

Interfaces gráficas. Dart & Flutter.



Dart



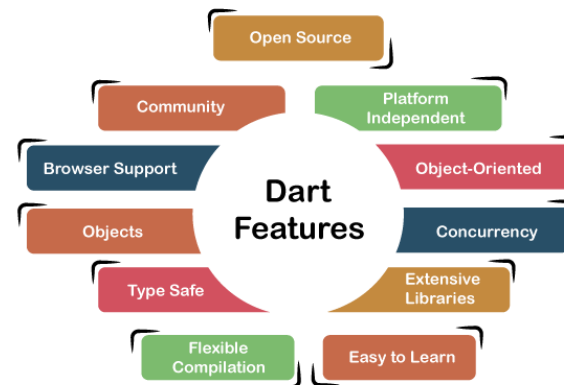
Flutter

- ▶ Índice.
 - ▶ De Java a Dart.
 - ▶ Tipos de datos básico.
 - ▶ Clases y objetos.
 - ▶ Colecciones.
 - ▶ Genéricos.
 - ▶ Programación funcional.
 - ▶ Herramienta pub.
 - ▶ Flutter.
 - ▶ Paradigmas
 - ▶ Entornos gráficos.
 - ▶ Fundamentos.
 - ▶ Widgets y eventos.
 - ▶ Sin estado y con estado.
 - ▶ Ciclo de vida.
 - ▶ Eventos.
 - ▶ Layouts
 - ▶ Navegación.
 - ▶ Recursos.

Interfaces gráficas. Dart & Flutter.

Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart.
 - ▶ Lenguaje de programación de Google.
 - ▶ Intento ser alternativa a Javascript (no lo ha conseguido).
 - ▶ Puesto 28 en la lista Tiobe (0.44% de uso).
 - ▶ Cierta popularidad con Flutter. Creación de aplicaciones.
 - ▶ Ejecución nativa en Chrome. Compilación a Javascript para otros navegadores.
 - ▶ Influido por: C#, Javascript, Java, CoffeSCript.



Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Tipos básicos.

- ▶ Number.

- ▶ Int.

- ▶ Double.

- ▶ String.

- ▶ Bool.

- ▶ Declaración implícita `Type name;`

- ▶ Declaración explícita, es capaz de conocer el tipo en la compilación.

- `var nombre=valor;`

- ▶ Tipo dinámico: Puede contener cualquier tipo de dato.

- `var nombre;`

- `nombre=4;`

- `nombre="variable dinámica";`

Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Clases y objetos.
 - ▶ Palabra reservada **class**.
 - ▶ **Uso de this solo cuando pueda llevar a confusión.**
 - ▶ Atributos.
 - ▶ Privados: Empiezan por `_`, ejemplo `int _edad`; ->Privada, `int edad` ->Pública.
 - ▶ Indicar si pueden tener valores nulos con `?` `Double? saldo`, iniciada a `null`, `double saldo=3.3`
 - ▶ Si se desea iniciar después: `later`
 - ▶ Getters y setters.
 - ▶ Para atributos privados.
 - ▶ Se puede usar la funcion `=>`
 - ▶ Palabras reservadas `get` y `set`.
 - ▶ Constructores similares a Java.
 - ▶ Permite valores por defecto, con `{ tipo nombre=valor, tipo2 nombre2=valor2 }`
 - ▶ No es necesario el operador **new** para crear objetos.
 - ▶ Constructores con nombre.

Interfaces gráficas. Dart & Flutter.

► De Java a Dart. Clases y objetos.

```
1 //definición de clase
2 class Point2D{
3     //atributos privados
4     int _x=0;
5     int _y=0;
6     //constructor con parámetros opciones
7     Point2D( {int x=10,int y=5}){
8         this._x=x;
9         this._y=y;
10    }
11    //getters y setters
12    int get x=> _x;
13    void set x(int x){
14        this._x=x;
15    }
16
17    int get y=> _y;
18    void set y(int y){
19        this._y=y;
20    }
21    //metodo to String sobrecargado
22    @override
23    String toString(){
24        return "Punto: X $_x Y: $_y";
25    }
26
27 }
```

```
28 void main() {
29     //por defecto
30     Point2D p1= Point2D();
31     print(p1);
32     //con parámetros por defecto
33     Point2D p2= Point2D(x:3);
34     print (p2);
35
36 }
```

```
Punto: X 10 Y: 5
Punto: X 3 Y: 5
```

Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Clases y objetos.

- ▶ Métodos (llamados funciones).

- ▶ Similar a Java.

- ▶ Métodos privados `_nombre`.

- `Tipo_devuelto nombre(Tipo nombre, {tipo_opcional nombre=valor,...}).`

- ▶ Parámetros por defecto con nombre.

- ▶ Valor devuelto.

- ▶ Evitar en lo posible el uso de `this`.

- ▶ No soporta la sobrecarga de métodos.

- ▶ Tipos primitivos por valor, objetos por referencia.

Interfaces gráficas. Dart & Flutter.

► De Java a Dart. Clases y objetos.

```
//método público
num center_distance(){
  num tempo=pow(_x,2)+pow(_y,2);
  return sqrt(tempo);
}
//método público al que se pasa un objeto por referencia
num distance(Point2D other){
  num tempo=pow((_x-other._x).abs(),2)+pow((_y-other.y).abs(),2);
  return sqrt(tempo);
}
//métodos con parámetros por defecto
num distance_optional({x=34, y=5}){
  num tempo=pow((_x-x).abs(),2)+pow((_y-y).abs(),2);
  return sqrt(tempo);
}
//método privado
num _private_distance(Point2D other){
  num tempo=pow((_x-other._x).abs(),2)+pow((_y-y).abs(),2);
  return sqrt(tempo);
}
void main(List<String> arguments) {
  Point2D p1 = Point2D(x:6,y:9);
  Point2D p2= new Point2D(x:5,y:5);
  print(p2.center_distance());
  print(p2.distance(p1));
  //error es método privado
  // p2._private_distance(p1);
}
```


Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Clases y objetos.
 - ▶ Herencia.
 - ▶ Palabra reservada extends.
 - ▶ Herencia simple.
 - ▶ Sin acceso a métodos y atributos privados
 - ▶ Necesario llamar a constructor por defecto. Similar a C++.
 - ▶ Sobreescritura de métodos con @override.
 - ▶ Llamar a métodos de la clase padre con super

Interfaces gráficas. Dart & Flutter.

► De Java a Dart. Clases y objetos.

```
class Person{
  String name;
  String surname;
  int _age;
  //otro tipo de constructor
  Person(this.name,this.surname,this._age);
  //get y set atributo privado
  int get age=> _age;
  set age(age)=> _age=age;
  @override
  String toString(){
    return "Name $name Surname $surname Age $age";
  }
}
```

```
void main(List<String> arguments) {
  //no es necesario el new
  Person p= Person("Paco", "Martinez Soria", 24);
  print(p);
  var e= Employee(20000, "SuperLopez", "", 30);
  p=e;
  print(p);
}
```

```
class Employee extends Person{
  double salary;
  //llamando al constructor de la clase base
  Employee(this.salary,String name,String surname,int age):super(name,surname,age);
  //sobreescritura de método del padre y llamada al mismo
  @override
  String toString(){
    return super.toString()+" Salary $salary";
  }
}
```

Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Clases y objetos.
 - ▶ Interfaces.
 - ▶ Palabra reservada implements.
 - ▶ No existen interfaces como tal, sino que se implementan clases concretas.
 - ▶ Solo se heredan la obligación de implementar los métodos, no el código.
 - ▶ Se pueden implementar de muchas clases.
 - ▶ Para que sea similar a Java se declara la clase junto con la palabra reservada interface.

```
interface class IMovable{  
    void moveLeft(){}  
    void moveRigth(){}  
    void moveUP(){}  
    void moveDown(){}  
    void move(){}  
}
```

Interfaces gráficas. Dart & Flutter.

