Triple-Slash Directives

Triple-slash directives are single-line comments containing a single XML tag. The contents of the comment are used as compiler directives.

Triple-slash directives are **only** valid at the top of their containing file. A triple-slash directive can only be preceded by single or multi-line comments, including other triple-slash directives. If they are encountered following a statement or a declaration they are treated as regular single-line comments, and hold no special meaning.

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

The `/// <reference path="..." />` directive is the most common of this group. It serves as a declaration of *dependency* between files.

Triple-slash references instruct the compiler to include additional files in the compilation process.

They also serve as a method to order the output when using `--out` or `--outFile`. Files are emitted to the output file location in the same order as the input after preprocessing pass.

Preprocessing input files

The compiler performs a preprocessing pass on input files to resolve all triple-slash reference directives. During this process, additional files are added to the compilation.

The process starts with a set of *root files*; these are the file names specified on the command-line or in the `"files"` list in the `tsconfig.json` file. These root files are preprocessed in the same order they are specified. Before a file is added to the list, all triple-slash references in it are processed, and their targets included. Triple-slash references are resolved in a depth first manner, in the order they have been seen in the file.

A triple-slash reference path is resolved relative to the containing file, if unrooted.

Errors

It is an error to reference a file that does not exist. It is an error for a file to have a triple-slash reference to itself.

Using `--noResolve`

If the compiler flag `--noResolve` is specified, triple-slash references are ignored; they neither result in adding new files, nor change the order of the files provided.

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

Similar to a `/// <reference path="..." />` directive, this directive serves as a declaration of *dependency*; a `/// <reference types="..." />` directive, however, declares a dependency on a package.

The process of resolving these package names is similar to the process of resolving module names in an `import` statement. An easy way to think of triple-slash-reference-types directives are as an `import` for declaration packages.

For example, including `/// <reference types="node" />` in a declaration file declares that this file uses names declared in `@types/node/index.d.ts`; and thus, this package needs to be included in the compilation along with the declaration file.

Use these directives only when you're authoring a `d.ts` file by hand.

For declaration files generated during compilation, the compiler will automatically add `/// <reference types="..." />` for you; A `/// <reference types="..." />` in a generated declaration file is added *if and only if* the resulting file uses any declarations from the referenced package.

For declaring a dependency on an `@types` package in a `.ts` file, use `--types` on the command line or in your `tsconfig.json` instead. See using `@types`, `typeRoots and types in tsconfig.json` files (./tsconfig-json.html#types-typeroots-and-types) for more details.

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

This directive allows a file to explicitly include an existing built-in *lib* file.

Built-in *lib* files are referenced in the same fashion as the `"lib"` compiler option in *tsconfig.json* (e.g. use `lib="es2015"` and not `lib="lib.es2015.d.ts"`, etc.).

For declaration file authors who relay on built-in types, e.g. DOM APIs or built-in JS run-time constructors like `Symbol` or `Iterable`, triple-slash-reference lib directives are the recommended. Previously these .d.ts files had to add forward/duplicate declarations of such types.

For example, adding `/// <reference lib="es2017.string" />` to one of the files in a compilation is equivalent to compiling with `--lib es2017.string`.

```
/// <reference lib="es2017.string" />

"foo".padStart(4);
```

`/// <reference no-default-lib="true"/>`

This directive marks a file as a *default library*. You will see this comment at the top of `lib.d.ts` and its different variants.

This directive instructs the compiler to *not* include the default library (i.e. `lib.d.ts`) in the compilation. The impact here is similar to passing `--noLib` on the command line.

Also note that when passing `--skipDefaultLibCheck`, the compiler will only skip checking files with `/// <reference no-default-lib="true"/>`.

`/// <amd-module />`

By default AMD modules are generated anonymous. This can lead to problems when other tools are used to process the resulting modules, such as bundlers (e.g. `r.js`).

The `amd-module` directive allows passing an optional module name to the compiler:

amdModule.ts

```
///<amd-module name="NamedModule"/>
export class C {
}
```

Will result in assigning the name `NamedModule` to the module as part of calling the AMD `define` :

amdModule.js

```
define("NamedModule", ["require", "exports"], function (require, exports) {
    var C = (function () {
        function C() {
        }
        return C;
    })();
    exports.C = C;
});
```

`/// <amd-dependency />`

> **Note**: this directive has been deprecated. Use `import "moduleName";` statements instead.

`/// <amd-dependency path="x" />` informs the compiler about a non-TS module dependency that needs to be injected in the resulting module's require call.

The `amd-dependency` directive can also have an optional `name` property; this allows passing an optional name for an amd-dependency:

```
/// <amd-dependency path="legacy/moduleA" name="moduleA"/>
declare var moduleA:MyType
moduleA.callStuff()
```

Generated JS code:

```
define(["require", "exports", "legacy/moduleA"], function (require, exports, moduleA) {
    moduleA.callStuff()
});
```