Intro to the TSConfig Reference

A TSConfig file in a directory indicates that the directory is the root of a TypeScript or JavaScript project...

Compiler Options

Top Level

1. files,
2. extends,
3. include,
4. exclude and
5. references

"compilerOptions"

Type Checking

1. allowUnreachableCode,
2. allowUnusedLabels,
3. alwaysStrict,
4. exactOptionalPropertyTypes,
5. noFallthroughCasesInSwitch,
6. noImplicitAny,
7. noImplicitOverride,
8. noImplicitReturns,
9. noImplicitThis,
10. noPropertyAccessFromIndexSignature,
11. noUncheckedIndexedAccess,
12. noUnusedLocals,
13. noUnusedParameters,
14. strict,
15. strictBindCallApply,
16. strictFunctionTypes,
17. strictNullChecks,
18. strictPropertyInitialization and
19. useUnknownInCatchVariables

Modules

1. allowArbitraryExtensions,
2. allowImportingTsExtensions,
3. allowUmdGlobalAccess,
4. baseUrl,
5. customConditions,
6. module,
7. moduleResolution,
8. moduleSuffixes,
9. noResolve,
10. paths,
11. resolveJsonModule,
12. resolvePackageJsonExports,
13. resolvePackageJsonImports,
14. rootDir,
15. rootDirs,
16. typeRoots and
17. types

Emit

1. declaration,
2. declarationDir,
3. declarationMap,

4. downlevelIteration,
5. emitBOM,
6. emitDeclarationOnly,
7. importHelpers,
8. importsNotUsedAsValues,
9. inlineSourceMap,
10. inlineSources,
11. mapRoot,
12. newLine,
13. noEmit,
14. noEmitHelpers,
15. noEmitOnError,
16. outDir,
17. outFile,
18. preserveConstEnums,
19. preserveValueImports,
20. removeComments,
21. sourceMap,
22. sourceRoot and
23. stripInternal

JavaScript Support

1. allowJs,
2. checkJs and
3. maxNodeModuleJsDepth

Editor Support

1. disableSizeLimit and
2. plugins

Interop Constraints

1. allowSyntheticDefaultImports,
2. esModuleInterop,
3. forceConsistentCasingInFileNames,
4. isolatedModules,
5. preserveSymlinks and
6. verbatimModuleSyntax

Backwards Compatibility

1. charset,
2. keyofStringsOnly,
3. noImplicitUseStrict,
4. noStrictGenericChecks,
5. out,
6. suppressExcessPropertyErrors and
7. suppressImplicitAnyIndexErrors

Language and Environment

1. emitDecoratorMetadata,
2. experimentalDecorators,
3. jsx,
4. jsxFactory,
5. jsxFragmentFactory,
6. jsxImportSource,
7. lib,
8. moduleDetection,
9. noLib,
10. reactNamespace,

11. target and
12. useDefineForClassFields

Compiler Diagnostics

1. diagnostics,
2. explainFiles,
3. extendedDiagnostics,
4. generateCpuProfile,
5. listEmittedFiles,
6. listFiles and
7. traceResolution

Projects

1. composite,
2. disableReferencedProjectLoad,
3. disableSolutionSearching,
4. disableSourceOfProjectReferenceRedirect,
5. incremental and
6. tsBuildInfoFile

Output Formatting

1. noErrorTruncation,
2. preserveWatchOutput and
3. pretty

Completeness

1. skipDefaultLibCheck and
2. skipLibCheck

Command Line

Watch Options

1. assumeChangesOnlyAffectDirectDependencies

"watchOptions"

watchOptions

1. watchFile,
2. watchDirectory,
3. fallbackPolling,
4. synchronousWatchDirectory,
5. excludeDirectories and
6. excludeFiles

"typeAcquisition"

typeAcquisition

1. enable,
2. include,
3. exclude and
4. disableFilenameBasedTypeAcquisition

Root Fields

Starting up are the root options in the TSConfig - these options relate to how your TypeScript or JavaScript project is set up.

Files - `files`

Specifies an allowlist of files to include in the program. An error occurs if any of the files can't be found.

```
{
  "compilerOptions": {},
  "files": [
    "core.ts",
    "sys.ts",
    "types.ts",
    "scanner.ts",
    "parser.ts",
    "utilities.ts",
    "binder.ts",
    "checker.ts",
    "tsc.ts"
  ]
}
```

This is useful when you only have a small number of files and don't need to use a glob to reference many files. If you need that then use `include`.

Default:

`false`

Related:

`include`

`exclude`

Extends - `extends`

The value of `extends` is a string which contains a path to another configuration file to inherit from. The path may use Node.js style resolution.

The configuration from the base file are loaded first, then overridden by those in the inheriting config file. All relative paths found in the configuration file will be resolved relative to the configuration file they originated in.

It's worth noting that `files`, `include`, and `exclude` from the inheriting config file *overwrite* those from the base config file, and that circularity between configuration files is not allowed.

Currently, the only top-level property that is excluded from inheritance is `references`.

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
  }
}
```

Properties with relative paths found in the configuration file, which aren't excluded from inheritance, will be resolved relative to the configuration file they originated in.

Default:

`false`

Released:

2.1

Include - `include`

Specifies an array of filenames or patterns to include in the program. These filenames are resolved relative to the directory containing the `tsconfig.json` file.

json

```
{
  "include": ["src/**/*", "tests/**/*"]
}
```

Which would include:

```
.
■■■ scripts                ■
■   ■■■ lint.ts            ■
■   ■■■ update_deps.ts     ■
■   ■■■ utils.ts           ■
■■■ src                    ✓
■   ■■■ client             ✓
■   ■   ■■■ index.ts       ✓
■   ■   ■■■ utils.ts       ✓
■   ■■■ server             ✓
■   ■   ■■■ index.ts       ✓
■■■ tests                  ✓
■   ■■■ app.test.ts        ✓
■   ■■■ utils.ts           ✓
■   ■■■ tests.d.ts         ✓
■■■ package.json
```

■■■ `tsconfig.json`

■■■ `yarn.lock`

`include` and `exclude` support wildcard characters to make glob patterns:

- `*` matches zero or more characters (excluding directory separators)
- `?` matches any one character (excluding directory separators)
- `**/` matches any directory nested to any level

If the last path segment in a pattern does not contain a file extension or wildcard character, then it is treated as a directory, and files with supported extensions inside that directory are included (e.g. `.ts`, `.tsx`, and `.d.ts` by default, with `.js` and `.jsx` if `allowJs` is set to true).

Default:

`[]` if `files` is specified; `**/*` otherwise.

Related:

`files`

`exclude`

Released:

2.0

Exclude - `exclude`

Specifies an array of filenames or patterns that should be skipped when resolving `include`.

Important: `exclude` *only* changes which files are included as a result of the `include` setting. A file specified by `exclude` can still become part of your codebase due to an `import` statement in your code, a `types` inclusion, a `/// <reference` directive, or being specified in the `files` list.

It is not a mechanism that **prevents** a file from being included in the codebase - it simply changes what the `include` setting finds.

Default:

node_modules bower_components jspm_packages `outDir`

Related:

`include`

`files`

References - `references`

Project references are a way to structure your TypeScript programs into smaller pieces. Using Project References can greatly improve build and editor interaction times, enforce logical separation between components, and organize your code in new and improved ways.

You can read more about how references works in the Project References section of the handbook

Default:

`false`

Compiler Options

These options make up the bulk of TypeScript's configuration and it covers how the language should work.

- Type Checking
- Modules
- Emit
- JavaScript Support
- Editor Support
- Interop Constraints
- Backwards Compatibility
- Language and Environment

- Compiler Diagnostics
- Projects
- Output Formatting
- Completeness
- Command Line
- Watch Options

#Type Checking

Allow Unreachable Code - `allowUnreachableCode`

When:

- `undefined` (default) provide suggestions as warnings to editors
- `true` unreachable code is ignored
- `false` raises compiler errors about unreachable code

These warnings are only about code which is provably unreachable due to the use of JavaScript syntax, for example:

```ts
function fn(n: number) {
  if (n > 5) {
    return true;
  } else {
    return false;
  }
  return true;
}
```

With `"allowUnreachableCode": false`:

```ts
function fn(n: number) {
  if (n > 5) {
    return true;
  } else {
    return false;
  }
  return true;
Unreachable code detected.7027Unreachable code detected.
}
```

Try

This does not affect errors on the basis of code which *appears* to be unreachable due to type analysis.

Released:

1.8

Allow Unused Labels - `allowUnusedLabels`

When:

- `undefined` (default) provide suggestions as warnings to editors
- `true` unused labels are ignored
- `false` raises compiler errors about unused labels

Labels are very rare in JavaScript and typically indicate an attempt to write an object literal:

ts

```
function verifyAge(age: number) {
  // Forgot 'return' statement
  if (age > 18) {
    verified: true;
Unused label.7028Unused label.
  }
}
```

Try

Released:

1.8

Always Strict - `alwaysStrict`

Ensures that your files are parsed in the ECMAScript strict mode, and emit “use strict” for each source file.

ECMAScript strict mode was introduced in ES5 and provides behavior tweaks to the runtime of the JavaScript engine to improve performance, and makes a set of errors throw instead of silently ignoring them.

- Recommended
  Default:

  `true` if `strict`; `false` otherwise.

  Related:

  `strict`

  Released:

  2.1

Exact Optional Property Types - `exactOptionalPropertyTypes`

With exactOptionalPropertyTypes enabled, TypeScript applies stricter rules around how it handles properties on `type` or `interfaces` which have a ? prefix.

For example, this interface declares that there is a property which can be one of two strings: ‘dark’ or ‘light’ or it should not be in the object.

ts

```
interface UserDefaults {
  // The absence of a value represents 'system'
  colorThemeOverride?: "dark" | "light";
}
```

Without this flag enabled, there are three values which you can set `colorThemeOverride` to be: “dark”, “light” and `undefined`.

Setting the value to `undefined` will allow most JavaScript runtime checks for the existence to fail, which is effectively falsy. However, this isn’t quite accurate; `colorThemeOverride: undefined` is not the same as `colorThemeOverride` not being defined. For example, `"colorThemeOverride" in settings` would have different behavior with `undefined` as the key compared to not being defined.

`exactOptionalPropertyTypes` makes TypeScript truly enforce the definition provided as an optional property:

ts

```ts
const settings = getUserSettings();

settings.colorThemeOverride = "dark";

settings.colorThemeOverride = "light";

// But not:

settings.colorThemeOverride = undefined;
```

```
Type 'undefined' is not assignable to type '"dark" | "light"' with 'exactOptionalPropertyTypes: true'. Consider adding 'undefi
```

- Recommended
  Released:

  4.4

No Fallthrough Cases In Switch - `noFallthroughCasesInSwitch`

Report errors for fallthrough cases in switch statements. Ensures that any non-empty case inside a switch statement includes either `break`, `return`, or `throw`. This means you won't accidentally ship a case fallthrough bug.

ts

```ts
const a: number = 6;

switch (a) {

  case 0:
```

```
Fallthrough case in switch.7029Fallthrough case in switch.
```

```ts
    console.log("even");

  case 1:

    console.log("odd");

    break;

}
```

Try

Released:

1.8

No Implicit Any - `noImplicitAny`

In some cases where no type annotations are present, TypeScript will fall back to a type of `any` for a variable when it cannot infer the type.

This can cause some errors to be missed, for example:

ts

```ts
function fn(s) {

  // No error?

  console.log(s.subtr(3));

}

fn(42);
```

Try

Turning on `noImplicitAny` however TypeScript will issue an error whenever it would have inferred `any`:

ts

```
function fn(s) {

Parameter 's' implicitly has an 'any' type.7006Parameter 's' implicitly has an 'any' type.

  console.log(s.subtr(3));

}
```

Try

- Recommended

  Default:

  `true` if `strict`; `false` otherwise.

  Related:

  - `strict`

No Implicit Override - `noImplicitOverride`

When working with classes which use inheritance, it's possible for a sub-class to get "out of sync" with the functions it overloads when they are renamed in the base class.

For example, imagine you are modeling a music album syncing system:

```ts
class Album {
  download() {
    // Default behavior
  }
}
class SharedAlbum extends Album {
  download() {
    // Override to get info from many sources
  }
}
```

Try

Then when you add support for machine-learning generated playlists, you refactor the `Album` class to have a 'setup' function instead:

```ts
class Album {
  setup() {
    // Default behavior
  }
}
class MLAlbum extends Album {
  setup() {
    // Override to get info from algorithm
  }
}
```

```ts
class SharedAlbum extends Album {
  download() {
    // Override to get info from many sources
  }
}
```

Try

In this case, TypeScript has provided no warning that `download` on `SharedAlbum` *expected* to override a function in the base class.

Using `noImplicitOverride` you can ensure that the sub-classes never go out of sync, by ensuring that functions which override include the keyword `override`.

The following example has `noImplicitOverride` enabled, and you can see the error received when `override` is missing:

```ts
class Album {
  setup() {}
}
class MLAlbum extends Album {
  override setup() {}
}
class SharedAlbum extends Album {
  setup() {}
This member must have an 'override' modifier because it overrides a member in the base class 'Album'.4114This member must have
}
```

Try

Released:

4.3

No Implicit Returns - `noImplicitReturns`

When enabled, TypeScript will check all code paths in a function to ensure they return a value.