► De Java a Dart. Clases y objetos.

```
Interface class State{
  EState _state=EState.NONE;
  EState get state=> _state;
  setDriving() => _state=EState.DRIVING;
  setRunning() => _state=EState.RUNNING;
  setWorking() => _state=EState.WORKING;
  setSleeping()=> _state=EState.SLEEPING;
}
//enumerados
enum EState {WORKING, SLEEPING, DRIVING,RUNNING, NONE}
```

```
void main(List<String> arguments) {
  State s = State();
  var e = Employee(20000, "SuperLopez", "", 30);
  print(s.state);
  print(e.state);
  e.setDriving();
  print(e.state);
}
}
```

```
class Employee extends Person implements State {
  double salary;
  EState _state = EState.NONE;
  //llamando al constructor de la clase base
  Employee(this.salary, String name, String surname,
    int age)
    : super(name, surname, age);
  //sobreescritura de método del padre y llamada al
  mismo
  @override
  String toString() {
    return super.toString() + " Salary $salary";
  }
  //al implementar se adquiere la obligación de
  implementar
  //los prototipos
  @override
  EState get state => _state;
  @override
  setDriving() => _state = EState.DRIVING;
  @override
  setRunning() => _state = EState.RUNNING;
  @override
  setWorking() => _state = EState.WORKING;
  @override
  setSleeping() => _state = EState.SLEEPING;
}
```

Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Clases y objetos.
 - ▶ Clases abstractas.
 - ▶ Mismo concepto que el Java u otros lenguajes.
 - ▶ No se pueden instanciar.
 - ▶ Uso palabra reservada `abstract` junto a `class`.
 - ▶ Puede tener métodos solo con el prototipo o con la implementación.
 - ▶ Las clases que extiendan de una clase abstracta han de implementar los métodos no implementados, con la anotación `@override`.
 - ▶ Se pueden usar para simular interfaces.

Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Clases y objetos.

```
cl
```

```
cla
```

```
vo
```

Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Clases y objetos.
 - ▶ Mixins.
 - ▶ Añadir comportamiento y funcionalidad sin necesidad de herencia directa.
 - ▶ Clase que proporciona métodos y propiedades que se pueden utilizar en otras clases sin requerir una relación de herencia directa

Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Clases y objetos.

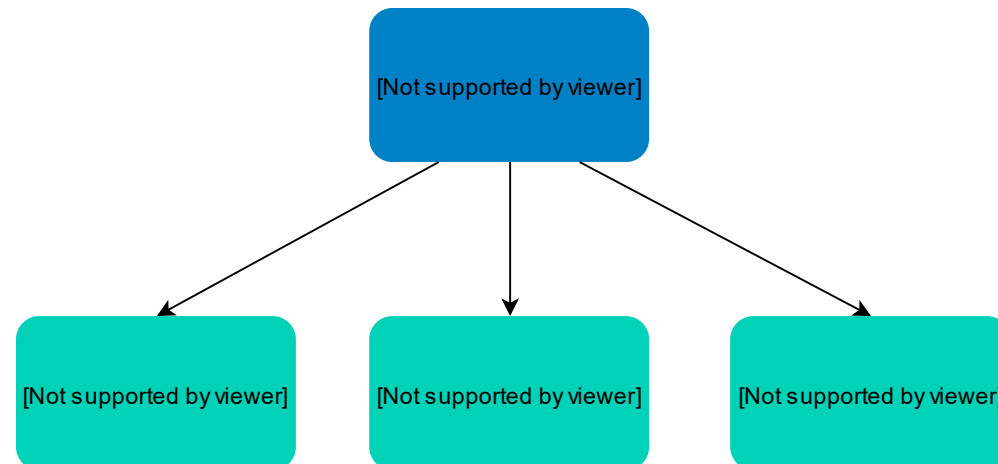
```
cl
```

```
cla
```

```
vo
```


Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Colecciones.
 - ▶ Núcleo del lenguaje.
 - ▶ Paquete/librería `dart:core` library.
 - ▶ `List`.
 - ▶ Constructor: `List.empty()`, `[]`, `[element1, element2, ...]`.
 - ▶ `Map`.
 - ▶ `Set`.
 - ▶ Constructor `Set()`, `{ }`, `{element1, element2...}`



Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Colecciones.

- ▶ Paquete/librería dart:collection library.

- ▶ Multitud de colecciones:

- ▶ DoubleLinkedList HashMap HashSet LinkedHashMap LinkedList
 - ▶ ListBase ListQueue MapView Queue SetBase
 - ▶ SplayTreeMap SplayTreeSet UnmodifiableListView UnmodifiableMapView

- ▶ Operaciones similares a las vistas en Java.

- ▶ Ejemplo LinkedList.

- ▶ Add addAll clear contains elementAt forEach
 - ▶ Remove toList toSet toString

- ▶ Posibilidad indicar el tipo de dato (clase) a almacenar como en Java <tipo_dato>

Interfaces gráficas. Dart & Flutter.

► De Java a Dart. Colecciones.

```
void main(List<String> arguments) {  
  HashMap<String, Employee> company = HashMap();  
  var e = Employee(20000, "SuperLopez", "", 30);  
  company[e.surname] = e;  
  e = Employee(21000, "Luisa", "Lanas", 28);  
  company[e.surname] = e;  
  e = Employee(25000, "Jefe", "El", 50);  
  company[e.surname] = e;  
  e = Employee(20000, "General", "Sintacha", 28);  
  company[e.surname] = e;  
  company.forEach((key, value) {  
    print("$key->$value");  
  });  
}
```

```
Sintacha->Name General Surname Sintacha Age 28 Salary 20000.0  
Lanas->Name Luisa Surname Lanas Age 28 Salary 21000.0 ->Name  
SuperLopez Surname Age 30 Salary 20000.0 El->Name Jefe Surname El  
Age 50 Salary 25000.0
```

Interfaces gráficas. Dart & Flutter.

► De Java a Dart. Genéricos.

- Similar a Java.
- Convección de nombres:
 - E-> Element.
 - K-> Key.
 - R-> Return.
 - T-> Type.

```
void main(List<String> arguments) {  
  Store<Employee> store;  
  store = Store();  
  var e = Employee(20000, "SuperLopez", "Heroe", 30);  
  store.add(e);  
  e = Employee(21000, "Luisa", "Lanas", 28);  
  store.add(e);  
  e = Employee(25000, "Jefe", "El", 50);  
  store.add(e);  
  e = Employee(20000, "General", "Sintacha", 28);  
  store.add(e);  
  store.next();  
  print(store.getActual());  
}
```

```
class Store<E> {  
  
  //puede ser nulo  
  List<E>? _elements;  
  int _actual = 0;  
  
  Store() {  
    //creacion de la lista  
    _elements = [];  
  }  
  
  void add(E item) {  
    _elements?.add(item);  
  }  
  
  void removeAt(int index) {  
    _elements?.removeAt(index);  
  }  
  
  void next() {  
    _actual++;  
    if (_actual >= _elements!.length) {  
      _actual = 0;  
    }  
  }  
  
  //puede ser nulo  
  E? getActual() {  
    return _elements?.elementAt(_actual);  
  }  
}
```

Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Programación funcional.

- ▶ Uso del operador `=>`.

- ▶ Sintaxis:

- ▶ `Return_type name(parameters) => { body }`

- ▶ `Return_type name(parameters) => simple_body`

- ▶ Ejemplo: `int ShowSum(int numOne, int numTwo) => numOne + numTwo;`

- ▶ Apuntar función lambda por una variable:

- `var predicado= (Employee e)=> e.salary!=0;`

- `predicado= (e) { e.salary!=0`

- ▶ Variables tipo función:

- `bool Function(Employee e) pred;`

- `pred= (e) { return e.salary<10000;;`

- `pred=(e) => e.salary<5000;`

- ▶

Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Programación funcional.
 - ▶ Uso intensivo en las colecciones.
 - ▶ Funciones lambda como parámetro.
 - ▶ Ejemplos:

```
//empleados que ganan más de 22000
var sueldo = 22000;
bool Function(Employee e) pred;
pred = (e) {
    return e.salary > sueldo;
};
company.where(pred).forEach((element) {
    print(element);
});
```

```
//obtener los apellidos y nombre ordenados alfabéticamente por
apellido
var l = company.map((e) => "${e.surname}, ${e.name}").toList();
l.sort((a, b) => a.compareTo(b));
l.forEach((element) {
    print(element);
});
```

Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart. Herramienta pub.
 - ▶ Similar a Maven.
 - ▶ Escrito en formato yaml.
 - ▶ Fichero pubspec.xml
 - ▶ Administrador de paquetes.
 - ▶ Gestión de plataformas.
 - ▶ Versiones del sdk.
 - ▶ Repositorio en <https://pub.dev/>
 - ▶ Comando pub.
 - ▶ get
 - ▶ Upgrade
 - ▶ publish...

```
name: dart_1
description: A sample command-line application.
version: 1.0.0
# repository: https://github.com/my_org/my_repo

environment:
  sdk: ^3.0.2

# Add regular dependencies here.
dependencies:
  # path: ^1.8.0

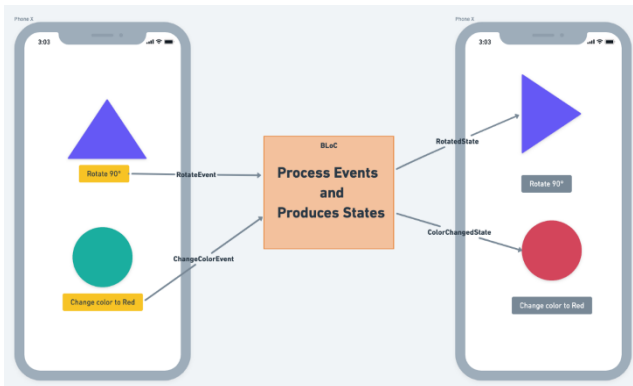
dev_dependencies:
  lints: ^2.0.0
  test: ^1.21.0
```

Interfaces gráficas. Dart & Flutter.

