example, we could ensure that an object has *all* the keys of some type, but no more:

```
type Colors = "red" | "green" | "blue";

// Ensure that we have exactly the keys from 'Colors'.
const favoriteColors = {
  "red": "yes",
  "green": false,
  "blue": "kinda",
  "platypus": false
} satisfies Record<Colors, unknown>;

// All the information about the 'red', 'green', and 'blue' properties are
// retained.
const g: boolean = favoriteColors.green;
```

Maybe we don't care about if the property names match up somehow, but we do care about the types of each property. In that case, we can also ensure that all of an object's property values conform to some type.

```
type RGB = [red: number, green: number, blue: number];

const palette = {
  red: [255, 0, 0],
  green: "#00ff00",
  blue: [0, 0]
} satisfies Record<string, string | RGB>;

// Information about each property is still maintained.
const redComponent = palette.red.at(0);
const greenNormalized = palette.green.toUpperCase();
```

For more examples, you can see the [issue proposing this](#) and the [implementing pull request](#). We'd like to thank [Oleksandr Tarasiuk](#) who implemented and iterated on this feature with us.

Unlisted Property Narrowing with the `in` Operator

As developers, we often need to deal with values that aren't fully known at runtime. In fact, we often don't know if properties exist, whether we're getting a response from a server or reading a configuration file. JavaScript's `in` operator can check whether a property exists on an object.

Previously, TypeScript allowed us to narrow away any types that don't explicitly list a property.

```
interface RGB {
  red: number;
  green: number;
  blue: number;
}

interface HSV {
  hue: number;
  saturation: number;
```

```

    value: number;
}

function setColor(color: RGB | HSV) {
  if ("hue" in color) {
    // 'color' now has the type HSV
  }
  // ...
}

```

Here, the type `RGB` didn't list the `hue` and got narrowed away, and leaving us with the type `HSV`.

But what about examples where no type listed a given property? In those cases, the language didn't help us much. Let's take the following example in JavaScript:

```

function tryGetPackageName(context) {
  const packageJSON = context.packageJSON;
  // Check to see if we have an object.
  if (packageJSON && typeof packageJSON === "object") {
    // Check to see if it has a string name property.
    if ("name" in packageJSON && typeof packageJSON.name === "string") {
      return packageJSON.name;
    }
  }

  return undefined;
}

```

Rewriting this to canonical TypeScript would just be a matter of defining and using a type for `context`; however, picking a safe type like `unknown` for the `packageJSON` property would cause issues in older versions of TypeScript.

```

interface Context {
  packageJSON: unknown;
}

function tryGetPackageName(context: Context) {
  const packageJSON = context.packageJSON;
  // Check to see if we have an object.
  if (packageJSON && typeof packageJSON === "object") {
    // Check to see if it has a string name property.
    if ("name" in packageJSON && typeof packageJSON.name === "string") {
      // ~~~~  

      // error! Property 'name' does not exist on type 'object'.
      return packageJSON.name;
    }
    // ~~~~  

    // error! Property 'name' does not exist on type 'object'.
  }

  return undefined;
}

```

This is because while the type of `packageJSON` was narrowed from `unknown` to `object`, the `in` operator strictly narrowed to types that actually defined the property being checked. As a result, the type of `packageJSON` remained `object`.

TypeScript 4.9 makes the `in` operator a little bit more powerful when narrowing types that *don't* list the property at all. Instead of leaving them as-is, the language will intersect their types with `Record<"property-key-being-checked", unknown>`.

So in our example, `packageJSON` will have its type narrowed from `unknown` to `object` to `object & Record<"name", unknown>`. That allows us to access `packageJSON.name` directly and narrow that independently.

```
interface Context {
    packageJSON: unknown;
}

function tryGetPackageName(context: Context): string | undefined {
    const packageJSON = context.packageJSON;
    // Check to see if we have an object.
    if (packageJSON && typeof packageJSON === "object") {
        // Check to see if it has a string name property.
        if ("name" in packageJSON && typeof packageJSON.name === "string") {
            // Just works!
            return packageJSON.name;
        }
    }
    return undefined;
}
```

TypeScript 4.9 also tightens up a few checks around how `in` is used, ensuring that the left side is assignable to the type `string | number | symbol`, and the right side is assignable to `object`. This helps check that we're using valid property keys, and not accidentally checking primitives.

For more information, [read the implementing pull request](#)

Auto-Accessors in Classes

TypeScript 4.9 supports an upcoming feature in ECMAScript called auto-accessors. Auto-accessors are declared just like properties on classes, except that they're declared with the `accessor` keyword.

```
class Person { accessor name: string; constructor(name: string) { this.name = name; } }
```

Under the covers, these auto-accessors "de-sugar" to a `get` and `set` accessor with an unreachable private property.

```
class Person { #__name: string; get name() { return this.#__name; } set name(value: string) { this.#__name = value; } constructor(name: string) { this.name = name; } }
```

You can [read up more about the auto-accessors pull request on the original PR](<https://github.com/microsoft/TypeScript/pull/49705>).

Checks For Equality on `Nan`

A major gotcha for JavaScript developers is checking against the value `Nan` using the built-in equality operators.

For some background, `Nan` is a special numeric value that stands for "Not a Number". Nothing is ever equal to `Nan` - even `Nan`!

```
console.log(Nan == 0) // false
console.log(Nan === 0) // false

console.log(Nan == Nan) // false
console.log(Nan === Nan) // false
```

But at least symmetrically *everything* is always not-equal to `Nan`.

```
console.log(Nan != 0) // true
console.log(Nan !== 0) // true

console.log(Nan != Nan) // true
console.log(Nan !== Nan) // true
```

This technically isn't a JavaScript-specific problem, since any language that contains IEEE-754 floats has the same behavior; but JavaScript's primary numeric type is a floating point number, and number parsing in JavaScript can often result in `Nan`. In turn, checking against `Nan` ends up being fairly common, and the correct way to do so is to use `[Number.isNaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/isNaN)` - *but* as we mentioned, lots of people accidentally end up checking with `someValue === Nan` instead.

TypeScript now errors on direct comparisons against `Nan`, and will suggest using some variation of `Number.isNaN` instead.

```
function validate(someValue: number) {
  return someValue !== Nan;
  // ~~~~~
  // error: This condition will always return 'true'.
  // Did you mean '!Number.isNaN(someValue)'?
}
```

We believe that this change should strictly help catch beginner errors, similar to how TypeScript currently issues errors on comparisons against object and array literals.

We'd like to extend our thanks to [Oleksandr Tarasiuk](https://github.com/a-tarasyuk) who [contributed this check](https://github.com/microsoft/TypeScript/pull/50626).

File-Watching Now Uses File System Events

In earlier versions, TypeScript leaned heavily on *polling* for watching individual files. Using a polling strategy meant checking the state of a file periodically for updates. On Node.js, `[fs.watchFile](https://nodejs.org/docs/latest-v18.x/api/fs.html#fswatchfilename-options-listener)` is the built-in way to

get a polling file-watcher. While polling tends to be more predictable across platforms and file systems, it means that your CPU has to periodically get interrupted and check for updates to the file, even when nothing's changed. For a few dozen files, this might not be noticeable; but on a bigger project with lots of files - or lots of files in `node_modules` - this can become a resource hog.

Generally speaking, a better approach is to use file system events. Instead of polling, we can announce that we're interested in updates of specific files and provide a callback for when those files *actually do change*. Most modern platforms in use provide facilities and APIs like `CreateIoCompletionPort`, `kqueue`, `epoll`, and `inotify`. Node.js mostly abstracts these away by providing [`fs.watch`](<https://nodejs.org/docs/latest-v18.x/api/fs.html#fswatchfilename-options-listener>). File system events usually work great, but there are [lots of caveats](<https://nodejs.org/docs/latest-v18.x/api/fs.html#caveats>) to using them, and in turn, to using the `fs.watch` API. A watcher needs to be careful to consider [inode watching](<https://nodejs.org/docs/latest-v18.x/api/fs.html#inodes>), [unavailability on certain file systems](<https://nodejs.org/docs/latest-v18.x/api/fs.html#availability>) (e.g. networked file systems), whether recursive file watching is available, whether directory renames trigger events, and even file watcher exhaustion! In other words, it's not quite a free lunch, especially if you're looking for something cross-platform.