```ts
function lookupHeadphonesManufacturer(color: "blue" | "black"): string {
Function lacks ending return statement and return type does not include 'undefined'.2366Function lacks ending return statement
  if (color === "blue") {
    return "beats";
  } else {
    ("bose");
  }
}
```

Try

Released:

1.8

No Implicit This - `noImplicitThis`

Raise error on ‘this’ expressions with an implied ‘any’ type.

For example, the class below returns a function which tries to access `this.width` and `this.height` – but the context for `this` inside the function inside `getAreaFunction` is not the instance of the Rectangle.

ts

```
class Rectangle {
  width: number;
  height: number;
  constructor(width: number, height: number) {
    this.width = width;
    this.height = height;
  }
  getAreaFunction() {
    return function () {
      return this.width * this.height;
'this' implicitly has type 'any' because it does not have a type annotation.'this' implicitly has type 'any' because it does n
2683'this' implicitly has type 'any' because it does not have a type annotation.'this' implicitly has type 'any' because it do
    };
  }
}
```

Try

- Recommended
  Default:

  `true` if `strict`; `false` otherwise.

  Related:

  - `strict`

  Released:

  2.0

No Property Access From Index Signature - `noPropertyAccessFromIndexSignature`

This setting ensures consistency between accessing a field via the “dot” (`obj.key`) syntax, and “indexed” (`obj["key"]`) and the way which the property is declared in the type.

Without this flag, TypeScript will allow you to use the dot syntax to access fields which are not defined:

ts

```
interface GameSettings {
  // Known up-front properties
  speed: "fast" | "medium" | "slow";
  quality: "high" | "low";
  // Assume anything unknown to the interface
  // is a string.
```

```
  [key: string]: string;
}
const settings = getSettings();
settings.speed;
(property) GameSettings.speed: "fast" | "medium" | "slow"
settings.quality;
(property) GameSettings.quality: "high" | "low"
// Unknown key accessors are allowed on
// this object, and are `string`
settings.username;
(index) GameSettings[string]: string
```

Try

Turning the flag on will raise an error because the unknown field uses dot syntax instead of indexed syntax.

```
ts
const settings = getSettings();
settings.speed;
settings.quality;
// This would need to be settings["username"];
settings.username;
Property 'username' comes from an index signature, so it must be accessed with ['username'].4111Property 'username' comes from
(index) GameSettings[string]: string
```

Try

The goal of this flag is to signal intent in your calling syntax about how certain you are this property exists.

Released:

4.2

No Unchecked Indexed Access - noUncheckedIndexedAccess

TypeScript has a way to describe objects which have unknown keys but known values on an object, via index signatures.

```
ts
interface EnvironmentVars {
  NAME: string;
  OS: string;
  // Unknown properties are covered by this index signature.
  [propName: string]: string;
}
declare const env: EnvironmentVars;
// Declared as existing
const sysName = env.NAME;
```

```
const os = env.OS;

const os: string

// Not declared, but because of the index

// signature, then it is considered a string

const nodeEnv = env.NODE_ENV;

const nodeEnv: string
```

Try

Turning on `noUncheckedIndexedAccess` will add `undefined` to any un-declared field in the type.

```
ts

declare const env: EnvironmentVars;

// Declared as existing

const sysName = env.NAME;

const os = env.OS;

const os: string

// Not declared, but because of the index

// signature, then it is considered a string

const nodeEnv = env.NODE_ENV;

const nodeEnv: string | undefined
```

Try

Released:

4.1

No Unused Locals - `noUnusedLocals`

Report errors on unused local variables.

```
ts

const createKeyboard = (modelID: number) => {

  const defaultModelID = 23;

'defaultModelID' is declared but its value is never read.6133'defaultModelID' is declared but its value is never read.

  return { type: "keyboard", modelID };

};
```

Try

Released:

2.0

No Unused Parameters - `noUnusedParameters`

Report errors on unused parameters in functions.

```
ts

const createDefaultKeyboard = (modelID: number) => {

'modelID' is declared but its value is never read.6133'modelID' is declared but its value is never read.
```

```
  const defaultModelID = 23;
  return { type: "keyboard", modelID: defaultModelID };
};
```

Try

Released:

2.0

Strict - `strict`

The `strict` flag enables a wide range of type checking behavior that results in stronger guarantees of program correctness. Turning this on is equivalent to enabling all of the *strict mode family* options, which are outlined below. You can then turn off individual strict mode family checks as needed.

Future versions of TypeScript may introduce additional stricter checking under this flag, so upgrades of TypeScript might result in new type errors in your program. When appropriate and possible, a corresponding flag will be added to disable that behavior.

- Recommended
  Related:
  - alwaysStrict
  - strictNullChecks
  - strictBindCallApply
  - strictFunctionTypes
  - strictPropertyInitialization
  - noImplicitAny
  - noImplicitThis
  - useUnknownInCatchVariables

  Released:

  2.3

Strict Bind Call Apply - `strictBindCallApply`

When set, TypeScript will check that the built-in methods of functions `call`, `bind`, and `apply` are invoked with correct argument for the underlying function:

```
ts
// With strictBindCallApply on
function fn(x: string) {
  return parseInt(x);
}
const n1 = fn.call(undefined, "10");
const n2 = fn.call(undefined, false);
Argument of type 'boolean' is not assignable to parameter of type 'string'.2345Argument of type 'boolean' is not assignable to
```

Otherwise, these functions accept any arguments and will return `any`:

```
ts
// With strictBindCallApply off
function fn(x: string) {
  return parseInt(x);
```

```
}

// Note: No error; return type is 'any'

const n = fn.call(undefined, false);
```

Try

- Recommended
  Default:

  `true` if `strict`; `false` otherwise.

  Related:

  `strict`

  Released:

  3.2

Strict Function Types - `strictFunctionTypes`

When enabled, this flag causes functions parameters to be checked more correctly.

Here's a basic example with `strictFunctionTypes` off:

ts

```
function fn(x: string) {
  console.log("Hello, " + x.toLowerCase());
}

type StringOrNumberFunc = (ns: string | number) => void;

// Unsafe assignment
let func: StringOrNumberFunc = fn;

// Unsafe call - will crash
func(10);
```

Try

With `strictFunctionTypes` *on*, the error is correctly detected:

ts

```
function fn(x: string) {
  console.log("Hello, " + x.toLowerCase());
}

type StringOrNumberFunc = (ns: string | number) => void;

// Unsafe assignment is prevented
let func: StringOrNumberFunc = fn;

Type '(x: string) => void' is not assignable to type 'StringOrNumberFunc'.
 Types of parameters 'x' and 'ns' are incompatible.
   Type 'string | number' is not assignable to type 'string'.
     Type 'number' is not assignable to type 'string'.2322Type '(x: string) => void' is not assignable to type 'StringOrNumber
 Types of parameters 'x' and 'ns' are incompatible.
   Type 'string | number' is not assignable to type 'string'.
     Type 'number' is not assignable to type 'string'.Try
```

During development of this feature, we discovered a large number of inherently unsafe class hierarchies, including some in the DOM. Because of this, the setting only applies to functions written in *function* syntax, not to those in *method* syntax:

```ts
type Methodish = {
  func(x: string | number): void;
};
function fn(x: string) {
  console.log("Hello, " + x.toLowerCase());
}
// Ultimately an unsafe assignment, but not detected
const m: Methodish = {
  func: fn,
};
m.func(10);
```

Try

- Recommended

  Default:

  `true` if `strict`; `false` otherwise.

  Related:

  `strict`

  Released:

  2.6

Strict Null Checks - `strictNullChecks`

When `strictNullChecks` is `false`, `null` and `undefined` are effectively ignored by the language. This can lead to unexpected errors at runtime.

When `strictNullChecks` is `true`, `null` and `undefined` have their own distinct types and you'll get a type error if you try to use them where a concrete value is expected.

For example with this TypeScript code, `users.find` has no guarantee that it will actually find a user, but you can write code as though it will:

```ts
declare const loggedInUsername: string;
const users = [
  { name: "Oby", age: 12 },
  { name: "Heera", age: 32 },
];
const loggedInUser = users.find((u) => u.name === loggedInUsername);
console.log(loggedInUser.age);
```

Try

Setting `strictNullChecks` to `true` will raise an error that you have not made a guarantee that the `loggedInUser` exists before trying to use it.

```ts
declare const loggedInUsername: string;
const users = [
```

```
  { name: "Oby", age: 12 },
  { name: "Heera", age: 32 },
];
const loggedInUser = users.find((u) => u.name === loggedInUsername);
console.log(loggedInUser.age);
'loggedInUser' is possibly 'undefined'.18048'loggedInUser' is possibly 'undefined'.Try
```

The second example failed because the array's `find` function looks a bit like this simplification:

```
ts
// When strictNullChecks: true
type Array = {
  find(predicate: (value: any, index: number) => boolean): S | undefined;
};
// When strictNullChecks: false the undefined is removed from the type system,
// allowing you to write code which assumes it always found a result
type Array = {
  find(predicate: (value: any, index: number) => boolean): S;
};
```

- Recommended

  Default:

  `true` if `strict`; `false` otherwise.

  Related:

  `strict`

  Released:

  2.0

Strict Property Initialization - `strictPropertyInitialization`

When set to true, TypeScript will raise an error when a class property was declared but not set in the constructor.

```
ts
class UserAccount {
  name: string;
  accountType = "user";
  email: string;
Property 'email' has no initializer and is not definitely assigned in the constructor.2564Property 'email' has no initializer
  address: string | undefined;
  constructor(name: string) {
    this.name = name;
    // Note that this.email is not set
  }
}
```

Try

In the above case:

- `this.name` is set specifically.
- `this.accountType` is set by default.
- `this.email` is not set and raises an error.
- `this.address` is declared as potentially `undefined` which means it does not have to be set.
- Recommended
  Default:

  `true` if `strict`; `false` otherwise.

  Related:

  - `strict`

  Released:

  2.7

Use Unknown In Catch Variables - `useUnknownInCatchVariables`

In TypeScript 4.0, support was added to allow changing the type of the variable in a catch clause from `any` to `unknown`. Allowing for code like:

```ts
try {
  // ...
} catch (err: unknown) {
  // We have to verify err is an
  // error before using it as one.
  if (err instanceof Error) {
    console.log(err.message);
  }
}
```

Try

This pattern ensures that error handling code becomes more comprehensive because you cannot guarantee that the object being thrown *is* a Error subclass ahead of time. With the flag `useUnknownInCatchVariables` enabled, then you do not need the additional syntax (`: unknown`) nor a linter rule to try enforce this behavior.

- Recommended
  Default:

  `true` if `strict`; `false` otherwise.

  Related:

  - `strict`

  Released:

  4.4

#Modules

Allow Arbitrary Extensions - `allowArbitraryExtensions`

In TypeScript 5.0, when an import path ends in an extension that isn't a known JavaScript or TypeScript file extension, the compiler will look for a declaration file for that path in the form of `{file basename}.d.{extension}.ts`. For example, if you are using a CSS loader in a bundler project, you might want to write (or generate) declaration files for those stylesheets:

```
css

/* app.css */

.cookie-banner {

  display: none;

}

ts

// app.d.css.ts

declare const css: {

  cookieBanner: string;

};

export default css;

ts

// App.tsx

import styles from "./app.css";

styles.cookieBanner; // string
```

By default, this import will raise an error to let you know that TypeScript doesn't understand this file type and your runtime might not support importing it. But if you've configured your runtime or bundler to handle it, you can suppress the error with the new `--allowArbitraryExtensions` compiler option.

Note that historically, a similar effect has often been achievable by adding a declaration file named `app.css.d.ts` instead of `app.d.css.ts` - however, this just worked through Node's `require` resolution rules for CommonJS. Strictly speaking, the former is interpreted as a declaration file for a JavaScript file named `app.css.js`. Because relative files imports need to include extensions in Node's ESM support, TypeScript would error on our example in an ESM file under `--moduleResolution node16` or `nodenext`.

For more information, read up the proposal for this feature and its corresponding pull request.

Allow Importing TS Extensions - `allowImportingTsExtensions`

`--allowImportingTsExtensions` allows TypeScript files to import each other with a TypeScript-specific extension like `.ts`, `.mts`, or `.tsx`.

This flag is only allowed when `--noEmit` or `--emitDeclarationOnly` is enabled, since these import paths would not be resolvable at runtime in JavaScript output files. The expectation here is that your resolver (e.g. your bundler, a runtime, or some other tool) is going to make these imports between `.ts` files work.

Allow Umd Global Access - `allowUmdGlobalAccess`

When set to true, `allowUmdGlobalAccess` lets you access UMD exports as globals from inside module files. A module file is a file that has imports and/or exports. Without this flag, using an export from a UMD module requires an import declaration.

An example use case for this flag would be a web project where you know the particular library (like jQuery or Lodash) will always be available at runtime, but you can't access it with an import.

Released:

3.5

Base URL - `baseUrl`

Sets a base directory from which to resolve bare specifier module names. For example, in the directory structure:

```
project

■■■ ex.ts

■■■ hello

■   ■■■ world.ts

■■■ tsconfig.json
```

With `"baseUrl": "./"`, TypeScript will look for files starting at the same folder as the `tsconfig.json`:

ts

```ts
import { helloWorld } from "hello/world";

console.log(helloWorld);
```

This resolution has higher priority than lookups from `node_modules`.

This feature was designed for use in conjunction with AMD module loaders in the browser, and is not recommended in any other context. As of TypeScript 4.1, `baseUrl` is no longer required to be set when using `paths`.

Custom Conditions - `customConditions`

`--customConditions` takes a list of additional conditions that should succeed when TypeScript resolves from an `exports` or `imports` field of a `package.json`. These conditions are added to whatever existing conditions a resolver will use by default.

For example, when this field is set in a `tsconfig.json` as so:

jsonc

```jsonc
{
  "compilerOptions": {
    "target": "es2022",
    "moduleResolution": "bundler",
    "customConditions": ["my-condition"]
  }
}
```

Any time an `exports` or `imports` field is referenced in `package.json`, TypeScript will consider conditions called `my-condition`.

So when importing from a package with the following `package.json`

jsonc

```jsonc
{
  // ...
  "exports": {
    ".": {
      "my-condition": "./foo.mjs",
      "node": "./bar.mjs",
      "import": "./baz.mjs",
      "require": "./biz.mjs"
    }
  }
}
```

TypeScript will try to look for files corresponding to `foo.mjs`.

This field is only valid under the `node16`, `nodenext`, and `bundler` options for `--moduleResolution`.

Related:

- `moduleResolution`
- `resolvePackageJsonExports`

resolvePackageJsonImports

Module - `module`

Sets the module system for the program. See the theory behind TypeScript's `module` option and its reference page for more information. You very likely want `"nodenext"` for modern Node.js projects and `preserve` or `esnext` for code that will be bundled.

Changing `module` affects `moduleResolution` which also has a reference page.

Here's some example output for this file:

ts

```ts
// @filename: index.ts
import { valueOfPi } from "./constants";
export const twoPi = valueOfPi * 2;
```

Try

CommonJS

ts

