imports

The `imports` module attempts to unify all import handling in ngtsc. It powers the compiler's reference system - how the compiler tracks classes like components, directives, NgModules, etc, and how it generates imports to them in user code. At its heart is the `Reference` abstraction, which combines a class `ts.Declaration` with any additional context needed to generate an import of that class in different situations.

In Angular, users do not import the directives and pipes they use in templates directly. Instead, they import an NgModule, and the NgModule exports a set of directives/pipes which will be available in the template of any consumer. When generating code for the template, though, the directives/pipes used there need to be imported directly. This creates a challenge for the compiler: it must choose an ES module specifier from which they can be imported, since the user never provided it.

Much of the logic around imports and references in the compiler is dedicated to answering this question. The compiler has two major modes of operation here:

1. Module specifier (import path) tracking

If a directive/pipe is within the user's program, then it can be imported directly. If not (e.g. the directive came from a library in `node_modules`), the compiler will look at the NgModule that caused the directive to be available in the template, look at its import, and attempt to use the same module specifier.

This logic is based on the Angular Package Format, which dictates that libraries are organized into entrypoints, and both an NgModule and its directives/pipes must be exported from the same entrypoint (usually an `index.ts` file).

Thus, if `CommonModule` is imported from the specifier '@angular/common', and its `NgIf` directive is used in a template, the compiler will always import `NgIf` from '@angular/common' as well.

It's important to note that this logic is transitive. If the user instead imported `BrowserModule` from '@angular/platform-browser' (which re-exports `CommonModule` and thus `NgIf`), the compiler will note that `BrowserModule` itself imported `CommonModule` from '@angular/common', and so `NgIf` will be imported from '@angular/common' still.

This logic of course breaks down for non-Angular Package Format libraries, such as "internal" libraries within a monorepo, which frequently don't use `index.ts` files or entrypoints. In this case, the user will likely import NgModules directly from their declaration (e.g. via a 'lib/module' specifier), and the compiler cannot simply assume that the user has exported all of the directives/pipes from the NgModule via this same specifier. In this case a compiler feature called "aliasing" kicks in (see below) and generates private exports from the NgModule file.

2. Using a `UnifiedModulesHost`

The `ts.CompilerHost` given to the compiler may optionally implement an interface called `UnifiedModulesHost`, which allows an absolute module specifier to be generated for any file. If a `UnifiedModulesHost` is present, the compiler will attempt to directly import all directives and pipes from the file which declares them, instead of going via the specifier of the NgModule as in the first mode described above. This logic is used internally in the Google monorepo.

This approach comes with a significant caveat: the build system may prevent importing from files which are not directly declared dependencies of the current compilation (this is known as "strict dependency checking"). This is a problem when attempting to consume a re-exported directive. For example, if the user depends only on '@angular/platform-browser', imports `BrowserModule` from '@angular/platform-browser' and attempts to use the re-exported `NgIf`, the compiler cannot import `NgIf` directly from its declaration within '@angular/common', which is a transitive (but not direct) dependency.

To support these re-exports, a compiler feature called "aliasing" will create a re-export of `NgIf` from within @angular/platform-browser when compiling that package. Then, the downstream application compiler can import `NgIf` via this "alias" re-export from a direct dependency, instead of needing to import it from a transitive dependency.

References

At its heart, the compiler keeps track of the types (classes) it's operating on using the `ts.Declaration` of that class. This suffices to *identify* a class; however, the compiler frequently needs to track not only the class itself, but how that class came to be known in a particular context. For example, the compiler might see that `CommonModule` is included in `AppModule`'s imports, but it also needs to keep track of from where `CommonModule` was imported to apply the logic of "module specifier tracking" described above.

To do this, the compiler will wrap the `ts.Declaration` of `CommonModule` into a `Reference`. A `Reference` is a pointer to a `ts.Declaration` plus any additional information and context about *how* that reference came to be.

Identifier tracking

Where possible, the compiler tries to use existing user-provided imports to refer to classes, instead of generating new imports. This is possible because `Reference`s keep track of any `ts.Identifier`s encountered which refer to the referenced class. If Angular, in the course of processing a `ts.Expression` (such as the `declarations` array of an NgModule), determines that the `ts.Identifier` points to a `Reference`, it adds the `ts.Identifier` to that `Reference` for future use.

The `Reference.getIdentityIn` method queries the `Reference` for a `ts.Identifier` that's valid in a given `ts.SourceFile`. This is used by the

`LocalIdentifierStrategy` when emitting an `Expression` for the `Reference` (see the description of `ReferenceEmitter` below).

Synthetic references In some cases, identifier tracking needs to be disabled for a `Reference`. For example, when the compiler synthesizes a `Reference` as part of "foreign function evaluation", the evaluated `ts.Identifier` may not be a direct reference to the `Reference`'s class at runtime, even if logically that interpretation makes sense in the context of the current expression.