- ▶ De Java a Dart.
- ▶ Flutter.
 - ▶ Paradigmas
 - ▶ Entornos gráficos.
 - ▶ Fundamentos.
 - ▶ Widgets y eventos.
 - ▶ Sin estado y con estado.
 - ▶ Ciclo de vida.
 - ▶ Eventos.
 - ▶ Layouts
 - ▶ Navegación.
 - ▶ Recursos

Interfaces gráficas. Dart & Flutter.

- ▶ Flutter. Paradigmas en las interfaces gráficas.
 - ▶ Imperativo. Se produce un cambio y se indica que hacer. Ejemplos: Android SDK, iOS UIKit.
 - ▶ Declarativa. Se produce un cambio y no se indica que hacer, sino lo que se desea. Ejemplos: SQL, programación funcional.
 - ▶ Reactiva. Los componentes responden ante cambios en otros elementos de forma independiente , sin necesidad de hacerlo de forma implícita. Más conocido: "React" de Meta (Facebook). Basado en patrón observer.
 - ▶ En flutter:
 - ▶ Solo se describe de la interfaz de usuario y el framework se encarga de usar esa configuración para crear o actualizar la interfaz de usuario según corresponda..
 - ▶ Un cambio en el estado produce un evento, que a su vez puede producir una reconstrucción de todo o parte del árbol de componentes.



$$\text{UI} = f(\text{state})$$

The layout on the screen Your build methods The application state

Interfaces gráficas. Dart & Flutter.

▶ Flutter. Entornos gráficos (GUI).

- ▶ Facilidad de uso.
- ▶ Basado en componentes: Clases y objetos.
- ▶ Eventos.
- ▶ Evolución.
 - ▶ Java: AWT, Swing, JavaFX.
 - ▶ C++: Windows, Qt, Gtk.
 - ▶ Resto de lenguajes: "Ports" de las anteriores o minoritaria.

▶ Actualidad:

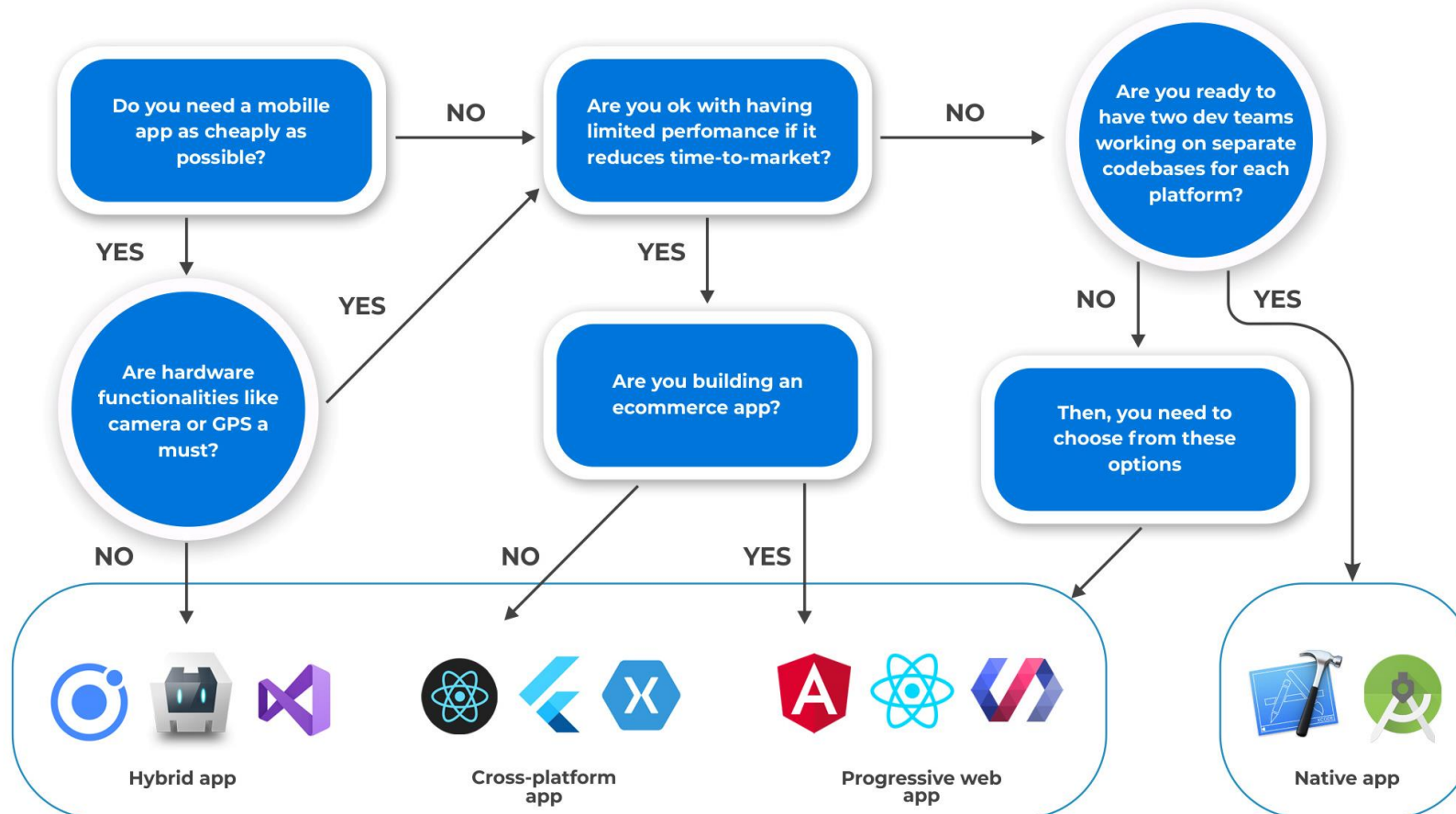
- ▶ Necesidad multiplataforma.
- ▶ Tecnologías similares a la web.
- ▶ Desarrollo nativo. iOS con Swift, Android con Java/Kotlin, Windows con C#
- ▶ Desarrollo con un SDK y compilación a nativa (Xamarin, ReactNative, Flutter).
- ▶ Desarrollo con tecnologías web (HTML, Javascript y CSS)+ librerías para acceso SO. (Ionic, Phonegap)



Interfaces gráficas. Dart & Flutter.

- ▶ Flutter. Entornos gráficos (GUI).

CHOOSE A DEV APPROACH FOR YOUR MOBILE APP



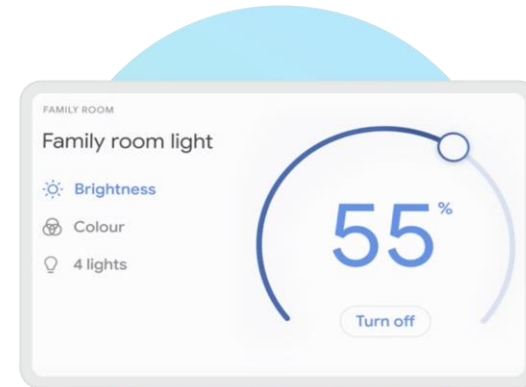
Interfaces gráficas. Dart & Flutter.