As a result, our default was to pick the lowest common denominator: polling. Not always, but most of the time.

Over time, we've provided the means to [choose other file-watching strategies](<https://www.typescriptlang.org/docs/handbook/configuring-watch.html>). This allowed us to get feedback and harden our file-watching implementation against most of these platform-specific gotchas. As TypeScript has needed to scale to larger codebases, and has improved in this area, we felt swapping to file system events as the default would be a worthwhile investment.

In TypeScript 4.9, file watching is powered by file system events by default, only falling back to polling if we fail to set up event-based watchers. For most developers, this should provide a much less resource-intensive experience when running in `--watch` mode, or running with a TypeScript-powered editor like Visual Studio or VS Code.

[The way file-watching works can still be configured](<https://www.typescriptlang.org/docs/handbook/configuring-watch.html>) through environment variables and `watchOptions` - and [some editors like VS Code can support `watchOptions` independently](<https://code.visualstudio.com/docs/getstarted/settings#:~:text=typescript%2etsserver%2ewatchOptions>). Developers using more exotic set-ups where source code resides on a networked file systems (like NFS and SMB) may need to opt back into the older behavior; though if a server has reasonable processing power, it might just be better to enable SSH and run TypeScript remotely so that it has direct local file access. VS Code has plenty of [remote extensions](<https://marketplace.visualstudio.com/search?term=remote&target=VSCode&category=All%20categories&sortBy=Relevance>) to make this easier.

You can [read up more on this change on GitHub](<https://github.com/microsoft/TypeScript/pull/50366>).

"Remove Unused Imports" and "Sort Imports" Commands for Editors

Previously, TypeScript only supported two editor commands to manage imports. For our examples, take the following code:

```
import { Zebra, Moose, HoneyBadger } from "./zoo";
import { foo, bar } from "./helper";

let x: Moose | HoneyBadger = foo();
```

The first was called "Organize Imports" which would remove unused imports, and then sort the remaining ones. It would rewrite that file to look like this one:

```
import { foo } from "./helper";
import { HoneyBadger, Moose } from "./zoo";

let x: Moose | HoneyBadger = foo();
```

In TypeScript 4.3, we introduced a command called "Sort Imports" which would *only* sort imports in the file, but not remove them - and would rewrite the file like this.

```
import { bar, foo } from "./helper";
import { HoneyBadger, Moose, Zebra } from "./zoo";

let x: Moose | HoneyBadger = foo();
```

The caveat with "Sort Imports" was that in Visual Studio Code, this feature was only available as an on-save command - not as a manually triggerable command.

TypeScript 4.9 adds the other half, and now provides "Remove Unused Imports". TypeScript will now remove unused import names and statements, but will otherwise leave the relative ordering alone.

```
import { Moose, HoneyBadger } from "./zoo";
import { foo } from "./helper";

let x: Moose | HoneyBadger = foo();
```

This feature is available to all editors that wish to use either command; but notably, Visual Studio Code (1.73 and later) will have support built in *and* will surface these commands via its Command Palette. Users who prefer to use the more granular "Remove Unused Imports" or "Sort Imports" commands should be able to reassign the "Organize Imports" key combination to them if desired.

You can [view specifics of the feature here](<https://github.com/microsoft/TypeScript/pull/50931>).

Go-to-Definition on `return` Keywords

In the editor, when running a go-to-definition on the `return` keyword, TypeScript will now jump you to the top of the corresponding function. This can be helpful to get a quick sense of which function a `return` belongs to.

We expect TypeScript will expand this functionality to more keywords [such as `await` and `yield`] (<https://github.com/microsoft/TypeScript/issues/51223>) or [`switch`, `case`, and `default`] (<https://github.com/microsoft/TypeScript/issues/51225>).

[This feature was implemented] (<https://github.com/microsoft/TypeScript/pull/51227>) thanks to [Oleksandr Tarasiuk] (<https://github.com/a-tarasyuk>).

Performance Improvements

TypeScript has a few small, but notable, performance improvements.

First, TypeScript's `forEachChild` function has been rewritten to use a function table lookup instead of a `switch` statement across all syntax nodes. `forEachChild` is a workhorse for traversing syntax nodes in the compiler, and is used heavily in the binding stage of our compiler, along with parts of the language service. The refactoring of `forEachChild` yielded up to a 20% reduction of time spent in our binding phase and across language service operations.

Once we discovered this performance win for `forEachChild`, we tried it out on `visitEachChild`, a function we use for transforming nodes in the compiler and language service. The same refactoring yielded up to a 3% reduction in time spent in generating project output.

The initial exploration in `forEachChild` was [inspired by a blog post] (<https://artemis.sh/2022/08/07/emulating-calculators-fast-in-js.html>) by [Artemis Everfree] (<https://artemis.sh/>). While we have some reason to believe the root cause of our speed-up might have more to do with function size/complexity than the issues described in the blog post, we're grateful that we were able to learn from the experience and try out a relatively quick refactoring that made TypeScript faster.

Finally, the way TypeScript preserves the information about a type in the true branch of a conditional type has been optimized. In a type like

```
interface Zoo<T extends Animal> {
  // ...
}

type MakeZoo<A> = A extends Animal ? Zoo<A> : never;
```

TypeScript has to "remember" that `A` must also be an `Animal` when checking if `Zoo<A>` is valid. This is basically done by creating a special type that used to hold the intersection of `A` with `Animal`; however, TypeScript previously did this eagerly which isn't always necessary. Furthermore, some faulty code in our type-checker prevented these special types from being simplified. TypeScript now defers intersecting these types until it's necessary. For codebases with heavy use of conditional types, you might witness significant speed-ups with TypeScript, but in our performance testing suite, we saw a more modest 3% reduction in type-checking time.

You can read up more on these optimizations on their respective pull requests:

- [`forEachChild` as a jump-table] (<https://github.com/microsoft/TypeScript/pull/50225>)
- [`visitEachChild` as a jump-table] (<https://github.com/microsoft/TypeScript/pull/50266>)

- [Optimize substitution types](<https://github.com/microsoft/TypeScript/pull/50397>)

Correctness Fixes and Breaking Changes

`lib.d.ts` Updates

While TypeScript strives to avoid major breaks, even small changes in the built-in libraries can cause issues. We don't expect major breaks as a result of DOM and `lib.d.ts` updates, but there may be some small ones.

Better Types for `Promise.resolve`

`Promise.resolve` now uses the `Awaited` type to unwrap Promise-like types passed to it. This means that it more often returns the right `Promise` type, but that improved type can break existing code if it was expecting `any` or `unknown` instead of a `Promise`. For more information, [see the original change](<https://github.com/microsoft/TypeScript/pull/33074>).

JavaScript Emit No Longer Elides Imports

When TypeScript first supported type-checking and compilation for JavaScript, it accidentally supported a feature called import elision. In short, if an import is not used as a value, or the compiler can detect that the import doesn't refer to a value at runtime, the compiler will drop the import during emit.

This behavior was questionable, especially the detection of whether the import doesn't refer to a value, since it means that TypeScript has to trust sometimes-inaccurate declaration files. In turn, TypeScript now preserves imports in JavaScript files.

```
// Input:
import { someValue, SomeClass } from "some-module";
/** @type {SomeClass} */
let val = someValue;

// Previous Output:
import { someValue } from "some-module";
/** @type {SomeClass} */
let val = someValue;

// Current Output:
import { someValue, SomeClass } from "some-module";
/** @type {SomeClass} */
let val = someValue;
```

More information is available at [the implementing change](<https://github.com/microsoft/TypeScript/pull/50404>).

exports is Prioritized Over **typesVersions**

Previously, TypeScript incorrectly prioritized the `typesVersions` field over the `exports` field when resolving through a `package.json` under `--moduleResolution node16`. If this change impacts your library, you may need to add `types@` version selectors in your `package.json`'s `exports` field.

```
{
  "type": "module",
  "main": "./dist/main.js"
  "typesVersions": {
    "<4.8": { ".": ["4.8-types/main.d.ts"] },
    "*": { ".": ["modern-types/main.d.ts"] }
  },
  "exports": {
    ".": {
      + "types@<4.8": "4.8-types/main.d.ts",
      + "types": "modern-types/main.d.ts",
      + "import": "./dist/main.js"
    }
  }
}
```

For more information, [see this pull request](<https://github.com/microsoft/TypeScript/pull/50890>).

substitute Replaced With **constraint** on **SubstitutionTypes**

As part of an optimization on substitution types, `SubstitutionType` objects no longer contain the `substitute` property representing the effective substitution (usually an intersection of the base type and the implicit constraint) - instead, they just contain the `constraint` property.