In these cases, the `References` are marked as `synthetic`, which disables all `ts.Identifier` tracking.

Owning modules

As described above, one piece of information the compiler tracks about a `Reference` is the module specifier from which it came. This is known as its "owning module".

For a `Reference`, the compiler tracks both the module specifier itself as well as the context file which contained this module specifier (which is important for TypeScript module resolution operations).

This information is tracked in `Reference.bestGuessOwningModule`. This field carries the "best guess" prefix because the compiler cannot verify that each `Reference` which was extracted from a given ES module is actually exported via that module specifier. This depends on the packaging convention the user chose to use. Since a `Reference` may not belong to any external module, `bestGuessOwningModule` may be `null`.

For convenience, the module specifier as a string is also made available as `Reference.ownedByModuleGuess`.

ReferenceEmitter

During evaluation of `ts.Expressions`, `References` to specific directives/pipes/etc are created. During code generation, imports need to be generated for a particular component's template function, based on these `References`. This job falls to the `ReferenceEmitter`.

A `ReferenceEmitter` takes a `Reference` as well as a `ts.SourceFile` which will contain the import, and generates an `Expression` which can be used to refer to the referenced class within that file. This may or may not be an `ExternalExpression` (which would generate an import statement), depending on whether it's possible to rely on an existing import of the class within that file.

`ReferenceEmitter` is a wrapper around one or more `ReferenceEmitStrategy` instances. Each strategy is tried in succession until an `Expression` can be determined. An error is produced if no valid mechanism of referring to the referenced class can be found.

`LocalIdentifierStrategy`

This `ReferenceEmitStrategy` queries the `Reference` for a `ts.Identifier` that's valid in the requested file (see "identifier tracking" for `Reference`s above).

`LogicalProjectStrategy`

This `ReferenceEmitStrategy` is used to import referenced classes that are declared in the current project, and not in any third-party or external libraries, whenever `rootDir` or `rootDirs` is set in the TS compiler options.

When `rootDir`(s) are present, multiple physical directories can be mapped into the same logical namespace. So consider two files `/app/app.cmp.ts` and `/lib/lib.cmp.ts`. Ordinarily, a relative import (such as the kind generated by `RelativePathStrategy`) from the former to the latter would be `../lib/lib.cmp`. However, if both `/app` and `/lib` are project `rootDirs`, then the files within are logically in the same "directory", and the correct import is `./lib.cmp`.

The `LogicalProjectStrategy` constructs `LogicalProjectPath`s between files and generates module specifiers that are relative imports within that namespace, honoring the project's `rootDirs` settings.

The `LogicalProjectStrategy` will decline to generate an import into any file which falls outside the project's `rootDirs`, as such a relative specifier is not representable in the merged namespace.

`RelativePathStrategy`

This `ReferenceEmitStrategy` is used to generate relative imports between two files in the project, assuming the layout of files on the disk maps directly to the module specifier namespace. This is the case if the project does not have `rootDir`/`rootDirs` configured in its TS compiler options.

`AbsoluteModuleStrategy`

This `ReferenceEmitStrategy` uses the `bestGuessOwningModule` of a `Reference` to generate an import of the referenced class.

Note that the `bestGuessOwningModule` only gives the module specifier for the import, not the symbol name. The user may have renamed the class as part of re-exporting it from an entrypoint, so the `AbsoluteModuleStrategy` searches the exports of the target module and finds the symbol name by which the class is re-exported, if it exists.

`UnifiedModulesStrategy`

This `ReferenceEmitStrategy` uses a `UnifiedModulesHost` to implement the major import mode #2 described at the beginning of this document.

Under this strategy, direct imports to referenced classes are constructed using globally valid absolute module specifiers determined by the `UnifiedModulesHost`.

Like with `AbsoluteModuleStrategy`, the `UnifiedModulesHost` only gives the module specifier and not the symbol name, so an appropriate symbol name must be determined by searching the exports of the module.

`AliasStrategy`

The `AliasStrategy` will choose the alias `Expression` of a `Reference`. This strategy is used before the `UnifiedModulesStrategy` to guarantee aliases are preferred to direct imports when available.

See the description of aliasing in the case of `UnifiedModulesAliasingHost` below.

Aliasing and re-exports

In certain cases, the exports written by the user are not sufficient to guarantee that a downstream compiler will be able to depend on directives/pipes correctly. In these circumstances the compiler's "aliasing" system creates new exports to bridge the gaps.

An `AliasingHost` interface allows different aliasing strategies to be chosen based on the needs of the current compilation. It supports two operations:

1. Determination of a re-export name, if needed, for a given directive/pipe.

When compiling an NgModule, the compiler will consult the `AliasingHost` via its `maybeAliasSymbolAs` method to determine whether to add re-exports of any directives/pipes exported (directly or indirectly) by the NgModule.