```ts
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
exports.twoPi = void 0;
const constants_1 = require("./constants");
exports.twoPi = constants_1.valueOfPi * 2;
```

Try

UMD

ts

```ts
(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", "./constants"], factory);
    }
})(function (require, exports) {
    "use strict";
    Object.defineProperty(exports, "__esModule", { value: true });
    exports.twoPi = void 0;
    const constants_1 = require("./constants");
    exports.twoPi = constants_1.valueOfPi * 2;
});
```

Try

AMD

ts

```ts
define(["require", "exports", "./constants"], function (require, exports, constants_1) {
    "use strict";
    Object.defineProperty(exports, "__esModule", { value: true });
    exports.twoPi = void 0;
    exports.twoPi = constants_1.valueOfPi * 2;
});
```

Try

System

ts

```ts
System.register(["./constants"], function (exports_1, context_1) {
    "use strict";
    var constants_1, twoPi;
    var __moduleName = context_1 && context_1.id;
    return {
        setters: [
            function (constants_1_1) {
                constants_1 = constants_1_1;
            }
        ],
        execute: function () {
            exports_1("twoPi", twoPi = constants_1.valueOfPi * 2);
        }
    };
});
```

Try

ESNext

ts

```ts
import { valueOfPi } from "./constants";
export const twoPi = valueOfPi * 2;
```

Try

ES2015/ES6/ES2020/ES2022

ts

```ts
import { valueOfPi } from "./constants";
export const twoPi = valueOfPi * 2;
```

Try

In addition to the base functionality of `ES2015`/`ES6`, `ES2020` adds support for dynamic `imports`, and `import.meta` while `ES2022` further adds support for top level `await`.

node16/nodenext

Available from 4.7+, the `node16` and `nodenext` modes integrate with Node's native ECMAScript Module support. The emitted JavaScript uses either `CommonJS` or `ES2020` output depending on the file extension and the value of the `type` setting in the nearest `package.json`. Module resolution also works differently. You can learn more in the handbook and Modules Reference.

preserve

In `--module preserve` (added in TypeScript 5.4), ECMAScript imports and exports written in input files are preserved in the output, and CommonJS-style `import x = require("...")` and `export = ...` statements are emitted as CommonJS `require` and `module.exports`. In other words, the format of each individual import or export statement is preserved, rather than being coerced into a single format for the whole compilation (or even a whole file).

```ts
import { valueOfPi } from "./constants";

const constants = require("./constants");

export const piSquared = valueOfPi * constants.valueOfPi;
```

Try

While it's rare to need to mix imports and require calls in the same file, this `module` mode best reflects the capabilities of most modern bundlers, as well as the Bun runtime.

> Why care about TypeScript's `module` emit with a bundler or with Bun, where you're likely also setting `noEmit`? TypeScript's type checking and module resolution behavior are affected by the module format that it *would* emit. Setting `module` gives TypeScript information about how your bundler or runtime will process imports and exports, which ensures that the types you see on imported values accurately reflect what will happen at runtime or after bundling.

None

```ts
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
exports.twoPi = void 0;
const constants_1 = require("./constants");
exports.twoPi = constants_1.valueOfPi * 2;
```

Try

Default:

`CommonJS` if `target` is `ES3` or `ES5`; `ES6`/`ES2015` otherwise.

Allowed:

- `none`
- `commonjs`
- `amd`
- `umd`
- `system`
- `es6/es2015`
- `es2020`
- `es2022`
- `esnext`
- `node16`
- `nodenext`
- `preserve`

Related:

moduleResolution

esModuleInterop

allowImportingTsExtensions

allowArbitraryExtensions

resolveJsonModule

Released:

1.0

Module Resolution - `moduleResolution`

Specify the module resolution strategy:

- `'node16'` or `'nodenext'` for modern versions of Node.js. Node.js v12 and later supports both ECMAScript imports and CommonJS `require`, which resolve using different algorithms. These `moduleResolution` values, when combined with the corresponding `module` values, picks the right algorithm for each resolution based on whether Node.js will see an `import` or `require` in the output JavaScript code.
- `'node10'` (previously called `'node'`) for Node.js versions older than v10, which only support CommonJS `require`. You probably won't need to use `node10` in modern code.
- `'bundler'` for use with bundlers. Like `node16` and `nodenext`, this mode supports package.json `"imports"` and `"exports"`, but unlike the Node.js resolution modes, `bundler` never requires file extensions on relative paths in imports.
- `'classic'` was used in TypeScript before the release of 1.6. `classic` should not be used.

There are reference pages explaining the theory behind TypeScript's module resolution and the details of each option.

Default:

`Classic` if `module` is `AMD`, `UMD`, `System`, or `ES6`/`ES2015`; Matches if `module` is `node16` or `nodenext`; `Node` otherwise.

Allowed:

`classic`

`node10`/`node`

`node16`

`nodenext`

`bundler`

Related:

module

paths

baseUrl

rootDirs

moduleSuffixes

customConditions

resolvePackageJsonExports

resolvePackageJsonImports

Module Suffixes - `moduleSuffixes`

Provides a way to override the default list of file name suffixes to search when resolving a module.

```
{
  "compilerOptions": {
```

```
    "moduleSuffixes": [".ios", ".native", ""]
  }
}
```

Given the above configuration, an import like the following:

```ts
import * as foo from "./foo";
```

TypeScript will look for the relative files `./foo.ios.ts`, `./foo.native.ts`, and finally `./foo.ts`.

Note the empty string `""` in `moduleSuffixes` which is necessary for TypeScript to also look-up `./foo.ts`.

This feature can be useful for React Native projects where each target platform can use a separate tsconfig.json with differing `moduleSuffixes`.

Released:

4.7

No Resolve - `noResolve`

By default, TypeScript will examine the initial set of files for `import` and `<reference` directives and add these resolved files to your program.

If `noResolve` is set, this process doesn't happen. However, `import` statements are still checked to see if they resolve to a valid module, so you'll need to make sure this is satisfied by some other means.

Paths - `paths`

A series of entries which re-map imports to lookup locations relative to the `baseUrl` if set, or to the tsconfig file itself otherwise. There is a larger coverage of `paths` in the `moduleResolution` reference page.

`paths` lets you declare how TypeScript should resolve an import in your `require`/`imports`.

```
{
  "compilerOptions": {
    "paths": {
      "jquery": ["./vendor/jquery/dist/jquery"]
    }
  }
}
```

This would allow you to be able to write `import "jquery"`, and get all of the correct typing locally.

```
{
  "compilerOptions": {
    "paths": {
        "app/*": ["./src/app/*"],
        "config/*": ["./src/app/_config/*"],
        "environment/*": ["./src/environments/*"],
        "shared/*": ["./src/app/_shared/*"],
        "helpers/*": ["./src/helpers/*"],
        "tests/*": ["./src/tests/*"]
    },
}
```

In this case, you can tell the TypeScript file resolver to support a number of custom prefixes to find code.

Note that this feature does not change how import paths are emitted by `tsc`, so `paths` should only be used to inform TypeScript that another tool has this mapping and will use it at runtime or when bundling.

Resolve JSON Module - `resolveJsonModule`

Allows importing modules with a `.json` extension, which is a common practice in node projects. This includes generating a type for the `import` based on the static JSON shape.

TypeScript does not support resolving JSON files by default:

```ts
// @filename: settings.json
{
    "repo": "TypeScript",
    "dry": false,
    "debug": false
}
// @filename: index.ts
import settings from "./settings.json";
Cannot find module './settings.json'. Consider using '--resolveJsonModule' to import module with '.json' extension.2732Cannot
settings.debug === true;
settings.dry === 2;
```

Try

Enabling the option allows importing JSON, and validating the types in that JSON file.

```ts
// @filename: settings.json
{
    "repo": "TypeScript",
    "dry": false,
    "debug": false
}
// @filename: index.ts
import settings from "./settings.json";
settings.debug === true;
settings.dry === 2;
This comparison appears to be unintentional because the types 'boolean' and 'number' have no overlap.2367This comparison appea
```

Resolve package.json Exports - `resolvePackageJsonExports`

`--resolvePackageJsonExports` forces TypeScript to consult the `exports` field of `package.json` files if it ever reads from a package in `node_modules`.

This option defaults to `true` under the `node16`, `nodenext`, and `bundler` options for `--moduleResolution`.

Default:

`true` when `moduleResolution` is `node16`, `nodenext`, or `bundler`; otherwise `false`

Related:

moduleResolution

customConditions

resolvePackageJsonImports

Resolve package.json Imports - `resolvePackageJsonImports`

`--resolvePackageJsonImports` forces TypeScript to consult the `imports` field of `package.json` files when performing a lookup that starts with # from a file whose ancestor directory contains a `package.json`.

This option defaults to `true` under the `node16`, `nodenext`, and `bundler` options for `--moduleResolution`.

Default:

`true` when `moduleResolution` is `node16`, `nodenext`, or `bundler`; otherwise `false`

Related:

moduleResolution

customConditions

resolvePackageJsonExports

Root Dir - `rootDir`

Default: The longest common path of all non-declaration input files. If `composite` is set, the default is instead the directory containing the `tsconfig.json` file.

When TypeScript compiles files, it keeps the same directory structure in the output directory as exists in the input directory.

For example, let's say you have some input files:

```
MyProj
■■■ tsconfig.json
■■■ core
■   ■■■ a.ts
■   ■■■ b.ts
■   ■■■ sub
■   ■   ■■■ c.ts
■■■ types.d.ts
```

The inferred value for `rootDir` is the longest common path of all non-declaration input files, which in this case is `core/`.

If your `outDir` was `dist`, TypeScript would write this tree:

```
MyProj
■■■ dist
■   ■■■ a.js
■   ■■■ b.js
■   ■■■ sub
■   ■   ■■■ c.js
```

However, you may have intended for `core` to be part of the output directory structure. By setting `rootDir: "."` in `tsconfig.json`, TypeScript would write this tree:

```
MyProj
■■■ dist
```

```
■   ■■■ core
■   ■   ■■■ a.js
■   ■   ■■■ b.js
■   ■   ■■■ sub
■   ■   ■   ■■■ c.js
```

Importantly, `rootDir` **does not affect which files become part of the compilation**. It has no interaction with the `include`, `exclude`, or `files` `tsconfig.json` settings.

Note that TypeScript will never write an output file to a directory outside of `outDir`, and will never skip emitting a file. For this reason, `rootDir` also enforces that all files which need to be emitted are underneath the `rootDir` path.

For example, let's say you had this tree:

```
MyProj
■■■ tsconfig.json
■■■ core
■   ■■■ a.ts
■   ■■■ b.ts
■■■ helpers.ts
```

It would be an error to specify `rootDir` as `core` *and* `include` as `*` because it creates a file (`helpers.ts`) that would need to be emitted *outside* the `outDir` (i.e. `../helpers.js`).

Default:

Computed from the list of input files.

Released:

1.5

Root Dirs - `rootDirs`

Using `rootDirs`, you can inform the compiler that there are many "virtual" directories acting as a single root. This allows the compiler to resolve relative module imports within these "virtual" directories, as if they were merged in to one directory.

For example:

```
 src
 ■■■ views
     ■■■ view1.ts (can import "./template1", "./view2`)
     ■■■ view2.ts (can import "./template1", "./view1`)
 generated
 ■■■ templates
         ■■■ views
             ■■■ template1.ts (can import "./view1", "./view2")
```

```
{
  "compilerOptions": {
    "rootDirs": ["src/views", "generated/templates/views"]
  }
}
```

This does not affect how TypeScript emits JavaScript, it only emulates the assumption that they will be able to work via those relative paths at runtime.

`rootDirs` can be used to provide a separate “type layer” to files that are not TypeScript or JavaScript by providing a home for generated `.d.ts` files in another folder. This technique is useful for bundled applications where you use `import` of files that aren’t necessarily code:

```
sh

 src

 ■■■ index.ts

 ■■■ css

     ■■■ main.css

     ■■■ navigation.css

 generated

 ■■■ css

     ■■■ main.css.d.ts

     ■■■ navigation.css.d.ts
```

```
{

  "compilerOptions": {

    "rootDirs": ["src", "generated"]

  }

}
```

This technique lets you generate types ahead of time for the non-code source files. Imports then work naturally based off the source file’s location. For example `./src/index.ts` can import the file `./src/css/main.css` and TypeScript will be aware of the bundler’s behavior for that filetype via the corresponding generated declaration file.