For more details, [read more on the original pull request](<https://github.com/microsoft/TypeScript/pull/50397>). Go to TOC

Install ASP.NET Core and TypeScript

First, install [ASP.NET Core](#) if you need it. This quick-start guide requires Visual Studio 2015 or 2017.

Next, if your version of Visual Studio does not already have the latest TypeScript, you can [install it](#).

Create a new project

1. Choose **File**
2. Choose **New Project** (Ctrl + Shift + N)
3. Search for **.NET Core** in the project search bar
4. Select **ASP.NET Core Web Application** and press the *Next* button



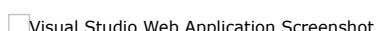
Visual Studio Project Window Screenshot

5. Name your project and solution. After select the *Create* button



Visual Studio New Project Window Screenshot

6. In the last window, select the **Empty** template and press the *Create* button



Visual Studio Web Application Screenshot

Run the application and make sure that it works.



A screenshot of Edge showing "Hello World" as success

Set up the server

Open **Dependencies** > **Manage NuGet Packages** > **Browse**. Search and install `Microsoft.AspNetCore.StaticFiles` and `Microsoft.TypeScript.MSBuild`:



The Visual Studio search for Nuget

Open up your `Startup.cs` file and edit your `Configure` function to look like this:

```

public void Configure(IApplicationBuilder app, IHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseDefaultFiles();
    app.UseStaticFiles();
}

```

You may need to restart VS for the red squiggly lines below `UseDefaultFiles` and `UseStaticFiles` to disappear.

Add TypeScript

Next we will add a new folder and call it `scripts`.

The Path of "Add" then "New Folder" in Visual Studio from a Web Project

The Path of "Add" then "New Folder" in Visual Studio from a Web Project



Add TypeScript code

Right click on `scripts` and click **New Item**. Then choose **TypeScript File** and name the file `app.ts`

A highlight of the new folder

A highlight of the new folder

Add example code

Add the following code to the `app.ts` file.

```

function sayHello() {
    const compiler = (document.getElementById("compiler") as HTMLInputElement)
        .value;
    const framework = (document.getElementById("framework") as HTMLInputElement)
        .value;
    return `Hello from ${compiler} and ${framework}!`;
}

```

Set up the build

Configure the TypeScript compiler

First we need to tell TypeScript how to build. Right click on `scripts` and click **New Item**. Then choose **TypeScript Configuration File** and use the default name of `tsconfig.json`



A screenshot showing the new file dialog with TypeScript JSON Config selected

Replace the contents of the `tsconfig.json` file with:

```
{
  "compilerOptions": {
    "noEmitOnError": true,
    "noImplicitAny": true,
    "sourceMap": true,
    "target": "es6"
  },
  "files": ["./app.ts"],
  "compileOnSave": true
}
```

- `noEmitOnError` : Do not emit outputs if any errors were reported.
- `noImplicitAny` : Raise error on expressions and declarations with an implied `any` type.
- `sourceMap` : Generates corresponding `.map` file.
- `target` : Specify ECMAScript target version.

Note: `"ESNext"` targets latest supported

`noImplicitAny` is good idea whenever you're writing new code — you can make sure that you don't write any untyped code by mistake. `"compileOnSave"` makes it easy to update your code in a running web app.

Set up NPM

We need to setup NPM so that JavaScript packages can be downloaded. Right click on the project and select **New Item**. Then choose **NPM Configuration File** and use the default name of `package.json`.



Screenshot of VS showing new file dialog with 'npm configuration file' selected

Inside the `"devDependencies"` section of the `package.json` file, add `gulp` and `del`

```
"devDependencies": {
  "gulp": "4.0.2",
  "del": "5.1.0"
}
```

Visual Studio should start installing gulp and del as soon as you save the file. If not, right-click `package.json` and then Restore Packages.

After you should see an `npm` folder in your solution explorer



Screenshot of VS showing npm folder

Set up gulp

Right click on the project and click **New Item**. Then choose **JavaScript File** and use the name of `gulpfile.js`

```
/// <binding AfterBuild='default' Clean='clean' />
/*
This file is the main entry point for defining Gulp tasks and using Gulp plugins.
Click here to learn more. http://go.microsoft.com/fwlink/?LinkId=518007
*/
var gulp = require("gulp");
var del = require("del");

var paths = {
  scripts: ["scripts/**/*.{js,ts,map}"],
};

gulp.task("clean", function () {
  return del(["wwwroot/scripts/**/*"]);
});

gulp.task("default", function (done) {
  gulp.src(paths.scripts).pipe(gulp.dest("wwwroot/scripts"));
  done();
});
```

The first line tells Visual Studio to run the task 'default' after the build finishes. It will also run the 'clean' task when you ask Visual Studio to clean the build.

Now right-click on `gulpfile.js` and click Task Runner Explorer.

Screenshot of right clicking on the "Gulpfile.js" with 'Task Runner Explorer' selected

Screenshot of right clicking on the "Gulpfile.js" with 'Task Runner Explorer' selected

If 'default' and 'clean' tasks don't show up, refresh the explorer:

Screenshot of task explorer with "Gulpfile.js" in it

Screenshot of task explorer with "Gulpfile.js" in it

Write a HTML page

Right click on the `wwwroot` folder (if you don't see the folder try building the project) and add a New Item named `index.html` inside. Use the following code for `index.html`

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <script src="scripts/app.js"></script>
  <title></title>
</head>
```

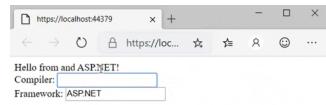
```

<body>
  <div id="message"></div>
  <div>
    Compiler: <input id="compiler" value="TypeScript"
onkeyup="document.getElementById('message').innerText = sayHello()" /><br />
    Framework: <input id="framework" value="ASP.NET"
onkeyup="document.getElementById('message').innerText = sayHello()" />
  </div>
</body>
</html>

```

Test

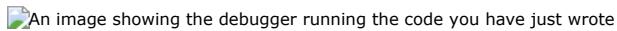
1. Run the project
2. As you type on the boxes you should see the message appear/change!



A GIF of Edge showing the code you have just wrote

Debug

1. In Edge, press F12 and click the Debugger tab.
2. Look in the first localhost folder, then scripts/app.ts
3. Put a breakpoint on the line with return.
4. Type in the boxes and confirm that the breakpoint hits in TypeScript code and that inspection works correctly.



An image showing the debugger running the code you have just wrote

Congrats you've built your own .NET Core project with a TypeScript frontend.

[Go to TOC](#)

Angular is a modern framework built entirely in TypeScript, and as a result, using TypeScript with Angular provides a seamless experience.

The Angular documentation not only supports TypeScript as a first-class citizen, but uses it as its primary language. With this in mind, [Angular's site](#) will always be the most up-to-date reference for using Angular with TypeScript.

Check out the [quick start guide here](#) to start learning Angular now!

[Go to TOC](#)

Babel vs `tsc` for TypeScript

When making a modern JavaScript project, you might ask yourself what is the right way to convert files from TypeScript to JavaScript?

A lot of the time the answer is "*it depends*", or "*someone may have decided for you*" depending on the project. If you are building your project with an existing framework like `tsdx`, `Angular`, `NestJS` or any framework mentioned in the [Getting Started](#) then this decision is handled for you.

However, a useful heuristic could be:

- Is your build output mostly the same as your source input files? Use `tsc`
- Do you need a build pipeline with multiple potential outputs? Use `babel` for transpiling and `tsc` for type checking

Babel for transpiling, `tsc` for types

This is a common pattern for projects with existing build infrastructure which may have been ported from a JavaScript codebase to TypeScript.

This technique is a hybrid approach, using Babel's `preset-typescript` to generate your JS files, and then using TypeScript to do type checking and `.d.ts` file generation.

By using babel's support for TypeScript, you get the ability to work with existing build pipelines and are more likely to have a faster JS emit time because Babel does not type check your code.

Type Checking and `d.ts` file generation

The downside to using babel is that you don't get type checking during the transition from TS to JS. This can mean that type errors which you miss in your editor could sneak through into production code.

In addition to that, Babel cannot create `.d.ts` files for your TypeScript which can make it harder to work with your project if it is a library.

To fix these issues, you would probably want to set up a command to type check your project using TSC. This likely means duplicating some of your babel config into a corresponding `tsconfig.json` and ensuring these flags are enabled:

