Libraries & imports

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.

ⓘ 提示

To learn why Dart uses underscores instead of access modifier keywords like `public` or `private`, consult 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:

```dart
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:

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

ⓘ 提示

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:

```dart
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:

```dart
// 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. Use deferred loading when you want to meet one or more of the following needs.

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

That doesn't mean Dart loads all the deferred components at start time. The web app can download deferred components via the web when needed.

The `dart` tool doesn't support deferred loading for targets other than web. If you're building a Flutter app, consult its implementation of deferred loading in the Flutter guide on deferred components.

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

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

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

```dart
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.

```dart
/// 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.