```
ts

// @filename: index.ts

import { appClass } from "./main.css";
```

Try

Default:

Computed from the list of input files.

Released:

2.0

Type Roots - `typeRoots`

By default all *visible* ”`@types`” packages are included in your compilation. Packages in `node_modules/@types` of any enclosing folder are considered *visible*. For example, that means packages within `./node_modules/@types/`, `../node_modules/@types/`, `../../node_modules/@types/`, and so on.

If `typeRoots` is specified, *only* packages under `typeRoots` will be included. For example:

```
{

  "compilerOptions": {

    "typeRoots": ["./typings", "./vendor/types"]

  }

}
```

This config file will include *all* packages under `./typings` and `./vendor/types`, and no packages from `./node_modules/@types`. All paths are relative to the `tsconfig.json`.

Related:

types

Types - types

By default all *visible* "@types" packages are included in your compilation. Packages in node_modules/@types of any enclosing folder are considered *visible*. For example, that means packages within ./node_modules/@types/, ../node_modules/@types/, ../../node_modules/@types/, and so on.

If types is specified, only packages listed will be included in the global scope. For instance:

```
{

  "compilerOptions": {

    "types": ["node", "jest", "express"]

  }

}
```

This tsconfig.json file will *only* include ./node_modules/@types/node, ./node_modules/@types/jest and ./node_modules/@types/express. Other packages under node_modules/@types/* will not be included.

What does this affect?

This option does not affect how @types/* are included in your application code, for example if you had the above compilerOptions example with code like:

ts

```
import * as moment from "moment";

moment().format("MMMM Do YYYY, h:mm:ss a");
```

The moment import would be fully typed.

When you have this option set, by not including a module in the types array it:

- Will not add globals to your project (e.g process in node, or expect in Jest)
- Will not have exports appear as auto-import recommendations

This feature differs from typeRoots in that it is about specifying only the exact types you want included, whereas typeRoots supports saying you want particular folders.

Related:

typeRoots

#Emit

Declaration - declaration

Generate .d.ts files for every TypeScript or JavaScript file inside your project. These .d.ts files are type definition files which describe the external API of your module. With .d.ts files, tools like TypeScript can provide intellisense and accurate types for un-typed code.

When declaration is set to true, running the compiler with this TypeScript code:

ts

```
export let helloWorld = "hi";
```

Try

Will generate an index.js file like this:

ts

```
export let helloWorld = "hi";
```

Try

With a corresponding helloWorld.d.ts:

ts

```ts
export declare let helloWorld: string;
```

Try

When working with `.d.ts` files for JavaScript files you may want to use `emitDeclarationOnly` or use `outDir` to ensure that the JavaScript files are not overwritten.

Default:

`true` if `composite`; `false` otherwise.

Related:

`declarationDir`

`emitDeclarationOnly`

Released:

1.0

Declaration Dir - `declarationDir`

Offers a way to configure the root directory for where declaration files are emitted.

example

```
■■■ index.ts
■■■ package.json
■■■ tsconfig.json
```

with this `tsconfig.json`:

```
{
  "compilerOptions": {
    "declaration": true,
    "declarationDir": "./types"
  }
}
```

Would place the d.ts for the `index.ts` in a `types` folder:

example

```
■■■ index.js
■■■ index.ts
■■■ package.json
■■■ tsconfig.json
■■■ types
    ■■■ index.d.ts
```

Related:

`declaration`

Released:

2.0

Declaration Map - `declarationMap`

Generates a source map for `.d.ts` files which map back to the original `.ts` source file. This will allow editors such as VS Code to go to the original `.ts` file when using features like *Go to Definition*.

You should strongly consider turning this on if you're using project references.

Released:

2.9

Downlevel Iteration - `downlevelIteration`

Downleveling is TypeScript's term for transpiling to an older version of JavaScript. This flag is to enable support for a more accurate implementation of how modern JavaScript iterates through new concepts in older JavaScript runtimes.

ECMAScript 6 added several new iteration primitives: the `for / of` loop (`for (el of arr)`), Array spread (`[a, ...b]`), argument spread (`fn(...args)`), and `Symbol.iterator`. `downlevelIteration` allows for these iteration primitives to be used more accurately in ES5 environments if a `Symbol.iterator` implementation is present.

Example: Effects on `for / of`

With this TypeScript code:

```ts
const str = "Hello!";
for (const s of str) {
  console.log(s);
}
```

Try

Without `downlevelIteration` enabled, a `for / of` loop on any object is downleveled to a traditional `for` loop:

```ts
"use strict";
var str = "Hello!";
for (var _i = 0, str_1 = str; _i < str_1.length; _i++) {
    var s = str_1[_i];
    console.log(s);
}
```

Try

This is often what people expect, but it's not 100% compliant with ECMAScript iteration protocol. Certain strings, such as emoji (■), have a `.length` of 2 (or even more!), but should iterate as 1 unit in a `for-of` loop. See this blog post by Jonathan New for a longer explanation.

When `downlevelIteration` is enabled, TypeScript will use a helper function that checks for a `Symbol.iterator` implementation (either native or polyfill). If this implementation is missing, you'll fall back to index-based iteration.

```ts
"use strict";
var __values = (this && this.__values) || function(o) {
    var s = typeof Symbol === "function" && Symbol.iterator, m = s && o[s], i = 0;
    if (m) return m.call(o);
    if (o && typeof o.length === "number") return {
        next: function () {
```

```
            if (o && i >= o.length) o = void 0;
            return { value: o && o[i++], done: !o };
        }
    };
    throw new TypeError(s ? "Object is not iterable." : "Symbol.iterator is not defined.");
};
var e_1, _a;
var str = "Hello!";
try {
    for (var str_1 = __values(str), str_1_1 = str_1.next(); !str_1_1.done; str_1_1 = str_1.next()) {
        var s = str_1_1.value;
        console.log(s);
    }
}
catch (e_1_1) { e_1 = { error: e_1_1 }; }
finally {
    try {
        if (str_1_1 && !str_1_1.done && (_a = str_1.return)) _a.call(str_1);
    }
    finally { if (e_1) throw e_1.error; }
}
```

Try

You can use tslib via `importHelpers` to reduce the amount of inline JavaScript too:

ts

```
"use strict";
var __values = (this && this.__values) || function(o) {
    var s = typeof Symbol === "function" && Symbol.iterator, m = s && o[s], i = 0;
    if (m) return m.call(o);
    if (o && typeof o.length === "number") return {
        next: function () {
            if (o && i >= o.length) o = void 0;
            return { value: o && o[i++], done: !o };
        }
    };
    throw new TypeError(s ? "Object is not iterable." : "Symbol.iterator is not defined.");
};
var e_1, _a;
var str = "Hello!";
```

```
try {
    for (var str_1 = __values(str), str_1_1 = str_1.next(); !str_1_1.done; str_1_1 = str_1.next()) {
        var s = str_1_1.value;
        console.log(s);
    }
}
catch (e_1_1) { e_1 = { error: e_1_1 }; }
finally {
    try {
        if (str_1_1 && !str_1_1.done && (_a = str_1.return)) _a.call(str_1);
    }
    finally { if (e_1) throw e_1.error; }
}
```

Try

Note: enabling `downlevelIteration` does not improve compliance if `Symbol.iterator` is not present in the runtime.

Example: Effects on Array Spreads

This is an array spread:

```
js
// Make a new array whose elements are 1 followed by the elements of arr2
const arr = [1, ...arr2];
```

Based on the description, it sounds easy to downlevel to ES5:

```
js
// The same, right?
const arr = [1].concat(arr2);
```

However, this is observably different in certain rare cases.

For example, if a source array is missing one or more items (contains a hole), the spread syntax will replace each empty item with `undefined`, whereas `.concat` will leave them intact.

```
js
// Make an array where the element at index 1 is missing
let arrayWithHole = ["a", , "c"];
let spread = [...arrayWithHole];
let concatenated = [].concat(arrayWithHole);
console.log(arrayWithHole);
// [ 'a', <1 empty item>, 'c' ]
console.log(spread);
// [ 'a', undefined, 'c' ]
console.log(concatenated);
// [ 'a', <1 empty item>, 'c' ]
```

Just as with `for / of`, `downlevelIteration` will use `Symbol.iterator` (if present) to more accurately emulate ES 6 behavior.

Related:

`importHelpers`

Released:

2.3

Emit BOM - `emitBOM`

Controls whether TypeScript will emit a byte order mark (BOM) when writing output files. Some runtime environments require a BOM to correctly interpret a JavaScript files; others require that it is not present. The default value of `false` is generally best unless you have a reason to change it.

Emit Declaration Only - `emitDeclarationOnly`

Only emit `.d.ts` files; do not emit `.js` files.

This setting is useful in two cases:

- You are using a transpiler other than TypeScript to generate your JavaScript.
- You are using TypeScript to only generate `d.ts` files for your consumers.

Related:

`declaration`

Released:

2.8

Import Helpers - `importHelpers`

For certain downleveling operations, TypeScript uses some helper code for operations like extending class, spreading arrays or objects, and async operations. By default, these helpers are inserted into files which use them. This can result in code duplication if the same helper is used in many different modules.

If the `importHelpers` flag is on, these helper functions are instead imported from the tslib module. You will need to ensure that the `tslib` module is able to be imported at runtime. This only affects modules; global script files will not attempt to import modules.

For example, with this TypeScript:

ts

```ts
export function fn(arr: number[]) {
  const arr2 = [1, ...arr];
}
```

Turning on `downlevelIteration` and `importHelpers` is still false:

ts

```ts
var __read = (this && this.__read) || function (o, n) {
    var m = typeof Symbol === "function" && o[Symbol.iterator];
    if (!m) return o;
    var i = m.call(o), r, ar = [], e;
    try {
        while ((n === void 0 || n-- > 0) && !(r = i.next()).done) ar.push(r.value);
    }
    catch (error) { e = { error: error }; }
    finally {
```

```
        try {
            if (r && !r.done && (m = i["return"])) m.call(i);
        }
        finally { if (e) throw e.error; }
    }
    return ar;
};
var __spreadArray = (this && this.__spreadArray) || function (to, from, pack) {
    if (pack || arguments.length === 2) for (var i = 0, l = from.length, ar; i < l; i++) {
        if (ar || !(i in from)) {
            if (!ar) ar = Array.prototype.slice.call(from, 0, i);
            ar[i] = from[i];
        }
    }
    return to.concat(ar || Array.prototype.slice.call(from));
};
export function fn(arr) {
    var arr2 = __spreadArray([1], __read(arr), false);
}
```

Try

Then turning on both downlevelIteration and `importHelpers`:

```
ts
import { __read, __spreadArray } from "tslib";
export function fn(arr) {
    var arr2 = __spreadArray([1], __read(arr), false);
}
```

Try

You can use noEmitHelpers when you provide your own implementations of these functions.

Related:

- noEmitHelpers
- downlevelIteration

Imports Not Used As Values - `importsNotUsedAsValues`

Deprecated in favor of verbatimModuleSyntax.

This flag controls how `import` works, there are 3 different options:

`remove`: The default behavior of dropping `import` statements which only reference types.

`preserve`: Preserves all `import` statements whose values or types 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.

This flag works because you can use `import type` to explicitly create an `import` statement which should never be emitted into JavaScript.

Default:

`remove`

Allowed:

- `remove`
- `preserve`
- `error`

Related:

- `preserveValueImports`
- `verbatimModuleSyntax`

Released:

3.8

Inline Source Map - `inlineSourceMap`

When set, instead of writing out a `.js.map` file to provide source maps, TypeScript will embed the source map content in the `.js` files. Although this results in larger JS files, it can be convenient in some scenarios. For example, you might want to debug JS files on a webserver that doesn't allow `.map` files to be served.

Mutually exclusive with `sourceMap`.

For example, with this TypeScript:

ts

```ts
const helloWorld = "hi";

console.log(helloWorld);
```

Converts to this JavaScript:

ts

```ts
"use strict";

const helloWorld = "hi";

console.log(helloWorld);
```

Try

Then enable building it with `inlineSourceMap` enabled there is a comment at the bottom of the file which includes a source-map for the file.

ts

```ts
"use strict";

const helloWorld = "hi";

console.log(helloWorld);

//# sourceMappingURL=data:application/json;base64,eyJ2ZXJzaW9uIjozLCJmaWxlIjoiaW5kZXguanMiLCJzb3VyY2VSb290IjoiIiwic291cmNlcyI6
```

Try

Released:

1.5

Inline Sources - `inlineSources`

When set, TypeScript will include the original content of the `.ts` file as an embedded string in the source map (using the source map's `sourcesContent` property). This is often useful in the same cases as `inlineSourceMap`.

Requires either sourceMap or inlineSourceMap to be set.

For example, with this TypeScript:

ts

```ts
const helloWorld = "hi";

console.log(helloWorld);
```

Try

By default converts to this JavaScript:

ts

```ts
"use strict";

const helloWorld = "hi";

console.log(helloWorld);
```

Try

Then enable building it with `inlineSources` and inlineSourceMap enabled there is a comment at the bottom of the file which includes a source-map for the file. Note that the end is different from the example in inlineSourceMap because the source-map now contains the original source code also.

ts

```ts
"use strict";

const helloWorld = "hi";

console.log(helloWorld);

//# sourceMappingURL=data:application/json;base64,eyJ2ZXJzaW9uIjozLCJmaWxlIjoiaW5kZXguanMiLCJzb3VyY2VSb290IjoiIiwic291cmNlcyI6
```

Try

Released:

1.5

Map Root - `mapRoot`

Specify the location where debugger should locate map files instead of generated locations. This string is treated verbatim inside the source-map, for example:

```
{
  "compilerOptions": {
    "sourceMap": true,
    "mapRoot": "https://my-website.com/debug/sourcemaps/"
  }
}
```

Would declare that `index.js` will have sourcemaps at `https://my-website.com/debug/sourcemaps/index.js.map`.

New Line - `newLine`

Specify the end of line sequence to be used when emitting files: 'CRLF' (dos) or 'LF' (unix).

Default:

`lf`

Allowed:

`crlf`

lf

Released:

1.5

No Emit - `noEmit`

Do not emit compiler output files like JavaScript source code, source-maps or declarations.

This makes room for another tool like Babel, or swc to handle converting the TypeScript file to a file which can run inside a JavaScript environment.

You can then use TypeScript as a tool for providing editor integration, and as a source code type-checker.

No Emit Helpers - `noEmitHelpers`

Instead of importing helpers with `importHelpers`, you can provide implementations in the global scope for the helpers you use and completely turn off emitting of helper functions.

For example, using this `async` function in ES5 requires a `await`-like function and `generator`-like function to run:

ts

```ts
const getAPI = async (url: string) => {
  // Get API
  return {};
};
```

Try

Which creates quite a lot of JavaScript:

ts

```ts
"use strict";
var __awaiter = (this && this.__awaiter) || function (thisArg, _arguments, P, generator) {
    function adopt(value) { return value instanceof P ? value : new P(function (resolve) { resolve(value); }); }
    return new (P || (P = Promise))(function (resolve, reject) {
        function fulfilled(value) { try { step(generator.next(value)); } catch (e) { reject(e); } }
        function rejected(value) { try { step(generator["throw"](value)); } catch (e) { reject(e); } }
        function step(result) { result.done ? resolve(result.value) : adopt(result.value).then(fulfilled, rejected); }
        step((generator = generator.apply(thisArg, _arguments || [])).next());
    });
};
var __generator = (this && this.__generator) || function (thisArg, body) {
    var _ = { label: 0, sent: function() { if (t[0] & 1) throw t[1]; return t[1]; }, trys: [], ops: [] }, f, y, t, g;
    return g = { next: verb(0), "throw": verb(1), "return": verb(2) }, typeof Symbol === "function" && (g[Symbol.iterator] = f
    function verb(n) { return function (v) { return step([n, v]); }; }
    function step(op) {
        if (f) throw new TypeError("Generator is already executing.");
        while (g && (g = 0, op[0] && (_ = 0)), _) try {
            if (f = 1, y && (t = op[0] & 2 ? y["return"] : op[0] ? y["throw"] || ((t = y["return"]) && t.call(y), 0) : y.next)
```

