

# Dart

Libraries and packages

Carmelo Villegas Cruz

# Entendiendo el concepto. Library



**Libraries** son una forma de estructurar un proyecto basado en **modularidad**.

Estas permiten al desarrollador dividir el código en múltiples archivos.

Haciendo uso de ella podemos compartir parte de su código (módulo) con otros desarrolladores.

# Libraries en Dart



Al igual que en muchos lenguajes de programación Dart usa “libraries” para permitir al desarrollador la posibilidad de estructurar el código (modularizar).

En Dart además tienen otro importante rol.

¿Cuál es ese nuevo rol?

Determinar que es visible o no para otras libraries.

# Importar y usar libraries en Dart



¿Cómo importar una *library*?

Para ello tenemos que usar la palabra reservada “*import*”

```
//Hemos definido la clase Person en el archivo  
//person_lib.dart
```

```
import 'person_lib.dart'  
void main() {  
    Person person = Person('Obi-Wan', 'Kenobi');  
}
```

# Importar y usar libraries en Dart



Dart

En el ejemplo anterior hemos importado el fichero 'person\_lib.dart' por lo que podíamos usar cualquier código en él.

A veces cuando importamos una **library** no necesitamos/usamos todo su contenido.

En Dart para hacer el código más limpio y menos propenso a errores y evitar conflictos en el nombrado, podemos usar `show`

# Importing show and hide



En muy pocas ocasiones cuando importamos una librería hacemos uso de todas la funcionalidad(clases) que nos ofrece.

¿Podríamos filtrar las clases que importamos?

Sí, lo indicamos con **show**

```
// import 'person_lib.dart' show Person, Student;
```

Podemos también indicar que NO importar:

```
// import 'person_lib.dart' hide Student;
```



# Usando prefijos

No existen **namespace** ni nada similar que identifique una **library** en un contexto.

Esto nos podría ocasionar algún quebradero de cabeza debido a los conflictos que podrían aparecer en el nombrado de variables.

Con el uso **show** and **hide** se puede reducir el riesgo pero aún seguiría existiendo.

# Usando prefijos



Dart

Library a

Person  
class

Library b

Person  
class

**main.dart**

```
import 'a.dart';
import 'b.dart';

Person p = Person();
```

¿Qué clase persona  
usa?



# Usando prefijos

Para evitar la ambigüedad anterior Dart propone una solución, prefijos.

Para definir un prefijo usamos **as**, esta se emplea después de añadir un **import**.

```
import 'a.dart' as libA;
import 'b.dart' as libB;
void main () {
    libA.Person jedi = libA.Person('Obi-Wan', 'Kenobi');
    print("Person A: ${jedi.fullName}");
}
```

# Usando prefijos



Sin el uso del prefijo no hubiéramos podido distinguir a qué clase Person estábamos intentando instanciar.

Si usamos prefijos tenemos que usarlo para todas las llamadas a miembros de la **library**.

```
import 'a.dart' as libA;  
main() {  
    //Para usar cualquier miembro dentro de a.dart es  
    //necesario usar el prefijo libA  
    libA.LoQueSea obj = libA.LoQueSea();  
}
```