2. Determination of an alias `Expression` for a directive/pipe, based on a re-export that was expected to have been generated previously.

When the user imports an NgModule from an external library (via a `.d.ts` file), the compiler will construct a "scope" of exported directives/pipes that this NgModule makes available to any templates. In the process of constructing this scope, the compiler creates `Reference`s for each directive/pipe.

As part of this operation, the compiler will consult the `AliasingHost` via its `getAliasIn` method to determine whether an alias `Expression` should be used to refer to each class instead of going through other import generation logic. This alias is saved on the `Reference`.

Because the first import of an NgModule from a user library to a `.d.ts` is always from a direct dependency, the result is that all `Reference`s to directives/pipes which can be used from this module will have an associated alias `Expression` specifying how to import them from that direct dependency, instead of from a transitive dependency.

Aliasing is currently used in two cases:

1. To address strict dependency checking issues when using a `UnifiedModulesHost`.
2. To support dependening on non-Angular Package Format packages (e.g. private libraries in monorepos) which do not have an entrypoint file through which all directives/pipes/modules are exported.

In environments with "strict dependency checking" as described above, an NgModule which exports another NgModule from one of its dependencies needs to export its directives/pipes as well, in order to make them available to the downstream compiler.

Aliasing under `UnifiedModulesHost`

A `UnifiedModulesAliasingHost` implements `AliasingHost` and makes full use of the aliasing system in the case of a `UnifiedModulesHost`.

When compiling an NgModule, re-exports are added under a stable name for each directive/pipe that's re-exported by the NgModule.

When importing that NgModule, alias `Expression`s are added to all the `Reference`s for those directives/pipes that are guaranteed to be from a direct dependency.

Private re-exports for non-APF packages

A `PrivateExportAliasingHost` is used to add re-exports of directives/pipes in the case where the compiler cannot determine that all directives/pipes are re-exported from a common entrypoint (like in the case of an Angular Package Format compilation).

In this case, aliasing is used to proactively add re-exports of directives/pipes to the file of any NgModule which exports them, ensuring they can be imported from the same module specifier as the NgModule itself. This is only done if the user has not already added such exports directly.

This `AliasingHost` does not tag any `Reference`s with aliases, and relies on the action of the `AbsoluteModuleStrategy` described above to find and select the alias re-export when attempting to write an import for a given `Reference`.

Default imports

This aspect of the `imports` package is a little different than the rest of the code as it's not concerned with directive/pipe imports. Instead, it's concerned with a different problem: preventing the removal of default import statements which were converted from type-only to value imports through compilation.

Type-to-value compilation

This occurs when a default import is used as a type parameter in a service constructor:

```
import Foo from 'foo';

@Injectable()
export class Svc {
  constructor(private foo: Foo) {}
}
```

Here, `Foo` is used in the type position only, but the compiler will eventually generate an `inject(Foo)` call in the factory function for this service. The use of `Foo` like this in the output depends on the import statement surviving compilation.

Due to quirks in TypeScript transformers (see below), TypeScript considers the import to be type-only and does not notice the additional usage as a value added during transformation, and so will attempt to remove the import. The default import managing system exists to prevent this.

It consists of two mechanisms:

1. A `DefaultImportTracker`, which records information about both default imports encountered in the program as well as usages of those imports added during compilation.
2. A TypeScript transformer which processes default import statements and can preserve those which are actually used.

This is accessed via `DefaultImportTracker.importPreservingTransformer`.

Why default imports are problematic

This section is the result of speculation, as we have not traced the TypeScript compiler thoroughly.

Consider the class:

```
import {Foo} from './foo';

class X {
  constructor(foo: Foo) {}
}
```

Angular wants to generate a value expression (`inject(Foo)`), using the value side of the `Foo` type from the constructor.

After transforms, this roughly looks like:

```
let foo_1 = require('./foo');

inject(foo_1.Foo);
```

The Angular compiler takes the `Foo` `ts.Identifier` from the import statement `import {Foo} from './foo'`, which has a "provenance" in TypeScript that indicates it's associated with the import statement. After transforms, TypeScript will scan the output code and notice this `ts.Identifier` is still present, and so it will choose to preserve the import statement.

If, however, `Foo` was a default import:

```
import Foo from './foo';
```

Then the generated code depends on a few factors (target/module/esModuleInterop settings), but roughly looks like:

```
let foo_1 = require('./foo');

inject(foo_1.default);
```

Note in this output, the `Foo` identifier from before has disappeared. TypeScript then does not find any `ts.Identifier`s which point back to the original import statement, and thus it concludes that the import is unused.

It's likely that this case was overlooked in the design of the transformers API.