```
            if (y = 0, t) op = [op[0] & 2, t.value];
            switch (op[0]) {
                case 0: case 1: t = op; break;
                case 4: _.label++; return { value: op[1], done: false };
                case 5: _.label++; y = op[1]; op = [0]; continue;
                case 7: op = _.ops.pop(); _.trys.pop(); continue;
                default:
                    if (!(t = _.trys, t = t.length > 0 && t[t.length - 1]) && (op[0] === 6 || op[0] === 2)) { _ = 0; continue;
                    if (op[0] === 3 && (!t || (op[1] > t[0] && op[1] < t[3]))) { _.label = op[1]; break; }
                    if (op[0] === 6 && _.label < t[1]) { _.label = t[1]; t = op; break; }
                    if (t && _.label < t[2]) { _.label = t[2]; _.ops.push(op); break; }
                    if (t[2]) _.ops.pop();
                    _.trys.pop(); continue;
            }
            op = body.call(thisArg, _);
        } catch (e) { op = [6, e]; y = 0; } finally { f = t = 0; }
        if (op[0] & 5) throw op[1]; return { value: op[0] ? op[1] : void 0, done: true };
    }
};
var getAPI = function (url) { return __awaiter(void 0, void 0, void 0, function () {
    return __generator(this, function (_a) {
        // Get API
        return [2 /*return*/, {}];
    });
}); };
```

[Try]

Which can be switched out with your own globals via this flag:

ts

```
"use strict";
var getAPI = function (url) { return __awaiter(void 0, void 0, void 0, function () {
    return __generator(this, function (_a) {
        // Get API
        return [2 /*return*/, {}];
    });
}); };
```

[Try]

Related:

[importHelpers]

Released:

1.5

No Emit On Error - `noEmitOnError`

Do not emit compiler output files like JavaScript source code, source-maps or declarations if any errors were reported.

This defaults to `false`, making it easier to work with TypeScript in a watch-like environment where you may want to see results of changes to your code in another environment before making sure all errors are resolved.

Released:

1.4

Out Dir - `outDir`

If specified, `.js` (as well as `.d.ts`, `.js.map`, etc.) files will be emitted into this directory. The directory structure of the original source files is preserved; see `rootDir` if the computed root is not what you intended.

If not specified, `.js` files will be emitted in the same directory as the `.ts` files they were generated from:

```
sh

$ tsc

example

■■■ index.js

■■■ index.ts
```

With a `tsconfig.json` like this:

```
{

  "compilerOptions": {

    "outDir": "dist"

  }

}
```

Running `tsc` with these settings moves the files into the specified `dist` folder:

```
sh

$ tsc

example

■■■ dist

■   ■■■ index.js

■■■ index.ts

■■■ tsconfig.json
```

Related:

- `out`
- `outFile`

Out File - `outFile`

If specified, all *global* (non-module) files will be concatenated into the single output file specified.

If `module` is `system` or `amd`, all module files will also be concatenated into this file after all global content.

Note: `outFile` cannot be used unless `module` is `None`, `System`, or `AMD`. This option *cannot* be used to bundle CommonJS or ES6 modules.

Related:

- `out`
- `outDir`

Released:

1.0

Preserve Const Enums - `preserveConstEnums`

Do not erase `const enum` declarations in generated code. `const enums` provide a way to reduce the overall memory footprint of your application at runtime by emitting the enum value instead of a reference.

For example with this TypeScript:

ts

```ts
const enum Album {
  JimmyEatWorldFutures = 1,
  TubRingZooHypothesis = 2,
  DogFashionDiscoAdultery = 3,
}
const selectedAlbum = Album.JimmyEatWorldFutures;
if (selectedAlbum === Album.JimmyEatWorldFutures) {
  console.log("That is a great choice.");
}
```

Try

The default `const enum` behavior is to convert any `Album.Something` to the corresponding number literal, and to remove a reference to the enum from the JavaScript completely.

ts

```ts
"use strict";
const selectedAlbum = 1 /* Album.JimmyEatWorldFutures */;
if (selectedAlbum === 1 /* Album.JimmyEatWorldFutures */) {
    console.log("That is a great choice.");
}
```

Try

With `preserveConstEnums` set to `true`, the `enum` exists at runtime and the numbers are still emitted.

ts

```ts
"use strict";
var Album;
(function (Album) {
    Album[Album["JimmyEatWorldFutures"] = 1] = "JimmyEatWorldFutures";
    Album[Album["TubRingZooHypothesis"] = 2] = "TubRingZooHypothesis";
    Album[Album["DogFashionDiscoAdultery"] = 3] = "DogFashionDiscoAdultery";
```

```
})(Album || (Album = {}));

const selectedAlbum = 1 /* Album.JimmyEatWorldFutures */;

if (selectedAlbum === 1 /* Album.JimmyEatWorldFutures */) {

    console.log("That is a great choice.");

}
```

Try

This essentially makes such `const enums` a source-code feature only, with no runtime traces.

Default:

`true` if `isolatedModules`; `false` otherwise.

Preserve Value Imports - `preserveValueImports`

Deprecated in favor of `verbatimModuleSyntax`.

There are some cases where TypeScript can't detect that you're using an import. For example, take the following code:

ts

```
import { Animal } from "./animal.js";

eval("console.log(new Animal().isDangerous())");
```

or code using 'Compiles to HTML' languages like Svelte or Vue. `preserveValueImports` will prevent TypeScript from removing the import, even if it appears unused.

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.

Related:

`isolatedModules`

`importsNotUsedAsValues`

`verbatimModuleSyntax`

Released:

4.5

Remove Comments - `removeComments`

Strips all comments from TypeScript files when converting into JavaScript. Defaults to `false`.

For example, this is a TypeScript file which has a JSDoc comment:

ts

```
/** The translation of 'Hello world' into Portuguese */

export const helloWorldPTBR = "Olá Mundo";
```

When `removeComments` is set to `true`:

ts

```
export const helloWorldPTBR = "Olá Mundo";
```

Try

Without setting `removeComments` or having it as `false`:

ts

```
/** The translation of 'Hello world' into Portuguese */
```

```ts
export const helloWorldPTBR = "Olá Mundo";
```

Try

This means that your comments will show up in the JavaScript code.

Source Map - `sourceMap`

Enables the generation of sourcemap files. These files allow debuggers and other tools to display the original TypeScript source code when actually working with the emitted JavaScript files. Source map files are emitted as `.js.map` (or `.jsx.map`) files next to the corresponding `.js` output file.

The `.js` files will in turn contain a sourcemap comment to indicate where the files are to external tools, for example:

```ts
// helloWorld.ts

export declare const helloWorld = "hi";
```

Compiling with `sourceMap` set to `true` creates the following JavaScript file:

```js
// helloWorld.js

"use strict";

Object.defineProperty(exports, "__esModule", { value: true });

exports.helloWorld = "hi";

//# sourceMappingURL=// helloWorld.js.map
```

And this also generates this json map:

```json
// helloWorld.js.map

{

  "version": 3,

  "file": "ex.js",

  "sourceRoot": "",

  "sources": ["../ex.ts"],

  "names": [],

  "mappings": ";;AAAa,QAAA,UAAU,GAAG,IAAI,CAAA"

}
```

Source Root - `sourceRoot`

Specify the location where a debugger should locate TypeScript files instead of relative source locations. This string is treated verbatim inside the source-map where you can use a path or a URL:

```
{

  "compilerOptions": {

    "sourceMap": true,

    "sourceRoot": "https://my-website.com/debug/source/"

  }

}
```

Would declare that `index.js` will have a source file at `https://my-website.com/debug/source/index.ts`.

Strip Internal - `stripInternal`

Do not emit declarations for code that has an `@internal` annotation in its JSDoc comment. This is an internal compiler option; use at your own risk, because the compiler does not check that the result is valid. If you are searching for a tool to handle additional levels of visibility within your `d.ts` files, look at api-extractor.

ts

```ts
/**
 * Days available in a week
 * @internal
 */
export const daysInAWeek = 7;
/** Calculate how much someone earns in a week */
export function weeklySalary(dayRate: number) {
  return daysInAWeek * dayRate;
}
```

Try

With the flag set to `false` (default):

ts

```ts
/**
 * Days available in a week
 * @internal
 */
export declare const daysInAWeek = 7;
/** Calculate how much someone earns in a week */
export declare function weeklySalary(dayRate: number): number;
```

Try

With `stripInternal` set to `true` the `d.ts` emitted will be redacted.

ts

```ts
/** Calculate how much someone earns in a week */
export declare function weeklySalary(dayRate: number): number;
```

Try

The JavaScript output is still the same.

- Internal

#JavaScript Support

Allow JS - `allowJs`

Allow JavaScript files to be imported inside your project, instead of just `.ts` and `.tsx` files. For example, this JS file:

js

```js
// @filename: card.js
export const defaultCardDeck = "Heart";
```

Try

When imported into a TypeScript file will raise an error:

```ts
// @filename: index.ts

import { defaultCardDeck } from "./card";

console.log(defaultCardDeck);
```

Try

Imports fine with `allowJs` enabled:

```ts
// @filename: index.ts

import { defaultCardDeck } from "./card";

console.log(defaultCardDeck);
```

Try

This flag can be used as a way to incrementally add TypeScript files into JS projects by allowing the `.ts` and `.tsx` files to live along-side existing JavaScript files.

It can also be used along-side `declaration` and `emitDeclarationOnly` to create declarations for JS files.

Related:

`checkJs`

`emitDeclarationOnly`

Released:

1.8

Check JS - `checkJs`

Works in tandem with `allowJs`. When `checkJs` is enabled then errors are reported in JavaScript files. This is the equivalent of including `// @ts-check` at the top of all JavaScript files which are included in your project.

For example, this is incorrect JavaScript according to the `parseFloat` type definition which comes with TypeScript:

```js
// parseFloat only takes a string

module.exports.pi = parseFloat(3.142);
```

When imported into a TypeScript module:

```ts
// @filename: constants.js

module.exports.pi = parseFloat(3.142);

// @filename: index.ts

import { pi } from "./constants";

console.log(pi);
```

Try

You will not get any errors. However, if you turn on `checkJs` then you will get error messages from the JavaScript file.

```ts
```

```
// @filename: constants.js

module.exports.pi = parseFloat(3.142);

Argument of type 'number' is not assignable to parameter of type 'string'.2345Argument of type 'number' is not assignable to p

// @filename: index.ts

import { pi } from "./constants";

console.log(pi);
```

Try

Related:

allowJs

emitDeclarationOnly

Released:

2.3

Max Node Module JS Depth - `maxNodeModuleJsDepth`

The maximum dependency depth to search under `node_modules` and load JavaScript files.

This flag can only be used when `allowJs` is enabled, and is used if you want to have TypeScript infer types for all of the JavaScript inside your `node_modules`.

Ideally this should stay at 0 (the default), and `d.ts` files should be used to explicitly define the shape of modules. However, there are cases where you may want to turn this on at the expense of speed and potential accuracy.

#Editor Support

Disable Size Limit - `disableSizeLimit`

To avoid a possible memory bloat issues when working with very large JavaScript projects, there is an upper limit to the amount of memory TypeScript will allocate. Turning this flag on will remove the limit.

Plugins - `plugins`

List of language service plugins to run inside the editor.

Language service plugins are a way to provide additional information to a user based on existing TypeScript files. They can enhance existing messages between TypeScript and an editor, or to provide their own error messages.

For example:

- ts-sql-plugin — Adds SQL linting with a template strings SQL builder.
- typescript-styled-plugin — Provides CSS linting inside template strings .
- typescript-eslint-language-service — Provides eslint error messaging and fix-its inside the compiler's output.
- ts-graphql-plugin — Provides validation and auto-completion inside GraphQL query template strings.

VS Code has the ability for a extension to automatically include language service plugins, and so you may have some running in your editor without needing to define them in your `tsconfig.json`.

#Interop Constraints

Allow Synthetic Default Imports - `allowSyntheticDefaultImports`

When set to true, `allowSyntheticDefaultImports` allows you to write an import like:

`ts`

```
import React from "react";
```

instead of:

ts

```ts
import * as React from "react";
```

When the module **does not** explicitly specify a default export.

For example, without `allowSyntheticDefaultImports` as true:

ts

```ts
// @filename: utilFunctions.js
const getStringLength = (str) => str.length;
module.exports = {
  getStringLength,
};
// @filename: index.ts
import utils from "./utilFunctions";
Module '"/home/runner/work/TypeScript-Website/TypeScript-Website/packages/typescriptlang-org/utilFunctions"' has no default ex
const count = utils.getStringLength("Check JS");
```

Try

This code raises an error because there isn't a `default` object which you can import. Even though it feels like it should. For convenience, transpilers like Babel will automatically create a default if one isn't created. Making the module look a bit more like:

js

```js
// @filename: utilFunctions.js
const getStringLength = (str) => str.length;
const allFunctions = {
  getStringLength,
};
module.exports = allFunctions;
module.exports.default = allFunctions;
```

This flag does not affect the JavaScript emitted by TypeScript, it's only for the type checking. This option brings the behavior of TypeScript in-line with Babel, where extra code is emitted to make using a default export of a module more ergonomic.

Default:

`true` if `esModuleInterop` is enabled, `module` is `system`, or `moduleResolution` is `bundler`; `false` otherwise.

Related:

`esModuleInterop`

Released:

1.8

ES Module Interop - `esModuleInterop`

By default (with `esModuleInterop` false or not set) TypeScript treats CommonJS/AMD/UMD modules similar to ES6 modules. In doing this, there are two parts in particular which turned out to be flawed assumptions:

a namespace import like `import * as moment from "moment"` acts the same as `const moment = require("moment")`

a default import like `import moment from "moment"` acts the same as `const moment = require("moment").default`

This mis-match causes these two issues:

> the ES6 modules spec states that a namespace import (`import * as x`) can only be an object, by having TypeScript treating it the same as = `require("x")` then TypeScript allowed for the import to be treated as a function and be callable. That's not valid according to the spec.
>
> while accurate to the ES6 modules spec, most libraries with CommonJS/AMD/UMD modules didn't conform as strictly as TypeScript's implementation.

Turning on `esModuleInterop` will fix both of these problems in the code transpiled by TypeScript. The first changes the behavior in the compiler, the second is fixed by two new helper functions which provide a shim to ensure compatibility in the emitted JavaScript:

ts

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

import _ from "lodash";

fs.readFileSync("file.txt", "utf8");

_.chunk(["a", "b", "c", "d"], 2);
```