```
  "compilerOptions": {
    // Ensure that .d.ts files are created by tsc, but not .js files
    "declaration": true,
    "emitDeclarationOnly": true,
    // Ensure that Babel can safely transpile files in the TypeScript project
    "isolatedModules": true
  }
```

For more information on these flags:

- `isolatedModules`

- `declaration`, `emitDeclarationOnly`

[Go to TOC](#)

DOM Manipulation

An exploration into the `HTMLElement` type

In the 20+ years since its standardization, JavaScript has come a very long way. While in 2020, JavaScript can be used on servers, in data science, and even on IoT devices, it is important to remember its most popular use case: web browsers.

Websites are made up of HTML and/or XML documents. These documents are static, they do not change. The *Document Object Model (DOM)* is a programming interface implemented by browsers in order to make static websites functional. The DOM API can be used to change the document structure, style, and content. The API is so powerful that countless frontend frameworks (jQuery, React, Angular, etc.) have been developed around it in order to make dynamic websites even easier to develop.

TypeScript is a typed superset of JavaScript, and it ships type definitions for the DOM API. These definitions are readily available in any default TypeScript project. Of the 20,000+ lines of definitions in `lib.dom.d.ts`, one stands out among the rest: `HTMLElement`. This type is the backbone for DOM manipulation with TypeScript.

You can explore the source code for the `DOM` type definitions

Basic Example

Given a simplified `index.html` file:

```
<!DOCTYPE html>
<html lang="en">
  <head><title>TypeScript Dom Manipulation</title></head>
  <body>
    <div id="app"></div>
    <!-- Assume index.js is the compiled output of index.ts -->
    <script src="index.js"></script>
  </body>
</html>
```

Let's explore a TypeScript script that adds a `<p>Hello, World!</p>` element to the `#app` element.

```
// 1. Select the div element using the id property
const app = document.getElementById("app");

// 2. Create a new <p></p> element programmatically
const p = document.createElement("p");

// 3. Add the text content
p.textContent = "Hello, World!";

// 4. Append the p element to the div element
app?.appendChild(p);
```

After compiling and running the `index.html` page, the resulting HTML will be:

```
<div id="app">
  <p>Hello, World!</p>
</div>
```

The Document Interface

The first line of the TypeScript code uses a global variable `document`. Inspecting the variable shows it is defined by the `Document` interface from the `lib.dom.d.ts` file. The code snippet contains calls to two methods, `getElementById` and `createElement`.

`Document.getElementById`

The definition for this method is as follows:

```
getElementById(elementId: string): HTMLElement | null;
```

Pass it an element id string and it will return either `HTMLElement` or `null`. This method introduces one of the most important types, `HTMLElement`. It serves as the base interface for every other element interface. For example, the `p` variable in the code example is of type `HTMLParagraphElement`. Also take note that this method can return `null`. This is because the method can't be certain pre-runtime if it will be able to actually find the specified element or not. In the last line of the code snippet, the new *optional chaining* operator is used in order to call `appendChild`.

`Document.createElement`

The definition for this method is (I have omitted the *deprecated* definition):

```
createElement<K extends keyof HTMLElementTagNameMap>(tagName: K, options?: ElementCreationOptions): HTMLElementTagNameMap[K];
createElement(tagName: string, options?: ElementCreationOptions): HTMLElement;
```

This is an overloaded function definition. The second overload is simplest and works a lot like the `getElementById` method does. Pass it any `string` and it will return a standard `HTMLElement`. This definition is what enables developers to create unique HTML element tags.

For example `document.createElement('xyz')` returns a `<xyz></xyz>` element, clearly not an element that is specified by the HTML specification.

For those interested, you can interact with custom tag elements using the `document.getElementsByTagName`

For the first definition of `createElement`, it is using some advanced generic patterns. It is best understood broken down into chunks, starting with the generic expression: `<K extends keyof HTMLElementTagNameMap>`. This expression defines a generic parameter `K` that is *constrained* to the keys of the interface `HTMLElementTagNameMap`. The map interface contains every specified HTML tag name and its corresponding type interface. For example here are the first 5 mapped values:

```
interface HTMLElementTagNameMap {
  "a": HTMLAnchorElement;
  "abbr": HTMLElement;
  "address": HTMLElement;
  "applet": HTMLAppletElement;
  "area": HTMLAreaElement;
  ...
}
```

Some elements do not exhibit unique properties and so they just return `HTMLElement`, but other types do have unique properties and methods so they return their specific interface (which will extend from or implement `HTMLElement`).

Now, for the remainder of the `createElement` definition: `(tagName: K, options?: ElementCreationOptions): HTMLElementTagNameMap[K]`. The first argument `tagName` is defined as the generic parameter `K`. The TypeScript interpreter is smart enough to *infer* the generic parameter from this argument. This means that the developer does not actually have to specify the generic parameter when using the method; whatever value is passed to the `tagName` argument will be inferred as `K` and thus can be used throughout the remainder of the definition. Which is exactly what happens; the return value `HTMLElementTagNameMap[K]` takes the `tagName` argument and uses it to return the corresponding type. This definition is how the `p` variable from the code snippet gets a type of `HTMLParagraphElement`. And if the code was `document.createElement('a')`, then it would be an element of type `HTMLAnchorElement`.

The `Node` interface

The `document.getElementById` function returns an `HTMLElement`. `HTMLElement` interface extends the `Element` interface which extends the `Node` interface. This prototypal extension allows for all `HTMLElements` to utilize a subset of standard methods. In the code snippet, we use a property defined on the `Node` interface to append the new `p` element to the website.

`Node.appendChild`

The last line of the code snippet is `app?.appendChild(p)`. The previous, `document.getElementById`, section detailed that the *optional chaining* operator is used here because `app` can potentially be null at run-time. The `appendChild` method is defined by:

```
appendChild<T extends Node>(newChild: T): T;
```

This method works similarly to the `createElement` method as the generic parameter `T` is inferred from the `newChild` argument. `T` is *constrained* to another base interface `Node`.

Difference between `children` and `childNodes`

Previously, this document details the `HTMLElement` interface extends from `Element` which extends from `Node`. In the DOM API there is a concept of `children` elements. For example in the following HTML, the `p` tags are children of the `div` element

```
<div>
  <p>Hello, World</p>
  <p>TypeScript!</p>
</div>

const div = document.getElementsByTagName("div")[0];

div.children;
// HTMLCollection(2) [p, p]

div.childNodes;
// NodeList(2) [p, p]
```

After capturing the `div` element, the `children` prop will return a `HTMLCollection` list containing the `HTMLParagraphElements`. The `childNodes` property will return a similar `NodeList` list of nodes. Each `p` tag will still be of type `HTMLParagraphElements`, but the `NodeList` can contain additional `HTML` nodes that the `HTMLCollection` list cannot.

Modify the html by removing one of the `p` tags, but keep the text.