- ▶ Flutter. Entornos gráficos (GUI).
 - ▶ Debate sobre tipos de aplicaciones.
 - ▶ ¿Se puede usar cualquier tecnología para cualquier aplicación?
 - ▶ En caso de que sea necesario el procesamiento de gran cantidad de datos, como puede ser vídeo ¿Qué tipo de tecnología usar?
 - ▶ En caso de tener poco tiempo para buscar un grupo de desarrollo para crear una aplicación que va a tener que funcionar en múltiples dispositivos. ¿Qué tipo de tecnología usar?
 - ▶ ¿Y si se dispone de más tiempo?
 - ▶ De las tecnologías más usadas, ¿Cuáles se utilizan en el desarrollo de aplicaciones gráficas?
<https://survey.stackoverflow.co/2022/#technology-most-popular-technologies>.
 - ▶ Buscar en los portales de empleo infojobs y monster.uk los empleos relacionados con:
 - ▶ Flutter.
 - ▶ React Native.
 - ▶ Kotlin
 - ▶ Ionic.
 - ▶ ¿Qué conclusión se extrae?

Interfaces gráficas. Dart & Flutter.

► Flutter. Fundamentos.

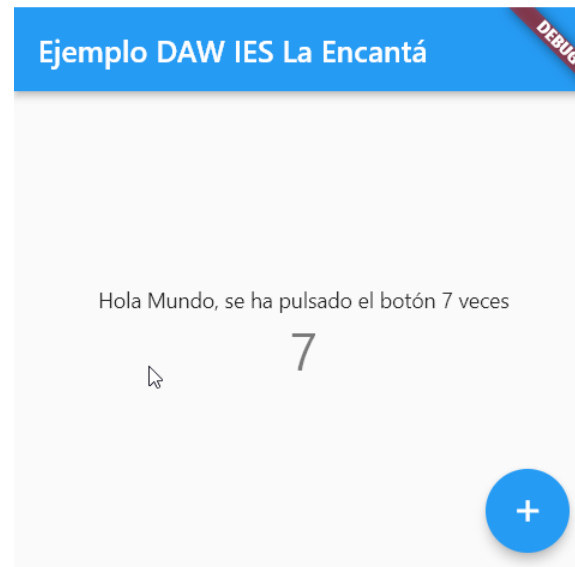
- Framework para la creación de aplicaciones multiplataforma.
- Bajo nivel: C/C++.
- Alto nivel: Dart.
- Compilación para diferentes plataformas: Android, ios, Windows, Web, Linux, Sistemas embebidos (vehículos, dispositivos ARM...)
- Necesario compilador y/o SDK para la plataforma nativa.



Interfaces gráficas. Dart & Flutter.

► Flutter.

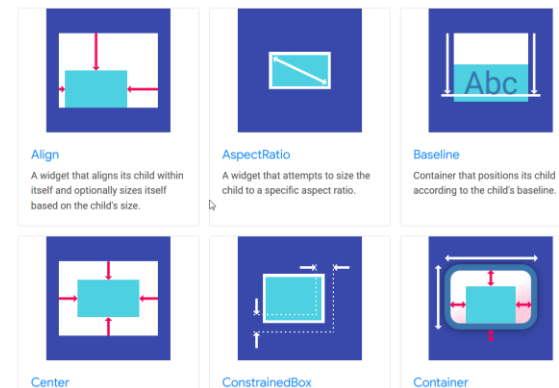
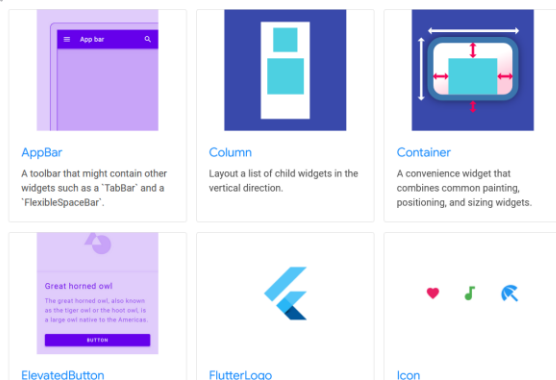
Práctica 1. Instalando Flutter .



Interfaces gráficas. Dart & Flutter.

- ▶ Flutter. Widgets y estados.
 - ▶ Las interfaces gráficas se crean a partir de componentes(objetos de clases concretas) con atributos y que a su vez contienen otros componentes.
 - ▶ Cada tecnología establece como se definen, crean o relacionan estos componentes.
 - ▶ Cada "framework" o librería define un conjunto de componentes propios, aunque existen muchos comunes como cuadros de texto, botones, cuadros de selección...
 - ▶ En Flutter se denominan "Widgets", y su **estructura es un árbol**. (Al igual que HTML)
 - ▶ Se puede consultar la lista completa en <https://docs.flutter.dev/ui/widgets>

See more widgets in the [widget catalog](#).

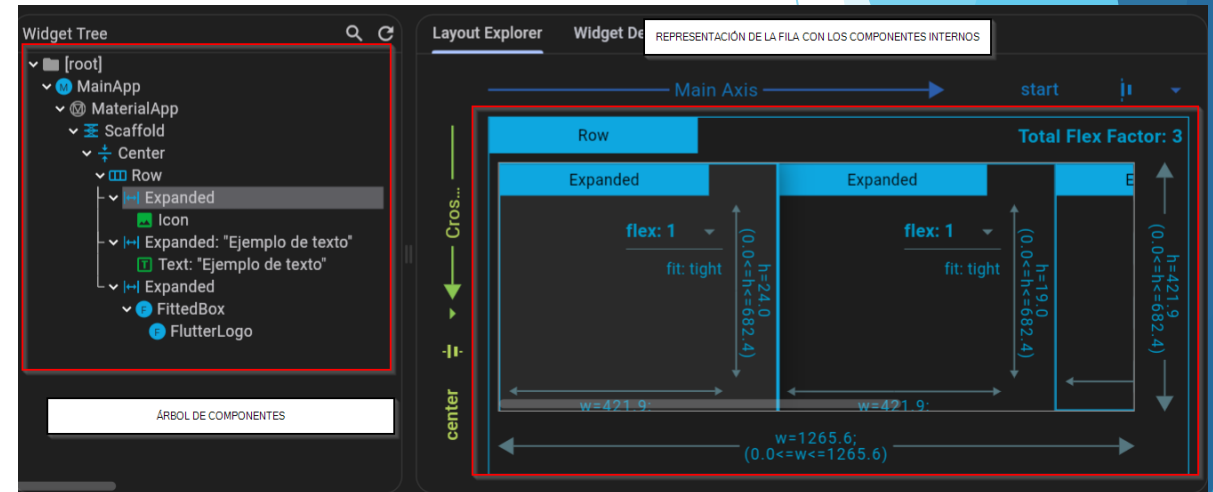


Interfaces gráficas. Dart & Flutter.

► Flutter. Widgets y estados.

► Ejemplo:

```
class MainApp extends StatelessWidget {  
  const MainApp({super.key});  
  @override  
  Widget build(BuildContext context) {  
    return const MaterialApp(  
      home: Scaffold(  
        body: Center(  
          child: Row(  
            children: <Widget>[  
              Expanded(  
                child: Icon(  
                  Icons.favorite,  
                  color: Colors.pink,  
                  size: 24.0,  
                  semanticLabel: 'Icono texto',  
                )),  
              Expanded(  
                child: Text('Ejemplo de texto', textAlign: TextAlign.center),  
              ),  
              Expanded(  
                child: FittedBox(  
                  child: FlutterLogo(),  
                ),  
              ),  
            ],  
          ),  
        ),  
      ),  
    );  
  }  
}
```



Interfaces gráficas. Dart & Flutter.

▶ Flutter. Widgets y estados.

- ▶ El árbol de componentes cambia de forma automática (programación reactiva) al producirse cambios en el estado de la aplicación.
- ▶ Todo componente hereda de dos clases:
 - ▶ **StatelessWidget**. Componente sin estado, no responde a los cambios, se crea, se añade al árbol y no se modifica en la ejecución del programa. Útil para crear componentes personalizados que contienen otros que si han de responder y cambiar. Ejemplo: El componente de la aplicación principal.
 - ▶ **StatefulWidget**. Ha de implementar el método `<Element extends State> createState()`, que devuelve un objeto de una clase que a su vez hereda de la clase `State`.
 - ▶ Al heredar de la clase `State` se ha de implementar el método `Widget build(BuildContext context)`.

