‹ Metadata Keywords ›

Libraries & imports

Contents

The `import` and `library` directives can help you create a modular and shareable code base. Libraries not only provide APIs, but are a unit of privacy: identifiers that start with an underscore (`_`) are visible only inside the library. *Every Dart file (plus its parts) is a library*, even if it doesn't use a `library` directive.

Libraries can be distributed using packages.

ⓘ If you're curious why Dart uses underscores instead of access modifier keywords like `public` or `private`, see SDK issue 33383.

Using libraries

Use `import` to specify how a namespace from one library is used in the scope of another library.

For example, Dart web apps generally use the dart:html library, which they can import like this:

```
import 'dart:html';
```

The only required argument to `import` is a URI specifying the library. For built-in libraries, the URI has the special `dart:` scheme. For other libraries, you can use a file system path or the `package:` scheme. The `package:` scheme specifies libraries provided by a package manager such as the pub tool. For example:

```
import 'package:test/test.dart';
```

ⓘ **Note:** *URI* stands for uniform resource identifier. *URLs* (uniform resource locators) are a common kind of URI.

Specifying a library prefix

If you import two libraries that have conflicting identifiers, then you can specify a prefix for one or both libraries. For example, if library1 and library2 both have an Element class, then you might have code like this:

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;

// Uses Element from lib1.
Element element1 = Element();

// Uses Element from lib2.
lib2.Element element2 = lib2.Element();
```

Importing only part of a library

If you want to use only part of a library, you can selectively import the library. For example:

```
// Import only foo.
import 'package:lib1/lib1.dart' show foo;

// Import all names EXCEPT foo.
import 'package:lib2/lib2.dart' hide foo;
```

Lazily loading a library

Deferred loading (also called *lazy loading*) allows a web app to load a library on demand, if and when the library is needed. Here are some cases when you might use deferred loading:

- To reduce a web app's initial startup time.
- To perform A/B testing—trying out alternative implementations of an algorithm, for example.
- To load rarely used functionality, such as optional screens and dialogs.

⚠ **Only `dart compile js` supports deferred loading.** Flutter and the Dart VM don't support deferred loading. To learn more, see issue #33118 and issue #27776.

To lazily load a library, you must first import it using `deferred as`.

```
import 'package:greetings/hello.dart' deferred as hello;
```

When you need the library, invoke `loadLibrary()` using the library's identifier.

```
Future<void> greet() async {
  await hello.loadLibrary();
  hello.printGreeting();
}
```

In the preceding code, the `await` keyword pauses execution until the library is loaded. For more information about `async` and `await`, see asynchrony support.

You can invoke `loadLibrary()` multiple times on a library without problems. The library is loaded only once.

Keep in mind the following when you use deferred loading:

- A deferred library's constants aren't constants in the importing file. Remember, these constants don't exist until the deferred library is loaded.
- You can't use types from a deferred library in the importing file. Instead, consider moving interface types to a library imported by both the deferred library and the importing file.
- Dart implicitly inserts `loadLibrary()` into the namespace that you define using `deferred as` *namespace*. The `loadLibrary()` function returns a `Future`.

The `library` directive

To specify library-level doc comments or metadata annotations, attach them to a `library` declaration at the start of the file.

```
/// A really great test library.
@TestOn('browser')
library;
```

Implementing libraries

See Create Packages for advice on how to implement a package, including:

- How to organize library source code.
- How to use the `export` directive.
- When to use the `part` directive.
- How to use conditional imports and exports to implement a library that supports multiple platforms.

‹ Metadata Keywords ›