```
<div>
  <p>Hello, World</p>
  TypeScript!
</div>

const div = document.getElementsByTagName("div")[0];

div.children;
// HTMLCollection(1) [p]

div.childNodes;
// NodeList(2) [p, text]
```

See how both lists change. `children` now only contains the `<p>Hello, World</p>` element, and the `childNodes` contains a `text` node rather than two `p` nodes. The `text` part of the `NodeList` is the literal `Node` containing the text `TypeScript!`. The `children` list does not contain this `Node` because it is not considered an `HTMLElement`.

The `querySelector` and `querySelectorAll` methods

Both of these methods are great tools for getting lists of dom elements that fit a more unique set of constraints. They are defined in `lib.dom.d.ts` as:

```
/** 
 * Returns the first element that is a descendant of node that matches selectors.
 */
querySelector<K extends keyof HTMLElementTagNameMap>(selectors: K):
```

```

HTMLTagNameMap[K] | null;
querySelector<K extends keyof SVGElementTagNameMap>(selectors: K):
SVGElementTagNameMap[K] | null;
querySelector<E extends Element = Element>(selectors: string): E | null;

/**
 * Returns all element descendants of node that match selectors.
 */
querySelectorAll<K extends keyof HTMLElementTagNameMap>(selectors: K):
NodeListOf<HTMLElementTagNameMap[K]>;
querySelectorAll<K extends keyof SVGElementTagNameMap>(selectors: K):
NodeListOf<SVGElementTagNameMap[K]>;
querySelectorAll<E extends Element = Element>(selectors: string): NodeListOf<E>;

```

The `querySelectorAll` definition is similar to `getElementsByName`, except it returns a new type: `NodeListOf`. This return type is essentially a custom implementation of the standard JavaScript list element. Arguably, replacing `NodeListOf<E>` with `E[]` would result in a very similar user experience. `NodeListOf` only implements the following properties and methods: `length`, `item(index)`, `forEach((value, key, parent) => void)`, and numeric indexing. Additionally, this method returns a list of `elements`, not `nodes`, which is what `NodeList` was returning from the `.childNodes` method. While this may appear as a discrepancy, take note that interface `Element` extends from `Node`.

To see these methods in action modify the existing code to:

```

<ul>
  <li>First :)</li>
  <li>Second!</li>
  <li>Third times a charm.</li>
</ul>

const first = document.querySelector("li"); // returns the first li element
const all = document.querySelectorAll("li"); // returns the list of all li
elements

```

Interested in learning more?

The best part about the `lib.dom.d.ts` type definitions is that they are reflective of the types annotated in the Mozilla Developer Network (MDN) documentation site. For example, the `HTMLElement` interface is documented by this [HTML Element page](#) on MDN. These pages list all available properties, methods, and sometimes even examples. Another great aspect of the pages is that they provide links to the corresponding standard documents. Here is the link to the [W3C Recommendation for HTMLElement](#).

Sources:

- [ECMA-262 Standard](#)
- [Introduction to the DOM](#)

[Go to TOC](#)

This quick start guide will teach you how to build TypeScript with [gulp](#) and then add [Browserify](#), [terser](#), or [Watchify](#) to the gulp pipeline. This guide also shows how to add [Babel](#) functionality using [Babelify](#).

We assume that you're already using [Node.js](#) with [npm](#).

Minimal project

Let's start out with a new directory. We'll name it `proj` for now, but you can change it to whatever you want.

```
mkdir proj
cd proj
```

To start, we're going to structure our project in the following way:

```
proj/
  └── src/
      └── dist/
```

TypeScript files will start out in your `src` folder, run through the TypeScript compiler and end up in `dist`.

Let's scaffold this out:

```
mkdir src
mkdir dist
```

Initialize the project

Now we'll turn this folder into an npm package.

```
npm init
```

You'll be given a series of prompts. You can use the defaults except for your entry point. For your entry point, use `./dist/main.js`. You can always go back and change these in the `package.json` file that's been generated for you.

Install our dependencies

Now we can use `npm install` to install packages. First install `gulp-cli` globally (if you use a Unix system, you may need to prefix the `npm install` commands in this guide with `sudo`).

```
npm install -g gulp-cli
```

Then install `typescript`, `gulp` and `gulp-typescript` in your project's dev dependencies. [Gulp-typeScript](#) is a gulp plugin for TypeScript.

```
npm install --save-dev typescript gulp@4.0.0 gulp-typescript
```

Write a simple example

Let's write a Hello World program. In `src`, create the file `main.ts`:

```
function hello(compiler: string) {
  console.log(`Hello from ${compiler}`);
}
hello("TypeScript");
```

In the project root, `proj`, create the file `tsconfig.json`:

```
{
  "files": ["src/main.ts"],
  "compilerOptions": {
    "noImplicitAny": true,
    "target": "es5"
  }
}
```

Create a `gulpfile.js`

In the project root, create the file `gulpfile.js`:

```
var gulp = require("gulp");
var ts = require("gulp-typescript");
var tsProject = ts.createProject("tsconfig.json");

gulp.task("default", function () {
  return tsProject.src().pipe(tsProject()).js.pipe(gulp.dest("dist"));
});
```

Test the resulting app

```
gulp
node dist/main.js
```

The program should print "Hello from TypeScript!".

Add modules to the code

Before we get to Browserify, let's build our code out and add modules to the mix. This is the structure you're more likely to use for a real app.

Create a file called `src/greet.ts`:

```
export function sayHello(name: string) {
  return `Hello from ${name}`;
}
```

Now change the code in `src/main.ts` to import `sayHello` from `greet.ts`:

```
import { sayHello } from "./greet";
console.log(sayHello("TypeScript"));
```

Finally, add `src/greet.ts` to `tsconfig.json`:

```
{
  "files": ["src/main.ts", "src/greet.ts"],
  "compilerOptions": {
    "noImplicitAny": true,
    "target": "es5"
  }
}
```

Make sure that the modules work by running `gulp` and then testing in Node:

```
gulp
node dist/main.js
```

Notice that even though we used ES2015 module syntax, TypeScript emitted CommonJS modules that Node uses. We'll stick with CommonJS for this tutorial, but you could set `module` in the options object to change this.

Browserify

Now let's move this project from Node to the browser. To do this, we'd like to bundle all our modules into one JavaScript file. Fortunately, that's exactly what Browserify does. Even better, it lets us use the CommonJS module system used by Node, which is the default TypeScript emit. That means our TypeScript and Node setup will transfer to the browser basically unchanged.

First, install browserify, `tsify`, and `vinyl-source-stream`. `tsify` is a Browserify plugin that, like `gulp-typescript`, gives access to the TypeScript compiler. `vinyl-source-stream` lets us adapt the file output of Browserify back into a format that `gulp` understands called `vinyl`.

```
npm install --save-dev browserify tsify vinyl-source-stream
```

Create a page

Create a file in `src` named `index.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello World!</title>
  </head>
  <body>
    <p id="greeting">Loading ...</p>
    <script src="bundle.js"></script>
  </body>
</html>
```

Now change `main.ts` to update the page:

```
import { sayHello } from "./greet";

function showHello(divName: string, name: string) {
  const elt = document.getElementById(divName);
  elt.innerText = sayHello(name);
}

showHello("greeting", "TypeScript");
```

Calling `showHello` calls `sayHello` to change the paragraph's text. Now change your gulpfile to the following:

```
var gulp = require("gulp");
var browserify = require("browserify");
var source = require("vinyl-source-stream");
var tsify = require("tsify");
var paths = {
  pages: ["src/*.html"],
};

gulp.task("copy-html", function () {
  return gulp.src(paths.pages).pipe(gulp.dest("dist"));
});

gulp.task(
  "default",
  gulp.series(gulp.parallel("copy-html"), function () {
    return browserify({
      basedir: ".",
      debug: true,
      entries: ["src/main.ts"],
      cache: {},
      packageCache: {}
    })
      .plugin(tsify)
      .bundle()
      .pipe(source("bundle.js"))
      .pipe(gulp.dest("dist"));
  })
);
```

This adds the `copy-html` task and adds it as a dependency of `default`. That means any time `default` is run, `copy-html` has to run first. We've also changed `default` to call Browserify with the `tsify` plugin instead of `gulp-typescript`. Conveniently, they both allow us to pass the same options object to the TypeScript compiler.

After calling `bundle` we use `source` (our alias for `vinyl-source-stream`) to name our output bundle `bundle.js`.

Test the page by running `gulp` and then opening `dist/index.html` in a browser. You should see "Hello from TypeScript" on the page.

Notice that we specified `debug: true` to Browserify. This causes tsify to emit source maps inside the bundled JavaScript file. Source maps let you debug your original TypeScript code in the browser instead of the bundled JavaScript. You can test that source maps are working by opening the debugger for your browser and putting a breakpoint inside `main.ts`. When you refresh the page the breakpoint should pause the page and let you debug `greet.ts`.

Watchify, Babel, and Terser

Now that we are bundling our code with Browserify and tsify, we can add various features to our build with browserify plugins.