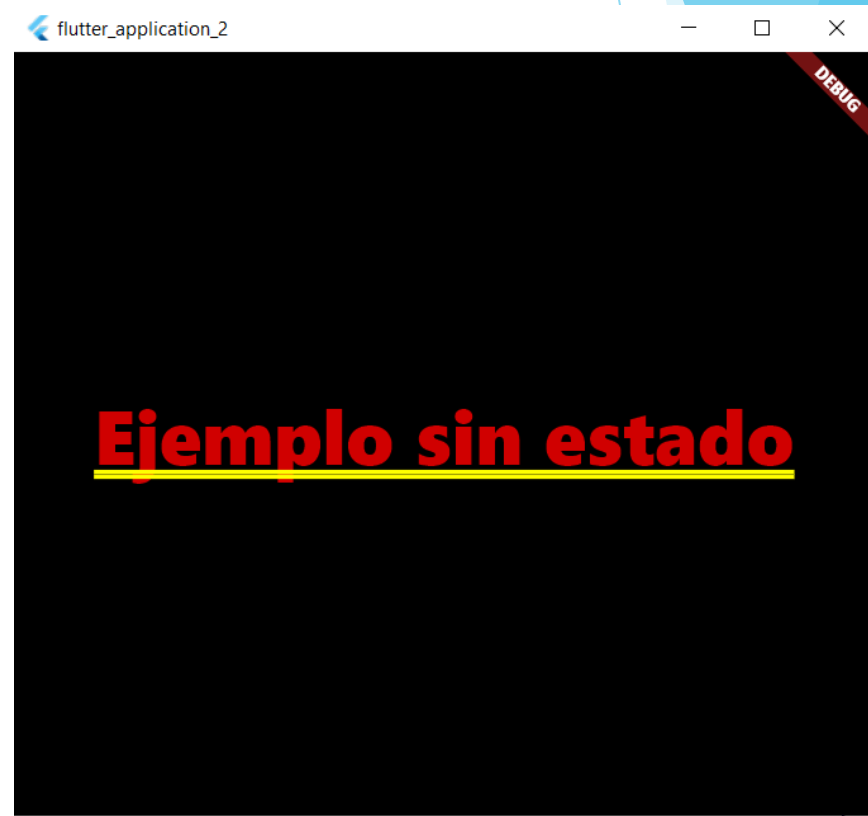
Everything is a Widget



Interfaces gráficas. Dart & Flutter.

► Flutter. Widgets y estados.

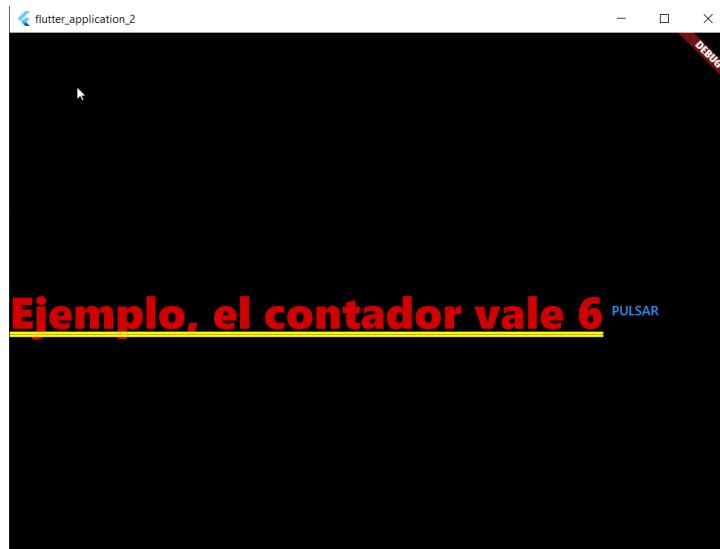
```
void main() {  
  runApp(const MyApp());  
}  
  
class MyApp extends StatelessWidget {  
  const MyApp({super.key});  
  
  // This widget is the root of your application.  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Flutter Demo',  
      theme: ThemeData(  
        primarySwatch: Colors.blue,  
      ),  
      home: Container(  
        alignment: Alignment.center,  
        child: const Text("Ejemplo sin estado"),  
      );  
    }  
  }  
}
```



Interfaces gráficas. Dart & Flutter.

► Flutter. Widgets y estados.

```
//es el widget con estado
class WidgetEstado extends StatefulWidget {
  @override
  EjemploEstado createState() {
    return EjemploEstado()
  }
}
```



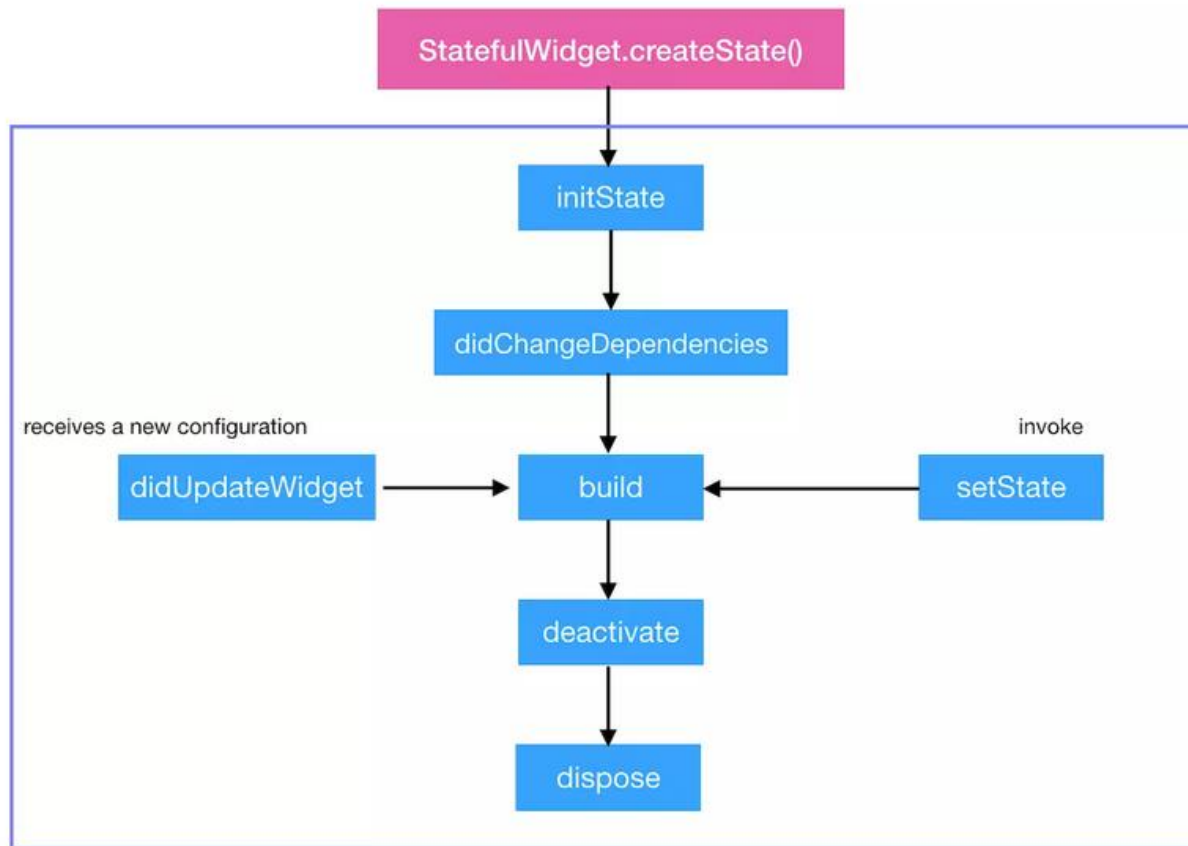
```
//representa el estado
class EjemploEstado extends State<WidgetEstado> {
  //atributo interno del estado
  int _contador = 0;
  //escuchador del evento que se "dispara" al pulsar el botón
  void _escuchadorevento() {
    //actualización del estado
    setState(() {
      //se modifica el atributo
      _contador++;
    });
  }
  @override
  Widget build(BuildContext context) {
    return Row(
      children: [
        //de forma automática (reactiva)
        Text("Ejemplo, el contador vale $_contador"),
        TextButton(
          //escuchador del evento
          onPressed: _escuchadorevento,
          child: Text("PULSAR"))
      ],
    );
  }
}
```

Interfaces gráficas. Dart & Flutter.