With `esModuleInterop` disabled:

ts

```ts
"use strict";

Object.defineProperty(exports, "__esModule", { value: true });

const fs = require("fs");

const lodash_1 = require("lodash");

fs.readFileSync("file.txt", "utf8");

lodash_1.default.chunk(["a", "b", "c", "d"], 2);
```

Try

With `esModuleInterop` set to `true`:

ts

```ts
"use strict";

var __createBinding = (this && this.__createBinding) || (Object.create ? (function(o, m, k, k2) {

    if (k2 === undefined) k2 = k;

    var desc = Object.getOwnPropertyDescriptor(m, k);

    if (!desc || ("get" in desc ? !m.__esModule : desc.writable || desc.configurable)) {

      desc = { enumerable: true, get: function() { return m[k]; } };

    }

    Object.defineProperty(o, k2, desc);

}) : (function(o, m, k, k2) {

    if (k2 === undefined) k2 = k;

    o[k2] = m[k];

}));

var __setModuleDefault = (this && this.__setModuleDefault) || (Object.create ? (function(o, v) {

    Object.defineProperty(o, "default", { enumerable: true, value: v });

}) : function(o, v) {

    o["default"] = v;
```

```
});
var __importStar = (this && this.__importStar) || function (mod) {
    if (mod && mod.__esModule) return mod;
    var result = {};
    if (mod != null) for (var k in mod) if (k !== "default" && Object.prototype.hasOwnProperty.call(mod, k)) __createBinding(r
    __setModuleDefault(result, mod);
    return result;
};
var __importDefault = (this && this.__importDefault) || function (mod) {
    return (mod && mod.__esModule) ? mod : { "default": mod };
};
Object.defineProperty(exports, "__esModule", { value: true });
const fs = __importStar(require("fs"));
const lodash_1 = __importDefault(require("lodash"));
fs.readFileSync("file.txt", "utf8");
lodash_1.default.chunk(["a", "b", "c", "d"], 2);
```

Try

Note: The namespace import `import * as fs from "fs"` only accounts for properties which are owned (basically properties set on the object and not via the prototype chain) on the imported object. If the module you're importing defines its API using inherited properties, you need to use the default import form (`import fs from "fs"`), or disable `esModuleInterop`.

Note: You can make JS emit terser by enabling `importHelpers`:

ts

```
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
const tslib_1 = require("tslib");
const fs = tslib_1.__importStar(require("fs"));
const lodash_1 = tslib_1.__importDefault(require("lodash"));
fs.readFileSync("file.txt", "utf8");
lodash_1.default.chunk(["a", "b", "c", "d"], 2);
```

Try

Enabling `esModuleInterop` will also enable `allowSyntheticDefaultImports`.

- Recommended
  Default:

  `true` if `module` is `node16` or `nodenext`; `false` otherwise.

  Related:

  `allowSyntheticDefaultImports`

  Released:

  2.7

Force Consistent Casing In File Names - `forceConsistentCasingInFileNames`

TypeScript follows the case sensitivity rules of the file system it's running on. This can be problematic if some developers are working in a case-sensitive file system and others aren't. If a file attempts to import `fileManager.ts` by specifying `./FileManager.ts` the file will be found in a case-insensitive file system, but not on a case-sensitive file system.

When this option is set, TypeScript will issue an error if a program tries to include a file by a casing different from the casing on disk.

- Recommended
  Default:

  `true`

Isolated Modules - `isolatedModules`

While you can use TypeScript to produce JavaScript code from TypeScript code, it's also common to use other transpilers such as Babel to do this. However, other transpilers only operate on a single file at a time, which means they can't apply code transforms that depend on understanding the full type system. This restriction also applies to TypeScript's `ts.transpileModule` API which is used by some build tools.

These limitations can cause runtime problems with some TypeScript features like `const enum`s and `namespace`s. Setting the `isolatedModules` flag tells TypeScript to warn you if you write certain code that can't be correctly interpreted by a single-file transpilation process.

It does not change the behavior of your code, or otherwise change the behavior of TypeScript's checking and emitting process.

Some examples of code which does not work when `isolatedModules` is enabled.

Exports of Non-Value Identifiers

In TypeScript, you can import a *type* and then subsequently export it:

ts

```ts
import { someType, someFunction } from "someModule";

someFunction();

export { someType, someFunction };
```

Try

Because there's no value for `someType`, the emitted `export` will not try to export it (this would be a runtime error in JavaScript):

js

```js
export { someFunction };
```

Single-file transpilers don't know whether `someType` produces a value or not, so it's an error to export a name that only refers to a type.

Non-Module Files

If `isolatedModules` is set, namespaces are only allowed in *modules* (which means it has some form of `import`/`export`). An error occurs if a namespace is found in a non-module file:

ts

```ts
namespace Instantiated {

Namespaces are not allowed in global script files when 'isolatedModules' is enabled. If this file is not intended to be a glob

  export const x = 1;

}
```

Try

This restriction doesn't apply to `.d.ts` files.

References to `const enum` members

In TypeScript, when you reference a `const enum` member, the reference is replaced by its actual value in the emitted JavaScript. Changing this TypeScript:

ts

```ts
declare const enum Numbers {
```

```
  Zero = 0,

  One = 1,

}

console.log(Numbers.Zero + Numbers.One);
```

Try

To this JavaScript:

ts

```
"use strict";

console.log(0 + 1);
```

Try

Without knowledge of the values of these members, other transpilers can't replace the references to `Numbers`, which would be a runtime error if left alone (since there are no `Numbers` object at runtime). Because of this, when `isolatedModules` is set, it is an error to reference an ambient `const enum` member.

Preserve Symlinks - `preserveSymlinks`

This is to reflect the same flag in Node.js; which does not resolve the real path of symlinks.

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).

With this enabled, 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.

Verbatim Module Syntax - `verbatimModuleSyntax`

By default, TypeScript does something called *import elision*. Basically, if you write something like

ts

```
import { Car } from "./car";

export function drive(car: Car) {

  // ...

}
```

TypeScript detects that you're only using an import for types and drops the import entirely. Your output JavaScript might look something like this:

js

```
export function drive(car) {

  // ...

}
```

Most of the time this is good, because if `Car` isn't a value that's exported from `./car`, we'll get a runtime error.

But it does add a layer of complexity for certain edge cases. For example, notice there's no statement like `import "./car";` - the import was dropped entirely. That actually makes a difference for modules that have side-effects or not.

TypeScript's emit strategy for JavaScript also has another few layers of complexity - import elision isn't always just driven by how an import is used - it often consults how a value is declared as well. So it's not always clear whether code like the following

ts

```
export { Car } from "./car";
```

should be preserved or dropped. If `Car` is declared with something like a `class`, then it can be preserved in the resulting JavaScript file. But if `Car` is only declared as a `type` alias or `interface`, then the JavaScript file shouldn't export `Car` at all.

While TypeScript might be able to make these emit decisions based on information from across files, not every compiler can.

The `type` modifier on imports and exports helps with these situations a bit. We can make it explicit whether an import or export is only being used for type analysis, and can be dropped entirely in JavaScript files by using the `type` modifier.

```ts
// This statement can be dropped entirely in JS output
import type * as car from "./car";

// The named import/export 'Car' can be dropped in JS output
import { type Car } from "./car";

export { type Car } from "./car";
```

`type` modifiers are not quite useful on their own - by default, module elision will still drop imports, and nothing forces you to make the distinction between `type` and plain imports and exports. So TypeScript has the flag `--importsNotUsedAsValues` to make sure you use the `type` modifier, `--preserveValueImports` to prevent *some* module elision behavior, and `--isolatedModules` to make sure that your TypeScript code works across different compilers. Unfortunately, understanding the fine details of those 3 flags is hard, and there are still some edge cases with unexpected behavior.

TypeScript 5.0 introduces a new option called `--verbatimModuleSyntax` to simplify the situation. The rules are much simpler - any imports or exports without a `type` modifier are left around. Anything that uses the `type` modifier is dropped entirely.

```ts
// Erased away entirely.
import type { A } from "a";

// Rewritten to 'import { b } from "bcd";'
import { b, type c, type d } from "bcd";

// Rewritten to 'import {} from "xyz";'
import { type xyz } from "xyz";
```

With this new option, what you see is what you get.

That does have some implications when it comes to module interop though. Under this flag, ECMAScript `imports` and `exports` won't be rewritten to `require` calls when your settings or file extension implied a different module system. Instead, you'll get an error. If you need to emit code that uses `require` and `module.exports`, you'll have to use TypeScript's module syntax that predates ES2015:

Input TypeScript

```ts
import foo = require("foo");
```

```ts
function foo() {}

function bar() {}

function baz() {}

export = {
  foo,
  bar,
  baz,
};
```

Output JavaScript

```js
const foo = require("foo");
```

```js
function foo() {}

function bar() {}

function baz() {}

module.exports = {
  foo,
  bar,
  baz,
};
```

While this is a limitation, it does help make some issues more obvious. For example, it's very common to forget to set the [`type` field in `package.json`](#) under `--module node16`. As a result, developers would start writing CommonJS modules instead of an ES modules without realizing it, giving surprising lookup rules and JavaScript output. This new flag ensures that you're intentional about the file type you're using because the syntax is intentionally different.

Because `--verbatimModuleSyntax` provides a more consistent story than `--importsNotUsedAsValues` and `--preserveValueImports`, those two existing flags are being deprecated in its favor.

For more details, read up on the original pull request and its proposal issue.

#Backwards Compatibility

Charset - `charset`

In prior versions of TypeScript, this controlled what encoding was used when reading text files from disk. Today, TypeScript assumes UTF-8 encoding, but will correctly detect UTF-16 (BE and LE) or UTF-8 BOMs.

- Deprecated
  Default:

  `utf8`

Keyof Strings Only - `keyofStringsOnly`

This flag changes the `keyof` type operator to return `string` instead of `string | number` when applied to a type with a string index signature.

This flag is used to help people keep this behavior from before TypeScript 2.9's release.

- Deprecated
  Released:

  2.9

No Implicit Use Strict - `noImplicitUseStrict`

You shouldn't need this. By default, when emitting a module file to a non-ES6 target, TypeScript emits a `"use strict";` prologue at the top of the file. This setting disables the prologue.

No Strict Generic Checks - `noStrictGenericChecks`

TypeScript will unify type parameters when comparing two generic functions.

```ts
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) {

  b = a; // Ok

  a = b; // Error

Type 'B' is not assignable to type 'A'.
 Types of parameters 'y' and 'y' are incompatible.
   Type 'U' is not assignable to type 'T'.
     'T' could be instantiated with an arbitrary type which could be unrelated to 'U'.2322Type 'B' is not assignable to type '
 Types of parameters 'y' and 'y' are incompatible.
   Type 'U' is not assignable to type 'T'.
     'T' could be instantiated with an arbitrary type which could be unrelated to 'U'.

}
```

Try

This flag can be used to remove that check.

Released:

2.4

Out - `out`

Use `outFile` instead.

The `out` option computes the final file location in a way that is not predictable or consistent. This option is retained for backward compatibility only and is deprecated.

- Deprecated
  Related:

outDir

outFile

Suppress Excess Property Errors - `suppressExcessPropertyErrors`

This disables reporting of excess property errors, such as the one shown in the following example:

ts

```ts
type Point = { x: number; y: number };

const p: Point = { x: 1, y: 3, m: 10 };
```

```
Object literal may only specify known properties, and 'm' does not exist in type 'Point'.2353Object literal may only specify k
```

This flag was added to help people migrate to the stricter checking of new object literals in TypeScript 1.6.

We don't recommend using this flag in a modern codebase, you can suppress one-off cases where you need it using `// @ts-ignore`.

Suppress Implicit Any Index Errors - `suppressImplicitAnyIndexErrors`

Turning `suppressImplicitAnyIndexErrors` on suppresses reporting the error about implicit anys when indexing into objects, as shown in the following example:

ts

```ts
const obj = { x: 10 };

console.log(obj["foo"]);
```

```
Element implicitly has an 'any' type because expression of type '"foo"' can't be used to index type '{ x: number; }'.
 Property 'foo' does not exist on type '{ x: number; }'.7053Element implicitly has an 'any' type because expression of type '"
 Property 'foo' does not exist on type '{ x: number; }'.Try
```

Using `suppressImplicitAnyIndexErrors` is quite a drastic approach. It is recommended to use a `@ts-ignore` comment instead:

ts

```ts
const obj = { x: 10 };

// @ts-ignore

console.log(obj["foo"]);
```

Try

Related:

noImplicitAny

#Language and Environment

Emit Decorator Metadata - `emitDecoratorMetadata`

Enables experimental support for emitting type metadata for decorators which works with the module `reflect-metadata`.