- Watchify starts gulp and keeps it running, incrementally compiling whenever you save a file. This lets you keep an edit-save-refresh cycle going in the browser.
- Babel is a hugely flexible compiler that converts ES2015 and beyond into ES5 and ES3. This lets you add extensive and customized transformations that TypeScript doesn't support.
- Terser compacts your code so that it takes less time to download.

Watchify

We'll start with Watchify to provide background compilation:

```
npm install --save-dev watchify fancy-log
```

Now change your gulpfile to the following:

```
var gulp = require("gulp");
var browserify = require("browserify");
var source = require("vinyl-source-stream");
var watchify = require("watchify");
var tsify = require("tsify");
var fancy_log = require("fancy-log");
var paths = {
  pages: ["src/*.html"],
};

var watchedBrowserify = watchify(
  browserify({
    basedir: ".",
    debug: true,
    entries: ["src/main.ts"],
    cache: {},
    packageCache: {}
  }).plugin(tsify)
);

gulp.task("copy-html", function () {
  return gulp.src(paths.pages).pipe(gulp.dest("dist"));
});

function bundle() {
  return watchedBrowserify
    .bundle()
```

```

    .on("error", fancy_log)
    .pipe(source("bundle.js"))
    .pipe(gulp.dest("dist"));
}

gulp.task("default", gulp.series(gulp.parallel("copy-html"), bundle));
watchedBrowserify.on("update", bundle);
watchedBrowserify.on("log", fancy_log);

```

There are basically three changes here, but they require you to refactor your code a bit.

1. We wrapped our `browserify` instance in a call to `watchify`, and then held on to the result.
2. We called `watchedBrowserify.on('update', bundle);` so that Browserify will run the `bundle` function every time one of your TypeScript files changes.
3. We called `watchedBrowserify.on('log', fancy_log);` to log to the console.

Together (1) and (2) mean that we have to move our call to `browserify` out of the `default` task. And we have to give the function for `default` a name since both Watchify and Gulp need to call it. Adding logging with (3) is optional but very useful for debugging your setup.

Now when you run Gulp, it should start and stay running. Try changing the code for `showHello` in `main.ts` and saving it. You should see output that looks like this:

```

proj$ gulp
[10:34:20] Using gulpfile ~/src/proj/gulpfile.js
[10:34:20] Starting "copy-html"...
[10:34:20] Finished "copy-html" after 26 ms
[10:34:20] Starting "default"...
[10:34:21] 2824 bytes written (0.13 seconds)
[10:34:21] Finished "default" after 1.36 s
[10:35:22] 2261 bytes written (0.02 seconds)
[10:35:24] 2808 bytes written (0.05 seconds)

```

Terser

First install Terser. Since the point of Terser is to mangle your code, we also need to install vinyl-buffer and gulp-sourcemaps to keep sourcemaps working.

```
npm install --save-dev gulp-terser vinyl-buffer gulp-sourcemaps
```

Now change your gulpfile to the following:

```

var gulp = require("gulp");
var browserify = require("browserify");
var source = require("vinyl-source-stream");
var terser = require("gulp-terser");
var tsify = require("tsify");
var sourcemaps = require("gulp-sourcemaps");
var buffer = require("vinyl-buffer");
var paths = {
  pages: ["src/*.html"],
};

gulp.task("copy-html", function () {
  return gulp.src(paths.pages).pipe(gulp.dest("dist"));
}

```

```
});
gulp.task(
  "default",
  gulp.series(gulp.parallel("copy-html"), function () {
    return browserify({
      basedir: ".",
      debug: true,
      entries: ["src/main.ts"],
      cache: {},
      packageCache: {}
    })
      .plugin(tsify)
      .bundle()
      .pipe(source("bundle.js"))
      .pipe(buffer())
      .pipe(sourcemaps.init({ loadMaps: true }))
      .pipe(terser())
      .pipe(sourcemaps.write("./"))
      .pipe(gulp.dest("dist"));
  });
});
```

Notice that `terser` itself has just one call — the calls to `buffer` and `sourcemaps` exist to make sure sourcemaps keep working. These calls give us a separate sourcemap file instead of using inline sourcemaps like before. Now you can run Gulp and check that `bundle.js` does get minified into an unreadable mess:

```
gulp
cat dist/bundle.js
```

Babel

First install Babelify and the Babel preset for ES2015. Like Terser, Babelify mangles code, so we'll need `vinyl-buffer` and `gulp-sourcemaps`. By default Babelify will only process files with extensions of `.js`, `.es`, `.es6` and `.jsx` so we need to add the `.ts` extension as an option to Babelify.

```
npm install --save-dev babelify@8 babel-core babel-preset-es2015 vinyl-buffer
gulp-sourcemaps
```

Now change your gulpfile to the following:

```
var gulp = require("gulp");
var browserify = require("browserify");
var source = require("vinyl-source-stream");
var tsify = require("tsify");
var sourcemaps = require("gulp-sourcemaps");
var buffer = require("vinyl-buffer");
var paths = {
  pages: ["src/*.html"],
};

gulp.task("copy-html", function () {
  return gulp.src(paths.pages).pipe(gulp.dest("dist"));
});

gulp.task(
  "default",
```

```
gulp.series(gulp.parallel("copy-html"), function () {
  return browserify({
    basedir: ".",
    debug: true,
    entries: ["src/main.ts"],
    cache: {},
    packageCache: {}
  })
    .plugin(tsify)
    .transform("babelify", {
      presets: ["es2015"],
      extensions: [".ts"]
    })
    .bundle()
    .pipe(source("bundle.js"))
    .pipe(buffer())
    .pipe(sourcemaps.init({ loadMaps: true }))
    .pipe(sourcemaps.write("./"))
    .pipe(gulp.dest("dist"));
});
```

We also need to have TypeScript target ES2015. Babel will then produce ES5 from the ES2015 code that TypeScript emits. Let's modify `tsconfig.json`:

```
{
  "files": ["src/main.ts"],
  "compilerOptions": {
    "noImplicitAny": true,
    "target": "es2015"
  }
}
```

Babel's ES5 output should be very similar to TypeScript's output for such a simple script.

[Go to TOC](#)

TypeScript doesn't exist in a vacuum. It was built with the JavaScript ecosystem in mind, and a lot of JavaScript exists today. Converting a JavaScript codebase over to TypeScript is, while somewhat tedious, usually not challenging. In this tutorial, we're going to look at how you might start out. We assume you've read enough of the handbook to write new TypeScript code.

If you're looking to convert a React project, we recommend looking at the [React Conversion Guide](#) first.

Setting up your Directories

If you're writing in plain JavaScript, it's likely that you're running your JavaScript directly, where your `.js` files are in a `src`, `lib`, or `dist` directory, and then ran as desired.

If that's the case, the files that you've written are going to be used as inputs to TypeScript, and you'll run the outputs it produces. During our JS to TS migration, we'll need to separate our input files to prevent TypeScript from overwriting them. If your output files need to reside in a specific directory, then that will be your output directory.

You might also be running some intermediate steps on your JavaScript, such as bundling or using another transpiler like Babel. In this case, you might already have a folder structure like this set up.

From this point on, we're going to assume that your directory is set up something like this:

```
projectRoot
├── src
│   ├── file1.js
│   └── file2.js
└── built
    └── tsconfig.json
```

If you have a `tests` folder outside of your `src` directory, you might have one `tsconfig.json` in `src`, and one in `tests` as well.

Writing a Configuration File

TypeScript uses a file called `tsconfig.json` for managing your project's options, such as which files you want to include, and what sorts of checking you want to perform. Let's create a bare-bones one for our project:

```
{
  "compilerOptions": {
    "outDir": "./built",
    "allowJs": true,
    "target": "es5"
  },
  "include": ["./src/**/*"]}
```

Here we're specifying a few things to TypeScript:

1. Read in any files it understands in the `src` directory (with `include`).

2. Accept JavaScript files as inputs (with `allowJs`).
3. Emit all of the output files in `built` (with `outDir`).
4. Translate newer JavaScript constructs down to an older version like ECMAScript 5 (using `target`).

At this point, if you try running `tsc` at the root of your project, you should see output files in the `built` directory. The layout of files in `built` should look identical to the layout of `src`. You should now have TypeScript working with your project.