- ▶ Flutter. Widgets y estados. Ciclo de vida.
 - ▶ Un componente con estado pasa por diferentes fases en la vida del mismo (del objeto).
 - ▶ `CreateState()`. Se ejecuta una vez, se crea el objeto y el estado.
 - ▶ `InitState()`. Se llama al añadirlo al árbol de componentes.
 - ▶ `DidChangeDependencies()`. Se llama la primera vez que se ejecuta `InitState()`
 - ▶ `Build()`. Se construye o reconstruye el componente. Se ejecuta al cambiar el estado.
 - ▶ `didUpdateWidget(Widget oldWidget)`. Si el padre cambia sus propiedades y se desea reconstruir sus hijos.
 - ▶ `SetState()`. Notificar al framework un cambio en el estado y que se ha de reconstruir (método `build()`)
 - ▶ `deactivate()`. Cuando se separa del árbol, pero puede volver a insertarse, por ejemplo, cambiar de pantalla y volver a la anterior.
 - ▶ `dispose()`. Eliminación permanente del árbol. Se tiene que borrar de la suscripciones...liberar recursos.
- ▶ Estos métodos se pueden sobrescribir, siendo el más común **Build()**.

Interfaces gráficas. Dart & Flutter.

- ▶ Flutter. Widgets y estados. Ciclo de vida.

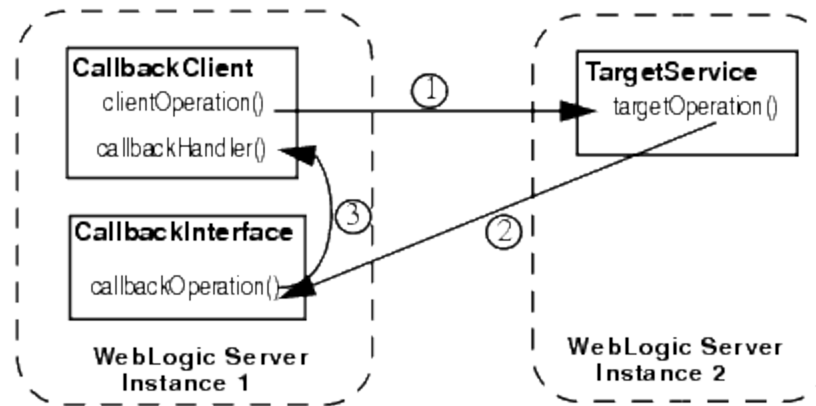


Interfaces gráficas. Dart & Flutter.

► Flutter. Widgets y estados. Eventos.

► Diferentes tipos:

- Cada componente puede tener un conjunto de eventos, a los que se asocia un escuchador de eventos (método), aquí llamados "CallBack".
- Widget Listener.
- GestureDetector.
- Notificaciones. Similar a la gestión de eventos en los JavaBeans.



Interfaces gráficas. Dart & Flutter.

► Flutter. Widgets y estados. Eventos.

- Cada componente puede tener un conjunto de eventos, a los que se asocia un escuchador de eventos (método), aquí llamados "CallBack".
- Depende de la naturaleza del componente.
- Consultar la documentación para conocer los eventos.

key → *Key*?

Controls how one widget replaces another widget in the tree.

final inherited

onFocusChange → *ValueChanged<bool>?*

Handler called when the focus changes.

final inherited

onHover → *ValueChanged<bool>?*

Called when a pointer enters or exits the button response area.

final inherited

onLongPress → *VoidCallback?*

Called when the button is long-pressed.

final inherited

onPressed → *VoidCallback?*

Called when the button is tapped or otherwise activated.

final inherited

Eventos TextButton

onKeyEvent → *ValueChanged<KeyEvent>?*

Called whenever this widget receives a keyboard event.

final

Eventos Teclado

onChanged → *VoidCallback?*

Called when one of the form fields changes.

final

onWillPop → *WillPopCallback?*

Enables the form to veto attempts by the user to dismiss the *ModalRoute* that contains the form.

final

Eventos formulario

```
TextField(  
  onChanged: (value) => {  
    print("Ha cambiado $value")  
  },  
  onTap: () => {  
    print("click/presionado ")  
  },  
  onTapOutside: (event) => {  
    print("Click fuera $event")  
  },  
)
```

Interfaces gráficas. Dart & Flutter.

► Flutter. Widgets y estados. Eventos.

► Widget Listener.

► Bajo nivel.

- Si tiene un hijo, este widget se remite al hijo para determinar el comportamiento del tamaño. Si no tiene un hijo, crece para adaptarse al padre.

`onPointerCancel` → `PointerCancelEventListener?`

Called when the input from a pointer that triggered an `onPointerDown` is no longer directed toward

`final`

`onPointerDown` → `PointerDownEventListener?`

Called when a pointer comes into contact with the screen (for touch pointers), or has its button pre:

`final`

`onPointerHover` → `PointerHoverEventListener?`

Called when a pointer that has not triggered an `onPointerDown` changes position.

`final`

`onPointerMove` → `PointerMoveEventListener?`

Called when a pointer that triggered an `onPointerDown` changes position.

`final`

`onPointerPanZoomEnd` → `PointerPanZoomEndEventListener?`

Called when a pan/zoom finishes.

`final`

`onPointerPanZoomStart` → `PointerPanZoomStartEventListener?`

Called when a pan/zoom begins such as from a trackpad gesture.

`final`

`onPointerPanZoomUpdate` → `PointerPanZoomUpdateEventListener?`

Called when a pan/zoom is updated.

`final`

`onPointerSignal` → `PointerSignalEventListener?`

Called when a pointer signal occurs over this object.

`final`

`onPointerUp` → `PointerUpEventListener?`

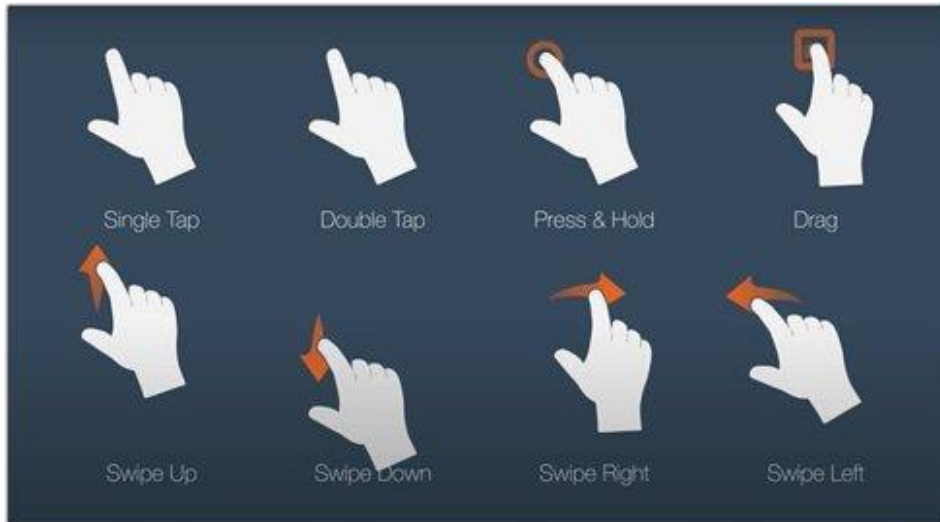
Called when a pointer that triggered an `onPointerDown` is no longer in contact with the screen.

`final`

```
Listener(  
  onPointerDown: (event) => {  
    print("puntero pulsado"),  
    print("X $event.position.dx Y: $event.position.dy")  
  },  
  child: Text(  
    '$_counter',  
    style: Theme.of(context).textTheme.headlineMedium,  
  )),
```


Interfaces gráficas. Dart & Flutter.

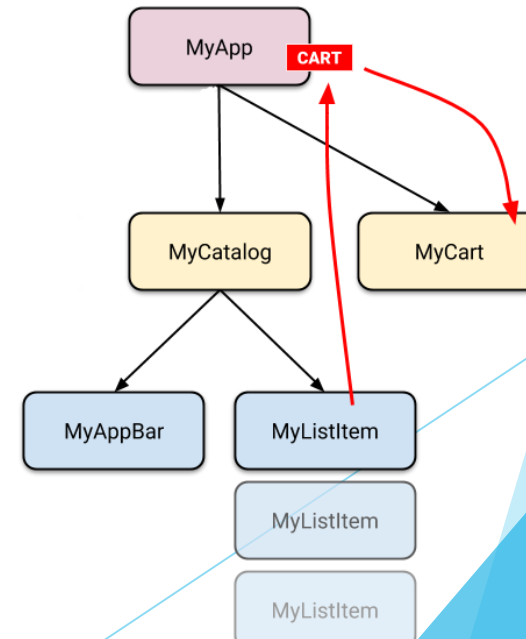
- ▶ Flutter. Widgets y estados. Eventos.
 - ▶ GestureDetector.
 - ▶ Alto nivel.
 - ▶ Gestos. Procesados, aporta más información. onTapDown, onTapUp, onDoubleTap, onVerticalDragEnd.
 - ▶ Diversidad de dispositivos.
 - ▶ Similar a Listener, pero procesamiento previo.



```
GestureDetector(  
  onTap: () => {  
    print("puntero pulsado"),  
  },  
  onLongPress: () => {  
    print("Pulsación larga")  
  },  
  child: Text(  
    '$_counter',  
    style:  
      Theme.of(context).textTheme.headlineMedium,  
  ),  
)
```

Interfaces gráficas. Dart & Flutter.