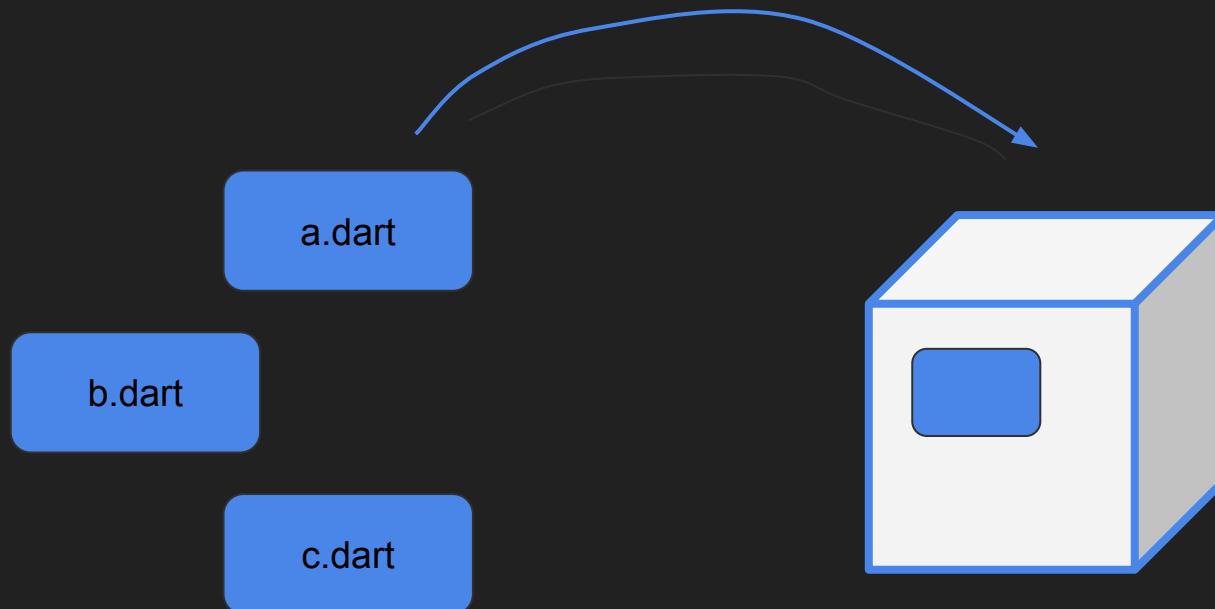
# Usando rutas



Tenemos que tener en cuenta la ubicación de la *library* a la hora de importarla. Podemos acceder a ella usando:

- Ruta relativa: `import 'directorio/a.dart'`
- Ruta absoluta: `import 'file:///c:/directorio/a.dart'`.  
Dependerá del sistema operativo en el que estemos trabajando.
- Desde una URL: `import 'http://www.iessaladillo.es/directorio/a.dart'`
- Desde un paquete.  
`import 'package:my_package/directorio/a.dart'`

# Packages



Ficheros fuente

Dart package

# Packages



La principal ventaja de usar y crear paquetes es que el código puede ser reusado y compartido.

En el ecosistema de un proyecto Dart el manejo y configuración de los paquetes se realiza a través de la herramienta ***pub***.

Esta nos permite tanto recibir (pull) como enviar (send) dependencias de pub.dartlang.org website y repositorio.

# Packages



En Dart nos podemos encontrar con dos tipos de *packages*

- **Application packages.**

No todo los paquetes se crean para ser compartidos.

Pueden tener dependencias de 'library' packages pero no están pensados para ser usado como dependencia para otros.

- **Library packages.**

Contiene código útil para otros. Se puede usar como dependencia y tener dependencias de otros.



# Estructura de un paquete

Haciendo uso de la herramienta pub que debe ser instalada a través del SDK de Dart podemos generar un proyecto.

En el caso de tener instalado Flutter no será necesario instalar Dart debido a que este es integrado dentro del SDK de Flutter.

Vamos a crear desde la línea de comando una aplicación Dart para así poder analizar su estructura.



Dart

# Estructura de un paquete

```
//Creamos un proyecto  
> dart create -t console-full my_app
```

# Estructura de un paquete



Resource Manager

1: Project

The screenshot shows a project structure in a code editor. At the top is a 'Project' dropdown and a '+' button. Below it is a tree view:

- dart\_package** (~/Developer/te)
  - .idea
  - bin
  - lib
    - dart\_package.dart
  - test
- .gitignore
- analysis\_options.yaml
- CHANGELOG.md
- pubspec.yaml
- README.md

**pubspec.yaml** -> Describe el paquete para el repositorio público.

**lib** -> Este es el lugar donde se ubican las libraries. Todo lo que pongamos en este directorio estará públicamente disponible para otros paquetes. Se conoce también como el “package public API”



# El fichero pubspec

Sirve para configurar y definir los aspectos más relevantes de un paquete en Dart.