Early Benefits

Even at this point you can get some great benefits from TypeScript understanding your project. If you open up an editor like [VS Code](#) or [Visual Studio](#), you'll see that you can often get some tooling support like completion. You can also catch certain bugs with options like:

- `noImplicitReturns` which prevents you from forgetting to return at the end of a function.
- `noFallthroughCasesInSwitch` which is helpful if you never want to forget a `break` statement between `case`s in a `switch` block.

TypeScript will also warn about unreachable code and labels, which you can disable with `allowUnreachableCode` and `allowUnusedLabels` respectively.

Integrating with Build Tools

You might have some more build steps in your pipeline. Perhaps you concatenate something to each of your files. Each build tool is different, but we'll do our best to cover the gist of things.

Gulp

If you're using Gulp in some fashion, we have a tutorial on [using Gulp](#) with TypeScript, and integrating with common build tools like Browserify, Babelify, and Uglify. You can read more there.

Webpack

Webpack integration is pretty simple. You can use `ts-loader`, a TypeScript loader, combined with `source-map-loader` for easier debugging. Simply run

```
npm install ts-loader source-map-loader
```

and merge in options from the following into your `webpack.config.js` file:

```
module.exports = {
  entry: './src/index.ts',
  output: {
    filename: './dist/bundle.js',
  },
  // Enable sourcemaps for debugging webpack's output.
  devtool: "source-map",
```

```

  resolve: {
    // Add '.ts' and '.tsx' as resolvable extensions.
    extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"],
  },
  module: {
    rules: [
      // All files with a '.ts' or '.tsx' extension will be handled by 'ts-
      loader'.
      { test: /\.tsx?$/, loader: "ts-loader" },

      // All output '.js' files will have any sourcemaps re-processed by 'source-
      map-loader'.
      { test: /\.js$/, loader: "source-map-loader" },
    ],
  },
  // Other options...
};

```

It's important to note that ts-loader will need to run before any other loader that deals with `.js` files.

The same goes for [awesome-typescript-loader](#), another TypeScript loader for Webpack. You can read more about the differences between the two [here](#).

You can see an example of using Webpack in our [tutorial on React and Webpack](#).

Moving to TypeScript Files

At this point, you're probably ready to start using TypeScript files. The first step is to rename one of your `.js` files to `.ts`. If your file uses JSX, you'll need to rename it to `.tsx`.

Finished with that step? Great! You've successfully migrated a file from JavaScript to TypeScript!

Of course, that might not feel right. If you open that file in an editor with TypeScript support (or if you run `tsc --pretty`), you might see red squiggles on certain lines. You should think of these the same way you'd think of red squiggles in an editor like Microsoft Word. TypeScript will still translate your code, just like Word will still let you print your documents.

If that sounds too lax for you, you can tighten that behavior up. If, for instance, you *don't* want TypeScript to compile to JavaScript in the face of errors, you can use the `noEmitOnError` option. In that sense, TypeScript has a dial on its strictness, and you can turn that knob up as high as you want.

If you plan on using the stricter settings that are available, it's best to turn them on now (see [Getting Stricter Checks](#) below). For instance, if you never want TypeScript to silently infer `any` for a type without you explicitly saying so, you can use `noImplicitAny` before you start modifying your files. While it might feel somewhat overwhelming, the long-term gains become apparent much more quickly.

Weeding out Errors

Like we mentioned, it's not unexpected to get error messages after conversion. The important thing is to actually go one by one through these and decide how to deal with the errors. Often these will be legitimate bugs, but sometimes you'll have to explain what you're trying to do a little better to TypeScript.

Importing from Modules

You might start out getting a bunch of errors like `Cannot find name 'require'`, and `Cannot find name 'define'`.. In these cases, it's likely that you're using modules. While you can just convince TypeScript that these exist by writing out

```
// For Node/CommonJS
declare function require(path: string): any;
```

or

```
// For RequireJS/AMD
declare function define(...args: any[]): any;
```

it's better to get rid of those calls and use TypeScript syntax for imports.

First, you'll need to enable some module system by setting TypeScript's `module` option. Valid options are `commonjs`, `amd`, `system`, and `umd`.

If you had the following Node/CommonJS code:

```
var foo = require("foo");
foo.doStuff();
```

or the following RequireJS/AMD code:

```
define(["foo"], function (foo) {
  foo.doStuff();
});
```

then you would write the following TypeScript code:

```
import foo = require("foo");
foo.doStuff();
```

Getting Declaration Files

If you started converting over to TypeScript imports, you'll probably run into errors like `Cannot find module 'foo'`.. The issue here is that you likely don't have *declaration files* to describe your library. Luckily this is pretty easy. If TypeScript complains about a package like `lodash`, you can just write

```
npm install -S @types/lodash
```

If you're using a module option other than `commonjs`, you'll need to set your `moduleResolution` option to `node`.

After that, you'll be able to import lodash with no issues, and get accurate completions.

Exporting from Modules

Typically, exporting from a module involves adding properties to a value like `exports` or `module.exports`. TypeScript allows you to use top-level export statements. For instance, if you exported a function like so:

```
module.exports.feedPets = function (pets) {
  // ...
};
```

you could write that out as the following:

```
export function feedPets(pets) {
  // ...
}
```

Sometimes you'll entirely overwrite the exports object. This is a common pattern people use to make their modules immediately callable like in this snippet:

```
var express = require("express");
var app = express();
```

You might have previously written that like so:

```
function foo() {
  // ...
}
module.exports = foo;
```

In TypeScript, you can model this with the `export =` construct.

```
function foo() {
  // ...
}
export = foo;
```

Too many/too few arguments

You'll sometimes find yourself calling a function with too many/few arguments. Typically, this is a bug, but in some cases, you might have declared a function that uses the `arguments` object instead of writing out any parameters:

```
function myCoolFunction() {
  if (arguments.length == 2 && !Array.isArray(arguments[1])) {
    var f = arguments[0];
    var arr = arguments[1];
    // ...
  }
  // ...
}
```

```

}
myCoolFunction(
  function (x) {
    console.log(x);
  },
  [1, 2, 3, 4]
);
myCoolFunction(
  function (x) {
    console.log(x);
  },
  1,
  2,
  3,
  4
);

```

In this case, we need to use TypeScript to tell any of our callers about the ways `myCoolFunction` can be called using function overloads.

```

function myCoolFunction(f: (x: number) => void, nums: number[]): void;
function myCoolFunction(f: (x: number) => void, ...nums: number[]): void;
function myCoolFunction() {
  if (arguments.length == 2 && !Array.isArray(arguments[1])) {
    var f = arguments[0];
    var arr = arguments[1];
    // ...
  }
  // ...
}

```

We added two overload signatures to `myCoolFunction`. The first checks states that `myCoolFunction` takes a function (which takes a `number`), and then a list of `number`s. The second one says that it will take a function as well, and then uses a rest parameter (`...nums`) to state that any number of arguments after that need to be `number`s.

Sequentially Added Properties

Some people find it more aesthetically pleasing to create an object and add properties immediately after like so:

```

var options = {};
options.color = "red";
options.volume = 11;

```

TypeScript will say that you can't assign to `color` and `volume` because it first figured out the type of `options` as `{}` which doesn't have any properties. If you instead moved the declarations into the object literal themselves, you'd get no errors:

```

let options = {
  color: "red",
  volume: 11,
};

```

You could also define the type of `options` and add a type assertion on the object literal.

```
interface Options {
  color: string;
  volume: number;
}

let options = {} as Options;
options.color = "red";
options.volume = 11;
```

Alternatively, you can just say `options` has the type `any` which is the easiest thing to do, but which will benefit you the least.

`any`, `Object`, and `{}`

You might be tempted to use `Object` or `{}` to say that a value can have any property on it because `Object` is, for most purposes, the most general type. However `any` is actually the type you want to use in those situations, since it's the most *flexible* type.

For instance, if you have something that's typed as `Object` you won't be able to call methods like `toLowercase()` on it. Being more general usually means you can do less with a type, but `any` is special in that it is the most general type while still allowing you to do anything with it. That means you can call it, construct it, access properties on it, etc. Keep in mind though, whenever you use `any`, you lose out on most of the error checking and editor support that TypeScript gives you.

If a decision ever comes down to `Object` and `{}`, you should prefer `{}`. While they are mostly the same, technically `{}` is a more general type than `Object` in certain esoteric cases.