- ▶ Flutter. Widgets y estados. Eventos.
 - ▶ Notificaciones. Similar a la gestión de eventos en los JavaBeans.
 - ▶ Crear eventos propios.
 - ▶ Cambios en diferentes lugares. Por ejemplo, añadir elemento a una colección y avisar a varios elementos gráficos.
 - ▶ Clase ChangeNotifier (Observable en Java)
 - ▶ ChangeNotifierProvider. Contiene al Observable, se crea en el padre y añade al contexto
 - ▶ Clase Consumer (observer en Java) que recibe la notificación.
 - ▶ Se almacena en el contexto y los hijos pueden acceder.
 - ▶ Al producirse un cambio.
 - ▶ En cualquier momento:
 - ▶ Librería externa pubspec.yaml -> `provider: ^6.0.0`



Interfaces gráficas. Dart & Flutter.

► Flutter. Widgets y estados. Eventos.

► Notificaciones. Ejemplo

```
class Dimmer extends ChangeNotifier {
  bool _state = false;
  Dimmer({state = false}) : super() {
    state = state;
  }
  void setOn() {
    this._state = true;
    notifyListeners();
  }
  void setOff() {
    this._state = false;
    notifyListeners();
  }
  void sswitch() {
    _state = !_state;
    notifyListeners();
  }
  bool get state { return _state; }
}
```

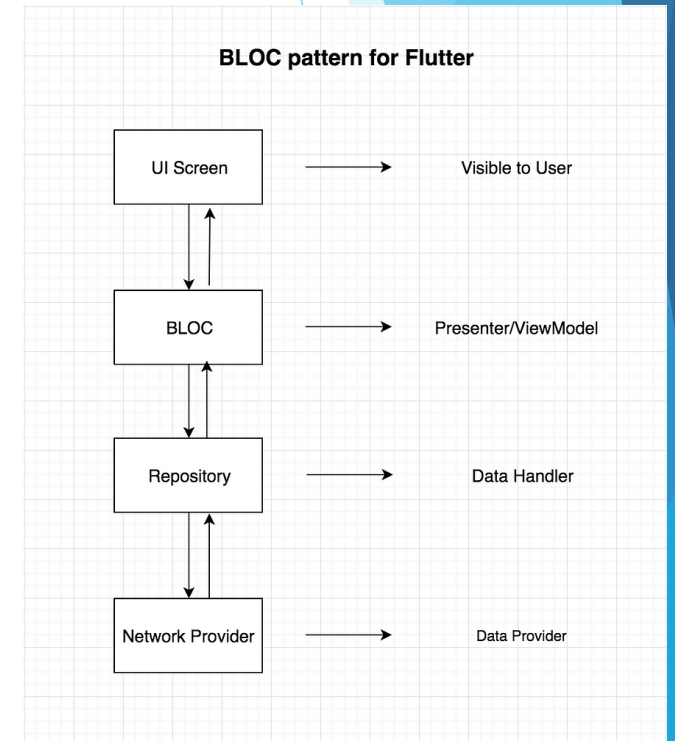
```
void main() {
  runApp(ChangeNotifierProvider(
    create: (context) => Dimmer(),
    child: const MyApp(),
  ));
}
```

```
class _MyHomePageState extends State<MyHomePage> {
  bool state = false;
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Column(
        children: [
          Light(),
          Switch(
            value: state,
            onChanged: (value) => {
              //al pulsar el boton se obtiene el contexto
              //y se cambia el estado, probando la propagación
              setState(() {
                state = value;
                value
                  ? context.read<Dimmer>().setOn()
                  : context.read<Dimmer>().setOff()
              });
            },
          ),
        ],
      ),
    );
  }
}
```

```
class Light extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Consumer<Dimmer>(
      builder: (context,
        dimmer, child) => Icon(
          Icons.lightbulb,
          color:
            dimmer.state ==
            true ?
              Colors.yellow :
              Colors.grey,
          size: 24.0,
        ));
  }
}
```

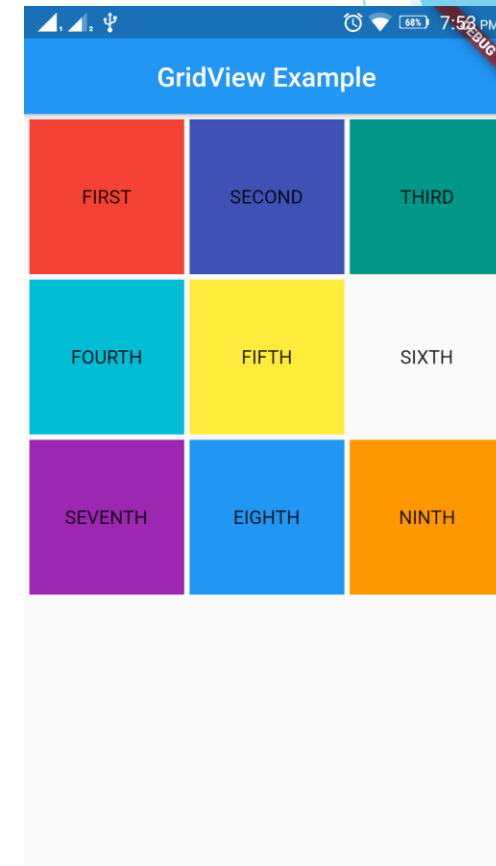
Interfaces gráficas. Dart & Flutter.

- ▶ Flutter. Widgets y estados. Eventos.
 - ▶ Otras librerías.
 - ▶ La gestión de estados y comunicación entre componentes no es una tarea sencilla.
 - ▶ Se puede realizar desde diferentes enfoques.
 - ▶ Librerías de terceros con otro enfoque:
 - ▶ Riverpod. <https://riverpod.dev>
 - ▶ SetState. Bajo nivel. El visto con less y ful
 - ▶ InheritedWidget & InheritedModel. Para comunicarse por todo el árbol.
 - ▶ Redux.
 - ▶ Block/Rx
 - ▶ ...



Interfaces gráficas. Dart & Flutter.

- ▶ Flutter. Widgets y estados. Layouts
 - ▶ Necesidad de distribuir los diferentes componentes en la aplicación.
 - ▶ Actualización de dimensiones de forma automática.
 - ▶ Múltiples dispositivos con diferentes resoluciones.
 - ▶ Widgets concretos denominados "layouts".
 - ▶ Dos tipos.
 - ▶ Solo un hijo.
 - ▶ Varios hijos.
 - ▶ Propiedades para configurar(alineación, bordes, tamaño....).
 - ▶ Similar al diseño HTML+CSS (Flexbox, div...)



Interfaces gráficas. Dart & Flutter.

► Flutter. Widgets y estados. Layouts

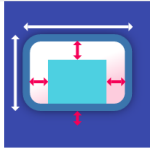
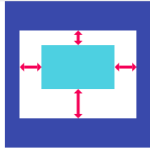
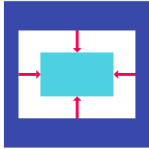
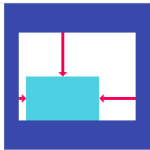


► Con un solo hijo:

► Usado normalmente para posicionarse con respecto al padre y posicionar al hijo.

► Los tipos más destacados:

- Container.
- Center.
- Padding.
- Align.
- Baseline.
- Transform.