For example, here is the TypeScript

ts

```ts
function LogMethod(

  target: any,

  propertyKey: string | symbol,

  descriptor: PropertyDescriptor

) {
```

```
  console.log(target);
  console.log(propertyKey);
  console.log(descriptor);
}
class Demo {
  @LogMethod
  public foo(bar: number) {
    // do nothing
  }
}
const demo = new Demo();
```

Try

With `emitDecoratorMetadata` not set to true (default) the emitted JavaScript is:

ts

```
"use strict";
var __decorate = (this && this.__decorate) || function (decorators, target, key, desc) {
    var c = arguments.length, r = c < 3 ? target : desc === null ? desc = Object.getOwnPropertyDescriptor(target, key) : desc,
    if (typeof Reflect === "object" && typeof Reflect.decorate === "function") r = Reflect.decorate(decorators, target, key, d
    else for (var i = decorators.length - 1; i >= 0; i--) if (d = decorators[i]) r = (c < 3 ? d(r) : c > 3 ? d(target, key, r)
    return c > 3 && r && Object.defineProperty(target, key, r), r;
};
function LogMethod(target, propertyKey, descriptor) {
    console.log(target);
    console.log(propertyKey);
    console.log(descriptor);
}
class Demo {
    foo(bar) {
        // do nothing
    }
}
__decorate([
    LogMethod
], Demo.prototype, "foo", null);
const demo = new Demo();
```

Try

With `emitDecoratorMetadata` set to true the emitted JavaScript is:

ts

```
"use strict";
var __decorate = (this && this.__decorate) || function (decorators, target, key, desc) {
    var c = arguments.length, r = c < 3 ? target : desc === null ? desc = Object.getOwnPropertyDescriptor(target, key) : desc,
    if (typeof Reflect === "object" && typeof Reflect.decorate === "function") r = Reflect.decorate(decorators, target, key, d
    else for (var i = decorators.length - 1; i >= 0; i--) if (d = decorators[i]) r = (c < 3 ? d(r) : c > 3 ? d(target, key, r)
    return c > 3 && r && Object.defineProperty(target, key, r), r;
};
var __metadata = (this && this.__metadata) || function (k, v) {
    if (typeof Reflect === "object" && typeof Reflect.metadata === "function") return Reflect.metadata(k, v);
};
function LogMethod(target, propertyKey, descriptor) {
    console.log(target);
    console.log(propertyKey);
    console.log(descriptor);
}
class Demo {
    foo(bar) {
        // do nothing
    }
}
__decorate([
    LogMethod,
    __metadata("design:type", Function),
    __metadata("design:paramtypes", [Number]),
    __metadata("design:returntype", void 0)
], Demo.prototype, "foo", null);
const demo = new Demo();
```

Try

Related:

experimentalDecorators

Experimental Decorators - `experimentalDecorators`

Enables experimental support for decorators, which is a version of decorators that predates the TC39 standardization process.

Decorators are a language feature which hasn't yet been fully ratified into the JavaScript specification. This means that the implementation version in TypeScript may differ from the implementation in JavaScript when it it decided by TC39.

You can find out more about decorator support in TypeScript in the handbook.

Related:

emitDecoratorMetadata

JSX - `jsx`

Controls how JSX constructs are emitted in JavaScript files. This only affects output of JS files that started in `.tsx` files.

- `react`: Emit `.js` files with JSX changed to the equivalent `React.createElement` calls
- `react-jsx`: Emit `.js` files with the JSX changed to `_jsx` calls
- `react-jsxdev`: Emit `.js` files with the JSX changed to `_jsx` calls
- `preserve`: Emit `.jsx` files with the JSX unchanged
- `react-native`: Emit `.js` files with the JSX unchanged

For example

This sample code:

```
tsx

export const HelloWorld = () => <h1>Hello world</h1>;
```

Default: `"react"`

```
tsx

import React from 'react';

export const HelloWorld = () => React.createElement("h1", null, "Hello world");
```

Try

Preserve: `"preserve"`

```
tsx

import React from 'react';

export const HelloWorld = () => <h1>Hello world</h1>;
```

Try

React Native: `"react-native"`

```
tsx

import React from 'react';

export const HelloWorld = () => <h1>Hello world</h1>;
```

Try

React 17 transform: `"react-jsx"`[1]

```
tsx

import { jsx as _jsx } from "react/jsx-runtime";

export const HelloWorld = () => _jsx("h1", { children: "Hello world" });
```

Try

React 17 dev transform: `"react-jsxdev"`[1]

```
tsx

import { jsxDEV as _jsxDEV } from "react/jsx-dev-runtime";

const _jsxFileName = "/home/runner/work/TypeScript-Website/TypeScript-Website/packages/typescriptlang-org/index.tsx";

export const HelloWorld = () => _jsxDEV("h1", { children: "Hello world" }, void 0, false, { fileName: _jsxFileName, lineNumber
```

Try

Allowed:

`preserve`

react

react-native

react-jsx

react-jsxdev

Related:

jsxFactory

jsxFragmentFactory

jsxImportSource

Released:

2.2

JSX Factory - `jsxFactory`

Changes the function called in `.js` files when compiling JSX Elements using the classic JSX runtime. The most common change is to use `"h"` or `"preact.h"` instead of the default `"React.createElement"` if using `preact`.

For example, this TSX file:

tsx

```tsx
import { h } from "preact";

const HelloWorld = () => <div>Hello</div>;
```

With `jsxFactory: "h"` looks like:

tsx

```tsx
const preact_1 = require("preact");

const HelloWorld = () => (0, preact_1.h)("div", null, "Hello");
```

Try

This option can be used on a per-file basis too similar to Babel's `/** @jsx h */` directive.

tsx

```tsx
/** @jsx h */

import { h } from "preact";

Cannot find module 'preact' or its corresponding type declarations.2307Cannot find module 'preact' or its corresponding type d

const HelloWorld = () => <div>Hello</div>;
```

Try

The factory chosen will also affect where the JSX namespace is looked up (for type checking information) before falling back to the global one.

If the factory is defined as `React.createElement` (the default), the compiler will check for `React.JSX` before checking for a global JSX. If the factory is defined as h, it will check for `h.JSX` before a global JSX.

Default:

`React.createElement`

Allowed:

Any identifier or dotted identifier.

Related:

jsx

jsxFragmentFactory

jsxImportSource

JSX Fragment Factory - `jsxFragmentFactory`

Specify the JSX fragment factory function to use when targeting react JSX emit with `jsxFactory` compiler option is specified, e.g. `Fragment`.

For example with this TSConfig:

```
{
  "compilerOptions": {
    "target": "esnext",
    "module": "commonjs",
    "jsx": "react",
    "jsxFactory": "h",
    "jsxFragmentFactory": "Fragment"
  }
}
```

This TSX file:

```
tsx
import { h, Fragment } from "preact";
const HelloWorld = () => (
  <>
    <div>Hello</div>
  </>
);
```

Would look like:

```
tsx
const preact_1 = require("preact");
const HelloWorld = () => ((0, preact_1.h)(preact_1.Fragment, null,
    (0, preact_1.h)("div", null, "Hello")));
```

Try

This option can be used on a per-file basis too similar to Babel's `/* @jsxFrag h */` directive.

For example:

```
tsx
/** @jsx h */
/** @jsxFrag Fragment */
import { h, Fragment } from "preact";
Cannot find module 'preact' or its corresponding type declarations.2307Cannot find module 'preact' or its corresponding type d
const HelloWorld = () => (
  <>
```

```
    <div>Hello</div>
  </>
);
```

Try

Default:

`React.Fragment`

Related:

jsx

jsxFactory

jsxImportSource

Released:

4.0

JSX Import Source - `jsxImportSource`

Declares the module specifier to be used for importing the `jsx` and `jsxs` factory functions when using jsx as `"react-jsx"` or `"react-jsxdev"` which were introduced in TypeScript 4.1.

With React 17 the library supports a new form of JSX transformation via a separate import.

For example with this code:

tsx

```tsx
import React from "react";

function App() {
  return <h1>Hello World</h1>;
}
```

Using this TSConfig:

```
{
  "compilerOptions": {
    "target": "esnext",
    "module": "commonjs",
    "jsx": "react-jsx"
  }
}
```

The emitted JavaScript from TypeScript is:

tsx

```tsx
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
const jsx_runtime_1 = require("react/jsx-runtime");
function App() {
    return (0, jsx_runtime_1.jsx)("h1", { children: "Hello World" });
}
```

Try

For example if you wanted to use `"jsxImportSource": "preact"`, you need a tsconfig like:

```
{
  "compilerOptions": {
    "target": "esnext",
    "module": "commonjs",
    "jsx": "react-jsx",
    "jsxImportSource": "preact",
    "types": ["preact"]
  }
}
```

Which generates code like:

tsx

```
function App() {
    return (0, jsx_runtime_1.jsx)("h1", { children: "Hello World" });
}
exports.App = App;
```

Try

Alternatively, you can use a per-file pragma to set this option, for example:

tsx

```
/** @jsxImportSource preact */
export function App() {
  return <h1>Hello World</h1>;
}
```

Would add `preact/jsx-runtime` as an import for the `_jsx` factory.

Note: In order for this to work like you would expect, your `tsx` file must include an `export` or `import` so that it is considered a module.

Default:

`react`

Related:

`jsx`

`jsxFactory`

Released:

4.1

Lib - `lib`

TypeScript includes a default set of type definitions for built-in JS APIs (like `Math`), as well as type definitions for things found in browser environments (like `document`). TypeScript also includes APIs for newer JS features matching the `target` you specify; for example the definition for `Map` is available if `target` is ES6 or newer.

You may want to change these for a few reasons:

- Your program doesn't run in a browser, so you don't want the `"dom"` type definitions
- Your runtime platform provides certain JavaScript API objects (maybe through polyfills), but doesn't yet support the full syntax of a given ECMAScript version
- You have polyfills or native implementations for some, but not all, of a higher level ECMAScript version

In TypeScript 4.5, lib files can be overridden by npm modules, find out more in the blog.

High Level libraries

Name	Contents
`ES5`	Core definitions for all ES3 and ES5 functionality
`ES2015`	Additional APIs available in ES2015 (also known as ES6) - `array.find`, `Promise`, `Proxy`, `Symbol`, `Map`, `Set`, `Reflect`, etc.
`ES6`	Alias for "ES2015"
`ES2016`	Additional APIs available in ES2016 - `array.include`, etc.
`ES7`	Alias for "ES2016"
`ES2017`	Additional APIs available in ES2017 - `Object.entries`, `Object.values`, `Atomics`, `SharedArrayBuffer`, `date.formatToParts`, typed arrays, etc.
`ES2018`	Additional APIs available in ES2018 - `async` iterables, `promise.finally`, `Intl.PluralRules`, `regexp.groups`, etc.
`ES2019`	Additional APIs available in ES2019 - `array.flat`, `array.flatMap`, `Object.fromEntries`, `string.trimStart`, `string.trimEnd`, etc.
`ES2020`	Additional APIs available in ES2020 - `string.matchAll`, etc.
`ES2021`	Additional APIs available in ES2021 - `promise.any`, `string.replaceAll` etc.
`ES2022`	Additional APIs available in ES2022 - `array.at`, `RegExp.hasIndices`, etc.
`ESNext`	Additional APIs available in ESNext - This changes as the JavaScript specification evolves
`DOM`	DOM definitions - `window`, `document`, etc.
`WebWorker`	APIs available in WebWorker contexts
`ScriptHost`	APIs for the Windows Script Hosting System

Individual library components

Name
`DOM.Iterable`
`ES2015.Core`
`ES2015.Collection`
`ES2015.Generator`
`ES2015.Iterable`
`ES2015.Promise`
`ES2015.Proxy`
`ES2015.Reflect`
`ES2015.Symbol`
`ES2015.Symbol.WellKnown`
`ES2016.Array.Include`
`ES2017.object`
`ES2017.Intl`
`ES2017.SharedMemory`
`ES2017.String`
`ES2017.TypedArrays`
`ES2018.Intl`
`ES2018.Promise`
`ES2018.RegExp`
`ES2019.Array`
`ES2019.Object`
`ES2019.String`
`ES2019.Symbol`
`ES2020.String`
`ES2020.Symbol.wellknown`
`ES2021.Promise`
`ES2021.String`
`ES2021.WeakRef`
`ESNext.AsyncIterable`
`ESNext.Array`
`ESNext.Intl`
`ESNext.Symbol`

This list may be out of date, you can see the full list in the TypeScript source code.

Related:

noLib

Released:

2.0

Module Detection - moduleDetection

This setting controls how TypeScript determines whether a file is a script or a module.

There are three choices:

"auto" (default) - TypeScript will not only look for import and export statements, but it will also check whether the "type" field in a package.json is set to "module" when running with module: nodenext or node16, and check whether the current file is a JSX file when running under jsx: react-jsx.

"legacy" - The same behavior as 4.6 and prior, usings import and export statements to determine whether a file is a module.

"force" - Ensures that every non-declaration file is treated as a module.

Default:

"auto": Treat files with imports, exports, import.meta, jsx (with jsx: react-jsx), or esm format (with module: node16+) as modules.

Allowed:

legacy

auto

force

Released:

4.7

No Lib - noLib

Disables the automatic inclusion of any library files. If this option is set, lib is ignored.

TypeScript *cannot* compile anything without a set of interfaces for key primitives like: Array, Boolean, Function, IArguments, Number, Object, RegExp, and String. It is expected that if you use noLib you will be including your own type definitions for these.

Related:

lib

React Namespace - reactNamespace

Use jsxFactory instead. Specify the object invoked for createElement when targeting react for TSX files.

Default:

React

Target - target

Modern browsers support all ES6 features, so ES6 is a good choice. You might choose to set a lower target if your code is deployed to older environments, or a higher target if your code is guaranteed to run in newer environments.

The target setting changes which JS features are downleveled and which are left intact. For example, an arrow function () => this will be turned into an equivalent function expression if target is ES5 or lower.

Changing target also changes the default value of lib. You may “mix and match” target and lib settings as desired, but you could just set target for convenience.

For developer platforms like Node there are baselines for the target, depending on the type of platform and its version. You can find a set of community organized TSConfigs at tsconfig/bases, which has configurations for common platforms and their versions.

The special ESNext value refers to the highest version your version of TypeScript supports. This setting should be used with caution, since it doesn’t mean the same thing between different TypeScript versions and can make upgrades less predictable.

Default:

ES3

Allowed:

- `es3`
- `es5`
- `es6/es2015`
- `es2016`
- `es2017`
- `es2018`
- `es2019`
- `es2020`
- `es2021`
- `es2022`
- `esnext`

Released:

1.0

Use Define For Class Fields - `useDefineForClassFields`

This flag is used as part of migrating to the upcoming standard version of class fields. TypeScript introduced class fields many years before it was ratified in TC39. The latest version of the upcoming specification has a different runtime behavior to TypeScript's implementation but the same syntax.

This flag switches to the upcoming ECMA runtime behavior.

You can read more about the transition in the 3.7 release notes.

Default:

`true` if `target` is `ES2022` or higher, including `ESNext`; `false` otherwise.

Released:

3.7

#Compiler Diagnostics

Diagnostics - `diagnostics`

Used to output diagnostic information for debugging. This command is a subset of `extendedDiagnostics` which are more user-facing results, and easier to interpret.