Está basado en yaml, es un formato estándar usado sobretodo en archivo de configuración. Tiene una estructura fácil de leer y seguir.

En la siguiente diapositiva veremos un ejemplo.

Para más información: <https://dart.dev/tools/pub/pubspec>

# El fichero pubspec



```
name: dart_package
description: A sample command-line application.
# version: 1.0.0
# homepage: https://www.example.com

environment:
  sdk: '>=2.8.1 <3.0.0'

#dependencies:
#  path: ^1.7.0

dev_dependencies:
  pedantic: ^1.9.0
  test: ^1.14.4
```

# Dependencias de terceros



Durante el desarrollo de nuestra aplicación nos podemos encontrar con la necesidad de usar paquetes de terceros.

¿Cómo procedo para incluirlos en mi proyecto?

¿Lo descargo de internet y lo añado a mi proyecto?

¿Lo añado directamente a la carpeta lib y espero a que mi IDE lo reconozca?

Solo basta con definir la dependencia en nuestro fichero de configuración pubspec e invocar el comando [flutter/dart] pub get desde la carpeta que contiene dicho archivo.

# Dependencias de terceros



```
//vamos añadir a nuestro proyecto un paquete que nos permita  
//validar emails.
```

```
//https://pub.dev/packages/email\_validator
```

```
//En el fichero pubspec añadimos:
```

```
dependencies:
```

```
  email_validator: '^2.0.1'
```

```
//Desde la línea de comandos y ubicados en el directorio  
//donde existe pubspec.yaml ejecutamos.
```

```
> [flutter/dart] pub get
```

# Dependencias de terceros



```
→ dart_package flutter pub get  
Running "flutter pub get" in dart_package... 1
```

La dependencia se ha descargado en nuestro sistema y en el proyecto se ha generado un fichero (en el caso de que no existiera) donde se registra la ubicación de este.

Al no incluir dicha dependencia en nuestro workspace estamos “aislando” nuestro código del exterior.

# Dependencias de terceros



En resumen para añadir una nueva dependencia a mi proyecto tenemos que añadir en el fichero pubspec dentro del apartado dependencies lo siguiente:

dependencies:

```
<package>: <constraints>
```

```
//Los constraints que nos podemos encontrar son para la  
//fuente de datos y la versión
```

# Constraints para la versión



Podemos definir las siguientes restricciones:

- Vacío: Si no ponemos nada, no restringimos a ninguna. Normalmente se usaría la versión más actual.

email\_validator:

```
//equivalente a email_validator: any
```

- Versión en concreto: Indicamos la versión exacta.

email\_validator: '1.0.5'

- Como mínimo: Indicamos que nos valdría cualquiera desde la versión indicada

email\_validator: '>1.0.5' //También se puede usar >=

# Constraints para la versión



- Como máximo: Al revés que la anterior, acepta como máximo la que indiquemos

```
email_validator: '<=1.0.5'
```

- Rango: Indicamos un rango.

```
email_validator: '>1.0 <1.0.5'
```

# Constraints para la versión



- Rango semántico: Como mínimo la indicada y como máximo la siguiente salto.

```
email_validator: '^1.0.0'
```

```
//Equivalente a email_validator: '>=1.0.0 <2.0.0'
```

```
email_validator: '^1.1.0'
```

```
//Equivalente a email_validator: '>=1.1.0 <1.2.0'
```

Más información en <https://dart.dev/tools/pub/versioning>

# Constraints para la fuente



Además de poder restringir por versión también podemos indicarle a nuestro archivo de configuración restricciones de la fuente desde donde se obtiene la dependencia.

Ejemplo de ello lo podemos ver en la siguiente URL:

<https://dart.dev/tools/pub/dependencies#dependency-sources>