Widgets de un solo hijo

 <p>Container</p> <p>Un widget de conveniencia que combina widgets comunes de dibujado, posicionado y dimensionado</p> <p>Documentación</p>	 <p>Padding</p> <p>Un widget que encuadra a sus hijos con el padding dado.</p> <p>Documentación</p>	 <p>Center</p> <p>Un widget que centra su hijo en sí mismo.</p> <p>Documentación</p>
 <p>Align</p> <p>Un widget que alinea a su hijo dentro de sí mismo y, opcionalmente, se dimensiona en función del tamaño del child.</p>	 <p>FittedBox</p> <p>Escala y posiciona a su hijo dentro de él de acuerdo con el ajuste.</p>	 <p>AspectRatio</p> <p>Un widget que intenta dimensionar al hijo a una relación de aspecto específica.</p>

Interfaces gráficas. Dart & Flutter.

▶ Flutter. Widgets y estados. Layouts







▶ Con varios hijos:

▶ Gestiona algunas características y distribución de los hijos.

▶ Los tipos más destacados:

- ▶ Row.
- ▶ Column.
- ▶ GridView.
- ▶ Table.
- ▶ ListView.
- ▶ Expanded.

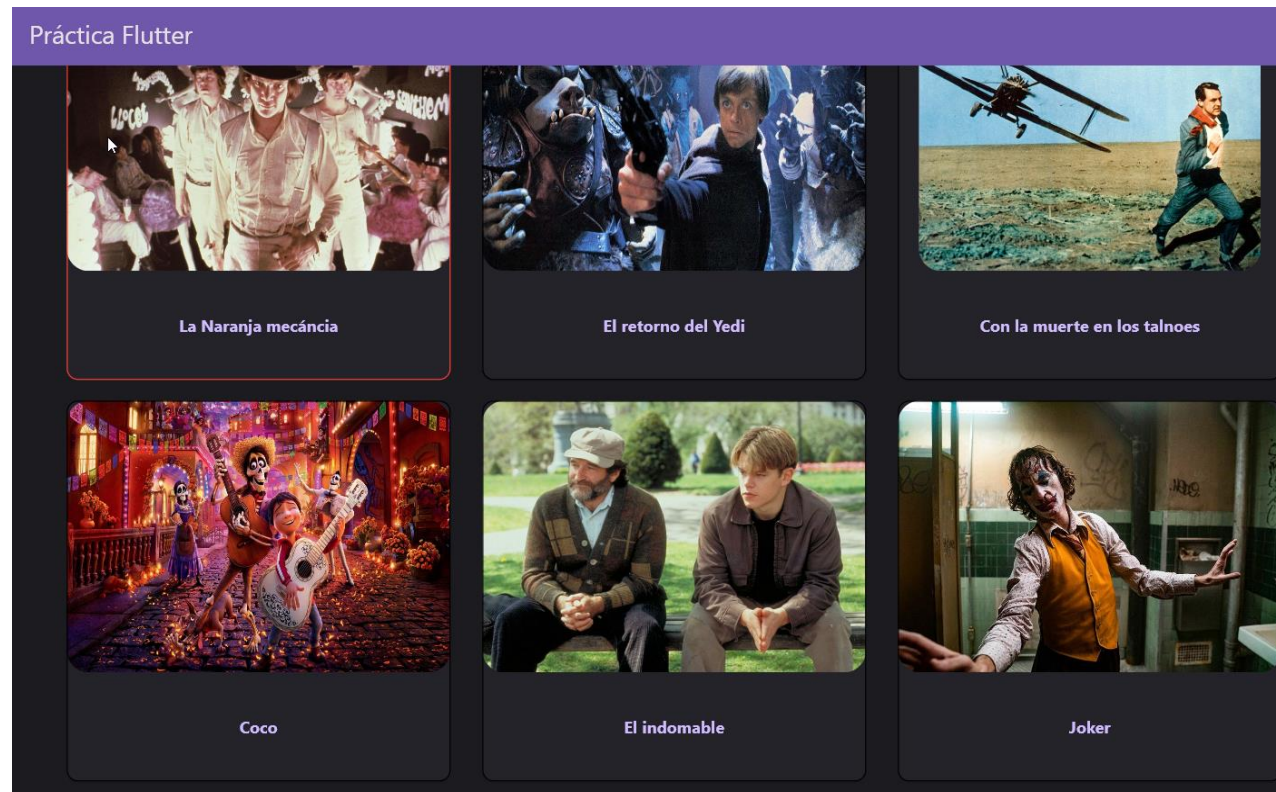
Widgets de múltiples hijos

 <p>Row Layout con una lista de widgets hijos en dirección horizontal.</p> <p>Documentación</p>	 <p>Column Layout con una lista de widgets hijos en dirección vertical.</p> <p>Documentación</p>	 <p>Stack Esta clase es útil si deseas superponer varios widgets hijos de una manera simple, por ejemplo, tener texto y una imagen, superpuestos con un degradado y un botón adjunto a la parte inferior.</p> <p>Documentación</p>
 <p>IndexedStack Una pila que muestra un solo hijo de una lista de hijos.</p>	 <p>GridView Una lista en parrilla consistente en un patrón repetido de celdas alineadas en un layout vertical y horizontal. El widget GridView implementa este componente.</p>	 <p>Flow Un widget que implementa el algoritmo de flow layout.</p>

Interfaces gráficas. Dart & Flutter.

► Flutter.

Práctica 2. Widgets, layouts y eventos.



Interfaces gráficas. Dart & Flutter.

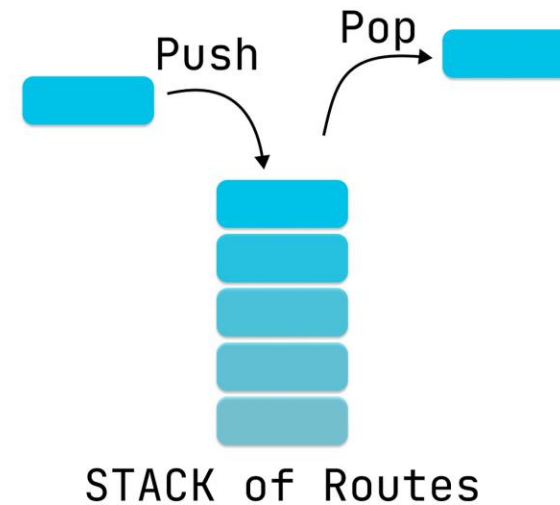
► Flutter. Navegación.

- Moverse por diferentes "pantallas".
- Pensado para dispositivos móviles.
- Gestión con estructura de tipo PILA. Push y pop de Widgets.
- Clase Navigator.
- Apilar la pantalla SecondRoute a la pila:

```
Navigator.push(  
  context,  
  MaterialPageRoute(builder: (context) => SecondRou  
);
```

► Desapilar.

```
Navigator.pop(context);
```



Interfaces gráficas. Dart & Flutter.

▶ Flutter. Navegación.

- ▶ Posible configurar rutas para reutilizar en cualquier punto de la aplicación.
- ▶ Publicación en objeto Navigator de MaterialApp.

▶ Definición:

```
MaterialApp(  
  title: 'Ejemplo rutas',  
  initialRoute: '/',  
  routes: {  
    '/': (context) => FirstScreen(), //ruta por defecto  
    '/second': (context) => SecondScreen(), //otra ruta  
  },  
)
```

▶ Uso:

```
Navigator.pushNamed(context, '/second');
```

Interfaces gráficas. Dart & Flutter.

▶ Flutter. Navegación.

▶ Paso de parámetros a la nueva pantalla.

- ▶ Definir en la página destino los parámetros necesarios en el constructor, que será un atributo de la clase (Widget).

```
DetailScreen({Key key, @required this.todo}) : super(key: key);
```

- ▶ Al llamar al navegador:

```
Navigator.push(  
  context,  
  MaterialPageRoute(  
    builder: (context) => DetailScreen(todo: todos[index]),  
  ),  
),
```

▶ Devolver valores de una pantalla.

- ▶ Al hacer pop, se dispone de un segundo parámetro opcional para devolver un objeto.
- ▶ Para esperar la vuelta del valor:

```
final result = await Navigator.push( context,  
  MaterialPageRoute(builder: (context) => SelectionScreen()),);
```

Interfaces gráficas. Dart & Flutter.

▶ Flutter. Recursos.

- ▶ Imágenes, ficheros JSON, texto, XML, sonidos, música.
- ▶ Necesarios incluir en la compilación nativa.
- ▶ Definición en fichero pubspec.yaml.
- ▶ Sección flutter:

```
flutter:
```

```
  assets:
```

- assets/my_icon.png
- imágenes/

- ▶ Añade el recurso my_icon.png y todo lo que esté en la carpeta imágenes.

▶ Acceso a las imágenes.

```
image: AssetImage('graphics/background.png'),
```