If you have been asked by a TypeScript compiler engineer to give the results using this flag in a compile, in which there is no harm in using `extendedDiagnostics` instead.

- Deprecated

  Related:

  - `extendedDiagnostics`

Explain Files - `explainFiles`

Print names of files which TypeScript sees as a part of your project and the reason they are part of the compilation.

For example, with this project of just a single `index.ts` file

sh

```
example

■■■ index.ts

■■■ package.json

■■■ tsconfig.json
```

Using a `tsconfig.json` which has `explainFiles` set to true:

```
json

{

  "compilerOptions": {

    "target": "es5",

    "module": "commonjs",

    "explainFiles": true

  }

}
```

Running TypeScript against this folder would have output like this:

```
› tsc

node_modules/typescript/lib/lib.d.ts

  Default library for target 'es5'

node_modules/typescript/lib/lib.es5.d.ts

  Library referenced via 'es5' from file 'node_modules/typescript/lib/lib.d.ts'

node_modules/typescript/lib/lib.dom.d.ts

  Library referenced via 'dom' from file 'node_modules/typescript/lib/lib.d.ts'

node_modules/typescript/lib/lib.webworker.importscripts.d.ts

  Library referenced via 'webworker.importscripts' from file 'node_modules/typescript/lib/lib.d.ts'

node_modules/typescript/lib/lib.scripthost.d.ts

  Library referenced via 'scripthost' from file 'node_modules/typescript/lib/lib.d.ts'

index.ts

  Matched by include pattern '**/*' in 'tsconfig.json'
```

The output above show:

- The initial lib.d.ts lookup based on `target`, and the chain of `.d.ts` files which are referenced
- The `index.ts` file located via the default pattern of `include`

This option is intended for debugging how a file has become a part of your compile.

Released:

4.2

Extended Diagnostics - `extendedDiagnostics`

You can use this flag to discover where TypeScript is spending its time when compiling. This is a tool used for understanding the performance characteristics of your codebase overall.

You can learn more about how to measure and understand the output in the performance section of the wiki.

Related:

diagnostics

Generate CPU Profile - `generateCpuProfile`

This option gives you the chance to have TypeScript emit a v8 CPU profile during the compiler run. The CPU profile can provide insight into why your builds may be slow.

This option can only be used from the CLI via: `--generateCpuProfile tsc-output.cpuprofile`.

```sh
npm run tsc --generateCpuProfile tsc-output.cpuprofile
```

This file can be opened in a chromium based browser like Chrome or Edge Developer in the CPU profiler section. You can learn more about understanding the compilers performance in the TypeScript wiki section on performance.

Default:

`profile.cpuprofile`

Released:

3.7

List Emitted Files - `listEmittedFiles`

Print names of generated files part of the compilation to the terminal.

This flag is useful in two cases:

- You want to transpile TypeScript as a part of a build chain in the terminal where the filenames are processed in the next command.
- You are not sure that TypeScript has included a file you expected, as a part of debugging the file inclusion settings.

For example:

```
example
■■■ index.ts
■■■ package.json
■■■ tsconfig.json
```

With:

```
{
  "compilerOptions": {
    "declaration": true,
    "listEmittedFiles": true
  }
}
```

Would echo paths like:

```
$ npm run tsc
path/to/example/index.js
path/to/example/index.d.ts
```

Normally, TypeScript would return silently on success.

List Files - `listFiles`

Print names of files part of the compilation. This is useful when you are not sure that TypeScript has included a file you expected.

For example:

```
example

■■■ index.ts

■■■ package.json

■■■ tsconfig.json
```

With:

```
{

  "compilerOptions": {

    "listFiles": true

  }

}
```

Would echo paths like:

```
$ npm run tsc

path/to/example/node_modules/typescript/lib/lib.d.ts

path/to/example/node_modules/typescript/lib/lib.es5.d.ts

path/to/example/node_modules/typescript/lib/lib.dom.d.ts

path/to/example/node_modules/typescript/lib/lib.webworker.importscripts.d.ts

path/to/example/node_modules/typescript/lib/lib.scripthost.d.ts

path/to/example/index.ts
```

Note if using TypeScript 4.2, prefer `explainFiles` which offers an explanation of why a file was added too.

Related:

`explainFiles`

Trace Resolution - `traceResolution`

When you are trying to debug why a module isn't being included. You can set `traceResolution` to `true` to have TypeScript print information about its resolution process for each processed file.

Released:

2.0

#Projects

Composite - `composite`

The `composite` option enforces certain constraints which make it possible for build tools (including TypeScript itself, under `--build` mode) to quickly determine if a project has been built yet.

When this setting is on:

The `rootDir` setting, if not explicitly set, defaults to the directory containing the `tsconfig.json` 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` defaults to `true`

You can find documentation on TypeScript projects in the handbook.

Related:

`incremental`

`tsBuildInfoFile`

Released:

3.0

Disable Referenced Project Load - `disableReferencedProjectLoad`

In multi-project TypeScript programs, TypeScript will load all of the available projects into memory in order to provide accurate results for editor responses which require a full knowledge graph like 'Find All References'.

If your project is large, you can use the flag `disableReferencedProjectLoad` to disable the automatic loading of all projects. Instead, projects are loaded dynamically as you open files through your editor.

Released:

4.0

Disable Solution Searching - `disableSolutionSearching`

When working with composite TypeScript projects, this option provides a way to declare that you do not want a project to be included when using features like *find all references* or *jump to definition* in an editor.

This flag is something you can use to increase responsiveness in large composite projects.

Released:

3.8

Disable Source Project Reference Redirect - `disableSourceOfProjectReferenceRedirect`

When working with composite TypeScript projects, this option provides a way to go back to the pre-3.7 behavior where d.ts files were used to as the boundaries between modules. In 3.7 the source of truth is now your TypeScript files.

Released:

3.7

Incremental - `incremental`

Tells TypeScript to save information about the project graph from the last compilation to files stored on disk. This creates a series of `.tsbuildinfo` files in the same folder as your compilation output. They are not used by your JavaScript at runtime and can be safely deleted. You can read more about the flag in the 3.4 release notes.

To control which folders you want to the files to be built to, use the config option `tsBuildInfoFile`.

Default:

`true` if `composite`; `false` otherwise.

Related:

`composite`

`tsBuildInfoFile`

Released:

3.4

TS Build Info File - `tsBuildInfoFile`

This setting lets you specify a file for storing incremental compilation information as a part of composite projects which enables faster building of larger TypeScript codebases. You can read more about composite projects in the handbook.

The default depends on a combination of other settings:

- If `outFile` is set, the default is `<outFile>.tsbuildinfo`.

- If `rootDir` and `outDir` are set, then the file is `<outDir>/<relative path to config from rootDir>/<config name>.tsbuildinfo` For example, if `rootDir` is `src`, `outDir` is `dest`, and the config is `./tsconfig.json`, then the default is `./tsconfig.tsbuildinfo` as the relative path from `src/` to `./tsconfig.json` is `../`.
- If `outDir` is set, then the default is `<outDir>/<config name>.tsbuildInfo`
- Otherwise, the default is `<config name>.tsbuildInfo`

  Default:

  `.tsbuildinfo`

  Related:

  `incremental`

  `composite`

  Released:

  3.4

#Output Formatting

No Error Truncation - `noErrorTruncation`

Do not truncate error messages.

With `false`, the default.

```ts
var x: {
  propertyWithAnExceedinglyLongName1: string;
  propertyWithAnExceedinglyLongName2: string;
  propertyWithAnExceedinglyLongName3: string;
  propertyWithAnExceedinglyLongName4: string;
  propertyWithAnExceedinglyLongName5: string;
  propertyWithAnExceedinglyLongName6: string;
  propertyWithAnExceedinglyLongName7: string;
  propertyWithAnExceedinglyLongName8: string;
};
// String representation of type of 'x' should be truncated in error message
var s: string = x;
```

```
Type '{ propertyWithAnExceedinglyLongName1: string; propertyWithAnExceedinglyLongName2: string; propertyWithAnExceedinglyLongN
2454Type '{ propertyWithAnExceedinglyLongName1: string; propertyWithAnExceedinglyLongName2: string; propertyWithAnExceedinglyL
```

With `true`

```ts
var x: {
  propertyWithAnExceedinglyLongName1: string;
  propertyWithAnExceedinglyLongName2: string;
  propertyWithAnExceedinglyLongName3: string;
  propertyWithAnExceedinglyLongName4: string;
  propertyWithAnExceedinglyLongName5: string;
```

```
  propertyWithAnExceedinglyLongName6: string;

  propertyWithAnExceedinglyLongName7: string;

  propertyWithAnExceedinglyLongName8: string;

};

// String representation of type of 'x' should be truncated in error message

var s: string = x;

Type '{ propertyWithAnExceedinglyLongName1: string; propertyWithAnExceedinglyLongName2: string; propertyWithAnExceedinglyLongN
2454Type '{ propertyWithAnExceedinglyLongName1: string; propertyWithAnExceedinglyLongName2: string; propertyWithAnExceedinglyL
```

Preserve Watch Output - `preserveWatchOutput`

Whether to keep outdated console output in watch mode instead of clearing the screen every time a change happened.

- Internal

Pretty - `pretty`

Stylize errors and messages using color and context, this is on by default — offers you a chance to have less terse, single colored messages from the compiler.

Default:

`true`

#Completeness

Skip Default Lib Check - `skipDefaultLibCheck`

Use `skipLibCheck` instead. Skip type checking of default library declaration files.

Skip Lib Check - `skipLibCheck`

Skip type checking of declaration files.

This can save time during compilation at the expense of type-system accuracy. For example, two libraries could define two copies of the same `type` in an inconsistent way. Rather than doing a full check of all `d.ts` files, TypeScript will type check the code you specifically refer to in your app's source code.

A common case where you might think to use `skipLibCheck` is when there are two copies of a library's types in your `node_modules`. In these cases, you should consider using a feature like yarn's resolutions to ensure there is only one copy of that dependency in your tree or investigate how to ensure there is only one copy by understanding the dependency resolution to fix the issue without additional tooling.

Another possibility is when you are migrating between TypeScript releases and the changes cause breakages in node_modules and the JS standard libraries which you do not want to deal with during the TypeScript update.

Note, that if these issues come from the TypeScript standard library you can replace the library using TypeScript 4.5's lib replacement technique.

- Recommended
  Released:

  2.0

#Command Line

#Watch Options

TypeScript 3.8 shipped a new strategy for watching directories, which is crucial for efficiently picking up changes to `node_modules`.

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 number of files in `node_modules`, whereas there are way fewer directories to track.

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 which allows users to tell the compiler/language service which watching strategies should be used to keep track of files and directories.

Assume Changes Only Affect Direct Dependencies - `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.

This can be considered a 'fast & loose' implementation of the watching algorithm, which can drastically reduce incremental rebuild times at the expense of having to run the full build occasionally to get all compiler error messages.

Released:

3.8

Watch Options

You can configure the how TypeScript `--watch` works. This section is mainly for handling case where `fs.watch` and `fs.watchFile` have additional constraints like on Linux. You can read more at Configuring Watch.

Watch File - `watchFile`

The strategy for how individual files are watched.

- `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 parent directory

Allowed:

```
fixedpollinginterval

prioritypollinginterval

dynamicprioritypolling

fixedchunksizepolling

usefsevents

usefseventsonparentdirectory
```

Released:

3.8

Watch Directory - `watchDirectory`

The strategy for how entire directory trees are watched under systems that lack recursive file-watching functionality.

- `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.

Allowed:

```
usefsevents

fixedpollinginterval

dynamicprioritypolling

fixedchunksizepolling
```

Released:

3.8

Fallback Polling - `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.

- `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.
- `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.

  Allowed:

  `fixedinterval`

  `priorityinterval`

  `dynamicpriority`

  `fixedchunksize`

  Released:

  3.8

Synchronous Watch Directory - `synchronousWatchDirectory`

Synchronously call callbacks and update the state of directory watchers on platforms that don`t support recursive watching natively. Instead of giving a small timeout to allow for potentially multiple edits to occur on a file.

```
{
  "watchOptions": {
    "synchronousWatchDirectory": true
  }
}
```

Exclude Directories - `excludeDirectories`

You can use `excludeFiles` to drastically reduce the number of files which are watched during `--watch`. This can be a useful way to reduce the number of open file which TypeScript tracks on Linux.

```
{
  "watchOptions": {
    "excludeDirectories": ["**/node_modules", "_build", "temp/*"]
  }
}
```

Exclude Files - `excludeFiles`

You can use `excludeFiles` to remove a set of specific files from the files which are watched.

```
{
  "watchOptions": {
    "excludeFiles": ["temp/file.ts"]
  }
}
```

Type Acquisition

Type Acquisition is only important for JavaScript projects. In TypeScript projects you need to include the types in your projects explicitly. However, for JavaScript projects, the TypeScript tooling will download types for your modules in the background and outside of your node_modules folder.

Enable - `enable`

Disables automatic type acquisition in JavaScript projects:

```
json

{

  "typeAcquisition": {

    "enable": false

  }

}
```

Include - `include`

If you have a JavaScript project where TypeScript needs additional guidance to understand global dependencies, or have disabled the built-in inference via `disableFilenameBasedTypeAcquisition`.

You can use `include` to specify which types should be used from DefinitelyTyped:

```
json

{

  "typeAcquisition": {

    "include": ["jquery"]

  }

}
```

Exclude - `exclude`

Offers a config for disabling the type-acquisition for a certain module in JavaScript projects. This can be useful for projects which include other libraries in testing infrastructure which aren't needed in the main application.

```
json

{

  "typeAcquisition": {

    "exclude": ["jest", "mocha"]

  }

}
```

Disable Filename Based Type Acquisition - `disableFilenameBasedTypeAcquisition`

TypeScript's type acquisition can infer what types should be added based on filenames in a project. This means that having a file like `jquery.js` in your project would automatically download the types for JQuery from DefinitelyTyped.

You can disable this via `disableFilenameBasedTypeAcquisition`.

```
json

{

  "typeAcquisition": {

    "disableFilenameBasedTypeAcquisition": true

  }

}
```

Released:

4.1