Getting Stricter Checks

TypeScript comes with certain checks to give you more safety and analysis of your program. Once you've converted your codebase to TypeScript, you can start enabling these checks for greater safety.

No Implicit `any`

There are certain cases where TypeScript can't figure out what certain types should be. To be as lenient as possible, it will decide to use the type `any` in its place. While this is great for migration, using `any` means that you're not getting any type safety, and you won't get the same tooling support you'd get elsewhere. You can tell TypeScript to flag these locations down and give an error with the `noImplicitAny` option.

Strict `null` & `undefined` Checks

By default, TypeScript assumes that `null` and `undefined` are in the domain of every type. That means anything declared with the type `number` could be `null` or `undefined`. Since `null` and `undefined` are such a frequent source of bugs in JavaScript and TypeScript, TypeScript has the `strictNullChecks` option to spare you the stress of worrying about these issues.

When `strictNullChecks` is enabled, `null` and `undefined` get their own types called `null` and `undefined` respectively. Whenever anything is *possibly* `null`, you can use a union type with the original type. So for instance, if something could be a `number` or `null`, you'd write the type out as `number | null`.

If you ever have a value that TypeScript thinks is possibly `null / undefined`, but you know better, you can use the postfix `!` operator to tell it otherwise.

```
declare var foo: string[] | null;
foo.length; // error - 'foo' is possibly 'null'
foo!.length; // okay - 'foo!' just has type 'string[]'
```

As a heads up, when using `strictNullChecks`, your dependencies may need to be updated to use `strictNullChecks` as well.

No Implicit `any` for `this`

When you use the `this` keyword outside of classes, it has the type `any` by default. For instance, imagine a `Point` class, and imagine a function that we wish to add as a method:

```
class Point {
    constructor(public x, public y) {}
    getDistance(p: Point) {
        let dx = p.x - this.x;
        let dy = p.y - this.y;
        return Math.sqrt(dx ** 2 + dy ** 2);
    }
}
// ...

// Reopen the interface.
interface Point {
    distanceFromOrigin(): number;
}
Point.prototype.distanceFromOrigin = function () {
    return this.getDistance({ x: 0, y: 0 });
};
```

This has the same problems we mentioned above - we could easily have misspelled `getDistance` and not gotten an error. For this reason, TypeScript has the `noImplicitThis` option. When that option is set, TypeScript will issue an error when `this` is used without an explicit (or inferred) type. The fix is to use a `this`-parameter to give an explicit type in the interface or in the function itself:

```
Point.prototype.distanceFromOrigin = function (this: Point) {
    return this.getDistance({ x: 0, y: 0 });
};
```

[Go to TOC](#)

TypeScript supports [JSX](#) and can correctly model the patterns used in React codebases like `useState`.

Getting Set Up With a React Project

Today there are many frameworks which support TypeScript out of the box:

- [Create React App - TS docs](#)
- [Next.js - TS docs](#)
- [Gatsby - TS Docs](#)

All of these are great starting points. We [use Gatsby](#) with TypeScript for [this website](#), so that can also be a useful reference implementation.

Documentation

Here are some of the best places to find up-to-date information on React and TypeScript:

- [React TypeScript Cheatsheets](#)
- [React & Redux in TypeScript](#)

[Go to TOC](#)

Let's get started by building a simple web application with TypeScript.

Installing TypeScript

There are two main ways to get the TypeScript available for your project:

- Via npm (the Node.js package manager)
- By installing TypeScript's Visual Studio plugins

Visual Studio 2017 and Visual Studio 2015 Update 3 include TypeScript language support by default but does not include the TypeScript compiler, `tsc`. If you didn't install TypeScript with Visual Studio, you can still [download it](#).

For npm users:

```
> npm install -g typescript
```

Building your first TypeScript file

In your editor, type the following JavaScript code in `greeter.ts`:

```
// @noImplicitAny: false
function greeter(person) {
    return "Hello, " + person;
}

let user = "Jane User";
document.body.textContent = greeter(user);
```

Compiling your code

We used a `.ts` extension, but this code is just JavaScript. You could have copy/pasted this straight out of an existing JavaScript app!

At the command line, run the TypeScript compiler:

```
tsc greeter.ts
```

The result will be a file `greeter.js` which contains the same JavaScript that you fed in. We're up and running using TypeScript in our JavaScript app!

Now we can start taking advantage of some of the new tools TypeScript offers. Add a `: string` type annotation to the 'person' function argument as shown here:

```
function greeter(person: string) {
    return "Hello, " + person;
}
```

```
let user = "Jane User";
document.body.textContent = greeter(user);
```

Type annotations

Type annotations in TypeScript are lightweight ways to record the intended contract of the function or variable. In this case, we intend the greeter function to be called with a single string parameter. We can try changing the call greeter to pass an array instead:

```
// @errors: 2345
function greeter(person: string) {
  return "Hello, " + person;
}

let user = [0, 1, 2];
document.body.textContent = greeter(user);
```

Re-compiling, you'll now see an error:

```
error TS2345: Argument of type 'number[]' is not assignable to parameter of type
'string'.
```

Similarly, try removing all the arguments to the greeter call. TypeScript will let you know that you have called this function with an unexpected number of parameters. In both cases, TypeScript can offer static analysis based on both the structure of your code, and the type annotations you provide.

Notice that although there were errors, the `greeter.js` file is still created. You can use TypeScript even if there are errors in your code. But in this case, TypeScript is warning that your code will likely not run as expected.

Interfaces

Let's develop our sample further. Here we use an interface that describes objects that have a `firstName` and `lastName` field. In TypeScript, two types are compatible if their internal structure is compatible. This allows us to implement an interface just by having the shape the interface requires, without an explicit `implements` clause.

```
interface Person {
  firstName: string;
  lastName: string;
}

function greeter(person: Person) {
  return "Hello, " + person.firstName + " " + person.lastName;
}

let user = { firstName: "Jane", lastName: "User" };
document.body.textContent = greeter(user);
```

Classes

Finally, let's extend the example one last time with classes. TypeScript supports new features in JavaScript, like support for class-based object-oriented programming.

Here we're going to create a `Student` class with a constructor and a few public fields. Notice that classes and interfaces play well together, letting the programmer decide on the right level of abstraction.

Also of note, the use of `public` on arguments to the constructor is a shorthand that allows us to automatically create properties with that name.

```
class Student {
  fullName: string;
  constructor(
    public firstName: string,
    public middleInitial: string,
    public lastName: string
  ) {
    this.fullName = firstName + " " + middleInitial + " " + lastName;
  }
}

interface Person {
  firstName: string;
  lastName: string;
}

function greeter(person: Person) {
  return "Hello, " + person.firstName + " " + person.lastName;
}

let user = new Student("Jane", "M.", "User");

document.body.textContent = greeter(user);
```

Re-run `tsc greeter.ts` and you'll see the generated JavaScript is the same as the earlier code. Classes in TypeScript are just a shorthand for the same prototype-based OO that is frequently used in JavaScript.

Running your TypeScript web app

Now type the following in `greeter.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>TypeScript Greeter</title>
  </head>
  <body>
    <script src="greeter.js"></script>
  </body>
</html>
```

Open `greeter.html` in the browser to run your first simple TypeScript web application!

Optional: Open `greeter.ts` in Visual Studio, or copy the code into the TypeScript playground. You can hover over identifiers to see their types. Notice that in some cases these types are inferred automatically for you. Re-type the last line, and see completion lists and parameter help based on the types of the DOM elements. Put your cursor on the reference to the greeter function, and hit F12 to go to its definition. Notice, too, that you can right-click on a symbol and use refactoring to rename it.

The type information provided works together with the tools to work with JavaScript at application scale. For more examples of what's possible in TypeScript, see the Samples section of the website.



[Go to TOC](#)

Colophon

This book is created by using the following sources:

- Typescript - English
- GitHub source: [microsoft/TypeScript-Website/packages/documentation/copy/en](https://github.com/microsoft/TypeScript-Website/packages/documentation/copy/en)
- Created: 2022-12-11
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- Viewer: <https://read-html-download-pdf.vercel.app/>
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>
- Viewer repo: <https://github.com/shinokada/read-html-download-pdf>